Programming Simplicity - Observations - Sampler

Paul Tarvydas

Programming Simplicity - Observations - Sampler

Paul Tarvydas

This book is for sale at
http://leanpub.com/programmingsimplicity-observations-sampler

This version was published on 2023-04-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2023 Paul Tarvydas

Table of Contents

Programming Is Not Coding

An Abbreviated History of Programming Languages.

The ground truth is *programming*, not *coding*.

The goal of programming is to control a machine.

A secondary goal of programming is to make it possible to change a program easily, or, to steal parts of a program and use them in other programs.

A tertiary goal of programming is to buff and polish specific notations, like textual programming languages.

Buffing and polishing code notations is a sub-goal of programming, and is, therefore, not as important as the main goal: controlling a machine. Coding - writing textual scripts in a programming language - is but a subset of the main goal.

What is the point of programming?

To control a machine.

To accurately break down an action in steps so small that even a machine can perform the steps.

Current electronic machines provide us with a set of steps called "opcodes".

Early Machines

Early machines were mechanical.

Early machines used gears and pulleys.

High Level Electronic Machines

Electronic machines - called *computers* - are machines where the gears and pulleys are replaced by electronics.

Early Programming of Computers

Tools

The earliest tools for programming computers were soldering Irons.

IDEs

The earliest IDEs - Integrated Development Environments - for programming computers were soldering stations which included a desk, a static mat, soldering iron, solder, resin and optionally, amenities like magnifying glasses, desk lamps, etc.

Process

Changing a program, consisted of unsoldering wires and connecting them elsewhere.

Drawbacks

The primitive programming tools and IDE were slow, but, orders of magnitude faster than changing the behaviour of machines by pulling gears and replacing them with other gears.

Other drawbacks included blisters on fingers. Recently soldered nodes would remain hot for a while and would create blisters if touched too soon.

Toggle Switches (Advanced Programming)

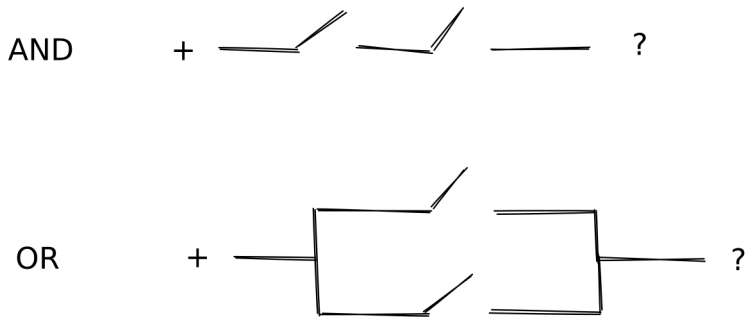
Toggle switches were used to replace soldering of wires.[1]

On-Off_Switch.jpg| 130

Revelation: toggle switches had 2 states “OFF” and “ON”.

Revelation: arranging toggle switches in sequential and parallel combinations produced physical equivalents of Boolean Logic (AND and OR functionality).

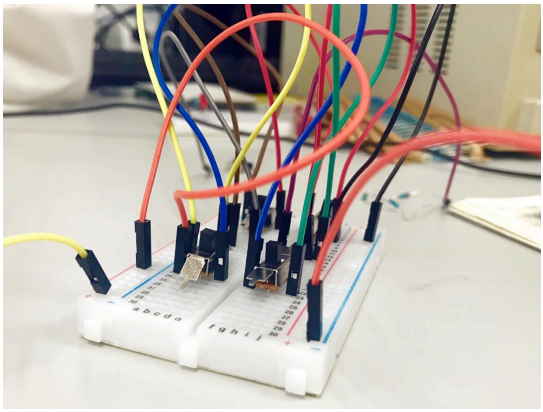
[1]Jason Zack at en.wikipedia, CC BY-SA 2.5 https://creativecommons.org/licenses/by-sa/2.5, via Wikimedia Commons



Boolean.svg

Breadboards (Advanced Programming)

Breadboards were used to allow moving and reconnecting wires without the need to solder.[1]



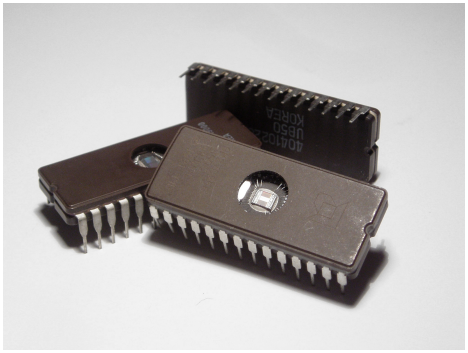
Breadboard_in_our_experiment.jpg

This was faster than programming by soldering, but, it was more expensive.

[1] https://commons.wikimedia.org/wiki/File:Breadboard_in_our_experiment.jpg

Miniaturization (Super Advanced Programming)

Switches were miniaturized down to the size of dust particles.[1]



Microchips.jpg

Drawback - Dust

The main drawback of miniaturization is that dust particles interfere with and block operation of switches when switches are that small.

The solution to this problem is to use clean rooms. Clean rooms keep dust out when making very small switches.[2]

[1] https://commons.wikimedia.org/wiki/File:Microchips.jpg
[2] https://commons.wikimedia.org/wiki/File:Clean_room.jpg

Clean_room.jpg

Drawback - $$$

The drawback of using clean rooms and miniaturization is that they are very expensive.

Only a few startups could afford to pay for clean rooms, e.g. Motorola, Fairchild, National Semiconductor, Texas Instruments, Mitel, etc.

HCI - Human Computer Interfaces - Keyboards and Displays

Problem #1: Ultra-small switches were too small for humans to interact with.

Problem #2: Ultra-small switches produced results that consisted of electrical impulses. The human body does not have precision sensors for detecting tiny electrical impulses.

Solution - Keyboards

Invent human-sized banks of switches - called QWERTY keyboards.[1]

Qwerty_Keyboard.JPG

[1] https://commons.wikimedia.org/wiki/File:Qwerty_Keyboard.JPG

Solution - Displays

Invent mapping from tiny electrical impulses to larger, human-sized displaying units.[2]

Vt100-adventure.jpg

Video screens. Display results on thin films of phosophorous painted onto the backside of glass tubes.

LED screens. Display results on grids of tiny light bulbs, called light emitting diodes.

Teletypes: Combination of printer and QWERTY keyboard as single units.[3]

[2] https://commons.wikimedia.org/wiki/File:Vt100-adventure.jpg
[3] https://upload.wikimedia.org/wikipedia/commons/9/9e/Teletype.jpg

Teletype

Assembler (Ultra Advanced Programming)

Revelation: strings of bytes (e.g. ASCII) could be "parsed" to convert mnemonic words into binary opcodes.

For example, the string "RET" could be pattern-matched and replaced by the hex byte 0xC9. This process mapped the 3-byte sequence

```
0x52
0x45
0x54
```

to the 1-byte sequence:

```
0xC9
```

Revelation: "RET" and "RETX" were parsed the same if a shortest match strategy was used. The fix was to use a longest match strategy and make certain characters special. Whitespace was used to mark the end of any word. We call such special characters *delimiters*. This led to the concept that whitespace is not allowed in mnemonic words. Note, that whitespace is used in prose, but is disallowed in programming languages. A phrase in prose consists of several words separated by whitespace, whereas a phrase in a programming language is composed of syntactic constructs, like `if then else`, using words that contain no whitespace.

So, the 3-byte sequence, above, became a 4-byte sequence[1]

[1] FYI - 0x20 is ASCII for "Space". Practically, "Tab" - 0x09 - was used more often for caveman-level pretty-printing purposes, but, still mapped to the same 1-byte sequence:

```
0x52
0x45
0x54
0x20
```

```
0xC9
```

It was discovered that *operands* of opcodes could, also, be parsed in similar ways. Comma was treated as a sub-class of delimiters:

```
MOV R0,(R7)+
```

Note that assembler used 2 very important, but subtle, tricks:

1. separation of operators and operands
2. line-oriented

Both of these tricks make it easier to write programs that write programs. Tricks such as these led to invention of apps call 'compilers'.

Separation of operators and operands

This is what is called "orthogonal code" and is espoused in Cordy's thesis[2].

The "orthogonal code" technique is familiar to most programmers in the form of the *gcc* compiler. An early form of orthogonal coding was Fraser/Davidson's Peephole technology[3].

[2] https://books.google.ca/books/about/An_Orthogonal_Model_for_Code_Generation.html?id=X0OaMQEACAAJ&redir_esc=y

[3] https://www.researchgate.net/publication/220404697_The_Design_and_Application_of_a_Retargetable_Peephole_Optimizer

GCC uses orthogonal code techniques first, then optimizes the generated code using techniques that are described in the Dragon Book[4] to generate assembler code that rivals assembler code written manually by human assembler programmers. *GCC* is so good, that just about no-one bothers to write assembler any more. At first, the concept of using compilers was thought to be too inefficient to be practical, and, was laughed at. *GCC* changed such ideas.

Unfortunately, the orthogonal programming technique has been ignored in the development of most programming languages.

Programming languages *could* be categorized as:

- operation phrases
- operands.

But, programming language designers fell under the spell of "one language to rule them all" and conflated control flow (operation phrases) with operands.

OOP (Object Oriented Programming) is a description of *operands* and could be used effectively in an orthogonal programming language.

Line-Oriented Source Code

Another "trick" that helps in writing programs that write programs is the use of normalization.

One form of normalization is the idea of using lines of text to delimit programming phrases.

For example:

[4] https://en.wikipedia.org/wiki/Principles_of_Compiler_Design (revised https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools)

```
1 MOV R0,(R7)+
2 MOV R1,(R7)+
```

Is pattern-matched as 2 separate lines. This "trick" makes pattern-matching easier.

"Easier" is a subjective term and a psychological trick. If a technique is "easy enough", it will be used, and, become an idiom, and, will relieve the minds of smart people to think about more lofty problems.

Line-oriented assembly code was so "easy" to automate that it allowed programmers to invent higher-level programming languages, and, compilers, and, interpreters.

"Compilers" are simply "apps" directed at programmers. Compilers stand on the shoulders of other apps-for-programmers, like assemblers.

UNIX Shell Pipelines - Line-Oriented

A shining example of line-oriented source code being used to create bigger and better programs is the UNIX shell and *pipelines*.

The UNIX shell hard-wired the notion of *lines of text* into its structure.

A slate of tools emerged from this simple, subtle assumption. Tools like *grep*, *REGEX*, *sed*, *awk*, etc.

Programming languages have evolved into using a *structured* syntax, where programming phrases (like `if then else`) cross line boundaries and are recursive.

The UNIX assumption of *everything is a line* is not good enough to handle most modern languages. Lines are not recursive, where the syntax of textual programming languages is recursive.

Aside - Tree-Oriented Source Code

A different approach to parsing is found in the language Lisp.

In Lisp, all source code is structured in the form of trees. Lisp uses the concept of *list*s. Lisp source code is written in the form of *parse tree*s made from *list*s. A unit of source - a Lisp function call - is a list (instead of a line) in a very rigid structure. First, comes the name of the function, and the rest of the list (if any) contains the arguments to the function. This format is often called Reverse Polish Notation[5].

Like line-oriented assembler, this normalized structure makes it "easy" to write code that writes code. Lisp notation led to the notion of *macros*[6]. Lisp macros are essentially code generators.

Unlike line-oriented assembler, the lisp list structure is recursive and can be used to write code that contains other code, i.e. functions that call functions that call functions, and so on.

[5] In fact, there is nothing "reverse" about this notation, it is a prefix notation.

[6] Lisp macros are much more powerful that what is thought of as macros in other languages, like C. Lisp macros are like plug-ins. Lisp macros are functions that modify the operation of the Lisp compiler and interpreter. Lisp macros are functions that run at compile-time.

High Level Languages (Super Ultra Advanced Programming)

The ease of writing programs that write programs led to the invention of bigger and better programming languages and compilers.

All programming languages devolve to assembler in the end, e.g. Haskell is converted into assembler by the Haskell compiler, Rust is converted into assembler by the Rust compiler, C++ is converted to assembler by the C++ compiler, and so on.

HLL - high level language - compilers are simply skins that make it more convenient for programmers to create assembler code. The machine - the computer - doesn't care how the assembler is created, it simply executes instructions given to it.

Programmers, though, want helper apps to help them catch bugs more easily.

Programming languages as text were invented to alleviate programming problems. The problems were perceive because of 1950's based biases and realities. At the time, CPUs were expensive and rare and memory was expensive. The majority of solutions to these problems included

- time-sharing (using 1 CPU to run many apps)
- GC - garbage collection - recycling memory
- mutation - recycling memory
- the call-stack was invented based on the notion that computers were simply bigger and better calculators ; this attitude implied that mathematics notation is "good enough" for programming computers

- prioritizing optimization of CPU effort over human-developer effort. In 2022++, CPUs are abundant and cheap, and memory is abundant and cheap, yet, programmers continue to use techniques developed to solve perceived problems of the 1950s. In 1950, we had Central Processing Units, in 2022++ we have Distributed electronic machines.
- Syntax checking - parsing - was invented to ensure that strings of non-whitespace characters were arranged in a sensible order that could be automatically transpiled into assembler (using compilers)
- Type checking was originally invented to optimize app speed on CPUs. CPUs could handle certain kinds of numbers faster than other kinds of numbers. For example, numbers that were integers in the range of -128..127 could fit in a byte, whereas floating point numbers needed to be represented using more bytes. Byte-oriented integers were often handled directly by the CPU hardware, whereas floating point numbers could only be manipulated using libraries of code. Calling library routines was much, much slower than using built-in CPU integer operations.

Once the menial tasks of low-level optimization were lifted from programmers' shoulders, it became possible to invent user-defined types and deeper type-checking of user-defined types.

Many "new" programming language designs add more developer debugging help in the form of incrementally better/deeper type-checking.

Various helper-apps for programmers were developed and are still being developed, such as

- syntax checking
- type checking
- IDEs

ASCII

The original programming languages had only numbers because those could be conveniently expressed in mathematical notation. For example, there was no concept of character strings. Strings were treated simply as arrays of byte-sized numbers.

ASCII (and EBCDIC) were invented to encode characters using small (byte) integer codes.

Programming languages were defined using this restricted encoding. For example, strings use the same character " to mark string beginnings and string ends. This choice is contrary to any sensible encoding and contradicts the concept of quotes used in written natural languages. This encoding makes it harder to parse strings and leads to epicycles involving multi-line strings and strings that contain quotes.

Since then, Unicode was invented. Unicode greatly expanded the range of possible integer codes that could be used to represent characters, yet, we continue to design languages with work-causing, bloat-causing assumptions from the 1950s. Programming languages could use different characters to demarcate beginnings and ends of strings, but, don't.

The drive to satisfy 1950s biases, such as ASCII, cut off development of alternate syntaxes, like diagrams and glyphs. At one point the language APL was popular but required special keyboards and displays and required that all text be written in upper case.

Early languages, like FORTRAN and LISP cut out the use of lower case letters. In part, this helped to "improve efficiency" of parsing (ASCII lower-case "a" differs from upper-case "A" by one bit, allowing succinct assembler tricks to be used). In part, this was an admission that programming languages were not the same as natural languages. It seemed reasonable to "program a computer" by using commands written only in upper case. We continue

to see repercussions of this decision even today. File names are schizophrenic - Linux treats file name case as significant, whereas MacOS ignores case in file names. This non-standardized disparity leads to surprises even in 2022++.

Code Bloat

Code in 2022++ is orders-of-magnitude larger than code in early computers.

Functional programming is fundamentally small and beautiful. Sector Lisp is a shining example of just how small programming languages can be made to be. Sector Lisp is less than 512 bytes[sic] in size. Sector Lisp's Garbage Collector is 40 bytes[sic] long.

At first, one assumes that Sector Lisp's smallness is only due to assembler tricks and avoidance of the inherent complexity of programming problems.

This is but a hand-waving argument that is used to ignore much deeper issues.

Assembler Tricks

If Sector Lisp's smallness was due only to assembler tricks, we would expect to see the *gcc* effect. Someone would have built compilers for most existing languages that produced orders-of-magnitude less assembler code than existing compilers are capable of producing.

This hasn't happened.

Programming languages produce bloated and large assembler code for reasons that go beyond simple application of assembler tricks.

Complexity

No concept is complicated.

Complication is only in the eyes of the beholders, i.e. complication is due to the use of unsuitable notations for expressing and solving specific problems.

When a concept appears to be complicated, it is because an inappropriate notation is being used to describe the concept. Apparent complication arises when too many concepts are force-fitted into a single notation.

The simple concept of *functional programming* has, at its core, a few fundamental principles:

- immutability
- stack-oriented evaluation
- function calls that are instantaneous
- lambdas as stack-based wrappers

Immutability - Referential Transparency

Immutability is vital to "referential transparency" in functional programming.

Referential transparency means being able to replace a unit of code, e.g. a function, by another unit of code. In hardware, this is called "pin compatible".

Microsoft Word's *Find and Replace* function is a form of referential transparency. It replaces one string of characters with another string of characters. Some rules must be followed to allow this feature to work without creating surprises. Referential Transparency applied to programming languages, also, requires a set of rules. The rules are very similar to the rules for using *Find and Replace*.

There is more than one way to achieve Referential Transparency.

The method used in hardware is to elide the insides of components, using black epoxy and plastic, and, to characterize components "from the outside" so that components can be compared and suitable substitutions can be made.

In functional programming, Referential Transparency is achieved by decreeing that mutation cannot happen *anywhere* in the codebase.

Computers, though, inherently support mutation. RAM (Random Access Memory) implies mutation.

So, there's a disconnect between functional programming and the reality of computer hardware.

Stack-Oriented Evaluation

Function calls with parameters always cause the parameters to be evaluated before the function is called.

The results of parameter evaluation are placed in temporary locations on the call-stack.

Instantaneous Function Calls

Pure functional notation works only if function call latency can be ignored.

In reality, CPUs implement function calls in hardware using operations and a global call-stack.

The operations CALL and RET take finite amounts of time when implemented in hardware.

This means that functional notation, while useful for some kinds of use-cases, is not able to express the full gamut of computer operations.

Programmers have force-fitted finite-time concepts into functional notation. This has resulted in unexpected bugs (see Mars Pathfinder

disaster[1]) and general agreement that certain kinds of programming constructs are "complicated". For example, "concurrency" is considered to be a difficult problem, causing issues such as "thread safety", "preemption", etc.

Furthermore, the assumption of instantaneous function calls - on paper - conflicts with the reality that hardware CALL/RET instructions are forms of ad-hoc *blocking*. On paper, a function can call[2] another function instantaneously, whereas in hardware, a function call suspends the caller until the callee returns a value. The callee needs a certain amount of time to complete, and, we cannot calculate that time without knowing exactly what other functions are called and what functions are called by those functions, etc.

Code Bloat Due to a Plethora of Types

Sector Lisp gets some of its leanness due to the fact that - like McCarthy's original Lisp - Sector Lisp has only 2 types

1. Atom
2. List

A machine needs lots of detail, while humans want to elide detail. Sector Lisp is a demonstration of how large the notational savings can be if detail is elided. The 2-type system provided by Sector Lisp is good enough for controlling a machine.

It seems that we need languages at both extremes - many types vs. very few types - and, we need a way to map from one extreme to the other, or, in only one-way, to map from a human-amenable language to a machine-executable language.

[1] https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft

[2] Note that the word *call* is not used for paper versions of the notation. The term *call* was invented with respect to computer programming.

In the past, it was considered difficult to build languages, so programming language designs attempted to straddle both extremes.

Today, in 2022++, technology such as Ohm-JS ("new and improved" PEG), it is possible to invent languages quickly (e.g in an afternoon). It is possible to tune one language for human use and another completely separate language for machine use.

We see the beginnings of this kind of separation in the Assembler vs. HLL split, but, due to the difficulty of inventing languages, the idea of inventing multiple languages has not been exploited. In fact, biases against the use of multiple languages have developed, probably due to the inadequacy of the one-language-to-rule-them-all approach. Existing languages are full of minute details and doo-dads which makes such languages hard to learn and makes the idea of using more than one language seem unfriendly.

The idea of inventing languages has, historically, been relegated to the field of compiler development. The concept of light-weight macros and text-manipulation tools, like those of UNIX, has been overshadowed by

- the fear of learning complicated new language syntaxes, which leads to the unsubstantiated fear of learning *any* new syntaxes
- the desire to flatten all layers of design into walls of details, and, the desire to "make the trains run on time", and, the desire to construct software in a clockwork manner. All of which makes software development accessible only to a select few.

The "answer" to these problems is to turn the problems on their side and to look at them from a new perspective and to re-examine the problems from first principles.

The problem is that software components are too complicated. The solution begins with understanding why the complication arises,

not to work around the complication. 7-line[3] BASIC programs were easy to understand and to develop. Issues such as "global variables" don't even arise when program sources are that small. The answer is not how to solve the problem of creating 8-line programs, but to how to bolt 7-line programs together, recursively in layers, so that no layer has more than 7 lines of code in it. Even a very large system - usually thought to be "complicated" - consists of 7 lines of code at the top level. We don't need to throw details away, we simply need to bury details. Once the top layer is understood, the reader might wish to know more details. More detail can be revealed by drilling down into sub-components and examing them (each of which contains no more than 7 lines of code). The fact that very complicated A.I. machines can be controlled 5 line Python programs indicates that the goal of small programs can be achieved.

In this scenario, each layer must be totally indepedent from lower layers. Each layer must be *understandable* in a stand-alone manner.

From a first principles viewpoint, we see problems like

1. dependency, especially hidden dependency due to synchrony, causes a trickle-down effect that makes layers not understandable on their own, and, makes them appear to be complicated
2. inheritance instead of parental authority - child code can change the behaviour of parent code, making the parent code not-understandable on its own, even if it is only 7 lines long.

GC (Garbage Collection) without the presence of mutation is small and simple, as characterized by Sector Lisp's 40-byte[sic] garbage collector.

Code Bloat Due to Type Checking - The State Explosion Problem

Imagine a simple piece of code to add two numbers.

[3] https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

```
number add2numbers (number a, number f) {
  ...
}

void main_not_bloated () {
  number a;
  number f;
  number result;
  result = add2numbers (a, f);
}
```

Humans understand the concepts of numbers. The numbers might be integers or they might be doubles or they might be bignums.

Machines, though, don't understand these differing concepts and need to be programmed to handle the various cases in various ways and efficiencies.

In some languages, data carries tags that describe the type of the data.

In other languages, the tags are pre-compiled out and removed before execution. The process of removing type tags must be done with enough care to allow the machines to know how to manipulate the data.

At the machine level, we might wish to transform the above code to handle the various cases of numbers as integers vs. numbers as floats.

```
float add2if (int a, float f) {
  ...
}

float add2ii (int a, int f) {
  ...
}

float add2ff (float a, float f) {
  ...
}

float add2fi (float a, int f) {
  ...
}


void main_bloated () {
  int a;
  float f;
  float result;
  result = add2if (a, f);
}
```

The machine-level code grows exponentially larger as new kinds of numbers, with different details, are included.

This effect is called the State Explosion Problem. The effect has traditionally been associated with the development of State Machines, but, as seen above, the effect creeps into other aspects of programming.

The problem becomes even worse when languages allow programmers to define new types. This is usually called “user-defined types” and “classes”.

Various ad-hoc attempts have been made to reduce this problem in

one-size-fits-all languages:

- the use of functions to wrap and parameterize similar runs of code
- inlining
- generic types
- etc.

Many of these approaches have runtime implications. For example, the use of functions to abstract code causes the runtime code to run slower (due, at least, to the use of CALL/RETURN opcodes).

A great amount of compile-time effort (burning CPU cycles) is required to compile away all runtime implications of these solutions. Again, the CPU effort is expended only to aid developers by checking for errors[4]. The machine doesn't care about type systems and other complicated structures.

What is the cost-benefit for such expended effort?

Compiling away all remnants of type checking is only fruitful once the design is known to work and is stable. This is, traditionally, called Production Engineering.

During Design, though, it is unnecessary to compile away all type checking, and, the extra time and effort in doing so causes interruptions in the design process. Design is done when the designer is "in the zone" (or "in flow"). Interrupting the flow can severly diminish the abilities of the designer, and, hurt the design.

Software development appears to be in the "cottage industry" phase, where one programmer[5] does all of the above work - from Design to Production Engineering.

[4] But, only errors that are expected by the programmer(s) who wrote the error-checking app.

[5] Or, one team of programmers.

Details kill

Details kill understandability.

It is the Architect's responsibility to make a design clear and understandable to other readers.

Types Needed During Design

To interfere as little as possible in the Design process, we need approximately two types:

- things
- recursive lists of things

Currently, developers use compiler-appeasement languages that insist on extreme amounts of detail to help compiler apps to check for errors. Compiler-appeasement languages, though, impede the progress of product development, often in subtle ways, e.g. by breaking development "flow" by insisting on extreme amounts of detail[6].

Error checkers like *lint* are more humane. *lint* does not change the development programming language. *Lint* does its best to look for potential errors. *Lint* is like a barnacle attached to the side of a program.

Lisp introduced the concept of gradual typing, but, most modern languages insist on making programmers specify types fully.

[6]As a simple example, any software tool that asks "where do you want me to save this?" up front, is asking the developer to stop developing to deal with information to satisfy the tool's problem. Likewise, something like "which kind of mind-map do you want to build?" is a development-breaker. Another deal-breaker is "declaration before use" which is a concept developed to appease 1950s biases. Computer hardware in 2022++ is fast enough to perform multi-pass compilation, but, programming language designers continue to insist on 1950s-style declaration-before-use.

Code Bloat Due to Word Size

Building a compiler that was portable across two memory architectures - 8-bit and 16-bit - I measured code size for the same source program[7].

The 8-bit version was 40% the size of the 16-bit version. I.E. the 16-bit version of the *same program* was more than twice the size of the 8-bit version. That's greater than 100% inflation.

The reason for the size increase is that larger word sizes are not needed to encode all instructions nor operands. Larger word sizes contain waste bits for instructions and operands that could be encoded more succinctly.

The trade-off involves memory fetching. Smaller word sizes need multiple fetch cycles to read large instructions and operands from memory. In an 8-bit architecture, this means that the most efficient opcodes and operands are in the 0..255 range. All other opcodes and operands take more time to read from memory.

Word size also affects internal circuitry - it takes less space in an IC to build circuits for handling 8-bit quantities than building the circuitry to handle the same quanities in 16-bit form.

Word size affects VLSI pinout. CPU chips typically have separate pins for each address line and for each data line. Connections inside a chip are very small, too small for handling by humans and early industrial machines. Pins are used to bring electrical connections from inside a chip to the outside world. Pins are little tabs of metal that are electrically connected to the points inside ICs. In the 1970s, ICs were mostly assembled by humans wielding soldering irons and various other kinds of wiring technologies (like wire-wrapping). With improvements in miniaturization, it was possible to shrink the size of pins while retaining the ability to assemble electric circuits using automated equipment (e.g. wave soldering) and humans wielding smaller-tipped soldering irons and magnifying glasses.

[7] I've lost the actual supporting data over time. I'm writing this from memory.

For a typical 8-bit architecture, like the 8080, Z80, MC6809, etc., the data bus consists of 8 pins and the address bus consists of 16 pins. 16-bit addresses are used in these specific architectures, but, this doesn't mean that all 8-bit machines use 16-bit addresses. This is a total of 8+16 pins, ie. 24 pins to access RAM. In fact, one or more extra pins are needed to manipulate the RAM (e.g. the Write line, a clock signal, power, etc.)

For 16-bit architectures, the pin count goes up. For example, the data bus needs 16 pins and the address bus is expanded to 24 or 32 pins (40 and 48 external pins, respectively.)

Other Syntaxes

Random comments about miscellaneous syntaxes / technologies …

Programming doesn't have to mean text-only. The goal is to control (use) a computer. The goal is to generate instructions for the computer to follow. Somehow.

Spreadsheets

- Maybe the most successful non-programmer syntax is the spreadsheet.
- flat grid
- list of builtin functions
- simple set of rules

    - grid + palette of possible functions to use
    - no decisions to be made about using nor extending the technology
    - simple, uncomplicated, "simplicty is the lack of nuance"
    - grade-school math, simple equations
    - simple syntax for accessing cells - row, column - no need to remember names
    - future: instead of naming cells, show different views
    - e.g. dependencies on other cells

        * Racket-like lines

    - e.g. synonyms, e.g. $A1$B2 is, also, called "Total", also, called "Sum of Column ($C3:$C99)"
    - use hotkey to toggle between $A1$B2 syntax and synonyms

  - there is no “one language to rule them all”, different users may want different names, allow switching via synonym toggling
  - synonyms can apply to code, too

    * “y = *slope* times *x* + *y offset*” is, also, “y = mx + b”
    * lambda functions are barking up this tree

Relational Programming

- PROLOG

  - granddaddy of declarative programming
  - uses backtracking to perform exhaustive search
  - sneered at in 1950s, due to biases that favoured speed optimization
  - backtracking driven by desire to conserve memory

- miniKanren

  - does not use backtracking, but, also, implements exhaustive search
  - coagulates all possible outcomes and feeds set of outcomes forward
  - prunes outcomes that cannot happen from the set of possibilities
  - final result = set of all possible ways to satisfy a goal
  - not concerned with expending memory, hence, doesn’t need to use backtracking

    * backtracking is difficult to implement
    * backtracking is difficult to understand

- what not how - declarative programming

  - Relational Programming splits problems into two parts

1. What
2. How

    * What is specified by the human programmer
    * How is implemented in an engine, freeing the human to think about the larger problem instead of dealing with details

Everything is a String

- Lisp treats everything as Lists, there are some languages that treat everything as strings
- SNOBOL

    - the granddaddy of string-based languages and string matching

- Icon

    - grew out of SNOBOL

- TCL/tk

    - TCL is a scripting language based on treating all data as strings of characters
    - tk is a library for creating graphical views on data that grew out of the ability to suppress details like how the data is actually structured (e.g. integers vs. strings, etc.)

        * the details are all still there, but are elided at the top level
        * the details are buried in the TCL engine

- Parsing

  - string-matching technology usually associated with compiler building
  - compilers are simply big transpiler apps
    * convert a string of characters into another string string of characters
    * high-level language text converted into assembler text (and/or to bits)

Scripting

- like “everything is a string”, but, includes files and redirection
- scripting is still *programming*, it uses scripts to control behaviour of computers
- example: /bin/sh
- example: emacs lisp
- example: cmd.exe
- often a “dynamic language” which punts typechecking to runtime

Hypercard

- https://en.wikipedia.org/wiki/HyperCard
- https://hcsimulator.com

Dataless Programming Languages

- dataless languages - don’t require programmers to define data, just declare the existence of data
  - e.g. “handles” in operating systems

  - e.g. S/SL (Syntax/Semantic Language, Holt, et al) https://research.cs.queensu.ca/home/cordy/pub/downloads/ssl/
  - powerful enough for implementing compilers (e.g. PT Pascal, Concurrent Euclid, etc.)

Concept: Orthogonal Programming Languages

- based on dataless languages, Cordy's Orthogonal Code Generator, and GCC's RTL
- OOP is 1/2 of the story
- control flow is the other 1/2
- data representation
  - Objects are often expressed as Classes with Methods
    * similar to mathematics concept of conditional functions
  - control flow is often expressed as *syntax*
    * syntax can, now, be quickly built using Ohm-JS (improved PEG)

WASM

- Lisp-like syntax
- uniform syntax
- intended to replace JavaScript in websites
- intended to be "more efficient" than JavaScript
- attempt to add type information on operands and operators while using recursive assembler-like syntax

VPLs

- StateCharts - StateCharts exhibit a lot of goodness - encapsulation - parental authority - diagrams instead of text for control-flow aspects - text can be used where it makes sense - reading of Harel's paper: https://guitarvydas.github.io/2020/12/09/StateCharts.html
- Scratch
  - ostensibly for children
  - hampered by synchronous mindset
  - https://scratch.mit.edu
- Full Metal Jacket
  - attempt at a pure functional visual syntax
  - https://www.fmjlang.co.uk/fmj/FMJ.html
- Drakon
  - rocket science
  - better than flowcharts (less ad-hoc)
  - transpiled to multiple languages, like Erlang, Python, Lua, etc.
  - https://drakon-editor.sourceforge.net
  - https://drakonhub.com/en/
- FBP
  - components that use streams of data
  - all components are asynchronous by default
  - https://en.wikipedia.org/wiki/Flow-based_programming
  - noFlo
    * grafts many concepts of FBP onto JavaScript

    * components are not asynchronous by default
    * https://noflojs.org

- node-red
  - one input port
    * need to look "inside" to see what inputs are supported
    * single input port assumption sprayed throughout node-red source code
      · not easily changeable
  - components are not asynchronous by default
    * function-calling (call-stack) remains at the heart of the notation/implementation
  - https://nodered.org
- LabVIEW
  - big, flat diagrams
  - doesn't encourage layered abstraction
  - encumbered by synchronous mindset
  - https://www.ni.com/en-ca/shop/labview.html
- ProGraph
  - diagrams of OO Objects
  - fundamentally synchronous, does not encourage independent layers
  - defunct
  - https://en.wikipedia.org/wiki/Prograph
- UML
  - "modeling" instead of "compiling"

        * very few constructs can be compiled to code, too abstract/ad-hoc
        * notable exception: StateCharts can be compiled to code

    - https://en.wikipedia.org/wiki/Unified_Modeling_Language

- hybrid

    - need detail to program a computer
    - text PLs provide detail for calculator-like operations
    - what does VPL provide beyond what text already provides?

- whither Diagram Compilers? Tech Diagrams?

Low-Code

- deja vu all over again, used to be called RAD
- RAD - Rapid Application Development,
- interpolate, cannot extrapolate

HTML

- declarative description of websites
- needs JS as a crutch for imperative operations that aren't handled in declarative syntax
- the most common portable syntax

    - every browser on every computer, on every phone and every tablet supports HTML
    - assembler is more common, but not portable

XML

- generalization of HTML
- attempt to parameterize HTML, resulting in complication

Declarative Programming

- Barliman
  - https://www.youtube.com/watch?v=er_lLvkklsk
- Relational Programming
  - swipl
    * https://www.swi-prolog.org
  - miniKanren
    * http://minikanren.org

TXL

- functional language for exploring syntaxes for existing languages
- http://txl.ca

AI generates Code

- generated / trained code based on code in github

- synchronous - since most code in github is fundamentally synchronous, the training data is fundamentally synchronous, hence, the generated code is fundamentally synchronou
- ChatGPT
  - https://openai.com/blog/chatgpt/
- Copilot
  - https://github.com/features/copilot

ROS, Behavior Trees

- Robot Operating System
  - complicated libary of functions that support asynchronous control using synchronous functions
  - https://www.ros.org
- Behavior Trees
  - epicycle to add *time* back into functional/synchronous code
  - "tick" is the same as "synchronous clock" in hardware design
  - builds asynchronous state machines on top of synchronous code, uses "tick" to step the state machines
  - https://navigation.ros.org/tutorials/docs/using_-groot.html

1950s Text-based Programming Languages

- The difference between 1950s text based programming and more modern diagrams is that in the 1950s text needed to be arranged in non-overlapping grids of characters, whereas in 2022++ *text* is just another figure on the diagram.
  - Figures can overlap.
  - Figures can be moved around.
  - Figures do not imply sequential sequencing.

Smalltalk

- Smalltalk, like Lisp, doesn't have much of a syntax
- control flow implemented using "tricks" such as passing closures ("blocks") to functions
  - e.g. `if` is essentially a function that consists of 3 closures
    1. test expression (returns an Object of type Boolean)
    2. `then` block ("ifTrue:" message to Boolean Object)
    3. `else` block ("ifFalse:" message to Boolean Object)
- Methods are synchronous function calls
  - "message passing" is synchronous, not asynchronous

Compilers

How do you build a compiler?

A compiler is just an app.

A compiler is an app aimed at developers.

A compiler is a “complicated” app and involves a lot of programming techniques.

The point of a compiler is to convert text, written in some higher-level language, to assembler text and, ultimately, to binary opcodes that sequence the operation of computers.

A programming language - compiled or interpreted - is just a caveman version of a programming IDE. Ultimately, programmers want to inject binary opcodes into electronic machines. Programming languages let programmers create opcodes while checking for a plethora of common mistakes.

The “worst” problems in building compilers are:

- how to generate fast/small binary code
- how to generate code quickly, so that developers have short turn-around times
    - early progams were written on punch-cards
    - punch-cards would be dropped into the hopper of a card reader, then the data on the cards was fed, electronically, to the computer, and, the computer would run the program and output results on paper
    - turn-around times for this process were something like 10’s of minutes

    * programmers did not want to stand in line at the card reader, so they chose to carefully preen their code before going to the card reader
    * turn-around times were around one program run per day
    * computer time was expensive, and, access to computers was “billed”

        · programmers avoided blowing out their accounts and billings by double-checking their code for simple mistakes

– running a compiler was a billable event, so compilers that were fast and small were favoured

Compilers vs. Interpreters

Interpreter

An interpreter is an app that inputs a program written in some programming language.

An interpreter steps through source code and immediately acts on every instruction written into the code.

If the interpreter encounters a *loop*, it seeks to the next instruction and blindly executes it - even it has already seen the instruction before.

A hand-held calculator is an example of an interpreter. You press numbers on the keypad then hit and operator key, like +. The calculator immediately interprets the operation and displays the result of the operation on an electronic display or on paper. Early versions of calculators were completely implemented using ad-hoc electronic circuitry. Even earlier, calculator were implemented entirely with mechanical wheels and pulleys. Even earlier than that, calculators were implemented in wood and paint (e.g. slide rules). Even earlier than that, calculators were implemented using brute force banks of humans who performed calculations on paper and clay tablets.

A CPU is an interpreter. It steps through an array of binary instructions (opcodes) and immediately executes each instruction as it is encountered.

The body of a CPU consists of a *case statement* that picks apart every instruction, in sequence, and a block of actions for each instruction.

The *case statement* and the *actions* are implemented as hard-wired bits of electronics.

Compiler

A Compiler is a specialized app for developers.

A compiler attempts to separate code into two big chunks

1. code which needs to burn CPU cycles at runtime
2. code which can be pre-calculated and does not need to use CPU cycles at runtime (this kind of code *does* need to burn CPU cycles at compile time, but, it is assumed that compile-time happens much less frequently than runtime, and therefore, compile-time costs can be amortized to reduce the final cost of the product from the end-users' point of view).

For example, imagine a piece of code such as

```
var a = 1;
var b = 2;
loop exactly 10 times
    var c = a + b;
end loop
```

An interpreter will blindly recalculate the value of c the same way 10 times.

A compiler will notice that `c` is redundantly calculated in the loop 10 times. A compiler will move the calculation out of the loop and try to make the code something like:

```
var a = 1;
var b = 2;
var c = a + b;
```

In fact, a compiler might try to do more, like noticing that `a` and `b` are constants and, therefore, `c` is a constant. But, we won't go there in this discussion.

A compiler restructures the code and then outputs binary opcodes that are "optimized" to reduce the number of CPU cycles burned at runtime, or, to decrease the size of the binary code (or, many other kinds of things).

A compiler does not directly execute the code, it simply optimizes the array of binary opcodes which will be interpreted at runtime.

How Do You Write Code That Figures Out What Can Be Pre-Compiled?

How do you write this kind of app? In other words, how do you write a compiler?

This is just like any programming problem. There are many solutions to this problem.

One very common approach is to redefine the programming language. The redefinition makes it easier for the compiler app to spot opportunities for optimization.

In essence, this approach puts the onus on the programmer to help out the compiler app.

I call this approach "compiler appeasement". It trades off the use of human brain cells for stroking the compiler app in "just the right way" in lieue of using those brain cells for thikning about more lofty problems.

A glaring example of compiler appeasement is the concept of *declaration before use*. The human programmer is expected to declare every variable that will be used in the program in order to allow the compiler to double-check for typos. Worse, *declaration before use* requires human programmers to restructure their code to ensure that declarations of variables all appear in the code before any actual instructions that use the declared variables appear in the code. This restriction was invented in the 1950s when it mattered that compilers be lean and fast. *Declaration before use* allows compiler-writers to write compiler apps that need to make only one pass through the source code. In 2022++, it is not necessary to make compilers this lean and this fast, but, programming language designers continue to use *declaration before use* rules in their language designs. Multi-pass conversion programs were well-known in the 1950s - assemblers were built to scan source code multiple times - so, it is obvious that the decision to use one-pass compilation was a conscious decision and was not made due to lack of existing techniques.

Another glaring example of compiler appeasement is the use of types and classes. During the design phase of a project, programmers don't necessarily want to specify all types in detail[1], but are required to do so in order to use modern programming languages. Lisp showed that it was possible to use gradual typing. ML showed that it was possible to infer types automatically (using a machine). These ideas are only slowly leaking out into the popular programming language world.

Compilers Are Interpreters

Compilers, are themselves, interpreters.

Compilers interpret their own app code, and, breathe-in programs,

[1]Some programmers like to lean on type checkers to help them iteratively produce internally-consistent designs. This is a *choice* and should not be a *requirement*.

and, spit-out optimized binaries. Compilers interpret their own code immediately with the goal of producing optimized binary code for other programs.

In analogy, compilers are like hand-held calculators that need more expensive hardware than that needed for calculators.

Tokens, Tokenization

One of the problems of writing compiler apps is the fact that certain character strings appear in the source code over and over again.

Scanning the same string repeatedly burns CPU cycles and introduces the chance of making compiler-writing mistakes in the string-scanning code.

To reduce these burdens, compilers do as much of the string-scanning up-front, and encode strings as cheaper-to-use numeric codes.

Parts of compilers that perform up-front string-scanning are called *scanners*.

In modern languages, scanners are simply glorified maps (hash tables). The scanner code collects up incoming characters and looks for word/phrase boundaries, usually defined by delimiters like whitespace. Once a complete word is found, it is replaced by a more-efficient code.

For example, the scanner might find the word "then" in the source code and replace it with a code number like 8. The downstream bits of the compiler "know" that 8 means "then", or they can look this information up in a table (map). The string "then" is usually 5 bytes long (four letters plus a NULL), whereas the code 8, is one byte long. The string "then" is potentially variable in length, whereas the code 8 is always fixed size. Variable-length string matching uses up more CPU cycles than using fixed-size codes, since, codes

are usually handled directly by the CPU whereas variable-string-matching usually requires many CPU instructions (usually in a Loop).

Delimeters are nothing special. Delimiters are simply characters. Each programming language defines a set of rules for what is considered to be a delimiter and what is considered to be some other kind of character. Over time, it has become conventional to treat whitespace as delimiters and characters in the ASCII range 33..127 as other kinds of characters. In fact, the range of characters is further sub-divided and classified as "digit", "letter", etc.

It has, also, become conventional to classify whole words into categories, broadly *keywords* and other *identifiers* and *operators*. Keywords are words that have special meaning in the language, like `if`, `then`, `else`, etc., and operators are usually single characters[2] that don't look like words to human eyes, e.g. `+`, `>`, `"`. Most other words fall into the bag of being *identifiers*, i.e. word-like strings that have no special meaning in the language, for example, words like "x", "xyz", etc.

Trees

Compiler-writers often structure their designs using *tree* data structures.

TODO: syntax tree

AST vs CST

Compiler-writers often use the abbreviation "AST" and "CST" to mean certain variations of tree data structures.

[2] sometimes more than one character, like "<=", ">>", "+=", "===", etc.

"AST" usually means *abstract syntax tree*.

"CST" usually means *concrete syntax tree*.

These data structures are almost the same, except that ASTs usually encode all of the *possible* choices whereas CSTs usually encode only what was pattern-matched in the incoming, user source code.

Usually, an AST forms a specification for the pattern matcher.

Usually, a CST is the output of a pattern matcher.

The pattern matcher is most often called a *parser*.

In analogy, a simple sample REGEX specification might look like:

```
1  .replace(/x\([uvw]\)z/,"\1")
```

In this case `/x\([uvw]\)z/` would be like the AST *before* the pattern-matching operation. `\1` would hold the CST *after* the pattern matching operation. Neither the AST nor the CST contain all of the user's input text, but the CST contains the users' input text broken down into pieces that were matched by the matching engine. In practice, a real CST produced by a parser would probably contain more pieces and more detail than can be shown in this simplified example.

In the above simple example, the AST encodes pattern-matching information such as:

1. first, match the letter "x"
2. then, match one letter which might be a "u" or a "v" or a "w".
3. then, match one more letter, "z".

This information is encoded in some way and handed to the pattern-matching engine. The engine is just a block of code that understands the encoding. The engine does the work of matching

the input source characters against the encoding. The encoding is usually in the form of a *tree*[3].

So, the engine receives a set of details encoded in a *tree* data structure and it produces a pared-down *tree* that contains a break-down of what was matched in the source code. The first kind of tree is called an AST. The second kind of tree is called a CST.

Tree Driven Compilation

A natural way to represent possible matches (parses) is to specify possibilities as *trees* and to construct pruned trees of what was actually matched in the source code. The specification tree is called an AST, and the match is called a CST.

Given that the pattern matcher - the parser - produces tree data structures, it is natural to hand bits of app code onto the trees, like Christmas ornaments hung on a bare Christmas tree, that do "the rest" of the work of coverting the input source code into arrays of binary opcodes.

Since pattern matching - scanning and parsing - has become easy to handle, the "rest of the work" is actually the difficult part of writing compilers.

This stuff is "just more code", but it can be overwhelmingly large and tricky to get right.

Firstly, the compiler needs to do semantic analysis, then it needs to do allocation, then it needs to emit dumb code, then it needs to optimize the dumb code to become better code, and so on.

Once a compiler has been fully written and debugged by hanging code bits off of the AST, the result is hard to understand - the code

[3]"Tree" is a shorthand word that programmers use to talk about a certain kind of data structure. Most normal humans like to draw equivalent data structures as a set of nested boxes, say on a whiteboard or on a scrap of paper. Programmers, though, like to unwind the nested boxes and draw them out in something that resembles an ORG Chart.

is a wall of overlapping details.

Can you improve on the use of trees?

Yes.

Can you improve on the whole process, breaking the compiler down into smaller, more understandable, more manageable chunks?

Yes.

Ohm-JS uses trees but keeps subsections highly separated. One section performs pattern matching, another section deals with the information contained in the trees. Using Ohm's concept of multiple semantic operations, it is possible to subdivide the big task of compiling text programs into smaller pieces each of which do "one thing well".

DSLs, APIs, Opcodes

As an aside, consider DSLs (Domain Specific Languages) and APIs (Application Programming Interfaces) and, even, CPUs themselves (opcodes).

The point of developing and using DSLs is to focus on smaller tasks. For example, SQL is a DSL designed for database manipulation. Instead of using ad-hoc code written in some 3GL programming language (e.g. Python, Haskell, Rust, etc.) it becomes possible to focus solely on the database aspect of a problem solution, using SQL.

APIs have a similar intent. APIs form tight funnels for accessing myriads of features provided by library code. APIs are tuned for focusing on specific problems and to eschew details that don't apply - at that level - to the specifc problems.

CPU opcodes are the same kind of thing. Internally, CPU chips are huge blobs of electronic circuitry. Opcodes define a narrow

access to the circuitry. The opcode API restricts what can be done, but covers the majority of use-cases needed to form an electronic solution to some problem.

Likewise, programming languages are APIs layered over top of opcodes. Programming Languages further restrict what is possible to do with the electronic circuitry, but, Programming Languages also provide checks for common mistakes made by programmers.

This DSL/API/opcode strategy can be taken further. One can define more and more specific solutions to use-cases. These solutions can build on existing DSL/API/opcode interfaces, tuning them for greater detail within smaller domains. From this viewpoint, something like SQL appears to be too grossly general. One doesn't want general solutions to the database problem, one wants specific solutions to specific applications of databases for specific use-cases. I use the term SCN - Solution Centric Notation - to mean such tiny, fine-tuned nano-DSLs. The main reason that SCNs are not more common is that building nano-DSLs appears to be difficult and time-consuming, causing programmers to program specific solutions using ad-hoc code written in 3GL programming languages, like Python, Rust, etc. The SCN "problem" then becomes "how does one build SCNs quickly (in an afternoon)?". There are ways to do this.

Do One Thing Well

The concept espoused by UNIX and especially shell pipelines is the idea of chopping a problem up into many smaller subdivisions, then addressing each subdivision in a more focussed manner without dealing with details that aren't related to the subdivision.

This doesn't mean that unrelated details are ignored, it simply means eliding details and not having them intrude on thinking about the problem/solution-at-hand.

This is much like what compilers do - they optimize programs by subdividing programs into two portions

1. the bits that need to be done at runtime
2. the bits that can be done before runtime. All of the details in the program are addressed, but, subdivisions are made as to *when* the details are addressed (compiletime or runtime)

This concept seems to have been buried by the mistaken belief that nested functions are the same as pipelines. Synchronous functions cannot chop a problem up into smaller independent parts because synchronous functions are synchronous and not independent. Synchronous functions rely on use of the callstack, whereas asynchronous components use queues instead of the callstack.

Syntax Driven Compilation

Compilers can be built in a pipeline manor subdividing the tasks into smaller parts, but, something needs to be used as glue to combine the smaller tasks into a larger whole.

Syntax-driven compilation uses “syntax” as the glue. Sequencing a walk through the syntax of a program allows programmers to hang bits of code off of the branches of the AST while focussing totally on a single sub-problem.

Syntax-driven compilation can be used to form a serial pipeline, where each stage in the pipeline uses the syntax of unique nano-DSLs to guide applications of bits of code.

For example, the first stage of semantic analysis is to gather up all of the declarations into a table, or a tree of tables based on scopes.

A syntax-driven first-pass of semantic analysis would accept a stripped-down syntax and look only at declarations in the language, skipping over all other constructs.

This first-pass would re-emit the incoming code mostly "as is", but remove all declarations. The subsequent passes are not interested in declarations, so they are not included in the downstream code. In essence, the CST that is sent further downstream is pruned to remove all of the details that have already been handled - in this case all declarations.

This first-pass of semantic analysis would visit every declaration and create entries in a symbol table. It would not bother to emit the declarations themselves. In places that this first-pass skips over incoming code, the first-pass simply re-emits the code to its output.

The parser pass, which precedes the semantic pass, would remove all syntactic noise to produce a machine-readable, albeit not human-friendly, tree of the important parts of the program-to-be-compiled.

This process is very similar to the concept of walking the CST, except that each pass is allowed to modify - pare down and reformat - the CST before sending it downstream to subsequent passes.

Efficiency

Changing a tree and then re-parsing it in subsequent passes might create efficiency issues.

Is it cheaper to keep the CST intact and to walk it multiple times, or, is it cheaper to modify the CST, or, is it cheaper to leave the CST unmodified and to simply skip over uninteresting parts?

From a Designers' perspective, it matters only in how one's attention is directed and focussed - it is easier for a human programmer to focus on a problem, when there is no noise to ignore [4]. I.E. using multiple passes and multiple nano-DSLs is useful in the Design phase of a project.

[4] This is similar to the debugging of "core dumps" in earlier programming systems. Eventually, core dumps were replaced by 1-line error messages that helped programmers to focus on mistakes and errors instead of spending time digging through too much information. I liken "throw" in modern languages to core dumps. Throws usually supply too much information and cause programmers to pause while trying to discern information that is meaningful to their specific problem (bug).

From a Production Engineers' perspective, though, size and speed matter - keeping the noise intact might help speed up the process or might use up less memory. I.E. from a Production Engineers' perspective, it is better to squish bits of code together in order to save time and space at runtime.

It might be possible build a smart editor that changes the opacity of already-considered CST nodes and allows nodes to be dynamically inserted and modified.

The pipeline strategy is, in fact, a 1950s optimization for keeping things small - K.I.S.S.[5]. In 1950, it was considered good form to make code footprints as small as possible.

Regardless of the 1950s emphasis which is no longer applicable, the KISS mentality is relevant today, in 2022++, as a way of subdividing problems into smaller and smaller bits, ignoring their actual code size.

Compiler Phases

Compilers can be subdivided into several main phases, regardless if the phases are made explicit or are conflated with code for other phases.

1. scanner
2. parser
3. semantic analysis
4. code emitter
5. optimization

[5] https://en.wikipedia.org/wiki/KISS_principle

Scanner

The Scanner phase accepts input text in the form of a stream of characters and outpus a revised stream where strings of characters have been replaced by Tokens.

Tokens are less expensive to handle in the subsequent phases.

Parser

The Parse is a pattern matching phase.

It checks that all tokens are arranged in a sequence that makes sense relative to the grammar of the language begin compiled.

When invalid token sequences are found, the Parser creates error messages.

The error messages produced by Parsers are called “syntax errors”.

A significant feature of Parser design is to keep the Parser chugging along when an error has been detected.

Parsers that quit after finding only the first error are considered unfriendly.

Again, in the 1950s, computers were much slower than in 2022++. Fixing only one syntax error at a time was a laborious, time-consuming process.

Semantic Analysis

The Semantics phase checks for mistakes that are not found in the Parsing phase.

To this end, Semantics analyzers consist of two main sub-phases:

1. gathering informatin from declarations

2. checking variable and identifier usage against gathered information.

The simplest checks include:

- checking for typos by ensuring that programmers have explicitly declared every identifier
- checking function calls by simply counting the number of parameters in function calls against how they are declared
- checking the "type" of every parameter in every function call against how the functions are declared
- checking the "type" of every variable and every expression to ensure that they are used in compatible ways, as defined by the programming language.

Modern type checkers do work well beyond the above-listed simple checks. One wonders whether the traditional parts of the Semantic Phase need to be further subdivided, e.g. shallow-check vs. deep-check. The driving questions would be "would the feedback to programmers be faster with such a subdivision? Is it worth doing deep-checks when violations of shallow-checks have been detected?"

Code Emitter

The Code Emitter phase produces first-cut code that corresponds to the program-to-be-compiled.

The pentultimate code emission strategy is used by GCC and espoused in Cordy's thesis "Orthogonal Code Generator"[6].

Code is first emitted for a fictitious architecture, then, secondly, fine-tuned and ported to the actual target architecture.

[6] https://books.google.ca/books/about/An_Orthogonal_Model_for_Code_Generation.html?id=X0OaMQEACAAJ&redir_esc=y

GCC uses RTL which was documented as a peephole technology by Fraser/Davidson[7]

5. optimization The optimizer refines the emitted code to be more optimal in some dimension, e.g. speed, size.

Some optimizations require global restructuring of the CST, which implies that this phase may precede the Code Emitter phase.

Full-blown optimization can be time-consuming and resource-consuming. Such full-blown optimization is not required during Design, but is required during Production Engineering. GCC supplies command-line options to control the amount of optimization, e.g -O options.

See the Dragon book[8] for further information about optimization.

Portability and Target Architectures

Each kind of CPU has a different set of opcodes with differing capabilities.

For example, some CPUs support operations, like ADD, that deal only with integers or certain sizes (e.g. 8-bits, 16-bits, etc). Other architectures support the same kinds of operations with larger ranges of operands (e.g. 32-bits and 64-bits in addition to smaller sizes).

The main factor driving such architectural decisions is cost. It costs more space on IC chips to support wider kinds of numbers. This cost affects factors like die size, pin count and final cost to the consumers.

Compilers, especially for modern languages, are expected to emit code for a wide class of architectures - and, to ensure that the code

[7]https://www.researchgate.net/publication/220404697_The_Design_and_Application_of_-a_Retargetable_Peephole_Optimizer

[8]https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools

is semantically the same for each architecture regardless of final instruction count and speed.

For example, programmers expect the C statement `a = b + c;` to work the same on 8-bit processors as well as on 64-bit processors. Type analysis provides information to compilers as to what kinds of instructions to emit. Peepholing[9] and MISTs[10] provide convenient ways to write decision trees that automate code sequence selection.

Peephole

A peepholer consists of a small, sliding window used to pattern match certain instruction sequences and to replace them by other instruction sequences.

For example, we invent a simple case:

```
1  ...
2  MOV R3,R5
3  MOV R0,1
4  MOV R2,R0
5  ADD R2,R3
6  ...
```

In this case, let us imagine a 2-instruction window.

In this case, the peepholer looks for redundant MOVes of a constant through one register into another, e.g.

```
1  MOV R0,1
2  MOV R2,R0
```

will be replaced by

[9] https://www.researchgate.net/publication/220404697_The_Design_and_Application_of_a_Retargetable_Peephole_Optimizer

[10] https://books.google.ca/books/about/An_Orthogonal_Model_for_Code_Generation.html?id=X0OaMQEACAAJ&redir_esc=y

```
MOV R2,1
```

The peepholer looks at the first pair of instructions

```
MOV R3,R5
MOV R0,1
```

and doesn't find the pattern that it's looking for. The peepholer does nothing and moves the window down to the next pair of instructions

```
MOV R0,1
MOV R2,R0
```

This is the pattern that it's looking for, so it replaces the pair of instructions with one instruction, then moves the window down to see:

```
MOV R2,1
ADD R2,R3
```

This does not match the pattern that it's looking for, so the peepholer does nothing, moves the window down and discovers that it is finished.

The final optimized code is:

```
...
MOV R3,R5
MOV R2,1
ADD R2,R3
...
```

which, in this case, is one instruction shorter (and faster).

It should be obvious that the window can be made larger and the set of patterns made more interesting and parameterized.

This strategy was easy to code up in *awk* and should be easy to code up in more recent languages that support REGEX, e.g. JavaScript, Python, etc. REGEX is not actually needed - any string matcher will do. REGEX makes it easy to specify string matches as long as the patterns don't cross line boundaries or break any REGEX rules.

MIST - Decision Tree

The term MIST means Machine Independent Strategy Tree.

The compiler produces code for a ficticious architecture that consists of a set of small operations - an IR (Intermediate Representation). For example, an IR operation might be `a := b + c`

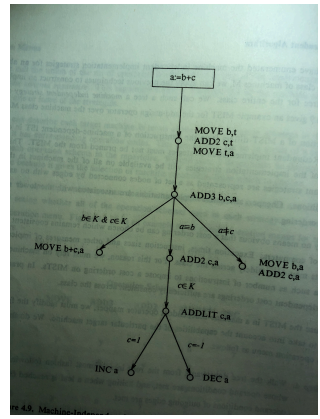
The tree is specified by a human compiler-writer and includes consideration for the kinds of operands that might be used with the IR. For example, the tree contains conditions like "if the operand is a constant integer that fits in 8 bits", or, "the operand is a temporary slot on the stack", or, "the operand is in a slot in the global heap", etc.

The compiler code includes an engine (a function) that walks the tree and selects the "best" code for a given intermediate instruction vs. a real CPU architecture.

Fundamentally a MIST encodes a decision tree - a set of choices for how code might be emitted

The snapshot below was taken from the original thesis. The MIST encodes some choices for `a:=b+c`. The default choice is near the top of the tree. As one moves down the tree, each desicion branch is guarded by a condition based on the operands, with "K" meaning "compile-time constant". For example, if the operand `b` is the same as `a` and the operand `c` is the constant `1`, the final choice is to use an

`INC` instruction. The thesis goes into more detail, including various IR instructions and various target architectures.



IMG_0090.png

REGEX - Regular Expressions

REGEX is a DSL for pattern matching strings on line boundaries.

REGEX syntax is fairly sparse. Patterns are specified as characters interspersed with special characters and symbols. REGEX specifies a way to capture sub-matches and names these submatches with numbers, e.g. `1`, `\2`, etc. When used for match replacement, REGEX can capture sub-matches and use the captures to create new strings.

For example, the JavaScript snippet below flips the matched characters and surrounds them with new (upper-case) characters.

```
1  var s = 'xaybz';
2  var s2 = s.replace (/x(a)y(b)z/, "F$2G$1H");
3  console.log (s2);
```

In UNIX versions of REGEX, the replacement numbers are preceded by backslashes, e.g. `\1`, `\2`, etc.

REGEX originated in the compiler world as a way to formalize the building of Scanners.

One of the first REGEX libraries, `regex.c`, was written by Henry Spencer[11]

REGEX can be found in various languages, like Perl, JavaScript, Python, Common Lisp, etc.

One can develop and test REGEXs on the website regex101[12].

The Dragon Book[13] explains algorithms for compiling REGEX syntax into state machines for pattern-matching strings.

BNF

A notation called Backus-Naur Form[14] - BNF - was invented for succinctly specifying pattern matchers.

Pattern matching specifications are called *grammars*.

Research into Language Theory focused on specifying *context free grammars*.

Context free grammars are a subset of more general grammars. Context free grammars consider patterns to match given any surrounding context. For example, "then" might be specified to match the same way anywhere in the source code. In a context free grammar the line

```
1  x = then
```

would always be considered to be

1. an identifier 'x'

[11]https://en.wikipedia.org/wiki/Henry_Spencer
[12]https://regex101.com
[13]https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools
[14]https://en.wikipedia.org/wiki/Backus-Naur_form

2. an operation '='
3. a keyword 'then'.

Whereas in a context-dependent grammar, the meaning of 'then' might be determined differently because it appears on the right-hand-side of an assignment. It might be that `then` is taken to be part of an 'if-then-else' statement in one context, but `then` is taken to be a variable name in an assignment context, like `x = then`. In this simple example, `then` has two different interpretations and the interpretations depend on the surrounding context.

All modern programming languages employ context free grammars. Very old languages, like some versions of FORTRAN, employed more ad-hoc heuristics for parsing characters that depended on surrounding context.

The context free grammar assumption makes parsing some kinds of sequences impossible, like recursive matching of matched brackets. (See PEG, below).

YACC, LEX, Bison, etc.

Several tools were developed for helping to write Scanners and Parsers.

Scanners were often written with the LEX program.

Parsers were often written with the YACC program.

Bison is a GNU variant of the YACC tools.

Newer compiler tools, like ANTLR and LLVM, have been developed.

PEG (Parsing Expression Grammars) work differently from YACC and CFG-based parser generators. PEG can parse patterns that CFGs can't.

- PEG
- WAM, gprolog, swipl?
- Small-C In the 1950s/1960s/1970s/1980s just about all software code was closed source. It was difficult to get to see how code was written.

Students at some universities were able to legally see the source code for UNIX. The cost for obtaining a source license for UNIX source code was high. Only some universities could afford to pay for license and to allow their students to see the code.

One magazine, Dr. Dobb's Journal of Calisthenics and Orthodontia, published the full source to a subset C compiler, called "Small C"[15].

The compiler could be compiled and run on PDP-11 computers and it produced assembler code for the 8080 CPU architecture. The assembler code produced by the compiler was in text form and could be inspected and understood.

The compiler did not optimize the generated assembler code. The compiler worked in a sub-optimal manner, using strings and string comparisons instead of tokens. This made it even easier to understand the mapping from C to 8080.

This simple compiler influenced a number of hobbyists and programmers, giving them a deep understanding of how compilers work.

PEG

PEG - Parsing Expression Grammars[16] automatically create Parsers in a manner that is significantly different from CFG-based parser generators.

[15] https://en.wikipedia.org/wiki/Small-C

[16] https://www.researchgate.net/publication/2882170_Parsing_Expression_Grammars

PEG makes it practical to parse constructs that CFG-based technologies cannot parse. Most notably, PEG can parse matching pairs of brackets. This simple difference makes is possible to imagine parsing in a new way. PEG makes it possible to write smaller grammars that "skip over" uninteresting bits of text. For example, to extract the names of all functions defined in a C file, using CFG technology, one must write a full grammar for C first. With PEG technology, it is possible to write grammars using less code and smaller grammars, saying something like "a function definition is an optional type, followed by a name followed by stuff enclosed in parentheses, followed by stuff enclosed in brace brackets". Only the function names are interesting - in this simple example - whereas all of the other stuff is noise that can be ignored. PEG allows you to loosely define "noise", whereas CFGs make you spell out all of the details contained in the "noise". So, instead of writing a huge grammar for a huge language, PEG lets us write smaller, 1-line grammars that match only what we're intested in.

PEG, though, produces confusing results when trying to parse source code that has mis-matched brackets, i.e. syntax errors.

This confusion is similar to what happens when strings in source code are improperly terminated and cross line boundaries.

Hence, small, PEG-based barnacle matchers are practical only for well-formed source. It might be better to use both technologies - CFG parsing while debugging syntax problems in source code, then PEG quickie parsers for everything else.

You don't imagine writing quikie parsers using CFG technologies, but, you can imagine writing such quickies using PEG technologies.

Ohm

Ohm, and Ohm-JS, are improved versions of PEG.

A subtle, but important, point is that PEG requires you to write grammars that include details about whitespace. Ohm fixes this problem.

Most PEG libraries, require you to embed "semantic" code into the grammar itself. Semantic code is the second half of pattern-matching-replacing programs. "Now that I've matched this input text, what do you want me to do with the matching bits?".

Embedding semantic code into grammars makes the grammars less readable. Grammars basically become walls of details and use-cases.

Ohm fixes this embedding problem.

Ohm grammars remain readable.

The semantics code is pushed off into another layer.

This may seem to be a minor UX detail, but it is significant. Keeping the grammar readable makes it possible for the Designer to concentrate only on the grammar at first. Once the grammar is debugged, the semantics can be added (elsewhere) and the additions make sense in the context of the Design. This kind of "separation of concerns" makes it possible to others, e.g. newly on-boarded employees, to come down the learning curve more quickly. Separation of concerns makes it possible to maintain and upgrade code with much more confidence and fewer bugs.

Ohm-JS, also, comes with a tool call Ohm-Editor. This is essentially a REPL for developing and debugging grammars. This tool is vital in speeding up development and making the result more robust.

Even if not using Ohm, one should consider using Ohm-Editor to help write grammars.

Ohm could become as useful as REGEX that is built-into programming languages. Ohm, based on PEG, can match text that REGEX cannot easily match. For example, Ohm can match multi-line constructs while REGEX was originally intended to match only

single-line constructs. Ohm can be used to match many modern programming languages, whereas, REGEX makes such matching hard work. Furthermore, the DSL syntax of REGEX is "uglier" than Ohm's syntax.

The use of a mixture of Ohm and REGEX matching, results in a new style of programming. Perl brought REGEX down from the compiler-building moutain. Ohm-JS can bring PEG down to programmers who would normally not consider building compilers, yet, want to perform simple pattern-matching chores, such as input validation. 1-line input validation is easily handled by REGEX. Ohm opens up the design space to include multi-line input validation and nano-syntaxes.

Call Return Spaghetti

Introduction

In this essay, I show that a diagram of a Call/Return system makes less sense than a diagram of a concurrent system.

I show the fundamental operation of a concurrent system and argue that it is inherently simpler than a system based on Call/Return.

Simple System

Fig. 1 contains a diagram of a simple system.

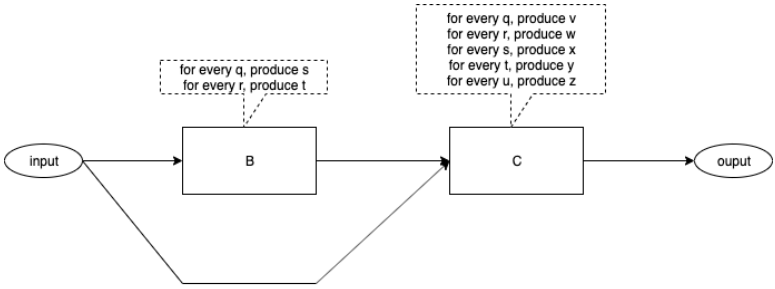


Fig. 1 A Simple System

The diagram contains one input[1] port and one output[2] port.

The diagram contains two components[3]. The algorithms for the boxes are straight-forward[4]. The algorithms are stated in terms of what each box outputs when inputs arrive at that box.

[1]The oval labelled "input".
[2]The oval labelled "output".
[3]Boxes labelled B and C.
[4]See the dashed callouts pointing to each box.

The flow of data within the diagram is shown by arrows.[5]

It appears that we have plugged two software components together to form a system.

What Happens?

What Happens When Events Arrive?

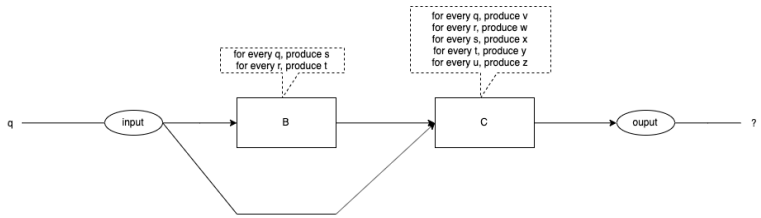


Fig. 2 An Event Arrives

What happens when an event arrives at the input? See Fig. 2.

The event, “q”, is injected into the input.

The algorithms specify exactly what each box does for any given input.

What is the expected output?

Do we see the expected output every time?

Do we see the expected output for every coding of the diagram?

Current State of the Art

The Code for Components B and C

We can implement the diagram in pseudo-code.

[5]The input flows to B and to C. B’s output flows to C. C’s output flows to the output.

Function B

```
function B(in) {
  if (in == q) {
    call C(s)
} else if (in == r) {
    call C(t)
} else {
  FatalError()
}
```

Fig. 3 Function B

Function C

```
function C(in) {
  if (in == q) {
    output <- v
 } else if (in == r) {
    output <- w
 } else if (in == s) {
    output <- x
 } else if (in == t) {
    output <- y
 } else if (in == u) {
    output <- z
} else {
   FatalError ()
}
```

Fig. 4 Function C

Code Version 1

Version 1 of the code might call component B first:

```
main () {
  call B(q)
  call C(q)
}
```

Fig. 1 Code Version 1

Code Version 2

Version 2 of the code might call C first:

```
main () {
  call C(q)
  call B(q)
 }
```

Fig. 1 Code Version 2

Final Output

Final Output The final output of the preceding routines depends on which version of the code we use.

Version 1 results in the following code path:

```
main {
 call B(q)
   B calls C(s)
     output <-- x
   C returns to B
  B returns to main
 call C(q)
    output <-- v
    C returns to main
}
main done
```

Fig. 1 Final Output for Version 1

The final output for Version 1 is x,v[6].

While version 2 results in the following code path:

```
main () {
  call C(q)
    output <-- v
  call B(q)
    B calls C(s)
      output <- x
      C returns to B
    B returns to main
}
main done
```

Fig. 1 Final Output for Version 2

The final output for Version 2 if v,x.

Version 1 and version 2 create different results.

[6] in left to right order

Control Flow

Fig. 5 shows the control flows for code versions 1 and 2.

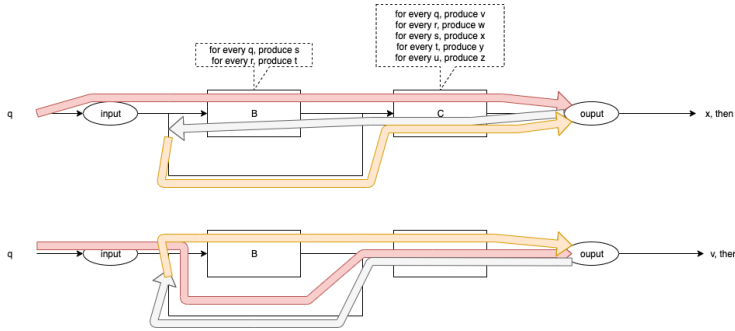


Fig. 5 Control Flow for Versions 1 & 2

The Desired Outcome

We want to plug software components together.[7]

We want the diagram(s) to mean exactly one thing.

We want the diagram(s) to mean the same thing every time.

This is possible.

I will show the event flow that we desire, in a series of diagrams, then, I will discuss how this flow can be achieved[8].

[7] After all, computer (digital) hardware is plugged together.
[8] Even on synchronous operating systems.

Event Delivery 1

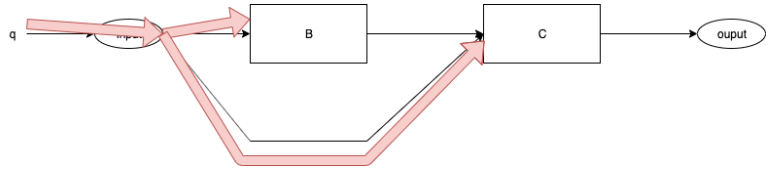


Fig. 5 Event q Delivered

Fig. 5 shows event “q” being delivered to B and C.

Nothing else happens, no routines are called.

After Event Delivery 1

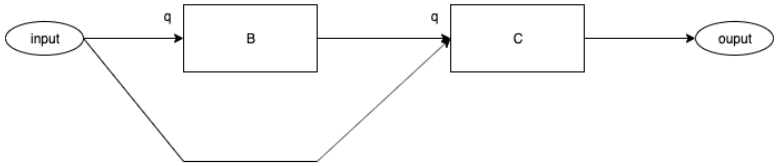


Fig. 5 After Event Delivery 1

Fig. 5 shows what the system looks like after Event Delivery 1 has occurred.

Both, Components B and C have an event “q” at their inputs.

(Neither Component has acted yet).

Two Possible Control Flow Paths

At this point, two control flow paths are possible:

1. Component B runs first.
2. Component C runs first.

I will draw a sequence of diagrams for each path.

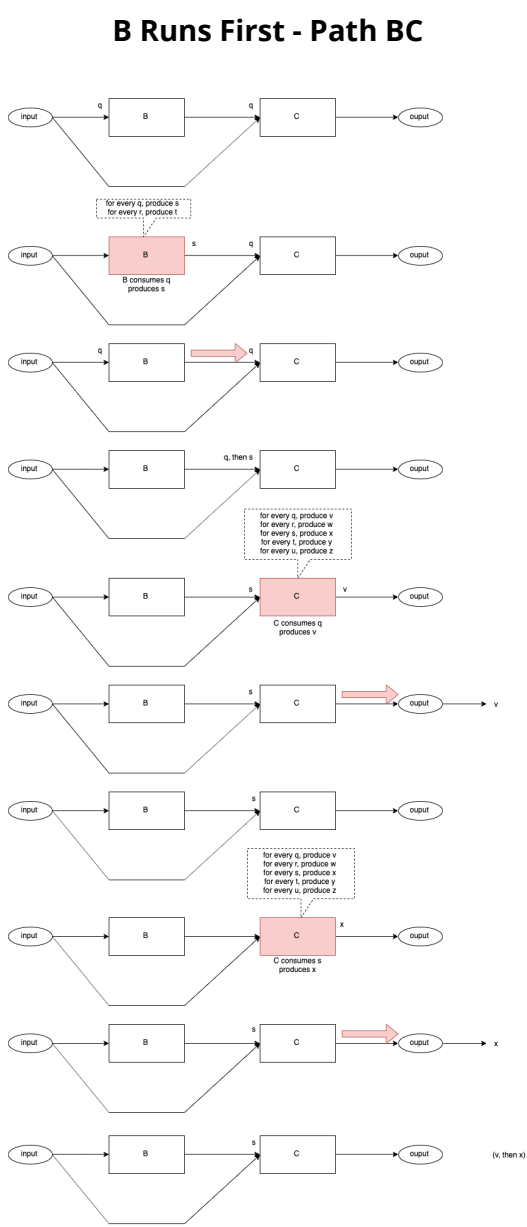


Fig. 6 Control Flow BC

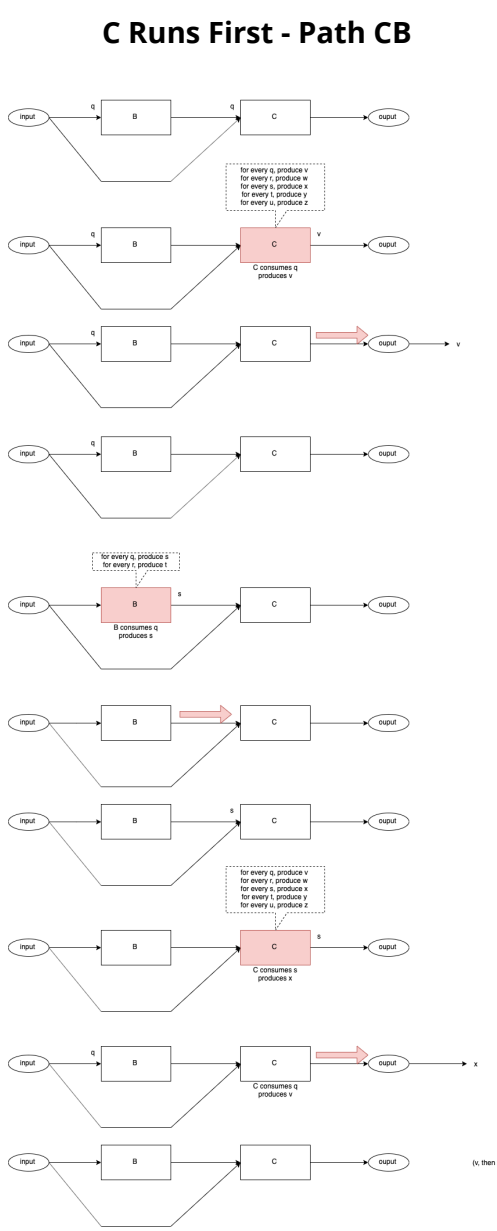


Fig. 7 Control Flow CB

Final Result

In both cases, Path BC and Path CB, the final result is the same - v is output first, then x is output.

Achieving the Desired Result

Requirements

- All Components have an input queue, onto which incoming events are placed.[9]
- Components cannot call one another.
- Components are asynchronous.
- A Dispatcher routine decides which Component will run and in what order[10].
- Components are ready if they have events in their input queues.
- Components consume one input event and produce as many output events[11] as necessary in reaction to the input event.
- Components perform a co-routine[12] dance with the Dispatcher. When a Component has processed a single event to completion, the the Component yields to the Dispatcher. The Dispatcher decides which Component will run next.[13] Components do not decide on the order of dispatching (as with call-return based code) nor can Components rely on any certain dispatching order.[14]

[9] In a production version, Components also have output queues. That requirement is a fine point, discussed elsewhere.

[10] The order is arbitrary. Components are fully asynchronous.

[11] 0 or more. In this example, each component produces exactly 1 output for each input, but this is not a requirement.

[12] FYI - This is easy to accomplish using closures and state-machine mentality. Discussed later. It is also easy to accomplish using threads, albeit this is overkill.

[13] There are many scheduling possibilities. For example, the Dispatcher may invoke a Component repeatedly until the Component's input queue is empty. Or, the Dispatcher may choose to work in a round-robin fashion. Fairness is not an issue (since a Component will eventually go idle when its input queue is empty).

[14] Components are truly asynchronous and must survive through any dispatching order.

- Events and data contained in events, are immutable.
- If a Component sends one event to multiple receivers, it must lock the routing wire.[15]

In addition,

- Components have no parameters, Send()[16] is used instead.
- Components have no return values. Send() is used instead.
- There is no syntax for exceptions[17]. Send() is used instead.

Using Threads

Operating system threads[18] can be used to trivially implement components.

Each Component has a mailbox[19] and it sends messages to it parent[20]. The parent routes the messages to the mailboxes of appropriate receivers[21].

Note - using threads is overkill. An operating system based thread involves hardware MMUs[22] and separate stacks. Operating system threads implement the out-dated notion of time-sharing. None of these are actually required to make this system work.

[15] In practical systems, this is not an issue. It becomes an issue for bare metal systems (no operating system) or systems where Components are distributed along "very slow" connections. I leave this "problem" to the Architect to solve in a manner suitable for the application. I simply want to give the Architect the tools to work with to build reliable systems. The Architect makes guarantees of reliability. This system provides only the bare minimum tool set.

[16] SEND() is the only mechanism for transferring data. Data can be transferred to any number of receivers up and down the line, using SEND(). There is no need for a specialized RETURN() expression.

[17] SEND() is the only mechanism for transferring data. Exceptions are simply data. Data can be transferred to any receiver using SEND().

[18] a.k.a. processes

[19] a.k.a. input queue

[20] which I call a schematic

[21] The routing information is based on the arrows on the diagram(s).

[22] Memory Management Units

Fairness

Fairness is not an issue.

Components run a single incoming event to completion, they yield only to the Dispatcher.

This system mimics, more closely, the modern ideas of distributed systems[23].

Thread Safety

Thread Safety is not an issue.

Components cannot share memory, hence, thread safety is not an issue.

Shared Memory

Shared memory is not an issue.

Components cannot share memory.

Components can only send immutable messages.

In very tightly coupled systems, we have the temptation to send pointers to large blobs of memory. The sender might mutate the blobs of memory before the message is read by the receiver.

This system makes no guarantees for such situations.

This system[24] gives the Architect all of the atomic tools necessary to create systems that work. For example, the memory-sharing issue was encountered 40+ years ago in TTL-based hardware systems. The solution was to use "double-buffering" and "overrun" flags. If a system could not switch between buffers quickly enough, then it created an overrun condition. A system which encountered overrun was deemed simply to be "too slow".

[23] e.g. IoT, multi-core and internet systems

[24] We call it Arrowgrams.

The Architect must make the calculation[25] of whether his/her design is "fast enough" for a given purpose.

Priority Inversion

Priority inversion is not an issue.

I don't use[26], nor specify priorities[27], hence, priority inversion cannot happen.[28]

Loops and Recursion

It turns out that Looping (and Recursion) is the exception, not the rule.

Components must not enter long-running loops (or deep recursion). Components must yield to the Dispatcher. Note that compilers could insert yields at the bottom of Loops to accomplish this behaviour.

The Dispatcher routine is the only routine in the system that runs a loop. It loops through a list of ready closures and, randomly, invokes a ready closure. When the closure finishes[29], the Dispatcher simply picks another ready closure to run.

Dynamic Routing

Dynamic routing is not an issue, because it's not supported.

Dynamic routing used to be called self-modifying code. Self modifying code is a bad idea.[30]

[25]Calculation is discussed elsewhere.

[26]This is not a flippant statement, regardless of how it sounds. It is based on hard-won experience with real products. Priorities can (almost) always be designed out of a system.

[27]This is system of atomic tools. Priorities are non-atomic and can be implemented using the atomic tools.

[28]If a system must act using priorities, then the Architect is referred to literature on hardware priorities1 fully documented some 40+ years ago. 1 NMI and IRQ levels.

[29]e.g. executes a RETURN

[30]Self-modifying code is especially bad from the maintenance perspective.

Using Closures

Most modern languages provide the concept of closures.[31] Closures might be called anonymous functions, or callbacks, or be embedded in concepts such as futures, etc.

Even C has a way to make closures, using function pointers.

The minimum closure required by this system requires some static, but not exported, data, and a temporary stack[32]. N.B. one stack, for the complete system, is enough[33].

OO[34] Objects and Blocks are very close, in principle, to the ideas expressed here. The difference is that I specify closures that run asynchronously whereas OO uses Objects that perform synchronous call-return and explicit naming of receivers. I recommend that closures send messages to their parents and do not refer to each other directly.

In my opinion, and experience, creating separate stacks for each closure and using MMUs is overkill. I don't wish to use time-sharing in my programs. I might use time-sharing if I were to build an operating system.[35]

If one imagines that closures contain state-machines, then, this method could be considered to be a system of communicating state machines. I think in terms of clockwork[36].

[31] Closures were explored in the 1950's using Lisp.

[32] A temporary stack is used for compiler-generated temporary values.

[33] One stack is needed for implemenationon von Neumann architectures. Maybe one stack is one too many in non-von Neumann architectures?

[34] Object Oriented

[35] I argue that we shouldn't use operating systems at all.

[36] Rendezvous techniques also form clockwork systems. Rendezvous has the drawback that it imposes synchronous operation of processes. This is anathema to concurrency. In a concurrent system, processes are asynchronous by default. Synchronization is the exception not the rule. If synchronization is needed, it must be explicitly designed (for example, see ACK/NAK protocols in networking, and, synchronization techniques used in TTL hardware of the 1980's).

Other Features

Reuse

This system emphasizes reuse of Architecture[37].

Architecture reuse is more valuable than code reuse.

Refactoring

Software Component Architectures composed in this manner can be easily refactored into other Architectures, simply by moving/deleting/adding arrows.

Components SEND() messages to their parents. Parents contain the routing tables[38]. Parents route messages between their children. This combination makes refactoring of Architectures easy[39].

Isolation

This system produces a natural hierarchical composition of Architectures.

Parents route messages between their children.

Parents act like Components in all other respects. Parents cannot SEND() messages to their peers. They can only SEND() messages upwards to their parents, and route messages of their direct children.

Global Variables

Global variables are not an issue.

[37]Reuse can be performed by cut/copy/paste. According to Paul Bassett, OO does not provide reuse. OO provides multiple-use (which leads to parameterization, which leads to accidental complexities, and so on).

[38]a.k.a. arrows

[39]Routing tables is nothing more than indirection. It also allows for reuse (as opposed to multiple use) of architectures.

Global variables cannot leak beyond the boundaries of their Components.

Global variables are not a problem, if properly encapsulated.

Global Types

Global Types in synchronous languages are just as bad as Global Variables in those languages.

Global anything is bad.

Encapsulation must be applied to every concept in software architecture.

Namespaces

A component has two external namespaces:

- The set of inputs.
- The set of outputs[40].

The internal namespace(s) of Components does not leak out.

All input names must be unique within the input namespace.

All output names must be unique within the output namespace.

The same name may appear in, both, the input and output namespaces.

Namespaces are naturally encapsulated in a hierarchical manner, due to hierarchical encapsulation of Components.

If two Components have exactly the same input namespace and the same output namespace, then the components are considered to be interchangeable, and "pin compatible".[41]

[40] I call them "input pins" and "output pins", resp., inspired by TTL hardware concepts.

[41] This is similar to referential transparency, but, without the constraint that pin-compatible components must produce the same outputs (this loosening of the same-output restriction allows upgrading).

Isolation of Control Flow

Control Flow within Components is naturally isolated by the fact that Components are truly asynchronous.

Control flow begins when a Component is invoked, and, control flow ends when the Component yields to the Dispatcher.

Control Flow does not leak beyond the boundaries of Components[42].

[42]Control Flow cannot leak because Components cannot CALL other components. Components can only SEND() messages upwards to their parents for routing. Corollary: the direction of SEND() is vertical-only - SEND() cannot be used to deliver messages sideways to peer components. Components can SEND() data upwards to their parent. A parent can route messages between children (and their own input and output pins). Parents can SEND() commands downwards to children.

Scalability

Complexity is not the problem.

There is no silver bullet. There are many silver bullets.

The main problem in software design is scalability.

We want to "plug" pieces tother like LEGO blocks.

Better scalability implies fewer dependencies.

Early hardware people got this "right". They took incredibly complicated devices (semiconductors made up of various kinds of rust) and built chips / ICs (integrated circuits).

Chips were black boxes. They had a set of input/output pins. The insides of the chips were inscrutable - encased in opaque epoxy.

Nothing leaked out of or into a chip except through the pins of the chip.

Properties of a chip were described in easily-measured terms:

- voltage on a pin
- current needed by a pin
- diagram / chart of the outputs, given a set of inputs
- timing.

Then, hardware designers "discovered" that point-to-point wiring between chips led to non-scalable designs.

They built a (small) hierarchy - chips mounted on boards plugged into backplanes.

The earliest backplanes were basically point-to-point wiring harnesses. For example, an early Wang word processor had a backplane with some 400 pins, allowing a chip on one board to send signals directly to a chip on another board.

Then, came the S100 bus. It had only 100 pins. It was well defined and documented. Certain connections were not allowed, even if they could be done more efficiently as point-to-point connections.

The idea of the Bus led to Apple computers and, ultimately, the IBM desktop computer. (There was more than one Bus definition, but the market shook those out).

Can software be built like chips? I argue Yes.

We need to build software in hierarchies.

Divide and conquer.

No leakage - of anything - between layers in a hierarchy. ("Anything" includes things like variables, types, control flow, dependencies of any kind, etc.).

Coming back to Complexity: we *don't care* how complicated a component is, as long as it is well-encapsulated and as long as we don't have to deal with any of its leakage of dependencies.

I see software as a hierarchy of black boxes. The Architect for each box chooses the best way to describe the design intent of a black box. The Engineer figures out how to dot the I's and cross the T's. The Production Engineer figures out how to make the black box "more efficient" and the Coder lays the bricks to implement the black box.

A good Architect will have a tool-belt full of Silver Bullets. Maybe a problem is best described in Relational terms, maybe a problem is best described as a State Machine (as a diagram, yet), maybe a problem can be broken down in a synchronous manner, etc., etc.

Insidious Form of Dependency

An insidious form of dependency that is overlooked is the "dynamic call chain" created by using a (usually hidden) stack to store state

and return addresses between function invocations. This dependency must be broken if software is to scale to new heights. Breaking this kind of dependency requires Concurrency. Concurrency, currently, has a bad name (i.e. it is thought to be a "hard" problem) because it has been tangled up with Time Sharing and Operating Systems. Most applications don't need Time Sharing and all applications, except Linux, MacOSX, Windows, etc., don't need to implement operating systems. Concurrency is *much* easier when Time-Sharing is removed.## Old Reality vs. New Reality Old reality - The old reality was: limited memory and one CPU (hence, the name "Central Processing Unit"). In this reality, it was reasonable to simulate concurrency and have one stack per process.

New Reality vs. Old Reality

New reality - Huge amounts of memory and many processors (none of those being "Central"). In this reality we can afford to have multiple stacks (e.g. one for each type) and use SEND() for every kind of data movement (displacing function parameters, return values, exceptions, all of which came about due to the Old Reality).

There is nothing "new" in the above ideas. Humanity has dealt with issues of isolation and timing many times before.

For example, businesses are built on the notion of hierarchy.

For example, music scores deal with real-time issues and sequencing.

Measuring Isolation

- No need for make
- No need for any package managers
- No need for tools like AutoConfig.

- Manuals that are only one page long.

Other measures:

- Using a hierarchy of DSLs to solve a given problem.
- Breaking out of a text-only syntax mentality, using DaS (Diagrams as Syntax).

5 Whys of Software Components

Q1: Why can't we plug software modules together like LEGO® blocks?

A: Because software is too complicated.

Q2: Why is software too complicated?

A: Because we can't plug software modules together like LEGO® blocks.

Q3: Why can't we plug software modules together?

A: Because the interfaces are way too complicated.

Q4: Why are interfaces too complicated?

A: Because libraries and APIs have hidden dependencies.

Q5: Why do libraries and APIs have hidden dependencies.

A: Because we use languages that hide dependencies.

Q6: What kind of dependencies are hidden by languages?

A: For one example, CALL / RETURN leaves breadcrumbs on stacks, forming dynamic dependency chains.

Q7: Why does CALL/RETURN leave breadcrumbs on the stack?

A: RETURN needs to follow the breadcrumbs back to the caller.

Q8: What happens when a CALLed routine calls another routine?

A: The callee becomes the caller and leaves another breadcrumb on the stack.

Q9: Why is a stack used? Why not use a "register"?

A: Stacks are used so that previously laid breadcrumbs are not overwritten. If we didn't have an automatic stack mechanism to save our breadcrumbs, we'd have to save them manually in some sort of list.

Q10: Is the stack of breadcrumbs a dynamic data structure?

A: Yes.

Q11: Is the stack of breadcrumbs a dependency chain?

A: Yes:

Q12: Is the stack not just a list?

A: Yes, but it is an optimized list.

Q13: Why is it necessary to optimize lists to make stacks?

A: Because we need to optimize memory usage.

Q14: Why do we need to optimize memory usage?

A: Because memory is expensive

Q15: Why is memory expensive?

A: Memory used to be expensive in the 1950's.

Q16: Is memory still expensive in 2020?

A: No.

Q17: Why do we use techniques to optimize memory usage, when memory is no longer expensive?

A: Uh, because we've always done it that way.

Q18: Did we notice that the ground truth has shifted? Memory used to be expensive, but is no longer expensive?

A: Uh, no, we didn't notice.

Q19: Why didn't we notice?

A: Uh, because we believe in building on the shoulders of others.

Q20: Are we in the weeds?

A: Yes.

Acknowledgement

5 Whys suggested by Daniel Pink in Masterclass: https://www.youtube.com/watch?v=My7hjBp4wH0

Git Could Do More

Github, Git, Diff, etc.

I consider "github" to be a whole gamut of technologies based on "diff".

- githhub
  - git
    * diff (a UNIX tool)

"Github" is a production version of "diff".

Automated DRY

Github could use "diff" in a bigger way. Instead of diff'ing lines of code, it should diff layers of design. (Which might be blocks of code).

Our PLs[1] are just glorified text editors.

A lot of what we think about and build into PLs is DRY[2].

RY[3] is what we want, but we waste brain power on DRY.

Machines could detect DRY for us. And refactor the code/design/etc. for us automagically.

Github could include technology to figure out DRY.

[1] PL means Programming Language
[2] DRY means Don't Repeat Yourself
[3] RY means Repeat Yourself

Git-based Editors

Our editors could figure out repeated lumps of text and make a golden copy for us.

The editor could show repeated lumps of source in gray and automagically update the golden copy when we edit the gray parts.

DRY vs. Component-Based Programming

One of the thrusts of Component Based Programming and of FDD is:

- Replace boiler-plate parts of your workflow with automation. Get the machine to do as much of your work as possible.

DRY (Don't Repeat Yourself) is a central lesson of Computer Science. But, it only applies to human-written code.

It's OK if the machine does RY, but if a human does RY, trouble looms (e.g. forgetting to update every instance of every piece of code).

Component-based programming allows copious RY. "Referential transparency" in FP is an RY-enabling feature. HyperCard was kinda component-based programming. Spreadsheets allow potzers to use RY more before they get into scalability issues.

Solutions that I know of:

1. Components. Wrap code into bundles and give each bundle an input API -AND- an output API. You can copy/paste and/or replace bundles that have exactly the same input and output APIs. All code goes into separate bundles. One can copy/paste bundled code without futzing with the code itself. When you change code inside a component, all other uses of that component see the same fix. Programmers believe in the

fiction that Libraries already do this, but, they don't, due to hidden dependencies caused by Call/Return.

2. If one utterly insists on using text, one might check out the research in "clone detection". I haven't checked this out, but I see that Cordy and Roy invented something called NiCad. Cordy is a simplicity hero. Cordy was mentored by Ric Holt. I learned a lot from Holt and Cordy.

Factbases

The trick to automating anything in software is to find a way to normalize the information.

We discovered how to normalize code very early on - we used a notation called assembler.

Assembler is code represented as triples - relation, subject, object.

For example

```
MOV R0, R1
```

is a triple. The relation is MOV. The subject is R0. The object is R1.

If the idea of triples sounds familiar, it's because you've already heard of it.

XML is triples[1]. The Semantic Web is triples.

So, what is the normalized format of data?

Triples!

We want to put data into the form: relation, subject, object.

Conveniently, this already exists. It is a function of two parameters:

```
relation(subject,object);
```

or,

```
fn(id,x);
```

Why is this a Good Thing?

Well, because we impose no structure on the data.

If the data has no structure, then it is easy to parse (in an automatic manner).

[1] XML has too much syntax to be easily readable.

If the data has no structure, then it can be used for other kinds of things. Things that the original programmer never thought of.

Note that organizing data into a data structure at compile time is just an optimization. In the 1950's, it seemed like a good idea to preserve CPU power by pre-compiling data into data structures.

Today, CPUs are cheap and abundant, we can waste CPU time building data structures at runtime. We don't have to worry about pre-compiling things.

How shall we waste CPU time?

By performing exhaustive search.

By using backtracking. Backtracking was verbotten in the 1900's, but no more.

I know, from experience, that I will be using PROLOG for doing exhaustive search. So, I will put a period ('.') at the end of each function and call this a fact.

```
fn(id,x).
```

Caution: use the principles of Shuhari here.

Shu: don't expand the definition of "x" in the above. It is one thing and one thing only.

For example, imagine that I have a rectangle R, with top-left (x,y) and a width, w, and a height, h.

In factbase notation, this becomes

```
1 rectangle(R,nil).
2 top_x(R,x).
3 top_y(R,y).
4 width(R,w).
5 height(R,h).
```

Resist the urge to do something like:

```
rectangle(R,[x,y,w,h]).
```

or

```
1 rectangle(R,nil).
2 top(R,[x,y]).
3 wh(R,[w,h]).
```

We will use CPU power[2] to glean various relationships about the data. In fact, one of the first relationships will be to create a bounding box for each rectangle (we will see this later).

CPU power is cheap.

Memory is cheap.

Don't waste brain power.

Don't try to predict the various ways in which data will be structured.[3]

PROLOG already knows how to deal with triples of the above form.

PROLOG knows how to search triples.

You know how to search triples (loops within loops within ...).

It gets boring after a while.

When it gets boring, automate.

Automation done before you know what you're doing is called premature optimization.

You can't know what you're doing unless you've done it manually several times[4].

Corollary: premature optimization happens in anything that is built before the 3rd version.

[2] exhaustive search

[3] Only tyrants tell you what to think. Be free. Let others be free.

[4] Fred Brooks says that you need to build something 2 times (and throw the results away). Then, knowing what you are doing, you build it a 3rd time.

Code is cheap, thinking is hard.

Code is cheap, experience is hard.

It is best to grow the factbase with new facts, rather than culling facts from the factbase. This allows maximum flexibility during design. Culling is just an optimization and should be left to Optimization Engineering.

Recap:

Code is normalized into triples, e.g. `MOV R0,R1`.

Data is normalized into triples, e.g. `top_x(R,x)`.

Factbases are the normalized (triple-ified) form of data.

Further Reading: see the Appendix for tools like gprolog, JS prolog, miniKanren, AWK.

The Universal Datatype

The Universal Datatype is a *relation*, e.g.

`relation(subject,object)`

Triples

Relations are also called a triples.

Assembler

`MOV R0,R1` is a triple.

Relation, Subject, Object.

Normalization

Data / Code represented as relations is normalized.

Normalization is the most-atomic form of representation.

Example:

`rectangle (5, 10, 20, 30)`

can be further atomized — normalized — as

```
rectangle (R1)
x (R1, 5)
y (R1, 10)
width (R1, 20)
height (R1, 30)
```

[Yes, normalization wastes space and CPU power, but, we have lots of each today.]

Factbase

I often use the term fact and put facts into a factbase.

Compilers

Compilers like triples, e.g. MOV R0,R1.

Optimization

Optimization is easier when target code/data has been normalized.

Peephole optimization is easy to do with even simple tools like awk when code / data has been normalized into triples.

Fraser/Davidson wrote a landmark paper[1] on peepholing which formed the basis of Gnu's gcc.

Normalizing code and optimizing it is not just for compilers. The techniques could be ratcheted up a notch to cover higher levels of software Architectures.

[1] https://dl.acm.org/doi/10.1145/357094.357098

Anecdote - Y2K and COBOL

We analyzed banking source code for Y2K problems.

We used TXL to convert all source code into factbases, then ran backtracking pattern-matching rules over the normalized code.

Pattern Matching Factbases

Backtracking

Exhaustive matching can be done with simple algorithms — backtracking.

Backtracking is easier when the data is normalized.

[That's why compiler writers aim at assembler when writing optimizers. Today, trees are used, but trees get in the way.]

PROLOG

PROLOG is one of the earliest attempts at backtracking.

[I have built a PROLOG in JavaScript. See https://guitarvydas.github.io/]

TXL

TXL is a functional, backtracking, parser language.

http://www.txl.ca/

MiniKanren

MiniKanren appears to be the successor to PROLOG-like languages.

MiniKanren can do seemingly-magical things https://www.youtube.com/watch?v=elLvkklsk (and https://github.com/webyrd/Barliman).

One has to wonder what the child of MiniKanren and AI might turn out like.

Programming Language Design

Imagine if all code were normalized to triples.

We'd be programming in assembler, or in the mostly-syntaxless Lisp.

Successive programming language designs have tried to remedy the problems of working in triples, for human consumption.

Programming languages have taken years to design and to perfect.

Now, using PEG parsers, we can build languages in a day[2].

We can tune a language for a specific problem.

I call these SCLs — Solution Centric Languages.

PEG vs. YACC

YACC embodies LR(k) theory.

YACC builds languages from the ground up.

PEG builds parsers.

Programmers can understand PEG. Compiler-writers understand (mostly) YACC.

[2]Especially if we cheat.

PEGs are easier to use than YACC.

YACC needs a scanner, e.g. LEX.

PEGs are all-in-one - scanner and parser, utilizing familiar REGEXP-like syntax.

PEG vs. REGEXP

PEG is like REGEXP, only better.

If you use REGEXPs, stop.

Use PEGs instead.

PEGs make it easy to match sequences that REGEXPs have a hard time with.[3]

Automation

Normalization leads to automation.

First, make it repetitive and boring.

Then automate.

Programming

Programming consists of two basic activities:

1. breathe in — pattern match
2. breathe out — rearrange and emit.[4]

[3] The difference lies in the fact that PEGs use a stack and allow you to easily write pattern-matching subroutines.

[4] (2) might also involve actions

If (2) occurs before (1) is finished, we get problems. FP is an attempt to fix such problems by throwing the baby out with the bathwater. State is not the problem — unscoped use of State is the problem].[5]

[5]https://guitarvydas.github.io/2020/12/09/Isolation.html. See, also, Structured Programming, StateCharts, etc.

Triples

Triples are everywhere, albeit optimized into oblivion.

Compiler-people like triples.

For example, `MOV R0,R1` is a triple (relation=`MOV`, subject=`R0`, object=`R1`).

Triples are like the *atoms* of programming. If you want to know what the nuts and bolts are, you look at the atoms. If you want to build things, though, you use *molecules* based on *atoms*.

This is simply a form of *divide and conquer*.

XML

AFAIK, XML started out life as triples. This doesn't mean that I am right, it is only what I believe.

Example of RDF triples:

https://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HTML/ch07s01.h

Example

An example is given here, from

https://www.javatpoint.com/xml-example

```
abc
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

As Triples

note id234 null from id234 “Jani” to id234 “Tove” heading id234 “Reminder” body id234 “Don’t forget me this weekend!”

Computer Science

All of CS breaks down into triples - something I call “simple”.

All of CS has been about trying to optimize triples to save space and CPU.

Very 1950’s.

In 202x’s, memory is cheap, CPU’s are cheap.

Time to rethink.

Human-time is still expensive.

Waste computer-time, not human-time.

Data Structures

Data structures are just optimizations of triples.

Data structures at compile-time are

- 1950's notions of how to optimize for CPU usage
- Design Intent (more human-readable, but hard to automatically optimize).

Curried Functions

Even curried functions are triples

```
fn x y -> (fn x) y
```

`(fn x) y` - looks like a double, but is really a triple - relation=fn, subject=x, object=y.

`(fn x)` is a double.

`(fn x) y` is a triple.

`(fn x)` is an unresolved triple.

PROLOG

PROLOG allows you to go into feature-itis, creating quadruples, quintuples, etc., but they are just optimizations of things that are fundamentally triples.

Anything in PROLOG that is "more than" a triple is a layer.

PROLOG is (can be):

- describing a problem as relations (triples)
- wasting CPU power rebuilding data structures (better than wasting human intellectual power).

Human Readability

Humans shouldn't have to read triples.

But, if you deconstruct everything into triples, then it becomes easier to write DSLs and the like. DSLs and languages are simply syntactic skins wrapped over bags of nuts and bolts (triples).

Agile TakeAways

Goal of Agile

The goal of Agile is to deal with the uncertainty of the Design process.

Agile deals with design uncertainty by involving the stake holder(s) multiple times during the design process.

Most customers, stake-holders, cannot specify Requirements with enough grueling detail to allow implementation. Most customers, stake-holders, do not want to deal with all of the details.

Agile is just a stop-gap measure applied to Design, until we find a better way to perform Design with less uncertainty.

Religion of Agile

Agile has become dogmatic - a religion.

Many programmers focus on the rituals of Agile, without addressing the goals of Agile.

Takeaways from Agile

- Stake-holders cannot (will not) specify all of the details of a problem.

Anti-Takeways from Agile

Sprints Are Too Long

2/3-week sprints are too long.

We should be striving for continuous improvement in the process of gathering requirements.

2 Hour Sprints

We should be striving for 2-hour sprints.

Redefining Sprints

We should be striving to change the concept of sprints into something even more productive.

Flexibility

We want:

- flexibility in the Design
- optimal behavior in the field.

This suggests that we should break programming down into two major categories:

1. Flexible Design
2. Optimization for Delivery.

The categories have different goals.

We should treat each category differently.

We should not conflate both categories into the same activity, nor the same PL[1].

Inflexibility

Optimization removes flexibility to achieve better performance.

Premature optimization is pervasive, yet hidden, in most apps written with current PLs.

Tell: Choosing Programming Language First

Choosing a PL before fully understanding a problem leads to later inflexibility.

The only "good" reason to hire programmers experienced in a certain language is to maintain and upgrade existing code.

Tell: Choosing Only One Programming Language

If the programming language does not fit the problem, aspects of the problem-solution must be force-fit into the programming language, e.g. Relational Programming languages do not fit output formattting (well).

The use of only one programming language is a tell.

All aspects of the solution might not suit themselves to a single, chosen language.

(Sometimes, all aspects of the solution do fit a given language. This is an exception, not the rule).

[1]PL means Programming Language

The use of a single programming language - for design - is indicative of the "programmers are interchangeable" mind-set, instead of the "best-of" mind-set. For example, Apple Computers lets the cost of their computers be determined by a "best-of features" mind-set, instead of allowing cost to drive which features are included.

Note that I argue that PLs are cheap to build[2], today.

Indirection

Indirection retains flexibility at the expense of optimization.

Multiple Syntaxes

Designs often have multiple facets.

Each facet might be best expressed in a different syntax.

Hence, an app should employ multiple syntaxes.

[I argue that SCNs - Solution Centric Notations - should be used. SCNs are cheap to build (hours instead of years). Multiple SCNs can be tuned for a specific problem. SCNs are so cheap, that one can build SCNs for a specific problem instead of resorting to one off-the-shelf PL].

Diagrammatic Syntaxes

There is no reason to use only text-based notations for a given problem.

Diagrams can be transpiled to executable code. I detail how to do this in other essays.

[2] "Cheap to build" means hours, not years, for implementation.

Architects' toolbelts[3] should include many textual and diagrammatic PLs.

Whiteboards

Most stake-holders and most CTOs use whiteboards to present their ideas to implementors.

Q: Can we capture the whiteboard drawings?

A: Yes, using photos and drawing editors.

Q: Can we manually transpile drawings into working code?

A: Yes, I document how to do this in other essays.

Q: Can we automatically transpile drawings into working code? (I.E. can we build compilers for diagrammatic languages?)

A: Yes, I document how to do this in other essays[4].

Reuse In The Large

It is not enough to reuse code.

It is not enough to reuse Architecture.

We should reuse organizational principles from other professions.

For example, Architecture/Engineering/Construction organizational structure could be borrowed from the construction industry.

Most software organizations don't yet understand that Engineering is not Coding.

[3] Paradigms are more important than syntax

[4] Modern hardware can run exhaustive-search languages, e.g. PROLOG and Relational Programming, which makes transpiling diagrams-to-code simple.

Code is cheap.

Code is cheap.

Thinking is hard.

Software Development Roles

See Software Development Roles[5].

[5] https://guitarvydas.github.io/2020/12/10/Software-Development-Roles.html

Compilers Are Too Slow

Oh, the irony!

The premature-optimization clique convinced programmers that interpreters were "too slow" and that compilers should be used instead.

Now, 50+ years down the road, we find that premature-optimization has resulted in:

- workflows that are too slow
- Mb of bloatware
- techniques that are so complicated that they can't be explained to the majority of professional programmers (think website makers, game makers, JS crafters, etc.)
- acceptance of buggy products - programmers just ship and update weekly, forgetting Q/A

The implication is obvious:

- don't use compilers until you have designs fleshed out and fully working[1][2]

Use compilers only when you need to Production Engineer a Design.

Failing that, buy a faster development system and keep complaining...

[1] It is OK to use dynamic languages to experiment with and develop designs. The idea that you know everything about a design without experimentation is called The Waterfall Method. This, The Waterfall Method, is what is considered to be "programming" currently. Programmers dive into a project with the assumption that they will succeed. The infrastructure for intermediate failures is not built up, programmers jump directly to optimizing designs (which are, at first, untested designs).

[2] PL means Programming Language

Efficiency

"Efficiency" is in the eye of the beholder.

1. If you are Designing a piece of software, then "efficiency" equates to turn-around time based on using a souped-up development machine.
2. If you are Production Engineering a piece of software to hand to clients, "efficiency" means making the result runnable on low-cost hardware (or lowering the cost-to-the-user)

Sector Lisp, FP in < 512 Bytes

FP (Functional Programming) can be small and beautiful.

Sector Lisp is smaller than 512 bytes ([sic], not K, not M, not G, but bytes) on current hardware.

Explain why your favourite language *needs* to be larger and slower than this.

Compiler-writer addage: the only good optimization is one that makes the overall code smaller, when the code for the optimization is included in the code base.

Forgotten

Programming is about making (electronic) machines do repetitive tasks for humans, in any way possible.

There is no rule that says that programmers *must* use characters and text to write programs.

CEOs and visionaries use white-boards and napkins.

We would *prefer* that our machines would tighten-up the process of Design, at least a bit.

We can use manual methods until automated transpilers appear.

Notation Worship

Programming language design has become a cult of notation-worship, where, for no good reason, it is presumed that programming notation must consist of grids of non-overlapping sequential small bitmaps, as invented in the mid-1900s.

This technique hasn't been updated to take into account 202x hardware advances.

Error Checking - Silly Mistakes

Yes, we would prefer that machines check our work for silly mistakes.

We already have syntax checkers.

Types are more-of-the-same kind of checking, but, currently, result in bloatware.

In developing projects in JavaScript, I found that a large majority of my bugs could have been caught - rapidly, using few CPU resources - if the system simply checked for simple type violations, like number-of-parameters to functions.

I didn't need full-blown type-checking, I didn't need TypeScript. I just needed something that would have caught my typos and alerted me to them (e.g. "function defined to take 3 args, but only 2 args were given in line XXX").

Most programming languages already do this. JavaScript doesn't[3].

[3] I am purposefully ignoring *strict mode* and *typescript*.

JavaScript provides a default value for missing parameters (i.e. “undefined”).

5-Line Programs

Programs that are only 5 lines long don’t need type-checking and local variables.

The goal shouldn’t be to appease the act of building 6-line programs, but, to ensure that programs do not grow beyond 5 lines in length.

Silly? Q: What is the average length of a TensorFlow script written in Python?

[5 is just a ball-park number meant for emphasis. Font size and window size play roles in locality-of-reference.]

[[Essays/5 Line Programs]]

Why Don’t We Use Diagrams For Programming?

Why don’t we use diagrams for writing programs?

A: because mid-1900s hardware did not support inexpensive overlapping bitmaps.

Dependencies

Why do we have *any* dependencies between software components?

Our programming languages, supposedly higher-level than assembler, should inhibit, not encourage, the creation of dependencies.

Depedencies come in several flavours, e.g.

- hard-wired names of called functions
- synchronocity-under-the-hood
- CALL/RETURN.

Programmers can make it their goal to expunge all dependencies.

Eating dogfood: expunge the use of *make*, *npm*, *packages*, *package managers*, using CALL/RETURN to invoke code libraries.

Gedanken Exercises

1. You have an apple and a ball on the table. You cut the apple in half. What happens to the ball? Does the ball depend on the apple?
2. You have a 2-story castle built out of LEGO pieces. You carefully remove the top-most piece and cut it in half. What happens to the castle?
3. You have a main piece of software that calls a library. You make a change to the library. What happens to the main piece of software? Can you *guarantee* that the main piece does not change?

Humans understand the notion of independence and free will. Programming does not, easily, support independence and defies human intuition. I argue that this is a main reason as to why programming is considered difficult - programming defies our intuition.

Simplicity - How Do You Build A Light Airplane?

How do you build a light airplane?

1. Build an airplane,
2. Add lightness.

L. J. (Ted) Rootham.

I.E. If you want something simple, start out with something simple. Adding baubles will not make something simpler.

The things that we call *programming languages* are simply notations for viewing and describing aspects of programming. As soon as a notation appears "difficult", programmers should switch to another notation instead of trying to force-fit all views of a problem into one notation. As already mentioned, Sector Lisp is an example of how simple a notation can be and how complex that same notation can become when force-fitted into situations that it does not describe well (e.g. functional programming vs. mutation and heaps and distributed programming and etc.).

What else is considered "difficult" today and could use a different notation? Multi-tasking, for one. The presence of dependencies, for another.

Suggestion: Type Checking Design Rules

Another suggestion: Do what compiler-writers did when they only wanted YACC to help, but didn't want all of the niggly details that came with using YACC. They used YACC to error-check their grammars, then they rewrote the grammars in another programming language.

Likewise, if ultra-type-checking helps you create a Design, use a language that gives you ultra-type-checking (e.g. Haskell) to check the consistency of a Design, then rewrite the code in some other language (e.g. JavaScript). You are responsible to not hack on the

rewritten code, if you ever want to change the Design and have it re-checked. This is not a huge problem. If programmers were to use Haskell->other-language-transpilation enough, someone would automate the process. Better yet, someone will find a better way to express type-checking than using full-blown Haskell.

Yes, it would be nice to have type-checking integrated into the Design loop, separated from compilation, but we're not there yet.

Still, some kind of checkable design rules might might be more useful than ad-hoc white-boarding.

ATM, I draw my designs in draw.io, then hand-compile to JS/Python. With Ohm-JS to help me, this is so easy that I haven't bothered to close the loop to use diagram-parsing technology to automate these steps.

Warping Programming Languages To Allow Compilation

Don't let the premature-optimization clique tell you what techniques can and cannot be used (e.g. REPLs) for programming.

Ask them them to build lint-ers instead of arguing that programming languages need to be altered for their whims.

Ohm-JS, PEG, TXL make the process of writing syntax-driven linters easier.

Appendix References

Sector Lisp[4]

[4] https://justine.lol/sectorlisp/

Why Do We Use Text For Programming Languages?

The only reason that we use text and not diagrams for programs is that in the 1950s, computer hardware could handle text, but couldn't handle vector graphics. We developed more modern technologies for non-programmers, but, programmers themselves are still stuck in the 1950s.

Worse yet, 1950s computers could only handle text that was arranged on grids of non-overlapping, fixed-size bitmaps.

Well, OK, the other reason for using textual programming languages is that we are used to using writing technologies based on clay tablets. After 2,000 years, we've progressed to using graphite and rubber on paper. We didn't imagine that computers were something other than "better clay tablets". Our continued use of 2D writing technologies has severely limited the way that we think about things. We write equations in 2D to express simple physical concepts, like sound, that are 4D phenomena (x/y/z/t). Mathematics seems complicated to many people, because it is not natural and is a round-about way of describing multi-dimensional phenomena that we experience daily. Complexity comes from the fact that we try to describe 4D effects by collapsing them down to 2D. For example, Mechanical Engineers are taught to draw multiple 2D views of simple 4D objects (top, front, side, simply ignoring time).

Computers allow us - for the first time? - to express and explore at least 4 dimensions of our reality. We can shift the camera perspective on a 3D model and "walk around it" - and - we can watch models evolve over time. We couldn't do this easily with

paper and other clay-tablet technologies and mathematics. They key word here is "easily". Mathematicians and Physicists are able to build 2D "models" of real phenomena, but only a select few can understand what's going on (usually after several years of University training). It seems that our notions about programming are based on the idea that "if it's good enough for mathematicians, it must be good for programming, too".

Further: Note that it is just plain hard to draw a sensible diagram using a synchronous (function-based) language. It becomes laughingly simple to draw diagrams if you use 0D components.

Rhetorical question: why can we draw diagrams of computer networks? Because each node on the network is - essentially - a 0D component.

Why don't we do everything this way?

Uh, because of our built-in fear-bias regarding "efficiency". We are preconditioned to worry about saving bytes instead of saving development time. Without handcuffing ourselves with clay-tablet 2D technologies, we *could* trickle-down better apps to end-users, but, we don't.

Further further: computers in 2022++ are *fundamentally different* from computers in the 1950s. In the 1950s, each computer had only 1 CPU and limited memory. In 2022++ we want to build distributed systems such as blockchain, IoT, robotics, internet, etc. Ideas from 1950s don't map very well to 2022++. But, we continue to force ourselves to use ideas from the 1950s. From a product-design perspective, modern CPUs are little more than fancy FPGAs. Modern CPUs don't deserve all of the adulation we heap upon them in the form of creating Sciences for building 1950s UIs (aka "programmming languages") and 1960s IDEs (aka "operating systems"). Our end-users suffer from our tools' deficiencies - we insist on having end-users pay premium prices for bloated operating systems and we insist on having end-users accept the fact that whatever

they buy will be buggy without recourse in Law[1].

Sigh.

[1]Real Engineers are forced to take courses in Tort Law and are reminded of their obligation to produce bug-free products lest they be sued. For example, in Canada, Engineers are given "iron rings", on graduation, to remind them of a buggy bridge design that turned out to be fatal.

FDD - Failure Driven Design

Slides

Failure is the Best Way to Learn

Two Ways of Looking at Development

1. It's going succeed
2. It's going to fail

Outlook Determines Workflow

- How do you write software if you are convinced that it will work the first time?
- How do you write software if you are convinced that it will fail?

FDD vs Regular Design

- FDD
  - How to build-in easy recovery from changes (failures)
  - meta-design
- Regular Design
  - How to build it so that it works first time, ignoring possibility of changes / failure
  - A to B design, straight ahead

Assuming Success

- Waterfall workflow
  - one-way, one direction (e.g. top to bottom)
  - no plan to iterate
  - failure comes as a "surprise"
    - hard to recover from failure

Waterfall

- antithesis of FDD == overconfidence == Waterfall design
- Waterfall: one direction: design->implement
- Waterfall: assumed that the design will succeed the first time
- Waterfall: early attempts / requirements fail to completely solve the problem, but, no recovery from failure is built into the workflow

---

Mythical Man Month

- Fred Brooks - fail, fail, succeed

Assuming Failure

- how to fail fast?
- how to recover quickly?

Failure

- The first several attempts at solving a problem will fail.

Failure vs. Success

- Development:
- When software works, we “abandon” it (ship it)
- When software fails, we continue working on it
- Most of the time, we work on failing / failed code

What Can Fail?

- Requirements
- Design
- Architecture
- Engineering
- Implementation
- Testability
  - for example: testability can drive a change back into Architecture, etc.

Learn by Failing

- Failure is said to be the best way to learn.
- What do we need to learn?
  - what the requirements are
  - all aspects of the problem space (the gotchas)

FDD - Strategies to Make Failure Less Painful / Bothersome

- iteration
- recursion / divide-and-conquer
- automation - rearrange, then push a button to rebuild everything

- layering design (see "Recursive Design, Iterative Design By Example (2), section "Bug 2" and section "Layering Solutions")
  - https://guitarvydas.github.io/2021/04/20/Recursive-Design,-Iterative-Design-By-Example-(2).html
- indirection
- create a notation, SCN (low-cost)
  - punt to toolbox languages
  - punt to foreign functions (DI & Details Kill)
- asking Why?

FDD

- Failure-Driven Development
- most of the time, the requirements will change
- most of the time, a design will have flaws in it
- most of the time, the implementation will need to be debugged and need repairs
- the number of failures >> the number of successes
- plan for failure, since failure happens more often than success

---

FDD How?

- fail fast
- build in backtracking => automation
- look to compiler technologies, transpilation
- Notations, not Languages nor DSLs

Notations, Not Languages

- Notation is a lightweight DSL
- Programming Language: heavyweight, high cost to build
- DSL: heavyweight, high cost to build
- Specialize Notation to Problem Space only
  - avoid generalizing
- YAGNI - You Aren't Going to Need It

Fail Fast

- divide problem, choose greatest risk, greatest unknown
- experiment with / implement unknown
- if unknown becomes known, defer it and choose next greatest risk (which is, now, the greatest risk)
- if unknown is "impossible", then fail and backtrack
  - redefine the problem / solution
- Testing cannot prove that a device works, but testing can prove that a device does not meet its specifications

---

Scientific Method is a Fail Fast Methodology

- A scientific theory is one which is falsifiable
- one can't prove a theory to be correct - 1 data point can only support a theory, but cannot prove it
- one can only disprove a theory - 1 data point can kill a theory

---

FDD How? Backtracking

- script everything, push a button to rebuild
- when a design fails,
  - repair requirements
  - repair the design
  - re-generate
  - try again

FDD How? Compiler Technology

- compilers pioneered automated transforms
- compilers pioneered portability

FDD How?

- don't write code
- write code that generates code

Failure : Automation

- If you assume that you will fail, you are encouraged to use automation and backtracking
- FDD workflow is: repair, push button and regenerate, try again
- Waterfall workflow is: confidence that it will work, design, then implement

  - waterfall: no assumption that attempts will fail most of the time,
  - success is assumed

Automation : Factbases

- To automate, use compiler technology
- MOV R1,R0 is a triple
- everything is a triple => easier to automate
- RTL, OCG, portability ... <= normalization
- Projectional Editing <= normalization

What is the LCD for Automation?

- Q: What is the LCD - Lowest Common Denominator?
- A: triples
- triple = `relation(subject,object)`
- curried function is `relation(subject)`, later applied to `object`

  - i.e. double X single => triple

Manual vs. Automated

- Manual work resists change.
- The time spent is not recoverable.
- Automated work accommodates change.

Are You Ever Finished?

- No, but you reach a point where a product can be shipped
- analogy: songwriting -
  - songwriter continuously tinkers with a song
  - but, making a recording draws a line
  - "Aqualung" live is, now, almost unrecognizable (jazzy beginning)
    * audience member yelled "play Aqualung" during the new intro
  - songs continue to evolve

Appendix

Appendix - DI

Design Intent

https://guitarvydas.github.io/2021/04/11/DI.html

https://guitarvydas.github.io/2020/12/09/DI-Design-Intent.html

---

Appendix - Recursive Design

https://guitarvydas.github.io/2020/12/09/Divide-and-Conquer-is-Recursive-Design.html

https://guitarvydas.github.io/2021/04/12/Recursive-Iterative-Design-By-Example.html

https://guitarvydas.github.io/2021/04/20/Recursive-Design,-Iterative-Design-By-Example-(2).html

https://guitarvydas.github.io/2021/03/18/Divide-and-Conquer-in-PLs.html

https://guitarvydas.github.io/2021/03/06/Divide-and-Conquer-YAGNI.html

https://guitarvydas.github.io/2020/12/09/Divide-and-Conquer.html

---

Appendix - Factbases

https://guitarvydas.github.io/2021/01/17/Factbases.html

https://guitarvydas.github.io/2021/03/16/Triples.html

Appendix - SCN - Notations

Solution Centric Notations

https://guitarvydas.github.io/2021/04/10/SCN.html

Appendix - Indirection

https://guitarvydas.github.io/2021/03/16/Indirect-Calls.html

Appendix - Toolbox Languages

https://guitarvydas.github.io/2021/03/16/Toolbox-Languages.html

---

Appendix - Why?

I watched Daniel Pink's Masterclass

Pink suggests asking "why?" repetitively, some 5 times to understand the problem more deeply

---

Appendix - 5 Whys of...

5 Whys of Multiprocessing: https://guitarvydas.github.io/2020/12/10/5-Whys-of-Multiprocessing.html

5 Whys of Full Preemption: https://guitarvydas.github.io/2020/12/10/5-Whys-of-Full-Preemption.html

5 Whys of Software Components: https://guitarvydas.github.io/2020/12/10/5-Whys-of-Software-Components.html

Appendix - Incremental Learning

A debugger can be used to observe the operation of someone else's code (or your own code).

- Stepping through code and interactively examining data structures is one way to understand the intended architecture.

- Fixing other peoples' mistakes can force you to think deeply about the code and data structure details. Incrementally, not in one big gulp.

Appendix - Details Kill

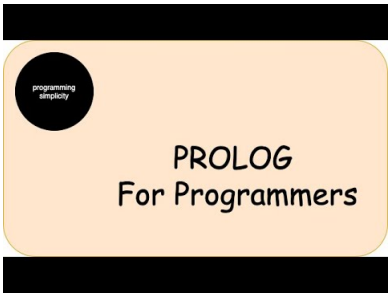
https://guitarvydas.github.io/2021/03/17/Details-Kill.html

elide details

- don't delete details, suppress them
- KISS
  - simplicity is the "lack of nuance"
  - complexity is the inclusion of too many details (in any one layer)

PROLOG for Programmers Introduction (in PROLOG)

video



PROLOG for Programmers[1]

Slides

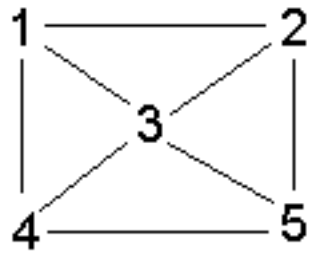
using: https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_-15.html

---

Declarative <== Relational

PROLOG == Relational

miniKanren == Relational



figure

factbase

```
edge(1,2).
edge(1,4).
edge(1,3).
edge(2,3).
edge(2,5).
edge(3,4).
edge(3,5).
edge(4,5).
```

basic relation

```
connected(X,Y) :- edge(X,Y).
connected(X,Y) :- edge(Y,X).
```

alternate basic relation

```
connected(X,Y) :- edge(X,Y) ; edge(Y,X).
```

---

inferring connections

Divide and Conquer

---

do-it style

---

top level

```
path(A,B,Path) :- BasicPath = [A], inferPath(A,B,BasicPat\
h,Path).
```

---

base case

```
inferPath(A,B,P,ResultPath) :- connected(A,B), ResultPath\
 = [B|P].
```

---

recursive case

```
inferPath(A,B,PriorPath,ResultPath) :-
       connected(A,C),
       C \== B,
       \+member(C,PriorPath),
       NewPath = [ C | PriorPath ],
       inferPath(C,B,NewPath,ResultPath).
```

PROLOGify - Better Relations

---

top level

```
path(A,B,Path) :- BasicPath = [A], inferPath(A,B,BasicPat\
h,Path).
```

->

```
path(A,B,Path) :- inferPath(A,B,[A],Path).
```

---

base case

```
inferPath(A,B,P,ResultPath) :- connected(A,B), ResultPath\
 = [B|P].
```

->

```
inferPath(A,B,P,[B|P]) :- connected(A,B).
```

---

recursive case

```
inferPath(A,B,PriorPath,ResultPath) :-
       connected(A,C),
       C \== B,
       \+member(C,PriorPath),
       NewPath = [ C | PriorPath ],
       inferPath(C,B,NewPath,ResultPath).
```

->

```
inferPath(A,B,Visited,Path) :-
       connected(A,C),
       C \== B,
       \+member(C,Visited),
       inferPath(C,B,[C|Visited],Path).
```

SWIPL at command line

```
swipl
  ?- consult(path).
  ?- path(5,1,R).
  ?- halt.
```

SWIPL in Bash script

```
#!/bin/bash
swipl -q \
    -g 'consult(path)' \
    -g 'use_module(library(http/json))' \
    -g 'bagof(R,path(5,1,R),B),write(B),nl.' \
    -g 'halt'
```

SWIPL with JSON

```
#!/bin/bash
  swipl -q \
      -g 'consult(path)' \
      -g 'use_module(library(http/json))' \
      -g 'bagof(R,path(5,1,R),B),json_write(user_output,B\
),nl.' \
      -g 'halt'
```

Transcript

I am gonna do a quick introduction of PROLOG for programmers using JR Fisher's tutorial. It's on this webpage here. One thing to notice is that we strive to do declarative programming. One form of declarative is relational programming. PROLOG was one of the first attempts at relational programming.

Mini kran is the current manifestation of relational program. I happen to be using eax org mode, and I'm gonna probably use a sw s w I PROLOG, although I've used G PROLOG a lot also. Let's have a look at the figure. It's Listed in that article it's basically five nodes forming a network, and there's connections between the nodes in, the various ways that are shown on this diagram.

The diagram shown can be described as a PROLOG da fact base. In PROLOG, we would say that there's an edge from one to two. The period acts like a semicolon in many languages. These are all relations. They're basically just straight out con fact constants, and if you look through the rest of it, you can get the idea of what's going on here.

If we look at the fact base, we see that there's an edge from one to two, but it doesn't say that two is connected to one. We can add a basic relation that, that shows this relationship that two nodes are connected. If there's an edge between X and Y or if there's an edge between y and.

In PROLOG, we can write this basic relation a different way, which is to use a semicolon operator. So basically we're saying connected is edge of X, Y or, or. edge of Yx in PROLOG, it is conventional to put all the or terms separated as, separate relations or in this form where, there's a semicolon between them and type relations are done with commas and we'll see that.

We can infer further connections by adding PROLOG rules, using guess what? Divide and conquer. There's the programmer, do it kind of style and the pro. Then we can prolog, gify it and turn the relations, the code into better relations.

Any attempt to solve this problem breaks down into two main problems and one envelope I call the envelope around the whole thing, the top level, there's a base case that says how we know when we're finished and the recursive case.

We'll start looking at the top level case, basically having a relation that signifies the path between A and B. And we write the path into a result called path. Fundamentally, every path starts out with square brackets in PROLOG mean list. So we the basic path always has a in it since that's where we start.

And then we infer new paths between A and B. As we go.

If we look at the base case it says that I, I. The, most basic level. If

there's something that satisfies the connection between A and B, then we're done and we just glue the result B onto the path. This is the way you write list cons and B onto the front of B, and then we're putting it into the variable called result path in the recursive case.

We look for connections between A and B. We keep a prior path around and we keep the result path. We invent a, new node and we call it c. And we look for a connection between if, A and C are connected and C is not equal to the final result, B, that we want and C is not a member of the prior path, then we cons C onto the prior path, creating a new path, and then we recur.

We try to find connections between c and b on the, and we produce a resulting path and we keep a prior path around. Now we'll look at how to write these relations in better PROLOG form. We'll look at the top level, the base case, and the recursive case separately. Now looking at the top level in PROLOG form, we take the do it style and rewrite it in PROLOG this way.

Programmers are used to writing a function signature, followed by. How to get there, how to do it. Whereas in PROLOG, everything is a relation, and that's not actually a function signature. It's just a shorthand for whatever is on the right hand side. So the left hand side really is just a, is a truism.

It says when the PROLOG run is finished, then this will be true. And here's how we, and here's how we infer that things are true in this case. What we, don't need a Temporary variable that assigns the list of containing only a into a temporary variable called Basic Path. We can just stick it directly into the relation and get rid of this.

In fact, the equal business in, PROLOG needs a little bit more of looking at. I won't dig into it, but try not to use it.

Likewise, the base case in PROLOG folds our original code, which kind of looked like this, with an equal sign into a relation that says everything in one fell swoop that that we can infer the path from

A to B and the result is b. Associate B put onto the result path, resulting path or put on, put in the resulting path is the previous path with the result.

Put B put onto it only if A and B are connected. again, this doesn't look like what programmers are used to because we're, showing things that programmers think of as do it code causing things on the left hand side. In fact, this is still a relationship that shows that infer path is true and we had a prior path and that the, the.

thing we called the resulting path is broken up into a head and a tail. A head where the head is the, new, the node and the tail is the previous path. And when we continue writing it this way, we can actually do more, more interesting things. Like we don't actually have to supply A and B.

We can ask given a. what are the po given a path and a prior path? What were the previous, what were the A and Bs that satisfied that relationship? Likewise, the recursive case of Chronification takes the the old style and writes it as a single relation that's recursive. And here we've gotten rid of.

Temporary variables like this. I've in this particular case, I took prior path and renamed it to visited which makes more sense to me in the final result. And we've dropped from five lines to four of again, this side is the abbreviation for these four lines. So if, we can show that these four lines are true, then this relation.

Is true. and PROLOG will find all of the matches for that relation. And in PROLOG, what we'll see is we can actually run some of this code in PROLOG. And we will see that it backtracks over. The code gives us one solution. If we give it a semicolon, it gives us another solution. If we give it another semi and keep doing that until it runs out of solutions, then it stops giving us solutions.

I've taken all of the code we've written so far and the facts and put them into a single file called Path pl. The the extensions, pl and pl are the, ones favored by the two PROLOGs that I use. S SWI PROLOG, and G PROLOG. Now, I'll take this. PROLOG file and

load it into PROLOG. So we'll switch to the terminal, we'll run the SW PROLOG and to load a file, I use the command called consult, and we don't need to give it a path name.

We do have to remember the dot. And that loads the.

We can qu do simple queries on the fact base. For example, we can ask to see if edge, if there's an edge between one and two. And it, and now I'm gonna, it says true. There was, I'm gonna put in a semicolon and then it'll say there are no more edges between one and. I can try an edge between two and one, and that says there isn't any.

There aren't any.

Now we can use the Most basic query that we had, the basic relation, which is connected two and one, and that returns true, you can say connected one and two, which should be true, and it is. Now we can try something more interesting. We can ask for what is the path between one five and we'll put the result in something called p.

And it says that you can get there going from five to one, to two to one, or from five to three to two to one, or five to four to three to two to one, five to four to one, five to three, to four to one, five to 3, 4, 1, 5, 3, 1, 5, 4, 3, 1, 5, 2, 3, 1. And that's,

Note that in PROLOG variables, logic variables are always capitalized. Everything else is lowercase. So if I say path one comma five, big R result, then it has no problem finding it. If I say path one, Lower case result, then it fails immediately. I can also put some of these commands into bash scripts.

I'll halt is the way to quit PROLOG. I go up here and I've I think I've created a file called G Path for running, g PROLOG. If I run that, Slash g then it'll load the, it'll consult the file path pl and then it'll, and then it'll leave me ready to do a query so I can do one five capital R, and then it'll give me the results halt.

I can do the same for SW PROLOG. I've created a bash file called

SW Path Bash, and it it's got a slightly different command line syntax, and we'll try running it. Change this slash. And it leaves me ready to do a query. It's, consulted the file called Path pl and it relieves me ready to do a query like that. And same thing. Halt period. Halt is the way you finish. And then you all have to remember the period.

Swipe PROLOG has the minus Q modifier, so I'll say minus q. I'll do a back slash, which is shell syntax, to continue a line and modify the file. This way. I then run it again, and there it is, ready for me to put in a query.

You can even put -l halt this again, halt. You can put the query in as a further command line argument minus G path five one big R and we can try that. Guess what? It ran the query, but it didn't show us the. Now we'll retry that with rewriting the query to have an output. So we're gonna collect all of the results are of pa of the query path R into a bag called B.

Then we're gonna write b. onto, stood out, and then we're gonna c cause a new line. And if I got that we should be able to run it. There it is. And I forgot to put the, I can put a halt at the end of that like this minus g halt. Let's try that. And that's the whole query and it finished.

Things get more interesting. If we write out the facts as JSON instead of PROLOG fact base format, I'll make the modifications to the shell script. Here we consult path.pl. We, I've added the line that uses the module called JSON. Then we do the bag of, and instead of calling we call JSON Wright.

This is user. This is stood out, user output in Prologis the bag and the N -l. We've saved it and then we try it again, and there we are. It's output is JSON.

The Holy Grail of Software Development

Video

video[1]

Transcript

The holy grail of software development. What is the holy grail? It's simple. We wanna build a piece of software and forget how it works. We just want to use it. We want to design it, bench, test it, then ship it with no further bugs in the field. This is possible. And digital hardware design, you could design, debug a design on a bench, ship it, and we would have zero bugs in the.

even though ICS that make up a electronic circuit are all asynchronous they're, what we call multitasking telecom. In the 1980s at least, had something called the four nines culture. They guaranteed a 99.99% uptime. It. It was so ubiquitous that people would call each

other during a power failure on a telephone and say, is the power out at your, where you.

And they wouldn't even notice that the phone was still working even though there was power failure. So in software, we have many bugs in the field. We have monthly, weekly, daily updates. We have something that I'm calling GitHub culture, where somebody looks at a repo and says it can't be any good if there weren't any recent pushes to the repo.

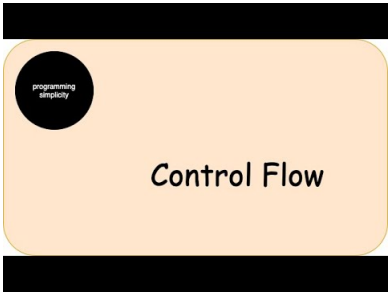
That's the wrong emphasis. Working code should be perfect. It should need no new modific. Sam Aaron, for example, teaches 10 year olds how to create multitasking programs. What's our problem? What's the difference between software and hardware development? That's a rhetorical question. I think it boils down to the use of global variables.

I think that all current languages use a hidden global variable that includes functional programming. And most CPU architectures the stack is a global variable.

Having that global variable hidden from view has been causing this accidental complexity. Call and return instructions provided by the hardware implicitly use the stack and it's a global variable. There's only one copy of it. We, talk. Things like thread safety, and we have this fiction that multitasking is hard, and all of that stuff is caused by the fact that we use a global variable called the stack.

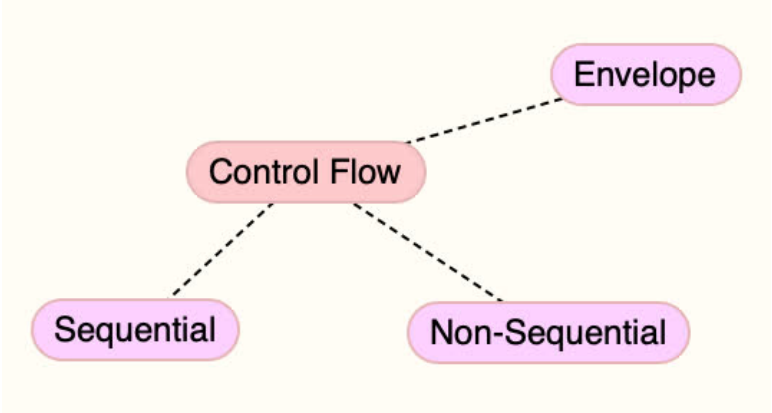
Control Flow

Video

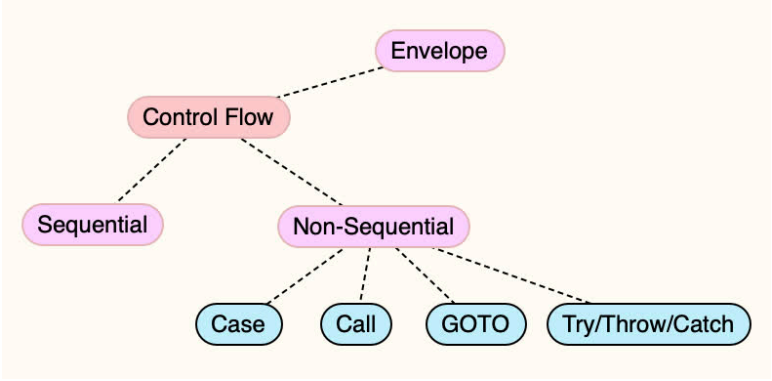


Control Flow video[1]

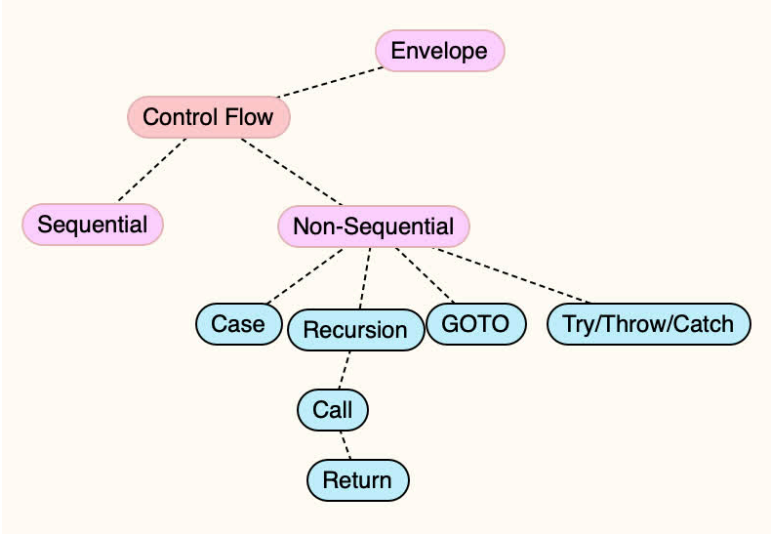
Slides



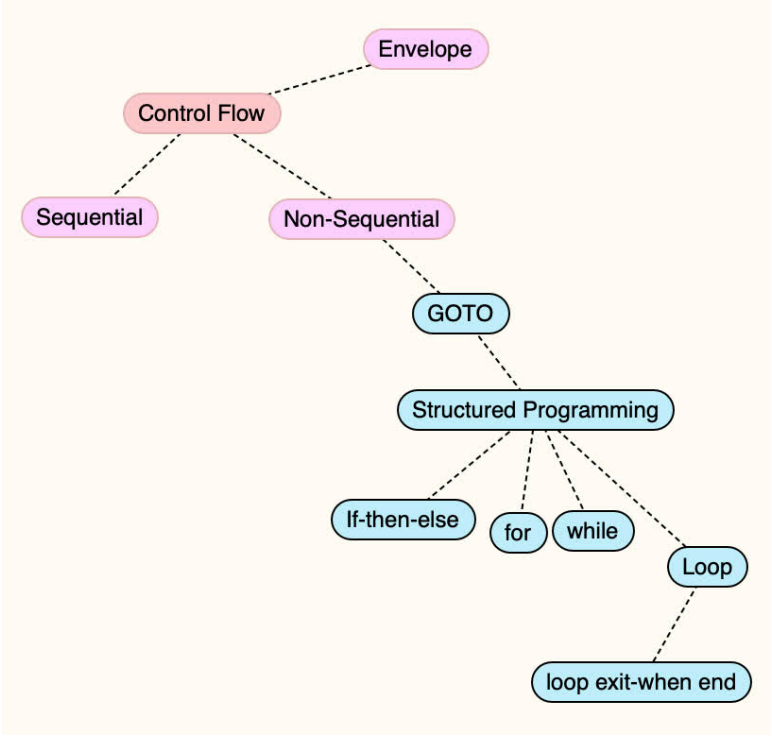
Control Flow 1



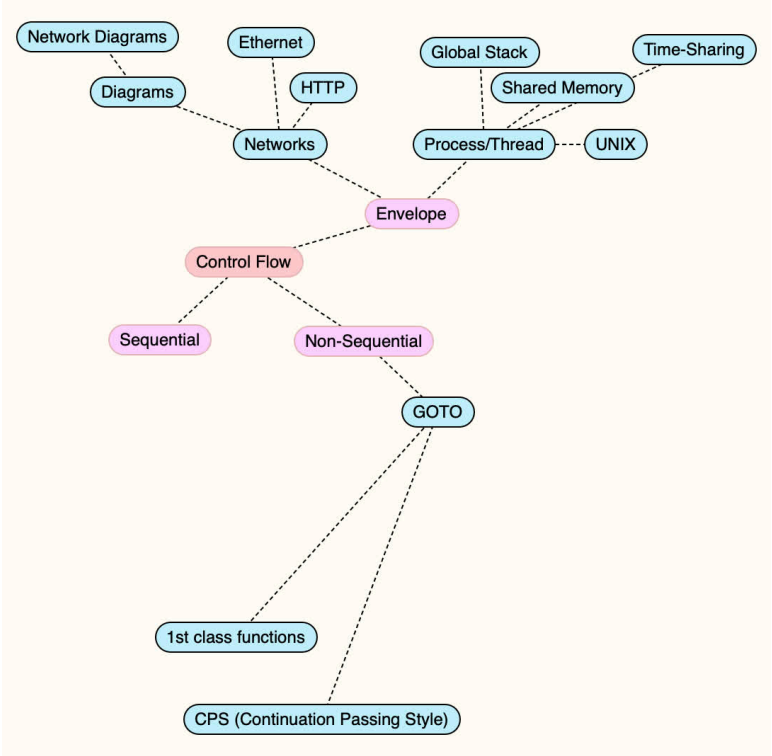
Control Flow 2



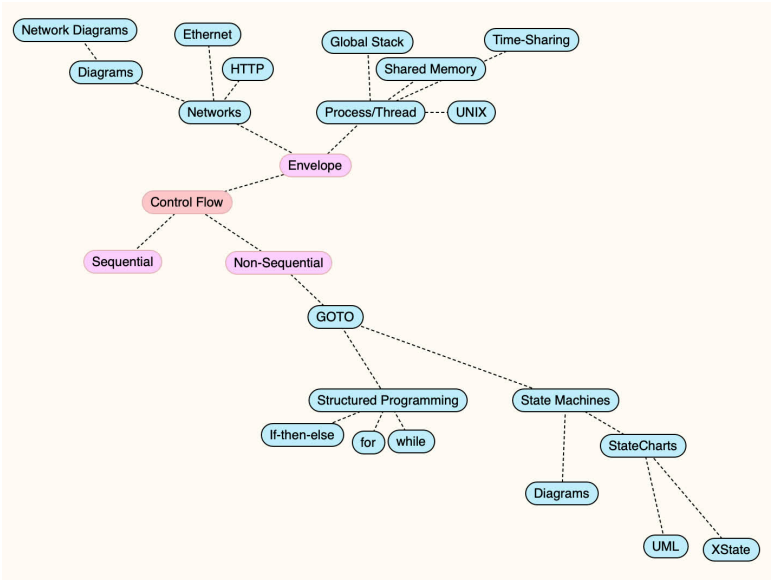
Control Flow 3



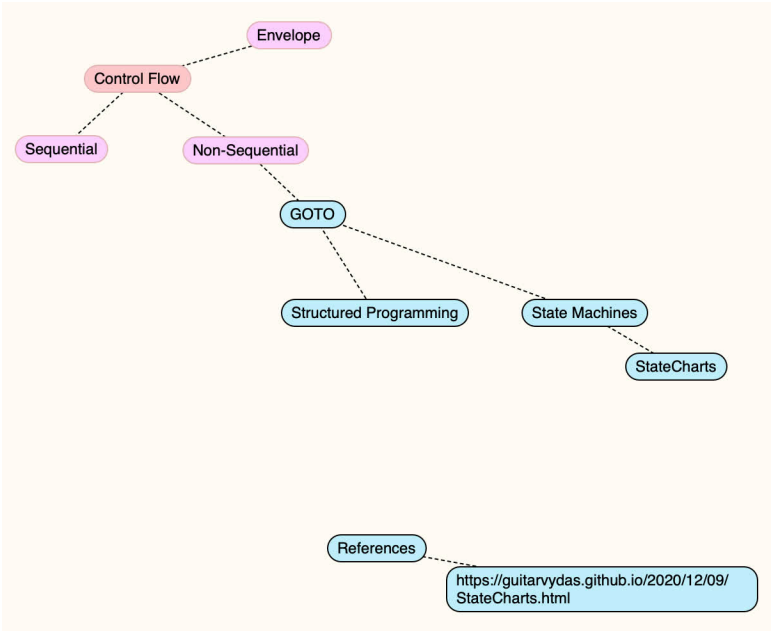
Control Flow 4



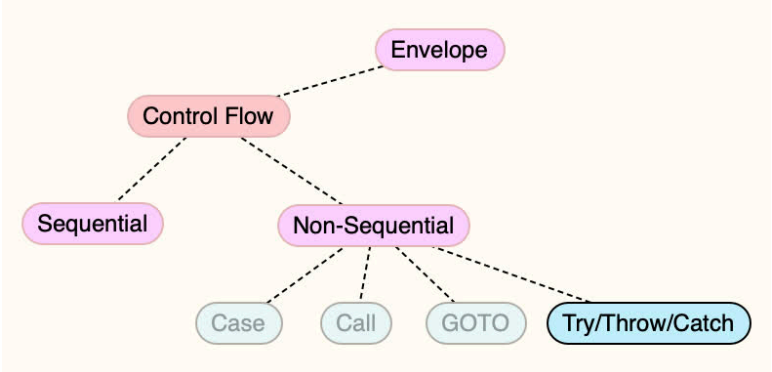
Control Flow 5



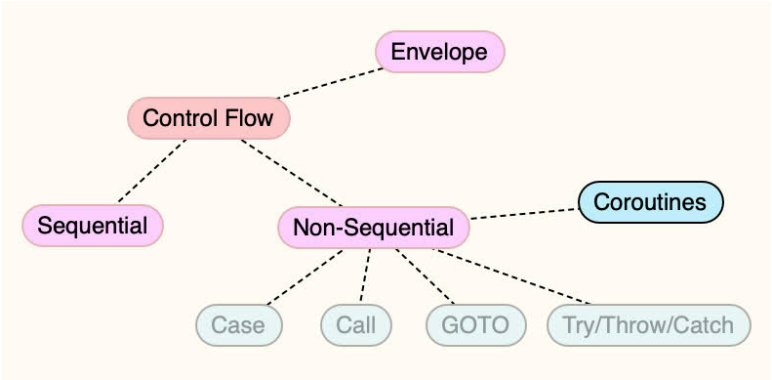
Control Flow 6



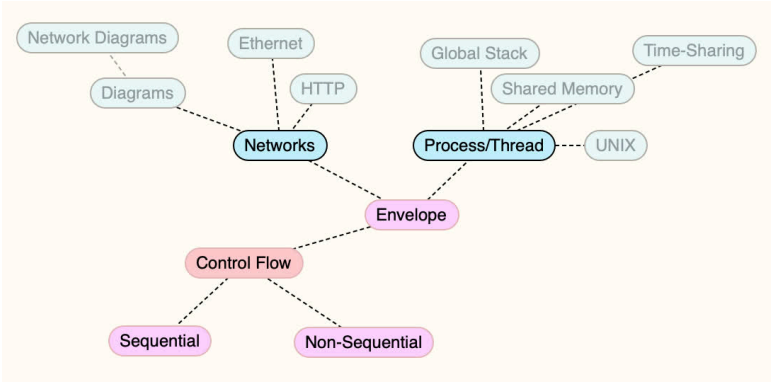
Control Flow 7



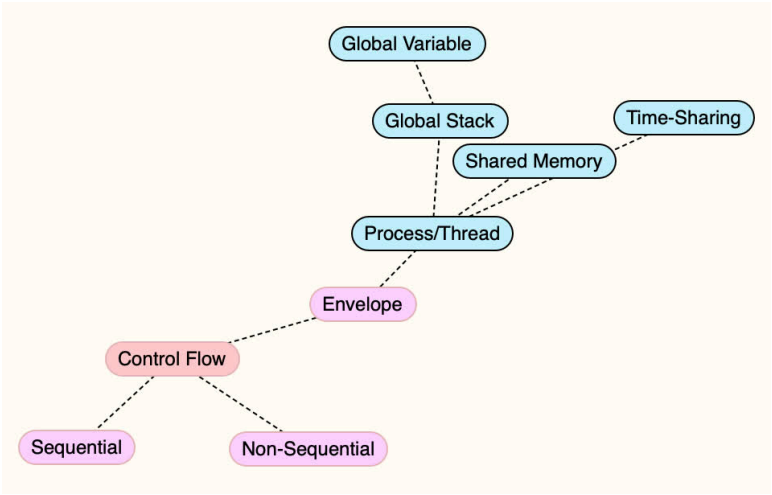
Control Flow 8



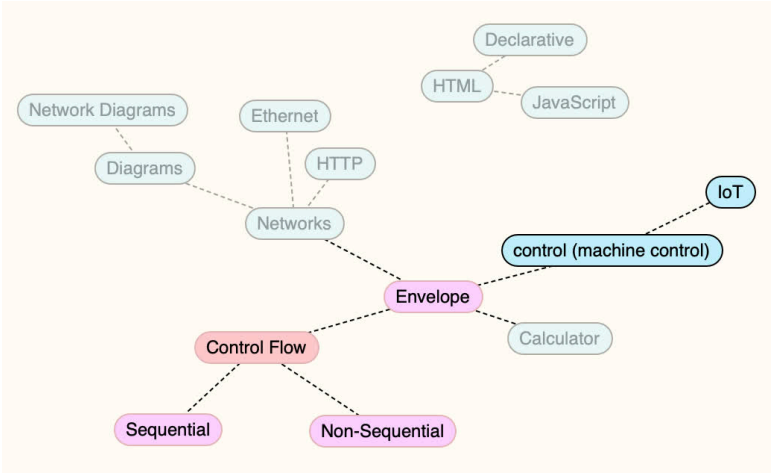
Control Flow 9



Control Flow 10



Control Flow 11



Control Flow 12

Transcript

I'm gonna talk about control flow. This is a map of where we're going. I'm gonna simplify it. I break control flow up into three main categories, sequential, non sequential, and then envelope or what we call multitasking these days.

- Sequential is where one action happens after another.
- Non-sequential control flow is where actions happen in an order that is different from one after another.
- Envelope is a non-sequential programming, but I think of it as being, higher level than non-sequential. Non-sequential breaks down into four main cases. Case call go to try throw and.

Call includes recursion and return. Some would say that, recursion is in fact the kind of, or is the, the uber parent of calling. And so I'll try to do that if I can. Here, there we go. Go to is the main way to change control flow in a, standard program. Go-to is not a problem, but unrestricted use of go-to is a problem.

Very early on we realized that, doing go-tos all over the place was a bad idea. We invented something called structured programming and that broke down into if then else for and while, and then a language called concurrent Euclid, in used something called a loop, which subs. Both for and while into one construct called the Loop Exit, when and end.

It might be noted that first class functions are really just a kind of go-to, which is even more unrestricted than the original go-to that we had. Continuation passing style comes from first class functions, and it is just the ultimate go-to that is available to us if we stop treating, go-to as something to be shunned and as a useful but very low level operation.

We can branch structured programming into other kinds of ways of, structured program. gave us only one way to constrain the use of go-tos. There are different ways, another way to constrain go-tos is by using state machines, they give much more flexible control flow const, or they give different flexible control flow constructs.

Structured programming gave us one way to constrain go-tos. There are different ways. . There are different ways to constrain go-tos than just using structured programming. One is state machines. Another one we'll see later is just, using an envelope. State machines are interesting, in that. In 1986, Harel wrote a paper called, about StateCharts.

I discuss state charts in my blog at this address. Try, throw and catch is a bag added onto the side of common programming languages to handle non-sequential, dynamic control flow chains. Co routines, albeit useful, have been left in the dust by most of our current programming languages.

Now we'll look at the, thing that I call an envelope, which breaks down into processes that are threads and networks.

looking at processes, we notice a, that they involve shared memory. They use a global variable called the stack, and they involve time sharing. All of these issues are issues based on early forms of computing. Back in the 1950s, CPUs and memory were expensive. These things have given us accidental complexities.

Now let's look at networks. This is the wave of the future of computing in my mind.

Networks break down at http and ethernet. Mostly, in networks. We also can draw sensible diagrams of things, although we haven't formalized it. Networking has, created a new breed of language. We call that language, HTML.

It is, declarative and it's not stack based, so it can, go across distributed processors. There are parts of HTML that we haven't figured out how to make declarative. So we use JavaScript and other

things, to help us, create program more interesting programs using HTML up to now we've been, trying to, shoehorn everything into a calculator style style of programming, embodied by the original use of computers, which was to, perform calculation ballistic calculations. A new form of computing, or a different form of computing is, distributed control flow or machine control. Tends to break the types of things we can do with current programming languages.

The most, distributed processing and distributed control is gonna be where IoT ends up, inform in Internet of Things. . Right now we're using, old fashioned techniques like, dropping this enormous library called Linux on top of small processors, when we could be shaving, these programs to, suit their purpose.

For example, controlling a refrigerator does not require all of Linux.