2024-04-01 Free Range Programming

$PATH

Shells should provide two $PATH-like environment variables for *each* project

1. The working directory for the project

2. A list of paths to libraries and tools used by the project.

We currently have ad-hoc collections of zillions of environment variables, and, we have a disjoint thing called the “cwd”. Options and anti-normalization (disjointedness) cause bloat, and, worse, bugs due to the possibility that every edge-case has not been explicitly addressed.

Here, extreme explicitness would help clarify what is intended.

0D arrows - explicit control flow, like DS

$PATH - once per terminal, should be once per project

Lisp is Assembler

Assembler is assembler with a line-oriented syntax.

Lisp is assembler with a recursive syntax.

Lisp is a Language for Programming With ASTs (Parse Trees)

Simple parse tree:

```
(Node left right)
```

Lists are parse trees.

Lisp encodes all source code as lists in prefix-notation. The operator always comes first, then the args. Usually, we think of the operator as “the function”. I’m just using different words for exactly the same construct.

A Lisp program is a parse tree, where the first item is a “node” and the args are branches below the node.

Lisp syntax allows nodes to be recursive. Parse trees contain nodes that are recursive.

Lisp is a program written as a parse tree. A Lisp program skips over the “syntax” bit and expresses the parse tree directly.

Non-lispers hate programming in Lisp because they prefer to have a machine convert “syntax” into parse trees. Lispers don’t mind programming in parse trees.

To be accurate, the parse tree is actually a CST, not an AST. People tend to use the term AST as an umbrella that includes CSTs.

ASTs are Abstract Syntax Trees.

CSTs are Concrete Syntax Trees.

ASTs encode all of the possibilities that can occur in a language.

CSTs encode only the parse tree of a specific program, shedding all of the other allowable possibilities. CSTs are culled subsets of the full-blown ASTs.

ASTs are used to specify languages at the front end of compilers.

When compilers parse actual programs, they convert the programs into CSTs, by matching the input against the generalized specification given as an AST. The rest of the compilation process is applied to CSTs.

When you write a compiler, you specify *all* of the possibilities in the language using an AST. Then, you hang bits of code onto each node of the AST, telling the compiler what to do when it sees something that matches the node.

When you actually compile a program with a compiler, not all of the semantic nodes in the AST fire. Only the semantic nodes that match up with the input program get fired. The nodes that do fire constitute the CST. If you were to interrupt the compiler and get it to output the current version of the input program as a tree, you would get a CST, since not all of the nodes in the full-blown AST apply to the given input.

Most compilers are written using full-blown ASTs and you never get to see the internal working CSTs.

Unless, of course, you are using Lisp, in which case you become the wetware version the syntax-parsing phase of the compiler and you write the CST down as Lisp “code”.

Projectional Editors

It seems to me that projectional editors could use Lisp as their common internal representation. One of the biggest tasks in projectional editor work is coming up with an internal representation of the parse tree. We already know how to do that - just write Lisp.

The other big task in projectional editors is mapping syntax onto parse trees and v.v. mapping parse trees back into syntax.

It seems to me that such syntactical mapping can be facilitated, or at least started, with PEG parsers. My favourite PEG language is OhmJS.

Mapping “syntax” onto “Lisp” is easier with OhmJS.

Mapping “Lisp” back onto “syntax” is less easy, but do-able. I’ve created a Scheme to Javascript transpiler using OhmJS. My diary of doing this is in https://guitarvydas.github.io/2020/12/09/OhmInSmallSteps.html. It helps to be able to target a language that has anonymous functions / closures. Most modern languages now have closures, so this isn’t such a big problem. The less “syntax” in the target language, the better.

Yet, I’ve managed to transpile to Python, possibly the worst target language of them all for this kind of task, due to Python’s non-recursive, indentation-based syntax. The trick is to transpile into a pseudo-Python that has a context-free, recursive syntax. Then, clean up and convert the pseudo-Python to real Python.

The most recent version of the Python indenter appears to be https://github.com/guitarvydas/eh/blob/master/indenter.js.

In this version, I encode magic brackets as “(-“ and “-)”, then delete them for non-Python target languages and calculate indentation when the target is Python.

Today, two years later, I would probably use Unicode characters for the brackets and leave all ASCII characters otherwise open for use in the target languages.

The idea of using “(-“ and “-)” was that the lisp pretty printer in Emacs could be used to indent generated code for eye-balling during development.

Holm's Prolog in Common Lisp

I ported Nils Holm's Prolog in Scheme to Common Lisp https://github.com/guitarvydas/cl-holm-prolog.

The original Scheme source is in http://www.t3x.org/bits/prolog6.html.

I found Holm's code and presentation to be easy to understand. In the past, I dug into On Lisp, PAIP, and, the WAM tutorial https://github.com/a-yiorgos/wambook?tab=readme-ov-file. I found Holm's discussion easiest to comprehend.

In fact, I started building a WAM myself but veered off onto some other project before fully finishing the WAM. It was beginning to work, but wasn't completely tested and debugged.The repo for the not-fully-tested, WIP WAM is https://github.com/guitarvydas/wam/tree/master.

Holm's Prolog in Javascript

I ported - automatically - Nils Holm's Prolog from Scheme to Javascript.

I don't use it much, since I just use SWIPL when I want to do exhaustive pattern matching.

The repo for the JS version appears to be in https://github.com/guitarvydas/js-prolog.

The port was done using OhmJS, as mentioned above.

Science and the Scientific Method

The scientific method is meant to be a *fail-fast* technology.

You ruthlessly attack a theory and nip it in the bud before it has a chance to spread.

Scientists don't create experiments to "support" a theory, instead, they create experiments to tear down a theory.

You can't *prove* a theory. You can only devise killer experiments that disprove a theory. A single data-point can disprove a theory, while any number of data-points cannot *prove* a theory.

The Michelson-Morley experiment is an example of good science. It disproved the then-current theory of the ether. The experiment didn't disprove the existence of an ether, it only disproved that particular explanation of how ether works. Strangely, it seems that the Michelson-Morley experiment has been misconstrued to mean that no kind of ether exists. The experiment proved no such thing. The Michelson-Morley experiment simply disproved one theory of ether, not the whole concept of ether. Apparently, even Einstein believed in the existence of ether, but, he used a different word for it - he called ether "space".

Simplifying Assumptions and Maxwell's Equations

Physicists learn, at an early age, to reduce a search space by applying "simplifying assumptions".

Applying simplifying assumptions makes it possible to deep-dive into a single aspect of a phenomena and to understand it in depth. Hopefully, after understanding the details of an *aspect* of the phenomenon, a simple explanation can be given.

Maxwell's Equations are an example of a useful set of simplifying assumptions. The equations ignore niggly details about the phenomenon of electricity and create a "simpler" explanation of a *slice* of the phenomenon.

That *slice* of electrical phenomena allows us to build useful gadgets using electronics. On the other hand, deep-thinkers like Robert Distinti are showing that Maxwell's Equations don't deal with all of the actual niggly details of the electrical phenomenon, and, that re-introducing those details into our equations, can lead to a deeper understanding of physics, itself.

A "simplifying assumption" is the idea of tossing out niggly details while considering some aspect of a phenomenon. In physics, a simplification is considered valid if its effects swamp out the effects of the niggly details, by an order of magnitude. This is written as "X >> Y", meaning that the effect of X is 10x more important than the effect of Y - for exploring some aspect of the phenomenon. If "X >> Y" does not hold, then the simplifying assumption is invalid, and the exploration needs to be re-thought. Usually, one can chip away at a phenomenon by finding multiple simplifying assumptions that are valid, and, describing aspects of the phenomenon is detail, while remembering that simplifications have been made. This is, also, called "divide and conquer".

In software, we see a very similar thing going on, but, we also see that simplified ideas are accepted as reality, instead of being thought of as only being *slices* of understanding.

For example, FP - Functional Programming - is a good simplifying assumption, for creating calculators, like complex ballistics calculators for the military. FP, though, is not convenient for describing the programming of computers for sequencing

applications, eg. iMovie, mouse handling, GUIs, robotics, blockchain, etc. This doesn't mean that FP is bad, it only means that the simplifying assumptions are invalid, and, that some other notation - programming language - should be used along with FP.

FP doesn't describe programming, it only describes a *slice* of programming.

FP uses the simplifying assumption of ignoring *time*. The idea of ignoring time is OK for time-less concepts, like compute-ing results, but, does not apply - conveniently - to other concepts, like sequencing things (timestamps, timeouts, state machines, etc.).

So, how do you handle these other kinds of concepts, like sequencing? You invent other simplifying assumptions and invent notations based on those simplifying assumptions. Can these new simplifying assumptions be used for expressing compute-ations? Nope, they are not as convenient to use as the FP simplifying assumption for such purposes. You need to use *multiple* notations. Trying to force *all* simplifying assumptions into a single notation - programming language - cannot possibly work, in the end. The result is a watered-down version of one, or of all, concepts.

The idea of building "general purpose" programming languages is basically misguided. You need multiple "special purpose" programming languages. We see this beginning to happen with concepts like DSLs, but, we can do even better.

We were held back by the realities of primitive computer hardware in the 1950s. Early machines were too expensive and too difficult to use, and, programming-language implementation was too infantile, so we needed to create unions of simplifying assumptions to reduce costs.

Today, though, reprogrammable electronic machine hardware (aka "computers") are inexpensive (Arduinos, Raspberry PIs, etc.) and memory is ridiculously abundant and cheap - we don't even bother to measure memory in terms of *bytes* and *Kb*, we think in terms of *Mb* and *Gb* regularly.

Programming languages, though, that we use today are all based on 1950s biases. We need programming languages that address, in new ways, the new realities of our current hardware.

In 1950, it was a stretch to think about anything but tiny bitmaps arranged in tight, non-overlapping grids (i.e. characters and ASCII and EBCDIC). Today's hardware can handle scalable graphics and multi-byte encodings of fundamental atomic elements (Unicode, HTML, etc.).

Thinking in terms of tiny, non-overlapping grids of bitmaps brought us CFGs (Context-Free Grammars). Now, we have PEGs (Parsing Expression Grammars)[1].

We haven't yet really opened the Pandora's Box that PEGs can bring to us.

In the early days of computing, there was a concept called "syntax directed translation". Today, there is a concept called "pattern matching". They are, essentially, the same idea.

Instead of "if-then-else", use a parser to determine control-flow. It used to be difficult to build parsers, but, today, with PEGs it's quite easy to build parsers. Software components can "talk" to each other using little-DSLs[2]. The little DSLs need only be machine readable, and not be constrained by the restrictions of human readability. And, we can have zillions of little DSLs, essentially one unique little-DSL for each software component.

Has this idea been tried before? Yes. PT-Pascal was built this way. Concurrent Euclid was built this way[3]. Even *gcc* uses this idea - it compiles to a little-DSL called "RTL" - a "virtual machine syntax". UNIX® pipelines are based on this idea, although the "little-DSL"s are usually constrained to be lines of text separated by a magic character (newline).

In fact, REGEX is a little-DSL designed for matching characters. Its syntax leaves a lot to be desired, but, it is a little-DSL, nonetheless. And, it is a little-DSL that co-exists, syntactically, with other programming languages.

---

[1] In fact, we've always had PEGs, but used a different name for them - "recursive descent". PEG technology makes this kind of thing much more accessible and convenient to use.

[2] I use the name "SCN" instead of "little-DSLs:. Solution Centric Notation. I will continue to use the term "little-DSL" in this essay, though.

[3] I strongly suspect that the languages Turing and Turing+ were built this way, but, I don't have direct experience with their implementations.

We've been using little-DSLs without realizing it. GDB has a little-DSL built into it directed at the needs of debugging.

The C preprocessor is a little-DSL unto itself that deals with textual substitution of preprocessor code into C code. The resulting C program is machine-readable, but, not very human-readable, which is OK, since few humans actually bother to look at the output of the C preprocessor.

We think in terms of "composition" of software components at runtime or compile-time. PEG allows us to think in terms of "syntactic composition".

Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*