Contents

Terms Used

common languages typical languages programmers use to write commercial software, (e.g., C, C++, Java, Javascript, Python, Ruby, Go, Perl, PHP, etc.). 13

pwn The act of dominating a person, place, or thing. (...or piece of code). 29

scat the excrement of an animal including but not limited to human; also heroin. 7

Technical Terms Used

Preface

The way we design and write software to do computation and AI today sucks. It's a vat of boiling poop, mixed with pee, slowly swirling and bubbling toward that dehydrated semi-solid state of goop that serves to repel and repulse most normal people only attracting the few unfortunate-fortunate folks that happen to be obsessed with scat.

Hrm, too much? Probably. I guess you'd expect me to use concrete examples and cite evidence to make my points, me being a professor and all. I mean, I could write something like *"The fundamental imperative programming model utilized in near all of the production software produced in the last four decades has not changed since blah blah blah..."* to meet expections. I'd certainly sound more credible and perhaps super smart. Well, I'm not going to do that here. Let's have fun. Afterall, Jaseci has never been work for me, its play. Very ambitious play granted, but play at it's core.

Everything here is based on my opinion and intution. That suffices for me, and I hope it does for you. I have spent many decades coding and leading teams who code, but its my gut that tells me that we can do better. This book describes my attempt at better. I hope you find value in it. If you do, awesome! If you don't, also awesome.

Chapter 1

Introduction

Chapter 2

What and Why is Jaseci?

2.1 Viewing the Problem Landscape Spacially

2.2 Compute via The Collective, The Worker Bee Model

Chapter 3

Abstrations of Jaseci

3.1 Graphs

There's something quite strange that has happend with our common languages over the years, ...decades. When you look at it, almost every data structure we programmers use to solve problems can be modeled formally as a graph, or a special case of a graph, (save perhaps hash tables). However no common language ustilizes the formal semantics of a graph as its first order abstraction for data or memory. I mean think about it, isn't it a bit odd that practically every data structure covered in the language-agnostic classic foundational work *Introduction to Algorithms* [1] can most naturally be be reasoned about as a graph, yet none of the common languages have built in and be designed around this primitive. I submit that the graph semantic is stupidly rich, very nice for us humans to reason about, and, most importantly for the purpose of Jaseci, is inherently well suited for a spacial conceptualization and reasoning about computational problems.

There are a few arguments that may pop into one's mind at this point of my conjecture.

- "Well there are graph libraries in my favorite language that implement graph symantics, why would I need a language to force the concept upon me?" or
- "Duh! Interacting wi th all data and memory through graphical abstractions will make the language slow as hell since memory in hardware is essitially a big array, what is this dude talking about!?!?"

For the former question, I counter with two points. First, the core design of a language will be based upon its inherent abstractions, and with graphs not being one such abstraction the language design isn't optimized to empower

programmers to nimbly do fancy gymnastics with the rich symantics of graphs. And second, libraries suck (See A.1).

For the latter question, I'd respond, "Have you SEEN the kind of abstractions in modern languages!?!? It's rediculous, lets look at python dictionaries, actually scratch that, lets keep it simple and look at dynamic typing in general. The runtime complexity to support dynamic typing is most certainly hgiher than a graph abstraction. Duh right back at'ya!"

3.1.1 Yes, But What Kind of Graphs

There are many categories of graphs to consider when thinking about the abstractions to add to Jaseci. There are rules to be defined as to the availabe semantics of the graphs. Should all graphs be directed graphs, should we allow the creation of undirected graphs, what about parallel edges or multigraph, are those explicitly expressible, can we express hypergraph, and what combination of these graphical sematics can be manifested and manipulated through the programming model. At this point I can feel your eyes getting droopy and your mind moving into that intermediary state between concious and sleeping, so let me cut to the answer.

In Jaseci, we elect to assume the following semantics:

1. Graphs are directed with a special case of a doubly directed edge type (which can be utilized practically as an undirected edge).

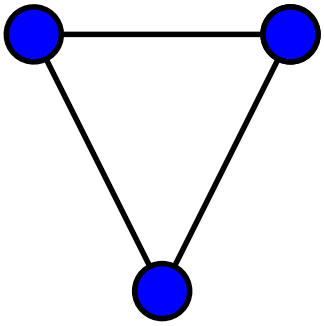
2. Both nodes and edges have their own distinct identities (i,e. an edge isn't representable as a pairing of two nodes). This point is important as both nodes and edges can have a context.

3. Multigraphs (i.e., parallel edges) are allowed, including self-loop edges.

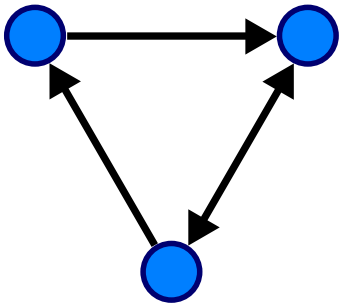
4. No hypergraphs as I wouldn't want my programmers heads to explode.

5. As an aside, I would describe Jaseci graphs as strictly directed multigraphs for which that leverages the semantics of parallel edges to create a laymans 'undirected edge' by shorthanding two directed edges pointed in opposite directions between the same two nodes.

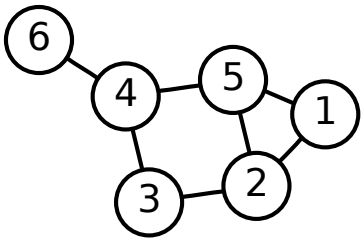
[NERD ALERT] I'd formally describe a Jaseci Graph as, [END NERD ALERT]



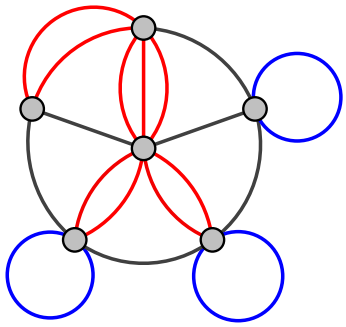
(a) 1a



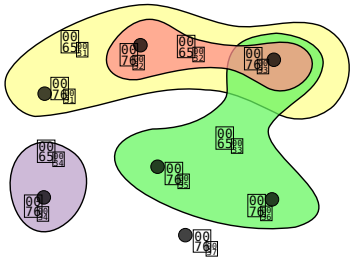
(b) 1b



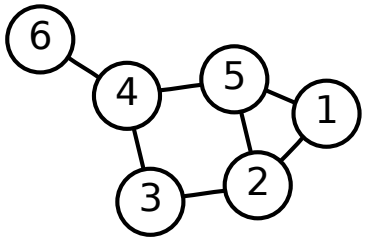
(c) 1a



(d) 1b



(e) 1a



(f) 1b

Figure 3.1: plots of....

3.1.2 Contexts

3.1.3 Nodes

3.1.4 Edges

3.2 Walkers

3.3 Abilities

3.4 Other Abstractions Not Yet Actualized

Chapter 4

Architecture of Jaseci and Jac

4.1 Anatomy of a Jaseci Application

4.2 The Jaseci Machine

4.2.1 Machine Core

4.2.2 Jaseci Cloud Server

Chapter 5

Interfacing a Jaseci Machine

5.1 JSCTL: The Jaseci Command Line Interface

5.2 Jaseci Rest API

Chapter 6

The Jac Programming Language

Chapter 7

Architecting Jaseci Core

Chapter 8

Architecting Jaseci Cloud Serving

Epilogue

Appendix A

Rants

A.1 Why Libraries Suck

Because they do.

Still need more reasons?

Well, if you dont already know, I'm not going to tell you.

Fine, I'll tell you.

1. They suck because they create dependancies for which you must have faith in the implementer of the library to maintain and keep bug free.
2. They suck because there are often at least 10 options to choose from with near exact features expressings slightly different idosyncratic ways.
3. They suck because they suck.

Don't get me wrong, we have to use libraries. I'm not saying go reimplement the wheel 15 thousand times over. But that doesn't mean they don't suck and should be avoided if possible. The best is to know your library inside and out so the moment you hit some suckitude you can pull in the library's source code into your own codebase and pwn it as your own.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.