JASECI & JAC

BIBLE

Jason Mars

v1.0

Warning: This book is a trigger factory.

If after reading that sentence you feel a sense of concern, this book WILL trigger you and you'll need to refer to the previous page and continue. If you are not concerned after reading the warning, continue with caution.

Contents

Not So Technical Terms Used

bleh mildly yucky. 50

christen to name or dedicate (something, such as a piece of code) by a ceremony that often involves breaking a bottle of champagne. 48

coder the superior human. 53

common languages typical languages programmers use to write commercial software, (e.g., C, C++, Java, Javascript, Python, Ruby, Go, Perl, PHP, etc.). 19

dope sick. 7

gobbledygook language that is meaningless or is made unintelligible by excessive use of abstruse technical terms; nonsense. 32

goo goo gaa gaa the language of babies. 49

grok to fully comprehend and understand deeply . 47

haxor leet spelling of hacker. 7, 26, 35

Jaseci jolt an insight derived from Jaseci that serves as a high voltage bolt of energy to the mind of a sharp coder.. 53

leet v. hyper-sophisticated from a coding perspective, n. a language used by leet haxors. 7, 26

pwn the act of dominating a person, place, or thing. (...or a piece of code). 76

redonkulous dope. 7

scat the excrement of an animal including but not limited to human; also heroin . 11

sick redonkulous. 7, 47

Technical Terms Used

contexts A set of key value pairings that serve as a data payload attributable to nodes and edges in Jaseci graphs. 21

directed graphs . 20

hypergraph . 20

multigraph . 20

sentinel Overseer of walkers and architype nodes and edges.. 32

undirected graphs . 20

walker An abstraction in the Jaseci machine and Jac programming language that represents a computational agent that computes and travels along nodes and edges of a graph. 53

Preface

The way we design and write software to do computation and AI today sucks. How much you ask? Hrm. . . , let me think. . . , 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 tantalized with scat.

Hrm, too much? Probably. I guess you'd expect me to use concrete examples and cite evidence to make my points, with me being a professor and all. I mean, I could write something like *"The imperative programming model utilized in near all of the production software produced in the last four decades has not fundamentally changed since blah blah blah..."* to meet expections. I'd certainly sound more credible and perhaps super smart. I have indeed grown accustomed to writing that way and boy has it gotten old. 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. . . no, *expert* opinion, and my intution. That suffices for me, and I hope it does for you. Even though I have spent many decades coding and leading coders working on the holy grail technical challenges of our time, I won't rely on that to assert my credibility. Let these ideas stand or die on thier own merit. 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, awesome!

Chapter 1

Introduction

Part I

World of Jaseci

Chapter 2

What and Why is Jaseci?

2.1 Viewing the Problem Landscape Spacially

2.2 Compute via The Collective, Ants in the Colony

Chapter 3

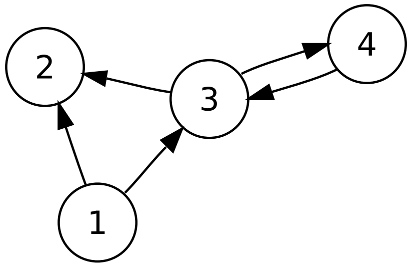
Abstrations of Jaseci

3.1 Graphs, the Friend that Never Gets Invited to the Party

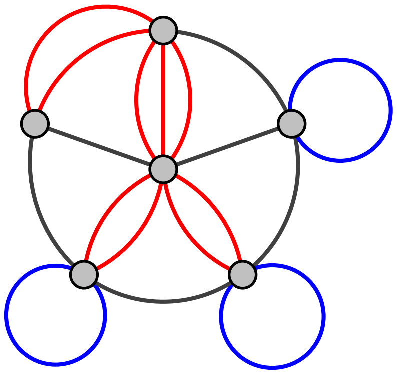
There's something quite strange that has happend with our common languages over the years, ...decades. When you look at it, almost every data structure we programmers use to solve problems can be modeled formally as a graph, or a special case of a graph, (save perhaps hash tables). Think about it, stacks, lists, queues, trees, heaps, and yes, even graphs, can be modeled with graphs. But, low and behold, no common language ustilizes the formal semantics of a graph as its first order abstraction for data or memory. I mean, isn't it a bit odd that practically every data structure covered in the language-agnostic classic foundational work *Introduction to Algorithms* [4] 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 the conceptualization and reasoning about computational problems, especially AI problems.

There are a few arguments that may pop into 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 with all data and memory through graphical abstractions will make the language sslloooowww as hell since memory in hardware is essitially a big array, what is this dude talking about!?!?"



(a) Directed graph with cycle between nodes three and four.



(b) Multigraph with parallel edges and self-loops

Figure 3.1: Examples of first order graph symantics supported by Jaseci.[1]

For the former of these two challenges, I counter with two points. First, the core design languages are always based upon their inherent abstractions. With graphs not being one such abstraction, the language's design will not be optimized to empower programmers to nimbly do gymnastics with the rich language symantics that correspond to the rich semantics graphs offer (You'll see what I mean in later chapters). 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 what would be needed to support graph symantics. 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 support in 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 or discouraged / banned, can we express hypergraph, and what combination of these graphical sematics should be able to 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.

[1]Images credits to wiki contributers [2, 3]

In Jaseci, we elect to assume the following semantics:

1. Graphs are directed (as per Figure 3.1a) with a special case of a doubly directed edge type which can be utilized practically as an undirected edge (imagine fusing the two edges between nodes 3 and 4 in the figure).
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 contexts.
3. Multigraphs (i.e., parallel edges) are allowed, including self-loop edges (as per Figure 3.1b).
4. Graphs are not required to be acyclic.
5. No hypergraphs, as I wouldn't want Jaseci programmers heads to explode.

As an aside, I would describe Jaseci graphs as strictly unstrict directed multigraphs 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 1 ***(time to let your eyes glaze over)***

I'd formally describe a Jaseci Graph as an 7-tuple $(N, E, C, s, t, c_N, c_E)$, where

1. N is the set of nodes in a graph
2. E is the set of edges in a graph
3. C is the set of all contexts
4. s: $E \to V$, maps the source node to an edge
5. t: $E \to V$, maps the target node to an edge
6. c_N: $N \to C$, maps nodes to contexts
7. c_E: $E \to C$, maps edges to contexts

An undriected edge can then be formed with a pair of edges (x, y) if three conditions are met,

1. $x, y \in E$
2. $s(x) = t(y)$, and $s(y) = t(x)$
3. $c_E(x) = c_E(y)$

If you happend to have read that formal definition and didn't enter deep comatose you may be wondering "Whoa, what was that context stuff that came outta nowhere! What's this guy trying to do here, sneaking a new concept in as if it was already introduced and described."

Worry not friend, lets discuss.

3.1.2 Putting it All Into Context

A key principle of Jaseci is to reshape and reimagine how we view data and memory. We do so by fusing the concept of data wit the intuitive and rich semantics of graphs as the lowest level primitive to view memory.

Nerd Alert 2 ***(time to let your eyes glaze over)***

A context is a representation of data that can be expressed simply as a 3-tuple $(\sum_K, \sum_V, p_K)$, where

1. $\sum_K$ is a finite alphabet of keys
2. $\sum_V$ is a finite alphabet of values
3. p_K is the pairing of keys to values

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

Now that we know what Jaseci is all about, next lets roll up our sleeves and jump in. One of the best ways to jump into Jaseci world is to gather some sample Jac programs and start tinkering with them.

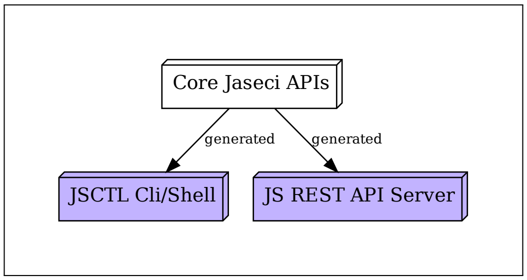


Figure 5.1: Jaseci Interface Architecture

Before we jump right into it, it's important to have a bit of an understanding of the the way the interface itself is architected from in the implementation of the Jaseci stack. Jaseci has a module that serves as its the core interface (summarized in Table 5.1) to the Jaseci machine. This interface is expressed as a set of method functions within a python class in Jaseci called `master`. (By the way, don't worry, it's ok to use "master", its not racialist, see Rant A.3 for more context). The 'client' expressions of that interface in the forms of a command line tool `jsctl` and a server-side REST API built using Django [1]. Figure 5.1 illustrates this architecture representing the relationship between core APIs and client side expressions.

If I may say so myself the code architecture of interface generation from function signatures is elegant, sexy, and takes advantage of the best python has to offer in terms of its support for introspection. With this approach, as the set of functions and their semantics change in the `master` API class, both the JSCTL Cli tool and the REST Server-side API changes. We dig into this and tons more in the Part III, so we'll leave the discussion on implementation

[1]Django [5] is a Python web framework for rapid development and clean, pragmatic design

architecture there for the moment. Lets jump right into how we get started playing with some leet Jaseci haxoring. First we start with JSCTL then dive into the REST API.

5.1 JSCTL: The Jaseci Command Line Interface

JSCTL or `jsctl` is a command line tool that provides full access to Jaseci. This tool is installed alongside the installation of the Jaseci Core package and should be accessible from the command line from anywhere. Let's say you've just checked out the Jaseci repo and you're in head folder. You should be able to execute the following.

```
haxor@linux:~/jaseci# pip3 install ./jaseci_core
Processing ./jaseci_core
...
Successfully installed jaseci-0.1.0
haxor@linux:~/jaseci# jsctl --help
Usage: jsctl [OPTIONS] COMMAND [ARGS]...

  The Jaseci Command Line Interface

Options:
  -f, --filename TEXT Specify filename for session state.
  -m, --mem-only Set true to not save file for session.
  --help Show this message and exit.

Commands:
  alias Group of ‘alias‘ commands
  architype Group of ‘architype‘ commands
  check Group of ‘check‘ commands
  config Group of ‘config‘ commands
  dev Internal dev operations
  edit Edit a file
  graph Group of ‘graph‘ commands
  login Command to log into live Jaseci server
  ls List relevant files
  object Group of ‘object‘ commands
  sentinel Group of ‘sentinel‘ commands
  walker Group of ‘walker‘ commands
haxor@linux:~/jaseci#
```

Here we've installed the Jaseci python package that can be imported into any python project with a directive such as `import jaseci`, and at the same time, we've installed the `jsctl` command line tool into our OS environment. At this point we can issue a call to say `jsctl --help` for any working directory.

Nerd Alert 3 ***(time to let your eyes glaze over)***

Python Code 5.1 shows the implementation of `setup.py` that is responsible for deploying the jsctl tool upon `pip3` installation of Jaseci Core.

Python Code 5.1: setup.py for Jaseci Core

```python
from setuptools import setup, find_packages
setup(
    name='jaseci',
    version='0.1.0',
    packages=find_packages(include=['jaseci', 'jaseci.*']),
    install_requires=[
        'click>=7.1.0,<7.2.0', 'click-shell>=2.0,<3.0',
        'numpy␣>=␣1.21.0,␣<␣1.22.0',
        'antlr4-python3-runtime>=4.9.0,<4.10.0',
        'requests',
        'flake8',
    ],
    package_data={"": ["*.ini"], },
    entry_points={
        'console_scripts': [
            'jsctl␣=␣jaseci.jsctl.jsctl:main'
        ]
    })
```

5.1.1 The Very Basics: CLI vs Shell-mode, and Session Files

This command line tool provides full access to the Jaseci core APIs via the command line, or a shell mode. In shell mode, all of the same Jaseci API functionally is available within a single session. To invoke shell-mode, simply execute `jsctl` without any commands and jsctl will enter shell mode as per the example below.

```
haxor@linux:~/jaseci# jsctl
Starting Jaseci Shell...
jaseci > graph create
{
  "context": {},
  "anchor": null,
  "name": "root",
  "kind": "generic",
  "jid": "urn:uuid:ef1eb3e4-91c3-40ba-ae7b-14c496f5ced1",
  "j_timestamp": "2021-08-15T15:15:50.903960",
  "j_type": "graph"
}
jaseci > exit
haxor@linux:~/jaseci#
```

Here we launched `jsctl` directly into shell mode for a single session and we can issue various calls to the Jaseci API for that session. In this example we issue a single call to `graph create`, which creates a graph within the Jaseci session with a single root node, then exit the shell with `exit`.

The exact behavior can be achieved without ever entering the shell directly from the command line as shown below.

```
haxor@linux:~/jaseci# jsctl graph create
{
  "context": {},
  "anchor": null,
  "name": "root",
  "kind": "generic",
  "jid": "urn:uuid:91dd8c79-24e4-4a54-8d48-15bee52c340b",
  "j_timestamp": "2021-08-15T15:40:12.163954",
  "j_type": "graph"
}
haxor@linux:~/jaseci#
```

All such calls to Jaseci's API (summarized in Table 5.1) can be issued either through shell-mode and CLI mode.

Session Files At this point, it's important to understand how sessions work. In a nutshell, a session captures the complete state of a jaseci machine. This state includes the status of memory, graphs, walkers, configurations, etc. The complete state of a Jaseci machine can be captured in a `.session` file. Every time state changes for a given session via the `jsctl` tool the assigned session file is updated. If you've been following along so far, try this.

```
haxor@linux:~/jaseci# ls *.session
js.session
haxor@linux:~/jaseci# jsctl graph list
[
  {
    "context": {},
    "anchor": null,
    "name": "root",
    "kind": "generic",
    "jid": "urn:uuid:ef1eb3e4-91c3-40ba-ae7b-14c496f5ced1",
    "j_timestamp": "2021-08-15T15:55:15.030643",
    "j_type": "graph"
  },
  {
    "context": {},
    "anchor": null,
    "name": "root",
    "kind": "generic",
    "jid": "urn:uuid:91dd8c79-24e4-4a54-8d48-15bee52c340b",
    "j_timestamp": "2021-08-15T15:55:46.419701",
    "j_type": "graph"
  }
]
haxor@linux:~/jaseci#
```

Note from the first call to `ls` we have a session file that has been created call `js.session`. This is the default session file `jsctl` creates and utilizes when called either in cli mode or shell mode. After listing session files, notices the call to `graph list` which lists the root nodes of all graphs created within a Jaseci machine's state. Note `jsctl` lists two such graph root nodes. Indeed these nodes correspond to the ones we've just created when contrasting cli mode and shell mode above. Having these two graphs demonstrates that across both instantiations of `jsctl` the same session, `js.session`, is being used. Now try the following.

```
haxor@linux:~/jaseci# jsctl -f mynew.session graph list
[]
haxor@linux:~/jaseci# ls *.session
js.session mynew.session
haxor@linux:~/jaseci#
```

Here we see that we can use the `-f` or `--filename` flag to specify the session file to use. In this case we list the graphs of the session corresponding to `mynew.session` and see the JSON representation of an empty list of objects. We then list session files and see that one was created for `mynew.session`. If we were to now type `jsctl --filename js.session graph list`, we would see a list of the two graph objects that we created earlier.

In-memory mode Its important to note that there is also an in-memory mode that can be created buy using the `-m` or `--mem-only` flags. This flag is particularly useful when you'd simply like to tinker around with a machine in shell-mode or you'd like to script some behavior to be executed in Jac and have no need to maintain machine state after completion. We will be using in memory session mode quite a bit, so you'll get a sense of its usage throughout this chapter. Next we actually see a workflow for tinkering.

5.1.2 A Simple Workflow for Tinkering

As you get to know Jaseci and Jac, you'll want to try things and tinker a bit. In this section, we'll get to know how `jsctl` can be used as the main platform for this play. A typical flow will involve jumping into shell-mode, writing some code, running that code to observe output, and in visualizing the state of the graph, and rendering that graph in dot to see it's visualization.

Install Graphvis Before we jump right in, let me strongly encourage you install Graphviz. Graphviz is open source graph visualization software package that includes a handy dandy command line tool call `dot`. Dot is also a standardized and open graph description language that is a key primitive of Graphviz. The `dot` tool in Graphviz takes dot code and renders it nicely. Graphviz is super easy to install. In Ubuntu simply type `sudo apt install graphviz`, or on mac type `brew install graphviz` and you're done! You should be able to call `dot` from the command line.

Ok, lets start with a scenario. Say you'd like to write your first Jac program which will include some nodes, edges, and walkers and you'd like to print to standard output and see what the graph looks like after you run an interesting walker. Let role play.

Lets hop into a `jsctl` shell.

```
haxor@linux:~/jaseci# jsctl -m
Starting Jaseci Shell...
jaseci >
```

Good, we're in! And we've set the session to be an in-memory session so no session file will be created or saved. For this play session we only care about the Jac program we write, which will be saved. The state of the Jaseci machine we run our toy program on doesn't really matter to us.

Now that we've got our shell running, we first want to create a blank graph. Remember, all walkers, Jaseci's primary unit of computation, must run on a node. As default, we can use the root node of a freshly created graph, hence we need to create a base graph. But oh no! We're a bit rusty and have forgotten how create our initial graph using `jsctl`. Let's navigate the help menu to jog our memories.

```
jaseci > help

Documented commands (type help <topic>):
========================================
alias check dev graph ls sentinel
architype config edit login object walker

Undocumented commands:
======================
exit help quit

jaseci > help graph
Usage: graph [OPTIONS] COMMAND [ARGS]...

  Group of `graph` commands

Options:
  --help Show this message and exit.

Commands:
  active Group of `graph active` commands
  create Create a graph instance and return root node graph object
  delete Permanently delete graph with given id
  get Return the content of the graph with format Valid modes:...
  list Provide complete list of all graph objects (list of root node...
  node Group of ‘graph node‘ commands
jaseci > graph create --help
Usage: graph create [OPTIONS]

  Create a graph instance and return root node graph object

Options:
  -o, --output TEXT Filename to dump output of this command call.
  -set_active BOOLEAN
  --help Show this message and exit.
jaseci >
```

Ohhh yeah! That’s it. After simply using `help` from the shell we were able to navigate to the relevant info for `graph create`. Let’s use this newly gotten wisdom.

```
jaseci > graph create -set_active true
{
  "context": {},
  "anchor": null,
  "name": "root",
  "kind": "generic",
  "jid": "urn:uuid:7aa6caff-7a46-4a29-a3b0-b144218312fa",
  "j_timestamp": "2021-08-15T21:34:31.797494",
  "j_type": "graph"
}
jaseci >
```

Great! With this command a graph is created and a single root node is born. `jsctl` shares with us the details of this root graph node. In Jaseci, graphs are referenced by their root nodes and every graph has a single root node.

Notice we’ve also set the `-set_active` parameter to true. This parameter informs Jaseci to use the root node of this graph (in particular the UUID of this root node) as the default parameter to all future calls to Jaseci Core APIs that have a parameter specifying a graph or node to operate on. This global designation that this graph is the ‘active’ graph is a convenience feature so we the user doesn’t have to specify this parameter for future calls. Of course this can be overridden, more on that later.

Next, lets write some Jac code for our little program. `jsctl` has a built in editor that is simple yet powerful. You can use either this built in editor, or your favorite editor to create the `.jac` file for our toy program. Let’s use the built in editor.

```
jaseci > edit fam.jac
```

The `edit` command invokes the built in editor. Though it’s a terminal editor based on `ncurses`, you can basically use it much like you’d use any wysiwyg editor with features like standard cut `ctrl-c` and paste `ctrl-v`, mouse text selection, etc. It's based on the phenomenal pure python project from Google called `ci_edit`. For more detailed help cheat sheet see Appendix C. If you must use your own favorite editor, simply be sure that you save the fam.jac file in the same working directory from which you are running the Jaseci shell. Now type out the toy program in Jac Code 5.2.

Jac Code 5.2: Jac Family Toy Program

```
node man;
node woman;

edge mom;
edge dad;
edge married;

walker create_fam {
    root {
        spawn here --> node::man;
        spawn here --> node::woman;
        --> node::man <-[married]-> --> node::woman;
        take -->;
    }
    woman {
        son = spawn here <-[mom]- node::man;
        son -[dad]-> <-[married]->;
    }
    man {
        std.out("I didn't do any of the hard work.");
    }
}
```

Don't worry if that looks like the most cryptic gobbledygook you've ever seen in your life. As you learn the Jac language, all will become clear. For now, lets tinker around. Now save and quit the editor. If you are using the built in editor thats simply a `ctrl-s, ctrl-q` combo.

Ok, now we should have a `fam.jac` file saved in our working directory. We can check from the Jaseci shell!

```
jaseci > ls
fam.jac
jaseci >
```

We can list files from the shell prompt. By default the `ls` command only lists files relevant to Jaseci (i.e., `*.jac`, `*.dot`, etc). To list all files simply add a `--all` or `-a`.

Now, on to what is on of the key operations. Lets "register" a sentinel based on our Jac program. A sentinel is the abstraction Jaseci uses to encapsulate compiled walkers and architype nodes and edges. You can think of registering a sentinel as compiling your jac program. The walkers of a given sentinel can then be invoked and run on arbitrary nodes of any graph. Let's register our Jac toy program.

```
jaseci > sentinel register -name fam -code fam.jac -set_active true
2021-08-15 18:03:38,823 - INFO - parse_jac_code: fam: Processing Jac code...
2021-08-15 18:03:39,001 - INFO - register_code: fam: Successfully registered code
{
  "name": "fam",
  "kind": "generic",
  "jid": "urn:uuid:cfc9f017-cb6c-4d06-bc45-758289c96d3f",
  "j_timestamp": "2021-08-15T22:03:38.823651",
  "j_type": "sentinel"
}
jaseci >
```

Ok, theres a lot that just happened there. First, we see some logging output that informs us that the Jac code is being processed (which really means the Jac program is being parsed and IR being generated). If there are any syntax errors or other issues, this is where the error output will be printed along with any problematic lines of code and such. If all goes well, we see the next logging output that the code has been successfully registered. The formal output is the relevant details of the successfully created sentinel. Note, that we've also made this the "active" sentinel meaning it will be used as the default setting for any calls to Jaseci Core APIs that require a sentinel be specified. At this point, Jaseci has registered our code and we are ready to run walkers!

But first, lets take a quick look at some of the objects loaded into our Jaseci machine. For this I'll briefly introduce the `alias` group of APIs.

```
jaseci > alias list
{
  "sentinel:fam": "urn:uuid:cfc9f017-cb6c-4d06-bc45-758289c96d3f",
  "fam:walker:create_fam": "urn:uuid:17598be7-e14f-4000-9d85-66b439fa7421",
  "fam:architype:man": "urn:uuid:c366518d-3b1e-41a3-b1ba-0b9a3ce6e1d6",
  "fam:architype:woman": "urn:uuid:7eb1c510-73ca-49eb-96aa-34357f77b4cb",
  "fam:architype:mom": "urn:uuid:8c9d2a66-4954-4d11-8109-a36b961eeea1",
  "fam:architype:dad": "urn:uuid:d80111e4-62e2-4694-bfaa-f3294d9520d8",
  "fam:architype:married": "urn:uuid:dc4974df-ea57-406e-9468-a1aa5260d306"
}
jaseci >
```

The `alias` set of APIs are designed as an additional set of convenience tools to simplify the referencing of various objects (walkers, architypes, etc) in Jaseci. Instead of having to use the UUIDs to reference each object, an alias can be used to refer to any object. These aliases can be created or removed utilizing the `alias` APIs.

Upon registering a sentinel, a set of aliases are automatically created for each object produced from processing the corresponding Jac program. The call to `alias list` lists all available aliases in the session. Here, we're using this call to see the objects that were created for our toy program and validate it corresponds to the ones we would expect from the Jac Program represented in JC 5.2. Everything looks good!

Now, for the big moment! lets run our walker on the root node of the graph we created and see what happens!

```
jaseci > walker run -name create_fam
I didn't do any of the hard work.
[]
jaseci >
```

Sweet!! We see the standard output we'd expect from our toy program. Hrm, as we'd expect, when it comes to the family, the man doesn't do much it seems.

But there were many semantics to what our toy program does. How do we visualize that the graph produced by or program is right. Well we're in luck! We can use Jaseci 'dot' features to take a look at our graph!!

```
jaseci > graph get -mode dot -o fam.dot
strict digraph root {
    "n0" [ id="550ce1bb405c4477947e019d1e8428eb", label="n0:root" ]
    "n1" [ id="e5c0a9b28f134313a28794a0c061bff1", label="n1:man" ]
    "n2" [ id="bc2d2f18e2de4190a50bec2a32392a4f", label="n2:woman" ]
    "n3" [ id="92ed7781c6674824905b149f7f320fcd", label="n3:man" ]
    "n1" -> "n3" [ id="76535f6c3f0e4b7483c31863299e2784", label="e0:dad" ]
    "n3" -> "n2" [ id="6bb83ee19f8b4f7eb93a11f5d4fa7f0a", label="e1:mom" ]
    "n1" -> "n2" [ id="0fc3550e75f241ce8d1660860cf4e5c9", label="e2:married", dir="both" ]
    "n0" -> "n2" [ id="03fcfb60667b4631b46ee589d982e1ce", label="e3" ]
    "n0" -> "n1" [ id="d1713ac5792e4272b9b20917b0c3ec33", label="e4" ]
}
[saved to fam.dot]
jaseci >
```

Here we've used the `graph get` core API to get a print out of the graph in dot format. By default `graph get` dumps out a list of all edge and node objects of the graph, however with the `-mode dot` parameter we've specified that the graph should be printed in dot. The `-o` flag specifies a file to dump the output of the command. Note that the `-o` flag for `jsctl` commands only outputs the formal returned data (json payload, or string) from a Jaseci Core API. Logging output, standard output, etc will not be saved to the file though anything reported by a walker using `report` will be saved. This output file directive is `jsctl` specific and work with any command given to `jsctl`.

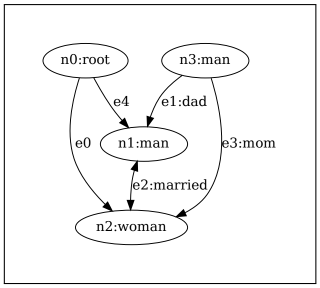


Figure 5.2: Graph for `fam.jac`

To see a pretty visual of the graph itself, we can use the `dot` command from Graphviz. Simply type `dot -Tpdf fam.dot`

`-o fam.pdf` and Voila! We can see the beautiful graph our toy Jac program has produced on its way to the standard output.

Awesomeness! We are Jac Haxors now!

5.2 Jaseci REST API

5.3 Full Spec of Jaseci Core APIs

5.3.1 Cheatsheet

Interface	**Parameters**
config delete	(name: str, do_check: bool = True)
config exists	(name: str)
config get	(name: str, do_check: bool = True)
config list	()
config set	(name: str, value: str, do_check: bool = True)
global delete	(name: str)
global sentinel set	(snt: jaseci.actor.sentinel.sentinel = None)
global sentinel unset	()
global set	(name: str, value: str)
global get	(name: str)
logger http clear	(log: str = 'all')
logger http connect	(host: str, port: int, url: str, log: str = 'all')
logger list	()
master createsuper	(name: str, set_active: bool = True, other_fields: dict =)
master active get	(detailed: bool = False)
master active set	(name: str)
master active unset	()
master create	(name: str, set_active: bool = True, other_fields: dict =)
master delete	(name: str)
master get	(name: str, mode: str = 'default', detailed: bool = False)
master list	(detailed: bool = False)
alias clear	()
alias delete	(name: str)
alias list	()
alias register	(name: str, value: str)
architype delete	(arch: jaseci.actor.architype.architype, snt: jaseci.actor.sentinel.sentinel = None)

architype get	(arch: jaseci.actor.architype.architype, mode: str = 'default', detailed: bool = False)
architype list	(snt: jaseci.actor.sentinel.sentinel = None, detailed: bool = False)
architype register	(code: str, encoded: bool = False, snt: jaseci.actor.sentinel.sentinel = None)
architype set	(arch: jaseci.actor.architype.architype, code: str, mode: str = 'default')
graph active get	(detailed: bool = False)
graph active set	(gph: jaseci.graph.graph.graph)
graph active unset	()
graph create	(set_active: bool = True)
graph delete	(gph: jaseci.graph.graph.graph)
graph get	(gph: jaseci.graph.graph.graph = None, mode: str = 'default', detailed: bool = False)
graph list	(detailed: bool = False)
graph node get	(nd: jaseci.graph.node.node, ctx: list = None)
graph node set	(nd: jaseci.graph.node.node, ctx: dict, snt: jaseci.actor.sentinel.sentinel = None)
object get	(obj: jaseci.element.element.element, depth: int = 0, detailed: bool = False)
object perms get	(obj: jaseci.element.element.element)
object perms grant	(obj: jaseci.element.element.element, mast: jaseci.element.element.element, read_only: bool = False)
object perms revoke	(obj: jaseci.element.element.element, mast: jaseci.element.element.element)
object perms set	(obj: jaseci.element.element.element, mode: str)
sentinel active get	(detailed: bool = False)
sentinel active global	(detailed: bool = False)
sentinel active set	(snt: jaseci.actor.sentinel.sentinel)
sentinel active unset	()
sentinel delete	(snt: jaseci.actor.sentinel.sentinel)
sentinel get	(snt: jaseci.actor.sentinel.sentinel = None, mode: str = 'default', detailed: bool = False)
sentinel list	(detailed: bool = False)
sentinel pull	(set_active: bool = True, on_demand: bool = True)
sentinel register	(name: str, code: str = ", encoded: bool = False, auto_run: str = 'init', ctx: dict = , set_active: bool = True)
sentinel set	(code: str, encoded: bool = False, snt: jaseci.actor.sentinel.sentinel = None, mode: str = 'default')
walker delete	(wlk: jaseci.actor.walker.walker, snt: jaseci.actor.sentinel.sentinel = None)
walker execute	(wlk: jaseci.actor.walker.walker)

walker get	(wlk: jaseci.actor.walker.walker, mode: str = ’default’, detailed: bool = False)
walker list	(snt: jaseci.actor.sentinel.sentinel = None, detailed: bool = False)
walker prime	(wlk: jaseci.actor.walker.walker, nd: jaseci.graph.node.node = None, ctx: dict =)
walker register	(snt: jaseci.actor.sentinel.sentinel = None, code: str = ”, encoded: bool = False)
walker run	(name: str, nd: jaseci.graph.node.node = None, ctx: dict = , snt: jaseci.actor.sentinel.sentinel = None)
walker set	(wlk: jaseci.actor.walker.walker, code: str, mode: str = ’default’)
walker spawn	(name: str, snt: jaseci.actor.sentinel.sentinel = None)
walker unspawn	(wlk: jaseci.actor.walker.walker)
walker summon	(self, key: str, wlk: jaseci.actor.walker.walker, nd: jaseci.graph.node.node, ctx: dict =)

Table 5.1: Full set of core Jaseci APIs

5.3.2 alias APIs

Alias APIs for creating nicknames for UUIDs and other long strings

The alias set of APIs provide a set of ‘alias’ management functions for creating and managing aliases for long strings such as UUIDs. If an alias’ name is used as a parameter value in any API call, that parameter will see the alias’ value instead. Given that references to all sentinels, walkers, nodes, etc. utilize UUIDs, it becomes quite useful to create pneumonic names for them. Also, when registering sentinels, walkers, architype handy aliases are automatically generated. These generated aliases can then be managed using the alias APIs. Keep in mind that whenever an alias is created, all parameter values submitted to any API with the alias name will be replaced internally by its value. If you get in a bind, simply use the clear or delete alias APIs.

5.3.2.1 alias clear

5.3.2.2 alias delete

5.3.2.3 alias list

5.3.2.4 alias register

5.3.3 architype APIs

Architype APIs for creating and managing Jaseci architypes

The architype set of APIs allow for the addition and removing of architypes. Given a Jac implementation of an architype these APIs are designed for creating, compiling, and managing architypes that can be used by Jaseci. There are two ways to add an architype to Jaseci, either through the management of sentinels using the sentinel API, or by registering independent architypes with these architype APIs. These APIs are also used for inspecting and managing existing arichtypes that a Jaseci instance is aware of.

5.3.3.1 architype delete

5.3.3.2 architype get

5.3.3.3 architype list

5.3.3.4 architype register

5.3.3.5 architype set

5.3.4 config APIs

Admin config APIs Abstracted since there are no valid configs in core atm, see jaseci_serv to see how used.

5.3.4.1 config delete

5.3.4.2 config exists

5.3.4.3 config get

5.3.4.4 config list

5.3.4.5 config set

5.3.5 global APIs

Admin global APIs

5.3.5.1 global delete

5.3.5.2 global sentinel set

5.3.5.3 global sentinel unset

5.3.5.4 global set

5.3.6 graph APIs

Graph APIs

5.3.6.1 graph active get

5.3.6.2 graph active set

5.3.6.3 graph active unset

5.3.6.4 graph create

5.3.6.5 graph delete

5.3.6.6 graph get

5.3.6.7 graph list

5.3.6.8 graph node get

5.3.6.9 graph node set

5.3.7 logger APIs

APIs for Jaseci Logging configuration

5.3.7.1 logger http clear

5.3.7.2 logger http connect

5.3.7.3 logger list

5.3.8 master APIs

Master APIs for creating nicknames for UUIDs and other long strings

5.3.8.1 master active get

5.3.8.2 master active set

5.3.8.3 master active unset

5.3.8.4 master create

5.3.8.5 master delete

5.3.8.6 master get

5.3.8.7 master list

5.3.9 object APIs

Object APIs for generalized operations on Jaseci objects

...

5.3.9.1 global get

5.3.9.2 object get

5.3.9.3 object perms get

5.3.9.4 object perms grant

5.3.9.5 object perms revoke

5.3.9.6 object perms set

5.3.10 public APIs

Public APIs

5.3.10.1 walker summon

5.3.11 sentinel APIs

Sentinel APIs

5.3.11.1 sentinel active get

5.3.11.2 sentinel active global

5.3.11.3 sentinel active set

5.3.11.4 sentinel active unset

5.3.11.5 sentinel delete

5.3.11.6 sentinel get

5.3.11.7 sentinel list

5.3.11.8 sentinel pull

5.3.11.9 sentinel register

5.3.11.10 sentinel set

5.3.12 super APIs

Super APIs for creating nicknames for UUIDs and other long strings

5.3.12.1 master createsuper

5.3.13 walker APIs

Walker APIs

5.3.13.1 walker delete

5.3.13.2 walker execute

5.3.13.3 walker get

5.3.13.4 walker list

5.3.13.5 walker prime

5.3.13.6 walker register

5.3.13.7 walker run

5.3.13.8 walker set

5.3.13.9 walker spawn create

5.3.13.10 walker spawn delete

5.3.13.11 walker spawn list

Part II

The Jac Programming Language

Chapter 6

Jac Language Overview and Basics

To articulate the sorcerer spells made possible by the wand that is Jaseci, I bestow upon thee, the Jac programming language. (Like the Harry Potter [6] simile there? Cool, I know ;-))

The name Jac take was chosen for a few reasons.

- "Jac" is three characters long, so its well suited for the file name extension `.jac` for Jac programs.
- It pulls its letters from the phrase **JA**seci **C**ode.
- And it sounds oh so sweet to say "Did you grok that sick Jac code yet!" Rolls right off the tongue.

This chapter provides the full deep dive into the language. By the end, you will be fully empowerd with Jaseci wizardry and get a view into the key insights and novelty in the coding style.

First lets quickly dispense with the mundane. This section covers the standard table stakes fodder present in pretty much all languages. These aspects of Jac must be covered for completeness, however you should be able to speed read this section. If you are unable to speed read this, perhaps you should give visual basic a try.

Figure 6.1: World's youngest coder with valid HTML on shirt.[1]

6.1 The Obligatory Hello World

Let's begin with what has become the unofficial official starting point for any introduction to a new language, the "hello world" program. Thank you Canada for providing one of the most impactful contributions in computer science with "hello world" becoming a meme both technically and socially. We have such love for this contribution we even tag or newborns with the phrase as per Fig. 6.1. I digress. Lets now christen our baby, Jaseci, with its "Hello World" expression.

Jac Code 6.1: Jaseci says Hello!

```
walker init {
    std.out("Hello World");
}
```

Simple enough right? Well let's walk through it. What we have here is a valid Jac program with a single walker defined. Remember a walker is our little robot friend that walks the nodes and edges of a graph and does stuff. In the curly braces, we articulate what our walker should do. Here we instruct our walker to utilize the standard library to call a print function denoted as `std.out` to print a single string, our star and esteemed string, "Hello World." The output to the screen (or wherever the OS is routing it's standard stream output) is simply,

```
Hello World
```

[1]Image credit to wiki contributer [1]

And there we have the most useless program in the world. Though. . . technically this program is AI. Its not as intelligent as the machine depicted in Figure 6.1, but one that we can understand much better (unless you speak "goo goo gaa gaa" of course). Let's move on.

6.2 Numbers, Arithmetic, and Logic

6.2.1 Basic Arithmetic Operations

Next we should cover the he simplest math operations in Jac. We build upon what we've learned so far with our conversational AI above.

Jac Code 6.2: Basic arithmetic operations

```
walker init {
    a = 4 + 4;
    b = 4 * -5;
    c = 4 / 4; # Evaluates to a floating point number
    d = 4 - 6;
    e = a + b + c + d;
    std.out(a, b, c, d, e);
}
```

The output of this groundbreaking program is,

```
8 -20 1.0 -2 -13.0
```

Jac Code 6.2 is comprised of basic math operations. The semantics of these expressions are pretty much the same as anything you may have seen before, and pretty much match the semantics we have in the Python language. In this Example, we also observe that Jac is an untyped language and variables can be declared via a direct assignment; also very Python'y. The comma separated list of the defined variables `a` - `e` in the call to `std.out` illustrate multiple values being printed to screen from a single call.

Additionally, Jac supports power and modulo operations.

Jac Code 6.3: Additional arithmetic operations

```
walker init {
    a = 4 ^ 4; b = 9 % 5; std.out(a, b);
}
```

Jac Code 6.3 outputs,

```
256 4
```

Here, we can also observe that, unlike Python, whitespace does not mater whatsoever. Languages utilizing whitespace to express static scoping should be criminalized. Yeah, I said it, see Rant A.2. Anyway, A corollary to this design decision is that every statement must end with a “`;`”. The wonderful `;`, A nod of respect goes to `C/C++/JavaScript` for bringing this beautiful code punctuation to the masses. Of course the `;` as code punctuation was first introduced with `ALGOL 58`, but who the heck knows that language. It sounds like some kind of plant species. Bleh. Onwards.

Nerd Alert 4 ***(time to let your eyes glaze over)***

Grammar 6.4 shows the lines from the formal grammar for Jac that corresponds to the parsing of arithmetic.

Grammar 6.4: Jac grammar clip relevant to arithmetic

```
arithmetic: term ((PLUS | MINUS) term)*;

term: factor ((MUL | DIV | MOD) factor)*;

factor: (PLUS | MINUS) factor | power;

power: func_call (POW factor)* | func_call index+;
```

(full grammar in Appendix B)

6.2.2 Comparison, Logical, and Membership Operations

Next we review the comparison and logical operations supported in Jac. This is relatively straight forward if you’ve programmed before. Let’s summarize quickly for completeness.

Jac Code 6.5: Comparision operations

```
walker init {
    a = 5; b = 6;
    std.out(a == b,
            a != b,
            a < b,
            a > b,
            a <= b,
            a >= b,
            a == b-1);
}
```

```
false true true false true false true
```

In order of appearance, we have tests for equality, non equality, less than, greater than, less than or equal, and greater than or equal. These tools prove indispensible when expressing functionality through conditionals and loops. Additionally,

Jac Code 6.6: Logical operations

```
walker init {
    a = true; b = false;
    std.out(a,
            !a,
            a && b,
            a || b,
            a and b,
            a or b,
            !a or b,
            !(a and b));
}
```

```
true false false true false true false true
```

Jac Code 6.6 presents the logical operations supported by Jac. In oder of appearance we have, boolean complement, logical and, logical or, another way to express and and or (thank you Python) and some combinations. These are also indispensible when using conditionals.

[NEED EXAMPLE FOR MEMBERSHIP OPERATIONS]

Nerd Alert 5 ***(time to let your eyes glaze over)***

Grammar 6.7 shows the lines from the formal grammar for Jac that corresponds to the parsing of comparison, logical, and membership operations.

Grammar 6.7: Jac grammar clip relevant to comparison, logic, and membership

```
logical: compare ((KW_AND | KW_OR) compare)*;

compare:
        NOT compare
        | arithmetic (
                (EE | LT | GT | LTE | GTE | NE | KW_IN | nin) arithmetic
        )*;

nin: NOT KW_IN;
```

(full grammar in Appendix B)

6.2.3 Assignment Operations

Next, lets take a look at assignment in Jac. In contrast to equality tests of `==`, assignment operations copy the value of the right hand side of the assignment to the variable or object on the left hand side.

Jac Code 6.8: Assignment operations

```
walker init {
    a = 4 + 4; std.out(a);
    a += 4 + 4; std.out(a);
    a -= 4 * -5; std.out(a);
    a *= 4 / 4; std.out(a);
    a /= 4 - 6; std.out(a);

    # a := here; std.out(a);
    # Noting existence of copy assign, described later
}
```

```
8
16
36
36.0
-18.0
```

As shown in Jac Code 6.8, there are a number of ways we can articulate an assignment. Of course we can simply set a variable equal to a particular value, however, we can go beyond that to set that assignment relative to its original value. In particular, we can use the short hand `a += 4 + 4;` to represent `a = a + 4 + 4;`. We will describe later an additional assignment type we call the copy assign. If you're simply dying of curiosity, I'll throw you a bone. This `:=` assignment only applies to nodes and edges and has the semantic of copying the member values of a node or edge as opposed to the particular node or edge a variable is pointing to. In a nutshell this assignment uses pass by value semantics vs pass by reference semantics which is default for nodes and edges.

Nerd Alert 6 ***(time to let your eyes glaze over)***

Grammar 6.9 shows the lines from the formal grammar for Jac that corresponds to the parsing of assignment operations.

Grammar 6.9: Jac grammar clip relevant to assignment

```
assignment:
       dotted_name index* EQ expression
       | inc_assign
       | copy_assign;

inc_assign:
       dotted_name index* (PEQ | MEQ | TEQ | DEQ) expression;

copy_assign: dotted_name index* CPY_EQ expression;
```

(full grammar in Appendix B)

Rank	Symbol	Description
1	`( ), [ ], ., ::, spawn`	Parenthetical/grouping, node/edge manipulation
2	`^, []`	Exponent, Index
3	`*, /, %`	Multiplication, division, modulo
4	`+, -`	Addition, subtraction
5	`==, !=, >=, <=, >, <, in, not in`	Comparison
6	`&&, ||, and, or`	Logical
7	`-->, <--, -[]->, <-[]-`	Connect
8	`=, +=, -=, *=, /=, :=`	Assignment

Table 6.1: Precedence of operations in Jac

6.2.4 Precedence in Jac

At this point in our discussion its important to note the precedence of operations in Jac. Table 6.1 summarizes this precedence. There are a number of new and perhaps interesting things that appear in this table that you may not have seen before. [JOKE] For now, don't hurt yourself trying to understand what they are and mean, we'll get there.

6.3 Foreshadowing Unique Graph Operations

Before we move on to more mundane basics that will continue to neutralize any kind of caffeine or methamphetamine buzz an experienced coder might have as they read this, lets enjoy a Jaseci jolt!

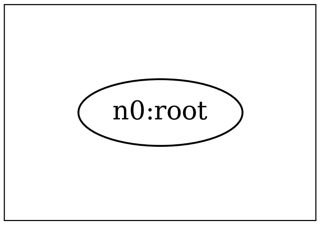


Figure 6.2: Graph in memory for simple Hello World program (JC 6.1)

As described before, all data in Jaseci lives in either a graph, or within the scope of a walker. A walker, executes when it is *engaged* to the graph, meaning it is located on a particular node of the graph. In the case of the Jac programs we've looked at so far, each program has specified one walker for which I've happened to choose the name `init`. By default these init walkers are invoked from the default root node of an empty graph. Figure 6.2 shows the complete state of memory for all of the Jac programs discussed thus far. The `init` walker in these cases does not *walk* anywhere and has only executed a set of operations on this default root node `n0`.

Let's have a quick peek at some slick language syntax for building this graph and traveling to new nodes.

Jac Code 6.10: Preview of graph operators

```
node simple;
edge back;

walker kewl_graph_creator {
    node_a = spawn here --> node::simple;
    here <-[back]- node_a;
    node_b = spawn here <--> node::simple;
    node_b --> node_a;
}
```

Jac Code 6.10 presents a sequence of operations that creates nodes and edges and produces a relatively simple complex graph. There is a bunch of new syntactic goodness presented in less than 10 lines of code and I certainly won't describe them all here. The goal is to simply whet your appetite on whats to come. But lets look at the state of our data (memory) shown in Figure 6.3.

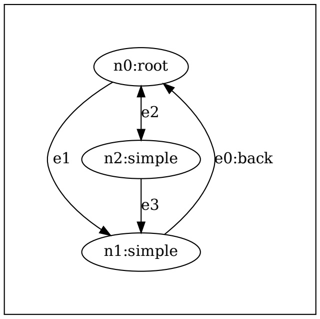


Figure 6.3: Graph in memory for JC 6.10

Yep, there a good bit going on here. in less than 10 lines of code we've done the following things:

1. Specified a new type of node we call a simple node.
2. Specified a new type of edge we call a back edge.
3. Specified a walker `kewl_graph_creator` and its behavior
4. Instantiated a outward pointing edge from the `n0:root` node.
5. Instantiated an instance of node type `simple`
6. Connected edge from from `root` to `n1`
7. Instantiated a `back` edge
8. Connected `back` edge from `n1` to `n0`
9. Instantiated another instance of node type `simple`, `n2`
10. Instantiated an undirected edge from the `n0:root` node.
11. Connected edge from `root` to `n2`
12. Instantiated an outward pointing edge from `n2`
13. Connected edge from `n2` to `n1`

Don't worry, I'll wait till that sinks in. . . Good? Well, if you liked that, just you wait.

This is going to get very interesting indeed, but first, on to more standard stuff. . .

6.4 Strings, Lists, and Dictionaries

6.5 Control Flow

Chapter 7

Graphs, Architypes, and Walkers in Jac

7.1 Structure of a Jac Program

[Introduce structure of a jac program]

[Specify the differnce between graph architypes, graph instantiations, and walkers]

[Present simple program that utilizes the structures]

[Present variations on articulating the same program]

Nerd Alert 7 ***(time to let your eyes glaze over)***

Grammar 7.1 shows the lines from the formal grammar for Jac that presents the high level structure of a Jac program.

Grammar 7.1: Jac grammar clip relevant to arithmetic

```
start: ver_label? element+ EOF;

element: architype | walker;

architype:
       KW_NODE NAME (COLON INT)? attr_block
       | KW_EDGE NAME attr_block
       | KW_GRAPH NAME graph_block;

walker:
       KW_WALKER NAME namespace_list LBRACE attr_stmt* walk_entry_block? (
              statement
              | walk_activity_block
       )* walk_exit_block? RBRACE;
```

(full grammar in Appendix B)

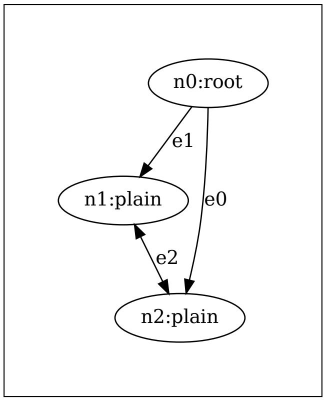


Figure 7.1: Graph in memory for JC 7.2

Jac Code 7.2: Simple walker creating and connected nodes

```
node plain;

walker init {
    node1 = spawn node::plain;
    node2 = spawn node::plain;
    node1 <--> node2;
    here --> node1;
    node2 <-- here;
}
```

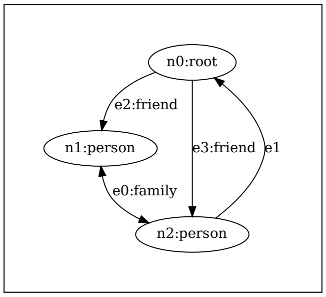


Figure 7.2: Graph in memory for JC 7.3

Jac Code 7.3: Creating named node types

```
node person;
edge family;
edge friend;

walker init {
    node1 = spawn node::person;
    node2 = spawn node::person;
    node1 <-[family]-> node2;
    here -[friend]-> node1;
    node2 <-[friend]- here;

    # named and unnamed edges and nodes can be mixed
    node2 --> here;
}
```

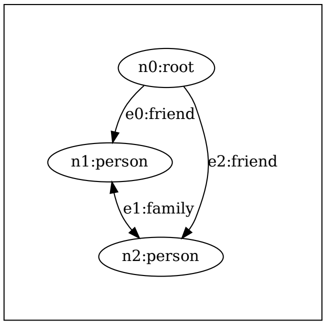


Figure 7.3: Graph in memory for JC 7.4

Jac Code 7.4: Connecting nodes within spawn statement

```
node person;
edge friend;
edge family;

walker init {
    node1 = spawn here -[friend]-> node::person;
    node2 = spawn node1 <-[family]-> node::person;
    here -[friend]-> node2;
}
```

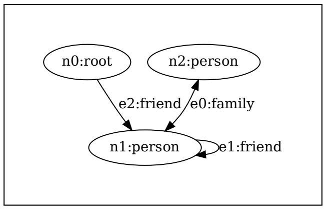


Figure 7.4: Graph in memory for JC 7.5

Jac Code 7.5: Chaining node connections using the connect operator

```
node person;
edge friend;
edge family;

walker init {
    node1 = spawn node::person;
    node2 = spawn node::person;
    node2 <-[friend]- here -[friend]-> node1
          <-[family]-> node2;
}
```

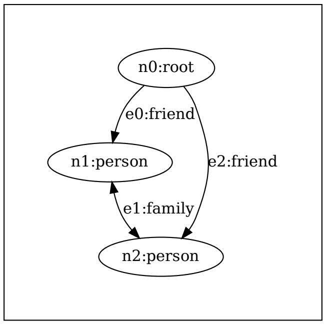


Figure 7.5: Graph in memory for JC 7.6

Jac Code 7.6: Walkers spawning other walkers

```
node person;
edge friend;
edge family;

walker friend_ties {
    for i in -[friend]->:
        std.out(here, 'is␣related␣to\n', i, '\n');
}

walker init {
    node1 = spawn here -[friend]-> node::person;
    node2 = spawn node1 <-[family]-> node::person;
    here -[friend]-> node2;
    spawn here walker::friend_ties;
}
```

```
graph:generic:root:urn:uuid:f93bca4a-a722-4fd7-b5e1-55372b4dd314 is related to
 node:node:person:urn:uuid:18411a74-60ac-4223-9d59-c3e6a8de7179

graph:generic:root:urn:uuid:f93bca4a-a722-4fd7-b5e1-55372b4dd314 is related to
 node:node:person:urn:uuid:2d251260-3086-4f4f-b5e0-fd36f6043ac7
```

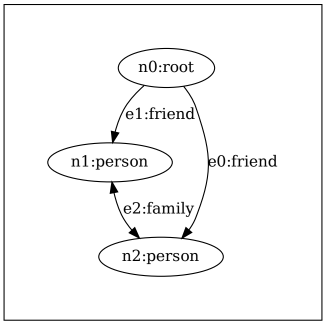


Figure 7.6: Graph in memory for JC 7.7

Jac Code 7.7: Getting returned values from spawned walkers

```
node person;
edge friend;
edge family;

walker friend_ties {
    has anchor fam_nodes;
    fam_nodes = -[friend]->;
}

walker init {
    node1 = spawn here -[friend]-> node::person;
    node2 = spawn node1 <-[family]-> node::person;
    here -[friend]-> node2;
    fam = spawn here walker::friend_ties;
    for i in fam:
        std.out(here, 'is␣related␣to\n', i, '\n');
}
```

```
graph:generic:root:urn:uuid:75d1050b-a010-4e6d-ad6a-c941d5ce57ce is related to
 node:node:person:urn:uuid:b1b6ead0-0fc6-4736-928a-f8500832fb3b

graph:generic:root:urn:uuid:75d1050b-a010-4e6d-ad6a-c941d5ce57ce is related to
 node:node:person:urn:uuid:914af4dd-6d5a-4f00-a70c-8871db4a8b95
```

Jac Code 7.8: Increasing elegance by remembering spawns are expressions

```
node person;
edge friend;
edge family;

walker friend_ties {
    has anchor fam_nodes;
    fam_nodes = -[friend]->;
}

```

```
walker init {
    node1 = spawn here -[friend]-> node::person;
    node2 = spawn node1 <-[family]-> node::person;
    here -[friend]-> node2;
    for i in spawn here walker::friend_ties:
        std.out(here, 'is␣related␣to\n', i, '\n');
}
```

Part III

Crafting Jaseci

Chapter 8

Architecting Jaseci Core

Chapter 9

Architecting Jaseci Cloud Serving

Epilogue

Appendix A

Rants

A.1 Libraries Suck

Because they do.

Still need more reasons?

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

. . .

Still there?

. . .

Fine, I'll tell you.

1. They suck because they create dependencies 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 expressing slightly different idiosyncratic 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.

A.2 Utilizing Whitespace for Scoping is Criminal (Yea, I'm looking at you Python)

This whitespace debathery perpetrated by Python and the like is one of the most perverted abuses of ASCII code 32 I've seen in computer science. It's an assault on the freedom of coders to decide the shape and structure of the beautiful sculptures their creative minds might want to actualize in syntax. Coder's fingers have a voice! And that voice deserves to be heard! The only folks that support this oppression are those in the 1% that get paid on a per line of code basis so they can lean on these whitespace mandates to pump up their salaries at the cost of coders everywhere.

"FREE THE PEOPLE! FREE THE CODE!"

"FREE THE PEOPLE! FREE THE CODE!"

"FREE THE PEOPLE! FREE THE CODE!"

A.3 "Using master is NOT Racist!" by a black dude

Rant here

Appendix B

Full Jac Grammar Specification

Grammar B.1: Full listing of Jac Grammar (antlr4)

```
grammar jac;

start: ver_label? element+ EOF;

element: architype | walker;

architype:
        KW_NODE NAME (COLON INT)? attr_block
        | KW_EDGE NAME attr_block
        | KW_GRAPH NAME graph_block;

walker:
        KW_WALKER NAME namespace_list LBRACE attr_stmt* walk_entry_block? (
                statement
                | walk_activity_block
        )* walk_exit_block? RBRACE;

ver_label: 'version' COLON STRING SEMI?;

namespace_list: COLON NAME (COMMA NAME)* |;

walk_entry_block: KW_WITH KW_ENTRY code_block;

walk_exit_block: KW_WITH KW_EXIT code_block;

walk_activity_block: KW_WITH KW_ACTIVITY code_block;

attr_block: LBRACE (attr_stmt)* RBRACE | COLON attr_stmt | SEMI;

attr_stmt: has_stmt | can_stmt;

```

```
graph_block: graph_block_spawn | graph_block_dot;

graph_block_spawn:
       LBRACE has_root KW_SPAWN code_block RBRACE
       | COLON has_root KW_SPAWN code_block SEMI;

graph_block_dot:
       LBRACE has_root dot_graph RBRACE
       | COLON has_root dot_graph SEMI;

has_root: KW_HAS KW_ANCHOR NAME SEMI;

has_stmt:
       KW_HAS KW_PRIVATE? KW_ANCHOR? has_assign (COMMA has_assign)* SEMI;

has_assign: NAME | NAME EQ expression;

can_stmt:
       KW_CAN dotted_name preset_in_out? event_clause? (
              COMMA dotted_name preset_in_out? event_clause?
       )* SEMI
       | KW_CAN NAME event_clause? code_block;

event_clause: KW_WITH (KW_ENTRY | KW_EXIT | KW_ACTIVITY);

preset_in_out:
       DBL_COLON NAME (COMMA NAME)* (DBL_COLON | COLON_OUT NAME)?;

dotted_name: NAME (DOT NAME)*;

code_block: LBRACE statement* RBRACE | COLON statement;

node_ctx_block: NAME (COMMA NAME)* code_block;

statement:
       code_block
       | node_ctx_block
       | expression SEMI
       | if_stmt
       | for_stmt
       | while_stmt
       | ctrl_stmt SEMI
       | report_action
       | walker_action;

if_stmt: KW_IF expression code_block (elif_stmt)* (else_stmt)?;

elif_stmt: KW_ELIF expression code_block;

else_stmt: KW_ELSE code_block;

for_stmt:
       KW_FOR expression KW_TO expression KW_BY expression code_block
       | KW_FOR NAME KW_IN expression code_block;

```

```
while_stmt: KW_WHILE expression code_block;

ctrl_stmt: KW_CONTINUE | KW_BREAK | KW_SKIP;

report_action: KW_REPORT expression SEMI;

walker_action:
	ignore_action
	| take_action
	| destroy_action
	| KW_DISENGAGE SEMI;

ignore_action: KW_IGNORE expression SEMI;

take_action: KW_TAKE expression (SEMI | else_stmt);

destroy_action: KW_DESTROY expression SEMI;

expression: assignment | connect;

assignment:
	dotted_name index* EQ expression
	| inc_assign
	| copy_assign;

inc_assign:
	dotted_name index* (PEQ | MEQ | TEQ | DEQ) expression;

copy_assign: dotted_name index* CPY_EQ expression;

connect: logical ( (NOT)? edge_ref expression)?;

logical: compare ((KW_AND | KW_OR) compare)*;

compare:
	NOT compare
	| arithmetic (
		(EE | LT | GT | LTE | GTE | NE | KW_IN | nin) arithmetic
	)*;

nin: NOT KW_IN;

arithmetic: term ((PLUS | MINUS) term)*;

term: factor ((MUL | DIV | MOD) factor)*;

factor: (PLUS | MINUS) factor | power;

power: func_call (POW factor)* | func_call index+;

func_call:
	atom (LPAREN (expression (COMMA expression)*)? RPAREN)?
	| atom DOT func_built_in
	| atom? DBL_COLON NAME spawn_ctx?;

```

```
func_built_in:
        | KW_LENGTH
        | KW_KEYS
        | KW_EDGE
        | KW_NODE
        | KW_DESTROY LPAREN expression RPAREN;

atom:
        INT
        | FLOAT
        | STRING
        | BOOL
        | node_edge_ref
        | list_val
        | dict_val
        | dotted_name
        | LPAREN expression RPAREN
        | spawn
        | DEREF expression;

node_edge_ref: node_ref | edge_ref (node_ref)?;

node_ref: KW_NODE DBL_COLON NAME;

walker_ref: KW_WALKER DBL_COLON NAME;

graph_ref: KW_GRAPH DBL_COLON NAME;

edge_ref: edge_to | edge_from | edge_any;

edge_to: '-->' | '-' ('[' NAME ']')? '->';

edge_from: '<--' | '<-' ('[' NAME ']')? '-';

edge_any: '<-->' | '<-' ('[' NAME ']')? '->';

list_val: LSQUARE (expression (COMMA expression)*)? RSQUARE;

index: LSQUARE expression RSQUARE;

dict_val: LBRACE (kv_pair (COMMA kv_pair)*)? RBRACE;

kv_pair: STRING COLON expression;

spawn: KW_SPAWN expression? spawn_object;

spawn_object: node_spawn | walker_spawn | graph_spawn;

node_spawn: edge_ref? node_ref spawn_ctx?;

graph_spawn: edge_ref graph_ref;

walker_spawn: walker_ref spawn_ctx?;

spawn_ctx: LPAREN (assignment (COMMA assignment)*)? RPAREN;
```

```

/* DOT grammar below */
dot_graph:
       KW_STRICT? (KW_GRAPH | KW_DIGRAPH) dot_id? '{' dot_stmt_list '}';

dot_stmt_list: ( dot_stmt ';'?)*;

dot_stmt:
       dot_node_stmt
       | dot_edge_stmt
       | dot_attr_stmt
       | dot_id '=' dot_id
       | dot_subgraph;

dot_attr_stmt: ( KW_GRAPH | KW_NODE | KW_EDGE) dot_attr_list;

dot_attr_list: ( '[' dot_a_list? ']')+;

dot_a_list: ( dot_id ( '=' dot_id)? ','?)+;

dot_edge_stmt: (dot_node_id | dot_subgraph) dot_edgeRHS dot_attr_list?;

dot_edgeRHS: ( dot_edgeop ( dot_node_id | dot_subgraph))+;

dot_edgeop: '->' | '--';

dot_node_stmt: dot_node_id dot_attr_list?;

dot_node_id: dot_id dot_port?;

dot_port: ':' dot_id ( ':' dot_id)?;

dot_subgraph: ( KW_SUBGRAPH dot_id?)? '{' dot_stmt_list '}';

dot_id:
       NAME
       | STRING
       | INT
       | FLOAT
       | KW_GRAPH
       | KW_NODE
       | KW_EDGE;

/* Lexer rules */
KW_GRAPH: 'graph';
KW_STRICT: 'strict';
KW_DIGRAPH: 'digraph';
KW_SUBGRAPH: 'subgraph';
KW_NODE: 'node';
KW_IGNORE: 'ignore';
KW_TAKE: 'take';
KW_SPAWN: 'spawn';
KW_WITH: 'with';
KW_ENTRY: 'entry';
KW_EXIT: 'exit';
```

```
KW_LENGTH: 'length';
KW_KEYS: 'keys';
KW_ACTIVITY: 'activity';
COLON: ':';
DBL_COLON: '::';
COLON_OUT: '::>';
LBRACE: '{';
RBRACE: '}';
KW_EDGE: 'edge';
KW_WALKER: 'walker';
SEMI: ';';
EQ: '=';
PEQ: '+=';
MEQ: '-=';
TEQ: '*=';
DEQ: '/=';
CPY_EQ: ':=';
KW_AND: 'and' | '&&';
KW_OR: 'or' | '||';
KW_IF: 'if';
KW_ELIF: 'elif';
KW_ELSE: 'else';
KW_FOR: 'for';
KW_TO: 'to';
KW_BY: 'by';
KW_WHILE: 'while';
KW_CONTINUE: 'continue';
KW_BREAK: 'break';
KW_DISENGAGE: 'disengage';
KW_SKIP: 'skip';
KW_REPORT: 'report';
KW_DESTROY: 'destroy';
DEREF: '&';
DOT: '.';
NOT: '!' | 'not';
EE: '==';
LT: '<';
GT: '>';
LTE: '<=';
GTE: '>=';
NE: '!=';
KW_IN: 'in';
KW_ANCHOR: 'anchor';
KW_HAS: 'has';
KW_PRIVATE: 'private';
COMMA: ',';
KW_CAN: 'can';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
POW: '^';
LPAREN: '(';
RPAREN: ')';
```

```
LSQUARE: '[';
RSQUARE: ']';
FLOAT: ([0-9]+)? '.' [0-9]+;
STRING: '"' ~ ["\r\n]* '"' | '\'' ~ ['\r\n]* '\'';
BOOL: 'true' | 'false';
INT: [0-9]+;
NAME: [a-zA-Z_] [a-zA-Z0-9_]*;
COMMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
PY_COMMENT: '#' ~[\r\n]* -> skip;
WS: [ \t\r\n] -> skip;
ErrorChar: .;
```

Appendix C

Help for ci_edit (built into jsctl)

C.1 Quick Tips (How to quit or exit)

To exit the program: hold the control key and press q. A shorthand for “hold the control key and press q” is ctrl+q.

Bibliography

[1] Wikimedia Commons. File:baby in wikimedia foundation "hello world" onesie.jpg — wikimedia commons, the free media repository, 2020. [Online; accessed 29-July-2021].

[2] Wikimedia Commons. File:directed graph no background.svg — wikimedia commons, the free media repository, 2020. [Online; accessed 13-July-2021].

[3] Wikimedia Commons. File:multi-pseudograph.svg — wikimedia commons, the free media repository, 2020. [Online; accessed 9-July-2021].

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

[5] Django Software Foundation. Django.

[6] J. K. Rowling. *Harry Potter and the Philosopher's Stone*, volume 1. Bloomsbury Publishing, London, 1 edition, June 1997.