JASECI

BIBLE

Jason Mars, ~~PhD~~ Ninja

v1.3

I welcome you, neophyte, to embark on the journey of becoming a true Jaseci Ninja!

Btw, this is a silly book, please take that seriously :-)

Contents

Not So Technical Terms Used

bleh mildly yucky. 76

christen to name or dedicate (something, such as a piece of code) by a ceremony that often involves breaking a bottle of champagne. 74

coder the superior human. 82

common languages typical languages programmers use to write commercial software, (e.g., C, C++, Java, Javascript, Python, Ruby, Go, Perl, PHP, etc.). 14

dope sick. 8

gobbledygook language that is meaningless or is made unintelligible by excessive use of abstruse technical terms; nonsense. 28

goo goo gaa gaa the language of babies. 75

grok to fully comprehend and understand deeply . 73

haxor leet spelling of hacker. 8, 20, 31

IMHO Acronym for "In My Humble Opinion". 153

Jaseci jolt an insight derived from Jaseci that serves as a high voltage bolt of energy to the mind of a sharp coder.. 82

leet v. hyper-sophisticated from a coding perspective, n. a language used by leet haxors. 8, 20

redonkulous dope. 8

sick redonkulous. 8, 73

Technical Terms Used

contexts A set of key value pairings that serve as a data payload attributable to nodes and edges in Jaseci graphs. 16

directed graphs a directed graph (or digraph) is a graph that is made up of a set of vertices connected by directed edges (think edges as arrows as opposed to lines) often called arcs. 15

dynamically typed language a language is dynamically typed if the type is associated with run-time values, and not named variables/fields/etc. This means that a programmer can code a bit quicker not having to specify types statically.. 155

hypergraph a hypergraph is a generalization of a graph in which an edge can join any number of vertices in contrast to connecting exactly two vertices. 15

multigraph a graph which is permitted to have multiple edges (also called parallel edges[1]), that is, edges that have the same end nodes. 15

sentinel Overseer of walkers and architype nodes and edges.. 28

undirected graphs a graph made up of vertices (also called nodes or points) which are connected symmetrically by edges (also called links or lines). 15

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. 82

WSL Windows Subsystem for Linux. 153

Preface

The way we design and write software to do computation and AI today is poop. How poopy you ask? Hrm..., let me think..., In my approximation, if you were to use it as a fuel source, it would be able to run all the blockchain transactions across the aggregate of current and future coins for a decade.

Hrm, too much? Probably. I guess you'd expect me to use sophisticated rhetoric and cite evidence to make my points. 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..."*. I'd certainly sound more credible perhaps. Well, though I have indeed grown accustomed to writing that way, boy has it gotten old.

I'm not going to do that for this book. Let's have fun. After all, Jaseci has always been more play (and art) than work. Very ambitious play granted, but play at it's core.

Everything here is based on my opinion...no, expert *ninja* opinion, and my intuition. That suffices for me, and I hope it does for you. Though I have spent decades coding and leading teams of coders and computer scientists working on the holy grail technical challenges of our time, I won't rely on that to assert credibility. ...(o_o)... Lets let these ideas stand or die on their 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

Coming soon!

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

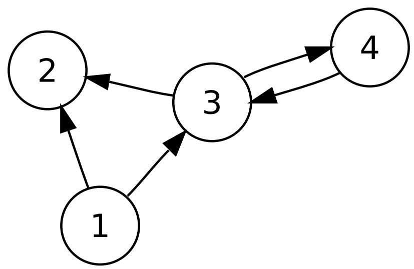
Abstrations of Jaseci

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

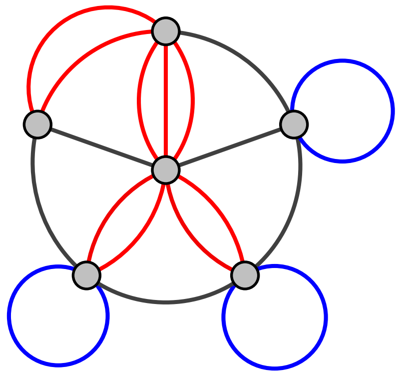
There's something quite strange that has happend with our common languages over the years, ...decades. When you look at it, almost every data structure we programmers use to solve problems can be modeled formally as a graph, or a special case of a graph, (save perhaps hash tables). Think about it, stacks, 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 ssllooowww 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 rich language symantics that correspond to the rich semantics graphs offer (You'll see what I mean in later chapters).

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 higher 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 with 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 Actions

3.5 Other Abstractions Not Yet Actualized in Current Jaseci

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.

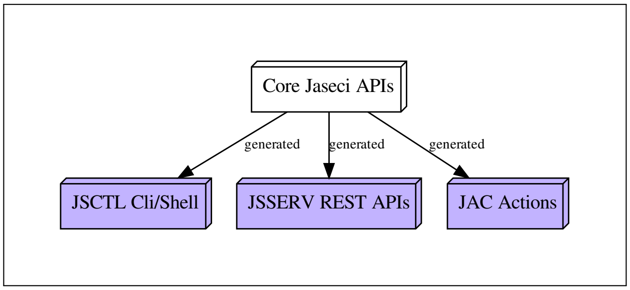


Figure 5.1: Jaseci Interface Architecture

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 B.2 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

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

dig into this and tons more in the Part IV, so we'll leave the discussion on implementation 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

```
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. 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`.

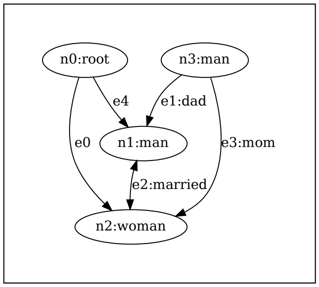


Figure 5.2: Graph for `fam.jac`

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

5.2.1 API Parameter Cheatsheet

Interface	**Parameters**
`walker callback`	`nd: node (*req), wlk: walker (*req), key: str (*req),` `↪ ctx: dict, _req_ctx: dict, global_sync: bool`
`walker summon`	`key: str (*req), wlk: walker (*req), nd: node (*req),` `↪ ctx: dict, _req_ctx: dict, global_sync: bool`
`walker register`	`snt: sentinel, code: str, encoded: bool`
`walker get`	`wlk: walker (*req), mode: str, detailed: bool`
`walker set`	`wlk: walker (*req), code: str (*req), mode: str`
`walker list`	`snt: sentinel, detailed: bool`
`walker delete`	`wlk: walker (*req), snt: sentinel`
`walker spawn create`	`name: str (*req), snt: sentinel`
`walker spawn delete`	`name: str (*req)`
`walker spawn list`	`detailed: bool`
`walker prime`	`wlk: walker (*req), nd: node, ctx: dict, _req_ctx:` `↪ dict`
`walker execute`	`wlk: walker (*req), prime: node, ctx: dict, _req_ctx:` `↪ dict, profiling: bool`

walker run	name: str (*req), nd: node, ctx: dict, _req_ctx: dict, ↪ snt: sentinel, profiling: bool
user create	name: str (*req), global_init: str, global_init_ctx: ↪ dict, other_fields: dict
alias register	name: str (*req), value: str (*req)
alias list	n/a
alias delete	name: str (*req)
alias clear	n/a
global get	name: str (*req)
global set	name: str (*req), value: str (*req)
global delete	name: str (*req)
global sentinel set	snt: sentinel
global sentinel unset	n/a
object get	obj: element (*req), depth: int, detailed: bool
object perms get	obj: element (*req)
object perms set	obj: element (*req), mode: str (*req)
object perms default	mode: str (*req)
object perms grant	obj: element (*req), mast: element (*req), read_only: ↪ bool
object perms revoke	obj: element (*req), mast: element (*req)
graph create	set_active: bool
graph get	gph: graph, mode: str, detailed: bool
graph list	detailed: bool
graph active set	gph: graph (*req)
graph active unset	n/a
graph active get	detailed: bool
graph delete	gph: graph (*req)
graph node get	nd: node (*req), ctx: list
graph node set	nd: node (*req), ctx: dict (*req), snt: sentinel
graph walk (cli only)	nd: node
sentinel register	name: str, code: str, code_dir: str, mode: str, ↪ encoded: bool, auto_run: str, auto_run_ctx: dict ↪ , auto_create_graph: bool, set_active: bool
sentinel pull	set_active: bool, on_demand: bool
sentinel get	snt: sentinel, mode: str, detailed: bool
sentinel set	code: str (*req), code_dir: str, encoded: bool, snt: ↪ sentinel, mode: str
sentinel list	detailed: bool
sentinel test	snt: sentinel, detailed: bool
sentinel active set	snt: sentinel (*req)
sentinel active unset	n/a

`sentinel active global`	`auto_run: str, auto_run_ctx: dict, auto_create_graph: bool, detailed: bool`
`sentinel active get`	`detailed: bool`
`sentinel delete`	`snt: sentinel (*req)`
`wapi`	`name: str (*req), nd: node, ctx: dict, _req_ctx: dict, snt: sentinel, profiling: bool`
`architype register`	`code: str (*req), encoded: bool, snt: sentinel`
`architype get`	`arch: architype (*req), mode: str, detailed: bool`
`architype set`	`arch: architype (*req), code: str (*req), mode: str`
`architype list`	`snt: sentinel, detailed: bool`
`architype delete`	`arch: architype (*req), snt: sentinel`
`master create`	`name: str (*req), global_init: str, global_init_ctx: dict, other_fields: dict`
`master get`	`name: str (*req), mode: str, detailed: bool`
`master list`	`detailed: bool`
`master active set`	`name: str (*req)`
`master active unset`	`n/a`
`master active get`	`detailed: bool`
`master self`	`detailed: bool`
`master delete`	`name: str (*req)`
`master createsuper`	`name: str (*req), global_init: str, global_init_ctx: dict, other_fields: dict`
`master allusers`	`num: int, start_idx: int`
`master become`	`mast: master (*req)`
`master unbecome`	`n/a`
`config get`	`name: str (*req), do_check: bool`
`config set`	`name: str (*req), value: str (*req), do_check: bool`
`config list`	`n/a`
`config index`	`n/a`
`config exists`	`name: str (*req)`
`config delete`	`name: str (*req), do_check: bool`
`logger http connect`	`host: str (*req), port: int (*req), url: str (*req), log: str`
`logger http clear`	`log: str`
`logger list`	`n/a`
`actions load local`	`file: str (*req)`
`actions load remote`	`url: str (*req)`
`actions load module`	`mod: str (*req)`
`actions list`	`name: str`
`jac build (cli only)`	`file: str (*req), out: str`
`jac test (cli only)`	`file: str (*req), detailed: bool`

jac run (cli only)	file: str (*req), walk: str, ctx: dict, profiling: ↪ bool
jac dot (cli only)	file: str (*req), walk: str, ctx: dict, detailed: bool

Table 5.1: Full set of core Jaseci APIs

5.3 Full Spec of Jaseci Core APIs

5.3.1 APIs for actions

This set action APIs enable the manual management of Jaseci actions and action libraries/sets. Action libraries can be loaded locally into the running instance of the python program, or as a remote container linked action library. In this mode, action libraries operate as micro-services. Jaseci will be able to dynamically and automatically make this decision for the user based on online monitoring and performance profiling.

5.3.1.1 `actions load local`

`cli: actions load local | api: actions_load_local | auth: admin`

args: `file: str (*req)`

This API will dynamically load a module based on a python file. The module is loaded directly into the running Jaseci python instance. This API also makes an attempt to auto detect and hot load any python package dependencies the file may reference via python's relative imports. This file is assumed to have the necessary annotations and decorations required by Jaseci to recognize its actions.

Parameters

`file` – The python file with full to load actions from. (i.e., /local/myact.py)

5.3.1.2 actions load remote

`cli: actions load remote | api: actions_load_remote | auth: admin`

args: url: str (*req)

This API will dynamically load a set of actions that are present on a remote server/micro-service. This server must be configured to interact with Jaseci properly. This is easily achieved using the same decorators used for local action libraries. Remote actions allow for higher flexibility in the languages supported for action libraries. If an library writer would like to use another language, the main hook REST api simply needs to be implemented. Please refer to documentation on creating action libraries for more details.

Parameters

`url` – The url of the API server supporting Jaseci actions.

5.3.1.3 actions load module

`cli: actions load module | api: actions_load_module | auth: admin`

args: mod: str (*req)

This API will dynamically load a module using python's module import format. This is particularly useful for pip installed action libraries as the developer can directly reference the module using the same format as a regular python import. As with load local, the module will be loaded directly into the running Jaseci python instance.

Parameters

`mod` – The import style module to load actions from. (i.e., jaseci_kit.bi_enc)

5.3.1.4 `actions list`

`cli: actions list | api: actions_list | auth: admin`

args: `name: str`

This API is used to list the loaded actions active in Jaseci. These actions include all types of loaded actions whether it be local modules or remote containers. A particular set of actions can be viewed using the name parameter.

Parameters

`name` – The name for a library for which to filter the view of shown actions. If left blank all actions from all loaded sets will be shown.

5.3.2 APIs for alias

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 register`

`cli: alias register | api: alias_register | auth: user`

`args: name: str (*req), value: str (*req)`

Either create new alias string to string mappings or replace an existing mappings of a given alias name. Once registered the alias mapping is instantly active.

Parameters

`name` – The name for the alias created by caller.

`value` – The value for that name to map to (i.e., UUID)

5.3.2.2 `alias list`

`cli: alias list | api: alias_list | auth: user`

`args: n/a`

Returns dictionary object of name to value mappings currently active. This API is quite useful to track not only the aliases the caller creates, but also the aliases automatically created as various Jaseci objects (walkers, architypes, sentinels, etc.) are created, changed, or destroyed.

5.3.2.3 `alias delete`

`cli: alias delete | api: alias_delete | auth: user`

args: name: str (*req)

Removes a specific alias by its name. Only the alias is removed no actual objects are affected. Future uses of this name will not be mapped.

Parameters

`name` – The name for the alias to be removed from caller.

5.3.2.4 `alias clear`

`cli: alias clear | api: alias_clear | auth: user`

args: n/a

Removes a all aliases. No actual objects are affected. Aliases will continue to be automatically generated when creating other Jaseci objects.

5.3.3 APIs for architype

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 register`

`cli: architype register | api: architype_register | auth: user`

args: `code: str (*req), encoded: bool, snt: sentinel`

This register API allows for the creation or replacement/update of an architype that can then be used by walkers in their interactions of graphs. The code argument takes Jac source code for the single architype. To load multiple architypes and walkers at the same time, use sentinel register API.

Parameters

`code` – The text (or filename) for an architypes Jac code

`encoded` – True/False flag as to whether code is encode in base64

`snt` – The UUID of the sentinel to be the owner of this architype

5.3.3.2 `architype get`

`cli: architype get | api: architype_get | auth: user`

args: `arch: architype (*req), mode: str, detailed: bool`

No documentation yet.

Parameters

`arch` – The architype being accessed

`mode` – Valid modes: default, code, ir,

`detailed` – Flag to give summary or complete set of fields

5.3.3.3 architype set

`cli: architype set | api: architype_set | auth: user`

args: arch: architype (*req), code: str (*req), mode: str

No documentation yet.

Parameters

- `arch` – The architype being set
- `code` – The text (or filename) for an architypes Jac code/ir
- `mode` – Valid modes: default, code, ir,

5.3.3.4 architype list

`cli: architype list | api: architype_list | auth: user`

args: snt: sentinel, detailed: bool

No documentation yet.

Parameters

- `snt` – The sentinel for which to list its architypes
- `detailed` – Flag to give summary or complete set of fields

5.3.3.5 `architype delete`

`cli: architype delete | api: architype_delete | auth: user`

args: `arch: architype (*req), snt: sentinel`

No documentation yet.

Parameters

`arch` – The architype being set

`snt` – The sentinel for which to list its architypes

5.3.4 APIs for config

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

5.3.4.1 `config get`

`cli: config get | api: config_get | auth: admin`

args: `name: str (*req), do_check: bool`

No documentation yet.

5.3.4.2 config set

cli: config set | api: config_set | auth: admin
args: name: str (*req), value: str (*req), do_check: bool
No documentation yet.

5.3.4.3 config list

cli: config list | api: config_list | auth: admin
args: n/a
No documentation yet.

5.3.4.4 config index

cli: config index | api: config_index | auth: admin
args: n/a
No documentation yet.

5.3.4.5 `config exists`

`cli: config exists | api: config_exists | auth: admin`
args: `name: str (*req)`
No documentation yet.

5.3.4.6 `config delete`

`cli: config delete | api: config_delete | auth: admin`
args: `name: str (*req), do_check: bool`
No documentation yet.

5.3.5 APIs for global

No documentation yet.

5.3.5.1 `global set`

`cli: global set | api: global_set | auth: admin`
args: `name: str (*req), value: str (*req)`
No documentation yet.

5.3.5.2 global delete

```
cli: global delete | api: global_delete | auth: admin
```

args: name: str (*req)

No documentation yet.

5.3.5.3 global sentinel set

```
cli: global sentinel set | api: global_sentinel_set | auth: admin
```

args: snt: sentinel

No documentation yet.

5.3.5.4 global sentinel unset

```
cli: global sentinel unset | api: global_sentinel_unset | auth:
admin
```

args: n/a

No documentation yet.

5.3.6 APIs for graph

No documentation yet.

5.3.6.1 graph create

cli: graph create | api: graph_create | auth: user
args: set_active: bool
No documentation yet.

5.3.6.2 graph get

cli: graph get | api: graph_get | auth: user
args: gph: graph, mode: str, detailed: bool
Valid modes: default, dot,

5.3.6.3 graph list

cli: graph list | api: graph_list | auth: user
args: detailed: bool
No documentation yet.

5.3.6.4 graph active set

cli: graph active set | api: graph_active_set | auth: user
args: gph: graph (*req)
No documentation yet.

5.3.6.5 graph active unset

cli: graph active unset | api: graph_active_unset | auth: user
args: n/a
No documentation yet.

5.3.6.6 graph active get

cli: graph active get | api: graph_active_get | auth: user
args: detailed: bool
No documentation yet.

5.3.6.7 `graph delete`

cli: `graph delete` | api: `graph_delete` | auth: `user`
args: `gph: graph (*req)`
No documentation yet.

5.3.6.8 `graph node get`

cli: `graph node get` | api: `graph_node_get` | auth: `user`
args: `nd: node (*req), ctx: list`
No documentation yet.

5.3.6.9 `graph node set`

cli: `graph node set` | api: `graph_node_set` | auth: `user`
args: `nd: node (*req), ctx: dict (*req), snt: sentinel`
No documentation yet.

5.3.6.10 graph walk

cli: graph walk (cli only)
args: nd: node
No documentation yet.

5.3.7 APIs for jac

No documentation yet.

5.3.7.1 jac build

cli: jac build (cli only)
args: file: str (*req), out: str
No documentation yet.

5.3.7.2 jac test

cli: jac test (cli only)
args: file: str (*req), detailed: bool
and .jir executables

5.3.7.3 jac run

cli: jac run (cli only)
args: file: str (*req), walk: str, ctx: dict, profiling: bool
and .jir executables

5.3.7.4 jac dot

cli: jac dot (cli only)
args: file: str (*req), walk: str, ctx: dict, detailed: bool
files and .jir executables

5.3.8 APIs for logger

No documentation yet.

5.3.8.1 logger http connect

cli: logger http connect | api: logger_http_connect | auth: admin
args: host: str (*req), port: int (*req), url: str (*req), log: str
Valid log params: sys, app, all

5.3.8.2 logger http clear

cli: logger http clear | api: logger_http_clear | auth: admin
args: log: str
Valid log params: sys, app, all

5.3.8.3 logger list

cli: logger list | api: logger_list | auth: admin
args: n/a
No documentation yet.

5.3.9 APIs for master

These APIs

5.3.9.1 master create

cli: master create | api: master_create | auth: user
args: name: str (*req), global_init: str, global_init_ctx: dict, ↪ other_fields: dict
other fields used for additional feilds for overloaded interfaces (i.e., Dango interface)

5.3.9.2 master get

cli: master get | api: master_get | auth: user
args: name: str (*req), mode: str, detailed: bool
Valid modes: default,

5.3.9.3 master list

cli: master list | api: master_list | auth: user
args: detailed: bool
No documentation yet.

5.3.9.4 master active set

cli: master active set | api: master_active_set | auth: user
args: name: str (*req)
NOTE: Specail handler included in general interface to api

5.3.9.5 `master active unset`

cli: master active unset | api: master_active_unset | auth: user
args: n/a
No documentation yet.

5.3.9.6 `master active get`

cli: master active get | api: master_active_get | auth: user
args: detailed: bool
No documentation yet.

5.3.9.7 `master self`

cli: master self | api: master_self | auth: user
args: detailed: bool
No documentation yet.

5.3.9.8 `master delete`

cli: master delete | api: master_delete | auth: user
args: name: str (*req)
No documentation yet.

5.3.10 APIs for object

...

5.3.10.1 `global get`

cli: global get | api: global_get | auth: user
args: name: str (*req)
No documentation yet.

5.3.10.2 `object get`

cli: object get | api: object_get | auth: user
args: obj: element (*req), depth: int, detailed: bool
No documentation yet.

5.3.10.3 `object perms get`

cli: `object perms get` | api: `object_perms_get` | auth: `user`
args: `obj`: `element` (*req)
No documentation yet.

5.3.10.4 `object perms set`

cli: `object perms set` | api: `object_perms_set` | auth: `user`
args: `obj`: `element` (*req), `mode`: `str` (*req)
No documentation yet.

5.3.10.5 `object perms default`

cli: `object perms default` | api: `object_perms_default` | auth: `user`
args: `mode`: `str` (*req)
No documentation yet.

5.3.10.6 `object perms grant`

cli: object perms grant | api: object_perms_grant | auth: user
args: obj: element (*req), mast: element (*req), read_only: bool
No documentation yet.

5.3.10.7 `object perms revoke`

cli: object perms revoke | api: object_perms_revoke | auth: user
args: obj: element (*req), mast: element (*req)
No documentation yet.

5.3.11 APIs for sentinel

No documentation yet.

5.3.11.1 sentinel register

cli: sentinel register | api: sentinel_register | auth: user

```
args:    name: str, code: str, code_dir: str, mode: str, encoded: bool
↪ , auto_run: str, auto_run_ctx: dict, auto_create_graph: bool,
↪ set_active: bool
```

Auto run is the walker to execute on register (assumes active graph is selected)

5.3.11.2 sentinel pull

cli: sentinel pull | api: sentinel_pull | auth: user

```
args: set_active: bool, on_demand: bool
```

No documentation yet.

5.3.11.3 sentinel get

cli: sentinel get | api: sentinel_get | auth: user

```
args: snt: sentinel, mode: str, detailed: bool
```

Valid modes: default, code, ir,

5.3.11.4 sentinel set

cli: sentinel set | api: sentinel_set | auth: user
args: code: str (*req), code_dir: str, encoded: bool, snt: sentinel, ↪ mode: str
Valid modes: code, ir,

5.3.11.5 sentinel list

cli: sentinel list | api: sentinel_list | auth: user
args: detailed: bool
No documentation yet.

5.3.11.6 sentinel test

cli: sentinel test | api: sentinel_test | auth: user
args: snt: sentinel, detailed: bool
No documentation yet.

5.3.11.7 `sentinel active set`

`cli: sentinel active set | api: sentinel_active_set | auth: user`
args: `snt: sentinel (*req)`
No documentation yet.

5.3.11.8 `sentinel active unset`

`cli: sentinel active unset | api: sentinel_active_unset | auth: user`
args: `n/a`
No documentation yet.

5.3.11.9 `sentinel active global`

`cli: sentinel active global | api: sentinel_active_global | auth: user`
args: `auto_run: str, auto_create_graph: bool, ↪ detailed: bool`
Exclusive OR with pull strategy

5.3.11.10 sentinel active get

cli: sentinel active get | api: sentinel_active_get | auth: user
args: detailed: bool
No documentation yet.

5.3.11.11 sentinel delete

cli: sentinel delete | api: sentinel_delete | auth: user
args: snt: sentinel (*req)
No documentation yet.

5.3.12 APIs for stripe

Set of APIs to expose jaseci stripe management

5.3.12.1 stripe product create

cli: stripe product create | api: stripe_product_create | auth: admin
args: name: str, description: str
No documentation yet.

5.3.12.2 stripe product price set

cli: stripe product price set | api: stripe_product_price_set | auth: admin

args: productId: str (*req), amount: float, interval: str

No documentation yet.

5.3.12.3 stripe product list

cli: stripe product list | api: stripe_product_list | auth: admin

args: detalied: bool

No documentation yet.

5.3.12.4 stripe customer create

cli: stripe customer create | api: stripe_customer_create | auth: admin

args: paymentId: str (*req), name: str, email: str, description: str

No documentation yet.

5.3.12.5 stripe customer get

cli: stripe customer get | api: stripe_customer_get | auth: admin

args: customerId: str (*req)

No documentation yet.

5.3.12.6 stripe customer payment add

cli: stripe customer payment add | api: stripe_customer_payment_add | auth: admin

args: paymentMethodId: str (*req), customerId: str (*req)

No documentation yet.

5.3.12.7 stripe customer payment delete

cli: stripe customer payment delete | api: stripe_customer_payment_delete | auth: admin

args: paymentMethodId: str (*req)

No documentation yet.

5.3.12.8 stripe customer payment get

cli: stripe customer payment get | api: stripe_customer_payment_get | auth: admin

args: customerId: str (*req)

No documentation yet.

5.3.12.9 stripe customer payment default

cli: stripe customer payment default | api: stripe_customer_payment_default | auth: admin

args: customerId: str (*req), paymentMethodId: str (*req)

No documentation yet.

5.3.12.10 stripe subscription create

cli: stripe subscription create | api: stripe_subscription_create | auth: admin

args: paymentId: str (*req), name: str (*req), email: str (*req), ↪ priceId: str (*req), customerId: str (*req)

TODO: name and email parameters not used!

5.3.12.11 stripe subscription delete

```
cli: stripe subscription delete | api: stripe_subscription_delete |
auth: admin
```

```
args: subscriptionId: str (*req)
```

No documentation yet.

5.3.12.12 stripe subscription get

```
cli: stripe subscription get | api: stripe_subscription_get | auth:
admin
```

```
args: customerId: str (*req)
```

No documentation yet.

5.3.12.13 stripe invoices list

```
cli: stripe invoices list | api: stripe_invoices_list | auth: admin
```

```
args: customerId: str (*req), subscriptionId: str (*req), limit: int,
↪ lastItem: str
```

No documentation yet.

5.3.13 APIs for super

No documentation yet.

5.3.13.1 master createsuper

cli: master createsuper | api: master_createsuper | auth: admin
args: name: str (*req), global_init: str, global_init_ctx: dict, ↪ other_fields: dict
other fields used for additional feilds for overloaded interfaces (i.e., Dango interface)

5.3.13.2 master allusers

cli: master allusers | api: master_allusers | auth: admin
args: num: int, start_idx: int
return and start idx specfies where to start NOTE: Abstract interface to be overridden

5.3.13.3 master become

cli: master become | api: master_become | auth: admin
args: mast: master (*req)
No documentation yet.

5.3.13.4 `master unbecome`

`cli: master unbecome | api: master_unbecome | auth: admin`
`args: n/a`
No documentation yet.

5.3.14 APIs for user

These User APIs enable the creation and management of users on a Jaseci machine. The creation of a user in this context is synonymous to the creation of a master Jaseci object. These APIs are particularly useful when running a Jaseci server or cluster in contrast to running JSCTL on the command line. Upon executing JSCTL a dummy admin user (super_master) is created and all state is dumped to a session file, though any users created during a JSCTL session will indeed be created as part of that session's state.

5.3.14.1 user create

cli: user create | api: user_create | auth: public

```
args:     name: str (*req), global_init: str, global_init_ctx: dict,
↪ other_fields: dict
```

This API is used to create users and optionally set them up with a graph and related initialization. In the context of JSCTL, any name is sufficient and no additional information is required. However, for Jaseci serving (whether it be the official Jaseci server, or a custom overloaded server) additional fields are required and should be added to the other fields parameter as per the specifics of the encapsulating server requirements. In the case of the official Jaseci server, the name field must be a valid email, and a password field must be passed through other fields. A number of other optional parameters can also be passed through other feilds.

This single API call can also be used to fully set up and initialize a user by leveraging the global init parameter. When set, this parameter attaches the user to the global sentinel, creates a new graph for the user, sets it as the active graph, then runs an initialization walker on the root node of this new graph. The initialization walker is identified by the name assigned to global init. The default empty string assigned to global init indicates this global setup should not be run.

Parameters

`name` – The user name to create. For Jaseci server this must be a valid email address.

`global_init` – The name of an initialization walker. When set the user is linked to the global sentinel and the walker is run on a new active graph created for the user.

`global_init_ctx` – Context to preload for the initialization walker

`other_fields` – This parameter is used for additional fields required for overloaded interfaces. This parameter is not used in JSCTL, but is used by Jaseci server for the additional parameters of password, is_activated, and is_superuser.

5.3.15 APIs for walker

No documentation yet.

5.3.15.1 walker callback

```
cli: walker callback | api: walker_callback | auth: public
```

```
args: nd: node (*req), wlk: walker (*req), key: str (*req), ctx: dict,
↪ _req_ctx: dict, global_sync: bool
```

along with the walker id and node id

5.3.15.2 walker summon

```
cli: walker summon | api: walker_summon | auth: public
```

```
args: key: str (*req), wlk: walker (*req), nd: node (*req), ctx: dict,
↪ _req_ctx: dict, global_sync: bool
```

along with the walker id and node id

5.3.15.3 walker register

cli: walker register | api: walker_register | auth: user
args: snt: sentinel, code: str, encoded: bool
No documentation yet.

5.3.15.4 walker get

cli: walker get | api: walker_get | auth: user
args: wlk: walker (*req), mode: str, detailed: bool
Valid modes: default, code, ir, keys,

5.3.15.5 walker set

cli: walker set | api: walker_set | auth: user
args: wlk: walker (*req), code: str (*req), mode: str
Valid modes: code, ir,

5.3.15.6 walker list

cli: walker list | api: walker_list | auth: user
args: snt: sentinel, detailed: bool
No documentation yet.

5.3.15.7 walker delete

cli: walker delete | api: walker_delete | auth: user
args: wlk: walker (*req), snt: sentinel
No documentation yet.

5.3.15.8 walker spawn create

cli: walker spawn create | api: walker_spawn_create | auth: user
args: name: str (*req), snt: sentinel
No documentation yet.

5.3.15.9 walker spawn delete

cli: walker spawn delete | api: walker_spawn_delete | auth: user

args: name: str (*req)

No documentation yet.

5.3.15.10 walker spawn list

cli: walker spawn list | api: walker_spawn_list | auth: user

args: detailed: bool

No documentation yet.

5.3.15.11 walker prime

cli: walker prime | api: walker_prime | auth: user

args: wlk: walker (*req), nd: node, ctx: dict, _req_ctx: dict

No documentation yet.

5.3.15.12 walker execute

cli: walker execute | api: walker_execute | auth: user

args: wlk: walker (*req), prime: node, ctx: dict, _req_ctx: dict,
↪ profiling: bool

No documentation yet.

5.3.15.13 walker run

cli: walker run | api: walker_run | auth: user

args: name: str (*req), nd: node, ctx: dict, _req_ctx: dict, snt:
↪ sentinel, profiling: bool

reports results, and cleans up walker instance.

5.3.15.14 wapi

cli: wapi | api: wapi | auth: user

args: name: str (*req), nd: node, ctx: dict, _req_ctx: dict, snt:
↪ sentinel, profiling: bool

No documentation yet.

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,

[1]Image credit to wiki contributer [1]

```
Hello World
```

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 B.1. 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)*;
```

(full grammar in Appendix C)

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 (cmp_op arithmetic)*;

cmp_op: EE | LT | GT | LTE | GTE | NE | KW_IN | nin;

nin: NOT KW_IN;
```

(full grammar in Appendix C)

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
```

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

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

```
expression: connect (assignment | copy_assign | inc_assign)?;

assignment: EQ expression;

copy_assign: CPY_EQ expression;

inc_assign: (PEQ | MEQ | TEQ | DEQ) expression;
```

(full grammar in Appendix C)

6.2.4 Precedence

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.

6.2.5 Primitive Types

Jac Code 6.10: Primitive types

```
walker init {
    a=5;
    std.out(a.type, '-', a);
    a=5.0;
    std.out(a.type, '-', a);
    a=true;
    std.out(a.type, '-', a);
    a=[5];
    std.out(a.type, '-', a);
    a='5';
    std.out(a.type, '-', a);
    a={'num': 5};
    std.out(a.type, '-', a);
}
```

```
JAC_TYPE.INT - 5
JAC_TYPE.FLOAT - 5.0
JAC_TYPE.BOOL - true
JAC_TYPE.LIST - [5]
JAC_TYPE.STR - 5
JAC_TYPE.DICT - {"num": 5}
```

6.2.5.1 Integers and Floats

6.2.5.2 Booleans

6.2.5.3 Lists and Strings

6.2.5.4 Dictionaries

6.2.5.5 Nodes and Edges

Jac Code 6.11: Basic arithmetic operations

```
walker init {
    nd = spawn here --> node::generic;
    std.out(nd.type, nd);
    std.out(nd.edge.type, nd.edge);
    std.out(nd.edge[0].type, nd.edge[0]);
}
```

```
JAC_TYPE.NODE jac:uuid:918900e4-9a35-4771-bce8-e1330d761bf6
JAC_TYPE.LIST ["jac:uuid:2930cfd6-7007-4942-b6ab-f28986819336"]
JAC_TYPE.EDGE jac:uuid:2930cfd6-7007-4942-b6ab-f28986819336
```

6.2.5.6 Specials

Jac Code 6.12: Basic arithmetic operations

```
walker init {
    a=null;
    std.out(a.type, '-', a);
    a=str;
    std.out(a.type, '-', a);
    std.out(null.type);
    std.out(null.type.type);
}
```

```
JAC_TYPE.NULL - null
JAC_TYPE.TYPE - JAC_TYPE.STR
JAC_TYPE.NULL
JAC_TYPE.TYPE
```

[Type type]

[Null]

6.2.5.7 Typecasting

Jac Code 6.13: Basic arithmetic operations

```
walker init {
```

```
    a=5.6;
    std.out(a+2);
    std.out((a+2).int);
    std.out((a+2).str);
    std.out((a+2).bool);
    std.out((a+2).int.float);

    if(a.str.type == str and !(a.int.type == str) and a.int.type == int):
        std.out("Types comes back correct");
}
```

```
7.6
7
7.6
true
7.0
Types comes back correct
```

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!

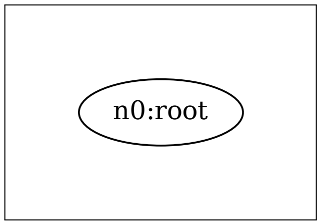


Figure 6.2: Graph in memory for simple Hello World program (JC 6.1)

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 6.14: 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 6.14 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.

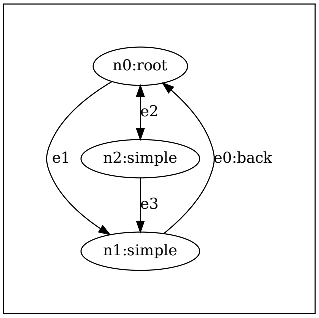


Figure 6.3: Graph in memory for JC 6.14

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 More on Strings, Lists, and Dictionaries

Jac Code 6.15: Built-in String Library

```
walker init {
    a="␣tEsting␣me␣␣";
    report a[4];
    report a[4:7];
    report a[3:-1];
    report a.str::upper;
    report a.str::lower;
    report a.str::title;
    report a.str::capitalize;
    report a.str::swap_case;
    report a.str::is_alnum;
    report a.str::is_alpha;
    report a.str::is_digit;
    report a.str::is_title;
    report a.str::is_upper;
    report a.str::is_lower;
    report a.str::is_space;
    report ’{"a":␣5}’.str::load_json;
    report a.str::count(’t’);
    report a.str::find(’i’);
    report a.str::split;
    report a.str::split(’E’);
    report a.str::startswith(’tEs’);
    report a.str::endswith(’me’);
    report a.str::replace(’me’, ’you’);
    report a.str::strip;
    report a.str::strip(’␣t’);
    report a.str::lstrip;
    report a.str::lstrip(’␣tE’);
    report a.str::rstrip;
    report a.str::rstrip(’␣e’);

    report a.str::upper.str::is_upper;
}
```

```
{
  "success": true,
  "report": [
    "t",
    "tin",
    "sting me ",
    " TESTING ME ",
    " testing me ",
    " Testing Me ",
    " testing me ",
    " TeSTING ME ",
    false,
    false,
    false,
    false,
    false,
    false,
    false,
    2,
    5,
    [
      "tEsting",
      "me"
    ],
    [
      " t",
      "sting me "
    ],
    false,
    false,
    " tEsting you ",
    "tEsting me",
    "Esting me",
    "tEsting me ",
    "sting me ",
    " tEsting me",
    " tEsting m",
    true
  ]
}
```

Op	Args	Description
.str::upper	none	
.str::lower	none	
.str::title	none	
.str::capitalize	none	
.str::swap_case	none	
.str::is_alnum	none	
.str::is_digit	none	
.str::is_title	none	
.str::is_upper	none	
.str::is_lower	none	
.str::is_space	none	
.str::load_json	none	
.str::count	(**substr**, start, end)	Returns the number of occurrences of a substring in the given string. Start and end specify range of indices to search
.str::find	(**substr**, start, end)	Returns the index of first occurrence of the substring (if found). If not found, it returns -1. Start and end specify range of indices to search.
.str::split	*optional* (separator, maxsplit)	Breaks up a string at the specified separator for maxsplit number of times and returns a list of strings. Default separators is ‘ ’ and maxsplit is unlimited.
.str::startswith		
.str::endswith		
.str::replace		
.str::strip	optional,	
.str::lstrip	optional,	
.str::rstrip	optional,	

Table 6.2: String operations in Jac

6.4.1 Library of String Operations

6.4.2 Library of List Operations

6.4.3 Library of Dictionary Operations

6.5 Control Flow

Jac Code 6.16: if statement

```
walker init {
    a = 4; b = 5;
    if(a < b): std.out("Hello!");
}
```

Op	Args	Description
.list::max	none	
.list::min	none	
.list::idx_of_max	none	
.list::idx_of_min	none	
.list::copy	none	Returns a shallow copy of the list
.list::deepcopy	none	Returns a deep copy of the list
.list::sort	none	
.list::reverse	none	
.list::clear	none	
.list::pop	optional,	
.list::index		
.list::append		
.list::extend		
.list::insert		
.list::remove		
.list::count		

Table 6.3: List operations in Jac

Op	Args	Description
.dict::items	none	
.dict::copy	none	Returns a shallow copy of the dictionary
.dict::deepcopy	none	Returns a deep copy of the dictionary
.dict::keys	none	
.dict::clear	none	
.dict::popitem	none	
.dict::values	none	
.dict::pop		
.dict::update		

Table 6.4: Dictionary operations in Jac

```
Hello!
```

Jac Code 6.17: else statement

```
walker init {
    a = 4; b = 5;
    if(a == b): std.out("A equals B");
    else: std.out("A is not equal to B");
}
```

```
A is not equal to B
```

Jac Code 6.18: elif statement

```
walker init {
    a = 4; b = 5;
    if(a == b): std.out("A equals B");
    elif(a > b): std.out("A is greater than B");
    elif(a == b - 1): std.out("A is one less than B");
    elif(a == b - 2): std.out("A is two less than B");
    else: std.out("A is something else");
}
```

```
A is one less than B
```

Jac Code 6.19: for loop

```
walker init {
    for i=0 to i<10 by i+=1:
        std.out("Hello", i, "times!");
}
```

```
Hello 0 times!
Hello 1 times!
Hello 2 times!
Hello 3 times!
Hello 4 times!
Hello 5 times!
Hello 6 times!
Hello 7 times!
Hello 8 times!
Hello 9 times!
```

Jac Code 6.20: for loop through list

```
walker init {
    my_list = [1, 'jon', 3.5, 4];
    for i in my_list:
        std.out("Hello", i, "times!");
}
```

```
Hello 1 times!
Hello jon times!
Hello 3.5 times!
Hello 4 times!
```

Jac Code 6.21: while loop

```
walker init {
    i = 5;
    while(i>0) {
        std.out("Hello", i, "times!");
        i -= 1;
    }
}
```

```
Hello 5 times!
Hello 4 times!
Hello 3 times!
Hello 2 times!
Hello 1 times!
```

Jac Code 6.22: break statement

```
walker init {
    for i=0 to i<10 by i+=1 {
        std.out("Hello", i, "times!");
        if(i == 6): break;
    }
}
```

```
Hello 0 times!
Hello 1 times!
Hello 2 times!
Hello 3 times!
Hello 4 times!
Hello 5 times!
Hello 6 times!
```

Jac Code 6.23: continue statement

```
walker init {
    i = 5;
```

```
    while(i>0) {
        if(i == 3){
            i -= 1; continue;
        }
        std.out("Hello", i, "times!");
        i -= 1;
    }
}
```

```
Hello 5 times!
Hello 4 times!
Hello 2 times!
Hello 1 times!
```

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]

[Code blocks]

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 namespaces? LBRACE attr_stmt* walk_entry_block? (
              statement
              | walk_activity_block
       )* walk_exit_block? RBRACE;
```

(full grammar in Appendix C)

7.2 Graphs as First Class Citizens

7.2.1 Connect and Spawn operations

Jac Code 7.2: Simple walker creating and connected nodes

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

Jac Code 7.3: Creating named node types

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

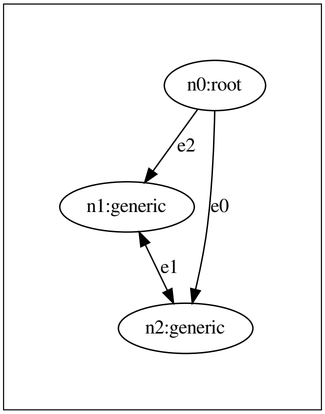


Figure 7.1: Graph in memory for JC 7.2

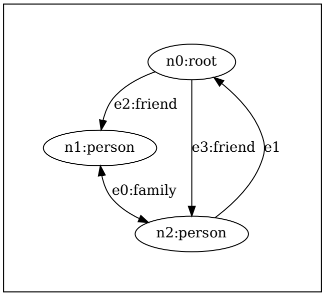


Figure 7.2: Graph in memory for JC 7.3

```
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;
}
```

Jac Code 7.4: Connecting nodes within spawn statement

```
node person;
```

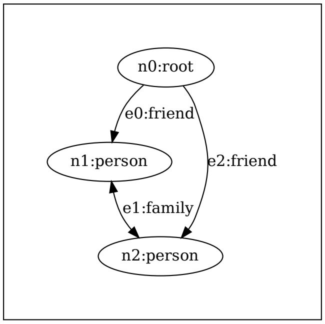


Figure 7.3: Graph in memory for JC 7.4

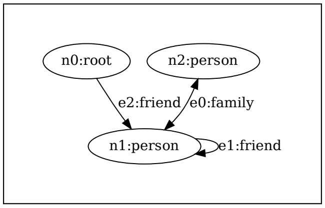


Figure 7.4: Graph in memory for JC 7.5

```
edge friend;
edge family;

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

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;
}
```

Another incredibly useful notion to consider about connect operations is that they can be chained. The same graph shown in Figure 7.4 can be achieved with the chained usage of the connect operation in line 8 of JC 7.5. Here nodes are chained in an intuitive left-to-right manor. Relatively sophisticated graph structures can be rapidly expressed using chained connect operations.

7.2.2 Static Graph Creation

7.2.2.1 Static Spawn Graphs

Jac Code 7.6: A Spawn style static graph

```
graph hlp_graph {
    has anchor graph_root;
    spawn {
        graph_root = spawn node::state(name="root_state");
        user_node = spawn node::user;

        state_home_price_inquiry = spawn node::state(name="home_price_inquiry");
        state_prob_of_approval = spawn node::state(name="prob_of_approval");

        graph_root -[user]-> user_node;

        graph_root -[transition(intent_label = "home price inquiry")]-> state_home_price_inquiry;
        graph_root -[transition(intent_label = "robability of loan approval")]-> state_prob_of_approval;
        state_home_price_inquiry -[transition(intent_label = "specifying location")]-> state_home_price_inquiry;
        state_home_price_inquiry -[transition(intent_label = "home price inquiry")]-> state_home_price_inquiry;

        state_home_price_inquiry -[transition(intent_label = "probability of loan approval")]-> state_prob_of_approval;
        state_prob_of_approval -[transition(intent_label = "home price inquiry")]-> state_home_price_inquiry;
    }
}
```

Jac Code 7.7: Associated DOT style static graph

```
graph acme_graph_dot {
    has anchor state_conv_root;
    graph G {
        state_conv_root [node=conv_state, name=conv_root]

        state_office_hour [node=conv_state, name=office_hour]
        state_payment_method [node=conv_state, name=payment_method]
        state_phone_number [node=conv_state, name=phone_number]
        state_email_address [node=conv_state, name=email_address]
        state_promotions [node=conv_state, name=promotions]

        state_cancel_appointment [node=conv_state, name=cancel_appointment]
        state_reschedule_appointment [node=conv_state, name=reschedule_appointment]
        state_refunds [node=conv_state, name=refunds]
        state_feedback [node=conv_state, name=feedback]

        state_service_inquiry [node=conv_state, name=service_inquiry]

        state_conv_root -> state_office_hour [edge=transition, intent="office hour"]
        state_conv_root -> state_payment_method [edge=transition, intent="payment method"]
        state_conv_root -> state_phone_number [edge=transition, intent="phone number"]
        state_conv_root -> state_email_address [edge=transition, intent="email address"]
        state_conv_root -> state_promotions [edge=transition, intent="promotions"]
        state_conv_root -> state_cancel_appointment [edge=transition, intent="cancel appointment"]
        state_conv_root -> state_reschedule_appointment [edge=transition, intent="reschedule appointment"]
        state_conv_root -> state_refunds [edge=transition, intent="refunds"]
        state_conv_root -> state_feedback [edge=transition, intent="feedback"]
        state_conv_root -> state_service_inquiry [edge=transition, intent="service inquiry"]
    }
}
```

7.2.2.2 Static DOT Graphs

Jac Code 7.8: A DOT style static graph

```
node test_node {
    has name;
}
edge special;
graph test_graph {
    has anchor graph_root;
    graph G {
        graph_root [node=test_node, name=root]
        node_1 [node=test_node, name=node_1]
        node_2 [node=test_node, name=node_2]
        graph_root -> node_1 [edge=special]
        graph_root -> node_2
    }
}
walker init {
    has nodes;
    with entry {
        nodes = [];
    }
    root {
        spawn here --> graph::test_graph;
        take --> node::test_node;
    }
    test_node {
        nodes += [here];
        take -[special]-> node::test_node;
    }
    report here;
}
```

```
{
  "success": true,
  "report": [
    {
      "context": {},
      "anchor": null,
      "name": "root",
      "kind": "generic",
      "jid": "urn:uuid:0ac65923-90b5-4c10-bda0-65ec6a2c36e7",
      "j_timestamp": "2022-03-21T00:41:16.715258",
      "j_type": "graph"
    },
    {
      "context": {
        "name": "root"
      },
      "anchor": null,
      "name": "test_node",
      "kind": "node",
      "jid": "urn:uuid:60e68110-7a11-446e-a333-57d75d12e7d7",
      "j_timestamp": "2022-03-21T00:41:16.750759",
      "j_type": "node"
    },
    {
      "context": {
        "name": "node_1"
      },
      "anchor": null,
      "name": "test_node",
      "kind": "node",
      "jid": "urn:uuid:fecae690-a50d-4f2c-91e2-e8ec083c5443",
      "j_timestamp": "2022-03-21T00:41:16.750876",
      "j_type": "node"
    }
  ]
}
```


Jac Code 7.9: Another DOT style static graph

```
node year {
    has color;
}
node month {
    has count, season;
```

```
}
node week;
node day;
edge parent;
edge child;
graph test_graph {
    has anchor A;
    strict graph G {
        H [node=year]
        C [node=week]
        E [node=day]
        D [node=day]

        A -> B // Basic directional edge
        B -- H // Basic non-directional edge
        B -> C [edge=parent] // Edge with attribute
        C -> D -> E [edge=child] // Chain edge

        A [color=red] // Node with DOT builtin graphing attr
        B [node=month, count=2] [season=spring]// Node with Jac attr
        A [node=year] // Multiple attr statement per node
    }
}
walker init {
    root {
        spawn here --> graph::test_graph;
    }
    take -->;
    report here.details['name'];
}
```

```
{
  "success": true,
  "report": [
    "root",
    "year",
    "month",
    "year",
    "week",
    "day",
    "day"
  ]
}
```


7.3 Walkers as the second First Class Citizens

Jac Code 7.10: 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
```

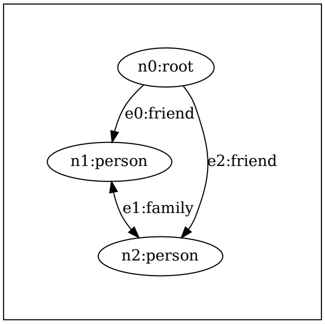


Figure 7.5: Graph in memory for JC 7.10

Jac Code 7.11: 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 7.12: Increasing elegance by remembering spawns are expressions

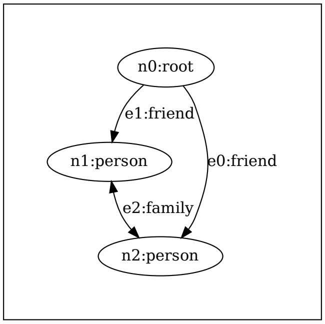


Figure 7.6: Graph in memory for JC 7.11

```
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');
}
```

Walkers are entry points to all valid jac programs

7.4 Architypes

7.4.1 Context on Nodes and Edges

Jac Code 7.13: Binding member contexts to nodes and edges

```
node person {
    has name;
    has age;
```

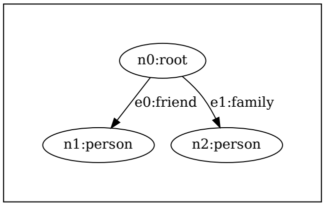


Figure 7.7: Graph in memory for JC 7.13

```
    has birthday, profession;
}

edge friend: has meeting_place;
edge family: has kind;

walker init {
    person1 = spawn here -[friend]-> node::person;
    person2 = spawn here -[family]-> node::person;
    person1.name = "Josh"; person1.age = 32;
    person2.name = "Jane"; person2.age = 30;
    e1 = -[friend]->.edge[0];
    e1.meeting_place = "college";
    e2 = -[family]->.edge[0];
    e2.kind = "sister";

    std.out("Context␣for␣our␣people␣nodes:");
    for i in -->: std.out(i.context);
    # or, for i in -->.node: std.out(i.context);
    std.out("\nContext␣for␣our␣edges␣to␣those␣people:");
    for i in -->.edge: std.out(i.context);
}
```

```
Context for our people nodes:
{'name': 'Josh', 'age': 32, 'birthday': '', 'profession': ''}
{'name': 'Jane', 'age': 30, 'birthday': '', 'profession': ''}

Context for our edges to those people:
{'meeting_place': 'college'}
{'type': 'sister'}
```

Jac Code 7.14: Binding contexts with less code

```
node person: has name, age, birthday, profession;
edge friend: has meeting_place;
edge family: has kind;

walker init {
    person1 = spawn here -[friend(meeting_place = "college")] ->
        node::person(name = "Josh", age = 32);
    person2 = spawn here -[family(kind = "sister")] ->
        node::person(name = "Jane", age = 30);

    std.out("Context␣for␣our␣people␣nodes␣and␣edges:");
    for i in -->: std.out(i.context, '\n', i.edge[0].context);
}
```

```
Context for our people nodes and edges:
{'name': 'Josh', 'age': 32, 'birthday': '', 'profession': ''}
 {'meeting_place': 'college'}
{'name': 'Jane', 'age': 30, 'birthday': '', 'profession': ''}
 {'type': 'sister'}
```

7.4.2 Copy Assignment Operator

Jac Code 7.15: Copy assigning from node to node

```
node person: has name, age, birthday, profession;
edge friend: has meeting_place;
edge family: has kind;

walker init {
    person1 = spawn here -[friend(meeting_place = "college")] ->
        node::person(name = "Josh", age = 32);
    person2 = spawn here -[family(kind = "sister")] ->
        node::person(name = "Jane", age = 30);

    twin1 = spawn here -[friend]-> node::person;
    twin2 = spawn here -[family]-> node::person;
    twin1 := person1;
    twin2 := person2;

    -->.edge[2] := -->.edge[0];
    -->.edge[3] := -->.edge[1];
```

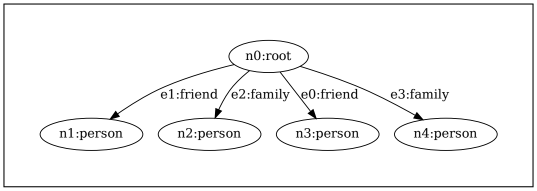


Figure 7.8: Graph in memory for JC 7.15

```

    std.out("Context for our people nodes and edges:");
    for i in -->: std.out(i.context, '\n', i.edge[0].context);
}
```

```
{'name': 'Josh', 'age': 32, 'birthday': '', 'profession': ''}
 {'meeting_place': 'college'}
{'name': 'Jane', 'age': 30, 'birthday': '', 'profession': ''}
 {'type': 'sister'}
{'name': 'Josh', 'age': 32, 'birthday': '', 'profession': ''}
 {'meeting_place': 'college'}
{'name': 'Jane', 'age': 30, 'birthday': '', 'profession': ''}
 {'type': 'sister'}
```

7.4.3 Plucking Values from Node and Edge Sets

Another very handy dandy feature when interacting with collections of nodes and edges is to quickly extract a list of all the values for a given `has` variable across the collection of nodes or edges. Lets look at an example.

Jac Code 7.16: Plucking values out of nodes and edges

```
node simple: has n_name;
edge conn: has e_name;

walker node_edge_plucking {
    with entry {
        for i=0 to i<3 by i+=1:
            spawn here -[conn(e_name="edge"+i.str)]-> node::simple(n_name=
                "node"+i.str);
    }
    std.out(-->.n_name);
```

```
10     std.out(-->.edge.e_name);
11 }
```

```
["node0", "node1", "node2"]
["edge0", "edge1", "edge2"]
```

As shown in JC 7.16 we are referencing the `has` variable of the architypes for the collection of `simple` nodes and `conn` edges on lines 8 and 9 respectively. As can be seen in the output, these references evaluate to a list of the values for the corresponding variables. Keep in mind this can work with a mixture of nodes and edges in a collection given they share a given `has` variable name.

7.4.4 Referencing and Dereferencing Nodes and Edges

Nodes and edges can be referenced and dereferenced. These operations are synonymous with they way references work in many languages and borrows the syntax of pointers in C/C++. In particular, the `&` is used to get the reference of an object and `*` is used to dereference object. However, in contrast to C/C++, instead of the references representing memory location in word format, references in Jac uses a unique identifier (in UUID format) for the object.

Jac Code 7.17: Rereferences and dereferences in Jac

```
node simple: has name;

walker ref_deref {
    with entry {
        for i=0 to i<3 by i+=1:
            spawn here --> node::simple(name="node"+i.str);
    }
    var = &(-->[0]);
    std.out('ref:', var);
    std.out('obj:', *var);
    std.out('info:',(*var).info);
}
```

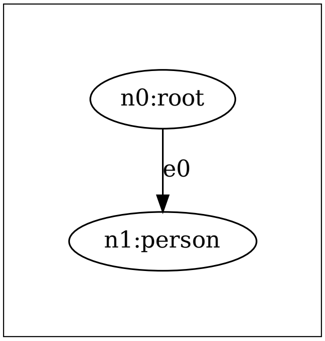


Figure 7.9: Graph in memory for JC 7.18 and 7.19


```
ref: urn:uuid:04295f7f-a5bf-4db3-87ce-e13653a81b25
obj: jac:uuid:04295f7f-a5bf-4db3-87ce-e13653a81b25
info: {"context": {"name": "node0"}, "anchor": null, "name": "simple", "
    ↪ kind": "node", "jid": "urn:uuid:04295f7f-a5bf-4db3-87ce-
    ↪ e13653a81b25", "j_timestamp": "2022-08-10T15:57:00.577287", "
    ↪ j_type": "node"}
```


JC 7.17 shows an example of the behavior of references and dereferences in Jac. Note that once dereferenced `var` is simply a UUID formatted string with the unique identifier of the object itself. This UUID is equivalent to the `jid` in the object `.info`. These referencing and dereferencing operations are quite useful for input and output of node locations to a client side, etc.

Nerd Alert 8 ***(time to let your eyes glaze over)***

Important Note: The internal representation of an instance of an architype is a string composed of any UUID that starts with `"jac:uuid:"`. This may change in the future but, if you were to manually assign such a string to a variable in a Jac program, the program will treat this variable like an object.

7.5 Actions and Abilities

7.5.1 Actions

Jac Code 7.18: Basic action in walker

```
node person {
    has name;
    has birthday;
}

walker init {
    can date.quantize_to_year;
    person1 = spawn here -->
        node::person(name="Josh", birthday="1995-05-20");
    birthyear = date.quantize_to_year(person1.birthday);
    std.out(birthyear);
}
```

```
1995-01-01T00:00:00
```

Jac Code 7.19: Basic action in node

```
node person {
    has name;
    has birthday;
    can date.quantize_to_year;
}

walker init {
    root {
        person1 = spawn here -->
            node::person(name="Josh", birthday="1995-05-20");
        take -->;
    }
    person {
        birthyear = date.quantize_to_year(here.birthday);
        std.out(birthyear);
    }
}
```

7.5.2 Fused Interactions Between Nodes and Actions

Jac Code 7.20: Basic action with presets and event triggers

```
node person {
    has name;
```

```
    has byear;
    can date.quantize_to_year::visitor.year::>byear with setter entry;
    can std.out::byear," from ",visitor.info:: with exit;
}

walker init {
    has year=std.time_now();
    root {
        person1 = spawn here -->
            node::person(name="Josh", byear="1992-01-01");
        take --> ;
    }
    person {
        spawn here walker::setter;
    }
}

walker setter {
    has year="1995-01-01";
}
```

```
1995-01-01T00:00:00 from {'context': {'year': '1995-01-01'}, 'anchor': None, 'name': 'setter', 'kind': 'walker', 'jid': 'urn:uuid:6bbf69c3-b95c-4a88-a783-cb793cec4034', 'j_timestamp': '2021-12-04T15:13:13.441516', 'j_type': 'walker'}
1995-01-01T00:00:00 from {'context': {'year': '2021-12-04T15:13:13.440803'}, 'anchor': None, 'name': 'init', 'kind': 'walker', 'jid': 'urn:uuid:7f9d1462-6562-4d4d-ba57-f069c74dfe1e', 'j_timestamp': '2021-12-04T15:13:13.438072', 'j_type': 'walker'}
```

Jac Code 7.21: Basic action with presets and event triggers

```
node person {
    has name;
    has birthday;
    can date.quantize_to_year with activity; # <-- walkers can call
}

walker init {
    root {
        person1 = spawn here -->
            node::person(name="Josh", birthday="1995-05-20");
        take -->;
```

```
    }
    person {
        birthyear = date.quantize_to_year(here.birthday);
        std.out(birthyear);
    }
}
```

[Only nodes can have with entry/exit‘’ and presets]

[can leave output (push returns) in node and walker]

```
1995-01-01T00:00:00 from {'context': {'year': '1995-01-01'}, 'anchor':
    ↪ None, 'name': 'setter', 'kind': 'walker', 'jid': 'urn:uuid:6
    ↪ bbf69c3-b95c-4a88-a783-cb793cec4034', 'j_timestamp': '2021-12-04
    ↪ T15:13:13.441516', 'j_type': 'walker'}
1995-01-01T00:00:00 from {'context': {'year': '2021-12-04T15
    ↪ :13:13.440803'}, 'anchor': None, 'name': 'init', 'kind': 'walker',
    ↪  'jid': 'urn:uuid:7f9d1462-6562-4d4d-ba57-f069c74dfe1e', '
    ↪ j_timestamp': '2021-12-04T15:13:13.438072', 'j_type': 'walker'}
```

7.5.3 Abilities

Jac Code 7.22: Actions and Abilities in Walkers

```
node person {
    has name;
    has byear;
    can set_year with setter entry {
        byear = visitor.year;
    }
    can print_out with exit {
        std.out(byear," from ",visitor.info);
    }
    can reset { #<-- Could add 'with activity' for equivalent behavior
        ::set_back_to_95;
        std.out("resetting year to 1995:", here.context);
    }
    can set_back_to_95: byear="1995-01-01";
}

walker init {
    has year=std.time_now();
```

```
    can setup {
        person1 = spawn here --> node::person;
        std.out(person1);
        person1::reset;
    }
    root {
        ::setup;
        take --> ;
    }
    person {
        spawn here walker::setter;
        person1::reset(name="Joe");
    }
}

walker setter {
    has year=std.time_now();
}
```

Jac Code 7.23: Abilities in nodes

```
node person {
    has name;
    has byear;
    can set_year with setter entry {
        byear = visitor.year;
    }
    can print_out with exit {
        std.out(byear," from ",visitor.info);
    }
    can reset { #<-- Could add 'with activity' for equivalent behavior
        byear="1995-01-01";
        std.out("resetting birth year to 1995:", here.context);
    }
}

walker init {
    has year=std.time_now();
    root {
        person1 = spawn here --> node::person;
        std.out(person1);
        person1::reset;
        take --> ;
    }
```

```
    person {
        spawn here walker::setter;
        here::reset(name="Joe");
    }
}

walker setter {
    has year=std.time_now();
}
```

7.5.4 here and visitor, the 'this' references of Jac

Observe the usage of `here` and `visitor` in the `person` node architype in JC 7.23. These are synonymous to the this reference present in many other languages except `here` point to the current node scope relevant to the execution point in the program and `visitor` points to the relevant walker scope relevant to that given point of execution. These references provide full access to all `has` variables and builtin attributes and operations of the referenced object instance.

Do note that in the context of the `person` node abilities in JC 7.23 a here reference to say `here.name = "joe";` would be equivalent to simply `name = "joe";` however to capture the `here.context` (or info/details/etc) the `here` reference becomes quite useful. The similar relationship applies to using `visitor` in walker abilities.

7.6 Inheritance

Chapter 8

Walkers Navigating Graphs

8.1 Taking Edges (and Nodes?)

8.1.1 Basic Walks

Jac Code 8.1: Basic example of walker traveling graph

```
node person: has name;

walker get_names {
    std.out(here.name);
    take -->;
}

walker build_example {
    node1 = spawn here --> node::person(name="Joe");
    node2 = spawn node1 --> node::person(name="Susan");
    spawn node2 --> node::person(name="Matt");
}

walker init {
    root {
        spawn here walker::build_example;
        take -->;
    }
    person {
```

```
        spawn here walker::get_names;
        disengage;
    }
}
```

Jac Code 8.2: Fan out style takes

```
node person: has name;

walker build_example {
    spawn here -[friend]-> node::person(name="Joe");
    spawn here -[friend]-> node::person(name="Susan");
    spawn here -[family]-> node::person(name="Matt");
}

walker init {
    root {
        spawn here walker::build_example;
        take -->;
    }
    person {
        std.out(here.name);
    }
}
```

8.1.2 Breadth First vs Depth First Walks

If you've played with the basic `take` command a bit you would notice that by default it results in a breadth first traversal of a graph. However, the `take` command is indeed quite flexible. You can specify an orientation of the `take` command to navigate with a breadth first or a depth first traversal.

Jac Code 8.3: Breadth first navigation with take vs depth first

```
node plain: has name;

graph example {
    has anchor head;
    spawn {
        n=[];
        for i=0 to i<7 by i+=1 {
            n.l::append(spawn node::plain(name=i+1));
```

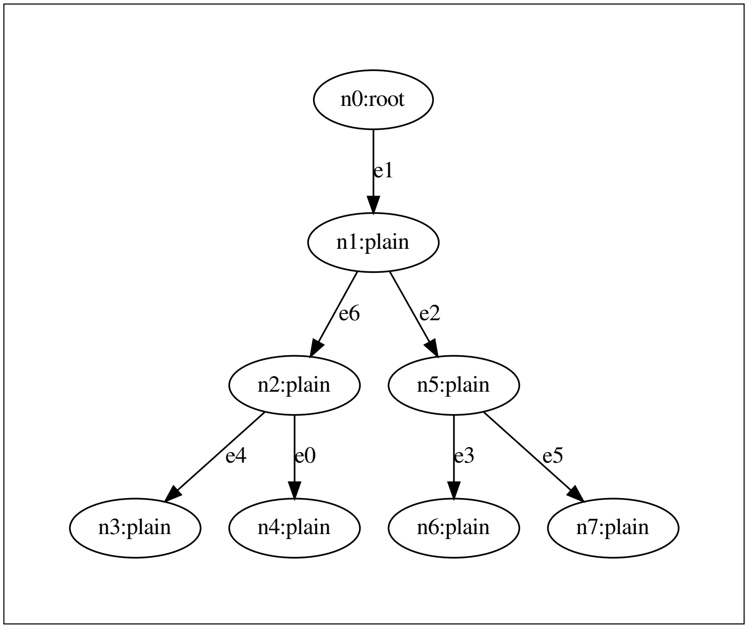


Figure 8.1: Graph in memory for JC 8.3

```
        }
        n[0] --> n[1] --> n[2];
                 n[1] --> n[3];
        n[0] --> n[4] --> n[5];
                 n[4] --> n[6];
        head=n[0];
    }
}

walker walk_with_breadth {
    has anchor node_order = [];
    node_order.l::append(here.name);
    take:bfs -->; #take:b can also be used
}

walker walk_with_depth {
    has anchor node_order = [];
    node_order.l::append(here.name);
    take:dfs -->; #take:d can also be used
}

```

```
30 walker init {
31     start = spawn here --> graph::example;
32     b_order = spawn start walker::walk_with_breadth;
33     d_order = spawn start walker::walk_with_depth;
34     std.out("Walk␣with␣Breadth:",b_order,"\nWalk␣with␣Depth:",d_order);
35 }
```

Take for example the program shown in JC 8.3. First we observe the definition of a static three level binary tree with the graph `example` on line 3. This is a vanilla structure as depicted in Figure 8.1. Two walkers are present in this example, one walker `walk\_with` ↪ `\_breadth`, for which we observe a call to `take:bfs -->;` indicating a breadth first traversal, and another walker `walk\_with\_breadth`, for which we observe a call to `take:` ↪ `bfs -->;` indicating a depth first traversal.

As can be seen in its output,

```
Walk with Breadth: [1, 2, 5, 3, 4, 6, 7]
Walk with Depth: [1, 2, 3, 4, 5, 6, 7]
{
  "success": true,
  "report": []
}
```

The print statement on line 34 demonstrate the order of nodes visited correspond to the specified traversal order.

Additionally, the short hand of `take:b -->;`, or `take:d -->;` could be used to specify breadth first or depth first traversals respectively.

8.2 Skipping and Disengaging

With walker traversing graphs with `take` commands, Jac introduces a few new handy control statements that are quite handy, namely, `skip` and `disengage`.

8.2.1 Skip

In the context of a walkers code block, the intuition bethind the abstraction of `skip` is that it instructs a walker to stop and forego all remaining computation on the current node and move to the next node (or complete computation if no nodes are queued up). Regardless

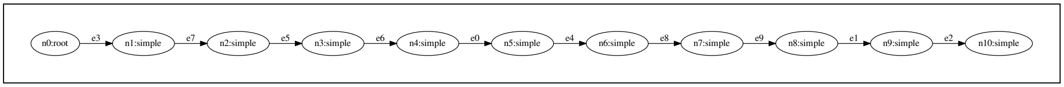


Figure 8.2: Graph in memory for JC 8.4 and JC 8.5

as to where in the walkers body the skip occurs, the entire remaining code in the walker is skipped and the walker moves on.

The skip directive can also be used in node/edge abilities. In this context, the skip simply foregoes the remaining execution of that ability itself.

Lets look at an example of a walker using the skip command.

Jac Code 8.4: Skipping nodes along a walk

```
1  global node_count=0;
2  node simple: has id;
3
4  walker init {
5      has output = [];
6      with entry {
7          t = here;
8          for i=0 to i<10 by i+=1 {
9              t = spawn t --> node::simple(id=global.node_count);
10             global.node_count+=1;
11         }
12     }
13     take -->;
14     simple {
15         if(here.id % 2==0): skip;
16         output.l::append(here.id);
17     }
18     output.l::append(here.info['name']);
19     with exit: std.out(output);
20 }
```

```
["root", 1, "simple", 3, "simple", 5, "simple", 7, "simple", 9, "simple"]
```

JC 8.4 shows an example of the skip command in practice. The init walker here traverses a simple chain of nodes as depicted in Figure 8.2. As can be seen in the output the skip command on line 15 causes only the odd elements to be added to the output array.

The semantics of the skip command is pretty much identical to the traditional break commands except it "breaks" out of a walker or ability as opposed to a loop. Another way to think of it is as a `return` of sorts.

8.2.2 Disengage

Disengage is a statement that can only be used inside a walker's code body and instructs the walker to halt all execution and 'disengage' from the graph (i.e. do not visit any more nodes). In practice this is essential a skip with a clearing of all future nodes to visit.

Lets look at an example of a walker using the `disengage` command.

Jac Code 8.5: Disengaging walker during walk

```
1 global node_count=0;
2 node simple: has id;
3
4 walker init {
5     has output = [];
6     with entry {
7         t = here;
8         for i=0 to i<10 by i+=1 {
9             t = spawn t --> node::simple(id=global.node_count);
10            global.node_count+=1;
11        }
12    }
13    take -->;
14    simple {
15        if(here.id % 2==0): skip;
16        if(here.id == 7): disengage;
17        output.l::append(here.id);
18    }
19    output.l::append(here.info['name']);
20    with exit: std.out(output);
21 }
```

```
["root", 1, "simple", 3, "simple", 5, "simple"]
```

JC 8.5 shows an example of the `disengage` command. The `init` walker here is almost identical to the implementation of JC 8.4 however we've added `if(here.id == 7): disengage;` on line 16. This cause our walker to stop its execution and complete its walk resulting in an effective truncation of the `output` array.

Note that, in addition to a basic `disengage;`, Jac also support a disengage-report shorthand of the format `disengage report "I'm disengaging";`. This directive results in a final report before the disengage executes.

8.2.3 Technical Semantics of Skip and Disengage

There are a number of important semantics of `skip` and `disengage` to keep in mind:

1. The `skip` statement can be used in the code bodies of walkers and abilities.
2. The `disengage` statement can only be used in the code body of walkers.
3. The `with exit` code block is not affected by `skip` or `disengage` statements. Upon a `disengage`, any code in a walker's `with exit` block will execute immediately after as the walker is exiting the graph.
4. An easy way to think about these semantics is as similar to the behavior of a traditional `return` (skip) and a `return` and stop walking (disengage).

8.3 Ignoring and Deleting

Jac Code 8.6: Ignoring edges during walk

```
node person: has name;
edge family;
edge friend;

walker build_example {
    spawn here -[friend]-> node::person(name="Joe");
    spawn here -[friend]-> node::person(name="Susan");
    spawn here -[family]-> node::person(name="Matt");
    spawn here -[family]-> node::person(name="Dan");
}

walker init {
    root {
        spawn here walker::build_example;
        ignore -[family]->;
        ignore -[friend(name=="Joe")]->;
        take -->;
    }
```

```
    person {
        std.out(here.name);
    }
}
```

```
Jac Code 8.7: Destorying nodes/edges during walk
node person: has name;
edge family;
edge friend;

walker build_example {
    spawn here -[friend]-> node::person(name="Joe");
    spawn here -[friend]-> node::person(name="Susan");
    spawn here -[family]-> node::person(name="Matt");
    spawn here -[family]-> node::person(name="Dan");
}

walker init {
    root {
        spawn here walker::build_example;
        for i in -[friend]->: destroy i;
        take -->;
    }
    person {
        std.out(here.name);
    }
}
```

8.4 Reporting Back as you Travel

```
Jac Code 8.8: Building reports as you walk
node person: has name;
edge family;
edge friend;

walker build_example {
    spawn here -[friend]-> node::person(name="Joe");
    spawn here -[friend]-> node::person(name="Susan");
    spawn here -[family]-> node::person(name="Matt");
```

```
    spawn here -[family]-> node::person(name="Dan");
}

walker init {
    root {
        spawn here walker::build_example;
        spawn -->[0] walker::build_example;
        take -->;
    }
    person {
        report here; # report print back on disengage
        take -->;
    }
}
```

8.5 Yielding Walkers

So far, we've looked at walkers that will walk the graph carrying state in context (`has` variables). But you may be wonder what happens after its walk? And does it keep that state like nodes and edges? Short answer is no. At the end of each walk a walker's state is cleared by default while node/edge state persists. That being said, there are situations where you'd want a walker to keep its state across runs, and perhaps, you may even want a walker to stop during a walk and wait to be explicitly called again updating just a few of it's dynamic state. This is where the `yield` keyword comes in.

Lets look at an example of yield in action.

Jac Code 8.9: Simple example of yielding walkers

```
global node_count=0;

node simple {has id;}

walker simple_yield {
    with entry {
        t=here;
        for i=0 to i<10 by i+=1 {
            t = spawn t --> node::simple(id=global.node_count);
            global.node_count+=1;
        }
    }
    report here.context;
```

```
14     take -->;
15     yield;
16 }
```

The `yield` keyword in JC 8.9 instructs the walker `simple_yield` to stop walking and wait to be called again, even though the walker is instructed to `take -->` edges. In this example, a single next node location is queued up and the walker reports a single `here.context` each time it's called, taking only 1 edge per call.

8.5.1 Yield Shorthands

Also note `yield` can be followed by a number of operations as a shorthand. For example line 14 and 15 in JC 8.9 could be combined to a single line with `yield take -->;`. We call this a yield-take. Shorthands include,

- Yield-Take: `yield take -->;`
- Yield-Report: `yield report "hi";`
- Yield-Disengage: `yield disengage;` and `yield disengage report "bye";`

In each of these cases, the `take`, `report`, and `disengage` executes with the yield.

8.5.2 Technical Semantics of Yield

There are a number of important semantics of `yield` to keep in mind:

1. Upon a `yield`, a report is returned back and cleared.
2. Additional report items from further walking will be return on subsequent `yield`s or walk completion.
3. Like the `take` command, the entire body of the walker will execute on the current node and actually yield at the end of this execution.
   - *Note: Keep in mind* `yield` *can be combined with* `disengage` *and* `skip` *commands.*
4. If a start node (aka a 'prime' node) is specified when continuing a walker after a `yield`, if there are additional walk locations the walker is scheduled to travel to, the walker will ignore this prime node and continue from where it left off on its journey.

5. If there are no nodes scheduled for the walker to go to next, a prime node must be specified (or the walker will continue from root by default).

6. `with entry` and `with exit` code blocks in the walker are not executed upon continuing from a `yield` or executing a `yeild` respectively. They execute only once starting and ending a walk though there may be many yields in between.

7. The state of which walkers are yielded and to be continued vs which walkers are being freshly run is kept at the level of the `master` (user) abstraction in Jaseci. At the moment, walkers that are summoned as public has undefined yield semantics. Developers should leverage the more lower level `walker spawn` and `walker execute` APIs for customized yield behaviors.

8.5.3 Walkers Yielding Other Walkers (i.e., Yielding Deeply)

In addition to the utility of calling walkers that yield from client, walkers also benefit from this abstraction when calling other walkers during a non-yielding walk. Lets take a look at a code example.

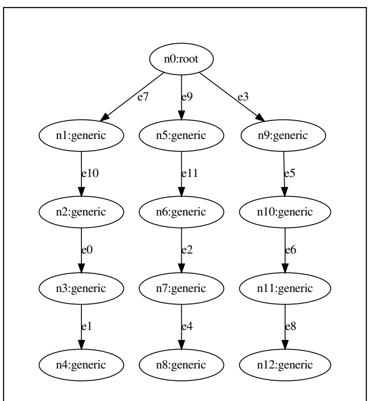
Jac Code 8.10: Walkers yielding other walkers

```
walker simple_yield {
    with entry {
        t=here;
        for i=0 to i<4 by i+=1:
            t = spawn t --> node::generic;
    }
    if(-->.length): yield take -->;
}

walker deep_yield {
    for i=0 to i<16 by i+=1 {
        spawn here walker::simple_yield;
    }
}
```

As shown in JC 8.10, the walker `deep_yield` does not yield itself, but enjoys the semantics of the yield command in `simple_yield`.

Figure 8.3 shows the graph created by JC 8.10. Though `deep_yield` does not yield, tt calls `simple_yield` 16 times and exits. These 16 calls trigger `walker::simple_yield` which in turn creates four chained



nodes off of the root node then walks the chain one step at a time while yielding after each step. The result is this very nice 17 node graph with a root node and 3 subtrees with 4 connected nodes each. Yep, this yeilding semantic is very handy indeed!

Chapter 9

Actions and Action Sets

9.1 Standard Action Library

9.1.1 date

`date.quantize_to_year`

args: date: str (*req)

No documentation yet.

`date.quantize_to_month`

args: date: str (*req)

No documentation yet.

`date.quantize_to_week`

args: date: str (*req)

No documentation yet.

`date.quantize_to_day`

args: date: str (*req)

No documentation yet.

`date.date_day_diff`

args: start_date: str (*req), end_date: str (*req)

No documentation yet.

9.1.2 file

`file.load_str`

args: fn: str (*req), max_chars: int

No documentation yet.

`file.load_json`

args: `fn: str (*req)`

No documentation yet.

`file.dump_str`

args: `fn: str (*req), s: str (*req)`

No documentation yet.

`file.append_str`

args: `fn: str (*req), s: str (*req)`

No documentation yet.

`file.dump_json`

args: `fn: str (*req), obj: _empty (*req), indent: int`

No documentation yet.

`file.delete`

args: `fn: str (*req)`

No documentation yet.

9.1.3 net

`net.max`

args: `item_set: jac_set (*req)`

No documentation yet.

`net.min`

args: `item_set: jac_set (*req)`

No documentation yet.

`net.root`

args: `meta: _empty (*req)`

No documentation yet.

9.1.4 rand

`rand.seed`

args: val: int (*req)

No documentation yet.

`rand.integer`

args: start: int (*req), end: int (*req)

No documentation yet.

`rand.choice`

args: lst: list (*req)

No documentation yet.

`rand.sentence`

args: min_lenth: int, max_length: int, sep: str

No documentation yet.

`rand.paragraph`

`args: min_lenth: int, max_length: int, sep: str`

No documentation yet.

`rand.text`

`args: min_lenth: int, max_length: int, sep: str`

No documentation yet.

`rand.word`

`args: n/a`

No documentation yet.

`rand.time`

`args: start_date: str (*req), end_date: str (*req)`

No documentation yet.

9.1.5 request

`request.get`

`args: url: str (*req), data: dict (*req), header: dict (*req)`

Param 1 - url Param 2 - data Param 3 - header
Return - response object

`request.post`

`args: url: str (*req), data: dict (*req), header: dict (*req)`

Param 1 - url Param 2 - data Param 3 - header
Return - response object

`request.put`

`args: url: str (*req), data: dict (*req), header: dict (*req)`

Param 1 - url Param 2 - data Param 3 - header
Return - response object

`request.delete`

`args: url: str (*req), data: dict (*req), header: dict (*req)`

Param 1 - url Param 2 - data Param 3 - header
Return - response object

`request.head`

`args: url: str (*req), data: dict (*req), header: dict (*req)`

Param 1 - url Param 2 - data Param 3 - header
Return - response object

`request.options`

`args: url: str (*req), data: dict (*req), header: dict (*req)`

Param 1 - url Param 2 - data Param 3 - header
Return - response object

`request.multipart_base64`

`args: url: str (*req), files: list (*req), header: dict (*req)`

Param 1 - url Param 3 - header Param 3 - file (Optional) used for single file Param 4 - files (Optional) used for multiple files Note - file and files can't be None at the same time
Return - response object

`request.file_download_base64`

`args: url: str (*req), header: dict (*req), encoding: str`

No documentation yet.

9.1.6 std

`std.log`

args: `args: _empty (*req)`

No documentation yet.

`std.out`

args: `args: _empty (*req)`

No documentation yet.

`std.js_input`

args: `prompt: str`

No documentation yet.

`std.err`

args: `args: _empty (*req)`

No documentation yet.

`std.sort_by_col`

args: `lst: list (*req), col_num: int (*req), reverse: bool`

Param 1 - list Param 2 - col number Param 3 - boolean as to whether things should be reversed
Return - Sorted list

`std.time_now`

args: `n/a`

No documentation yet.

`std.set_global`

args: `name: str (*req), value: _empty (*req), meta: _empty (*req)`

Param 1 - name Param 2 - value (must be json serializable)

`std.get_global`

args: `name: str (*req), meta: _empty (*req)`

Param 1 - name

std.actload_local
args: filename: str (*req), meta: _empty (*req)
No documentation yet.

std.actload_remote
args: url: str (*req), meta: _empty (*req)
No documentation yet.

std.actload_module
args: module: str (*req), meta: _empty (*req)
No documentation yet.

std.destroy_global
args: name: str (*req), meta: _empty (*req)
No documentation yet.

```
std.set_perms
```

```
args: obj: element (*req), mode: str (*req), meta: _empty (*req)
```

Param 1 - target element Param 2 - valid permission (public, private, read only)
Return - true/false whether successful

```
std.get_perms
```

```
args: obj: element (*req)
```

Param 1 - target element
Return - Sorted list

```
std.grant_perms
```

```
args: obj: element (*req), mast: element (*req), read_only: bool (*req
↪ ), meta: _empty (*req)
```

Param 1 - target element Param 2 - master to be granted permission Param 3 - Boolean read only flag
Return - Sorted list

```
std.revoke_perms
```

```
args: obj: element (*req), mast: element (*req), meta: _empty (*req)
```

Param 1 - target element Param 2 - master to be revoked permission
Return - Sorted list

`std.get_report`

args: meta: _empty (*req)

No documentation yet.

9.1.7 vector

`vector.cosine_sim`

args: vec_a: list (*req), vec_b: list (*req)

Param 1 - First vector Param 2 - Second vector
Return - float between 0 and 1

`vector.dot_product`

args: vec_a: list (*req), vec_b: list (*req)

Param 1 - First vector Param 2 - Second vector
Return - float between 0 and 1

`vector.get_centroid`

args: vec_list: list (*req)

Param 1 - List of vectors
Return - (centroid vector, cluster tightness)

vector.softmax
args: vec_list: list (*req)
Param 1 - List of vectors Return - (centroid vector, cluster tightness)

vector.sort_by_key
args: data: dict (*req), reverse: _empty, key_pos: _empty
Param 1 - List of items Param 2 - if Reverse Param 3 (Optional) - Index of the key to be used for sorting if param 1 is a list of tuples. Deprecated

9.2 Building Your Own Library

Chapter 10

Imports, File I/O, Tests, and More

10.1 Tests in Jac

Jac Code 10.1: Tests Example

```
node testnode {
    has yo, bro;
}

node apple {
    has v1, v2;
}

node banana {
    has x1, x2;
}

graph dummy {
    has anchor graph_root;
    spawn {
        graph_root = spawn node::testnode (yo="Hey␣yo!");
        n1=spawn node::apple(v1="I'm␣apple");
        n2=spawn node::banana(x1="I'm␣banana");
        graph_root --> n1 --> n2;
```

```
    }
}

walker init {
    has num=4;
    report here.context;
    report num;
    take -->;
}

test "assert␣should␣be␣valid"
with graph::dummy by walker::init {
    assert (num==4);
    assert (here.x1=="I'm␣banana");
    assert <--[0].v1=="I'm␣apple";
}

test "assert␣should␣fail"
with graph::dummy by walker::init {
    assert (num==4);
    assert (here.x1=="I'm␣banana");
    assert <--[0].v1=="I'm␣Apple";
}

test "assert␣should␣fail,␣add␣internal␣except"
with graph::dummy by walker::init {
    assert (num==4);
    assert (here.x1=="I'm␣banana");
    assert <--[10].v1=="I'm␣apple";
}
```

```
Testing "assert should be valid": [PASSED in 0.00s]
Testing "assert should fail": [FAILED in 0.00s]
('JAC Assert Failed', '<-- [ 0 ] . v1 == "I\'m Apple" ')
Testing "assert should fail, add internal except": [FAILED in 0.00s]
('JAC Assert Failed', '<-- [ 10 ] . v1 == "I\'m apple" ', IndexError('
    ↪ list index out of range'))
{
  "tests": 3,
  "passed": 1,
  "failed": 2,
  "success": false
}
```

10.2 Imports

Jac Code 10.2: Imports Example

```
import {graph::dummy, node::{banana, apple, testnode}} with "./jac_tests.
    ↪ jac";
# import {*} with "./jac_tests.jac";
# import {graph::dummy, node*} with "./jac_tests.jac";

walker init {
    has num=4;
    with entry {
        spawn here --> graph::dummy;
    }
    report here.context;
    report num;
    take -->;
}
```

```
{
  "success": true,
  "report": [
    {},
    4,
    {
      "yo": "Hey yo!",
      "bro": null
    },
    4,
    {
      "x1": "I'm banana",
      "x2": null
    },
    4
  ]
}
```


10.3 File I/O

Jac Code 10.3: File I/O Example

```
walker init {
    fn="fileiotest.txt";
    a = {'a': 5};
    file.dump_json(fn, a);
    b=file.load_json(fn);
    b['a']+=b['a'];
    file.dump_json(fn, b);
    c=file.load_str(fn);
    file.append_str(fn, c);
    c=file.load_str(fn);
    report c;
}
```

10.4 Visualizing Graph with Dot Output

A very useful feature of the Jaseci stack is the ability to dump a snapshot of a graph in memory as `dot` output. There are two core interfaces to access this feature. The first is the `graph get` api. Simply set the `mode` parameter to "dot" and a dot representation of the graph will be printed. This API is present in both `jsctl` and the REST api. The other is to use texttjac dot [filename]. This will run the program specified in filename, then print the state of the graph at the end of the program run as dot output. This `jac dot` api is only available through `jsctl`. For both of these apis, a `detailed` parameter can be used to get more information embedded in the dot output. In particular, any context variables that are string will be included in the nodes and edges of the dot output.

Part III

Jaseci AI Kit

Part IV

Crafting Jaseci

Chapter 11

Architecting Jaseci Core

Chapter 12

Architecting Jaseci Cloud Serving

Epilogue

Appendix A

Jumping Right In, TLDR sytle

If you're the kind of haxor that doesn't want to read a huge book and just wants to get hacking ASAP, this chapter is for you!! This chapter will make a few assumptions. Firstly, it is assumed that you are in a linux environment and will have command of the line that takes commands. Coincidental, this is commonly referred to as the *command line*. Secondly, this command line will be one that accepts linux style commands in a `bash` format. If you've never heard of bash, Google it. Thirdly and lastly, you will be using the only IDE true ninjas use, namely `VSCode`. If these conditions apply to your environment, you're good. If they don't but you use Linux, you're still good (as you're almost certainly competent enough at this stuff to be able to easily be able to make the necessary adjustments to get things working in your environment.)

We start this journey from the perspective of having a fresh vanilla install of the minimal version of Ubuntu 20+. Ubuntu is a distribution or (flavor) of linux that is likely the most popular and accessible in the market. I say likely because I don't know for sure, but if it isn't I'd be shocked!

A.1 Installation

First and foremost, lets check what os we're running at the moment.

```
haxor@linux:~$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.4 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.4 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/
    ↪ privacy-policy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
haxor@linux:~$
```

Ok good, we're running Ubuntu 20.04.4 LTS as the `PRETTY_NAME=` indicates.

Now immediately execute `sudo apt update` and `sudo apt upgrade` as two separate commands, don't ask why just do it.

A.1.1 Python Environment

Next, we need to have Python installed. Python is the programming language and runtime that Jaseci is primarily built upon. It's also the language that 99.999% of everyone uses for AI research and products (and myriad other things). It's also my favorite as of late, well, second favorite after Jac. Lets check to see. Simply enter the command,

```
haxor@linux:~$ python3 --version
-bash: python3: command not found
haxor@linux:~$
```

Some of you at this point might see a python version that is ¿= 3.8. If you see this you're good, you have Python installed. We don't see this in this example. That is because we have the *minimal* Ubuntu. So we have to install it.

```
haxor@linux:~$ sudo apt install python3 python3-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-common binutils-x86-64-linux-gnu build-essential ca-
      ↪ certificates cpp cpp-9 dirmngr dpkg-dev fakeroot g++ g++-9 gcc
      ↪ gcc-9 gcc-9-base gnupg
...
Do you want to continue? [Y/n] y
...
Processing triggers for libc-bin (2.31-0ubuntu9.7) ...
Processing triggers for ca-certificates (20210119~20.04.2) ...
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
haxor@linux:~$
```

The line `sudo apt install python3 python3-pip` instructs Ubuntu to install both the `python3` package as well as the `python3-pip` package. Note in the example there is a point where it will ask you if you want to continue, just press Y and let it go. This step could take some time in principle, but we are almost there!

Lets next check again that we have python installed.

```
haxor@linux:~$ python3 --version
Python 3.8.10
haxor@linux:~$ pip --version
pip 20.0.2 from /usr/lib/python3/dist-packages/pip (python 3.8)
haxor@linux:~$
```

Yes! We're in great shape, we've also checked that `pip` is install and that looks good as well. Note that we can also replace `pip` with `pip3` and everything should work as well.

A.1.2 Installing Jaseci

Now that we have Python setup, we can use the `pip` install Jaseci itself. `pip` is Python's official package manager. This command line tool allows users of Python to install packages or code libraries that go beyond the standard libraries that come with Python out of the box. There is a public repository of libraries that is open to all the haxors of the world called PyPI [7] that houses pretty much all the published python packages of the world. Jaseci lives there throuh two packages, `jaseci` and `jaseci-serv`. For the moment we need only concern ourselves with `jaseci` as we get started. When we're ready to launch amazing tech stacks to production on scalable cloud infrastructure we'll pull down `jaseci-serv`.

Now, lets install Jaseci!

```
haxor@linux:~$ pip install jaseci
Collecting jaseci
  Downloading jaseci-1.3.1.1-py3-none-any.whl (154 kB)
     |||||||||||||||||||||||||||||||||| 154 kB 4.5 MB/s
...
Successfully installed jaseci-1.3.1.1
haxor@linux:~$
```

TADA! We've pulled down Jaseci and are good to go! In this case we've installed Jaseci version `1.3.1.1`, your version should be at least this one but probably higher depending on when you're reading this. If its say a year after this moment that I'm writing this book and it's still 1.3.1.1, something very very wrong has happened. Indeed, if its two weeks later and nothing has changed, call 911 and report a missing person, seriously.

To validate that everything works, lets check the command line tool `jsctl` is present. `jsctl` is a command line tool that give full control and access to the Jaseci computational model. In particular, and for the sake of this chapter, we will use this tool to build and run programs, generate source for visualizing data and graphs, building artificial intelligence (AI) programs, hot loading fancy AI models, pushing implementations live to Jaseci servers and much much more. Now lets make sure we have access to this very powerful cli tool.

```
haxor@linux:~$ 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:
  actions Group of ‘actions‘ commands
  alias Group of ‘alias‘ commands
  architype Group of ‘architype‘ commands
  clear Clear terminal
  config Group of ‘config‘ commands
  edit Edit a file
  global Group of ‘global‘ commands
  graph Group of ‘graph‘ commands
  jac Group of ‘jac‘ commands
  logger Group of ‘logger‘ commands
  login Command to log into live Jaseci server
  ls List relevant files
  master Group of ‘master‘ commands
  object Group of ‘object‘ commands
  reset Reset jsctl (clears state)
  sentinel Group of ‘sentinel‘ commands
  stripe Group of ‘stripe‘ commands
  tool Internal book generation tools
  walker Group of ‘walker‘ commands
haxor@linux:~$
```

If you see this output, you're in business!! If you don't, something went wrong and you should phone a friend, (but first make sure you didn't miss anything above).

A.1.3 VSCode and the Jac Language Extension

This is technically optional but... I strongly recommend you install and use VSCode with Jaseci. VSCode IMHO, is the best code editor on the planet. I regard it as the choice Sake to sip with my Jaseci omakase. Personally, I use an Ubuntu flavored WSL VSCode environment.

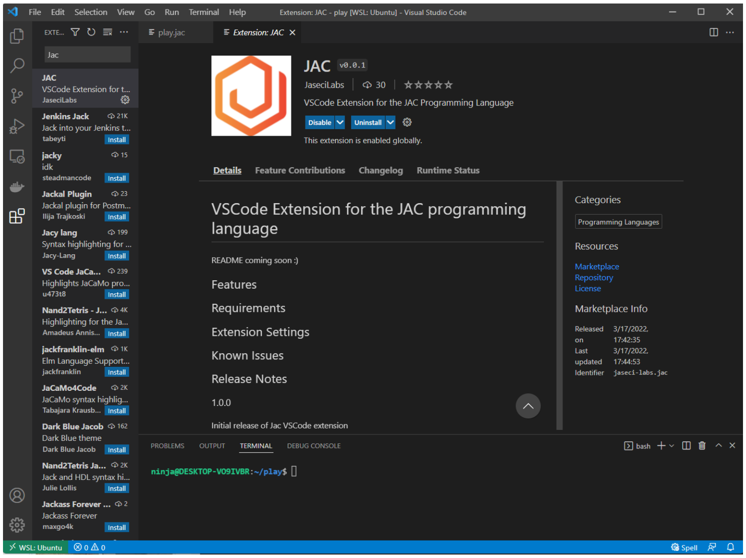


Figure A.1: The Wonderful Jac Language extension in VSCode.

In VSCode, you can search for and install the Jac language extensions as per Figure A.1. As you can see, at the time I clipped this image, its quite new and doesn't really have a readme. You won't need one, it just provides syntax highlighting for `.jac` files at the moment. But it makes Jac code look beautiful, so it's a must have.

A.2 Coding in Jac

Jac, which is short hand for **Ja**seci **C**ode, is a programming language designed for building programs for Jaseci. The language itself is inspired by a mixture of Javascript and Python and can be used standalone or as glue code for libraries built in other languages ecosystems. Jac is to Python, what Python is to C, what C is to assembly language for scalable sophisticated applications running in the cloud. In this section, we'll cover basics to advanced assuming no programming experience. Though we'll try to cover everything from first time coders to pros, we'll move fast through some of the rudimentary concepts so have your Google ready if you need to drill in a bit more of some of the basic programming concepts. Lets Jump in!

A.2.1 Jac Basics

Launch VSCode, spool up a terminal window, and lets tinker with an example. We'll start with Jac Code A.1. I'd strongly recommend you type out this example (instead of cutting and pasting) especially if this might be your first time programming or are a little rusty with Python and or Javascript. It's the best way to learn!

Jac Code A.1: Example program introducing basic syntax.

```
1  walker init {
2      x = 34 - 30; # This is a comment
3      y = "Hello";
4      z = 3.45;
5
6      if(z==3.45 or y=="Bye"){ # if statement with only thing true
7          x=x-1;
8          y=y+"␣World"; # the + between two strings concatinate them
9      }
10
11     std.out(x);
12     for i=0 to i<3 by i+=1: # For loop with single line block style
13         std.out(x-i,'-', y); # prints to screen
14     report [x, y+'s']; # adds data to payload
15 }
```

This first example Jac Code A.1 shows a simple program example demonstrating a number of basic language features. Firstly, observe that the first three assignments in the program to `x`, `y`, and `z` does not specify any types indicating that Jac is a dynamically typed language. This means the types are inferred from the assignment of variables, and these types can change dynamically as new assignments are applied to the same variables. This feature is designed to work almost exactly like the dynamic typing in Python.

Next we find a conditional statement much like any other language. Do note operators like the Python inspired `or` is supported along side the C/C++/Javascript `||` operator. Other such operators include `and` (`\&\&`), `not` (`!`), etc.

After the conditional we have a library call `std.out(x)` on line 11. This call prints the value of x to the screen. `std.out` in Jac is equivalent to the the `print` in Python and analogous to the `printf`, `cout`, and `console.log` you'd find in C, C++, and Javascript respectively. A suite of core standard library operations for the language has the preamble of `std`.

Output:

```
3
3 - Hello World
2 - Hello World
1 - Hello World
{
  "success": true,
  "report": [
    [
      3,
      "Hello Worlds"
    ]
  ]
}
```

A.2.2 Types in Jac

[Types example]

Jac Code A.2: First Example

```
walker init {
    a=5;
    b=5.0;
    c=true;
    d='5';
    e=[a, b, c, d, 5];
    f={'num': 5};

    summary = {'int': a, 'float': b, 'bool': c,
               'string': d, 'list': e, 'dict': f};

    std.out(summary);
}
```

Output:

```
{"int": 5, "float": 5.0, "bool": true, "string": "5", "list": [5, 5.0,
    ↪ true, "5", 5], "dict": {"num": 5}}
```

A.2.3 Fun with Lists and Dictionaries

[Fun with Lists and Dictionaries]

Jac Code A.3: First Example

```
walker init {
    d = {'four':4, 'five':5};
    b = d.dict::copy; # equal to b=d.d::copy;
    b['four'] += b['five'];
    std.out(d.d::keys, d.d::values, d.d::items, b.d::items);

    b_vals = b.d::values;
    b_vals.list::append(6.5); # equal to b=d.d::copy;
    std.out(b_vals);
    b_vals.l::sort; std.out(b_vals);
    b_vals.l::reverse; std.out(b_vals);
}
```

Output:

```
["four", "five"] [4, 5] [["four", 4], ["five", 5]] [["four", 9], ["five",
    ↪ 5]]
[9, 5, 6.5]
[5, 6.5, 9]
[9, 6.5, 5]
```

A.2.4 Control Flow

[Fun with Control Flow]

Jac Code A.4: First Example

```
walker init {
    fav_nums=[];

    for i=0 to i<10 by i+=1:
        fav_nums.l::append(i*2);
    std.out(fav_nums);

    fancy_str = "";
    for i in fav_nums {
```

```
        fancy_str = fancy_str + "two * " + i.str +
                        " = " + (i*2).str + ", ";
    }
    std.out(fancy_str);

    count_down = fav_nums[-1];
    while (count_down > 0) {
        count_down -= 1;
        if (count_down == 14):
            continue;
        std.out("I'm at countdown "+count_down.str);
        if (count_down == 10):
            break;
    }
}
```

Output:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
two * 0 = 0, two * 2 = 4, two * 4 = 8, two * 6 = 12, two * 8 = 16, two *
    ↪ 10 = 20, two * 12 = 24, two * 14 = 28, two * 16 = 32, two * 18 =
    ↪ 36,
I'm at countdown 17
I'm at countdown 16
I'm at countdown 15
I'm at countdown 13
I'm at countdown 12
I'm at countdown 11
I'm at countdown 10
```

A.2.5 Graphs in Jac

[Bringing Graphs in with special operators]

Jac Code A.5: First Example

```
node person {
    has name="Anon";
}

edge strong;
edge weak;
```

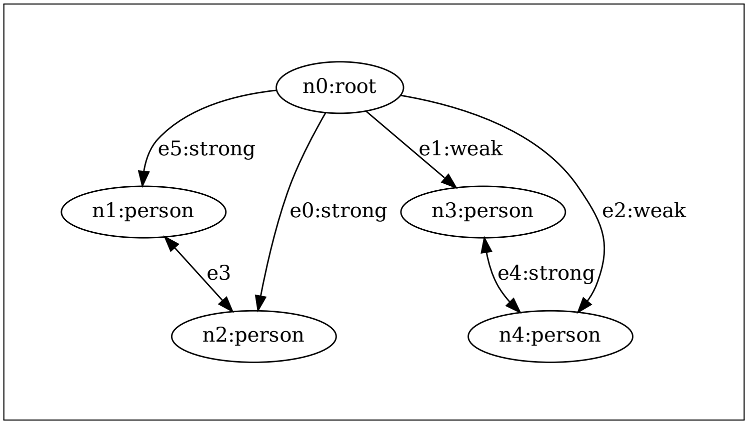


Figure A.2: Graph in memory for JC A.5

```

walker init {
        person1 = spawn here -[strong]-> node::person(name="Joe");
        person2 = spawn here -[strong]-> node::person;
        person3 = spawn here -[weak]-> node::person;
        person4 = spawn here -[weak]-> node::person(name="Mike");

        person1 <--> person2;
        person3 <-[strong]-> person4;

        for i in -->:
            std.out(i.context);
}
```

Output:

```
{"name": "Joe"}
{"name": "Anon"}
{"name": "Anon"}
{"name": "Mike"}
```

A.2.6 Navigating Graphs with Walkers

[Walking Graphs]

Jac Code A.6: First Example

```
node state {
    has response="I'm silly state ";
}

node hop_state;

edge hop;

walker init {
    has state_visits=0, save_root;

    root {
        save_root = here;
        hop1 = spawn here -[hop]-> node::hop_state;
        hop2 = spawn here -[hop]-> node::hop_state;
    }

    hop_state:
        spawn here walker::hop_buildout;

    state {
        state_visits += 1;
        std.out(here.response+state_visits.str);
    }

    take -->;
    with exit {
        report spawn save_root walker::hop_counter;
    }
}

walker hop_buildout {
    spawn here --> node::state;
    spawn here --> node::state;
    spawn here --> node::state;
}

walker hop_counter {
    has anchor num=0; take -->; hop_state { num+=1; }
}
```

Output:

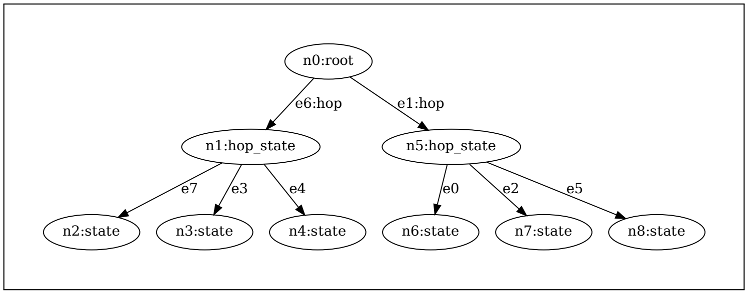


Figure A.3: Graph in memory for JC A.6

```
I'm silly state 1
I'm silly state 2
I'm silly state 3
I'm silly state 4
I'm silly state 5
I'm silly state 6
{
  "success": true,
  "report": [
    2
  ]
}
```

A.2.7 Compute in Nodes

[Compute into the Nodes]

Jac Code A.7: First Example

```
node state {
    has name = rand.word().str::upper;
    has response = "I'm a silly bot. ";
    has user_utter;

    can speak with entry {
        std.out("I'm "+name+". And I currently have " + visitor.info['name'] +
                "  on me! ");
    }

```

```
    can listen with talker exit {
        user_utter = visitor.utterance;
        std.out("I␣heard␣'"+user_utter+"'\n");
        std.out(response);
    }

    can test_path with hop_counter entry {
        visitor.path.l::append(&here);
    }
}

walker init {
    root {
        n1 = spawn here --> node::state;
        n2 = spawn here --> node::state;
    }
    spawn here walker::talker;
    spawn here walker::hop_counter;
}

walker talker {
    has utterance, path = [];
    utterance = rand.sentence();
    take -->;
}

walker hop_counter {
    has anchor path = [];
    take -->;

    with exit { std.out("\nHopper's␣path:", path); }
}
```

Output:

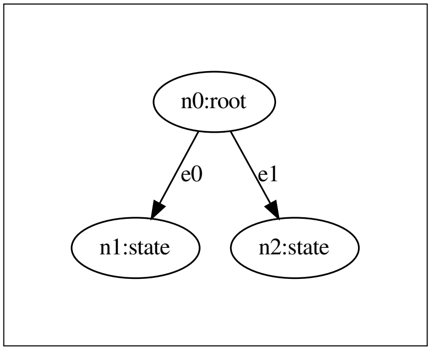


Figure A.4: Graph in memory for JC A.7

```
I'm DOLOREM. And I currently have talker on me!
I heard 'Magnam quaerat ut qui velit consectetur consectetur.'

I'm a silly bot.
I'm EIUS. And I currently have talker on me!
I heard 'Quisquam eius numquam amet ut porro velit amet numquam ut.'

I'm a silly bot.
I'm DOLOREM. And I currently have hop_counter on me!
I'm EIUS. And I currently have hop_counter on me!

Hopper's path: ["urn:uuid:d5be01eb-db6f-4692-9471-05ccf081ffc1", "urn:
    ↪ uuid:e7dd97bf-050c-4b36-afa5-38963935c933"]
```

A.2.8 Static Graphs

[Static graphs]

Jac Code A.8: First Example

```
node person {
    has name="Anon";
}

edge strong;
edge weak;

graph basic_gph {
```

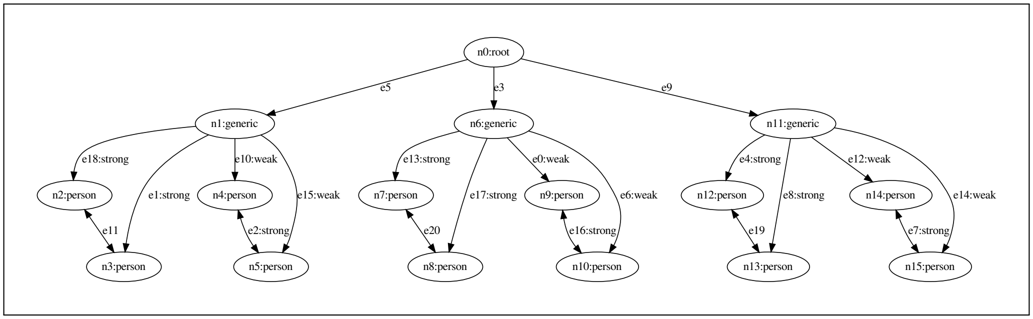


Figure A.5: Graph in memory for JC A.8

```
    has anchor root;
    spawn {
        root = spawn node::generic;
        person1 = spawn root -[strong]-> node::person(name="Joe");
        person2 = spawn root -[strong]-> node::person;
        person3 = spawn root -[weak]-> node::person;
        person4 = spawn root -[weak]-> node::person(name="Mike");

        person1 <--> person2;
        person3 <-[strong]-> person4;
    }

}

walker init {
    spawn here --> graph::basic_gph;
    spawn here --> graph::basic_gph;
    spawn here --> graph::basic_gph;
}
```

A.2.9 Writing Tests

[Tests]

Jac Code A.9: First Example

```
node person: has name="Anon";

graph basic {
```

```
    has anchor root;
    spawn {
        root = spawn node::generic;
        person1 = spawn root --> node::person(name="Joe");
        person2 = spawn root --> node::person;
        person3 = spawn root --> node::person;
        person4 = spawn root --> node::person(name="Mike");
        person1 <--> person2;
        person3 <--> person4;
    }

}

walker tally {
    has count=0, visited=[];
    count += 1;

    if(here not in visited) {
        visited.l::append(here);
        take -->;
    }
}

test "Size␣of␣basic␣graph"
with graph::basic by walker::tally {
    assert(visited.length == 5);
    assert(count > 5);
}

test "Size␣of␣a␣bit␣fancier␣graph"
with graph {
    has anchor root;
    spawn {
        root = spawn node::generic;
        spawn root --> graph::basic; spawn root --> graph::basic;
    }
} by walker::tally {
    assert(visited.length == 11);
    assert(count > 11);
}
```

Output:

```
Testing "Size of basic graph": [PASSED in 0.00s]
Testing "Size of a bit fancier graph": [PASSED in 0.01s]
{
  "tests": 2,
  "passed": 2,
  "failed": 0,
  "success": true
}
```

A.3 Jac Hacking Workflow

In this section, we discuss a typical workflow and organization of a Jac coding project. To this end, we will be creating a simple toy chatbot project and examine it's file organization and development workflow. First, lets take a look at the files for this project.

```
haxor@linux:~/toybot$ ls
cai.jac edges.jac faq_answers.txt load_faq.jac nodes.jac static_conv.jac
    ↪ tests.jac
haxor@linux:~$
```

Now lets take a look a what each of these files represent:

- `cai.jac` - This is the main file for the project to which the various other elements (nodes, edges, graphs, etc) are imported from other files in the directory.
- `nodes.jac` - This file houses the node architypes created for this application. Functionality is specified in both the walkers and as node abilities.
- `edges.jac` - This file contains the edge architypes we've specified in the design of our conversational AI. These edges represent various types of transitions we can make throughout the converstation.
- `static_conv.jac` - This file contains a static conversational graph that represents the posible conversational flows via state nodes and transition edges.
- `load_faq.jac` - This file contains a static constructor for graph elements to correspond to frequently asked questions by loading them from a file.
- `faq_answers.txt` - This file specifies a list of answers to frequently asked questions, we'll be using a model that only depends on the answers themselves.
- `tests.jac` - This file is where we house all the tests for our project.

A.3.1 Using Imports

Jac Code A.10: Main CAI Jac App

```
import {node::{state, hop_state}} with "./nodes.jac";
import {edge::{trans_ner, trans_intent, trans_qa}} with "./edges.jac";
import {graph::basic_gph} with "./static_conv.jac";
import {graph::faq_gph} with "./load_faq.jac";



walker init {
    root {
        spawn here --> graph::basic_gph;
        spawn -->[0] -[trans_intent(intent="about chat bots")]-> graph::faq_gph;
    }
    with exit {
        spawn -->[0] walker::talker;
    }
}

walker talker {
    has utterance="";
    has use_cmd = true, path = [];
    if(use_cmd and here.details['name'] != 'hop_state'):
        utterance = std.input("> ");
    take -->;
}
```

Jac Code A.11: Nodes for CAI

```
node state {
    has name = rand.word();
    has response="I'm a silly bot.";
    has user_utter;

    can speak with entry {
        std.out(response + " I'm current on "+name+" node");
    }

    can listen with talker exit {
        user_utter = visitor.utterance;
        visitor.path.l::append(&here);
```

```
        std.out("I␣heard␣"+user_utter+".");
    }

    can test_path with get_states entry {
        visitor.path.l::append(&here);
    }
}

node hop_state {
    has name;
    can log with exit {
        std.log("A␣walker␣is␣walking␣right␣over␣me.");
    }
}
```

Jac Code A.12: edges for CAI

```
edge trans_ner { has entities; }
edge trans_intent { has intent; }
edge trans_qa { has embed; }
```

A.3.2 Leveraging Static Graphs for Quick Prototyping

Jac Code A.13: Static Conversational Graph

```
import {edge::{trans_ner, trans_intent, trans_qa}} with "./edges.jac";
import {node::{state, hop_state}} with "./nodes.jac";

graph basic_gph {
    has anchor conv_root;
    spawn {
        conv_root = spawn node::state(name="Conv␣Root");

        appt = spawn conv_root -[trans_intent(intent="appointment")]->
            node::hop_state(name="Appointments");

        spawn appt -[trans_intent(intent="create")]->
            node::state(name="Create␣an␣appoitnment");
        spawn appt -[trans_intent(intent="cancel")]->
            node::state(name="Cancel␣an␣appoitnment");
        spawn appt -[trans_intent(intent="reschedule")]->
            node::state(name="Reschedule␣an␣appoitnment");
```

```

        service = spawn conv_root -[trans_intent(intent="service␣info")]->
            node::hop_state(name="Services");

        spawn service -[trans_intent(intent="manicures")]->
            node::state(name="About␣manicures");
        spawn service -[trans_intent(intent="haircuts")]->
            node::state(name="About␣haircuts");
        spawn service -[trans_intent(intent="makeup")]->
            node::state(name="About␣makeup");
    }

}
```

A.3.3 Test Driven Development

Jac Code A.14: Tests for CAI

```
import {*} with "./cai.jac";

walker get_states {
    has anchor path = [];
    take -->;
}

test "Travesal␣touches␣all␣nodes"
with graph::basic_gph by walker::get_states {
    std.out(path.length);
    assert(path.length==7);
}
```

A.3.4 File I/O

Jac Code A.15: FAQ Graph Loader

```
import {edge::{trans_ner, trans_intent, trans_qa}} with "./edges.jac";
import {node::{state, hop_state}} with "./nodes.jac";

graph faq_gph {
    has anchor faq_root;
```

```
    spawn {
        faq_root = spawn node::state(name="Faq Root");

        answers = file.load_str('./faq_answers.txt').str::split('&&&');

        for i in answers:
            spawn faq_root -[trans_qa]-> node::state(response=i);
    }

}
```

```
A chatbot is an artificial intelligence (AI) based computer program that
    ↪ can interact with a human either via voice or text through
    ↪ messaging applications, websites, mobile apps or through the
    ↪ telephone.
&&&
Conversational chatbots have been around for decades now. In the past,
    ↪ there have been many unsuccessful attempts to build a chatbot that
    ↪  successfully mimics human conversation. However, not thats solved
    ↪  with the creation of me!
&&&
During the chatbot design process, it is important to keep your user in
    ↪ mind as it will help you define the right chatbot features,
    ↪ functionality and build human-like interactions.
&&&
In order for a chatbot to function properly, it is crucial for the
    ↪ program to access your knowledge base, website, internal databases
    ↪ , existing documents, or other sources of information.
```

A.3.5 Building to JIR

A.4 AI with Jaseci Kit

A.4.1 Installing Jaseci Kit

```
haxor@linux:~$ pip install jaseci-kit
Collecting jaseci-kit
  Downloading jaseci_kit-1.3.3.5-py3-none-any.whl (34 kB)
Collecting tensorflow<3.0.0,>=2.8.0
  Downloading tensorflow-2.8.0-cp38-cp38-manylinux2010_x86_64.whl (497.6
      ↪ MB)
     |||||||||||||||||||||||||||||||||| 497.6 MB 8.9 MB/s
...
Successfully installed ... jaseci-kit-1.3.3.5 ...
haxor@linux:~$
```

A.4.2 Loading Actions from Jaseci Kit

```
haxor@linux:~$ jsctl -m
Starting Jaseci Shell...
jaseci > actions list
[
  "net.max",
  "net.min",
  "net.root",
  "rand.seed",
  ...
  "date.quantize_to_month",
  "date.quantize_to_week",
  "date.quantize_to_day",
  "date.date_day_diff"
]
jaseci >
```

```
jaseci > actions load module jaseci_kit.use_qa
2022-04-16 22:01:52.612881: W tensorflow/stream_executor/platform/default
    ↪ /dso_loader.cc:64] Could not load dynamic library 'libcudart.so
    ↪ .11.0'; dlerror: libcudart.so.11.0: cannot open shared object file
    ↪ : No such file or directory
2022-04-16 22:01:52.612908: I tensorflow/stream_executor/cuda/cudart_stub
    ↪ .cc:29] Ignore above cudart dlerror if you do not have a GPU set
    ↪ up on your machine.
2022-04-16 22:02:05.269074: W tensorflow/stream_executor/platform/default
    ↪ /dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1';
    ↪ dlerror: libcuda.so.1: cannot open shared object file: No such
    ↪ file or directory
2022-04-16 22:02:05.269104: W tensorflow/stream_executor/cuda/cuda_driver
    ↪ .cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2022-04-16 22:02:05.269127: I tensorflow/stream_executor/cuda/
    ↪ cuda_diagnostics.cc:156] kernel driver does not appear to be
    ↪ running on this host (vanillabox-589f9b897c-k2ncs): /proc/driver/
    ↪ nvidia/version does not exist
2022-04-16 22:02:05.269232: I tensorflow/core/platform/cpu_feature_guard.
    ↪ cc:151] This TensorFlow binary is optimized with oneAPI Deep
    ↪ Neural Network Library (oneDNN) to use the following CPU
    ↪ instructions in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the
    ↪ appropriate compiler flags.
{
  "success": true
}
jaseci >
```

```
jaseci > actions list
[
  "net.max",
  "net.min",
  "net.root",
  "rand.seed",
  ...
  "date.quantize_to_month",
  "date.quantize_to_week",
  "date.quantize_to_day",
  "date.date_day_diff",
  "use.question_encode",
  "use.enc_question",
  "use.answer_encode",
  "use.enc_answer",
  "use.cos_sim_score",
  "use.dist_score",
  "use.qa_score"
]
jaseci >
```

A.4.3 Using AI in Jac

[Adding some AI]

Jac Code A.16: Universal Sentence Encoding QA in Jac

```
walker init {
    can use.enc_question, use.enc_answer;

    answers = ['I am 20 years old', 'My dog is hungry', 'My TV is broken'
        ];
    question = "If I wanted to fix something what should I fix?";

    q_enc = use.enc_question(question);
    a_enc = use.enc_answer(answers); # can take lists or single strings

    a_scores=[];

    for i in a_enc:
        a_scores.l::append(vector.cosine_sim(q_enc, i));

```

```
    report a_scores;
}
```

Output:

```
{
  "success": true,
  "report": [
    [
      0.010415227400767156,
      0.034413563053388725,
      0.08458081860660219
    ]
  ]
}
```

A.5 Launching a Jaseci Web Server

A.6 Deploying Jaseci at Scale

A.6.1 Quick-start with Kubectl

A.6.2 Managing Jac in Cloud

Appendix B

Rants

B.1 Utilizing Whitespace for Scoping is Criminal (Yea, I'm looking at you Python)

This whitespace debauchery perpetrated by Python and the like is one of the most perverse 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!"

B.2 "Using 'master' in Git is NOT Racist!" by a black dude

[Insert Rant Here]

Appendix C

Full Jac Grammar Specification

Grammar C.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 namespaces? LBRACE attr_stmt* walk_entry_block? (
                statement
                | walk_activity_block
        )* walk_exit_block? RBRACE;

ver_label: 'version' COLON STRING SEMI?;

namespaces: COLON name_list;

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 name_list? (KW_ENTRY | KW_EXIT | KW_ACTIVITY);

preset_in_out:
        DBL_COLON expr_list? (DBL_COLON | COLON_OUT expression);

dotted_name: NAME DOT NAME;

name_list: NAME (COMMA NAME)*;

expr_list: expression (COMMA expression)*;

code_block: LBRACE statement* RBRACE | COLON statement;
```

```
node_ctx_block: name_list code_block;

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

destroy_action: KW_DESTROY expression SEMI;

report_action: KW_REPORT expression SEMI;

walker_action: ignore_action | take_action | KW_DISENGAGE SEMI;

ignore_action: KW_IGNORE expression SEMI;

take_action: KW_TAKE expression (SEMI | else_stmt);

expression: connect (assignment | copy_assign | inc_assign)?;

assignment: EQ expression;

copy_assign: CPY_EQ expression;
```

```
inc_assign: (PEQ | MEQ | TEQ | DEQ) expression;

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

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

compare: NOT compare | arithmetic (cmp_op arithmetic)*;

cmp_op: EE | LT | GT | LTE | GTE | NE | KW_IN | nin;

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:
        atom (LPAREN expr_list? RPAREN)?
        | atom? DBL_COLON NAME spawn_ctx?;

atom:
        INT
        | FLOAT
        | STRING
        | BOOL
        | NULL
        | NAME
        | node_edge_ref
        | list_val
        | dict_val
        | LPAREN expression RPAREN
        | spawn
        | atom DOT built_in
        | atom DOT NAME
        | atom index_slice
        | ref
        | deref
        | any_type;
```

```
ref: '&' expression;

deref: '*' expression;

built_in:
        cast_built_in
        | obj_built_in
        | dict_built_in
        | list_built_in
        | string_built_in;

cast_built_in: any_type;

obj_built_in: KW_CONTEXT | KW_INFO | KW_DETAILS;

dict_built_in: KW_KEYS | LBRACE name_list RBRACE;

list_built_in: KW_LENGTH | KW_DESTROY COLON expression COLON;

string_built_in:
        TYP_STRING DBL_COLON NAME (LPAREN expr_list RPAREN)?;

node_edge_ref:
        node_ref filter_ctx?
        | edge_ref (node_ref filter_ctx?)?;

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 (spawn_ctx | filter_ctx)? ']')? '->';

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

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

list_val: LSQUARE expr_list? RSQUARE;

index_slice:
        LSQUARE expression RSQUARE
        | LSQUARE expression COLON 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 (spawn_assign (COMMA spawn_assign)*)? RPAREN;

filter_ctx:
        LPAREN (filter_compare (COMMA filter_compare)*)? RPAREN;

spawn_assign: NAME EQ expression;

filter_compare: NAME cmp_op expression;

any_type:
        TYP_STRING
        | TYP_INT
        | TYP_FLOAT
        | TYP_LIST
        | TYP_DICT
        | TYP_BOOL
        | KW_NODE
        | KW_EDGE
        | KW_TYPE;
```

```

/* 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 */
TYP_STRING: 'str';
TYP_INT: 'int';
TYP_FLOAT: 'float';
TYP_LIST: 'list';
TYP_DICT: 'dict';
TYP_BOOL: 'bool';
KW_TYPE: 'type';
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_CONTEXT: 'context';
KW_INFO: 'info';
KW_DETAILS: 'details';
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';
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]+;
NULL: 'null';
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: .;
```

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.

[7] The Python Foundation. Pypi: The python package index.