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 in places, please take that seriously :-)

Contents

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 all the way through 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. There are indeed places that I take that professory tone, but expect places where we have fun. (Truthfully, thats the only way I can maintain sanity writing this tome! :-P)

Oh, and 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 an 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?

Contents

2.1 TL;DR

Modern production applications are *multi-service*, spanning multiple individual programs (database, memcache, logging, application logic, AI models, etc) interfacing each other over APIs to realize a single product functionality. Creating such applications at scale is technically challenging, requires a highly-skilled developer team, is rife with complexity, and is, for many, prohibitively costly. This complexity is in stark contrast to the era of computing where a state of the art software product was a single binary that ran on one machine and could be developed by a single programmer. Though a number of important abstractions and technologies have emerged to help mitigate the complexity of building multi-service applications, the creation of sophisticated production software in practices is still highly complex and requires a team of engineers.

In this work, we present a wholistic top-down re-envisioning of the system stack from the programming language level down through the system architecture to bridge this complexity gap. The key goal of our design is to address the critical need for the programmer to articulate solutions with higher level abstractions at the problem level while having the runtime system stack subsume and hide a broad scope of diffuse sub-applications and inter-machine resources. This work also presents the design of a production-grade realization of such a system stack architecture called **Jaseci**, and corresponding programming language **Jac**. Jac and Jaseci has been released as open source and has been leveraged by real product teams to accelerate developing and deploying sophisticated AI products and other applications at scale. Jac has been utilized in commercial production environments to accelerate AI development timelines by ∼10x, with the Jaseci runtime automating the decisions and optimizations typically falling in the scope of manual engineering roles on a team such as what should and should not be a microservice and changing those decisions dynamically.

2.2 Introduction and Motivation

There has been a fundamental paradigm shift in the landscape of how we build software over the last 2 decades. Originally, the compute stack was envisaged with the assumption that a single program would run on a single machine. In this traditional model, system software abstractions subsumed the management of resources for processor, memory, disk and physically connected peripherals within the context of the machine. However, this landscape rapidly changed with the evolution toward software being served on the backbone provided by the internet. Now, an 'application' is realized through the cooperation of multiple distinct sub-applications (services) running collaboratively. For example a single application my contain one or more self-contained database, memcache, logging, application logic, and AI model applications interfacing each other over APIs as shown in Figure 2.1 (left). We call these applications *diffuse applications*.

This work contends that the fundamental programming paradigms in computing has not evolved at pace. The abstractions envisioned during the era of the single machine computational model is still present at the programming interface and throughout the runtime stack leading to significant and costly complexity.

To address this complexity, two keystone abstractions have recently emerged to facilitate the development of these diffuse applications. The first of these abstractions is the introduction and rapid dissemination of containerization service platforms. With what started as a key insight articulated in "The Datacenter as a Computer," Google would innovate their Borg system and ultimately released it open source as Kubernetes. With Kubernetes, the underlying hardware resources would be abstracted away with the introduction of *pods* (virtual machines), and other resources that can be virtually networked together and otherwise configured irrespective of the physical hardware. Today, Kubernetes is the most prevalent containerized service abstraction layer in cloud computing. The second of keystone abstraction would be coined "Severless Computing" and gained prominence with the introduction of Amazons Lamda functions. This FaaS abstraction would facilitate the development of diffuse applications at the level of functions and abstract away the underlying

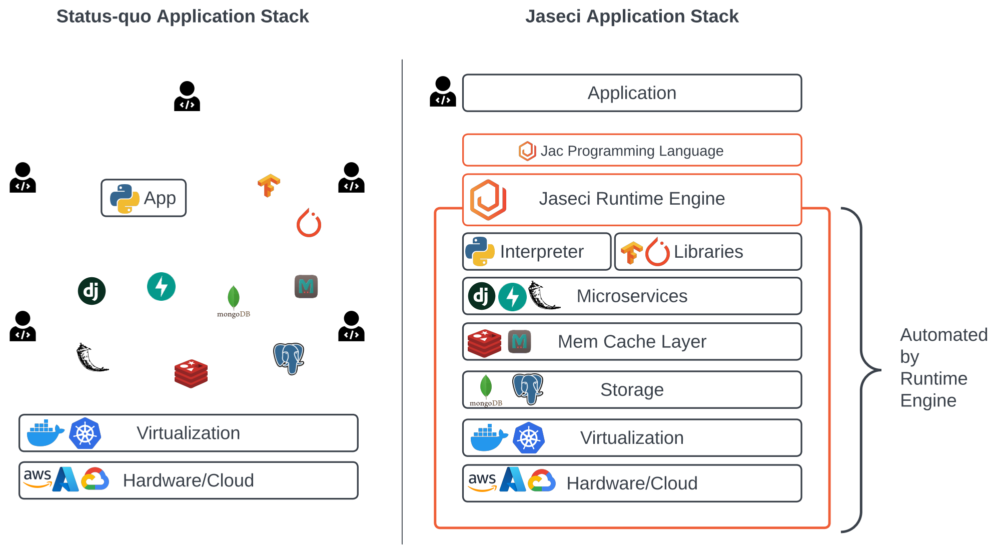


Figure 2.1: Comparison between status quo development of production grade diffuse *applications (left), and the Jaseci technology stack that hides and automates an expanded set of subsystems through raising the level of abstraction (right).*

containerized service ecosystem. A programer can simply make function calls in their favorite language without every needing to be aware of where the function will run nor the system level resources that would be allocated or managed.

Though these two abstractions have been highly impactful, these innovations in our stack architecture represent a bottom up evolution of abstractions. As a result, programmers are still left with single-machine abstractions at the programming interface and must grapple with a significant amount of complexity. For example, traditional languages and their runtime stacks are predominately designed with the goal of hiding and managing intra-machine resources while what is needed for diffuse applications is the hiding and management of inter-machine resources. Analogous to the virtualization and management of allocated memory on the heap provided by garbage collectors in modern languages (intra-machine), the virtualization and management of resources such as microservice creation, scheduling and orchestration alongside policies for organizing distributed databases, mem caches, logging and other highly complex subsystems (inter-machine) is not only needed, but as we show in this work, possible and practical. Without this raising of the level of abstraction, it has become prohibitively difficult for a single engineer to invent, build, deploy, launch, and scale modern cutting edge applications.

To the best of our knowledge, we are not aware of a thorough, wholistic, and top-down design of a serverless programming paradigm and computational stack from the language level down through the system runtime stack to hide this expanded set of resources.

In this work, we present a wholistic design approach with the goal of abstracting away and automating a new class of underlying systems, allowing a programmer to articulate solutions and diffuse applications at the problem level. We present the design of a *diffuse runtime execution engine* we call **Jaseci**, and a *data-spacial programming language* we call **Jac**.

The design of Jaseci and Jac has initially been inspired to by sophisticated emerging AI applications at scale and is driven by two key insightss.

- *Higher level abstractions are needed at the language level to allow single creators to work at the problem level to build end-to-end diffuse AI products.*
- *A new set of abstractions across the language runtime and system stack is needed to automate and hide the class of inter-machine resources from the programmer.*

To this end we present techniques across two categories,

1. *Jac Language* - A language that introduces a new set of abstractions, namely **data-spacial scoping** and **agent oriented programming**. These abstractions natively facilitates the emerging need to reason about and solve problems with graph representations as well as the need for algorithmic modularity and encapsulation to hide a new class of inter-machine resources.
2. *Jaseci Diffuse Runtime Engine* - A runtime that raises the abstraction layer to the problem solving level where the runtime engine subsumes responsibility for not only for the optimization of program code, but the orchestration, configuration, and optimization of the full cloud compute stack and inter-machine resources (such tasks as container formation, scaling and optimization).

Jaseci and Jac is fully functional, open-source [7, 8, 9], and used in production for four real-world products today. These commercial products were built entirely on the Jaseci staci and includes Myca [10], HomeLendingPal [6], ZeroShotBot [14] and TrueSelph [13]. Across these and other projects, the Jac language has been used by dozens of programmers in the creation of production software and Jaseci deployments support tens of thousands of production queries per day currently. In practice, our initial infrastructure has been leveraged in practice to achieve 10x reduction in development time and near 100% elimination of typical backend code needed for a complicated AI based application.

The specific contributions of this paper include:

- We formulate the problem of development complexity and present a top down programing paradigm and runtime stack for diffuse applications.
- We describe the design and implementation Jaseci's **diffuse runtime execution engine**.

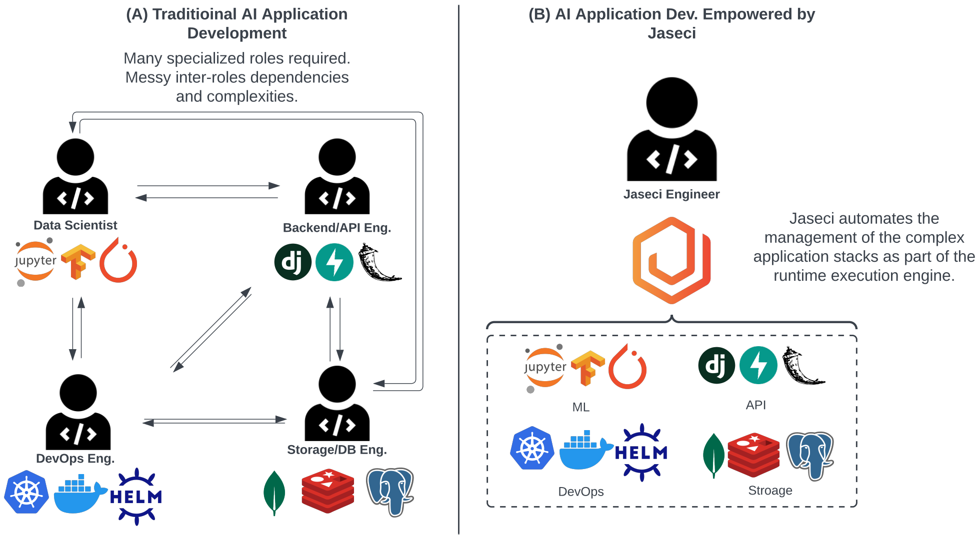


Figure 2.2: Comparison of typical development team required to realize production grade AI application today (left), and the ability of a single software developer to realize such an application with Jaseci (right).

- We introduce Jac, a language that implements a **data-spacial** programming paradigm (the first of its kind).
- We describe the utility of Jaseci and Jac through real world case studies of building out a real production scale-out product.

We find that the wholistic design philosophy and resulting paradigm of Jaseci and Jac is a promising one. Multiple development teams have adopted the data spacial programming model of Jac and the diffuse runtime execution engine in Jaseci to build sophisticated AI products with significantly reduced complexity and teaming.

2.3 The Case for Change

Though recent advancements in serverless computing has been instrumental in improving the ability of teams to more rapidly develop software, significant challenges remain in the development of cutting edge applications and products in our current compute landscape. An demonstrative problem domain with this challenge are those characterized by applications that include sophisticated AI pipelines on their critical path.

2.3.1 Problem Scenario

Figure 2.2A shows the typical set of often siloed roles needed to create software in this environment. The first critical role needed is an *architect / tech-lead* responsible for architecting the software solution across disparate components, programming languages, frameworks, and SDKs. If a microservice ecosystem is needed (which is a must for modern AI applications), the architect will also decide what will and won't be its own service (container) and define the interfaces between these disparate services. For the AI model work, the role of a *data scientist / ML engineer* is needed. This role typically works primarily in Jupyter notebooks selecting, creating, training and tuning ML models to support application features. Production software engineering is typically outside of the scope of this expertise in practice. The role of a *backend engineer* is needed for implementing the main services of the application and taking the code out of Jupyter notebooks to build the models into the backend (server-side) of the application. The *backend engineer* is also responsible for supporting new features and creating their API interfaces for *frontend* engineers. One of the key roles any software team needs to deploy an AI product is a *DevOps engineer*. This role is solely responsible for deploying and configuring containers to run on a cloud and ensure these containers are operational and scaled to the load requirements of the software. This responsibility covers configuring software instance pods, database pods, caching layers, logging services, and parameterizing replicas and auto-scaling heuristics.

In this traditional model of software engineering, many challenges and complexity emerge. An example is the (quite typical) scenario of the first main server-side implementation of the application being a monoservice while DB, caching, and logging are microservices. As the *ML engineer* introduces models of increasing size, the *dev-ops* person alerts the team that the cloud instances, though designated as *large*, only have 8gb of ram. Meanwhile new AI models being integrated exceed this limit. This event leads to a re-architecture of the main monoservice to be split out AI models into microservices and interfaces being designed or adopted leading to significant backend work / delays. In this work, we aim to create a solution that would move all of this decisioning and work under the purview of the automated runtime system.

Ultimately, the mission of Jaseci is to accelerate and democratize the development and deployments of end-to-end scalable AI applications as presented in Figure 2.2B. To this end, we present a novel set of higher level abstractions for programming sophisticated software in a micro-service/serverless AI and a full stack architecture and programming model that abstracts away and automates much of the complexity of building diffuse applications on a distributed compute substrate of potentially thousands of compute nodes.

2.4 A Higher Level Language

Traditionally in computer science, the task of raising the level abstraction in a computational model has primarily been for the goal of increasing programmer productivity. This

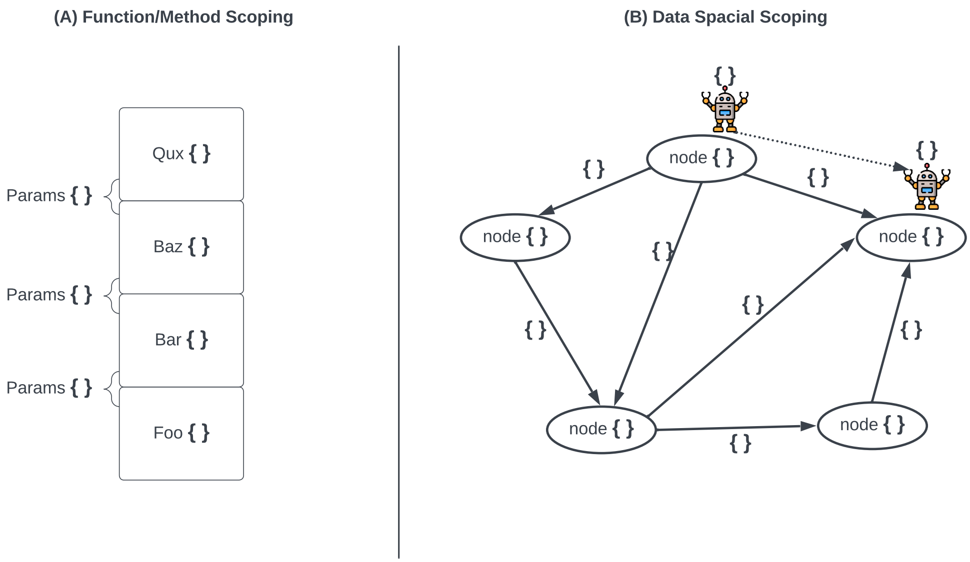


Figure 2.3: A visualization of the behavior of scopes and problem solving abstractions provided by the near ubiquitous function / method based languages (left) and the data spacial programming model (right).

productivity comes from allowing engineers to function at the problem level while hiding the complexity of the underlying system. The Jac language introduces a set of new abstractions guided by these principles based on two key insights. First, Jac recognizes the emerging need for programmers to reason about and solve problems with graph representations of data. Second, Jac further supports the need for algorithmic modularity and encapsulation to change and prototype production software in place of prior running codebases. Based on these insights, we introduce two new sets of abstractions. As shown in Figure 2.3b, Jac's **data-spacial scoping** natively facilitates graph based problem solving by replacing the traditional *temporal* notion of scope with a function's activation record with scoping that is flattened and spatially laid out in graph structure. This type of scoping allows for richer semantics for the organization of the data relevant to the problem being solved. Figure 2.3b also depicts Jac's **agent oriented programming** as little robots. Each robot carries scope with it as it walks and performs compute relevant to where it sits on the graph. These 'agent' abstractions capture the need for algorithmic modality and encapsulation when introducing solutions to already sophisticated codebases. Jac can be used solely to build out complete solutions or as glue code with components built in other languages. By leveraging these new language abstractions, HomeLendingPal [6] was able to create a production grade conversational AI experience with ~300 lines of code in contrast to the tens of thousands it would take to build in a traditional programming language.

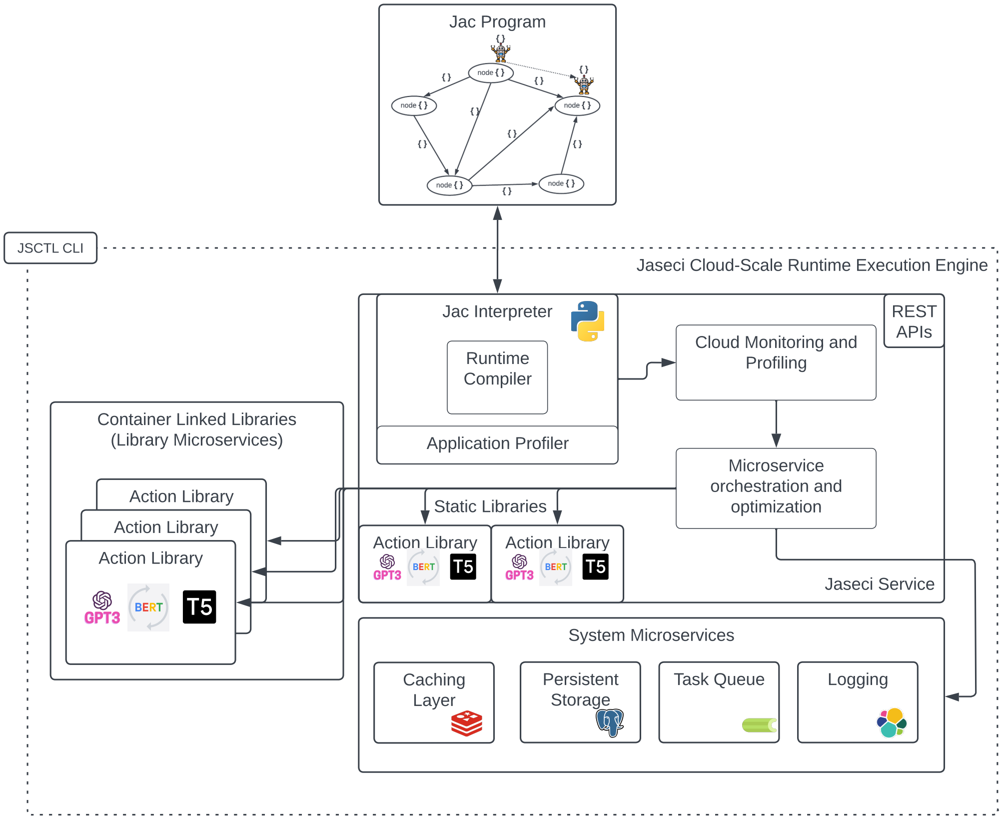


Figure 2.4: The architecture of the Jaseci diffuse runtime execution. The runtime stack includes and combines information from interpreter level profiling, cloud monitoring and profiling, microservice orchestrator and optimizer. Container linked libraries are also depicted.

2.5 A Novel Underlying Technology Stack

Jaseci's cloud-scale runtime engine presents a higher level abstraction of the software stack. The *diffuse* runtime engine subsumes responsibility not only for the optimization of program code, but also the orchestration, configuration, and optimization of constituent micro services and the full cloud compute stack. Duties such as container formation, microservice scaling, scheduling and optimization are automated by the runtime. For example, as shown in Figure 2.4c Jaseci introduces the concept of **container linked libraries** to complement traditional notions of statically and dynamically linked libraries. From the programmers perspective, they need not know whether a call to a library is fused with the running programming instance or a remote call to a microservice somewhere in a cluster. The decisioning of what *should* be a microservice and what should be statically in the programs object scope is made automatically and seamlessly by the Jaseci **microservice orchestration engine**. Underlying in-cluster microservices are encapsulated and hidden with this abstraction. With the runtime having full visibility and control over the diffuse application, high complexity runtime decisions and heuristics such as autoscaling is brought under the purview of the runtime software stack, relieving the need of manual configuration. With this Jaseci runtime, a single frontend engineer was able to implement the full ZeroShotBot [14] application (which uses a number of transformer neural networks) without writing a single line of traditional 'backend' code. This implementation currently support tens of thousands of queries a day across about ∼12 business customers with tens of thousands of individual end users in a single production environment.

2.6 Battle Testing so Far...

Jaseci is available on Github [8] under MIT open source license and is composed of an ecosystem of tools spanning 3 packages. These include **Jaseci Core**, its core execution engine, **Jaseci Serv**, its diffuse runtime cloud-scale execution engine, and **Jaseci Kit**, a collection of cutting edge AI engines provided by the Jaseci community. In addition to these main codebases, an experimental toolkit we call **Jaseci Studio** is in development to provide visual programming and debugging tooling for developers building with Jaseci.

There are a number of notable examples of Jaseci's use in production. These users include four selected start-up companies that have adopted Jac and Jaseci as their development engine and have already launched their products built using Jaseci.

myca.ai [10] - a B2C personal productivity platform that uses AI to understand personal behavior trends and help users allocate their time, prioritize their tasks and achieve personal growth goals. Using Jaseci, myca.ai's back-end development only took 1 month and myca.ai was launched within 3 months' development to the public. Myca.ai is one of the fast growing personal growth tool and has received positive feedback from their users.

ZeroShotBot [14] - a B2B company that develops a cutting edge conversational AI platform using Jaseci. The product development took 2 months and was done by frontend engineers. Zeroshotbot has gained significant market traction and has been in business discussions with major logos such as Volaris, Pizzahut to provide readily deployable FAQ chatbots.

Truselph [13] - A minority founded startup. Truselph creates an avatar of the person and builds conversational intelligence that allows the general public to interact with the avatar and ask questions, while the avatar will be able to provide personalized answers with emotions and facial expressions. Truselph is in partnership with Lenovo to co-develop Truselph powered Kiosks for retail stores and is in business discussions with chains such as Sephora.

Home Lending Pal [6] - an AI Powered Mortgage Advisor. Home Lending Pal is a minority founded start-up that helps people, especially under-served minority population to navigate through the mortgage and home purchase process. Home Lending Pal adopted Jaseci to provide two main product features: 1 - personalized mortgage advice and 2 - Kev, an AI-powered chatbot that will answer users questions about the process and give them a plan to improve their finances.

2.7 In a Nutshell

Jaseci is a novel computational model invented, designed and implemented to address this challenge. Jaseci includes a novel programming model we call *data-spacial programming* and a runtime engine we call the *diffuse execution environment* to enable rapid development of large scale and nimble AI applications. Our initial infrastructure has been used in practice to achieve 10x reduction in development time and near 100% elimination of typical backend code needed for a complicated AI based application. Jaseci [7] was open sourced in 2021 [8] [9]. Today Jaseci is in production with 4 distinct commercial products built on the engine, including Myca [10], HomeLendingPal [6], ZeroShotBot [14] and TrueSelph [13].

Chapter 3

Abstrations of Jaseci

Contents

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 insanely 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!?!?"

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!"

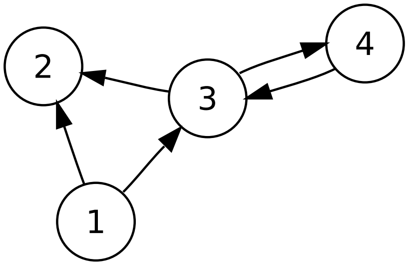
3.1.1 Yes, But What Kind of Graphs

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.

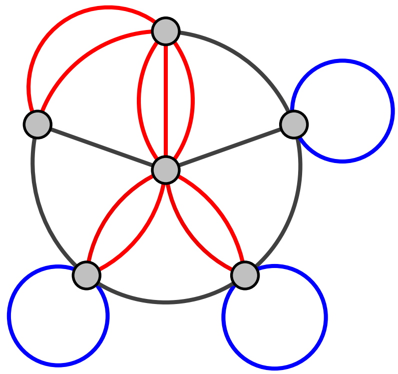
In Jaseci, we elect to assume the following semantics:

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.

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



(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]

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

One of the most important abstractions introduced in Jaseci is that of the `walker`. The semantics of this abstraction is unlike any that has existed in any programming language before.

In a nutshell, a walker is a unit of execution that retains state (its local scope) as it travels over a graphs. Walkers 'walk' from node to node in the graph and executing its body.

The walker's body is specified with an opening and closing braces (`{ }`) and is executed to completion on each node it lands on. In this sense a walker iterates while spooling through a sequence of nodes that it 'takes' using the `take` keyword. We call each of these iterations *node-bound iterations*.

Variables declared in a walker's body takes two forms: its *context variables*, those that retain state as it travels from node to node in a graph, and its *local variables*, those that are reinitialized for each node-bound iterations.

Walkers present a new way of thinking about programmatic execution distinct from the near-ubiquitous function based asbtraction in other languages. Instead of a functions scope being temporally pushed onto an ever increasing stack as functions call other functions. Scopes can be spacially laid out on a graph and walkers can hop around the graph taking its scope with it. A key difference in this model is in its introduction of data spacial problem solving. In the former function-based model scopes become unaccessible upon the sub-call of a function until that function returns. In contrast, walkers can access any scope at any time in a modular way.

When solving problems with walkers, a developer can think of that walker as a little self-contained robot or agent that can retain context as it spacially moves about a graph, interacting with the context in nodes and edges of that graph.

In addition to the introduction of the `take` command to support new types of control flow for node-bound iterations. The keywords and semantics of `disengage`, `skip`, and `ignore` are also introduced. These instruct walkers to stop walking the graph, skip over a node for execution, and ignore certain paths of the graph. These semantics are describe in more detail later in the book.

[Entrypoints to a jac program, init recognized as default]

3.3 Abilities

Nodes, edges, and walkers can have `abilities`. The body of an ability is specified with an opening and closing braces (`{ }`) within the specification of a node, edge, or walker and specify a unit of execution.

Abilities are most closely analogous to methods in a traditional object oriented program, however they do not have the same semantics of a traditional function. An ability can only interact within the scope of context and local variables of the node/edge/walker for which it is affixed and do not have a `return` semantic. (Though it is important to note, that abilities can always access the scope of the executing walker using the `visitor` special variable as described below)

When using abilities, a developer can think of these as self-contained in-memory/in-data compute operations.

3.4 `here` and `visitor`

At every execution point in a Jac/Jaseci program there are two scopes visible, that of the walker, and that of the node it is executing on. These contexts can be referenced with the special variables `here` and `visitor` respectively. Walkers use `here` to refer to the context of the node it is currently executing on, and abilities can use `visitor` to refer to the context of the current walker executing.

3.5 Actions

Actions enables bindings to functionality specified outside of Jac/Jaseci and behave as function calls with returns. These are analogous to library calls in traditional languages. This external functionality in practice takes the form of direct binding to python implementations that are packaged up as a Jaseci action library.

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

Note: This action interface is the abstraction that allows Jaseci to do it's fancy inter-machine optimizations, auto-scaling, auto-componentization etc.

Chapter 4

Architecture of Jaseci and Jac

Contents

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

Contents

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.

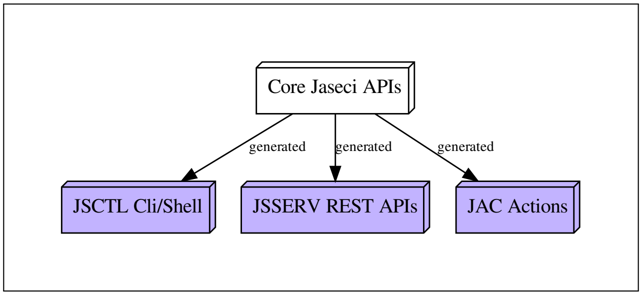


Figure 5.1: Jaseci Interface Architecture

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 not P.C. unless you make it not P.C.). The 'client' expressions of that interface in the forms of a command line tool `jsctl` and a server-side REST API built using Django [1]. Figure 5.1 illustrates this architecture representing the relationship between core APIs and client side expressions.

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

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

```
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 4 ***(time to let your eyes glaze over)***

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

Python Code 5.1: setup.py for Jaseci Core

```python
from setuptools import setup, find_packages

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.2 A Simple Workflow for Tinkering

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

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

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

Lets hop into a `jsctl` shell.

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

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

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

```
jaseci > help

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

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

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

  Group of `graph` commands

Options:
  --help Show this message and exit.

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

  Create a graph instance and return root node graph object

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

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

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

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

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

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

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

The `edit` command invokes the built in editor. Though it's a terminal editor based on `ncurses`, you can basically use it much like you'd use any wysiwyg editor with features like standard cut `ctrl-c` and paste `ctrl-v`, mouse text selection, etc. It's based on the phenomenal pure python project from Google called `ci_edit`. For more detailed help cheat sheet see Appendix. 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

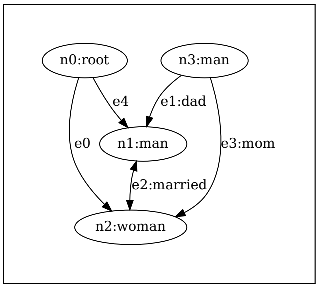


Figure 5.2: Graph for `fam.jac`

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`.

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
info	n/a
walker summon	key: str (*req), wlk: Walker (*req), nd: Node (*req), ctx: dict (\{\}), _req_ctx: dict (\{\}), global_sync: bool (True)
walker callback	nd: Node (*req), wlk: Walker (*req), key: str (*req), ctx: dict (\{\}), _req_ctx: dict (\{\}), global_sync: bool (True)
walker register	snt: Sentinel (None), code: str (), dir: str (/), encoded: bool (False)
walker get	wlk: Walker (*req), mode: str (default), detailed: bool (False)
walker set	wlk: Walker (*req), code: str (*req), mode: str (default)
walker list	snt: Sentinel (None), detailed: bool (False)
walker spawn create	name: str (*req), snt: Sentinel (None)
walker spawn list	detailed: bool (False)
walker spawn delete	name: str (*req)
walker spawn clear	n/a
walker yield list	detailed: bool (False)
walker yield delete	name: str (*req)
walker yield clear	n/a
walker prime	wlk: Walker (*req), nd: Node (None), ctx: dict (\{\}), _req_ctx: dict (\{\})

walker execute	wlk: Walker (*req), prime: Node (None), ctx: dict ↪ (\{\}), _req_ctx: dict (\{\}), profiling: bool (↪ False)
walker run	name: str (*req), nd: Node (None), ctx: dict (\{\}) ↪ , _req_ctx: dict (\{\}), snt: Sentinel (None), ↪ profiling: bool (False), is_async: bool (False)
walker queue check	task_id: str ()
walker queue wait	task_id: str (*req)
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 (None)
global sentinel unset	n/a
object get	obj: Element (*req), depth: int (0), detailed: bool (↪ False)
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 (False)
object perms revoke	obj: Element (*req), mast: Element (*req)
graph create	set_active: bool (True)
graph get	gph: Graph (None), mode: str (default), detailed: bool ↪ (False)
graph list	detailed: bool (False)
graph active set	gph: Graph (*req)
graph active unset	n/a
graph active get	detailed: bool (False)
graph delete	gph: Graph (*req)
graph node get	nd: Node (*req), keys: list ([])
graph node view	nd: Node (None), detailed: bool (False), show_edges: ↪ bool (True), node_type: str (), edge_type: str ()
graph node set	nd: Node (*req), ctx: dict (*req), snt: Sentinel (None ↪)
graph walk (cli only)	nd: Node (None)

sentinel register	name: str (default), code: str (), code_dir: str (./), mode: str (default), encoded: bool (False), auto_run: str (init), auto_run_ctx: dict (\{\}), auto_create_graph: bool (True), set_active: bool (True)
sentinel pull	set_active: bool (True), on_demand: bool (True)
sentinel get	snt: Sentinel (None), mode: str (default), detailed: bool (False)
sentinel set	code: str (*req), code_dir: str (./), encoded: bool (False), snt: Sentinel (None), mode: str (default)
sentinel list	detailed: bool (False)
sentinel test	snt: Sentinel (None), detailed: bool (False)
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 (False), detailed: bool (False)
sentinel active get	detailed: bool (False)
sentinel delete	snt: Sentinel (*req)
wapi	name: str (*req), nd: Node (None), ctx: dict (\{\}), _req_ctx: dict (\{\}), snt: Sentinel (None), profiling: bool (False)
architype register	code: str (*req), encoded: bool (False), snt: Sentinel (None)
architype get	arch: Architype (*req), mode: str (default), detailed: bool (False)
architype set	arch: Architype (*req), code: str (*req), mode: str (default)
architype list	snt: Sentinel (None), detailed: bool (False)
architype delete	arch: Architype (*req), snt: Sentinel (None)
master create	name: str (*req), global_init: str (), global_init_ctx: dict (\{\}), other_fields: dict (\{\})
master get	name: str (*req), mode: str (default), detailed: bool (False)
master list	detailed: bool (False)
master active set	name: str (*req)
master active unset	n/a
master active get	detailed: bool (False)
master self	detailed: bool (False)
master delete	name: str (*req)

master createsuper	name: str (*req), global_init: str (), global_init_ctx ↪ : dict (\{\}), other_fields: dict (\{\})
master allusers	limit: int (0), offset: int (0), asc: bool (False)
master become	mast: Master (*req)
master unbecome	n/a
config get	name: str (*req), do_check: bool (True)
config set	name: str (*req), value: str (*req), do_check: bool (↪ True)
config refresh	name: str (*req)
config list	n/a
config index	n/a
config exists	name: str (*req)
config delete	name: str (*req), do_check: bool (True)
logger http connect	host: str (*req), port: int (*req), url: str (*req), ↪ log: str (all)
logger http clear	log: str (all)
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 (False)
jac run (cli only)	file: str (*req), walk: str (init), ctx: dict (\{\}), ↪ profiling: bool (False)
jac dot (cli only)	file: str (*req), walk: str (init), ctx: dict (\{\}), ↪ detailed: bool (False)

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_ai_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 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.

5.3.2.1 `architype register`

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

`args: code: str (*req), encoded: bool (False), snt: Sentinel (None)`

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

Returns

Fields include 'architype': Architype object if created otherwise null 'success': True/False whether register was successful 'errors': List of errors if register failed 'response': Message on outcome of register call

5.3.2.2 `architype get`

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

```
args:    arch: Architype (*req), mode: str (default), detailed: bool (
↪ False)
```

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

Returns

Fields include (depends on mode) 'code': Formal source code for architype 'ir': Intermediate representation of architype 'architype': Architype object print

5.3.2.3 `architype set`

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

`args: arch: Architype (*req), code: str (*req), mode: str (default)`

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,

Returns

Fields include (depends on mode) 'success': True/False whether set was successful 'errors': List of errors if set failed 'response': Message on outcome of set call

5.3.2.4 `architype list`

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

args: `snt`: `Sentinel` (`None`), `detailed`: `bool` (`False`)

No documentation yet.

Parameters

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

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

Returns

List of architype objects

5.3.2.5 `architype delete`

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

args: `arch`: `Architype` (`*req`), `snt`: `Sentinel` (`None`)

No documentation yet.

Parameters

`arch` – The architype being set

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

Returns

Fields include (depends on mode) 'success': True/False whether command was successful 'response': Message on outcome of command

5.3.3 APIs for config

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

5.3.3.1 `config get`

cli: config get | api: config_get | auth: admin
args: name: str (*req), do_check: bool (True)
No documentation yet.

5.3.3.2 `config set`

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

5.3.3.3 `config refresh`

cli: config refresh | api: config_refresh | auth: admin
args: name: str (*req)
No documentation yet.

5.3.3.4 `config list`

`cli: config list | api: config_list | auth: admin`

args: `n/a`

No documentation yet.

5.3.3.5 `config index`

`cli: config index | api: config_index | auth: admin`

args: `n/a`

No documentation yet.

5.3.3.6 `config exists`

`cli: config exists | api: config_exists | auth: admin`

args: `name: str (*req)`

No documentation yet.

5.3.3.7 config delete

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

5.3.4 APIs for global

No documentation yet.

5.3.4.1 global set

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

5.3.4.2 global delete

cli: global delete | api: global_delete | auth: admin
args: name: str (*req)
No documentation yet.

5.3.4.3 `global sentinel set`

`cli: global sentinel set | api: global_sentinel_set | auth: admin`
`args: snt: Sentinel (None)`
No documentation yet.

5.3.4.4 `global sentinel unset`

`cli: global sentinel unset | api: global_sentinel_unset | auth: admin`
`args: n/a`
No documentation yet.

5.3.5 APIs for graph

No documentation yet.

5.3.5.1 `graph create`

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

5.3.5.2 graph get

cli: graph get | api: graph_get | auth: user

args: gph: Graph (None), mode: str (default), detailed: bool (False)

Valid modes: default, dot,

5.3.5.3 graph list

cli: graph list | api: graph_list | auth: user

args: detailed: bool (False)

No documentation yet.

5.3.5.4 graph active set

cli: graph active set | api: graph_active_set | auth: user

args: gph: Graph (*req)

No documentation yet.

5.3.5.5 `graph active unset`

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

5.3.5.6 `graph active get`

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

5.3.5.7 `graph delete`

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

5.3.5.8 graph node get

cli: graph node get | api: graph_node_get | auth: user
args: nd: Node (*req), keys: list ([])
No documentation yet.

5.3.5.9 graph node view

cli: graph node view | api: graph_node_view | auth: user
args: nd: Node (None), detailed: bool (False), show_edges: bool (True) ↪ , node_type: str (), edge_type: str ()
No documentation yet.

5.3.5.10 graph node set

cli: graph node set | api: graph_node_set | auth: user
args: nd: Node (*req), ctx: dict (*req), snt: Sentinel (None)
No documentation yet.

5.3.5.11 graph walk

```
cli: graph walk (cli only)
```

args: nd: Node (None)

No documentation yet.

5.3.6 APIs for jac

No documentation yet.

5.3.6.1 jac build

```
cli: jac build (cli only)
```

args: file: str (*req), out: str ()

No documentation yet.

5.3.6.2 jac test

```
cli: jac test (cli only)
```

args: file: str (*req), detailed: bool (False)

and .jir executables

5.3.6.3 jac run

cli: jac run (cli only)

```
args: file: str (*req), walk: str (init), ctx: dict ({}), profiling:
↪ bool (False)
```

and .jir executables

5.3.6.4 jac dot

cli: jac dot (cli only)

```
args:  file: str (*req), walk: str (init), ctx: dict ({}), detailed:
↪ bool (False)
```

files and .jir executables

5.3.7 APIs for logger

No documentation yet.

5.3.7.1 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 (
↪ all)
```

Valid log params: sys, app, all

5.3.7.2 logger http clear

```
cli: logger http clear | api: logger_http_clear | auth: admin
```

args: log: str (all)

Valid log params: sys, app, all

5.3.7.3 logger list

```
cli: logger list | api: logger_list | auth: admin
```

args: n/a

No documentation yet.

5.3.8 APIs for master

These APIs

5.3.8.1 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)

5.3.8.2 master get

cli: master get | api: master_get | auth: user

args: name: str (*req), mode: str (default), detailed: bool (False)

Valid modes: default,

5.3.8.3 master list

cli: master list | api: master_list | auth: user

args: detailed: bool (False)

No documentation yet.

5.3.8.4 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

5.3.8.5 master active unset

cli: master active unset | api: master_active_unset | auth: user

args: n/a

No documentation yet.

5.3.8.6 master active get

cli: master active get | api: master_active_get | auth: user

args: detailed: bool (False)

No documentation yet.

5.3.8.7 master self

cli: master self | api: master_self | auth: user

args: detailed: bool (False)

No documentation yet.

5.3.8.8 master delete

cli: master delete | api: master_delete | auth: user

args: name: str (*req)

No documentation yet.

5.3.9 APIs for object

...

5.3.9.1 global get

cli: global get | api: global_get | auth: user

args: name: str (*req)

No documentation yet.

5.3.9.2 object get

cli: object get | api: object_get | auth: user

args: obj: Element (*req), depth: int (0), detailed: bool (False)

No documentation yet.

5.3.9.3 object perms get

cli: object perms get | api: object_perms_get | auth: user

args: obj: Element (*req)

No documentation yet.

5.3.9.4 object perms set

cli: object perms set | api: object_perms_set | auth: user

args: obj: Element (*req), mode: str (*req)

No documentation yet.

5.3.9.5 object perms default

cli: object perms default | api: object_perms_default | auth: user

args: mode: str (*req)

No documentation yet.

5.3.9.6 object perms grant

cli: object perms grant | api: object_perms_grant | auth: user

args: obj: Element (*req), mast: Element (*req), read_only: bool (
↪ False)

No documentation yet.

5.3.9.7 object perms revoke

cli: object perms revoke | api: object_perms_revoke | auth: user

args: obj: Element (*req), mast: Element (*req)

No documentation yet.

5.3.9.8 info

cli: info | api: info | auth: public

args: n/a

No documentation yet.

5.3.10 APIs for queue

APIs used for celery configuration and monitoring

5.3.10.1 walker queue check

cli: walker queue check | api: walker_queue_check | auth: user

args: task_id: str ()

No documentation yet.

5.3.10.2 walker queue wait

cli: walker queue wait | api: walker_queue_wait | auth: user

args: task_id: str (*req)

No documentation yet.

5.3.11 APIs for sentinel

A sentinel is a unit in Jaseci that represents the organization and management of a collection of architypes and walkers. In a sense, you can think of a sentinel as a complete Jac implementation of a program or API application. Though its the case that many sentinels can be interchangeably across any set of graphs, most use cases will typically be a single sentinel shared by all users and managed by an admin(s), or each users maintaining a single sentinel customized for their individual needs. Many novel usage models are possible, but I'd point the beginner to the model most analogous to typical server side software development to start with. This model would be to have a single admin account responsible for updating a single sentinel that all users would share for their individual graphs. This model is achieved through using `sentinel_register`, `sentinel_active_global`, and `global_sentinel_set`.

5.3.11.1 `sentinel register`

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

```
args:      name: str (default), code: str (), code_dir: str (./), mode
↪ : str (default), encoded: bool (False), auto_run: str (init)
↪ , auto_run_ctx: dict ({}), auto_create_graph: bool (True),
↪ set_active: bool (True)
```

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 (True), on_demand: bool (True)
```

No documentation yet.

5.3.11.3 `sentinel get`

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

```
args: snt: Sentinel (None), mode: str (default), detailed: bool (False
↪ )
```

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 (False), snt
↪ : Sentinel (None), mode: str (default)

Valid modes: code, ir,

5.3.11.5 sentinel list

cli: sentinel list | api: sentinel_list | auth: user

args: detailed: bool (False)

No documentation yet.

5.3.11.6 sentinel test

cli: sentinel test | api: sentinel_test | auth: user

args: snt: Sentinel (None), detailed: bool (False)

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_run_ctx: dict ({}), auto_create_graph: bool (False), detailed: bool (False)

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 (False)

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 super

No documentation yet.

5.3.12.1 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)

5.3.12.2 `master allusers`

**`cli: master allusers	api: master_allusers	auth: admin`**
`args: limit: int (0), offset: int (0), asc: bool (False)`
return and offset specfies where to start NOTE: Abstract interface to be overridden

5.3.12.3 `master become`

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

5.3.12.4 `master unbecome`

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

5.3.13 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.

5.3.13.1 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.

5.3.14 APIs for walker

The walker set of APIs are used for execution and management of walkers. Walkers are the primary entry points for running Jac programs. The primary API used to run walkers is **walker_run**. There are a number of variations on this API that enable the invocation of walkers with various semantics.

5.3.14.1 walker register

cli: walker register | api: walker_register | auth: user

args: snt: Sentinel (None), code: str (), dir: str (/), encoded: bool
↪ (False)

Though the common case is to register entire sentinels, a user can also register individual walkers one at a time. This API accepts code for a single walker (i.e., walker...}}).

5.3.14.2 walker get

cli: walker get | api: walker_get | auth: user

args: wlk: Walker (*req), mode: str (default), detailed: bool (False)

Valid modes: default, code, ir, keys,

5.3.14.3 walker set

cli: walker set | api: walker_set | auth: user

args: wlk: Walker (*req), code: str (*req), mode: str (default)

Valid modes: code, ir,

5.3.14.4 walker list

cli: walker list | api: walker_list | auth: user

args: snt: Sentinel (None), detailed: bool (False)

No documentation yet.

5.3.14.5 walker spawn create

cli: walker spawn create | api: walker_spawn_create | auth: user

args: name: str (*req), snt: Sentinel (None)

No documentation yet.

5.3.14.6 walker spawn list

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

args: detailed: bool (False)

No documentation yet.

5.3.14.7 walker spawn delete

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

args: name: str (*req)

No documentation yet.

5.3.14.8 walker spawn clear

```
cli: walker spawn clear | api: walker_spawn_clear | auth: user
```

args: n/a

No documentation yet.

5.3.14.9 walker yield list

cli: walker yield list | api: walker_yield_list | auth: user
args: detailed: bool (False)
No documentation yet.

5.3.14.10 walker yield delete

cli: walker yield delete | api: walker_yield_delete | auth: user
args: name: str (*req)
No documentation yet.

5.3.14.11 walker yield clear

cli: walker yield clear | api: walker_yield_clear | auth: user
args: n/a
No documentation yet.

5.3.14.12 walker prime

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

```
args:  wlk: Walker (*req), nd: Node (None), ctx: dict ({}), _req_ctx:
↪ dict ({})
```

No documentation yet.

5.3.14.13 walker execute

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

```
args: wlk: Walker (*req), prime: Node (None), ctx: dict ({}), _req_ctx
↪ : dict ({}), profiling: bool (False)
```

No documentation yet.

5.3.14.14 walker run

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

```
args:    name: str (*req), nd: Node (None), ctx: dict ({}), _req_ctx:
↪ dict ({}), snt: Sentinel (None), profiling: bool (False), is_async
↪ : bool (False)
```

reports results, and cleans up walker instance.

5.3.14.15 wapi

```
cli: wapi | api: wapi | auth: user

args:   name: str (*req), nd: Node (None), ctx: dict ({}), _req_ctx:
↪ dict ({}), snt: Sentinel (None), profiling: bool (False)
```

No documentation yet.

5.3.14.16 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 (True)
```

along with the walker id and node id

5.3.14.17 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 (True)
```

along with the walker id and node id

Part II

The Jac Programming Language

Chapter 6

Jac Language Overview and Basics

Contents

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 [11] 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.

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

- 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.

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 {
```

[1]Image credit to wiki contributer [1]

```
    std.out("Hello World");
}
```

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

```
Hello World
```

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

6.2 Numbers, Arithmetic, and Logic

6.2.1 Basic Arithmetic Operations

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

Jac Code 6.2: Basic arithmetic operations

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

The output of this groundbreaking program is,

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

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

Additionally, Jac supports power and modulo operations.

Jac Code 6.3: Additional arithmetic operations

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

Jac Code 6.3 outputs,

```
256 4
```

Here, we can also observe that, unlike Python, whitespace does not mater whatsoever. Languages utilizing whitespace to express static scoping should be criminalized. Yeah, I said it, see Rant A.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 5 ***(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 B)

6.2.2 Comparison, Logical, and Membership Operations

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

Jac Code 6.5: Comparision operations

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

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

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

Jac Code 6.6: Logical operations

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

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

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

[NEED EXAMPLE FOR MEMBERSHIP OPERATIONS]

Nerd Alert 6 ***(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 B)

6.2.3 Assignment Operations

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

Jac Code 6.8: Assignment operations

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

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

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

As shown in Jac Code 6.8, there are a number of ways we can articulate an assignment. Of

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

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 7 ***(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 B)

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!

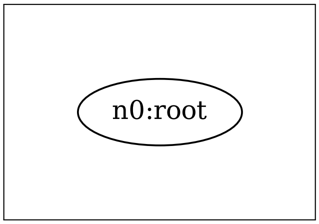


Figure 6.2: Graph in memory for simple Hello World program (JC 6.1)

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.

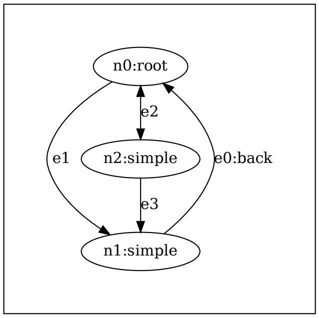


Figure 6.3: Graph in memory for JC 6.14

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::join`	(params)	Join elements of the sequence (params) separated by the string separator that calls the join function.
`.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;
```

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	(key, default)	Returns value of key if exists otherwise default
.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

```
    if(a < b): std.out("Hello!");
}
```

```
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

Contents

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 8 ***(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 B)

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;
```

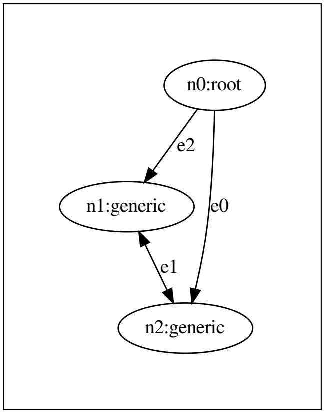


Figure 7.1: Graph in memory for JC 7.2

```
}
```

Jac Code 7.3: Creating named node types

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

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

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

Jac Code 7.4: Connecting nodes within spawn statement

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

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

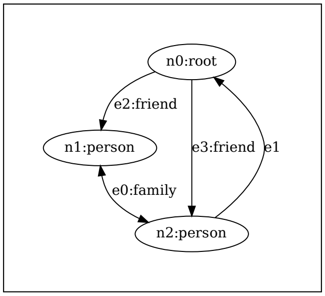


Figure 7.2: Graph in memory for JC 7.3

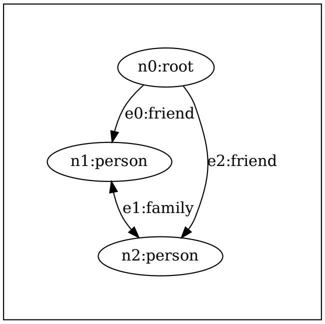


Figure 7.3: Graph in memory for JC 7.4

```
    here -[friend]-> node2;
}
```

Jac Code 7.5: Chaining node connections using the connect operator

```
1 node person;
2 edge friend;
3 edge family;
4
5 walker init {
6     node1 = spawn node::person;
7     node2 = spawn node::person;
8     node2 <-[friend]- here -[friend]-> node1 <-[family]-> node2;
9 }
```

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

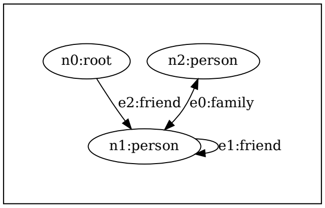


Figure 7.4: Graph in memory for JC 7.5

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
```

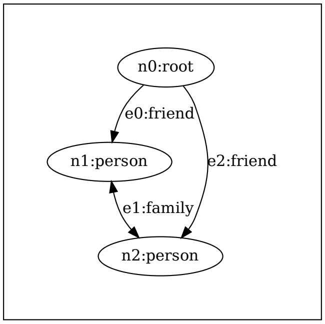


Figure 7.5: Graph in memory for JC 7.10

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

```
node person;
```

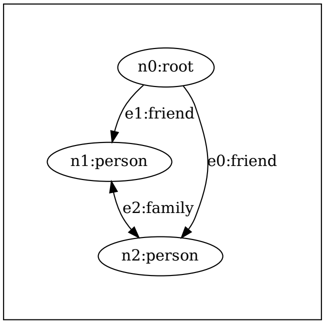


Figure 7.6: Graph in memory for JC 7.11

```
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;
    has birthday, profession;
}
```

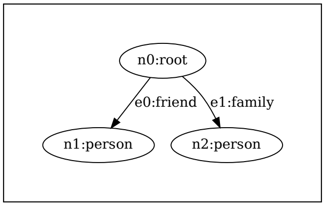


Figure 7.7: Graph in memory for JC 7.13

```

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];

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

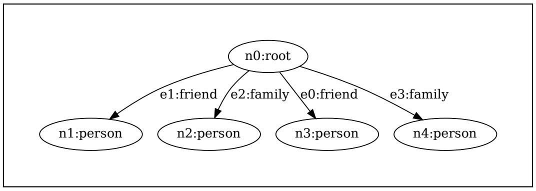


Figure 7.8: Graph in memory for JC 7.15

```
{'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);
    std.out(-->.edge.e_name);
}
```

```
["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);
}
```

```
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.

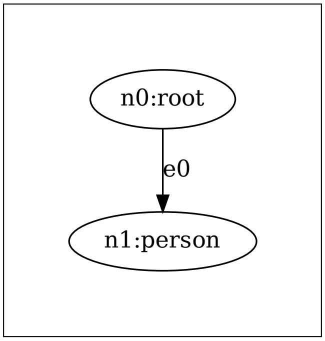


Figure 7.9: Graph in memory for JC 7.18 and 7.19

Nerd Alert 9 ***(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: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'}
```

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

Contents

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);
    }
}
```

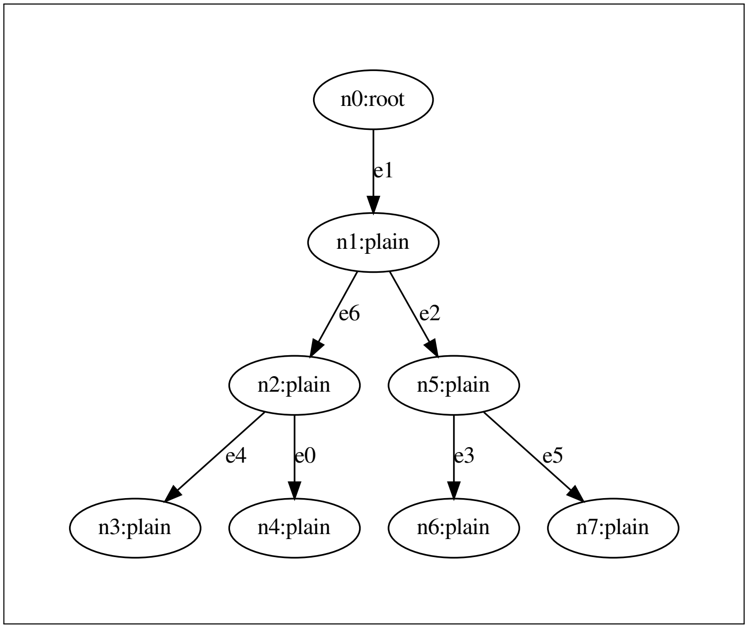


Figure 8.1: Graph in memory for JC 8.3

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

```
14         head=n[0];
15     }
16 }
17
18 walker walk_with_breadth {
19     has anchor node_order = [];
20     node_order.l::append(here.name);
21     take:bfs -->; #take:b can also be used
22 }
23
24 walker walk_with_depth {
25     has anchor node_order = [];
26     node_order.l::append(here.name);
27     take:dfs -->; #take:d can also be used
28 }
29
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 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

```
global node_count=0;
node simple: has id;

walker init {
    has output = [];
    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;
        }
    }
    take -->;
    simple {
        if(here.id % 2==0): skip;
        output.l::append(here.id);
    }
    output.l::append(here.info['name']);
    with exit: std.out(output);
}
```

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

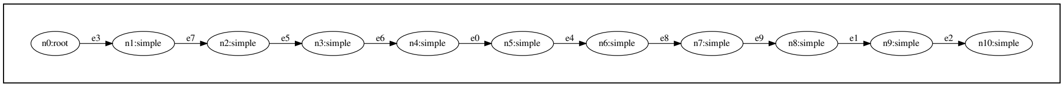


Figure 8.2: Graph in memory for JC 8.4 and JC 8.5

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

```
global node_count=0;
node simple: has id;

walker init {
    has output = [];
    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;
        }
    }
    take -->;
    simple {
        if(here.id % 2==0): skip;
        if(here.id == 7): disengage;
        output.l::append(here.id);
    }
    output.l::append(here.info['name']);
    with exit: std.out(output);
}
```

```
["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;
        }
```

```
12     }
13     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

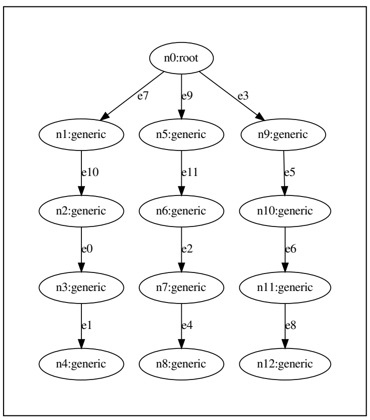


Figure 8.3: Graph in memory for JC 8.10

with 4 connected nodes each. Yep, this yeilding semantic is very handy indeed!

Chapter 9

Actions and Action Sets

Contents

9.1 Standard Action Library

9.1.1 date

No documentation yet.

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.datetime_now

args: n/a

No documentation yet.

date.date_now

args: n/a

No documentation yet.

date.timestamp_now

args: n/a

No documentation yet.

date.date_day_diff

args: start_date: str (*req), end_date: str (None)

No documentation yet.

9.1.2 file

No documentation yet.

file.load_str

args: fn: str (*req), max_chars: int (None)

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 (None)

No documentation yet.

file.delete

args: fn: str (*req)

No documentation yet.

9.1.3 mail

No documentation yet.

mail.send

args: sender: _empty (*req), recipients: _empty (*req), subject:
↪ _empty (*req), text: _empty (*req), html: _empty (*req)

No documentation yet.

9.1.4 net

This library of actions cover the standard operations that can be run on graph elements (nodes and edges). A number of these actions accept lists that are exclusively composed of instances of defined architype node and/or edges. Keep in mind that a `jac_set` is simply a list that only contains such elements.

net.max

args: `item_set: JacSet (*req)`

This action will return the maximum element in a list of nodes and/or edges based on an anchor has variable. Since each node or edge can only specify a single anchor this action enables a handy short hand for utilizing the anchor variable as the representative field for performing the comparison in ranking. This action does not support arhcitypes lacking an anchor.
For example, if you have a node called `movie review` with a field `has anchor score` ↪ `= .5;` that changes based on sentiment analysis, using this action will return the node with the highest score from the input list of nodes.

Parameters

`item_set` – A list of node and or edges to identify the maximum element based on their respective anchor values

Returns

A node or edge object

net.min

args: item_set: JacSet (*req)

This action will return the minimum element in a list of nodes and/or edges. This action exclusively utilizes the anchor variable of the node/edge arhcitype as the representative field for performing the comparison in ranking. This action does not support arhcitypes lacking an anchor. (see action max for an example)

Parameters

item_set – A list of node and or edges to identify the minimum element based on their respective anchor values

Returns

A node or edge object

net.pack

args: item_set: JacSet (*req), destroy: bool (False)

This action takes a subgraph as a collection of nodes in a list and creates a generic dictionary representation of the subgraph inclusive of all edges between nodes inside the collection. Note that any edges that are connecting nodes outside of the list of nodes are omitted from the packed subgraph representation. The complete context of all nodes and connecting edges are retained in the packed dictionary format. The unpack action can then be used to instantiate the identical subgraph back into a graph. Packed graphs are highly portable and can be used for many use cases such as exporting graphs and subgraphs to be imported using the unpack action.

Parameters

item_set – A list of nodes comprising the subgraph to be packed. Edges can be included in this list but is ultimately ignored. All edges from the actual nodes in the context of the source graph will be automatically included in the packed dictionary if it contects two nodes within this input list.

destroy – A flag indicating whether the original graph nodes covered by pack operation should be destroyed.

Returns

A generic and portable dictionary representation of the subgraph

`net.unpack`

args: `graph_dict: dict (*req)`

This action takes a dictionary in the format produced by the packed action to instantiate a set of nodes and edges corresponding to the subgraph represented by the pack action. The original contexts that were pack will also be created. Important Note: When using this unpack action, the unpacked collections of elements returned must be connected to a source graph to avoid memory leaks.

Parameters

`graph_dict` – A dictionary in the format produced by the pack action.

Returns

A list of the nodes and edges that were created corresponding to the input packed format. Note: Must be then connected to a source graph to avoid memory leak.

`net.root`

args: `n/a`

This action returns the root node for the graph of a given user (master). A call to this action is only valid if the user has an active graph set, otherwise it return null. This is a handy way for any walker to get to the root node of a graph from anywhere.

Returns

The root node of the active graph for a user. If none set, returns null.

9.1.5 rand

No documentation yet.

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 (4), max_length: int (10), sep: str ( )

No documentation yet.

rand.paragraph

args: min_lenth: int (4), max_length: int (8), sep: str ( )

No documentation yet.

`rand.text`

args: `min_lenth: int (3), max_length: int (6), sep: str (\n\n)`

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.

9.1.6 request

No documentation yet.

`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 (utf-8)

No documentation yet.

9.1.7 std

No documentation yet.

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 (False)`

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)`

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

std.get_global

`args: name: str (*req)`

Param 1 - name

std.actload_local

args: filename: str (*req)

No documentation yet.

std.actload_remote

args: url: str (*req)

No documentation yet.

std.actload_module

args: module: str (*req)

No documentation yet.

std.destroy_global

args: name: str (*req)

No documentation yet.

std.set_perms

```
args: obj: Element (*req), mode: str (*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
↪ )
```

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)
```

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

std.get_report

args: n/a

No documentation yet.

9.1.8 vector

No documentation yet.

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 (False), key_pos: _empty (
↪ None)
```

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

Contents

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 textttjac 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

Part V

Guided Tours and Epilogue

Chapter 13

Installation and Coding Environment

Contents

If you're the kind of haxor that doesn't want to read a huge book and just wants to get hacking ASAP, this part of the book 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!

Nerd Alert 10 ***(time to let your eyes glaze over)***

The test environment I use to test these types of things is a vanilla Ubuntu environment I spool up in Kubernetes cluster. I'll throw it here below if helpful for anyone. You can also just use the `ubuntu` docker container to validate these steps as well.
In my case, I log into the box using `kubectl exec -it <podname> -- bash`, then after updating/upgrading packages I immediately run, `apt install sudo`, `adduser haxor`, `usermod -aG sudo haxor`, `su haxor`. At this point I'm "logged in as haxor" and I can pretend that I'm you :-).

YAML 13.1: K8s Manifest for a minimal vanilla Ubuntu test environment.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: vanillabox
spec:
  selector:
    pod: vanillabox
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vanillabox
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: vanillabox
  template:
    metadata:
      labels:
        pod: vanillabox
      name: vanillabox
    spec:
      containers:
        - name: vanillabox
          image: ubuntu
          command: ["/bin/sleep", "3650d"]
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
```

13.1 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.

13.1.1 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.

13.1.2 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 [12] 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).

Now, if you care about launching to production, and you want to build some amazing AI products and experiences, you also want to install `jaseci-serv` and `jaseci-ai-kit`. Lets do exactly what we did with `jaseci` and run,

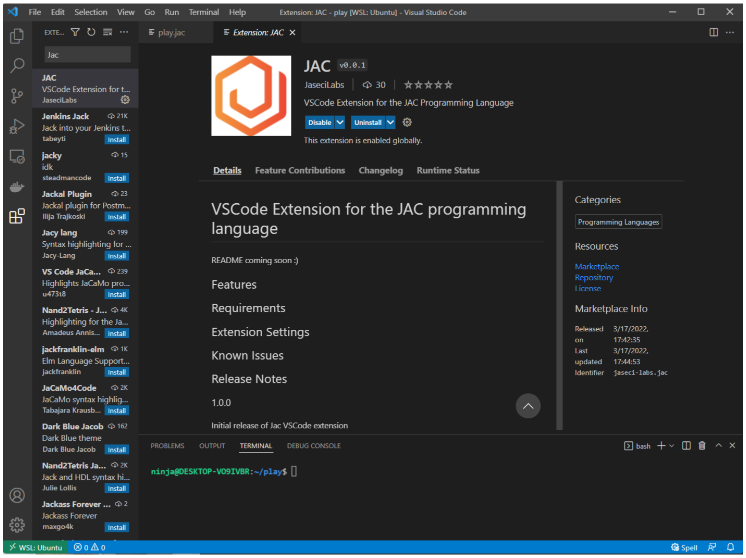


Figure 13.1: The Wonderful Jac Language extension in VSCode.

```
haxor@linux:~$ sudo apt install git cmake # Just to make sure you have it
...
haxor@linux:~$ pip install jaseci-serv
...
haxor@linux:~$ pip install jaseci-ai-kit
...
```

Now, `jaseci-ai-kit` is going to take a bit of time to run, so be patient. It's all worth it since you'll be pulling down some beefy AI technology stuff (`tensorflow` and `pytorch` stacks to be specific.)

13.1.3 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 alongside my Jaseci Omakase.

In VSCode, you can search for and install the Jac language extensions as per Figure 13.1. 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.

Nerd Alert 11 ***(time to let your eyes glaze over)***

...Personally, find an Ubuntu flavored WSL VSCode environment to be the way to go these days. In a past life I was a 100% Mac person for it's Unix based foundation. But WSL got soooooo good, and I had to switch! (plus there is insufficient gaming goodness in Mac-land). Anyway, I digress...

Chapter 14

Building CanoniCai

Contents

ConanoCai is a canonical example of a conversational AI built with Jaseci and Jac end to end. It was coded by Yiping Kang, and this section has major content contributions from Yiping Kang and Shawn Jemmont.

14.1 Build a Conversational AI System with Jaseci

In this tutorial, you are going to learn how to build a state-of-the-art conversational AI system with Jaseci and the Jac language. You will learn the basics of Jaseci, training state-of-the-art AI models, and everything in between, in order to create an end-to-end fully-functional conversational AI system.

Excited? Hell yeah! Let's jump in.

14.1.1 Preparation

To install jaseci, run this in your development environment:

```
pip install jaseci
```

To test the installation is successful, run:

```
jsctl --help
```

`jsctl` stands for the Jaseci Command Line Interface. If the command above displays the help menu for `jsctl`, then you have successfully installed jaseci.

> **Note**
>
> Take a look and get familiarized with these commands while you are at it. `jsctl` will be frequently used throughout this journey.

14.1.2 Background

A few essential concepts to get familiar with.

14.1.2.1 Graph, nodes, edges

Refer to relevant sections of the Jaseci Bible.

14.1.2.2 Walker

Refer to relevant sections of the Jaseci Bible.

14.2 Automated FAQ answering chatbot

Our conversational AI system will consist of multiple components. To start, we are going to build a chatbot that can answer FAQ questions without any custom training, using zeroshot NLP models. At the end of this section, you will have a chatbot that, when given a question, searches in its knowledge base for the most relevant answer and returns that answer.

The use case here is a Tesla FAQ chatbot. We will be using the list of FAQs from https://www.tesla.com/en_SG/support/faq.

> **Note**
>
> This architecture works for any FAQ topics and use cases. Feel free to pick another product/website/company's FAQ if you'd like!

14.2.1 Define the Nodes

We have 3 different types of nodes:

- `root`: This is the root node of the graph. It is a built-in node type and each graph has one root node only.
- `faq_root`: This is the entry point of the FAQ handler. We will make the decision on the most relevant answer at this node.
- `faq_state`: This node represents a FAQ entry. It contains a candidate answer from the knowledge base.

Now let's define the custom node types.

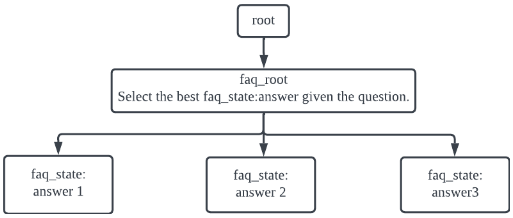


Figure 14.1: Architecture of FAQ Bot

```
node faq_root;
node faq_state {
    has question;
    has answer;
}
```

The `has` keyword defines a node's variables. In this case, each `faq_state` has a `question` and `answer`.

> **Warning**
>
> The `root` node does not need explicit definition. It is a built-in node type. Avoid using `root` as a custom node type.

14.2.2 Build the Graph

For this FAQ chatbot, we will build a graph as illustrated here:

The idea here is that we will decide which FAQ entry is the most relevant to the incoming question at the `faq_root` node and then we will traverse to that node to fetch the corresponding answer.

To define this graph architecture:

```
// Static graph definition
graph faq {
    has anchor faq_root;
    spawn {
        // Spawning the nodes
        faq_root = spawn node::faq_root;
        faq_answer_1 = spawn node::faq_state(
            question="How do I configure my order?",
            answer="To configure your order, log into your Tesla account."
```

```
        );
        faq_answer_2 = spawn node::faq_state(
            question="How do I order a tesla",
            answer="Visit our design studio to place your order."
        );
        faq_answer_3 = spawn node::faq_state(
            question="Can I request a test drive",
            answer="Yes. You must be a minimum of 25 years of age."
        );

        // Connecting the nodes together
        faq_root --> faq_answer_1;
        faq_root --> faq_answer_2;
        faq_root --> faq_answer_3;
    }
}
```

Let's break down this piece of code.

We observe two uses of the `spawn` keyword. To spawn a node of a specific type, use the `spawn` keyword for:

```
faq_answer_1 = spawn node::faq_state(
    question="How do I configure my order?",
    answer="To configure your order, log into your Tesla account.",
);
```

In the above example, we just spawned a `faq_state` node called `faq_answer_1` and initialized its `question` and `answer` variables.

> **Note**
>
> The `spawn` keyword can be used in this style to spawn many different jaseci objects, such as nodes, graphs and walkers.

The second usage of `spawn` is with the graph:

```
graph faq {
    has anchor faq_root;
    spawn {
        ...
    }
}
```

In this context, the `spawn` designates a code block with programmatic functionality to spawn a subgraph for which the root node of that spawned graph will be the `has anchor faq_root`.

In this block:

- We spawn 4 nodes, one of the type `faq_root` and three of the type `faq_state`.
- We connect each of the faq answer states to the faq root with `faq_root --> faq_answer_*`.
- We set the `faq_root` as the anchor node of the graph. As we will later see, spawning a graph will return its anchor node.

> **Warning**
>
> An anchor node is required for every graph block. It must be assigned inside the spawn block of the graph definition.

14.2.3 Initialize the Graph

Similar to nodes, in order to create the graph, we will use the `spawn` keyword.

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

This is the first walker we have introduced, so let's break it down.

- The walker is called `init`.
- It contains logic specifically for the `root` node, meaning that the code inside the `root {}` block will run **only** on the `root` node. This syntax applies for any node types, as you will see very soon. Every Jac program starts with a single root node, but as you will later learn, a walker can be executed on any node, though the root is used by default if none is specified.
- `spawn here --> graph::faq` creates an instance of the `faq` graph and connects its anchor node to `here`, which is the node the walker is currently on.

> **Note**
>
> `init` can be viewed as similar to `main` in Python. It is the default walker to run when no specific walkers are specified for a `jac run` command.
>
> `here` is a very powerful keyword. It always evaluates to the specific node the walker is currently on. You will be using `here` a lot throughout this tutorial.

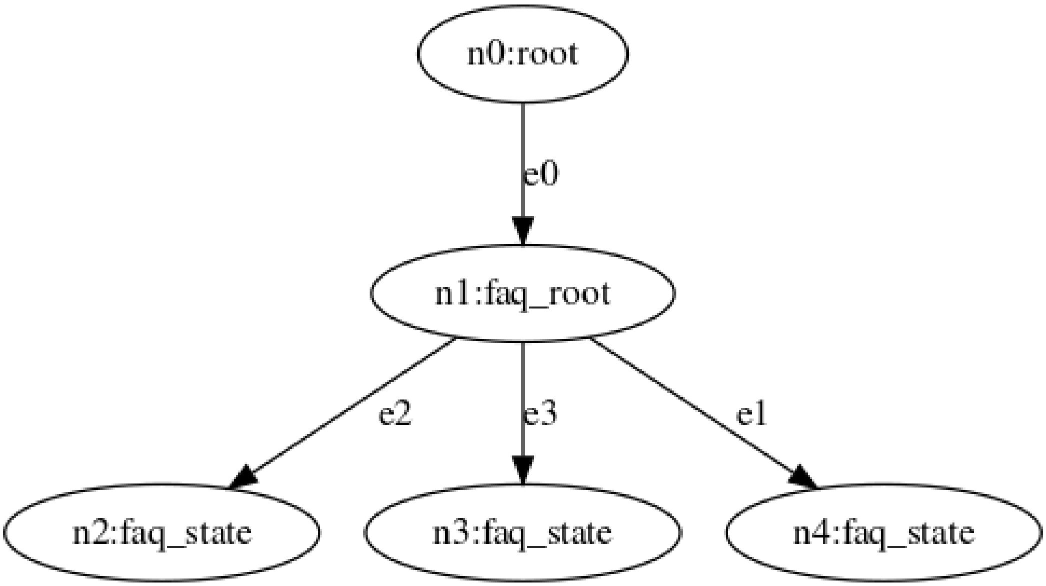


Figure 14.2: Dot output for Faq graph

14.2.4 Run the `init` Walker

Now, let's run the init walker to initialize the graph. First put all of the above code snippet into a single jac file and name it `main.jac`, including

- nodes definition
- graph definition
- init walker

Run `jsctl` to get into the jaseci shell environment:

```
jsctl
```

Inside the `jsctl` shell,

```
jaseci > jac dot main.jac
```

This command runs the `init` walker of the `main.jac` program and prints the state of its graph in DOT format after the walker has finished. The DOT language is a popular graph description language widely used for representing complex graphs.

The output should look something like this

```
strict digraph root {
```

```
    "n0" [ id="0955c04e4ff945b4b836748ef2bbd98a", label="n0:root" ]
    "n1" [ id="c1240d79110941c1bc2feb18581951bd", label="n1:faq_root" ]
    "n2" [ id="55333be285c246db88181ac34d16cd20", label="n2:faq_state" ]
    "n3" [ id="d4fa8f2c46ca463f9237ef818e086a29", label="n3:faq_state" ]
    "n4" [ id="f7b1c8ae82af4063ad53646adc5544e9", label="n4:faq_state" ]
    "n0" -> "n1" [ id="a718fd6c938149269d3ade2af2eb023c", label="e0" ]
    "n1" -> "n2" [ id="3757cb15851249b4b6083d7cb3c34f8e", label="e1" ]
    "n1" -> "n4" [ id="626ce784a8f5423cae5d5d5ca857fc5c", label="e2" ]
    "n1" -> "n3" [ id="a609e7b54bde4a6a9c9711afdb123241", label="e3" ]
}
```

Note

We are not going to cover the DOT syntax. There are many resources online if you are interested, e.g., https://graphviz.org/doc/info/lang.html

Note

There are tools available to render a graph in DOT format. For example, https://dreampuf.github.io/GraphvizOnline has a WSIWYG editor to render dot graph in real time.

Congratulations! You have just created your first functional jac program!

14.2.5 Ask the Question

Alright, we have initialized the graph. Now it's time to create the code for the question-answering. We will start with a simple string matching for the answer selection algorithm. For this, we will create a new walker called `ask`.

```
walker ask {
    has question;
    root {
        question = std.input("AMA > ");
        take --> node::faq_root;
    }
    faq_root {
        take --> node::faq_state(question==question);
    }
    faq_state {:
        std.out(here.answer);
    }
```

```
}
```

This walker is more complex than the `init` one and introduces a few new concepts so let's break it down!

- Similar to nodes, walkers can also contain `has` variables. They define variables of the walker. They can also be passed as parameters when calling the walker.
- `std.input` and `std.out` read and write to the command line respectively.
- This walker has logic for three types of node: `root`, `faq_root` and `faq_state`.
  - `root`: It simply traverses to the `faq_root` node.
  - `faq_root`: This is where the answer selection algorithm is. We will find the most relevant `faq_state` and then traverse to that node via a `take` statement. In this code snippet, we are using a very simple (and limited) string matching approach to try to match the predefined FAQ question with the user question.
  - `faq_state`: Print the answer to the terminal.

Before we run this walker, we are going to update the `init` walker to speed up our development process

```
walker init {
    root {
        spawn here --> graph::faq;
        spawn here walker::ask;
    }
}
```

This serves as a shorthand so that we can initialize the graph and ask a question in one command.

> **Note**
>
> This demonstrates how one walker can spawn another walker using the `spawn` keyword.

Time to run the walker!

```
jaseci > jac run main.jac
```

`jac run` functions very similarly to `jac dot`, with the only difference being that it doesn't return the graph in DOT format. Try giving it one of the three questions we have predefined and it should respond with the corresponding answer.

14.2.6 Introducing Universal Sentence Encoder

Now, obviously, what we have now is not very "AI" and we need to fix that. We are going to use the Universal Sentence Encoder QA model as the answer selection algorithm. Universal

Sentence Encoder is a language encoder model that is pre-trained on a large corpus of natural language data and has been shown to be effective in many NLP tasks. In our application, we are using it for zero-shot question-answering, i.e. no custom training required.

Jaseci has a set of built-in libraries or packages that are called Jaseci actions. These actions cover a wide-range of state-of-the-art AI models across many different NLP tasks. These actions are packaged in a Python module called `jaseci_ai_kit`.

To install `jaseci_ai_kit`:

```
pip install jaseci_ai_kit
```

Now we load the action we need into our jaseci environment

```
jaseci > actions load module jaseci_ai_kit.use_qa
```

Let's update our walker logic to use the USE QA model:

```
walker ask {
    can use.qa_classify;
    has question;
    root {
        question = std.input(">");
        take --> node::faq_root;
    }
    faq_root {
        answers = -->.answer;
        best_answer = use.qa_classify(
            text = question,
            classes = answers
        );
        take --> node::faq_state(answer==best_answer["match"]);
    }
    faq_state {
        std.out(here.answer);
    }
}
```

Even though there are only 5 lines of new code, there are many interesting aspects, so let's break it down!

- `-->.answer` collects the `answer` variable of all of the nodes that are connected to `here/faq_root` with a `-->` connection.
- `use.qa_classify` is one of the action supported by the USE QA action set. It takes in a question and a list of candidate answers and return the most relevant one.

Now let's run this new updated walker and you can now ask questions that are relevant to the answers beyond just the predefined ones.

14.2.7 Scale it Out

So far we have created a FAQ bot that is capable of providing answer in three topics. To make this useful beyond just a prototype, we are now going to expand its database of answers. Instead of manually spawning and connecting a node for each FAQ entry, we are going to write a walker that automatically expands our graph:

```
walker ingest_faq {
    has kb_file;
    root: take --> node::faq_root;
    faq_root {
        kb = file.load_json(kb_file);
        for faq in kb {
            answer = faq["answer"];
            spawn here --> node::faq_state(answer=answer);
        }
    }
}
```

An example knowledge base file look like this

```
[
  {
    "question": "I have a Model 3 reservation, how do I configure my order?",
    "answer": "To configure your order, log into your Tesla Account and select manage on your existing reservation to configure your Tesla. Your original USD deposit has now been converted to SGD. "
  },
  {
    "question": "How do I order a Tesla?",
    "answer": "Visit our Design Studio to explore our latest options and place your order. The purchase price and estimated delivery date will change based on your configuration."
  },
  {
    "question": "Can I request a Test Drive?",
    "answer": "Yes, you can request for a test drive. Please note that drivers must be a minimum of 25 years of age and not exceeding 
```

```
65 years of age, hold a full driving license with over 2 years of driving experience. Insurance conditions relating to your specific status must be reviewed and accepted prior to the test drive."
  }
]
```

Save the above json in a file named `tesla_faq.json` and make sure it is in the same location as `main.jac`. Let's now update the `init` walker. Because we are going to use the `ingest_faq` walker to generate the graph, we won't need the static graph definition.

```
walker init {
    root {
        spawn here --> node::faq_root;
        spawn here walker::ingest_faq(kb_file="tesla_faq.json");
        spawn here walker::ask;
    }
}
```

What we are doing here is

- Spawning a `faq_root` node
- Running the `ingest_faq` walker to create the neccessary `faq_state` nodes based on the question-answer entries in the `tesla_faq.json` file.
- Launching the `ask` walker

Let's run the program one more time and test it out!

```
jaseci > jac run main.jac
```

Note

Try more varied questions. Now we have a longer answer with more rich information, it has a higher coverage of information that will be able to answer more questions.

Note

If you are feeling adventurous, try downloading the complete list of entires on the Tesla FAQ page and use it to create a production-level FAQ bot. See if you can push the model to its limit!

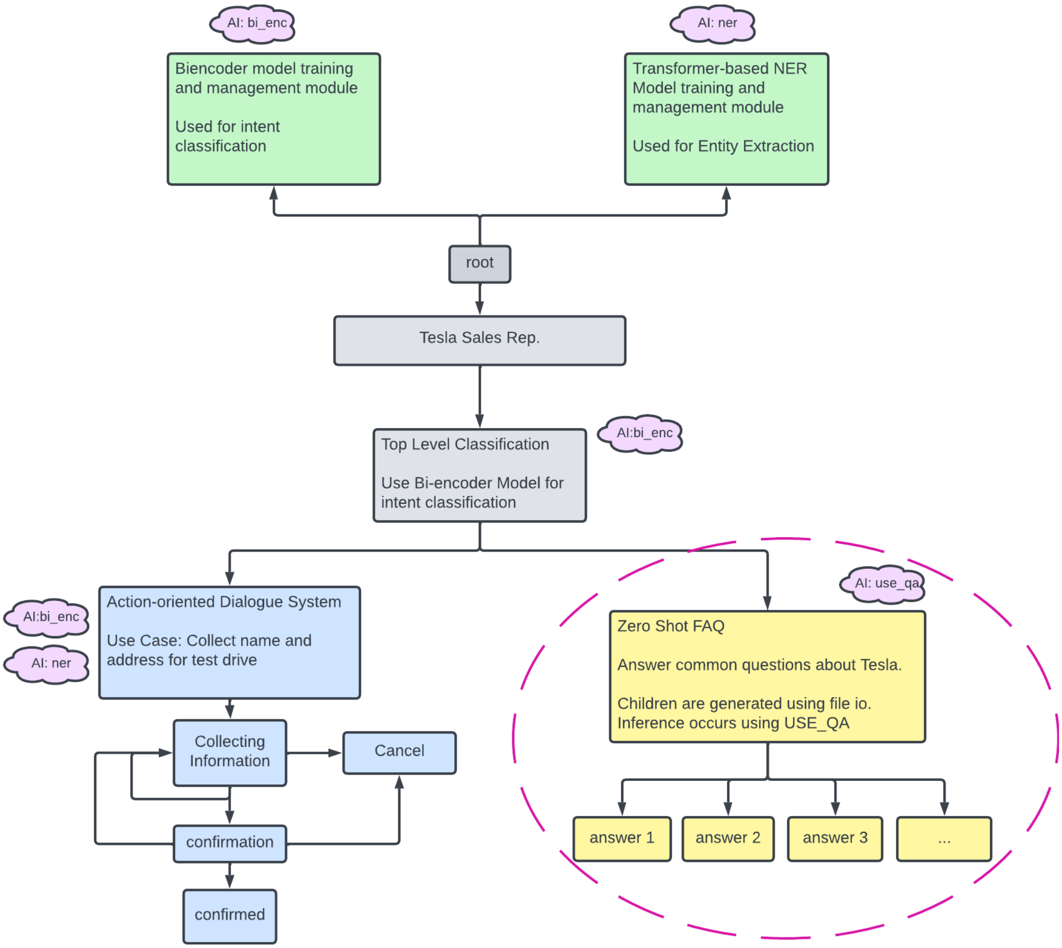


Figure 14.3: Full architecture of Tesla AI

14.3 Next up!

Here is a preview on what's next to come in this journey!

On the right is the architecture diagram of the complete system we are going to build. Here are the major components:

- Zero-shot FAQ (what we have built so far).
- Action-oriented Multi-turn Dialogue System.
- Training and inference with an intent classification model.
- Training and inference with an entity extraction model.
- Testing.
- Deploying your Jac application to a production environment.
- Training data collection and curation.

14.4 A Multi-turn Action-oriented Dialogue System

14.4.1 Introduction

In the previous section, we built a FAQ chabot. It can search in a knowledge base of answers and find the most relevant one to a user's question. While ths covers many diverse topics, certain user request can not be satisfied by a single answer. For example, you might be looking to open a new bank account which requires mulitple different pieces of information about you. Or, you might be making a reservation at a restaurant which requires information such as date, time and size of your group. We refer to these as action-oriented conversational AI requests, as they often lead to a certain action or objective.

When interacting with a real human agent to accomplish this type of action-oriented requests, the interaction can get messy and unscripted and it also varies from person to person. Again, use the restaurant reservation as an example, one migh prefer to follow the guidance of the agent and provide one piece of information at a time, while others might prefer to provide all the neccessary information in one sentence at the beginning of the interaction.

Therefore, in order to build a robust and flexible conversational AI to mimic a real human agent to support these types of messy action-oriented requests, we are going to need an architecture that is different than the single-turn FAQ.

And that is what we are going to build in this section – a multi-turn action-oriented dialogue system.

> **Warning**
>
> Create a new jac file (`dialogue.jac`) before moving forward. We will keep this program separate from the FAQ one we built. But, KEEP the FAQ jac file around, we will

> integrate these two systems into one unified conversational AI system later.

14.4.2 State Graph

Let's first go over the graph architecture for the dialogue system. We will be building a state graph. In a state graph, each node is a conversational state, which represents a possible user state during a dialgoue. The state nodes are connected with transition edges, which encode the condition required to hop from one state to another state. The conditions are often based on the user's input.

14.4.3 Define the State Nodes

We will start by defining the node types.

```
node dialogue_root;

node dialogue_state {
    has name;
    has response;
}
```

Here we have a `dialogue_root` as the entry point to the dialogue system and multiple `dialogue_state` nodes representing the conversational states. These nodes will be connected with a new type of edge `intent_transition`.

14.4.4 Custom Edges

```
edge intent_transition {
    has intent;
}
```

This is the first custom edge we have introduced. In jac, just like nodes, you can define custom edge types. Edges are also allowed `has` variables.

In this case, we created an edge for intent transition. This is a state transition that will be triggered conditioned on its intent being detected from the user's input question.

> **Note**
>
> Custom edge type and variables enable us to encode information into edges in addition to nodes. This is crucial for building a robust and flexible graph.

14.4.5 Build the graph

Let's build the first graph for the dialogue system.

```
graph dialogue_system {
    has anchor dialogue_root;
    spawn {
        dialogue_root = spawn node::dialogue_root;
        test_drive_state = spawn node::dialogue_state(
            name = "test_drive",
            response = "Your test drive is scheduled for Jan 1st, 2023."
        );
        how_to_order_state = spawn node::dialogue_state (
            name = "how_to_order",
            response = "You can order a Tesla through our design studio."
        );

        dialogue_root -[intent_transition(intent="test drive")]-> test_drive_state;
        dialogue_root -[intent_transition(intent="order a tesla")]-> how_to_order_state;
    }
}
```

We have already covered the syntax for graph definition, such as the `anchor` node and the `spawn` block in the previous section. Refer to the FAQ graph definition step if you need a refresher.

We have a new language syntax here `dialogue_root -[intent_transition(intent="test drive")]-> test_drive_state;`. Let's break this down! * If you recall, we have used a similar but simpler syntax to connect two nodes with an edge `faq_root --> faq_state;`. This connect `faq_root` to `faq_state` with a **generic** edge pointing to `faq_state`; * In `dialogue_root -[intent_transition(intent="test drive")]-> test_drive_state;`, we are connecting the two states with a **custom** edge of the type `intent_transition`. * In addition, we are initializing the variable `intent` of the edge to be `test drive`.

To summarize, with this graph, a user will start at the dialogue root state when they first start the conversation. Then based on the user's question and its intent, we will

14.4.6 Initialize the graph

Let's create an `init` walker to for this new jac program.

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

Put all the code so far in a new file and name it `dialogue.jac`.

Let's initialize the graph and visualize it.

```
jaseci > jac dot dialogue.jac
```

```
strict digraph root {
    "n0" [ id="7b4ee7198c5b4dcd8acfcf739d6971fe", label="n0:root" ]
    "n1" [ id="7caf939cfbce40d4968d904052368f30", label="n1:dialogue_root" ]
    "n2" [ id="2e06be95aed449b59056e07f2077d854", label="n2:dialogue_state" ]
    "n3" [ id="4aa3e21e13eb4fb99926a465528ae753", label="n3:dialogue_state" ]
    "n1" -> "n3" [ id="6589c6d0dd67425ead843031c013d0fc", label="e0:intent_transition" ]
    "n1" -> "n2" [ id="f4c9981031a7446b855ec91b89aaa5ee", label="e1:intent_transition" ]
    "n0" -> "n1" [ id="bec764e7ee4048898799c2a4f01b9edb", label="e2" ]
}
```

14.4.7 Build the Walker Logic

Let's now start building the walker to interact with this dialogue system.

```
walker talk {
    has question;
    root {
        question = std.input("> ");
        take --> node::dialogue_root;
    }
    dialogue_root {
        take -[intent_transition(intent==question)]-> node::dialogue_state;
    }
    dialogue_state {
```

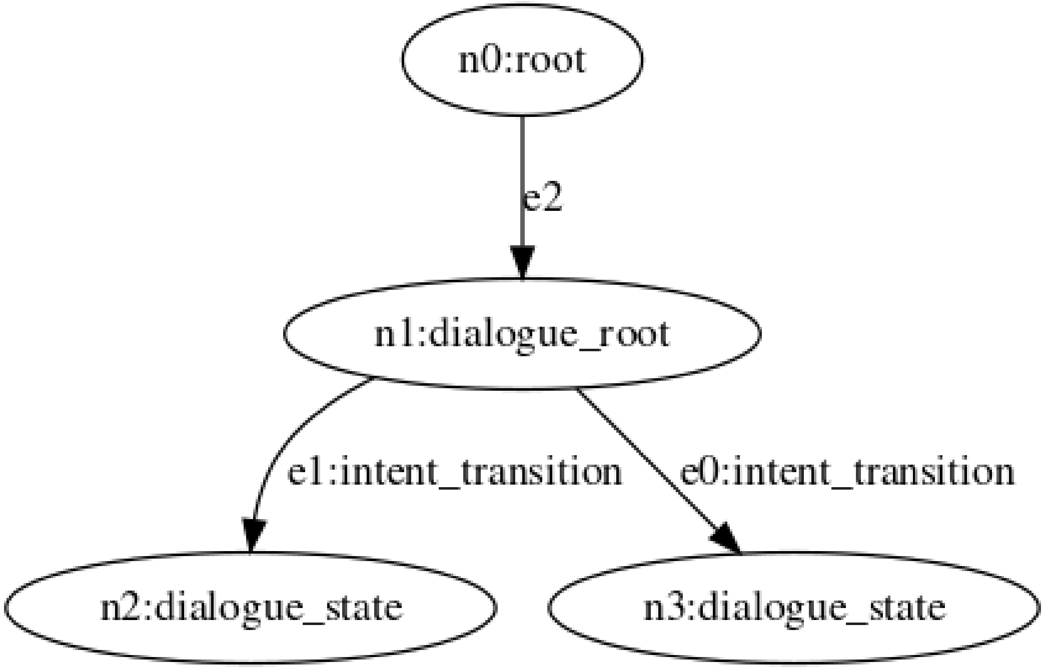


Figure 14.4: DOT of the dialogue system

```
        std.out(here.response);
    }
}
```

Similar to the first walker we built for the FAQ system, we are starting with a simple string matching algorithm. Let's update the init walker to include this walker.

```
walker init {
    root {
        spawn here --> graph::dialogue_system;
        spawn here walker::talk;
    }
}
```

Try out the following interactions

```
$ jsctl jac run dialogue.jac
> test drive
Your test drive is scheduled for Jan 1st, 2023.
{
  "success": true,
  "report": [],
  "final_node": "urn:uuid:9b8d9e1e-d7fb-4e6e-ae86-7ef7c7ad28a7",
  "yielded": false
}
```

and

```
$ jsctl jac run dialogue.jac
> order a tesla
You can order a Tesla through our design studio.
{
  "success": true,
  "report": [],
  "final_node": "urn:uuid:168590aa-d579-4f22-afe7-da75ab7eefa3",
  "yielded": false
}
```

What is happening here is based on the user's question, we are traversing the corresponding dialogue state and then return the response of that state. For now, we are just matching the incoming question with the intent label as a simple algorithm, which we will now replace with an AI model.

> **Note**
>
> Notice we are running jsctl commands directly from the terminal without first entering the jaseci shell? Any jsctl commands can be launched directly from the terminal by just prepending it with jsctl. Try it with the other jsctl comamnds we have encountered so far, such as jac dot.

14.4.8 Intent classificaiton with Bi-encoder

Let's introduce an intent classification AI model. Intent Classification is the task of detecting and assigning an intent to a given piece of text from a list of pre-defined intents, to summarize what the text is conveying or asking. It's one of the fundamental tasks in Natural Language Processing (NLP) with broad applications in many areas.

There are many models that have been proposed and applied to intent classification. For this tutorial, we are going to use a Bi-Encoder model. A Bi-encoder model has two transformer-based encoders that each encodes the input text and candidate intent labels into embedding vectors and then the model compare the similarity between the embedding vectors to find the most relevant/fitting intent label.

> **Note**
>
> If you don't fully understand the Bi-encoder model yet, do not worry! We will provide the neccessary code and tooling for you to wield this model as a black box. But, if you are interested, here is a paper for you to read up on it https://arxiv.org/pdf/1908.10084.pdf!

Now let's train the model. We have created a jac program and sample training data for this. They are in the code directory next to this tutorial. Copy bi_enc.jac and clf_train_1.json to your working directory.

Let's first load the Bi-encoder action library into Jaseci.

```
$ jsctl
jaseci > actions load module jaseci_ai_kit.bi_enc
```

We have provided an example training file that contains some starting point training data for the two intents, test drive and order a tesla.

```
jaseci > jac run bi_enc.jac -walk train -ctx "{\"train_file\":␣\"
    ↪ clf_train_1.json\"}"
```

We are still using `jac run` but as you have noticed, this time we are using some new arguments. So let's break it down. * `-walk` specifies the name of the walker to run. By default, it runs the `init` walker. * `-ctx` stands for `context`. This lets us provide input parameters to the walker. The input parameters are defined as `has` variables in the walker.

> **Warning**
>
> `-ctx` expects a json string that contains a dictionary of parameters and their values. Since we are running this on the command line, you will need to escape the quotation marks `"` properly for it to be a valid json string. Pay close attention to the example here `-ctx "{\"train_file\": \"clf_train_1.json\"}"` and use this as a reference.

You should see an output block that looks like the following repeating many times on your screen:

```
...
Epoch : 5
loss : 0.10562849541505177
LR : 0.0009854014598540146
...
```

Each training epoch, the above output will print with the training loss and learning rate at that epoch. By default, the model is trained for 50 epochs.

If the training successfully finishes, you should see `"success": true` at the end.

Now that the model has finished training, let's try it out! You can use the `infer` walker to play with the model and test it out! `infer` is short for inference, which means using a trained model to run prediction on a given input.

```
jaseci > jac run bi_enc.jac -walk infer -ctx "{\"labels\": [\"test drive\", \"order a tesla\"]}"
```

Similar to training, we are using `jac run` to specifically invoke the `infer` walker and provide it with custom parameters. The custom paremeter is the list of candidate intent labels, which are `test drive` and `order a tesla` in this case, as these were the intents the model was trained on.

```
jaseci > jac run bi_enc.jac -walk infer -ctx "{\"labels\": [\"test drive\", \"order a tesla\"]}"
Enter input text (Ctrl-C to exit)> i want to order a tesla
{"label": "order a tesla", "score": 9.812651595405981}
Enter input text (Ctrl-C to exit)> i want to test drive
{"label": "test drive", "score": 6.931458692617463}
Enter input text (Ctrl-C to exit)>
```

In the output here, `label` is the predicted intent label and `score` is the score assigned by the model to that intent.

> **Note**
>
> One of the advantage of the bi-encoder model is that candidate intent labels can be dynamically defined at inference time, post training. This enables us to create custom contextual classifiers situationally from a single trained model. We will leverage this later as our dialogue system becomes more complex.

Congratulations! You just trained your first intent classifier, easy as that.

The trained model is kept in memory and active until they are explicitly saved with `save_model`. To save the trained model to a location of your choosing, run

```
jaseci > jac run bi_enc.jac -walk save_model -ctx "{\"model_path\": \"dialogue_intent_model\"}"
```

Similarly, you can load a saved model with `load_model`

```
jaseci > jac run bi_enc.jac -walk load_model -ctx "{\"model_path\": \"dialogue_intent_model\"}"
```

Always remember to save your trained models!

> **Warning**
>
> `save_model` works with relative path. When a relative model path is specified, it will save the model at the location relative to **location of where you run jsctl**. Note that until the model is saved, the trained weights will stay in memory, which means that it will not persisit between `jsctl` session. So once you have a trained model you like, make sure to save them so you can load them back in the next jsctl session.

14.4.9 Integrate the Intent Classifier

Now let's update our walker to use the trained intent classifier.

```
walker talk {
    has question;
    can bi_enc.infer;
    root {
        question = std.input("> ");
        take --> node::dialogue_root;
    }
```

```
    dialogue_root {
        intent_labels = -[intent_transition]->.edge.intent;
        predicted_intent = bi_enc.infer(
            contexts = [question],
            candidates = intent_labels,
            context_type = "text",
            candidate_type = "text"
        )[0]["predicted"]["label"];
        take -[intent_transition(intent==predicted_intent)]-> node::
            ↪ dialogue_state;
    }
    dialogue_state {
        std.out(here.response);
    }
}
```

`intent_labels = -[intent_transition]->.edge.intent` collects the `intent` variables of all the outgoing `intent_transition` edges. This represents the list of candidate intent labels for this state.

Try playing with different questions, such as

```
$ jsctl
jaseci > jac run dialogue.jac
> hey yo, I heard tesla cars are great, how do i get one?
You can order a Tesla through our design studio.
{
  "success": true,
  "report": [],
  "final_node": "urn:uuid:af667fdf-c2b0-4443-9ccd-7312bc4c66c4",
  "yielded": false
}
```

14.4.10 Making Our Dialogue System Multi-turn

Dialogues in real life have many turn of interaction. Our dialogue system should also support that to provide a human-like conversational experinece. In this section, we are going to take the dialogue system to the next level and create a multi-turn dialogue experience.

Before we do that we need to introduce two new concepts in Jac: node abilities and inheritance.

14.4.10.1 Node Abilities

Node abilities are code that encoded as part of each node type. They often contain logic that read, write and generally manipulate the variables and states of the nodes. Node abilities are defined with the `can` keyword inside the definition of nodes, for example, in the code below, `get_plate_number` is an ability of the `vehicle` node.

```
node vehicle {
    has plate_numer;
    can get_plate_numer {
        report here.plate_number;
    }
}
```

To learn more about node abilities, refer to the relevant sections of the Jaseci Bible. > **Note** > > Node abilities look and function similarly to member functions in object-oriented programming (OOP). However, there is a key difference in the concepts. Node abilities are the key concept in data-spatial programming, where the logic should stay close to its working set data in terms of the programming syntax.

14.4.10.2 Inheritance

Jac supports inheritance for nodes and edges. Node variables (defined with `has`) and node abilities (defined with `can`) are inherited and can be overwritten by children nodes.

Here is an example:

```
node vehicle {
    has plate_number;
    can get_plate_number {
        report here.plate_number;
    }
}

node car:vehicle {
    has plate_number = "RAC001";
}

node bus:vehicle {
    has plate_number = "SUB002";
}
```

To learn more about inheritance in Jac, refer to the relevant sections of the Jaseci Bible.

14.4.11 Build the Multi-turn Dialogue Graph

Now that we have learnt about node abilities and node inheritance, let's put these new concepts to use to build a new graph for the multi-turn dialogue system

There are multiple parts to this so let's break it down one by one

14.4.11.1 Dialogue State Specific Logic

With the node abilities and node inheritance, we will now introduce state specific logic. Take a look at how the `dialogue_root` node definition has changed.

```
node dialogue_state {
    can bi_enc.infer;
    can tfm_ner.extract_entity;

    can classify_intent {
        intent_labels = -[intent_transition]->.edge.intent;
        visitor.wlk_ctx["intent"] = bi_enc.infer(
            contexts = [visitor.question],
            candidates = intent_labels,
            context_type = "text",
            candidate_type = "text"
        )[0]["predicted"]["label"];
    }

    can extract_entities {
        // Entity extraction logic will be added a bit later on.
    }

    can init_wlk_ctx {
        new_wlk_ctx = {
            "intent": null,
            "entities": {},
            "prev_state": null,
            "next_state": null,
            "respond": false
        };
        if ("entities" in visitor.wlk_ctx) {
            // Carry over extracted entities from previous interaction
            new_wlk_ctx["entities"] = visitor.wlk_ctx["entities"];
        }
        visitor.wlk_ctx = new_wlk_ctx;
    }
```

```
    can nlu {}
    can process {
        if (visitor.wlk_ctx["prev_state"]): visitor.wlk_ctx["respond"] = true;
        else {
            visitor.wlk_ctx["next_state"] = net.root();
            visitor.wlk_ctx["prev_state"] = here;
        }
    }
    can nlg {}
}

node dialogue_root:dialogue_state {
    has name = "dialogue_root";
    can nlu {
        ::classify_intent;
    }
    can process {
        visitor.wlk_ctx["next_state"] = (-[intent_transition(intent== visitor.wlk_ctx["intent"])]->)[0];
    }
    can nlg {
        visitor.response = "Sorry I can't handle that just yet. Anything else I can help you with?";
    }
}
```

There are many interesting things going on in these ~30 lines of code so let's break it down! * The `dialogue_state` node is the parent node and it is similar to a virtual class in OOP. It defines the variables and abilities of the nodes but the details of the abilities will be specified in the inheriting children nodes. * In this case, `dialogue_state` has 4 node abilities: * `can nlu`: NLU stands for Natural Language Understanding. This ability will analyze user's incoming requset and apply AI models. * `can process`: This ability uses the NLU results and figure out the next dialogue state the walker should go to. * `can nlg`: NLG stands for Natural Language Generation. This abilitiy will compose response to the user, often based on the results from `nlu`. * `can classify_intent`: an ability to handle intent classification. This is the same intent classification logic that has been copied over from the walker. * `can extract_entities`: a new ability with a new AI model – entity extraction. We will cover that just in a little bit (read on!). * Between these four node abilities, `classify_intent` and `extract_entities` have concrete logic defined while `nlu` and `nlg` are "virtual node abilities", which will be specified in each of the inheriting children. * For example, `dialogue_root` inherit from `dialogue_state` and overwrites `nlu` and `nlg`: * for `nlu`, it invokes intent classification because it needs to decide what's the intent of the user (test drive vs order a tesla). * for `nlg`, it just has a general fall-back response in case the system can't handle user's ask. * **New Syntax**: `visitor` is the walker that is "visiting" the node. And through `visitor.*`, the node abilities can access and update the context of the walker. In this case, the node abilities are updating the `response` variable in the walker's context so that the walker can return the response to its caller, as well as the `wlk_ctx` variable that will contain various walker context as the walker traverse the graph. * the `init_wlk_ctx` ability initializes the `wlk_ctx` variable for each new question.

In this new node architecture, each dialogue state will have its own node type, specifying their state-specific logic in `nlu`, `nlg` and `process`. Let's take a look!

```
node how_to_order_state:dialogue_state {
    has name = "how_to_order";
    can nlg {
        visitor.response = "You can order a Telsa through our design studio";
    }
}

node test_drive_state:dialogue_state {
    has name = "test_drive";
    can nlu {
        if (!visitor.wlk_ctx["intent"]): ::classify_intent;
        ::extract_entities;
    }
    can process {
        // Check entity transition
        required_entities = -[entity_transition]->.edge[0].context["entities"];
        if (vector.sort_by_key(visitor.wlk_ctx["entities"].d::keys) == vector.sort_by_key(required_entities)) {
            visitor.wlk_ctx["next_state"] = -[entity_transition]->[0];
            visitor.wlk_ctx["prev_state"] = here;
        } elif (visitor.wlk_ctx["prev_state"] and !visitor.wlk_ctx["prev_state"].context["name"] in ["test_drive", "td_confirmation"]){
            next_state = -[intent_transition(intent==visitor.wlk_ctx["intent"])]->;
            if (next_state.length > 0 and visitor.wlk_ctx["intent"] != "no") {
                visitor.wlk_ctx["next_state"] = next_state[0];
                visitor.wlk_ctx["prev_state"] = here;
            } else {
                visitor.wlk_ctx["respond"] = true;
```

```
            }
        } else {
            visitor.wlk_ctx["respond"] = true;
        }
    }
    can nlg {
        if ("name" in visitor.wlk_ctx["entities"] and "address" not in visitor.wlk_ctx["entities"]):
            visitor.response = "What is your address?";
        elif ("address" in visitor.wlk_ctx["entities"] and "name" not in visitor.wlk_ctx["entities"]):
            visitor.response = "What is your name?";
        else:
            visitor.response = "To set you up with a test drive, we will need your name and address.";
    }
}

node td_confirmation:dialogue_state {
    has name = "test_drive_confirmation";
    can nlu {
        if (!visitor.wlk_ctx["intent"]): ::classify_intent;
    }
    can process {
        if (visitor.wlk_ctx["prev_state"]): visitor.wlk_ctx["respond"] = true;
        else {
            visitor.wlk_ctx["next_state"] = -[intent_transition(intent==visitor.wlk_ctx["intent"])]->[0];
            visitor.wlk_ctx["prev_state"] = here;
        }
    }
    can nlg {
        visitor.response =
            "Can you confirm your name to be " + visitor.wlk_ctx["entities"]["name"][0] + " and your address as " + visitor.wlk_ctx["entities"]["address"][0] + "?";
    }
}

node td_confirmed:dialogue_state {
    has name = "test_drive_confirmed";
    can nlg {
```

```
        visitor.response = "You are all set for a Tesla test drive!";
    }
}

node td_canceled:dialogue_state {
    has name = "test_drive_canceled";
    can nlg {
        visitor.response = "No worries. We look forward to hearing from you in the future!";
    }
}
```

- Each dialogue state now has its own node type, all inheriting from the same generic `dialogue_state` node type.
- We have 4 dialogue states here for the test drive capability:
  - `test_drive`: This is the main state of the test drive intent. It is responsible for collecting the neccessary information from the user.
  - `test_drive_confirmation`: Ths is the state for user to confirm the information they have provided are correct and is ready to actually schedule the test drive.
  - `test_drive_confirmed`: This is the state after the user has confirmed.
  - `test_drive_canceled`: User has decided, in the middle of the dialogue, to cancel their request to schedule a test drive.
- The `process` ability contains the logic that defines the conversational flow of the dialogue system. It uses the data in `wlk_ctx` and assign a `next_state` which will be used by the walker in a `take` statement, as you will see in a just a little bit.
- **New Syntax**: The code in `test_drive_state`'s ability demonstrates jac support for list and dictionary. To access the list and dictionary-specific functions, first cast the variable with `.l`/`.list` for list and `.d`/`.dict` for dictionaries, then proceed with : to access the built-in functions for list and dictioinaries. For more on jac's built-in types, refer to the relevant sections of the Jaseci Bible.
  - Specifically in this case, we are comparing the list of entities of the `entity_transition` edge with the list of entities that have been extracted by the walker and the AI model (stored in `wlk_ctx["entities]`). Since there can be multiple entities required and they can be extracted in arbitrary order, we are sorting and then comparing here.
- **New Syntax**: `-[entity_transition]->.edge` shows how to access the edge variable. Consider `-[entity_transition]->` as a filter. It returns all valid nodes that are connected to the implicit `here` via an `entity_transition`. On its own, it will return all the qualified nodes. When followed by `.edge`, it will return the set of edges that are connected to the qualified nodes.

You might notice that some states do not have a `process` ability. These are states that do not have any outgoing transitions, which we refer to as leaf nodes. If these nodes are reached, they indicate that a dialogue has been completed end to end. The next state for these node will be returning to the root node so that the next dialogue can start fresh. To facilitate this, we will add the following logic to the `process` ability of the **parent `dialogue_state` node** so that by default, any nodes inheriting it will follow this rule.

```
node dialogue_state {
...
    can process {
        if (visitor.wlk_ctx["prev_state"]): visitor.wlk_ctx["respond"] =
            ↪ true;
        else {
            visitor.wlk_ctx["next_state"] = net.root();
            visitor.wlk_ctx["prev_state"] = here;
        }
    }
...
}
```

> **Note**
>
> Pay attention to the 4 dialogue states here. This pattern of `main` -> `confirmation` -> `confirmed` -> `canceled` is a very common conversational state graph design pattern and can apply to many topics, e.g., make a restaurant reservation and opening a new bank account. Essentially, almost any action-oriented requests can leverage this conversational pattern. Keep this in mind!

14.4.11.2 Entity Extraction

Previously, we have introduced intent classification and how it helps to build a dialogue system. We now introduce the second key AI models, that is specifically important for a multi-turn dialogue system, that is entity/slot extraction.

Entity extraction is a NLP task that focuses on extracting words or phrases of interests, or entities, from a given piece of text. Entity extraction, sometimes also referred to as Named Entity Recognition (NER), is useful in many domains, including information retrieval and conversational AI. We are going to use a transformer-based entity extraction model for this exercise.

Let's first take a look at how we are going to use an entity model in our program. Then we will work on training an entity model.

First, we introduce a new type of transition:

```
edge entity_transition {
    has entities;
```

```
}
```

Recall the `intent_transition` that will trigger if the intent is the one that is being predicted. Similarly, the idea behind an `entity_transition` is that we will traverse this transition if all the specified entities have been fulfilled, i.e., they have been extracted from user's inputs.

With the `entity_transition`, let's update our graph

```
graph dialogue_system {
    has anchor dialogue_root;
    spawn {
        dialogue_root = spawn node::dialogue_root;
        test_drive_state = spawn node::test_drive_state;
        td_confirmation = spawn node::td_confirmation;
        td_confirmed = spawn node::td_confirmed;
        td_canceled = spawn node::td_canceled;

        how_to_order_state = spawn node::how_to_order_state;

        dialogue_root -[intent_transition(intent="test drive")]-> test_drive_state;
        test_drive_state -[intent_transition(intent="cancel")]-> td_canceled;
        test_drive_state -[entity_transition(entities=["name", "address"])]-> td_confirmation;
        test_drive_state -[intent_transition(intent="provide name or address")]-> test_drive_state;
        td_confirmation - [intent_transition(intent="yes")]-> td_confirmed;
        td_confirmation - [intent_transition(intent="no")]-> test_drive_state;
        td_confirmation - [intent_transition(intent="cancel")]-> td_canceled;

        dialogue_root -[intent_transition(intent="order a tesla")]-> how_to_order_state;
    }
}
```

Your graph should look something like this!

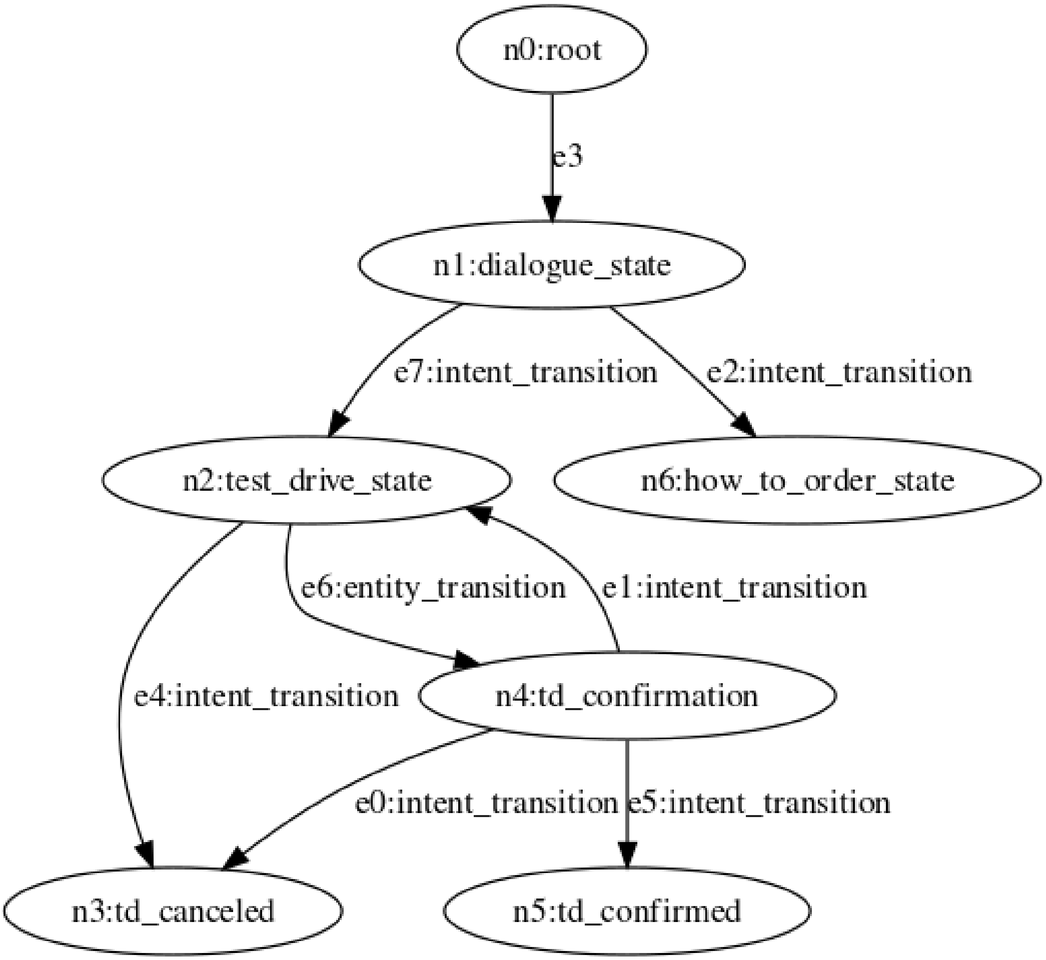


Figure 14.5: Multi-turn Dialogue Graph

14.4.12 Update the Walker for Multi-turn Dialogue

Let's now turn our focus to the walker logic

```
walker talk {
    has question;
    has wlk_ctx = {};
    has response;
    root {
        take --> node::dialogue_root;
    }
    dialogue_state {
        if (!question) {
            question = std.input("Question (Ctrl-C to exit)> ");
            here::init_wlk_ctx;
        }
        here::nlu;
        here::process;
        if (visitor.wlk_ctx["respond"]) {
            here::nlg;
            std.out(response);
            question = null;
            take here;
        } else {
            take visitor.wlk_ctx["next_state"] else: take here;
        }
    }
}
```

The walker logic looks very different now. Let's break it down! * First off, because the intent classification logic is now a node ability, the walker logic has become simpler and, more importantly, more focused on graph traversal logic without the detailed (and occasionally convoluted) logic required to process to interact with an AI model. * **New Syntax**: `here` ↪ `::nlu` and `here::nlg` invokes the node abilities. `here` can be subtitied with any node variables, not just the one the walker is currently on.

Now that we have explained some of the new language syntax here, let's go over the overall logic of this walker. For a new question from the user, the walker will 1. analyze the question (`here:nlu`) to identify its intent (`predicted_intent`) and/or extract its entities (`extracted_entities`). 2. based on the NLU results, it will traverse the dialogue state graph (the two `take` statements) to a new dialogue state 3. at this new dialogue state, it will perform NLU, specific to that state (recall that `nlu` is a node ability that varies from node to node) and repeat step 2 4. if the walker can not make any state traversal anymore (`take ... else {}`), it will construct a response (`here::nlg`) using the information it has gathered so far (the walker's context) and return that response to the user.

If this still sounds fuzzy, don't worry! Let's use a real dialogue as an example to illustrate this.

```
Turn #1:
    User: hey i want to schedule a test drive
    Tesla AI: To set you up with a test drive, we will need your name and
        ↪ address.

Turn #2:
    User: my name is Elon and I live at 123 Main Street
    Tesla AI: Can you confirm your name to be Elon and your address as 123
        ↪  Main Street?

Turn #3:
    User: Yup! that is correct
    Tesla AI: You are all set for a Tesla test drive!
```

At turn #1, * The walker starts at `dialogue_root`. * The `nlu` at `dialogue_root` is called and classify the intent to be `test drive`. * There is an `intent_transition(` ↪ `test_drive)` connecting `dialogue_root` to `test_drive_state` so the walker `takes` ↪ itself to `test_drive_state` . * We are now at `test_drive_state`, its `nlu` requires `entity_extraction` which will look for `name` and `address` entities. In this case, neither is provided by the user. * As a result, the walker can no longer traverse based on the `take` rules and thus construct a response based on the `nlg` logic at the `test_drive_state`.

At turn #2, * The walker starts at `test_drive_state`, picking up where it left off. * `nlu` at `test_drive_state` perform intent classification and entity extractions. This time it will pick up both name and address. * As a result, the first `take` statement finds a qualified path and take that path to the `td_confirmation` node. * At `td_confirmation`, no valid take path exists so a response is returned.

> **Note**
>
> Turn #3 works similiarly as turn #1. See if you can figure out how the walker reacts at turn #3 yourself!

14.4.13 Train an Entity Extraction Model

Let's now train an entity extraction model! We are using a transformer-based token classification model.

First, we need to load the actions. The action set is called `tfm_ner` (`tfm` stands for transformer).

```
jaseci > actions load module jaseci_ai_kit.tfm_ner
```

> **Warning**
>
> If you installed `jaseci_ai_kit` prior to September 5th, 2022, please upgrade via `pip install --upgrade jaseci_ai_kit`. There has been an update to the module that you will need for remainder of this exercise. You can check your installed version via `pip show jaseci_ai_kit`. You need to be on version 1.3.4.6 or higher.

Similar to Bi-encoder, we have provided a jac program to train and inference with this model, as well as an example training dataset. Go into the `code/` directory and copy `tfm_ner.jac` and `ner_train.json` to your working directory. We are training the model to detect two entities, `name` and `address`, for the test drive use case.

Let's quickly go over the training data format.

```
[
    "sure my name is [tony stark](name) and i live at [10880 malibu point 
        ↪ california](address)",
    "my name is [jason](name)"
]
```

The training data is a json list of strings, each of which is a training example. `[]` indicate the entitiy text while the `()` following it defines the entity type. So in the example above, we have two entities, `name:tony stark` and `address: 10880 malibu point california`.

To train the model, run

```
jaseci > jac run tfm_ner.jac -walk train -ctx "{\"train_file\": \"
    ↪ ner_train.json\"}"
```

After the model is finished training, you can play with the model using the `infer` walker

```
jaseci > jac run tfm_ner.jac -walk infer
```

For example,

```
jaseci > jac run tfm_ner.jac -walk infer
Enter input text (Ctrl-C to exit)> my name is jason
[{"entity_text": "jason", "entity_value": "name", "conf_score":
    ↪ 0.5514775514602661, "start_pos": 11, "end_pos": 16}]
```

The output of this model is a list of dictionaries, each of which is one detected entitiy. For each detected entity, `entity_value` is the type of entity, so in this case either `name` or `address`; and `entity_text` is the detected text from the input for this entity, so in this case the user's name or their address.

Let's now update the node ability to use the entity model.

```
node dialogue_state {
    ...
    can extract_entities {
        res = tfm_ner.extract_entity(visitor.question);
        for ent in res {
            ent_type = ent["entity_value"];
            ent_text = ent["entity_text"];
            if (!(ent_type in visitor.wlk_ctx["entities"])){
                visitor.wlk_ctx["entities"][ent_type] = [];
            }
            visitor.wlk_ctx["entities"][ent_type].l::append(ent_text);
        }
    }
    ...
}
```

There is one last update we need to do before this is fully functional. Because we have more dialogue states and a more complex graph, we need to update our classifier to include the new intents. We have provided an example training dataset at `code/clf_train_2.json`. Re-train the bi-encoder model with this dataset.

> **Note**
>
> Refer to previous code snippets if you need a reminder on how to train the bi-encoder classifier model.

> **Note**
>
> Remember to save your new entity extraction model!

Now try running the walker again with `jac run dialogue.jac`!

Congratulations! You now have a fully functional multi-turn dialogue system that can handle test drive requests!

14.5 Unify the Dialogue and FAQ Systems

So far, we have built two separate conversational AI systems, a FAQ system that automatically scales with the available question-answer pairs and a multi-turn action-oriented dialogue system that can handle complex requests. These two systems serve different use cases and can be combined to a single system to provide a flexible and robust conversational AI experience. In this section, we are going to unify these two systems into one coherent conversational AI system.

While these two systems rely on different AI models, they share many of the same logic flow. They both follow the general steps of first analyizing user's question with NLU AI models, make decision on the next conversational state to be and then construct and return a response to the user. Leveraging this shared pattern, we will first unify the node architecture of the two systems with a single parent node type, `cai_state` (`cai` is short of conversational AI).

```
node cai_state {
    has name;
    can init_wlk_ctx {
        new_wlk_ctx = {
            "intent": null,
            "entities": {},
            "prev_state": null,
            "next_state": null,
            "respond": false
        };
        if ("entities" in visitor.wlk_ctx) {
            // Carry over extracted entities from previous interaction
            new_wlk_ctx["entities"] = visitor.wlk_ctx["entities"];
        }
        visitor.wlk_ctx = new_wlk_ctx;
    }
    can nlu {}
    can process {
        if (visitor.wlk_ctx["prev_state"]): visitor.wlk_ctx["respond"] =
          ↪ true;
        else {
            visitor.wlk_ctx["next_state"] = net.root();
            visitor.wlk_ctx["prev_state"] = here;
        }
    }
    can nlg {}
}
```

Note that the logic for `init_wlk_ctx` and the default `process` logic have been hoisted up into `cai_state` as they are shared by the dialogue system and FAQ system. You can remove these two abilities from `dialogue_state` node, as it will be inheriting them from `cai_state` now.

We then update the defintion of `dialogue_state` in `dialogue.jac` to inherit from `cai_state` ↪ :

```
node dialogue_state:cai_state{
    // Rest of dialogue_state code remain the same
}
```

Before we move on, we will take a quick detour to introduce multi-file jac program and how import works in jac.

14.5.1 Multi-file Jac Program and Import

Jac's support for multi-file is quite simple. You can import object definitions from one jac file to another with the `import` keyword. With `import {*} with "./code.jac"`, everything from `code.jac` will be imported, which can include nodes, edges, graph and walker definition. Alternaitvely, you can import specific objects with `import {node::state} with "./code.` ↪ `jac"`.

To compile a multi-file Jac program, you will need one jac file that serves as the entry point of the program. This file need to import all the neccessary components of the program. Chained importing is supported.

Once you have the main jac file (let's call it `main.jac`), you will need to compile it and its imports into a single `.jir` file. `jir` here stands for Jac Intermediate Representation. To compile a jac file, use the `jac build` command

```
jaseci > jac build main.jac
```

If the compilation is successful, a `.jir` file with the same name will be generated (in this case, `main.jir`). `jir` file can be used with `jac run` or `jac dot` the same way as the `jac` source code file.

> **Note**
>
> The `jir` format is what you will use to deploy your jac program to a production jaseci instance.

14.5.2 Unify FAQ + Dialogue Code

For `faq_state`, we need to now define the `nlu` and `nlg` node abilities for FAQ. So let's update the following in `faq.jac` First, `faq_root`

```
node faq_root:cai_state {
    can use.qa_classify;
    can nlu {
        if (!visitor.wlk_ctx["prev_state"]) {
            answers = -->.answer;
            best_answer = use.qa_classify(
                text = visitor.question,
                classes = answers
            );
            visitor.wlk_ctx["intent"] = best_answer["match"];
        }
    }
    can process {
        if (visitor.wlk_ctx["prev_state"]): visitor.wlk_ctx["respond"] = true;
        else {
            for n in --> {
                if (n.context["answer"] == visitor.wlk_ctx["intent"]){
                    visitor.wlk_ctx["next_state"] = n;
                    break;
                }
            }
            visitor.wlk_ctx["prev_state"] = here;
        }
    }
    can nlg {
        visitor.response = "I can answer a variety of FAQs related to Tesla. What can I help you with?";
    }
}
```

At this point, if you have been following this journey along, this code should be relatively easy to understand. Let's quickly break it down. * For FAQ, the `nlu` logic uses the USE QA model to find the most relevant answer. Here we are re-using the `intent` field in the walker context to save the matched answer. You can also opt to create another field dedicated to FAQ NLU result. * For the traversal logic, this is very similar to the previous FAQ logic, i.e. find the `faq_state` node connected to here that contains the most relevant answer. * `for n in -->` iterates through all the nodes connected with an outgoing edge from the current node. You can use `.context` on any node variables to access its variables.

And the logic for the `faq_state` that contains the answer is relatively simple;

```
node faq_state:cai_state {
    has question;
    has answer;
    can nlg {
        visitor.response = here.answer;
    }
}
```

With these new nodes created, let's update our graph definition. We have renamed our graph to be `tesla_ai` and the `dialogue.jac` file to `tesla_ai.jac`.

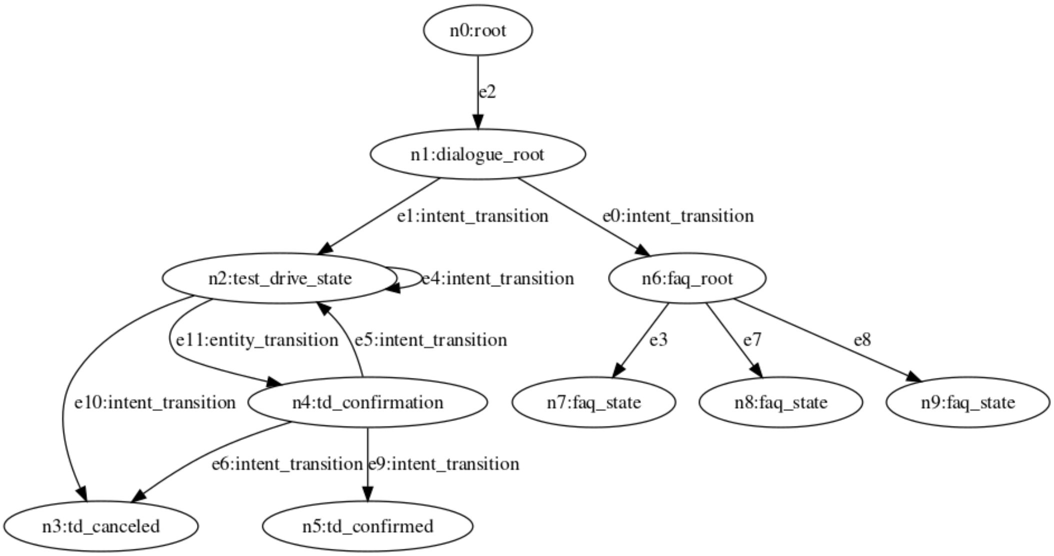
```
graph tesla_ai {
    has anchor dialogue_root;
    spawn {
        dialogue_root = spawn node::dialogue_root;
        test_drive_state = spawn node::test_drive_state;
        td_confirmation = spawn node::td_confirmation;
        td_confirmed = spawn node::td_confirmed;
        td_canceled = spawn node::td_canceled;

        dialogue_root -[intent_transition(intent="test drive")]-> test_drive_state;
        test_drive_state -[intent_transition(intent="cancel")]-> td_canceled;
        test_drive_state -[entity_transition(entities=["name", "address"]) ]-> td_confirmation;
        test_drive_state -[intent_transition(intent="provide name or address")]-> test_drive_state;
        td_confirmation - [intent_transition(intent="yes")]-> td_confirmed ;
        td_confirmation - [intent_transition(intent="no")]-> test_drive_state;
        td_confirmation - [intent_transition(intent="cancel")]-> td_canceled;

        faq_root = spawn graph::faq;
        dialogue_root -[intent_transition(intent="i have a question")]-> faq_root;
    }
}
```

One thing worth pointing out here is that we are spawning a graph inside a graph spawn block.

Our graph should now looks like this!



Here comes the biggest benefit of our unified node architecture – the exact same walker logic can be shared to traverse both systems. The only change we need to make is to change from `dialogue_state` to `cai_state` to apply the walker logic to a more generalized set of nodes.

```
walker talk {
    ...
    root {
        take --> node::dialogue_root;
    }
    cai_state {
        if (!question) {
            question = std.input("Question (Ctrl-C to exit)> ");
            here::init_wlk_ctx;
        }
        ...
    }
}
```

Update the graph name in the `init` walker as well.

```
walker init {
    root {
        spawn here --> graph::tesla_ai;
```

```
        spawn here walker::talk;
    }
}
```

To compile the program,

```
jaseci > jac build tesla_ai.jac
```

As mentioned before, if the compiliation succeedd, a `tesla_ai.jir` will be generated.

> **Note**
>
> Run into issues at this build step? First check if all the imports are set up correctly.

Running a `jir` is just like running a `jac` file

```
jaseci > jac run tesla_ai.jir
```

One last step, since we introduce a new intent `i have a questions`, we need to update our classifier model again. This time, use the `clf_train_3.json` example training data.

The model is trained? Great! Now run the jir and try questions like "I have some telsa related questions" then following with FAQ questioins!

Congratulations! You have created a single conversational AI system that is capable of answering FAQs and perform complex multi-step actions.

14.6 Bring Your Application to Production

Typing in questions and getting responses via `jsctl` in terminal is a quick and easy way of interactively test and use your program. But the ultimate goal of building any products is to eventually deploying it to production and having it serve real users via standard interface such as RESTful API endpoints. In this section, we will cover a number of items related to bringing your jac program to production.

14.6.1 Introducing `yield`

`yield` is a jac keyword that suspend the walker and return a response, which then can be resumed at a later time with the walker context retained. Walker context includes its `has` variables and its node traversal plan (i.e., any nodes that have been queued by previously executed `take` statements). This context retention is done on a per-user basis. `yield` is a great way to maintaining user-specific context and history in between walker calls. To learn more about `yield`, refer to the relevant sections of the Jaseci Bible.

In the case of our conversational AI system, it is essential for our walker to remember the context information gained from previous interactions with the same user. So let's update our walker with `yield`.

```
walker talk {
    has question, interactive = false;
    has wlk_ctx = {
        "intent": null,
        "entities": {},
        "prev_state": null,
        "next_state": null,
        "respond": false
    };
    has response;
    root {
        take --> node::dialogue_root;
    }
    cai_state {
        if (!question and interactive) {
            question = std.input("Question (Ctrl-C to exit)> ");
            here::init_wlk_ctx;
        } elif (!question and !interactive){
            std.err("ERROR: question is required for non-interactive mode");
            disengage;
        }
        here::nlu;
        here::process;
        if (visitor.wlk_ctx["respond"]) {
            here::nlg;
            if (interactive): std.out(response);
            else {
                yield report response;
                here::init_wlk_ctx;
            }
            question = null;
            take here;
        } else {
            take visitor.wlk_ctx["next_state"] else: take here;
        }
    }
}
```

Two new syntax here: * `report` returns variable from walker to its caller. When calling a walker via its REST API, the content of the API response payload will be what is reported. * `yield report` is a shorthand for yielding and reporting at the same time. This is equivalane to `yield; report response;`.

14.6.2 Introduce `sentinel`

`sentinel` is the overseer of walkers, nodes and edges. It is the abstraction Jaseci uses to encapsulate compiled walkers and architype nodes and edges. The key operation with respesct to `sentinel` is "register" a sentinel. You can think of registering a `sentinel` as a 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 program

```
jaseci > sentinel register tesla_ai.jir -set_active true -mode ir
```

Three things are happening here: * First, we registered the `jir` we compiled earlier to new sentinel. This means this new sentinel now has access to all of our walkers, nodes and edges. `-mode ir` option speciifes a `jir` program is registered instead of a `jac` program. * Second, with `-set_active true` we set this new sentinel to be the active sentinel. In other words, this sentinel is the default one to be used when requests hit the Jac APIs, if no specific sentinels are specified. * Third, `sentinel register` has automatically creates a new `graph` (if no currently active graph) and run the `init` walker on that graph. This behavior can be customized with the options `-auto_run` and `-auto_create_graph`.

To check your graph

```
jaseci > graph get -mode dot
```

This will return the current active graph in DOT format. This is the same output we get from running `jac dot` earlier. Use this to check if your graph is successfully created.

Once a sentinel is registered, you can update its jac program with

```
jaseci > sentinel set -snt SENTINEL_ID -mode ir tesla_ai.jir
```

To get the sentinel ID, you can run one of the two following commands

```
jaseci > sentinel get
```

or

```
jaseci > sentinel list
```

`sentinel get` returns the information about the current active sentinel, while `sentinel list` returns all available sentinels for the user. The output will look something like this

```
{
  "version": null,
  "name": "main.jir",
  "kind": "generic",
  "jid": "urn:uuid:817b4ff4-e6b7-4296-b383-55515e1e8b4a",
  "j_timestamp": "2022-08-04T20:23:16.952641",
  "j_type": "sentinel"
}
```

The `jid` field is the ID for the sentinel. (`jid` stands for jaseci ID).

With a sentinel and graph, we can now run walker with

```
jaseci > walker run talk -ctx {\"question\": \"I want to schedule a test drive\"}
```

And with `yield`, the next walker run will pick up where it leaves off and retain its variable states and nodes traversal plan.

14.6.3 Tests

Just like any program, a set of automatic tests cases with robust coverage is essential to the success of the program through development to production. Jac has built-in tests support and here is how you create a test case in jac.

```
import {*} with "tesla_ai.jac";

test "testing the Tesla conv AI system"
with graph::tesla_ai by walker::talk(question="Hey I would like to go on a test drive"){
    res = std.get_report();
    assert(res[-1] == "To set you up with a test drive, we will need your name and address.");
}
```

Let's break this down. * `test "testing the tesla conv AI system"` names the test. * `with graph::tesla_ai` specify the graph to be used as the text fixture. * `by walker::talk` specify the walker to test. It will be spawned on the anchor node of the graph. * `std.get_report()` let you access the report content of the walker so that you can set up any assertion neccessary with `assert`.

To run jac tests, save the test case(s) in a file (say `tests.jac`) and import the neccessary walkers and graphs. Then run

```
jaseci > jac test tests.jac
```

This will execute all the test cases in `tests.jac` squentially and report success or any assertion failures.

14.6.4 Running Jaseci as a Service

So far, we have been interacting jaseci through `jsctl`. jaseci can also be run as a service serving a set of RESTful API endpoints. This is useful in production settings. To run jaseci as a service, first we need to install the `jaseci_serv` package.

```
pip install jaseci_serv
```

Then launching a jaseci server is as simple as

```
jsserv makemigrations
jsserv migrate
jsserv runserver 0.0.0.0:3000
```

This will launch a Django RESTful API server at localhost and port 3000. The Jaseci server supports a wide range of API endpoints. All the `jsctl` commands we have used throughput this tutorial have an equivalent API endpoint, such as `walker_run` and `sentinel_register`. As a matter of fact, the entire development journey in this tutorial can be done completely with a remote jaseci server instance. You can go to `localhost:3000/docs` to check out all the available APIs.

14.7 Improve Your AI Models with Crowdsource

Coming soon!

Chapter 15

A Coding Tour

Contents

15.1 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!

15.1.1 Jac Basics

Launch VSCode, spool up a terminal window, and lets tinker with an example. We'll start with Jac Code 15.1. 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 15.1: Example program introducing basic syntax.

```
walker init {
    x = 34 - 30; # This is a comment
    y = "Hello";
    z = 3.45;

    if(z==3.45 or y=="Bye"){ # if statement with only thing true
        x=x-1;
        y=y+" World"; # the + between two strings concatinate them
    }

    std.out(x);
    for i=0 to i<3 by i+=1: # For loop with single line block style
        std.out(x-i,'-', y); # prints to screen
    report [x, y+'s']; # adds data to payload
}
```

This first example Jac Code 15.1 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"
    ]
  ]
}
```

15.1.2 Types in Jac

[Types example]

Jac Code 15.2: 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}}
```

15.1.3 Fun with Lists and Dictionaries

[Fun with Lists and Dictionaries]

Jac Code 15.3: 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]
```

15.1.4 Control Flow

[Fun with Control Flow]

Jac Code 15.4: 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
```

15.1.5 Graphs in Jac

[Bringing Graphs in with special operators]

Jac Code 15.5: First Example

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

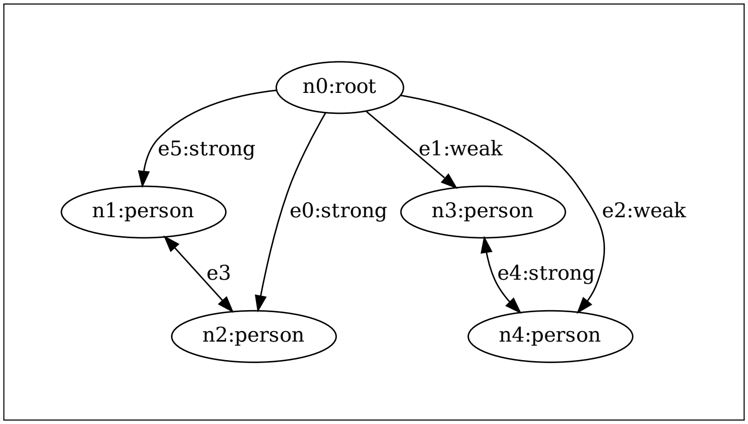


Figure 15.1: Graph in memory for JC 15.5

```

edge strong;
edge weak;

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"}
```

15.1.6 Navigating Graphs with Walkers

[Walking Graphs]

Jac Code 15.6: 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 {
```

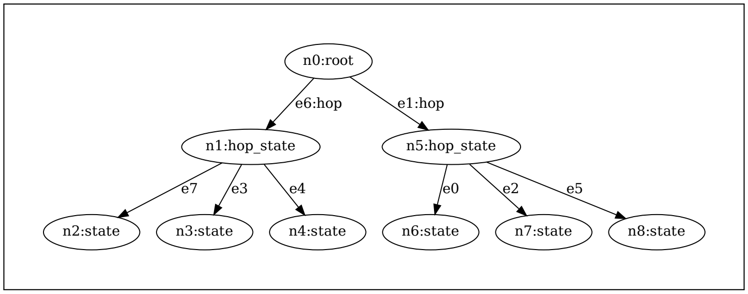


Figure 15.2: Graph in memory for JC 15.6

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

Output:

```
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
  ]
}
```

15.1.7 Compute in Nodes

[Compute into the Nodes]

Jac Code 15.7: 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:

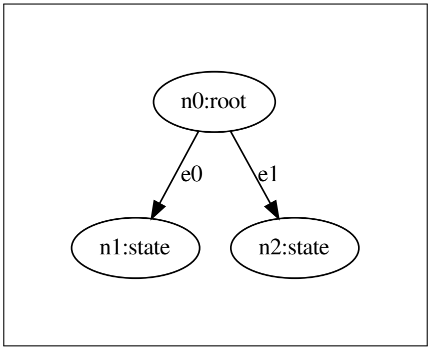


Figure 15.3: Graph in memory for JC 15.7

```
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"]
```

15.1.8 Static Graphs

[Static graphs]

Jac Code 15.8: First Example

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

edge strong;
edge weak;

graph basic_gph {
    has anchor root;
```

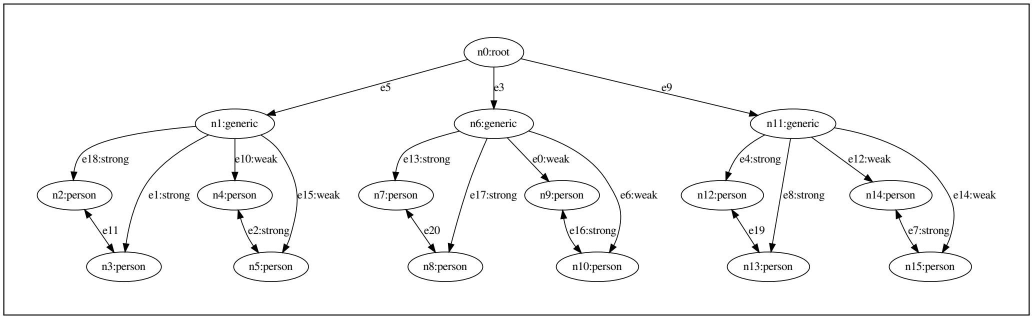


Figure 15.4: Graph in memory for JC 15.8

```
    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;
}
```

15.1.9 Writing Tests

[Tests]

Jac Code 15.9: 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
}
```

15.2 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.

15.2.1 Using Imports

Jac Code 15.10: 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 15.11: 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 15.12: edges for CAI

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

15.2.2 Leveraging Static Graphs for Quick Prototyping

Jac Code 15.13: 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");
    }

}
```

15.2.3 Test Driven Development

Jac Code 15.14: 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);
}
```

15.2.4 File I/O

Jac Code 15.15: 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.
```

15.2.5 Building to JIR

15.3 AI with Jaseci Kit

15.3.1 Installing Jaseci Kit

```
haxor@linux:~$ pip install jaseci-ai-kit
Collecting jaseci-ai-kit
  Downloading jaseci_ai_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-ai-kit-1.3.3.5 ...
haxor@linux:~$
```

15.3.2 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_ai_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 >
```

15.3.3 Using AI in Jac

[Adding some AI]

Jac Code 15.16: 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
    ]
  ]
}
```

15.4 Launching a Jaseci Web Server

15.5 Deploying Jaseci at Scale

15.5.1 Quick-start with Kubectl

15.5.2 Managing Jac in Cloud

Epilogue

Appendix A

Rants

A.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!"

Appendix B

Full Jac Grammar Specification

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

```
grammar jac;

start: ver_label? element+ EOF;

element: architype | walker;

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

walker:
	KW_WALKER NAME 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] HomeLendingPal. Intelligent mortgage advisor - home lending pal. `https://www.homelendingpal.com/`, 2022. [Online; accessed 10-May-2022].

[7] JaseciLabs. Jaseci home. `https://jaseci.org/`, 2022. [Online; accessed 10-May-2022].

[8] JaseciLabs. The official jaseci code repository. `https://github.com/Jaseci-Labs/jaseci`, 2022. [Online; accessed 10-May-2022].

[9] JaseciLabs. Profile of jaseclabs pypi. `https://pypi.org/user/jasecilabs/`, 2022. [Online; accessed 10-May-2022].

[10] myca. Myca.ai, growth via reflection. `https://myca.ai/`, 2022. [Online; accessed 10-May-2022].

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

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

[13] TrueSelph. Unleash your true selph, trueselph. `https://trueselph.com/`, 2022. [Online; accessed 10-May-2022].

[14] ZeroShotBot. Next-gen ai, zeroshotbot. `https://zeroshotbot.com/`, 2022. [Online; accessed 10-May-2022].