JASECI & JAC

BIBLE

Jason Mars

v1.0

Warning: This book is a trigger factory.

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

Contents

Not So Technical Terms Used

bleh mildly yucky. 30

christen to name or dedicate (something, such as a piece of code) by a ceremony that often involves breaking a bottle of champagne. 28

common languages typical languages programmers use to write commercial software, (e.g., C, C++, Java, Javascript, Python, Ruby, Go, Perl, PHP, etc.). 17

dope sick. 5

goo goo gaa gaa the language of babies. 29

grok to fully comprehend and understand deeply . 27

pwn the act of dominating a person, place, or thing. (...or a piece of code). 44

redonkulous dope. 5

scat the excrement of an animal including but not limited to human; also heroin . 9

sick redonkulous. 5, 27

Technical Terms Used

contexts A set of key value pairings that serve as a data payload attributable to nodes and edges in Jaseci graphs. 19

directed graphs . 18

hypergraph . 18

multigraph . 18

undirected graphs . 18

Preface

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

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

Everything here is based on my opinion. . . no, *expert* opinion, and my intution. That suffices for me, and I hope it does for you. Even though I have spent many decades coding and leading coders working on the holy grail technical challenges of our time, I won't rely on that to assert my credibility. Let these ideas stand or die on thier own merit. Its my gut that tells me that we can do better. This book describes my attempt at better. I hope you find value in it. If you do, awesome! If you don't, awesome!

Chapter 1

Introduction

Part I

The World of Jaseci

Chapter 2

What and Why is Jaseci?

2.1 Viewing the Problem Landscape Spacially

2.2 Compute via The Collective, The Worker Bee Model

Chapter 3

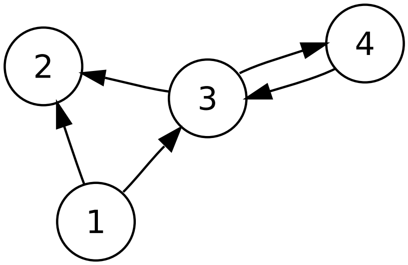
Abstrations of Jaseci

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

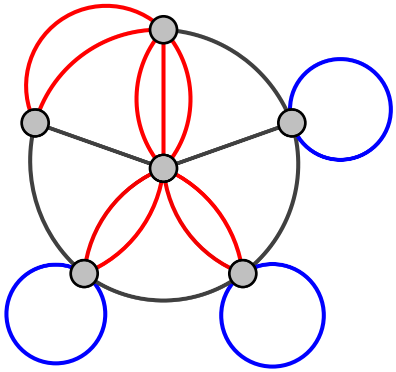
There's something quite strange that has happend with our common languages over the years, ...decades. When you look at it, almost every data structure we programmers use to solve problems can be modeled formally as a graph, or a special case of a graph, (save perhaps hash tables). Think about it, stacks, lists, queues, trees, heaps, and yes, even graphs, can be modeled with graphs. But, low and behold, no common language ustilizes the formal semantics of a graph as its first order abstraction for data or memory. I mean, isn't it a bit odd that practically every data structure covered in the language-agnostic classic foundational work *Introduction to Algorithms* [4] can most naturally be be reasoned about as a graph, yet none of the common languages have built in and be designed around this primitive. I submit that the graph semantic is stupidly rich, very nice for us humans to reason about, and, most importantly for the purpose of Jaseci, is inherently well suited for the conceptualization and reasoning about computational problems, especially AI problems.

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

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



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



(b) Multigraph with parallel edges and self-loops

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

For the former of these two challenges, I counter with two points. First, the core design languages are always based upon their inherent abstractions. With graphs not being one such abstraction, the language's design will not be optimized to empower programmers to nimbly do gymnastics with the rich language symantics that correspond to the rich semantics graphs offer (You'll see what I mean in later chapters). And second, libraries suck (See A.1).

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

3.1.1 Yes, But What Kind of Graphs

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

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

In Jaseci, we elect to assume the following semantics:

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

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

Nerd Alert 1 ***(time to let your eyes glaze over)***

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

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

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

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

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

Worry not friend, lets discuss.

3.1.2 Putting it All Into Context

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

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

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

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

3.2 Walkers

3.3 Abilities

3.4 Other Abstractions Not Yet Actualized

Chapter 4

Architecture of Jaseci and Jac

4.1 Anatomy of a Jaseci Application

4.2 The Jaseci Machine

4.2.1 Machine Core

4.2.2 Jaseci Cloud Server

Chapter 5

Interfacing a Jaseci Machine

5.1 JSCTL: The Jaseci Command Line Interface

5.2 Jaseci Rest API

Part II

Jaseci Jac'd Up

Chapter 6

The Jac Programming Language

To articulate the sourcer spells made possible by the wand that is Jaseci, I bestow upon thee, the Jac programming language. (Like the Harry Potter [5] 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 extention `.jac` for Jac programs.
- It pulls its letters from the phrase **JA**seci **C**ode.
- And it sounds oh so sweet to say "Did you grok that sick Jac code yet!" Rolls right off the tongue.

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

6.1 Getting the Basics Out of the Way

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

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

6.1.1 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" becaming a meme both technically and socially. We have such love for this contribution we even tag or newborns with the phrase as per Fig. 6.1. I digress. Lets now christen our baby, Jaseci, with its "Hello World" expression.

Jac Code 6.1: Jaseci says Hello!

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

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

```
Hello World
```

[1] Image credit to wiki contributer [1]

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

6.1.2 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 experisions 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 langauge and variables can be delcared via a direct assignment; also very Python’y. The comma separated list of the defined variables `a - e` in the call to `std.out` illustrate multiple values being printed to screen from a single call.

Additionally, Jac supports power and modulo operations.

Jac Code 6.3: Additional arithmetic operations

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

Jac Code 6.3 outputs,

```
256 4
```

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

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

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

Grammar 6.4: Jac grammar clip relevant to arithmetic

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

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

factor: (PLUS | MINUS) factor | power;

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

(full grammar in Appendix B)

6.1.3 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 appearnce, 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 appearnce 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 4 ***(time to let your eyes glaze over)***

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

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

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

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

nin: NOT KW_IN;
```

(full grammar in Appendix B)

6.1.4 Assignment Operations

Next, lets take a look at assingment 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 assinment. Of course we can simply set a variable equal to a particular value, however, we can go beyond that to set that assinment 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 curiousity, 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 5 ***(time to let your eyes glaze over)***

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

Grammar 6.9: Jac grammar clip relevant to assignment

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

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

copy_assign: dotted_name index* CPY_EQ expression;
```

(full grammar in Appendix B)

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

Table 6.1: Precedence of operations in Jac

6.1.5 Precedence in Jac

At this point in our discussion its important to note the precedence of opperations 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.

6.1.6 Foreshadowing Unique Graph Operations

Jac Code 6.10: Preview of graph operators

```
node simple;
edge back;

walker init {
    node_a = spawn here --> node::simple;
    here <-[back]- node_a;

    node_b = spawn here <--> node::simple;
    node_b --> node_a;
}
```

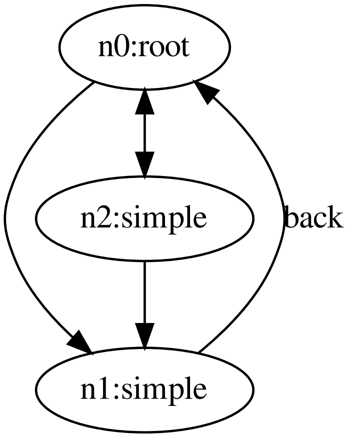


Figure 6.2: A preview of whats to come in Jac Land!

Part III

Crafting Jaseci

Chapter 7

Architecting Jaseci Core

Chapter 8

Architecting Jaseci Cloud Serving

Epilogue

Appendix A

Rants

A.1 Libraries Suck

Because they do.

Still need more reasons?

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

. . .

Still there?

. . .

Fine, I'll tell you.

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

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

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

This whitespace debautchery perpetrated by Python and the like is one of the most perverted abuses of ASCII code 32 I've seen in computer science. It's an assult 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;

/* Sentinels handle these top rules */
start: 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 LBRACE attr_stmt* walk_entry_block? (
		statement
		| walk_activity_block
	)* walk_exit_block? RBRACE;

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;

/* Need to be heavily simplified */ can_stmt:
        KW_CAN dotted_name preset_in_out? event_clause? (
                COMMA dotted_name preset_in_out? event_clause?
        )* SEMI
        | KW_CAN NAME event_clause? code_block;

event_clause: KW_WITH (KW_ENTRY | KW_EXIT | KW_ACTIVITY);

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

dotted_name: NAME (DOT NAME)*;

code_block: LBRACE statement* RBRACE | COLON statement;

node_ctx_block: NAME (COMMA NAME)* code_block;

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

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

elif_stmt: KW_ELIF expression code_block;

else_stmt: KW_ELSE code_block;

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

while_stmt: KW_WHILE expression code_block;

ctrl_stmt: KW_CONTINUE | KW_BREAK | KW_SKIP;
```

```

report_action: KW_REPORT expression SEMI;

walker_action:
       ignore_action
       | take_action
       | destroy_action
       | KW_DISENGAGE SEMI;

ignore_action: KW_IGNORE expression SEMI;

take_action: KW_TAKE expression (SEMI | else_stmt);

destroy_action: KW_DESTROY expression SEMI;

expression: assignment | connect;

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

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

copy_assign: dotted_name index* CPY_EQ expression;

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

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

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

nin: NOT KW_IN;

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

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

factor: (PLUS | MINUS) factor | power;

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

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

func_built_in:
       | KW_LENGTH
       | KW_KEYS
```

```
        | KW_EDGE
        | KW_NODE
        | KW_DESTROY LPAREN expression RPAREN;

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

node_edge_ref: node_ref | edge_ref (node_ref)?;

node_ref: KW_NODE DBL_COLON NAME;

walker_ref: KW_WALKER DBL_COLON NAME;

graph_ref: KW_GRAPH DBL_COLON NAME;

edge_ref: edge_to | edge_from | edge_any;

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

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

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

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

index: LSQUARE expression RSQUARE;

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

kv_pair: STRING COLON expression;

spawn: KW_SPAWN expression? spawn_object;

spawn_object: node_spawn | walker_spawn | graph_spawn;

node_spawn: edge_ref? node_ref spawn_ctx?;

graph_spawn: edge_ref graph_ref;

walker_spawn: walker_ref spawn_ctx?;

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

/* DOT grammar below */
dot_graph:
```

```
        KW_STRICT? (KW_GRAPH | KW_DIGRAPH) dot_id? '{' dot_stmt_list '}';

dot_stmt_list: ( dot_stmt ';'?)*;

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

dot_attr_stmt: ( KW_GRAPH | KW_NODE | KW_EDGE) dot_attr_list;

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

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

dot_edge_stmt: (dot_node_id | dot_subgraph) dot_edgeRHS dot_attr_list?;

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

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

dot_node_stmt: dot_node_id dot_attr_list?;

dot_node_id: dot_id dot_port?;

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

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

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

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

```
COLON: ':';
DBL_COLON: '::';
COLON_OUT: '::>';
LBRACE: '{';
RBRACE: '}';
KW_EDGE: 'edge';
KW_WALKER: 'walker';
SEMI: ';';
EQ: '=';
PEQ: '+=';
MEQ: '-=';
TEQ: '*=';
DEQ: '/=';
CPY_EQ: ':=';
KW_AND: 'and' | '&&';
KW_OR: 'or' | '||';
KW_IF: 'if';
KW_ELIF: 'elif';
KW_ELSE: 'else';
KW_FOR: 'for';
KW_TO: 'to';
KW_BY: 'by';
KW_WHILE: 'while';
KW_CONTINUE: 'continue';
KW_BREAK: 'break';
KW_DISENGAGE: 'disengage';
KW_SKIP: 'skip';
KW_REPORT: 'report';
KW_DESTROY: 'destroy';
DEREF: '&';
DOT: '.';
NOT: '!' | 'not';
EE: '==';
LT: '<';
GT: '>';
LTE: '<=';
GTE: '>=';
NE: '!=';
KW_IN: 'in';
KW_ANCHOR: 'anchor';
KW_HAS: 'has';
KW_PRIVATE: 'private';
COMMA: ',';
KW_CAN: 'can';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
POW: '^';
LPAREN: '(';
RPAREN: ')';
LSQUARE: '[';
RSQUARE: ']';
FLOAT: ([0-9]+)? '.' [0-9]+;
```

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

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