The Jaseci Machine & Jac Language

Processing Architecture and Programming Language for High Complexity AI

Dr. Jason Mars

PUBLISHED BY PUBLISHER

JASONMARS.ORG

First printing, September 2020

Contents

1. Introduction

Jaseci Machine

2. Contexts and Actions

2.1 Contexts

In Jaseci, a *context element* represents a data element with a corresponding unique identifier of that data element. This abstraction is analogous to the traditional view of addressible memory in a computer system [CITE] for which which each word (data value) is addressible with a 32-bit or 64-bit memory address (identifier). However, a context element is untyped and in principle unbounded in size. Context have no requirements or restrictions for the kind or type of data element that is stored and only requires the identifier to be unique.

Though contexts are sufficent to enable all representations and management of data in the Jaseci machine, we also introduce the abstraction of *context sets* that represents an explicit set of contexts bound to a single identifier. This *context set* abstraction provides the utility of grouping, organizing, and handeling collections of related contexts as a single unit.

R

> The practical implmenetation of a Jaseci machine described in this book uses URN UUIDs [CITE] for the identifer.

2.1.1 Formal Definition of Contexts and Context Sets

Definitions 2.1.1 and 2.1.2 formally describe contexts and context sets.

Definition 2.1.1 — context. A *context* is a representation of data that can be expressed as a 2-tuple (k, v), where

1. k is a unique identifier of the context
2. v is an encoding of the data represented by the context

Definition 2.1.2 — context set. A *context set* is a representation of data that can be expressed as a 2-tuple (k, C), where
1. k is a unique identifier of the context set
2. C is an explicit finite set of contexts and context sets

2.1.2 Example

■ **Example 2.1** For each trip a shopper has made to a grocery store, suppose we'd like to use contexts to represent what the shopper bought and the most expensive item. Let the set $I = \{item_1, item_2, item_3, \ldots, item_n\}$ be the set of all distinct items in a given store and $P = \{item | item \in I \text{ and was paid for by shopper}\}$. We construct the context and context set

$$A = (k, item_i \,|\, item_i \text{ is the first } item \in I \wedge P \text{ sorted by cost.}) \quad (2.1)$$
$$B = (k, I \wedge P) \quad (2.2)$$

■

3. Jaseci Graphs, Nodes and Edges

3.1 Graphs

3.1.1 Formal Definition

Name Graph

> **Definition 3.1.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

3.2 Nodes

3.2.1 Formal Definition

Name Node

> **Definition 3.2.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

3.3 Edges

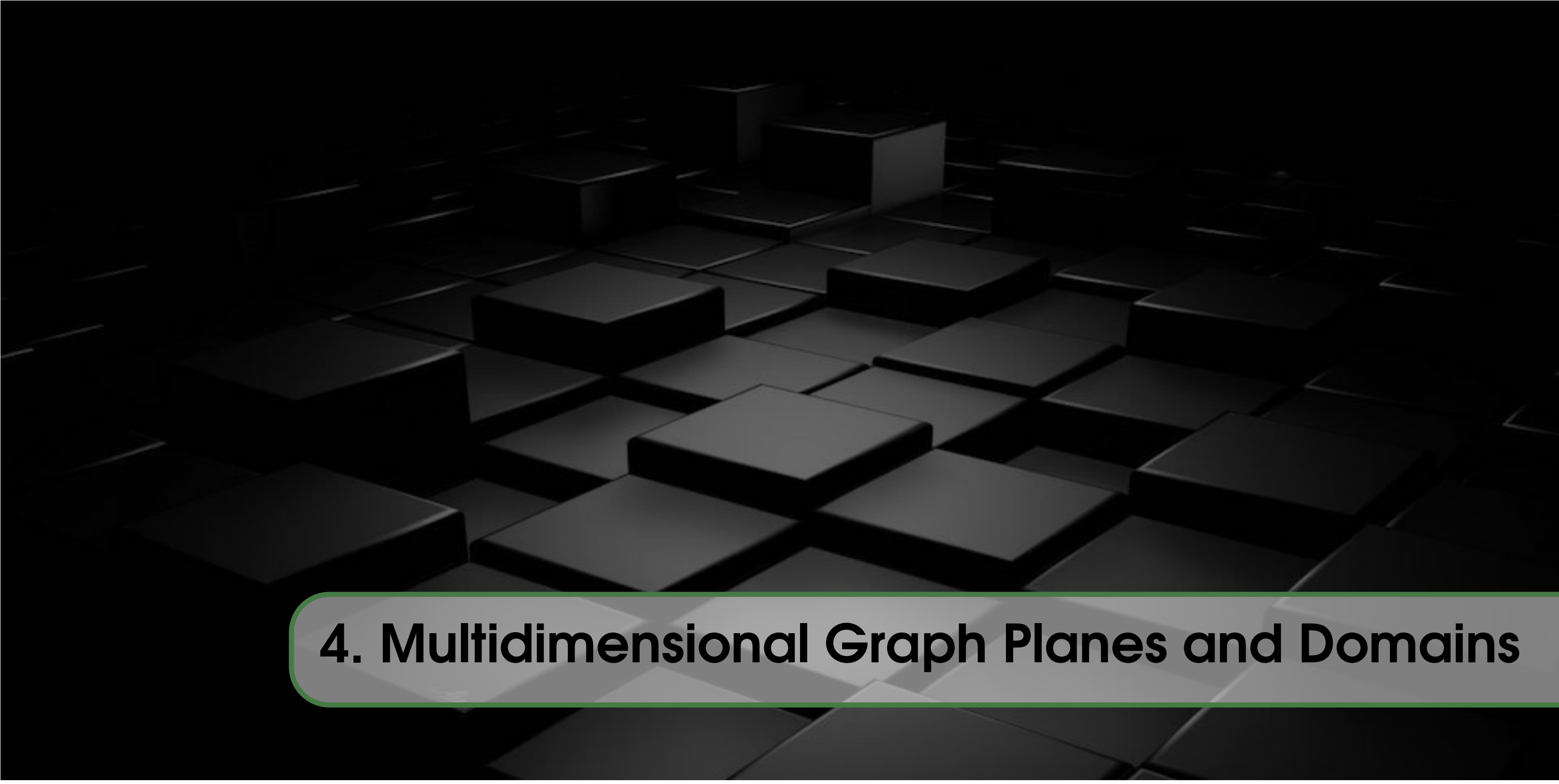
3.3.1 Formal Definition

Name Edge

Definition 3.3.1 — action. A is a representation of data that can be expressed as a 2-tuple (k, v), where

1. k is a unique identifier of the context
2. v is an encoding of the data represented by the context

Description Go deeper



4. Multidimensional Graph Planes and Domains

4.1 Graph Planes

4.1.1 Formal Definition

Name Graph plane

> **Definition 4.1.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

4.2 Domain Nodes

4.2.1 Formal Definition

Name Domain node

> **Definition 4.2.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

5. Walkers, Architypes and Sentinels

5.1 Architypes

5.1.1 Formal Definition

Name Graph

> **Definition 5.1.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

5.2 Walker

5.2.1 Formal Definition

Name Graph

> **Definition 5.2.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

5.3 Sentinels

5.3.1 Formal Definition

Name Graph

Definition 5.3.1 — action. A is a representation of data that can be expressed as a 2-tuple (k, v), where

1. k is a unique identifier of the context
2. v is an encoding of the data represented by the context

Description Go deeper

6. A Jaseci Machine

6.1 A Jaseci Machine

6.1.1 Formal Definition

Name Graph

> **Definition 6.1.1 — action.** A is a representation of data that can be expressed as a 2-tuple (k, v), where
> 1. k is a unique identifier of the context
> 2. v is an encoding of the data represented by the context

Description Go deeper

II Jac Language

7. Language Overview and Basics

Jac is dynamically typed.

Jac is pass by reference.

Jac is whitespace agnostic.

Jac does not have traditional notion of functions but walkers instead. Deliberate to encourage thought about problems in a spacial graph paradigmn that is equivilant in power but more natural for many complex AI problems.

Jac Code 7.1: Hello World

```
walker init {
    std.log('Hello World');
}
```

Listing 7.1 shows the canonical 'hello world' program as it would be most simply implemented in Jac.

Jac also has the notion of single statement line code blocks.

Jac Code 7.2: Hello World

```
walker init: std.log('Hello World');
```

Program 7.1 is equivilant to Program 7.2.

7.1 Numbers and Arithmetic

Basic Arithmetic Operations

The simplest math operations in Jac.

Code

Jac Code 7.3: 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);
}
```

Output

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

Description

Additionally, Jac supports power and modulo operations.

Jac Code 7.4: Additional arithmetic operations

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

Output

```
256 4
```

Description

Comparison Operations

Jac Code 7.5: 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);
}
```

Output

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

Description

Logical Operations

Jac Code 7.6: 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));
}
```

Output

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

Description

Assignment Operations

Jac Code 7.7: 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
}
```

Output

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

Description

Foreshadowing Unique Graph Operations

Jac Code 7.8: Preview of graph operators

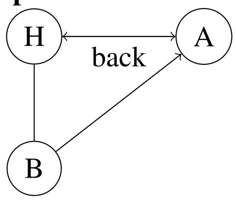
```
edge back;

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

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

Output

Description



Precedence

Rank	Symbol	Description
1	() [] . :: –> <– spawn	Parenthetical/grouping, node/edge manipulation
2	^	Exponent
3	* / %	Multiplication, division, modulo
4	+ -	Addition, subtraction
5	== != >= <= > <	Comparison
6	&& || and or	Logical
7	= += -= *= /= :=	Assignment

Table 7.1: Precedence of operations in Jac

7.2 Strings and List

Coming soon.

7.3 Control Flow

Conditional Statements

Jac Code 7.9: if statements

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

Output

```
Hello!
```

Description

Jac Code 7.10: 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");
}
```

Output

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

Description

Jac Code 7.11: 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");
}
```

Output

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

Description

Loops

Jac Code 7.12: for loops

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

Output

```
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!
```

Description

Jac Code 7.13: for loops iterating through lists

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

Output

```
TEST CASE NOT GENERATED YET
```

Description

R

Remember, though this looks very much like python, the : operator here indicates single line block. Braces should be used for multiline code blocks, e.g.,

```
for i in my_list {
    if(i == 'jon'): i = 5;
    std.out("Hello", i, "times!");
}
```

Jac Code 7.14: while loops

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

Output

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

Description

Loop Control Statements

Jac Code 7.15: break statement

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

Output

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

Description

Jac Code 7.16: continue statement

```
walker init {
    i = 5;
    while(i>0) {
        if(i == 3){
            i -= 1; continue;
        }
        std.out("Hello", i, "times!");
        i -= 1;
    }
}
```

Output

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

Description

8. Archetypes and Walkers

Define Architiype

Define Walkers

Describe their interaction and roles

8.1 Walkers and Graphs as First Order Citizens

Define and describe walker in Jac

Introduce graphs, nodes, edges

Jac Code 8.1: Simple walker creating and connected nodes

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

Output

```
```

Description

Jac Code 8.2: Creating named node types

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

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;
}
```

Output

Description

Jac Code 8.3: Connecting nodes within spawn statement

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

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

Output

Description

Jac Code 8.4: Chaining node connections using the connect operator

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

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

Output

Description

Jac Code 8.5: Walkers spawning other walkers

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

walker family_ties {
    for i in -[family]->:
        std.out(here, ' is related to ', i);
```

```
}

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

Output

```
```

Description

```
Jac Code 8.6:  Getting returned values from spawned walkers
node person;
edge assists;
edge family;

walker family_ties {
    has anchor fam_nodes;
    fam_nodes = -[family]->:
}

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

Output

```
```

Description

R

Remember spawn statements are expressions so they can be used as such, e.g.,

```
for i in spawn here walker::family_ties:
    std.out(here, ' is related to ', i);
```

8.2 Architypes and Actions

Nodes, and edges

Spawning nodes

Binding compute to architypes

```
Jac Code 8.7:  Binding member contexts to nodes and edges
```

```
node person {
    has name;
    has age;
    has birthday, profession;
}

edge friend: has meeting_place;
edge family: has type;

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;
    -[friend]->[0].meeting_place = "college";
    -[family]->[0].type = "sister"
    std.out(--> node);
}
```

Output

Description

Jac Code 8.8: Binding contexts with less code

```
node person {
    has name;
    has age;
    has birthday, profession;
}

edge friend: has meeting_place;
edge family: has type;

walker init {
    person1 = spawn here -[friend(meeting_place="college")]->
        node::person(name="Josh");
    person2 = spawn here -[family(type="sister")]->
        node::person(name="Jane");
    std.out(--> node);
}
```

Output

Description

Jac Code 8.9: Adding actions to architypes

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

walker init {
    person1 = spawn here -->
```

```
        node::person(name="Josh", birthday="1995-05-20");
    birthyear = date.quantize_to_year(person1.birthday);
    std.out(birthyear);
}
```

Output

```
```

Description

Jac Code 8.10: Triggering actions on entry and exit

```
node person {
    has name;
    has bday, byear;
    can date.quantize_to_year::bday::>byear with entry;
    can std.out::byear," from ",bday:: with exit;
}

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

Output

```
```

Description

R

The `node` definition in Jac Code 8.9 is equivalent to

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

The `with activity` keywords indicates the action will be called by walkers.

9. Navigating Graphs

Jac Code 9.1: Walking graphs by taking edges

```
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;
    }
}
```

Output

```
```

Description

Jac Code 9.2: Fan out style walks

```
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);
    }
}
```

Output

Description

Jac Code 9.3: Ignoring paths on a 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]->;
        take -->;
    }
    person {
        std.out(here.name);
    }
}
```

Output

Description

Jac Code 9.4: Destroying (deleting) nodes

```
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);
    }
}
```

Output

Description

Jac Code 9.5: Generating reports throughout a 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 -->;
    }
}
```

Output

Description

10. Putting It All Together: LifeLogify

Jac Code 10.1: LifeLogify's Archetypes

```
node life {
    has anchor owner;
    can infer.year_from_date;
}

node year {
    has anchor year;
    can infer.month_from_date;
}

node month {
    has anchor month;
    can infer.year_from_date;
    can infer.week_from_date;
}

node week {
    has anchor week;
    can infer.month_from_date;
    can infer.day_from_date;
}

node day {
    has anchor day;
    can infer.day_from_date;
}

node workette {
    has name, order, date, owner, status, snooze_till;
```

```
    has note, is_MIT, is_ritual;
}

edge past;

edge parent;
```

Jac Code 10.2: Finding the most proximal day in graph

```
walker get_latest_day {
    has before_date;
    has anchor latest_day;
    if(!before_date): before_date = std.time_now();
    if(!latest_day): latest_day = 0;
    root: take --> node::life;
    life {
        ignore --> node::year > infer.year_from_date(before_date);
        take net.max(--> node::year);
    }
    year {
        ignore node::month > infer.month_from_date(before_date);
        take net.max(--> node::month)
        else {
            ignore here;
            take <-- node::life;
        }
    }
    month {
        ignore node::week > infer.week_from_date(before_date);
        take net.max(--> node::week)
        else {
            ignore here;
            take <-- node::year ==
                infer.year_from_date(before_date);
        }
    }
    week {
        ignore node::day > infer.day_from_date(before_date);
        take net.max(--> node::day)
        else {
            ignore here;
            take <-- node::month ==
                infer.month_from_date(before_date);
        }
    }
    day {
        latest_day = here;
        report here;
    }
}
```

Jac Code 10.3: Get day if present otherwise create day

```
walker get_gen_day {
    has date;
    has anchor day_node;
```

```
    if(!date): date=std.time_now();
    root: take --> node::life;
    life: take --> node::year == infer.year_from_date(date) else {
            new = spawn here --> node::year ;
            new.year = infer.year_from_date(date);
            take --> node::year == infer.year_from_date(date);
        }
    year: take --> node::month == infer.month_from_date(date) else {
            new = spawn here --> node::month;
            new.month = infer.month_from_date(date);
            take --> node::month == infer.month_from_date(date);
        }
    month: take --> node::week == infer.week_from_date(date) else {
            new = spawn here --> node::week;
            new.week = infer.week_from_date(date);
            take --> node::week == infer.week_from_date(date);
        }
    week: take --> node::day == infer.day_from_date(date) else {
            latest_day = spawn here walker::get_latest_day;
            new = spawn here --> node::day;
            new.day = infer.day_from_date(date);
            if(latest_day and infer.day_from_date(date) ==
                infer.day_from_date(std.time_now())) {
                spawn latest_day walker::carry_forward(parent=new);
                take new;
            }
            elif(latest_day) {
                take latest_day;
            }
            else: take new;
        }
    day {
        day_node = here;
        report here;
    }
}
```

Jac Code 10.4: Get child workettes

```
walker get_workettes {
    day, workette {
        for i in  --> node::workette:
            report i;
    }
}
```

Jac Code 10.5: Delete a workette and it's children

```
walker delete_workette {
    workette {
        take --> node::workette;
        destroy here;
    }
}
```

Jac Code 10.6: Create a child workette

```
walker create_workette {
    day, workette {
        new = spawn here -[parent]-> node::workette;
        report new;
    }
}
```

Jac Code 10.7: Get workette and all derivitive workettes

```
walker get_workettes_deep {
    day {
        take --> node::workette;
    }
    workette {
        report here;
        take --> node::workette;
    }
}
```

Jac Code 10.8: Automatically copy and link prior day's workettes

```
walker carry_forward {
    has parent;
    day {
        take --> node::workette;
    }
    workette {
        if(here.status == 'done' or
        here.status == 'eliminated') {
            disengage;
        }
        new_workette = spawn here <-[past]- node::workette;
        new_workette <-[parent]- parent;
        new_workette := here;
        spawn --> node::workette
            walker::carry_forward(parent=new_workette);
    }
}
```

Jac Code 10.9: Generate random data for 2019

```
walker gen_rand_life {
    has num_workettes;
    root: take --> node::life;

    life {
        num_workettes = 5;
        num_days = rand.integer(2, 4);
        for i=0 to i<num_days by i+=1 {
            spawn here walker::get_gen_day(
                date=rand.time("2019-01-01", "2019-12-31")
            );
        }
        take -->;
```

```
    }
    year, month, week { take -->; }
    day {
        if(here.day == infer.day_from_date(std.time_now())): skip;
    }
    day, workette {
        if(num_workettes == 0): disengage;
        gen_num = rand.integer(3, 5);
        for i=0 to i<gen_num by i+=1 {
            spawn here -[parent]->
                node::workette(name=rand.sentence());
        }
        take --> ;
        num_workettes -= 1;
    }
}
```

Jac Code 10.10: Connect a life node to root

```
walker init {
    has owner;
    has anchor life_node;
    take (--> node::life == owner) else {
        life_node = spawn here --> node::life;
        life_node.owner = owner;
        disengage;
    }
}
```

11. Standard Library of Actions

12. Language Grammar and Specification

Grammar Rules `start, element`

```
grammar jac;

start: element*;

element: architype | walker;
```

Example

```
node person;
edge friend;

walker friend_network {}
```

Description Every program begins with a number of architypes followed by walkers.

Grammar Rule `architype`

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

Example

```
```

Description

Grammar Rule `architype`

```
walker: KW_WALKER NAME LBRACE (attr_stmt)* statement* RBRACE;
```

Example

```
```

Description

Grammar Rule `architype`

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

Example

Description

Grammar Rule `architype`

```
attr_stmt: has_stmt | can_stmt;
```

Example

Description

Grammar Rule `architype`

```
has_stmt: KW_HAS KW_ANCHOR? NAME (COMMA NAME)* SEMI;
```

Example

Description

Grammar Rule `architype`

```
can_stmt:
    KW_CAN dotted_name preset_in_out? (KW_WITH KW_MOVE)? (
        COMMA dotted_name preset_in_out? (KW_WITH KW_MOVE)?
    )* SEMI;
```

Example

Description

Grammar Rule `architype`

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

Example

Description

Grammar Rule `architype`

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

Example

Description

Grammar Rule `architype`

```
node_ctx_block: NAME (COMMA NAME)* code_block;
```

Example

```
```

Description

Grammar Rule `architype`

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

Example

```
```

Description

Grammar Rule `architype`

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

Example

```
```

Description

Grammar Rule `architype`

```
elif_stmt: KW_ELIF expression code_block;
```

Example

```
```

Description

Grammar Rule `architype`

```
else_stmt: KW_ELSE code_block;
```

Example

```
```

Description

Grammar Rule `architype`

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

Example

```
```

Description

Grammar Rule `architype`

```
while_stmt: KW_WHILE expression code_block;
```

Example

Description

Grammar Rule archetype

```
ctrl_stmt: KW_CONTINUE | KW_BREAK | KW_DISENGAGE | KW_SKIP;
```

Example

Description

Grammar Rule archetype

```
action_stmt:
    ignore_action
    | take_action
    | report_action
    | destroy_action;
```

Example

Description

Grammar Rule archetype

```
ignore_action: KW_IGNORE expression SEMI;
```

Example

Description

Grammar Rule archetype

```
take_action: KW_TAKE expression (SEMI | else_stmt);
```

Example

Description

Grammar Rule archetype

```
report_action: KW_REPORT expression SEMI;
```

Example

Description

Grammar Rule archetype

```
destroy_action: KW_DESTROY expression SEMI;
```

Example

Description

Grammar Rule archetype

```
expression: assignment | connect;
```

Example

Description

Grammar Rule archetype

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

Example

Description

Grammar Rule archetype

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

Example

Description

Grammar Rule archetype

```
copy_assign: dotted_name CPY_EQ expression;
```

Example

Description

Grammar Rule archetype

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

Example

Description

Grammar Rule archetype

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

Example

Description

Grammar Rule archetype

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

Example

Description

Grammar Rule `architype`

```
nin: NOT KW_IN;
```

Example

```
```

Description

Grammar Rule `architype`

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

Example

```
```

Description

Grammar Rule `architype`

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

Example

```
```

Description

Grammar Rule `architype`

```
factor: (PLUS | MINUS) factor | power;
```

Example

```
```

Description

Grammar Rule `architype`

```
power: func_call (POW factor)*;
```

Example

```
```

Description

Grammar Rule `architype`

```
func_call:
        atom (LPAREN (expression (COMMA expression)*)? RPAREN)?
            ;
```

Example

```
```

Description

Grammar Rule `architype`

```
atom:
        INT
        | FLOAT
        | STRING
        | BOOL
        | array_ref
```

```
         | attr_ref
         | node_ref
         | edge_ref (node_ref)? /* Returns nodes even if edge */
         | list_val
         | dotted_name
         | LPAREN expression RPAREN
         | spawn;
```

Example

```
```

Description
Grammar Rule architype

```
array_ref: dotted_name (LSQUARE expression RSQUARE)+;
```

Example

```
```

Description
Grammar Rule architype

```
attr_ref: dotted_name DBL_COLON dotted_name;
```

Example

```
```

Description
Grammar Rule architype

```
node_ref: KW_NODE (DBL_COLON NAME)?;
```

Example

```
```

Description
Grammar Rule architype

```
edge_ref: edge_to | edge_from | edge_any;
```

Example

```
```

Description
Grammar Rule architype

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

Example

```
```

Description
Grammar Rule architype

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

Example

```
```

Description
Grammar Rule architype

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

Example

```
```

Description
Grammar Rule architype

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

Example

```
```

Description
Grammar Rule architype

```
spawn: KW_SPAWN expression spawn_object;
```

Example

```
```

Description
Grammar Rule architype

```
spawn_object: node_spawn | walker_spawn;
```

Example

```
```

Description
Grammar Rule architype

```
node_spawn: edge_ref node_ref spawn_ctx?;
```

Example

```
```

Description
Grammar Rule architype

```
walker_spawn: KW_WALKER DBL_COLON NAME spawn_ctx?;
```

Example

```
```

Description
Grammar Rule architype

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

Example

```
```

Description

III

API Substrate for Computation

13. Expressing Jaseci Computation through APIs

The primary method for executing computations using and Jaseci is to submit Jac programs to a cloud service that implements the Jaseci machine.

[Benefits of doing it in the cloud, abstraction, scale, elasticity, machine configuration ease, etc]

To realize this cloud based operational and execution model a standard API interface is needed. We define this API using well established REST API principles [1, 2].

14. General Operations

15. Jac Execution API

16. Jaseci Object API

IV Case Studies

17. LifeLogify

18. FPWSM

19. Conversational AI

This statement requires citation [2]; this one is more specific [3, page 162].

V

Appendix

A. Jaseci Shell

B. Jac Language Grammar

Jac Code B.1: Jac Language Grammar

```
grammar jac;

/* Sentinels handle these top rules */
start: element*;

element: architype | walker;

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

walker: KW_WALKER NAME LBRACE (attr_stmt)* statement* RBRACE;

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

attr_stmt: has_stmt | can_stmt;

has_stmt: KW_HAS KW_ANCHOR? NAME (COMMA NAME)* SEMI;

can_stmt:
    KW_CAN dotted_name preset_in_out? (KW_WITH KW_MOVE)? (
        COMMA dotted_name preset_in_out? (KW_WITH KW_MOVE)?
    )* SEMI;

preset_in_out: DBL_COLON NAME (COMMA NAME)* (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
    | action_stmt;

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;

while_stmt: KW_WHILE expression code_block;

ctrl_stmt: KW_CONTINUE | KW_BREAK | KW_DISENGAGE;

action_stmt: ignore_action | take_action | report_action;

ignore_action: KW_IGNORE expression SEMI;

take_action: KW_TAKE expression (SEMI | else_stmt);

report_action: KW_REPORT expression SEMI;

expression: assignment | connect;

assignment:
    dotted_name EQ expression
    | inc_assign
    | copy_assign;

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

copy_assign: dotted_name CPY_EQ expression;

connect: logical (edge_ref expression)?;

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

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

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

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

factor: (PLUS | MINUS) factor | power;

power: func_call (POW factor)*;

func_call:
    atom (LPAREN (expression (COMMA expression)*)? RPAREN)?;

atom:
    INT
    | FLOAT
    | STRING
    | array_ref
    | attr_ref
    | node_ref
    | edge_ref (node_ref)?
    | list_val
    | dotted_name
    | LPAREN expression RPAREN
    | spawn;

array_ref: dotted_name (LSQUARE expression RSQUARE)+;

attr_ref: dotted_name DBL_COLON dotted_name;

node_ref: KW_NODE (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;

spawn: KW_SPAWN expression spawn_object;

spawn_object: node_spawn | walker_spawn;

node_spawn: edge_ref node_ref spawn_ctx?;

walker_spawn: KW_WALKER DBL_COLON NAME spawn_ctx?;

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

Jac Code B.2: Jac Language Lexer Rules

```
/* Lexer rules */
KW_NODE: 'node';
KW_IGNORE: 'ignore';
KW_TAKE: 'take';
```

```
KW_MOVE: 'entry' | 'activity' | 'exit';
KW_SPAWN: 'spawn';
KW_WITH: 'with';
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_REPORT: 'report';
DOT: '.';
NOT: '!';
EE: '==';
LT: '<';
GT: '>';
LTE: '<=';
GTE: '>=';
NE: '!=';
KW_ANCHOR: 'anchor';
KW_HAS: 'has';
COMMA: ',';
KW_CAN: 'can';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
POW: '^';
LPAREN: '(';
RPAREN: ')';
LSQUARE: '[';
RSQUARE: ']';
FLOAT: [0-9]+ '.' [0-9]+;
STRING: '"' ~ ["\r\n]*␣'"' | '\'' ~ ['\r\n]* '\'';
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

Articles

[2] James Smith. "Article title". In: 14.6 (Mar. 2013), pages 1–8 (cited on pages 47, 54).

Books

[1] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. OReilly Media, Inc., 2013. ISBN: 1449358063 (cited on page 47).

[3] John Smith. *Book title*. 1st edition. Volume 3. 2. City: Publisher, Jan. 2012, pages 123–200 (cited on page 54).

Index