# Final Project Report

V. Atlidakis, G. Koloventzos, M. Lecuyer, Y. Lu, and A. Swaminathan
- Team 14 -

May 20, 2014

# Contents

**Contents** **1**

**1 Introduction** **3**
- 1.1 Language Category and Design . . . . . 3
- 1.2 Language Environment . . . . . 4
- 1.3 Examples . . . . . 4

**2 Language Tutorial** **7**
- 2.1 General Instructions . . . . . 7
- 2.2 Sample Programs . . . . . 10
- 2.3 Conclusion . . . . . 13

**3 Language Reference Manual** **15**
- 3.1 Lexical conventions . . . . . 15
- 3.2 Types . . . . . 16
- 3.3 Scope . . . . . 16
- 3.4 Errors . . . . . 16
- 3.5 Expressions . . . . . 17
- 3.6 Assignments . . . . . 20
- 3.7 Statements . . . . . 20
- 3.8 Grammar . . . . . 21

**4 Project Plan** **23**
*By Vaggelis, project manager.*
4.1 Specifications . . . 23
4.2 Meetings and Communication . . . 23
4.3 Development . . . 24
4.4 Implementation Style Sheet . . . 24
4.5 Testing . . . 24
4.6 Responsibilities . . . 24
4.7 Project Milestones . . . 25
4.8 Project Log . . . 25

**5 Language Evolution** **27**
*By Mathias and Yiren, language gurus.*
5.1 Look and Feel . . . 27
5.2 Implementation and Tools . . . 27
5.3 Added Features . . . 27
5.4 Dropped Features . . . 28

**6 Translator Architecture** **29**
*By Mathias, system architect.*
6.1 Architecture . . . 29
6.2 Interfaces . . . 29
6.3 Modules . . . 30
6.4 Organization . . . 31

**7 Development and Run-time Environment** **33**
*By Georgios, system integrator.*
7.1 Development . . . 33
7.2 Runtime Environment . . . 33

**8 Test Plan** **35**
*By Arun and Yiren, testers.*
8.1 Methodology . . . 35
8.2 Relevant Files and Descriptions . . . 36
8.3 Selected Test Cases . . . 37

**9 Conclusions** **39**
9.1 Lessons learned as a team . . . 39
9.2 Lessons learned by each team member . . . 39
9.3 Advice for future teams . . . 41
9.4 Suggestions . . . 41

**10 Appendix** **43**
10.1 GIT Log . . . 43

10.2 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

# Chapter 1

# Introduction

Many computer science projects now involve runing programs on multiple machines, whether to collect data, run analysis, or parallelize computation. At a high level, an end-user operates on a pool of machines and allocates jobs, usually represented in a scripting language. Our purpose is to build a declarative programming language, named Maestro, for users to easily express units of computations (a.k.a. jobs) and their dependencies. The runtime then distributes the jobs and ensures an order that doesn't conflict with expressed dependencies.

As an example, a researcher wants to run an experiment that is divided into 3 steps: populate a third party service, collect data from the service, and analyze the collected data. The data collection has to run an hour after population, but the analysis can start as soon as the data is collected. With Maestro, we can express this in just a few lines, as shown in 1.3.

Prior related work in this area includes the so-called infrastructure configuration management frameworks, some of the most popular being Puppet [?] and Chef [?]. Typical infrastructure configuration management allows system administrators to express infrastructure configuration dependencies. Our approach differs in that it provides end-users the ability to express dependencies related to job work-flows.

In the rest of the document, we first present a high level overview of the language. We then dig deeper with a tutorial and a reference manual. We finally explain the architecture and the process of building the language.

## 1.1 Language Category and Design

**Declarative** The goal is to produce a declarative language to run jobs (scripts) in a distributed way. The programmer only has to register workers and declare job dependencies; the language will schedule and run the jobs.

**Dynamically typed** Maestro is dynamically typed. On top of the classic types it supports a job type.

**Garbage collected** Memory is handled by piggy-backing on Python's garbage collector. Users should not have to manage their memory in a high level, domain-specific language, where performance is not critical.

**No exceptions** Error checking is handled at runtime, but errors cannot be rescued or bubbled up to avoid hard to debug situations from asynchronous callbacks.

**Program structure** Programs are written as a sequence of function definitions and declarations are stored in a file or a REPL command line interface.

## 1.2 Language Environment

The language environment has three major characteristics:

**Workers Pub/Sub** The user shouldn't have to manage his workers after they are launched; he only needs to call a very basic program to register his workers. Maestro will then transparently assign jobs to workers and manage dependencies. Jobs are assigned through a Publish/Subscribe system based on Redis.

**Job queue** Under the hood, jobs are stored in a priority queue. When a worker becomes available, Maestro will search the job queue in order of highest priority first until it finds a job with all dependencies fulfilled. This job will be assigned to the available worker.

**Interpreter and REPL** Maestro is interpreted and has a REPL to be able to quickly try things out and iterate. For really basic job launching, users shouldn't even need to open a text editor.

## 1.3 Examples

Example 1: Create and run 2 jobs $A$ and $B$ in parallel. Then run job $C$, only after both $A$ and $B$ have succeeded.

```
a = Job("print.pl", "Hello, ");      // job scripts
b = Job("print.rb", "world!");
c = Job("yay.py");
run((a <-> b) -> c);
```

Example 2: Make a measurement that is divided into 3 steps: populate (send 100 emails), collect data (1 hour after population step), and analyze the data.

```
populate_scripts = []
range(100).each(i) {
  p = Job("send_email_number.rb", i);
  populate_scripts = populate_scripts + [p];
}
b = Job("collect_data.rb");
Job c = Job("analyze.py");
run(a -> Wait(3600) -> b -> c);
```

# Chapter 2

# Language Tutorial

The purpose of this tutorial is to demonstrate features and functionality supported by Maestro. We also provide examples to explain how Maestro should be used to create jobs, describe dependencies amongst jobs, and execute jobs on workers. Our examples include a simple "Hello World" program as well as more complex programs which cover Maestro's functionality in detail.

## 2.1 General Instructions

### Define Workers in Maestro

We use a Redis [?] server to manage workers and disctribute jobs remotely. To add a worker to the workers' pool, the only thing one has to do is run the following Maestro program: "worker("0.0.0.0:6379");" (replace 0.0.0.0:6379 by the IP and port of the Redis server). This will subscribe the worker to a job channel in Redis, where all Jobs will be published for execution when a master calls the run command.
The master calls "sercive("0.0.0.0:6379");" (replace 0.0.0.0:6379 by the IP and port of the Redis server) at the beginning of its program.

### Variables and Expressions in Maestro

In Maestro variables are reserved memory locations which store values. Practically, when a user declares a variable memory space is allocated.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, user can store integers, Jobs, or strings.

#### Assignment operator

Maestro is a dynamically typed language and variables types are not explicitly declared. Variable types are decided when a variable is assigned a value with the equal assignment operator “=”.

The operand on the left side of assignment operator “=” must be a variable name and the operand on the right side of assignment operator “=” must be a value. The variable on the left side of “=” is assigned the value on the right side of “=”.

Example:

```
a = 10;
x = "john";
path = "/afs/columbia.edu/users/phd/username/my_script.sh";
```

The above code snippet demonstrates the following assignments: In line one variable “a” is assigned integer value 10. In line two variable “x” is assigned string “john”. In line three variable “path” is assigned a string representing the absolute path of some script. Finally, in line four variable “p” is assigned boolean value true.

### Job

The most important type of variable that can be created in Maestro, is the Job variable.

Example:

```
a = Job ("abc.rb", "my_arg");
```

In this case variable “a” is assigned a job related to script “abc.rb”, and “my_arg” and argument that will be passed to “abc.rb” when it is called.

## each

“each” keyword is used to feed a list of variables as input to a block of statements. Its syntax is:

*[val1, val2, ..., valN].each(var){ statements}*

The interpretation is that “statements” are executed as many times as the number of variables in the list [val1, val2, ..., valN]. For the first execution, var is assigned the value val1 and is fed to the statements. For the second execution var is assigned the value of val2 and is fed to the statements. This process is repeated until every value from 1 to N has been assign to var.

Example:

```
vals = ["tiny.txt", "medium.txt", "large.txt"];
vals.each(var){
    a = Job("/usr/bin/grep", "compiler", var);
    run(a);
}
```

In the above code snippet, Job "a" is instantiated and executed three times with different arguments.

## Operators in Maestro

Maestro language supports left associative operators with special meaning: operators $->$ and $<->$ for concurrency. Also operators $<\sim>$, $\sim<$ and $\sim>$ for job dependencies.

### Asynchronous operator ($->$)

To allow user to define that 2 processes must run one after another we implement the dependency operator ($->$). This operator specifies that the process at right of the arrow can start as soon as the process on the left-hand side of the arrow terminates successfully, but not before it finishes.

### Concurrent operator ($<->$)

Unlike the asynchronous operator, the concurrent operator ($<->$) denotes that 2 jobs can run in parallel. Failure of the one does not preclude the other.

### Equal dependency operator ($<\sim>$)

The equal priority operator ($<\sim>$) states that two jobs have the same priority.

### Higher priority operator ($\sim>$)

The Higher Priority operator ($\sim>$) states that the job on its left should run before the job on its right if they can both run. This is a best effort guaranty, not a strong one.

### Post dependency operator ($\sim<$)

Post dependency operator ($\sim<$) states that the job on its left should run after the job on its right. This is a best effort guaranty, not a strong one.

## Jobs and Functions in Maestro

$Job(string < name >,\ string < script\_path >,\ arg1,\ arg2, ...,\ argN);$

Construction of a *Job* type requires the following variables:

- *name* which is the name of the job.
- *script_path* which is the script to be executed.

- A list of arguments related to script under *script_path*.

For each Maestro job a log file is created under the user's current working directory. This log file is named after < *script_path* > along with an additional ".log" suffix. The purpose of a log file is to store diagnostic messages and output generated by the respective job script.

Example:

```
b = Job("xRay.rb","arg1", "arg2");
c = Job("telesphorus.py", "arg1");
```

The above snippet creates two jobs "b" and "c" as instances of class *Job*. Job "b" is assigned the execution of script "xRay.rb" with arguments "arg1" and "arg2". Job "c" is assigned the execution of script "telesphorus.py" with arguments "arg1" and "arg2".

#### Maestro Functions

This section we present core functions of Maestro.

*run*(***expr***);

*run* function has one argument, which is an expression related to jobs. *expr* is a comma separated list of Job of list of Job types. The denpendency operator returns a list of Jobs.

```
a = Job(script="abc.pl", "10");
b = Job(script="xRay.rb", "mailbox");
c = Job("telesphorus.py", "android_logs");
run(a, b -> c);
```

## 2.2 Sample Programs

In order to run the programs, we just need to call maestro program_path. If the Maestro program is not in the PATH, it need to be specified (e.g. /home/user/bin/-maestro).
We can also start the REPL with maestro, and write the commands.

### Example Program 1: Hello World

Consider a Ruby script named *hello_world.rb* containing the following:

```
#!/usr/bin/env ruby
puts 'Hello World!'
```

Now consider a Maestro program like the following:

```
1 // Hello World
2 a = Job("hello_world.rb");
3 run(a);
```

In the above code snippet we demonstrate how to create of a job running a hello world script. This job is then initialized with the script of our hello world program and assigned to variable 'a'. In line 3, *run*() starts job 'a'. Maestro will find an idle worker and execute our script hello_world.rb. Afterwards, error code of job 'a' is evaluated. If script hello_world.rb terminates properly ,its stdout will be loged on the master. Otherwise, stderr will be logged.

## Example Program 2: Distributed Hello World

Consider a Ruby script named *print.rb* that prints its first argument:

```
#!/usr/bin/env ruby
puts ARGV.first
```

We then start workers running this maestro program:

```
1 worker("0.0.0.0:6379");
```

The Maetro program can be:

```
1 service("0.0.0.0:6379");
2 a = Job("print.rb", "Hello");
3 b = Job("print.rb", "World");
4 c = Job("print.rb", "!");
5 run(a -> b -> Wait(10) -> c);
```

There is a lot going on in this example. It uses the concurrency capabilities of Maestro to express dependencies and waiting times. We can also see how to run a Job with an argument.
First, we declare the IP:PORT of Redis (here localhost on the default Redis port). It will be used to distribute the Jobs.
3 jobs are created, and dependencies amongst them are specified. Line 5 shows a simple example of how to introduce dependencies in Maestro. The expression $a- > b$ states that job 'a' will run before 'b', potentially on different servers. Afterwards, $- > c$ indicates that the job 'c' must run after the successful termination of the previous jobs ('a' and 'b').
The Wait() call creates a fake Job which only purpose it to delay the next Job. The hard dependency system ensures 'c' will not run before 10 seconds after 'b' finished.
If we wanted 'a' and 'b' to run in parallel, and add the ! at the end after 10 seconds, we could call:

```
run((a <-> b) -> Wait(10) -> c);
```

You can see that Maestro offers an easy way to distinguish between concurrent and non concurrent jobs. This will help the user create and easily distribute batch processes.

## Example Program 3:

```
1 service("0.0.0.0:6379");
2 a = Job("abc.pl");
3 b = Job("xRay.rb");
4 c = Job("telesphorus.py");
5 d = Job("nimbledriod.py");
6 ((a~>(b <~> c))~<d);
7 run((a,b,c,d);
```

This example demonstrates soft dependencies in Maestro.
'Soft' dependencies must be all in one line. Operators are left associative but we use parentheses in order to help the user understand how 'soft' dependencies work. Line 5 is where all the magic happens. First, $(b <\sim> c)$ means that jobs 'b' and 'c' can run synchronously if there are enough resources. Second, $(a \sim> expr)$ means that job 'a' should run before *expr*. Finally, $(expr \sim< d)$ means that job 'd' should run before *expr*.

## Example Program 4:

```
service("0.0.0.0:6379");
list = [];
range(100).each(i) {
  a = Job("send_email.pl", "" + i, "test@maestro.com");
  list = list + a;
}
b = Job("collect_data.rb");
list -> Wait(3600) -> b;
run(list, b);
```

This examples shows how to schedule many instances of the same Job with different arguments, and wait for all of them to be done before starting another script. In this example, send_email takes an email number and an email address as arguments, and sends the email to the address. The program creates one Job per email so that it can be distributed. Then it waits for an hour after the last email, and starts some data collection dependent on the emails. It assumes all workers have access to the email list so that they can retrieve the right one (it could be hard coded in the script or through a distributed file system). Execution of job ‘b’ will start only on success of all email sending, and after an one-hour wait.

**Example Program 5:**

```
master("0.0.0.0:6379");
a = Job("split.rb", "/tmp/big_file_name.data", "10");  # split file in 10
maps = map(a, "map.rb", 10);  # run 10 map.rb jobs
red = reduce(maps, "reduce.rb");  # reduce the results
run(a, maps, red);
```

This example shows how to use the map/reduce built in functions to distribute data analysis of a big file. This assumes that all workers have access to the file (e.g. over a Network File System), or that split.rb can scp the pieces of the file in the workers. Then map will create 10 Jobs and have them run on the outputs of split.rb. These Jobs will be the map.rb script and have as argument one line of split output. Reduce will then create one Job with the reduce.rb program, with its arguments being the outputs of the previous maps.
Let us assume the big file contains words, and we want to count the number of appearances for each different word:

- split.rb can split the big file in chunks and output the name of each chunk.
- map.rb can take a file path, and return a dictionary with the words as keys and an int for the value, in the JSON format.
- reduce.rb can take n arguments as JSON strings and parse them. Then it merges the dictionaries and returns the merged in JSON.

## 2.3 Conclusion

This language tutorial presents the core functionality of Maestro. It includes examples demonstrating how Maestro primitives should be used to create jobs, specify dependencies, and run jobs. For a complete analysis refer to our Language Reference Manual.

# Chapter 3

# Language Reference Manual

Maestro is a declarative language built to easily schedule jobs with hard and soft dependencies, locally or over the network. Its goal is to provide a DSL to quickly express dependencies between jobs and run them, either from the REPL or from a file. It exposes high level functions and operators to declare complex logic. Maestro is meant for scripting, not for developing large scale programs. It thus doesn't support functions, or conditionals. However, it supports loops with an each statement and exposes a map/reduce interface that allows for complex declarations.

## 3.1 Lexical conventions

### Whitespace

Whitespaces are either a tab, a new line, or a white space. At least one white space is required to separate identifiers from each other, and certain operators.

### Comments

Comments are denoted by "#" or "//" and represent a comment that ends with the end of the current line.

### Identifiers

An identifier is a sequence of letters, digits, and underscores. The first character must not be a digit.

### Keywords

The following identifiers are used as keywords, and are reserved names:
each range

### Reserved Identifiers

The following characters are reserved: `+ - / * % -> => <-> ~> ~< ~, ( ) " .`

## 3.2 Types

Maestro is dynamically typed and doesn't allow users to create new types.

### Basic Types

**Job** A Job represents a program that can be run on a worker. It encapsulates the path of the program and exposes outputs and errors.

**Int** Ints are signed integers of arbitrary size.

**String** Strings are delimited by " and have to be on one line.

### Derived Types

**List** List are ordered collections of one or more values. These objects can be of any type. The literal representation is a comma separated sequence of expressions surrounded by brackets ([ and ]).

## 3.3 Scope

Maestro is a scripting language used to quickly distribute interdependent jobs. Its intended use is for scripts that glue tasks together, not to build and maintain large programs. The scope of a variable is thus the entire script (global scope), or the whole lifetime of the interpreter.

## 3.4 Errors

There are two kinds of errors in Maestro:

- logic or type errors will result in the program crashing with an error message.
- Job errors create a log on the local machine. If a later Job depend on this Job, the program will crash. If not, it will continue uninterrupted.

## 3.5 Expressions

### Identifier

An identifier evaluates to the value of the corresponding variable.

```
expression:
    identifier
```

### Parenthesized expression

A parenthesized expression takes the value of the evaluated expression.

```
expression:
    ( expression )
```

### Function call

A function call is an expression. The arguments are comma separated and are evaluated from left to right before the call. It is possible to have 0 arguments.

```
expression:
    identifier ( expression-list )

expression-list:
    expression-list , expression
    | expression
```

#### Built in functions

The following functions are built into the language, and their names are reserved:
map reduce run Job

**Job(string, string..)** Job takes a string that is the path of a file to execute for the job, and any number of strings that will be passed as arguments. It returns a list with one Job object.

**Wait(int)** Creates a Job which only sleeps for the given time. It can be inserted in the dependencies to wait between Jobs. It returns a list with one Job object that just waits.

**worker(string)** This subscribes the worker to a job channel on redis running at the IP:PORT denoted by the string. From now on, the worker will pushed jobs to be execute on that channel. It doesn't return anything and crashes if redis doesn't answer.

**service(string)** This function is used to declare the IP:PORT (denoted by the string argument) on which the redis server will listens. From now on, the service will publish jobs that need to run on the job channel of the redis for a worker to execute. It doesn't return anything and crashed if the redis doesn't answer.

**range(int)** Returns a list of ints from 0 to int-1.

**map(Job / list, string, int)** the first argument is a job that will run first. For every line in its stdout, map will create a Job with the string argument and the stdout line as an argument, up to "int" Jobs. If a list of strings is passed instead, these strings will be passed as arguments. It returns the list of created Jobs. This call is asynchronous.

**reduce(list, string)** the first list is a list of jobs that will run first (usually the output of a map). String is the path to the reduce program. There will be one Job created with that program, with arguments that are the output from the Jobs in the list.

**run(list)** run takes an arbitrary number of comma separated lists of Jobs and runs them. If they have dependencies, these dependencies will run first.

## Lists

Lists are created with the brackets notation. E.g. $list = [a, b, c, ...]$

expression:
    [ expression-list ]

## Operators in expressions

expression:
    expression OPERATOR expression

## Operators

Operators are left associative and have the following precedences (same line means same precedence, lower line means higher precedence):

```
-> => ~> ~<
+ - <-> <~>
/ * %
```

## Operators description

$+$

The result is the sum of the expressions if they evaluate to an Int.

The result is the concatenation of both expressions if both expressions evaluate to a list.
The result is the concatenation of the strings if both expressions evaluate to a string.
The result is the concatenation of the string and the Int in a string format if one expression evaluates to a string and the other to an Int.

**-**

The result is the difference of the expressions if they evaluate to an Int.

*****

The result is the multiplication of the expressions if they evaluate to an Int.

**/**

The result is the Integer division of the expressions if they evaluate to an Int.
When dividing by 0, the program crashes.

**%**

The % operator yields the remainder from the division of the first expression by the second, if both expressions evaluate to an Int.

**->**

The dependency operator (->) requires both expressions to be lists of Jobs.
The call is asynchronous and adds all Jobs on the right hand side list as dependencies for all Jobs on the left hand side list.
It returns a list of Jobs that is the concatenation of both lists.

**=>**

The distributed dependency operator (=>) requires both expressions to be lists of Jobs.
The call is asynchronous and adds all Jobs in the right hand side list as dependencies for all Jobs on the left hand side list. Moreover, the dependent Jobs (right hand side) will be run on the same worker as the previous Job (left hand side).
It returns a list of Jobs that is the concatenation of both lists.

**<->**

The parallel operator (<->) requires both expressions to be lists of Jobs.
The call is asynchronous and no dependencies are added. It is a visual way to concatenate lists of Jobs.
It returns a list of Jobs that is the concatenation of both lists.

**~>**

The soft dependency operator (~>) requires both expressions to be lists of Jobs.
The call is asynchronous and puts the priority of all right hand side Jobs to be 1 less than the lowest priority of the left hand side Jobs.

It returns a list of Jobs that is the concatenation of both lists.
A soft dependency is a priority indication: if more than one Job can run, Maestro will run the one with higher priority, but there is no order guarantee, as opposed to dependencies.

~<

The soft dependency operator (~<) requires both expressions to be lists of Jobs. The call is asynchronous and puts the priority of all right hand side Jobs to be 1 more than the highest priority of the left hand side Jobs.
It returns a list of Jobs that is the concatenation of both lists.
A soft dependency is a priority indication: if more than one Job can run, Maestro will run the one with higher priority, but there is no order guarantee, as opposed to when using dependencies.

<~>

The soft parallel operator (~) requires both expressions to be lists of Jobs.
The call is asynchronous and no priorities are changed. It is a visual way to concatenate lists of Jobs.
It returns a list of Jobs that is the concatenation of both lists.

## 3.6 Assignments

The value of the variable whose name is the identifier value is replaced by the value of the expression evaluates to. If the variable doesn't exist, it is created: the programmer doesn't need to declare the variable beforehand.
The assignment is also an expression that has the variable final value as value, which is useful for one liners.
The default value of the variable is not determined.

```
expression:
    identifier = expression
```

## 3.7 Statements

Statements are executed in sequence.

### statement

Statements are semicolon ended expressions.

```
statement:
    expression ;
    | ;
```

### statement-list

Most statement are expression statements which have the form:

```
statement-list:
        statement statement-list
        | statement
```

### each statement

The each statement is a basic loop, and can be called on the Jobs type.

```
statement:
        expression .each ( identifier ) statement-block
```

The first expression has to evaluate to a list type. The identifier is the name of a variable (global scope like all variables) that will be assigned each element of the list before running the statement list.

```
statement-block:
        { statement-list }
```

### Program definition

A program consists of a list of statements evaluated from left to right.

```
program:
        statement program
```

### REPL

In the REPL, inputs are read and evaluated every time the user hits enter.

## 3.8 Grammar

```
expression:
        expression + expression
        | expression - expression
        | expression * expression
        | expression / expression
        | expression % expression
        | expression -> expression
        | expression => expression
        | expression <-> expression
        | expression ~> expression
        | expression ~< expression
        | expression ~ expression


expression:
        ( expression )

expression:
        identifier

expression:
        identifier ( expression-list )

expression-list:
        expression-list , expression
        | expression

expression:
        [ expression-list ]

expression:
        identifier = expression

statement:
        expression ;
        | ;

program:
        statement program

statement-list:
        statement statement-list
        | statement

statement:
        expression .each ( identifier )  statement-list

statement-block:
        { statement-list }
```

# Chapter 4

# Project Plan

*By Vaggelis, project manager.*

Our first important decision as a team was what language we were going to implement and whether it would be domain-specific or not. We took a look at the examples from previous years and found that the most compelling ones did not attempt to replace general purpose languages like C/C++, but rather addressed quotidian concerns. So upon deciding on a domain-specific language, we met in the second week and began brainstorming problems that could be reasonably and competently solved in a class project. Some of our ideas included a probabilistic programming language that would make it easy to specify joint and conditional distributions, as well as a food-recipe language akin to What2Wear. We each wrote sample programs, did research, and presented our ideas. At the third week, we decided that we would implement Maestro and assigned roles to each team member.

## 4.1 Specifications

Our next step was defining core specifications of the Maestro and getting a small kernel up and running. This eventually occurred sometime between the whitepaper and spring break, at which point we met with our mentor, Junde Huang. Junde had some good advice for organizing work flow. He also proposed adding a graphics unit to the language (which was ambitious and cool but did not eventually happen).

After spring break, we wrote our language reference manual and tutorial and started working on an implementation of additional features.

## 4.2 Meetings and Communication

During the semester we tried to meet weekly. The main purpose of our meetings was to ensure that each team member had a full understanding of what she/he was expected to do, and also a full understanding of what other members were responsible for. Deadlines on deliverables called for extra scheduled meetings, and we also communicated through emails.

## 4.3 Development

For our development the single most important tool was GIT, a distributed version control system. Multiple GIT branches were created during the semester to keep track of our project. They included branches for source code subversion (e.g., tests, translation, and the Maestro backend) as well as branches for written manuscripts (e.g., whitepaper, tutorial, and LRM). For more details refer to Section 7.1.

## 4.4 Implementation Style Sheet

A five member team is relatively easy to manage and integrate relative to corporate settings, yet we discovered that good abstraction of responsibilities is essential: this allows the members of the team to work concurrently and make progress independently as the development evolves. We did not set any restrictions on text editors or operating systems, but did ask of each other readable, commented python code. We also asked that partial commits of code snippets come with meaningful commit messages. The source code and all written manuscripts of our project live under the private Columbia github repository.

## 4.5 Testing

To make sure that the entire workflow was smooth and to avoid bugs, we created a custom testing suit, built in python. Our testing environment is thoroughly described in Section 8.1.

## 4.6 Responsibilities

In Table 4.1 we summarize responsibilities of each team member.

| **Team Member** | **Responsibility** |
|---|---|
| V. Atlidakis | Backend Redis-based distributed communication protocol. |
| M. Lecuyer | Lexical & Syntax analysis, SDD to produce the AST, and translator. |
| G. Koloventzos | Semantic analysis. |
| Y. Lu | Test suite and language syntax |
| A. Swaminathan | Design and implementation of test suite |

Table 4.1: **Responsibilities.**

## 4.7 Project Milestones

In Table 4.2 we summarize project milestones and the date each was fulfilled.

| **Date** | **Milestone** |
|---|---|
| Feb 24 | Language whitepaper |
| Mar 17 | Basic Compiler Front End (lexer and parser) |
| Mar 26 | Language reference manual and tutorial |
| Mar 30 | Basic language kernel (without semantic actions) |
| Mar 30 | Hello World (locally) |
| Apr 15 | Initial test suite |
| May 7 | Semantics & type checking complete |
| May 8 | Additional language features |
| May 8 | Backend redis-based distributed communication protocol. |
| May 10 | Final report |
| May 11 | Presentation |

Table 4.2: **Project milestones.**

## 4.8 Project Log

In Section 10.1 we present a sample snapshot of some git commits at the maestro branch that was used as our master source code branch.

# Chapter 5

# Language Evolution

*By Mathias and Yiren, language gurus.*

## 5.1 Look and Feel

The goal of the language was clear from the very beginning - Maestro was designed to make it easy to schedule jobs, with dependencies between them, or in parallel, locally, or over the network. With that in mind, we were mostly able to hem to the original specifications in the whitepaper. In order to get an idea of the uses cases we had in mind and the semantics we wanted, we all wrote three to five programs in maestro. Great ideas came out of this exercise, like our dependencies operators.

## 5.2 Implementation and Tools

There were, however, some changes and feature additions. Originally, the team had decided to do the project in C. C became Python when it became clear that Maestro would be used primarily as a scripting language. We thus decided to implement an interpreter instead of a compiler. We also chose Python because we wanted to use some of its higher level libraries to implement network operations, distributed map/reduce, and dynamic dependencies between Jobs with lambdas. We thus decided to use PLY to implement Maestro's frontend.

Redis came into the picture when we started to distribute jobs on remote workers. Fortunately it was compatible with the architecture, since it had always been on our "would be great to have" list. We use the python Redis package to interface with the Pub/Sub feature of redis.

## 5.3 Added Features

The first big feature we changed is the way we do recursion in Maestro. We were planning classic for and while loops, but realized we were always looping over lists. We thus changed the for loops to the each syntax. This is similar to how loops are handled in Ruby and is much more compact and enjoyable.

Another feature that we introduced late is map/reduce. When thinking about the use cases (after we had a basic "Hello World" working), most of them involved complex processing and dependencies, and where a bit verbose or even too hard to express because we needed to wait on other jobs' outputs. We thus introduced a map/reduce abstraction, which allows us to keep our asynchronous model, but still use the output of previous jobs. This is really adapted to remote job distribution and execution, and fits perfectly into Maestro.

We also added overloading for + and map functions: the behavior changes depending on the type of the arguments. We didn't think of it beforehand, but when we hit the problem it was a natural extension.

A last unexpected improvement we added was nice console outputs of the job queue and the state of dependencies. The language reference manual and the tutorial were updated as we changed features.

## 5.4 Dropped Features

There still are some features on our "would be great to have list": We wanted to add more error handling features, functions and lambdas for dependency callbacks, and scoping for variables. We believe our architecture can support it, but we chose to focus on remote job execution.

# Chapter 6

# Translator Architecture

*By Mathias, system architect.*

## 6.1 Architecture

Maestro is organized as a pipeline that processes the program and its input from a file, or from a REPL, and interprets the code. Figure 6.1 gives a high level overview of Maestro's architecture. In the next sections, we present the interfaces between different parts of our interpreter, and then describe each module.

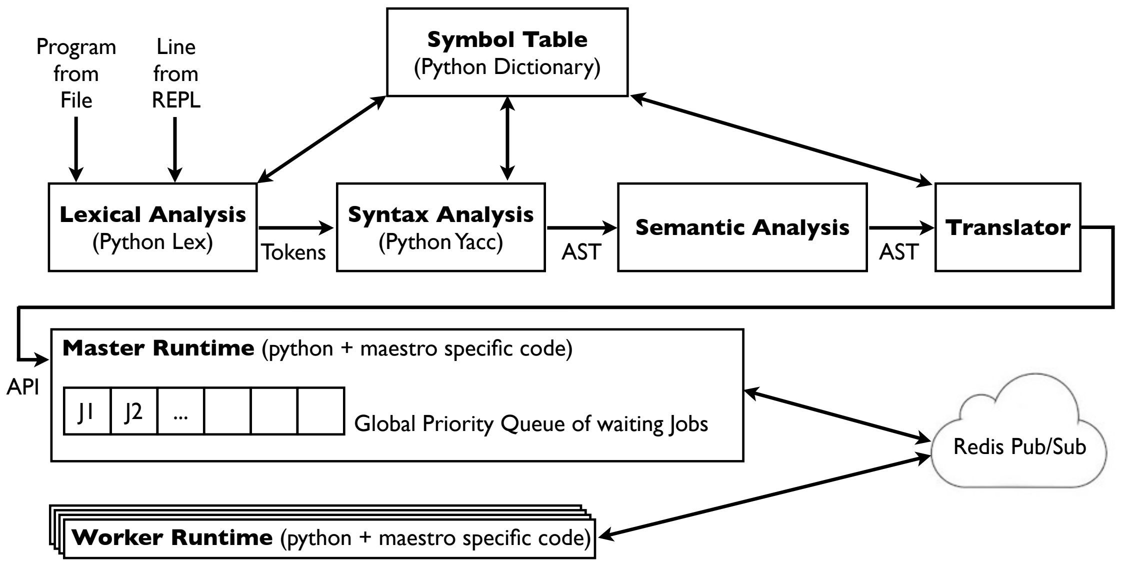
Program from File
Line from REPL
Symbol Table (Python Dictionary)
Lexical Analysis (Python Lex)
Tokens
Syntax Analysis (Python Yacc)
AST
Semantic Analysis
AST
Translator
API
Master Runtime (python + maestro specific code)
J1
J2
...
Global Priority Queue of waiting Jobs
Redis Pub/Sub
Worker Runtime (python + maestro specific code)

Figure 6.1: Maestro Architecture

## 6.2 Interfaces

The two main interfaces are the AST and the API. We will describe them in turn.

**AST** The AST is produced by the syntax analyser. It is a tree with each node having children, or a value if it is itself a leaf. The node also comprises the type of the expression it represents. It is used by the syntax analyser and the translator.

**API** The API is exposed by the backend, and gives Maestro-specific features to the translator. In particular, it exposes a Job object, central to our language, and functions to set dependencies and add Jobs to the run queue.

## 6.3 Modules

**Lexer** The lexer takes the program or the REPL lines, and produces tokens for the syntax analyser. Comments are discarded by the lexer directly.

The Lexer was written by Mathias with PLY.

**Syntax Analyser** The Syntax Analyser contains the grammar for Maestro, and uses the lexer's output to produce the AST using bottom-up parsing and an S-attribute SDD.

The Syntax Analyser was written with PLY. Mathias wrote the grammar and SDD and Georgios handled error recovery.

**Semantic Analyzer** The Semantic Analyser walks the AST to verify type compatibility and that variables were declared before being accessed.

The Semantic Analyser was written by Georgios.

**Translator** The translator translates the AST into Python and executes it. It is coded as a recursive function that keeps track of computed values and types. It makes extensive use of the backend API for Job creation and management, and leverages python expressiveness to implement complex features like map/reduce and dynamic code dependencies.

The translator was written by Mathias.

**Backend** The Backend exposes Maestro specific abstraction and handles the Job scheduling and distribution. Because it's fairly complex and interesting, but doesn't have its own section, we descibe it in more length here. To support Maestro's job scheduling and dispatching for remote execution, we built a distributed communication protocol whose main components are: (i) a local job queue, (ii) a Redis publish/subscribe communication channel, and (iii) a pool of workers dedicated to job execution. The local job queue is used to keep track of active jobs and their dependencies and is bound to a runtime environment. A custom Daemon spins periodically on the job queue and dispatches for execution jobs with resolved dependencies. Since job execution is either local or remote, job dispatching is optional. On local execution, the job dispatcher requests a shell on the local machine for job execution. On remote execution, the job dispatcher sends requests to check which are the active workers on the Redis channel, and afterwards dispatches (publishes) a job to the channel. One of the subscribed active workers executes the job and sends back the output, on a job specific redis channel. When the runtime Daemon receives the output of the dispatched job via the job-specific channel, it prints its stdout and also saves it in a log file under the current working directory.

Most of the Backend code was written by Vaggelis. Georgios and Mathias added some glue and small features.

**Testing** : Testing is a key part of any complex software system. This is why it was assigned two persons. Our testing framework will be described in details in a later chapter.

The testing frameworks and most of the tests were written by Arun and Yiren, and integrated from informal tests used by developers of new features.

## 6.4 Organization

The hard thing with a multi-person project with many parts depending on each other is figuring out a way to all work in parallel. Even if we didn't always succeed, we took several steps to maximize parallelism. First, we had an early "hello world" program working by implementing a basic lexer and semantic analyser and a basic backend. We did not produced an AST and called backend functions directly, but having an end to end program helped a great deal in thinking about more advanced features we wanted.

We then added the production of an AST from the lexical analyser, while still computing values on the fly to keep being able to run programs. This allowed us to add semantic analysis, and work on the translator that traverses the AST. When we finally switched to working on a translator, adding features from the backend like remote Jobs was much easier.

# Chapter 7

# Development and Run-time Environment

*By Georgios, system integrator.*

## 7.1 Development

Our interpreter was developed in a Unix-based environment, specifically Debian, Ubuntu and MacOS. Each member used their favorite text editor, including Vim, Sublime, and Emacs. We used Github to host our repository. As version control system we used Git. Some of us used tig for terminal access to git in text mode. We did not used any issue tracking tools as our main development was done in sync with all members in the same room. Our presentation was created, edited and distributed with Google Drive.

We used Python 2.7.3 and the Python-Lex-Yacc (PLY) to create and test our language. A generic Makefile was created for the purpose of running all tests at once. Most of our examples are bash scripts. For more sophisticated scripts such as map-reduce, we used ruby.

Our interpreter can take a file as argument. A REPL is started if no argument is specified by the user.

## 7.2 Runtime Environment

One of the main goals of maestro was to distribute jobs and test them in computers with different architectures and operating systems, so we wanted it to have as few environment dependencies as possible. Towards that end, our interpreter has 2 running options. For distributed functionality, there are a couple of configuration issues to first take care of: firstly, our interpreter needs a PC that can connect to the internet and can spawn threads. Secondly, download the following packages from Python.

- *pip install redis*
- *pip install colorama*

Our user does not require an active redis server. An online machine with the redis server installed will be used as a master channel. The user only needs to retrieve the ip of the machine and the redis port that is listening.

### Local

When run locally our interpreter creates processes to run jobs. No changes are needed in the environment.

### Distributed

For a distributed run the user must launch a master and a worker in redis in localhost. To create a master the user can specify *service(IP:PORT)*. and for a worker he can specify *worker(IP:PORT)*. Both of these functions are handled by our interpreter.

# Chapter 8

# Test Plan

*By Arun and Yiren, testers.*

## 8.1 Methodology

We did not rely on any existing Python unit-test frameworks for our testing purposes. Instead, we developed a customized testing engine for our project. The engine, described in the tests section, consists of several Python files. Each file contains a test written in Maestro that checks both the robustness of the compiler (i.e. its ability to identify and parse tokens), and the final output of the program.

Presented below is a sample unit test.

```
#!maestro
master("0.0.0.0:6379");
# check what happens when a job is run without being declared
a = Job('print.py', 'foo');
c = Job('print.py' 'bar');
run(a -> b -> c);  # Job b is not declared
```

The test first checks if two Jobs are created successfully. The file *print.py* is a Python program that writes the argument passed to it to a file *check.txt*. Once executed, Job *a* must run the argument *foo* through the program *print.py*. Similarly, Job *b* must run the argument *bar* through *print.py*. Once the two Jobs have been created successfully, we attempt to pass an undeclared Job to the *run* method, which executes a single Job or a list of Jobs in the given order. In this case, we check to see if the Maestro compiler identifies undeclared Job *b* as an error and handles it elegantly. Once the test completes execution, we check the expected output against the output that was piped to *check.txt* to confirm success or failure.

We have a total of 15 testcases in our engine that test Maestro's overall performance in different steps.

## 8.2 Relevant Files and Descriptions

**testframework.py**

This is the main framework file. It can execute all Maestro test-cases or individual ones, depending on how it is called.
*python testframework.py all* executes all test cases in the *Tests* folder. Replacing *all* with the test-name will execute individual tests.

**run.sh**

Similar to testframework.py, but written in bash script.

**Tests\print.py**

This program is run by every Job created in our test cases. It writes the argument that it receives into file *check.txt*.

**Tests**

This folder contains all 75 Maestro test cases.

**TestsOutput\check.txt**

The checkfile to which *print.py* writes its output. We compare the contents of the file against the expected output to determine success or failure of a given Job.

## 8.3 Selected Test Cases

1. This test checks the performance of Maestro when presented with circular dependencies.

   ```
   #!maestro

   a = Job("./tmp/test.sh", "bla");
   b = Job("./tmp/test.sh", "foo");
   c = Job("./tmp/test.sh", "chk");

   run(a->b->c->a); //there is a circular dependency here
   ```

   Here job $a$ depends on $b$, $b$ depends on $c$ and job $c$ in-turn depends on job $a$.

2. This test checks the performance of Maestro on self dependencies.

   ```
   #!maestro

   a = Job("./tmp/test.sh", "bla");
   b= Job("./tmp/test.sh", "bla");

   run(a->a); //self-dependency
   ```

   Here job $a$ depends on itself.

3. This test checks Maestro's response to an instance of imbalanced parentheses.

```
#!maestro

a = Job("./tmp/test.sh", "bla");
b= Job("./tmp/test.sh", "foo"; //Imbalanced parenthesis
c = Job("./tmp/test.sh", "bla");

run(a);
```

4. This test case checks what happens when we provide a Map-Reduce job to Maestro.

```
#! maestro

a = Job("./cut.rb", "./all.txt", "3");

maps = map(a, "./count.rb", 3);
red = reduce(maps, "./reduce.rb");

run(a, maps, red);
```

# Chapter 9

# Conclusions

## 9.1 Lessons learned as a team

- For a large project such as writing an interpreter, it is important to have a game plan ready ahead of time. It became especially apparent to us that we needed to be able to manage our time well, since each team member got exponentially busier as the semester went on.
- We learned to make full use of version control using GitHub. This helped each of us stay on track with others' progress, and stay in sync through the course of the semester.
- Communication was key in making this project successful. We learned that we had to be able to talk to our team members with an open mind, and at the same time also make our point come across in a respectful way.

## 9.2 Lessons learned by each team member

1. **Arun Swaminathan**
   Being a part of the team that built Maestro has been my first experience with working on a large-scale project. I learned a lot about how to work with a team and deliver code on time. One of the biggest takeaways for me is my knowledge of the intrinsics of compilers. I had to dive into the nitty-gritty of what I, as a developer, had taken for granted for most of my coding lifetime. Being able to understand *why* a compiler does what it does, *how* it does it and understanding how all of its parts come together to make a robust piece of software is of the utmost importance.
2. **Yiren Lu**
   As much as modularity was emphasized in the instructions for the project, I realized how difficult this is to implement in practice: chunking up the compiler in

such a way that every person can work independently (as opposed to conditioned on someone else finishing their part) is practically an exercise in gerrymandering. This was doubly hard in this instance, because the components of a compiler come together more or less linearly - lexer, parser, AST, semantic analysis. You can't do the semantic analysis without the AST; you can't test nontrivial programs without the semantic analysis. Some of these problems probably could have been avoided if we had a skeleton compiler working front to back, early on in the semester - Mathias was actually very good about putting the lexer and parser together before spring break - but some minor bugs turned out to take longer than expected to fix. Something to remember for next time.

3. **Vaggelis Atlidakis**
   Interaction and communication within a team of five member may seem trivial, but it is not. The main challenge was that, although we all had good programming skills, none of us had good knowledge of the procedures involved in building a compiler. In order to contribute to the implementation of Maestro, we had to continuously learn new things and use them to contribute to the evolution of our project. The biggest take-away from this joint effort is that good communication can drive great results.

4. **Mathias Lecuyer** Knowing how a programming language works, with first hand experience, is very valuable. I think it will help me understand better the DSLs I use everyday, and ask the right questions about semantics when I create one (even if it's usually inside a language like Ruby rather than as a standalone language). This is, I think, my main takeaway from this class. I also really enjoyed the more formal parts, especially lambda calculus.

   My second take away is on the human side of programming, and especially parallelization. This is not an easily solved problem in software, and not surprisingly it is also tricky with developers. Some people are not proactive in their involvement, and it is very easy to leave them aside. However with the right push, they can make great contributions, and it is important to find the right job for them and integrate them. In our team, it seemed very linked to the separation of the project into well defined blocks. We managed to split the project into only three well-defined blocks, which we ended up working on in parallel: frontend, backend, and testing. This meant that it was not possible for each of the five members to own a part of the code, and probably contributed to some lack of involvement early in the project. Moreover, testing was hard at the beginning because the language was breaking all the time and was not very well defined. When we added the AST and the translation, we became more agile and the project was more stable, so it became easier to contribute. I think that in the end every one of us found a way to make very valuable contributions.

5. **Georgios Koloventzos** It was a challenge to go from theory to a full-fledged interpreter for distributing jobs: understanding how the lexer and parser really

work in order to do dynamic type checking and traversal of the abstrax syntax tree; making decisions about the structure of the language and the content of error messages. Simple feature changes which were seemingly easy to articulate also forced us to restructure a lot what we had already implemented. And like probably most teams, we ended up with some ideas that didn't quite make it into the project. Nevertheless we have a strong distributed job orchestrator which may help scientists around the world test, execute and analyze data.

## 9.3 Advice for future teams

Two main pieces of advice that we would give to future teams: 1) Pick your language carefully and 2) Integrate continuously. When we were brainstorming project ideas, a couple of different ones were bandied around and we were very organized (probably the most organized ever) about listing their pros and cons. We were each assigned to do research and write sample programs for how we envisioned the language we came up with. Some other ideas that eventually weren't voted in were a probabilistic programming language as well as a language for creating recipes out of input food ingredients. After some discussion, we eventually picked Maestro.

We were lucky because Maestro turned out to be the right scope - it was organized around a central demand - being able to run jobs potentially in parallel or with dependences - so basically all aspects of the language bent to that imperative. It was also cleverly turing-complete. Since Maestro is a programming language to organize and run other programs, it can technically run any program that those programs can run.

The second piece of advice derives from something we did less well. It is tempting to say that we're done with 2/3 of the project when 2/3 of the pieces have been completed. But until all these pieces have been integrated, we've essentially done very little. Integration is the biggest challenge, and it gets harder the longer you wait to do it, not only because there are more pieces, but because it's harder to parallelize the work (other team members are more and more out of the loop). The optimal way to integrate is constantly.

## 9.4 Suggestions

Each of us had different thoughts on which components of the class should be kept and tossed out, but here are a few that we were consistent on:

1. More referencing back and forth to the initial compiler diagram. Some of us often felt lost in the nitty gritty. For instance, it took until reviewing for the final exam to really realize the significance of Ershov numbers, and where they fit into the bigger picture.

2. Go more in depth in the first introduction to lambda calculus. Some of us had never taken the computer science theory course here (or had previously been

exposed to lambda calculus but only superficially). For instance, beta reductions require that you replace all free instances of the variable in the expression. But the variables that are free in the expression turn out to be bound in the abstraction. This was a source of confusion.

3. If you're short on time, you can probably cut some of the arithmetic and boolean lambda calculus in the last lecture. There was too much to motivate properly, so many of us ended up trying to memorize the lambda abstractions for true and false, which undermines the spirit of lambda calculus in the first place.

4. We enjoyed the guest lectures. They were amusing and different and should be kept.

# Chapter 10

# Appendix

## 10.1 GIT Log

```
* commit 31eefb3
| Author: Mathias Lecuyer <mathias@lecuyer.me>
| Date:   6 hours ago
|
|     with right order for deps
|
* commit af24c50
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   9 hours ago
|
|     Handle exception at server to keep it running after bad executions
|
* commit b131a1b
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   10 hours ago
|
|     print args of script executing; some doc
|
* commit ad963ef
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   11 hours ago
|
|     Clean ups
|
* commit 9ebae86
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   12 hours ago
|
|     Added tests. All green.
|
* commit ffde30b
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   18 hours ago
|
|     BugFix: compute_args  fix
|
* commit 5fd30b6
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   19 hours ago
|
|     BugFix: When worker runs behind router get external IP
|
* commit 3b4f5d4
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   20 hours ago
|
|     F)(&(*^^*% JSON to json
|
* commit f24a3ba
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   20 hours ago
|
|     fix for mathematical operators
|
* commit 25c0d5c
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   22 hours ago
|
|     change SIGINT into SIGQUIT to be compatible with the REPL
|
* commit 90d666e
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   22 hours ago
|
|     refactor jobs: check active subscribers and associate  with one
|
* commit 5fda9b6
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   23 hours ago
|
|     Worker prints the name of script if executes
|
* commit 2347e11
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   23 hours ago
|
|     clean-up worker
|
* commit f170f61
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   23 hours ago
|
|     refactor worker:  if job not for your IP do nothing
|
* commit 4da4def
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   24 hours ago
|
|     some beautifications for pep8 semantic analysis
|
* commit 06ebd3f
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   24 hours ago
|
|     some beautifications for pep8 mlex file
```

```
|
* commit 4edc3da
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   24 hours ago
|
|     some beautifications for pep8
|
* commit 1de694e
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   24 hours ago
|
|     add semantic for all mathematical functions; some beautifications for pep8
|
* commit 70fe65a
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   25 hours ago
|
|     refactor to avoid print checks
|
* commit c5a0480
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   25 hours ago
|
|     Add support so that each job is executed by one worker
|
* commit 2b786f8
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   25 hours ago
|
|     Minor fixes
|
|     Tests that are SUPPOSED to fail are still printed as a fail. Working on
|     it.
|
* commit d0677d3
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   26 hours ago
|
|     Job arguments type checking
|
* commit f58aca6
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   26 hours ago
|
|     fix for line in dependancy
|
* commit 36b4d8b
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   28 hours ago
|
|     Fixes for MR tests
|
* commit 948b086
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   28 hours ago
|
|     small syntax fix for import
|
* commit e4de7f7
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   28 hours ago
|
|     adding handler for job dispatching killing
|
* commit 1d8b5ba
| Author: Mathias Lecuyer <mathias@lecuyer.me>
| Date:   28 hours ago
|
|     soft deps
|
* commit 8ae6653
| Author: Mathias Lecuyer <mathias@lecuyer.me>
| Date:   28 hours ago
|
|     soft deps
|
* commit 88e3547
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   28 hours ago
|
|     Adding handler for worker shut-down
|
* commit 08bb7d9
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   28 hours ago
|
|     semantic analysis for map second and third argument
|
*   commit 6f5ca7d
|\  Merge: 820f849 5e27c42
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   29 hours ago
| |
| |     start to support soft deps
| |
| * commit 5e27c42
| | Author: arunswaminathan <arun-swaminathan@hotmail.com>
| | Date:   29 hours ago
| |
| |     Basic test scripts
| |
| * commit ce05880
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   29 hours ago
| |
| |     Adding service host:port support
| |
* | commit 820f849
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   29 hours ago
| |
| |     start to support soft deps
| |
* | commit de7c2ae
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   29 hours ago
| |
| |     start to support soft deps
| |
* | commit 9a209ca
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   30 hours ago
| |
| |     misc
| |
* | commit a1b1aef
|/  Author: Mathias Lecuyer <mathias@lecuyer.me>
|   Date:   30 hours ago
|
|       added a requirements file for dependencies
|
* commit 758bed6
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   29 hours ago
|
|     semantic for second argument of reduce
|
* commit 4aa5766
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   29 hours ago
|
|     semantic for first argument of reduce
|
* commit 277df12
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   29 hours ago
|
|     Updating tests
|
* commit 42bf751
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   29 hours ago
|
|     Kill GlobalQueue; if runtime of a worker only redis matters
|
* commit 9d3d151
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   29 hours ago
|
|     BugFix: temp script file must have execute permissions
|
*   commit 00db440
|\  Merge: 70de0e0 fb25ec0
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   29 hours ago
| |
| |     Merge branch 'maestro' of github.com:Mathias Lecuyer/plt into maestro
| |
| * commit fb25ec0
| | Author: arunswaminathan <arun-swaminathan@hotmail.com>
| | Date:   29 hours ago
| |
| |     bug fix
| |
| * commit a2d283c
| | Author: arunswaminathan <arun-swaminathan@hotmail.com>
| | Date:   30 hours ago
| |
| |     Dependencies test
| |
| * commit df49039
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   30 hours ago
| |
| |     fix for the range argument
| |
| * commit 58de994
| | Author: arunswaminathan <arun-swaminathan@hotmail.com>
| | Date:   30 hours ago
| |
| |     changed execution path to ./maestro
| |
* | commit 70de0e0
|/  Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
|   Date:   29 hours ago
|
|       BugFix: execute scripts other than bash
|
* commit 49bf14b
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   30 hours ago
|
|     adding line in error messages
|
* commit 9a24ae0
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   30 hours ago
|
|     adding lines in nodes for error messages
|
*   commit ac6821c
|\  Merge: 9d3b421 8731719
| | Author: arunswaminathan <arun-swaminathan@hotmail.com>
| | Date:   30 hours ago
| |
| |     Merge branch 'maestro' of https://github.com/Mathias Lecuyer/plt into maestro
| |
| |     Conflicts:
| |      src/my_file.ms
| |
| * commit 8731719
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   30 hours ago
| |
| |     return right val for worker
| |
| * commit 706521c
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   31 hours ago
| |
| |     Invoking worker constructor from pipeline
| |
| * commit 72039ce
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   32 hours ago
| |
| |     Updating backend
| |
| * commit 6f2a291
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   33 hours ago
| |
| |     Merging stuff
| |
| * commit b2f9a7e
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   32 hours ago
| |
| |     with master and worker skeleton
| |
| * commit 5dcc74b
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   32 hours ago
| |
| |     adding range semantics and more detailed parsing of tree
| |
| * commit 294fa5e
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   32 hours ago
| |
| |     minor fix im myacc
| |
| * commit a50e0ae
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   32 hours ago
| |
| |     scripts for map red with jobs
| |
| * commit be11ad5
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   32 hours ago
| |
| |     map reduce with cut job
| |
| * commit d824368
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   33 hours ago
| |
| |     map reduce from list
| |
| * commit 317ae64
| | Author: Mathias Lecuyer <mathias@lecuyer.me>
| | Date:   33 hours ago
| |
| |     map reduce from list
| |
| * commit 2f5b535
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   34 hours ago
| |
| |     adding an if for leafs (int, string)
| |
| * commit 4a73bf1
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   34 hours ago
| |
| |     changing importing files for repl to work
| |
| * commit a822edc
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   2 days ago
| |
| |     no math operations for type job
| |
| * commit a53f0f6
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   2 days ago
| |
| |     undefined variables
| |
| * commit c35de06
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   2 days ago
| |
| |     removing print
| |
| * commit 3477cf9
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   2 days ago
| |
| |     having the variables in a list
| |
| * commit 4328ef6
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     map reduce
| |
| * commit 9295ff3
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     map reduce
| |
| * commit a089bc5
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     small fix on job deps args
| |
| * commit 00aaadd
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     small fix on job deps args
| |
| * commit 2a0eb61
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     small fix on job deps args
| |
| * commit 736e0f7
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     small change in jobs for map reduce
| |
| * commit 5428d33
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     big refactor to get types at runtime
| |
| * commit 444022d
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     with list creation
| |
| * commit d2c9b7d
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     with list creation
| |
| * commit 8d146fa
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     fixed the list issue
| |
| * commit 2284352
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   2 days ago
| |
| |     stuff
| |
| *   commit c27681a
| |\  Merge: 6d65531 dea9ff0
| | | Author: ren <ren@ren-K55A.(none)>
| | | Date:   2 days ago
| | |
| | |     Merge branch ’maestro’ of https://github.com/Mathias Lecuyer/plt into maestr
| | |
| | * commit dea9ff0
| | | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | | Date:   2 days ago
| | |
| | |     adding some more for the semantic analysis
| | |
| | * commit bb45641
| | | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | | Date:   2 days ago
| | |
| | |     error changes for operator
| | |
| | * commit fc39109
| | | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | | Date:   2 days ago
| | |
| | |     Fix annoying list-flattening related bug
| | |
| | * commit c1b373e
| | | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | | Date:   2 days ago
| | |
| | |     removed stupid prints
| | |
| | * commit f48cbdd
| | | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | | Date:   2 days ago
| | |
| | |     fixed embeded list problem in run
| | |
| | * commit 778c213
| | | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | | Date:   2 days ago
| | |
| | |     with minus bug fixed
| | |
| * | commit 6d65531
| |/  Author: ren <ren@ren-K55A.(none)>
| |   Date:   2 days ago
| |
| |       some mapreduce stuff
| |
* | commit 9d3b421
|/  Author: arunswaminathan <arun-swaminathan@hotmail.com>
|   Date:   30 hours ago
|
|       Updates tests and framework
|
* commit f0a979d
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     loops on lists :)
|
* commit ec3a384
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     changing tests
|
* commit 4a7f2b6
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     adding semantic for -> and <->
|
* commit b237a91
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     adding more checks
|
* commit 661209b
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   3 days ago
|
|     Print some stats while running jobs
|
* commit 03e9f7e
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     with / %
|
* commit e3a1c96
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     with -
|
* commit 7c1b68f
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     added + operators
|
* commit 81269e0
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   3 days ago
|
|     Expose Wait constructor for fake objects
|
|     It must be called with an integer value. Currently, it doesn't work
|     because the translation file invoke it with wrong arg type.
|
* commit d86b83c
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     changing tests
|
* commit 73e8eae
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   3 days ago
|
|     Log files under cwd
|
* commit a3d9e3e
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   3 days ago
|
|     Remove finalreport folder from this branch
|
* commit 3172fe1
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   3 days ago
|
|     BugFix: In job construct and run. Arguments are now properly passed
|
* commit 773264d
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     join is in string library not in list
|
* commit 2af9584
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     no prints
|
* commit 1581c9b
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     striped strings
|
* commit e93aca8
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     stuff
|
* commit 5fe4356
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     a simple test file
|
* commit 49c5c84
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     some fixes
|
* commit 68835af
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     more functions
|
*   commit f07d56d
|\  Merge: b212555 4c837bc
```

```
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   3 days ago
| |
| |     Merge branch 'maestro' of github.com:Mathias Lecuyer/plt into maestro
| |
| * commit 4c837bc
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   3 days ago
| |
| |     added comma
| |
| *   commit 63842c3
| |\  Merge: 66cc0b3 6b465a3
| | | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | | Date:   3 days ago
| | |
| | |     stuff
| | |
| * | commit 66cc0b3
| | | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | | Date:   3 days ago
| | |
| | |     misc
| | |
* | | commit b212555
| |/  Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
|/|   Date:   3 days ago
| |
| |       put back networkx import, mistakenly removed
| |
* | commit 6b465a3
|/  Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
|   Date:   3 days ago
|
|       Adding suport for log files
|
* commit 802230d
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     with ast and translation, huge refactor
|
* commit 806ac72
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   3 days ago
|
|     Added checks to correctness of executed job
|
|     Read output from CheckFile after job execution and compare to expected
|     output
|
* commit 1b5cb6f
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   3 days ago
|
|     Edited to write output to CheckFile
|
* commit 384fb02
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   3 days ago
|
|     Check file
|
|     File for checking correctness of job
|
* commit cd9da27
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     with job type
|
* commit a11566c
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     with job type
|
* commit 79ccace
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     adding some semantic checks
|
* commit 8f43c14
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     adding swap in gitignore
|
* commit f5fc281
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   3 days ago
|
|     changing the job initialization
|
* commit 79afc6b
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     ast with nodes
|
* commit 9be7503
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     real prgn and struct for semantic analysis and translation
|
* commit eecbc24
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     real prgn and struct for semantic analysis and translation
|
* commit a624554
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     added notion of a program
|
* commit 61b115f
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   3 days ago
|
|     added comments
|
* commit 22153e9
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   4 days ago
|
|     Structure and more for final report
|
* commit 41f9bb0
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   4 days ago
|
|     started error checking
|
* commit 7b7e7d2
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   4 days ago
|
|     bin for calling maestro
|
* commit fbbbb6b
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   4 days ago
|
|     refactor
|
* commit d3f129e
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   4 days ago
|
|     cannot create job inside run anymore
|
* commit fff6393
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   4 days ago
|
|     adding some output
|
* commit 1028e2e
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   5 days ago
|
|     Bug fix
|
* commit 6236dc3
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   2 weeks ago
|
|     Add Makefile
|
|     Just type make to run all tests
|
* commit df5695e
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   3 weeks ago
|
|     BugFix: Exit main upon Exception
|
* commit 848a332
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   4 weeks ago
|
|     Changes to headers
|
* commit afd5566
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   4 weeks ago
|
|     Bash script for test framework
|
|     Do not edit.
|
* commit cb16ffc
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   4 weeks ago
|
|     Test framework
|
|     Skeleton ready. Work in progress.
|
* commit f281c92
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   4 weeks ago
|
|     minors
|
* commit 633a705
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   4 weeks ago
|
|     minor edit
|
*   commit 40e3fc6
|\  Merge: 466c5f7 b47784e
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   5 weeks ago
| |
| |     merged
| |
| * commit b47784e
| | Author: ren <ren@ren-K55A.(none)>
| | Date:   5 weeks ago
| |
| |     some unix cleanup
| |
* | commit 466c5f7
|/  Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
|   Date:   5 weeks ago
```

```
|
|       fixed on the fly run bug
|
* commit fcea84c
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   5 weeks ago
|
|     AST, still value on the fly too, with a few bugs when calling the backend functi
|
*   commit 5852477
|\  Merge: 607867f 8862105
| | Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| | Date:   5 weeks ago
| |
| |     AST in work, doesn't work yet
| |
| * commit 8862105
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   5 weeks ago
| |
| |     adding example for circular dependencies
| |
| * commit ca2cc84
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   5 weeks ago
| |
| |     adding the circular dependencies
| |
| * commit b632296
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   5 weeks ago
| |
| |     pep8 for the jobs.py
| |
| * commit 577df19
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   5 weeks ago
| |
| |     adding a smple for finding cycles in a graph
| |
| * commit 2876568
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   5 weeks ago
| |
| |     addintion to skip the blank lines
| |
| * commit 5359ddf
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   6 weeks ago
| |
| |     revised file handling
| |
| * commit 2d1b47f
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   6 weeks ago
| |
| |     Revert "Temporary workaround for the bug we encounter with the TA - This is to
| |
| |     This reverts commit 6261910c13af4b3b67def69e46cf32daddb26248.
| |
| * commit 6261910
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   6 weeks ago
| |
| |     Temporary workaround for the bug we encounter with the TA
| |
| * commit 14bdbbd
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   6 weeks ago
| |
| |     pep8 for myacc.py
| |
| * commit b3edf77
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   6 weeks ago
| |
| |     adding file parsing
| |
| * commit 50e62ae
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   6 weeks ago
| |
| |     adding demo file
| |
| * commit 982d532
| | Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| | Date:   6 weeks ago
| |
| |     new command line I have found on the net
| |
| * commit aa25b20
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   6 weeks ago
| |
| |     Remove diagnostic messages; code is good to go for now
| |
| * commit d246527
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   6 weeks ago
| |
| |     Add a bash demo script
| |
| * commit 67956e8
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   6 weeks ago
| |
| |     Add "+x" to ruby script
| |
| * commit 8f8a00b
| | Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| | Date:   6 weeks ago
| |
| |     Do not use this commit. It partialy fixes a bug
| |
* | commit 607867f
|/  Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
|   Date:   5 weeks ago
|
|       begin AST
|
* commit 38cba97
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   6 weeks ago
|
|     test script
|
* commit 402c68b
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   7 weeks ago
|
|     Paramiko instead of rsh; Arun will have this task
|
* commit 30f3b15
| Author: Georgios Koloventzos <g.koloventzos@gmail.com>
| Date:   7 weeks ago
|
|     changing the name of the job
|
* commit 5500054
| Author: arunswaminathan <arun-swaminathan@hotmail.com>
| Date:   7 weeks ago
|
|     Update jobs.py
|
|     Changed subprocess Popen to run on remote shell (rsh) with local IP address and
|
* commit c0eeb67
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   8 weeks ago
|
|     added some comments to explain
|
* commit e0c8903
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   8 weeks ago
|
|     basic parser working with embeded action, AST structure reeady but not needed/us
|
* commit 6fa1513
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   8 weeks ago
|
|     fixed a typo
|
* commit bf5e9f0
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   8 weeks ago
|
|     setup.py should not be at the src folder
```

```
|
* commit 6580926
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   8 weeks ago
|
|     Minor update - clean up of helpers
|
* commit 35adc72
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   8 weeks ago
|
|     misc
|
* commit bbecf0c
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   8 weeks ago
|
|     parser parses pretty much all the logic, no AST yet
|
* commit 2bca7b1
| Author: ren <ren@ren-K55A.(none)>
| Date:   8 weeks ago
|
|     adding testscripts
|
* commit 1eb6361
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   8 weeks ago
|
|     Restucture of directory and adding some tests
|
* commit 1752d49
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   8 weeks ago
|
|     Adding Queue class
|
|     The idea is that whebever run is invoked, a  class queue
|     is created and job names specified in run are enqueued there.
|     The jobs dedicated to run are previously constructed, thus
|     we only need to add their dependencies, before enqueue them.
|
* commit 04f9d77
| Author: Vaggelis Atlidakis <vatlidak@cs.columbia.edu>
| Date:   8 weeks ago
|
|     Initial commit of semantic actions implementation
|
|     TO DO:
|     add some test
|
* commit 4ab8d61
| Author: Mathias Lecuyer <mathias.lecuyer@gmail.com>
| Date:   9 weeks ago
|
|     first version of lexer and parser
```

## 10.2 Source Code

### REPL (George)

```
import sys
from maestro_cmd import Console
from myacc import *
import threading
from helpers import jobqueue

if __name__ == '__main__':
    threading.Thread(target=jobqueue.GlobalJobQueue.poll_for_jobs).start()
    if len(sys.argv) == 1:
        console = Console(parser)  # while True:
        console.cmdloop()  # try:
    elif len(sys.argv) == 2:
        try:
            f = open(sys.argv[1])
        except IOError:
            print 'cannot open', sys.argv[1]
            sys.exit(-1)
        prgm = f.read()
        result = pipeline(prgm)
        print result
        f.close()
    else:
        print "Usage: python myacc.py <file_name>"
    jobqueue.GlobalJobQueue.stop()


## console.py
## Author:   James Thiele
## Date:     27 April 2004
## Version:  1.0
## Location: http://www.eskimo.com/~jet/python/examples/cmd/
## Copyright (c) 2004, James Thiele

import os
import cmd
import readline
from myacc import *

class Console(cmd.Cmd):

    def __init__(self, pars):
        cmd.Cmd.__init__(self)
        self.prompt = "maestro> "
        self.parser = pars

    ## Command definitions ##
    def do_hist(self, args):
        """Print a list of commands that have been entered"""
        print self._hist

    def do_exit(self, args):
        """Exits from the console"""
        return -1

    ## Command definitions to support Cmd object functionality ##
    def do_EOF(self, args):
        """Exit on system end of file character"""
        return self.do_exit(args)

    def do_help(self, args):
        """Get help on commands
           'help' or '?' with no arguments prints a list of commands for which
           help is available 'help <command>' or
           '? <command>' gives help on <command>
        """
        ## The only reason to define this method is for the help text in the
        ## doc string
        cmd.Cmd.do_help(self, args)

    ## Override methods in Cmd object ##
    def preloop(self):
        """Initialization before prompting user for commands.
           Despite the claims in the Cmd documentaion, Cmd.preloop()
           is not a stub.
        """
        cmd.Cmd.preloop(self)   ## sets up command completion
        self._hist    = []      ## No history yet
        self._locals  = {}      ## Initialize execution namespace for user
        self._globals = {}

    def postloop(self):
        """Take care of any unfinished business.
           Despite the claims in the Cmd documentaion,
           Cmd.postloop() is not a stub.
        """
        cmd.Cmd.postloop(self)   ## Clean up command completion
        print "Exiting..."

    def precmd(self, line):
        """ This method is called after the line has been input but before
            it has been interpreted. If you want to modifdy the input line
            before execution (for example, variable substitution) do it here.
        """
        self._hist += [ line.strip() ]
        return line

    def postcmd(self, stop, line):
        """If you want to stop the console, return something that evaluates
        to true.
           If you want to do some post command processing, do it here.
        """
        return stop

    def emptyline(self):
        """Do nothing on empty input line"""
        pass

    def default(self, line):
        """Called on an input line when the command prefix is not recognized.
           In that case we execute the line as Python code.
        """
        try:
            result = pipeline(line)
#            result = pipeline(line)
#           exec(line) in self._locals, self._globals

        except Exception, e:
            print e.__class__, ":", e
```

## Lexer (Mathias)

```
import ply.lex as lex
# List of token names.   This is always required
tokens = (
   'ID',
   'INT',
   'DEP',
   'NODEP',
   'SOFTPDEP',
   'SOFTNDEP',
   'SOFTNODEP',
   'LP',
   'RP',
   'LB',
   'RB',
   'LC',
   'RC',
   'ASSIGN',
   'COMMA',
   'STR',
   'SC',
   'ADDOP',
   'MULOP',
   'EACH',
)
# Regular expression rules for simple tokens
t_NODEP     = r'<->'
t_SOFTPDEP   = r'~>'
t_SOFTNDEP   = r'~<'
t_SOFTNODEP = r'<~>'
t_LP        = r'\('
t_RP        = r'\)'
t_LB        = r'\['
t_RB        = r'\]'
t_LC        = r'{'
t_RC        = r'}'
t_ASSIGN    = r'            = '
t_STR       = r'"[^"]+"'
t_SC        = r';'
t_COMMA     = r','
t_INT       = r'[0-9]+'
t_ADDOP     = r'[+\-]'
t_MULOP     = r'[/*%]'
t_EACH      = r'\.each'

def t_DEP(t):
    r'->'
    return t

# A regular expression rule with some action code
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_\d]*'
    return t

# remove comments
def t_COMMENT(t):
    r'(//|\#).*'
    pass

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

## Parser (Mathias, George)

```
import sys
import ply.yacc as yacc
import pipeline.semantic_analysis as sa
import pipeline.translation as t

# Get the token map from the lexer
from mlex import tokens
precedence = (
    ('left', 'ASSIGN'),
    ('left', 'ADDOP', 'DEP', 'SOFTPDEP', 'SOFTNDEP'),
    ('left', 'MULOP', 'NODEP', 'SOFTNODEP'),
)

lines = 0

def p_program(p):
    'PRGM : STMTLIST'
    line = p.lineno(1)
    node = Node('prgm', [p[1].node], line=line)
    p[0] = AST_obj(node)

def p_stmt_list(p):
    '''STMTLIST : STMTLIST STMT
                | STMT'''
    line = p.lineno(1)
    if len(p) == 2:
        node = Node('stmt-list', [p[1].node], line=line)
    else:
        node = Node('stmt-list', [p[1].node, p[2].node], line=line)
    p[0] = AST_obj(node)

def p_stmt(p):
    '''STMT : E SC
            | SC'''
    p[0] = p[1]

def p_stmt_error(p):
    'STMT : error'
    #line = p.lineno(0) # line number of error
    #print "Syntax error in statement line " + str(line)

def p_stmt_block(p):
    'STMTBLOCK : LC STMTLIST RC'
    p[0] = p[2]

def p_list_loop(p):
    'STMT : E EACH LP ID RP STMTBLOCK'
    line = p.lineno(1)
    _type = 'list'
    id_node = Node('id', [], 'mut', value=p[4], leaf=True, line=line)
    node = Node('list-loop', [p[1].node, id_node, p[6].node], _type, line=line)
    p[0] = AST_obj(node)

# LII is a comma separated list of Expressions
def p_func_call(p):
    'E : ID LP LII RP'
    line = p.lineno(1)
    _type = type_for_func(p[1])
    node = Node(p[1], [p[3].node], _type, line=line)
    p[0] = AST_obj(node)

def p_func_call_error(p):
    'E : ID LP error RP'
    line = p.lineno(1)  # line number of error
    print "Syntax error in function call, line ", line

# assign a variable:
# - put the name in the sym_table
# - the expresion gets the value for 1 liners
def p_assign(p):
    'E : ID ASSIGN E'
    line = p.lineno(1)
    _type = p[3].node._type
    sym_table[p[1]] = [None, _type]
    node = Node('=', [p[1], p[3].node], _type, line=line)
    p[0] = AST_obj(node)

# strings for Job names
def p_e_str(p):
    'E : STR'
    line = p.lineno(1)
    _type = 'string'
    node = Node('str', [], _type, value=str(p[1][1:-1]), leaf=True, line=line)
    p[0] = AST_obj(node)

def p_e_int(p):
    'E : INT'
    line = p.lineno(1)
    _type = 'int'
    node = Node('int', [], _type, value=int(p[1]), leaf=True, line=line)
    p[0] = AST_obj(node)
```

```
def p_math_op(p):
    '''E : E MULOP E
         | E ADDOP E'''
    line = p.lineno(1)
    _type = type_for_op(p[1].node._type, p[3].node._type, p[2])
    if _type == None:
        line = p.lineno(2)
        print "Wrong type for mathematical operator in line " + str(line)
        raise SyntaxError
    node = Node(p[2], [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)

# lists
def p_e_list(p):
    'E : LI'
    p[0] = p[1]

def p_list(p):
    'LI : LB LII RB'
    p[0] = p[2]
    line = p.lineno(1)
    _type = 'list'
    node = Node('list', [p[2].node], _type, line=line)
    p[0] = AST_obj(node)

# arguments of a function or inside of a list for later
def p_list_inside_grow(p):
    'LII : LII COMMA E'
    _type = 'list'
    line = p.lineno(1)
    node = Node('list-concat', [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)

def p_list_inside_orig(p):
    'LII : E'
    _type = 'list'
    line = p.lineno(1)
    node = Node('list-orig', [p[1].node], _type, line=line)
    p[0] = AST_obj(node)

# <->
def p_e_nodep(p):
    'E : E NODEP E'
    line = p.lineno(2)
    _type = 'list'
    node = Node('<->', [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)

# ->
def p_e_dep(p):
    'E : E DEP E'
    _type = 'list'
    line = p.lineno(2)
    node = Node('->', [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)
```

```
# ~>
def p_e_softpdep(p):
    'E : E SOFTPDEP E'
    _type = 'list'
    line = p.lineno(2)
    node = Node('~>', [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)

# ~>
def p_e_softndep(p):
    'E : E SOFTNDEP E'
    _type = 'list'
    line = p.lineno(2)
    node = Node('~<', [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)

# <~>
def p_e_softnodep(p):
    'E : E SOFTNODEP E'
    _type = 'list'
    line = p.lineno(2)
    node = Node('<~>', [p[1].node, p[3].node], _type, line=line)
    p[0] = AST_obj(node)

# ()
def p_e_parenthesize(p):
    'E : LP E RP'
    p[0] = p[2]

# that's a variable: fetch it in the symbol table
def p_e_id(p):
    'E : ID'
    line = p.lineno(1)
    try:
        _type = sym_table[p[1]][-1]
    except:
        print "Undefined variable " + p[1] + " at line " + str(line)
        raise SyntaxError
    node = Node('id', [], _type, value=p[1], leaf=True, line=line)
    p[0] = AST_obj(node)

# Error rule for syntax errors
def p_error(p):
    print "Syntax error in input: " + str(p)

#type helpers
def type_for_func(name):
    if name == 'Job' or name == 'Wait':
        return 'job'
    elif name in ['run', 'range', 'map', 'reduce']:
        return 'list'

def type_for_op(type1, type2, op):
    if type1 == "job" or type2 == "job":
        print "No mathematic operations for type jobs"
        return None
    if op == '+':
        return type_for_sum(type1, type2)
    elif type1 == type2 == 'int':
        return type1
    else:
        return None
#        raise SyntaxError

def type_for_sum(type1, type2):
    if (type1 == 'int' and type2 == 'string') \
            or (type2 == 'int' and type1 == 'string'):
        return 'string'
    if type1 == type2 == 'int':
        return type1
    return None
#    raise SyntaxError

# Symbol table
sym_table = {}  # map[symbol][value, type]

# AST node structure
class Node:
    def __init__(self, operation, children=None,  \
                       _type=None, value=None, leaf=False, line=-1):
        self._type = _type
        self.operation = operation
        if children:
            self.children = children
        else:
            self.children = []
        self.leaf = leaf
        self.value = value
        self.line = line

# we add one layer of abstraction to be able to get values and syblings
# on top of node
class AST_obj:
    def __init__(self, node=None, value=None, syblings=None):
        self.node = node
        self.syblings = syblings

# Build the parser
parser = yacc.yacc()

# pipeline for execution
def pipeline(code):
    try:
        astree = parser.parse(code)
        if astree == None:
            return None
        ast = astree.node
    except:
        return None
    #sa.traverse(ast)
    sem = sa.analyse(ast)
    if sem == None:
        return "Semantic error. See above!"
    result = t.execute(ast, sym_table)
    # return result
```

## Test Framework (Arun, Yiren)

```
#!/usr/bin/python
#Usage "python test_framework.py <1,2...>"  or "python test_Framework.py all"
import sys
from subprocess import Popen, PIPE, STDOUT
from colorama import init,Fore, Back, Style
init()
global output
output= ""
global failCount
failCount = 0
files=['syntax_err','imbalanced_parenthesis','undeclared_job',
       'undeclared_dependency','my_file','single_run','multiple_run',
       'multiple_declaration','multiple_declaration2', 'hard_dependency',
       'hard_dependency2', 'circular_dependency', 'self_dependency',
       'long_dependency','mr_job']
flag=['circular_dependency','self_dependency','syntax_err',
      'imbalanced_parenthesis','undeclared_job','undeclared_dependency']
#def parseErr(stderr):
# global failCount
# if not stderr:
# print (Fore.GREEN + "Maestro Program Passed")
# else:
# print (Fore.RED + "Maestro Program Failed")
# failCount = failCount + 1
# print "*************************"
def parseOut(stdout, filename):
global failCount
global output
with open ("TestsOutput/check.txt", "r") as myfile:
data=myfile.read().replace('\n', '')

if "Illegal" in stdout and flag.count(filename)==0:
print (Fore.RED + "Test Failed - Illegal token resulting in syntax error")
failCount = failCount + 1
with open ("tests/new_tests/log.txt", "a") as myfile:
myfile.write(filename+ ".ms     -Failed"+"\n")
myfile.write("********** \n")
myfile.write(stdout+"\n")
elif "Syntax error" in stdout and flag.count(filename)==0:
failCount = failCount + 1
print (Fore.RED + "Test Failed - Syntax error in input")
with open ("tests/new_tests/log.txt", "a") as myfile:
myfile.write(filename+ ".ms     -Failed"+"\n")
myfile.write("********** \n")
myfile.write(stdout+"\n")
elif "Undefined" in stdout and flag.count(filename)==0:
failCount = failCount + 1
print (Fore.RED + "Test Failed - Undefined variable in input")
with open ("tests/new_tests/log.txt", "a") as myfile:
myfile.write(filename+ ".ms     -Failed"+"\n")
myfile.write("********** \n")
myfile.write(stdout+"\n")
elif "circular" in stdout and flag.count(filename)==0:
failCount = failCount + 1
print (Fore.RED + "Test Failed - Circular dependency in input")
with open ("tests/new_tests/log.txt", "a") as myfile:
myfile.write(filename+ ".ms    -Failed"+"\n")
myfile.write("********** \n")
myfile.write(stdout+"\n")

elif (data!="check") and flag.count(filename)==0:
failCount=failCount+1
print (Fore.RED + "Test Failed - Expected output not found at file")
print "*************************"
print Fore.BLACK + "Expected Output: check"+" \nOutput read at file: " + data
print "\n"
with open ("tests/new_tests/log.txt", "a") as myfile:
myfile.write(filename+ ".ms   -Failed"+"\n")
myfile.write(stdout+"\n")


else:
print (Fore.GREEN + "Test Passed!")
with open ("tests/new_tests/log.txt", "a") as myfile:
myfile.write(filename+ ".ms   -Passed"+"\n")
myfile.write("********** \n")
myfile.write(stdout+"\n")
with open ("TestsOutput/check.txt", "w") as myfile:
myfile.write(" ")
if len(sys.argv)!= 2:
print "Wrong input. Usage: 'python test_bash.py <1,2,...>' or 'python test_bash.py all
elif sys.argv[1] == 'all':
with open ("tests/new_tests/log.txt", "w") as myfile:
myfile.write("")
for i in range(0,15):
output = str(i)
#oldCount = failCount
cmd = './maestro tests/new_tests/'+files[i]+'.ms'
p = Popen(cmd, shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
stdout, stderr = p.communicate()
print (Fore.BLUE + "------------------------------")
print "TEST "+files[i]
print "------------------------------"
#if stderr:
#print (Fore.BLACK + stderr)
#parseErr(stderr)
if stdout:
print (Fore.YELLOW + stdout)
parseOut(stdout, files[i])
else:
print (Fore.YELLOW + "Empty Output")
print ""
print (Fore.BLUE+"--------------------------------------------------------------------
print (Fore.GREEN+"\n****************************")
print ("Tests Passed: "+str(15-failCount)+" / 15")
print ("Check log for test details")
print ("****************************")
#if failCount-oldCount == 2:
#failCount = failCount - 1
#failCount = failCount - 1
#passCount = 77 - failCount
#print (Fore.BLUE + "*************************")
#print (Fore.BLUE + "Tests Passed : " + str(passCount)+"/77")
#print "*************************"
else:
with open ("tests/new_tests/log.txt", "w") as myfile:
myfile.write("")
output = sys.argv[1]
cmd = './maestro tests/new_tests/'+sys.argv[1]+'.ms'
p = Popen(cmd, shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
stdout, stderr = p.communicate()
#print "stdout is "+stdout
print (Fore.BLUE + "-------------------------------")
print "TEST "+sys.argv[1]+".ms"
print "-----------------------------"
print (Fore.BLACK + "")
#if stderr:
#print (Fore.BLACK + stderr)
#parseErr(stderr)
#print "error is "+errno
if stdout:
print (Fore.YELLOW + stdout)
parseOut(stdout, sys.argv[1])
else:
print (Fore.BLACK + "Empty Output")
print ""
```

## Semantic Analysis (George)

```
assign = []

def analyse(ast):
    error = {'->': "dependency operator",
            '<->': "non dependency operator",
            '-': "minus operator",
            '+': "plus operator"}
    new = type(ast)
    if new is str or new is list:
        return str(new)
    if ast.operation == "id":
        if ast.value not in assign:
            print "Variable " + ast.value + " not previously declared\
                     at line " + str(ast.line)
            return None
        return ast._type
```

```
#for being dynamic
if ast.operation == "=":
    assign.append(ast.children[0])
    typex = "assign"
    for node in ast.children:
        typex = analyse(node)
        if typex == None:
            break
    return typex
#if -> or <->
if ast.operation == '<->' or ast.operation == '->' \
   or ast.operation == '<~>' or ast.operation == '~>' \
    or ast.operation == '~<':
    #find type of first child
    type1 = analyse(ast.children[0])
    #find of second
    type2 = analyse(ast.children[1])
    #if both the same and job return job
    if type1 == type2 and type1 == "job":
        return type1
    else:
        print "Type error(sem) " + xstr(type1) + " " \
                + ast.operation + " " + xstr(type2) + \
                " at line " + str(ast.line)
        return None
if ast.operation == '-' or ast.operation == '/':
    type1 = analyse(ast.children[0])
    type2 = analyse(ast.children[1])
    if type1 == type2 and type1 == "int":
        return type1
    else:
        print "minus error"
        return None
if ast.operation == "range":
    child = ast.children[0].children[0]
    type1 = analyse(child)
    if type1 != "int":
        print "Function range needs int as argument got " + type1 \
                + " at line " + str(ast.line)
        return None
    return type1
if ast.operation == "reduce":
    child_type = ast.children[0].children[0].children[0]._type
    if child_type != "list":
        print "Function reduce needs list as first argument got " + \
                child_type + " at line " + str(ast.line)
        return None
    child_type = ast.children[0].children[1]._type
    if child_type != "string":
        print "Function reduce needs string as second argument got " + \
                child_type + " at line " + str(ast.line)
        return None
    return "list"
if ast.operation == "map":
    child_type = ast.children[0].children[1]._type
    if child_type != "int":
        print "Function map needs int as first argument got " + child_type\
                    + " at line " + str(ast.line)
            return None
        child_type = ast.children[0].children[0].children[1]._type
        if child_type != "string":
            print "Function map needs string as second argument got " + \
                    child_type + " at line " + str(ast.line)
            return None
        return child_type
    if ast.operation == "Job":
        leaf = []
        analyse_leafs(ast, leaf)
        if sorted(leaf)[0] != sorted(leaf)[-1]:
            print "Job argument not a string at line " + str(ast.line)
            return None
        return ast._type
    #for leaf int, str,....
    if not ast.children:
        return ast._type
    for node in ast.children:
        typex = analyse(node)
        if typex == None:
            break
    return typex

def analyse_leafs(node, leafs):
    if node.leaf:
        leafs.append(node._type)
    else:
        for n in node.children:
            analyse_leafs(n, leafs)

def xstr(s):
    if s is None:
        return 'None'
    return str(s)

def traverse(ast, level=0):
    if type(ast) is str:
        print "\t" * level + ast
        return
#    if ast.operation == 'id':
#        print "\t" * level + xstr(ast._type) + " " + xstr(ast.value)
    if ast.operation == "=":
        print "\t" * level + xstr(ast._type) + " " + xstr(ast.operation) \
                    + " " + xstr(ast.value) + " " + xstr(ast.leaf)
    else:
        print "\t" * level + xstr(ast._type) + " " + xstr(ast.operation) \
                    + " " + xstr(ast.value) + " " + xstr(ast.leaf)
    for node in ast.children:
#        print node
#        for l in node:
        traverse(node, level + 1)
```

## Semantic Actions (Mathias)

```
import helpers.jobs as hj
import helpers.workers as hw

# recursive function to translate and execute the AST
```

```
def execute(ast, sym_table):
    op = ast.operation
    # if we have a leaf, just return the value associated
    if ast.leaf:
        if op == 'id':
            return sym_table[ast.value]
        else:
            return [ast.value, ast._type]

    # if it's not a leaf, we need to translate the op behavior and execute it
    if op == 'prgm':
        val = execute(ast.children[0], sym_table)
        ast.value = val[0]
        return val
    elif op == 'stmt-list':
        children_exec = [execute(c, sym_table) for c in ast.children]
        val = children_exec[-1]
        ast.value = val[0]
        return val
    elif op == 'Job':  # all args should be strings
        children_exec = [execute(c, sym_table) for c in ast.children][0]
        args = [c[0] for c in children_exec[0]]
        ast.value = hj.Job(args[0], args[1:])
        return [ast.value, ast._type]
    elif op == 'worker':
        args = execute(ast.children[0], sym_table)
        addr = args[0][0][0]
        ast.value = hw.worker(addr)
        return [ast.value, 'None']
    elif op == 'service':
        args = execute(ast.children[0], sym_table)
        addr = args[0][0][0]
        ast.value = hj.service(addr)
        return [ast.value, 'None']
    elif op == 'Wait':  # 1 int arg
        args = execute(ast.children[0], sym_table)
        time = str(args[0][0][0])
        ast.value = hj.Wait(time)
        return [ast.value, ast._type]
    elif op == 'run':
        children_exec = [execute(c, sym_table) for c in ast.children]
        # args = [c[0] for c in children_exec[0]]
        args = type_flatten(children_exec)
        ast.value = hj.run(args)
        return children_exec
    elif op == 'range':
        arg = execute(ast.children[0], sym_table)[0][0]  # first 0 for value,
                                                         # second because it's a list
        ast.value = [[x, 'int'] for x in range(arg[0])]
        return [ast.value, 'list']
    elif op == 'map':
        args = execute(ast.children[0], sym_table)[0]
        prior = args[0]
        map_script_path = args[1][0]
        map_workers = args[2][0]
        map_jobs = []
        if prior[1] == 'job':
            cut_job = prior[0]
            for i in range(map_workers):
                m = hj.Job(map_script_path, deps_jobs=[cut_job], \
                        deps_args=(lambda x: [x[i]]))
                dep([m], [cut_job])
                map_jobs.append([m, 'job'])
        elif prior[1] == 'list':
            for arg in prior[0]:
                m = hj.Job(map_script_path, [arg[0]])
                map_jobs.append([m, 'job'])
        ast.value = [mj[0] for mj in map_jobs]
        return [map_jobs, 'list']
    elif op == 'reduce':
        args = execute(ast.children[0], sym_table)[0]
        priors = type_flatten([args[0]])
        reduce_script_path = args[1][0]
        m = hj.Job(reduce_script_path, deps_jobs=priors, \
                deps_args=(lambda x: x))
        dep([m], priors)
        return [[[m, 'job']], 'list']
    elif op == 'list-loop':
        var = ast.children[1].value
        l = execute(ast.children[0], sym_table)[0]
        for x in l:
            sym_table[var] = x  # x is [value, type]
            execute(ast.children[2], sym_table)
        return [l, 'list']
    elif op == '=':
        children_exec = execute(ast.children[-1], sym_table)
        sym_table[ast.children[0]] = children_exec  # the exec is [value, type]
        ast.value = children_exec[0]
        return children_exec
    elif op == 'list':
        children_exec = [execute(c, sym_table) for c in ast.children]
        ast.value = children_exec[0][0]
        return children_exec[0]
    elif op == 'list-concat':
        children_exec = [execute(c, sym_table) for c in ast.children]
        ast.value = children_exec[0][0] + [children_exec[-1]]
        return [ast.value, ast._type]
    elif op == 'list-orig':
        children_exec = [execute(c, sym_table) for c in ast.children][0]
        ast.value = children_exec
        return [[children_exec], 'list']
    elif op == '<->':
        children_exec = [type_flatten([execute(c, sym_table)]) \
                         for c in ast.children]
        ast.value = nodep(children_exec[0], children_exec[1])
        return [[[j, 'job'] for j in ast.value], 'list']
    elif op == '->':
        children_exec = [type_flatten([execute(c, sym_table)]) \
                         for c in ast.children]
        ast.value = dep(children_exec[1], children_exec[0])
        return [[[j, 'job'] for j in ast.value], 'list']
    elif op == '~>':
        children_exec = [type_flatten([execute(c, sym_table)]) \
                         for c in ast.children]
        ast.value = softpdep(children_exec[1], children_exec[0])
        return [[[j, 'job'] for j in ast.value], 'list']
    elif op == '~<':
        children_exec = [type_flatten([execute(c, sym_table)]) \
                         for c in ast.children]
        ast.value = softndep(children_exec[1], children_exec[0])
        return [[[j, 'job'] for j in ast.value], 'list']
    elif op == '<~>':
        children_exec = [type_flatten([execute(c, sym_table)]) \
                         for c in ast.children]
        ast.value = softnodep(children_exec[0], children_exec[1])
        return [[[j, 'job'] for j in ast.value], 'list']
    elif op == '+':
        children_exec = [execute(c, sym_table)[0] for c in ast.children]
        t1 = ast.children[0]._type
        t2 = ast.children[1]._type
        if t1 == 'string' and t2 == 'int':
            ast.value = children_exec[0] + str(children_exec[1])
        elif t2 == 'string' and t1 == 'int':
            ast.value = str(children_exec[0]) + children_exec[1]
        else:
            ast.value = reduce(lambda x,y: x+y, children_exec)
        return [ast.value, ast._type]
    elif op == '-':
        children_exec = [execute(c, sym_table)[0] for c in ast.children]
        ast.value = children_exec[0] - children_exec[1]
        return [ast.value, ast._type]
    elif op == '/':
        children_exec = [execute(c, sym_table)[0] for c in ast.children]
        ast.value = children_exec[0] / children_exec[1]
        return [ast.value, ast._type]
    elif op == '%':
        children_exec = [execute(c, sym_table)[0] for c in ast.children]
        ast.value = children_exec[0] % children_exec[1]
        return [ast.value, ast._type]
    else:
        return None

# flatten helper: flattens a list of embeded lists
def flatten(lst):
    return sum( ([x] if not isinstance(x, list) else flatten(x)
             for x in lst), [] )

def type_flatten(lst):
    return sum( [[x[0]] if not isinstance(x[0], list) else type_flatten(x[0])
             for x in lst], [] )

# Dependencies helper
def nodep(ljobs, rjobs):
    if type(ljobs) is not list:
        ljobs = [ljobs]
    if type(rjobs) is not list:
        rjobs = [rjobs]
    return flatten(ljobs + rjobs)
def dep(jobs, depend_on_jobs):
    if type(jobs) is not list:
        jobs = [jobs]
    if type(depend_on_jobs) is not list:
        depend_on_jobs = [depend_on_jobs]
    hj.add_dependencies(jobs, depend_on_jobs)
    return flatten(jobs + depend_on_jobs)
def softnodep(ljobs, rjobs):
    if type(ljobs) is not list:
        ljobs = [ljobs]
    if type(rjobs) is not list:
        rjobs = [rjobs]
    ljobs = flatten(ljobs)
    rjobs = flatten(rjobs)
    hj.add_soft_equal_dependencies(ljobs, rjobs)
    return flatten(ljobs + rjobs)
def softpdep(jobs, depend_on_jobs):
    if type(jobs) is not list:
        jobs = [jobs]
    if type(depend_on_jobs) is not list:
        depend_on_jobs = [depend_on_jobs]
    jobs = flatten(jobs)
    depend_on_jobs = flatten(depend_on_jobs)
    hj.add_soft_p_dependencies(jobs, depend_on_jobs)
    return flatten(jobs + depend_on_jobs)
def softndep(jobs, depend_on_jobs):
    if type(jobs) is not list:
        jobs = [jobs]
    if type(depend_on_jobs) is not list:
        depend_on_jobs = [depend_on_jobs]
    jobs = flatten(jobs)
    depend_on_jobs = flatten(depend_on_jobs)
    hj.add_soft_n_dependencies(jobs, depend_on_jobs)
    return flatten(jobs + depend_on_jobs)
```

## Backend (Vagelis, Mathias, and George)

```
'''
Implementation of semantic actions
'''
from networkx import DiGraph
import json
import redis
import jobqueue
import subprocess
import time
import uuid
import random
import os
```

```
depen_graph = DiGraph()

class Job():
    '''Constructor of class should be supplied with job name and
    respective script name
    '''
    def __init__(self, script='', arguments=[], deps_jobs=None, deps_args=None):
        self._dependencies = []
        self._host = ''
        self._port = ''
        self._channel = ''
        self._script = script
        self._arguments = arguments
        self._stderr = None
        self._stdout = None
        self._errno = None  # errno is None since job has not run
        self._deps_jobs = deps_jobs
        self._deps_args = deps_args
        self.soft_priority = 0
        depen_graph.add_node(self)
        # log is not needed if any script is associated
        if not script:
            return
        # create log file; truncate it if it exists
        self._logfile_name = os.getcwd() + '/' + self._script + ".log"
        try:
            self.f = open(self._logfile_name, "w")
        except Exception, error:
            print "Unhandled Exxception:", error,\
                    " while creating log file for job:", self._script
        self.f.close()

    def set_cluster(self, host, port, channel):
        self._host = host
        self._port = port
        self._channel = channel

    def stdout(self):
        return self._stdout

    def script(self):
        return self._script

    def dependencies(self):
        return self._dependencies

    def perror(self):
        return self._errno, self._stderr

    def add_dependency(self, depends_on):
        '''Add names of jobs on which current object
        depends on.
        '''
        for job in depends_on:
            self._dependencies.append(job)
            depen_graph.add_edge(self, job)

    def compute_args(self):
        if self._deps_args:
            self._arguments = self._deps_args(sum([j.stdout().split("\n") \
                        for j in self._deps_jobs], []))
    def run(self):
        self.compute_args()
        if not self._host:
            self._run_localy()
        else:
            self._run_remotely()

    def _run_localy(self):
        '''run a job localy'''
        # need error checkong of what Popen returns
        try:
            # execute command
            pcommand = [self._script] + self._arguments
            os.chmod(self._script, 0766)
            s = subprocess.Popen(pcommand, stdout=subprocess.PIPE,
                                        stderr=subprocess.PIPE)
            # get return values
            self._stdout, self._stderr = s.communicate()
            self._errno = s.returncode

            self._log()
        except Exception, error:
            print "Unhandled Exception:", error, "for job:", self._script,\
                    "with arguments:", self._arguments
            return

    def _run_remotely(self):
        '''run a job remotelly'''
        host = self._host
        port = self._port
        channel = self._channel
        try:
            connection_pool = redis.Redis(host, port)
        except Exception, error:
            print "Redis Error:", error
            return

        # read script and publish it to channel
        try:
            f = open(self._script, "r")
            arguments = self._arguments
            script_body = f.read()
            f.close()
        except Exception, error:
            print "Unable to open script:", self._script
            print "Exception:", error
            return

        # construct message. First line is the arguments,
        # following lines are the body of the script
        job_key = uuid.uuid1().hex

        # get current subscribers
        workers = []
        for item in connection_pool.client_list():
            if item['cmd'] == 'subscribe':
                workers.append(item['addr'].split(':')[0])
        # if no listen workers end
        # posible bug here
        if not workers:
            print "NO WORKERS"
            return
        jobsworker =  random.choice(list(set(workers)))

        request = {'job_key': job_key,\
                     'jobs_worker': jobsworker,\
                     'job_arguments': ' '.join(arguments),\
                     'script_name': self._script,\
                     'script_body': script_body}
        # encode
        jrequest = json.dumps(request)

        # publish message
        connection_pool.publish(channel, jrequest)

        # get response in job specific channel
        pubsub = connection_pool.pubsub()
        pubsub.subscribe(job_key)

        for item in pubsub.listen():
            if item['type'] == 'message':
                response = json.loads(item['data'])
                self._stdout = response['stdout']
                self._stderr = response['stderr']
                self._errno = response['errno']
                break

        # log stderr, stdout, and errno
        self._log()
        return

    def can_run(self):
        '''check if dependencies are fullfilled.
        Current instrance can run only if all jobs from
        its dependency list have successed ( errno is zero )
        '''
        if not self._dependencies:
            return True
        for job in self._dependencies:
            if job.perror()[0] != 0:
                return False
        return True

    def _log(self):
        # file is created by class constructor. Append logs.
        f = open(self._logfile_name, "a")
        print >> f, "STDOUT:"
        print >> f, "---------"
        print >> f, self._stdout
        print >> f, "---------"
        print >> f, "STDERR:"
        print >> f, "---------"
        print >> f, self._stderr
        print >> f, "#########"
        f.close()
```

```
class Wait(Job):
    '''fake class to create wait objects'''
    def __init__(self, sleepinterval):
        Job.__init__(self)
        self._sleepinterval = int(sleepinterval)

    def run(self):
        time.sleep(self._sleepinterval)
        self._errno = 0

    def _log(self):
        pass

def add_dependencies(jobs, depend_on):
    for job in jobs:
        job.add_dependency(depend_on)

def add_soft_p_dependencies(jobs, depend_on):
    m_dep = max([x.soft_priority for x in depend_on]) + 1
    for job in jobs:
        job.soft_priority = m_dep
    jobqueue.GlobalJobQueue.sort

def add_soft_n_dependencies(jobs, depend_on):
    m_dep = min([x.soft_priority for x in depend_on]) - 1
    for job in jobs:
        job.soft_priority = m_dep
    jobqueue.GlobalJobQueue.sort

def add_soft_equal_dependencies(jobs, depend_on):
    m_dep = min([x.soft_priority for x in depend_on])
    for job in jobs + depend_on:
        job.soft_priority = m_dep
    jobqueue.GlobalJobQueue.sort

def service(host_port):
    try:
         jobqueue.GlobalServiceHost = host_port.split(":")[0]
         jobqueue.GlobalServicePort = host_port.split(":")[1]
    except Exception, error:
        print "Unable to get service host and IP. Running locally."
        jobqueue.GlobalServiceHost = ''
        jobqueue.GlobalServicePort = ''

def run(JobsList):
    host = jobqueue.GlobalServiceHost
    port = jobqueue.GlobalServicePort
    channel = 'maestro_channel'
    '''Enqueue jobs'''
    if isCyclic(depen_graph):
        print "Your jobs have circular dependencies"
        return False

    # print some stats so that user knows what's going on
    for job in JobsList:
        job.set_cluster(host, port, channel)
        print "Job: \"%s\"" % job.script(),\
                    "has: %d" % len(job.dependencies()),\
                    "unresolved dependencies."
        jobqueue.GlobalJobQueue.enqueue(job)
    jobqueue.GlobalJobQueue.sort

def isCyclicUtil(graph, vertex, visited, recstack):
    if not visited[vertex]:
        visited[vertex] = True
        recstack[vertex] = True
    for des in graph.successors(vertex):
        if not visited[des] and isCyclicUtil(graph, des, visited, recstack):
            return True
        elif recstack[des]:
            return True
    recstack[vertex] = False
    return False

def isCyclic(graph):
    visited = {}
    recstack = {}
    for node in graph.nodes():
        visited[node] = False
        recstack[node] = False
    for node in graph.nodes():
        if isCyclicUtil(graph, node, visited, recstack):
            return True
    return False
'''
Workers API
'''
import urllib
import jobqueue
import json
import redis
import subprocess
import  os
import signal
import sys
import uuid

def signal_handler(signal, frame):
    print "\nSee u soon :-)\nGoodBye!"
    sys.exit(0)

signal.signal(signal.SIGQUIT, signal_handler)

class worker():
    '''Constructor of class'''
    def __init__(self, host_port='', channel=''):
        jobqueue.GlobalJobQueue.stop()
        self.worker_key = job_key = uuid.uuid1().hex
        # get arguments
        try:
            self.host = host_port.split(':')[0]
            self.port = host_port.split(':')[1]
            if not channel:
                self.channel = 'maestro_channel'
            else:
                self.channel = channel
        except Exception, error:
            print "Malformed IP:PORT pair:\"%s\"" % ip_port
            print error
            return
        # instanciate pool
        try:
            self.connection_pool = redis.Redis(self.host, self.port)
            print "Connection pool at: <%s:%s>" % (self.host, self.port)
        except Exception, error:
            print "Failed to launch ConnectionPool"
            print error
            return
        #subscribe to channel
        try:
            self.pubsub = self.connection_pool.pubsub()
            self.pubsub.subscribe(self.channel)
        except Exception,  error:
            print "Unable to subscribe to channel"
            print error
            return
        # start polling for messages
        print "Polling for messages"
        for item in self.pubsub.listen():
            if item['type'] == 'message':
                # if exception keep up the server. The client may
                # get a bad response, but the server must survive
                try:
                    self.execute(item)
                except Exception, error:
                    print  "Unhandled Exception:", error
                    continue
    def execute(self, item):
        '''function to run job localy and publish back its output'''
       # decode request body
        request = json.loads(item['data'])
        jobs_worker = request['jobs_worker']
        # if job is not assigned to you terminate
        if jobs_worker != getexternalip():
            return
        #parse request
        job_key = request['job_key']
        arguments = request['job_arguments'].split(' ')[:]
        script_body = request['script_body']
        script_name = request['script_name']
        # create temp file
        f = open("./this_will_never_exist.sh", "w")
        f.write(script_body)
        f.close()
        try:
            os.chmod("./this_will_never_exist.sh", 0766)
        except Exception, error:
            print "Unhandled Exception:", error
        # execute command
        pcommand = ['./this_will_never_exist.sh'] + arguments
        s = subprocess.Popen(pcommand, stdout=subprocess.PIPE,
                                      stderr=subprocess.PIPE)
        # print what you executed
        print "Just executed job:", script_name,\
                 "with arguments:", arguments
        # get return values
        stdout, stderr = s.communicate()
        errno = s.returncode
        # remove temp file
        os.remove("./this_will_never_exist.sh")
        # send back response to custom job channel
        response = {'stdout': stdout,
                    'stderr': stderr,
                    'errno': errno}
        jresponse = json.dumps(response)
        self.connection_pool.publish(job_key, jresponse)

def getexternalip():
    '''This function returns the external IP of the machine.
    Subscribers are registered in Redis channel with their
    external IP, which is used to identify them when jobs
    are dispatched
    '''
    ip = urllib.urlopen('http://www.biranchi.com/ip.php').read()
    return ip[3:]
```

## Job Dispatcher Daemon (Vaggelis)

```
'''
Implementation of job queue
'''
import jobs
import time
import threading
import signal
import sys

class JobQueue():
    def __init__(self):
        self.mutex = threading.Lock()
        self._Q = []
        self._Run = True
        signal.signal(signal.SIGQUIT, self.signal_handler)

    def enqueue(self, job):
         self.mutex.acquire()
         self._Q.append(job)
         self.mutex.release()

    def dequeue(self, job):
        self.mutex.acquire()
        self._Q.remove(job)
        self.mutex.release()
    def stop(self):
        self._Run = False

    def execute(self, job):
        print "Running job: \"%s\"" % job.script()
        job.run()
        (errno, stderr) = job.perror()
        if errno != 0:
            print "Error while executing Job: \"%s\"" % job.script()
            print stderr
        else:
            print job.stdout()

    def poll_for_jobs(self):
        '''Thread main loop'''
        while self._Run or self._Q:
            # loop on a copy of the queue to avoid deadlocks
            temp = list(self._Q)
            for job in temp:
                if job.can_run():
                    threading.Thread(target=self.execute, args=[job]).start()
                    self.dequeue(job)
            time.sleep(0.5)

    def sort(self):
        sorted(self._Q, key=lambda x: x.soft_priority)

    def signal_handler(signal, frame):
        self._Run = False
        self._Q = None
        print "\nSee u soon :-)\nGoodBye!"
        sys.exit(0)

GlobalJobQueue = JobQueue() # construct global queue
GlobalServiceHost = ''
GlobalServicePort = ''
```