FlowScript DSL and JobSystem Integration Report

0.1 Job System FFI

0.1.1 FFI Communication

The FFI utilized by the JobSystem offers communication via JSON objects. The system sends and recieves C-String pointers with data serialized as a JSON. Deserialization then occurs within the JobSystem.

0.1.2 JobSystem FFI Features

1. The ability for other languages to query Job 'handles,' which are descriptors containing information about Job status and allow the calling program to block and wait for jobs to complete, similar to a traditional OS thread handle.
2. The ability for multiple JobSystems to be created and operated on
3. Closure-based interface, which allows registerable jobs to adhere to one constraint, which is to consume a JSON and return a JSON.

0.2 FlowScript Design

0.2.1 Design Goal

Create a minimal data-oriented language

0.3 Syntax

0.3.1 Variables

JSON objects that are only defined during execution) begin with a user-defined name, followed by ''. Example: `foo{}`

0.3.2 Execution Order and Data Passing

Data may be passed to/from `Process`es and variables using the `->` operator. The `->` assumes the sink is dependent on the source and execution will be order as such to ensure the source has fully completed before passing data to the sink.

0.3.3 Conditional Statements

Based on a `Process`'s output, the flow of execution may be controlled with a condition statement. Condition statements can be invoked by applying in front of a Process definition, for example: `[Process]`. The JSON output of `Process` must be in the form {`"result" : int, "value" : {...}`}. `result` refers to the index of the branch that should be taken. Given the following FlowScript: `input -> [Process] -> ([AnotherProcessA], [AnotherProcessB])`. If the `result` key was 0, execution would branch to `AnotherProcessA`, if it was 1, execution would branch to `AnotherProcessB`. The JSON object within the `value` is then directly passed to the branched `Process`. Additionally, a branch can be pointed to with a `'` referring to an existing or future `Process`. An example of this would be: {`"value": 1`} `-> [AddOne]'a -> [isGreaterThan10] -> ('a, ReturnValue{})`, which will repeatedly branch back to `AddOne`, until `isGreaterThan10` returns a `"result"` of 1.

0.4 Parallelism with Fork/Join

Similar to forking on UNIX, fork copies states into separate processes, joining them when they finish. A `fork` may be performed with the `<` operator. The forked section will only be completed with a join, with the `>` operator. An example of this usage for processing in parallel: `files -> <(FileProcessor)> -> outputs`. The fork accepts an array of values and each individual value is passed a separate forked `Process`.

Additionally, another variant of forking with multiple processes in the tuple (...) will use provide each individual index of the input JSON array to their corresponding `Process`: `{[{"file": {...}}, {"file": {...}}]} -> <([ProcessA], [ProcessB])> -> outputs{}`.

0.5 Translations from FlowScript to DOT

0.5.1 Process execution

1. **FlowScript:** `{"target" : "test"} -> make -> clangparse -> contextadd -> output{}`

2. **DOT Code:**

```
digraph {
    {
        node [shape=ellipse]
        maketarget [data="{\"target\" : \"test\"}"]
        output
    }

    {
        node [shape=box]
        make
        clangparse
        contextadd
    }

    maketarget -> make -> clangparse -> contextadd -> output
}
```

0.6 Conditional Jumping

1. **FlowScript:** `{"target": "test"} -> maketarget -> makestatus -> (error_output{},`
`clangparse -> parsestatus -> (error_output{},`
`contextadd -> contextstatus -> (error_output{}, success_output{}))`

2. **DOT Code:**

```
digraph {
{
node [shape=ellipse]
maketarget [data="{\"target\" : \"test\"}"]
success_output
error_output
}

{
node [shape=box]
make
clangparse
contextadd
}
{
node [shape=triangle]
makestatus
```

```
    parsestatus
    contextstatus
    }
    maketarget -> make -> makestatus -> {error_output, clangparse}
    clangparse -> parsestatus -> {error_output, contextadd}
    contextadd -> contextstatus -> {error_output, success_output}
    }
```

0.7 Parallelism

Parallelism is achieved in FlowScript via fork/join with trapezium/invtrapezium. Forking to achieve parallelism may be done both explicitly and implicitly. The fork command accepts a data array, which can then be directed to Processes. In the case where only a single process is directed from the work, each process is performed in parallel until it joined. When more than two Processes leave a fork, the data in each index of the data array gets passed to their corresponding job. Example of implicit parallelism:
FlowScript:

```
{[{"file1": {}}, {"file2": {}}]} -> <fork -> [compile_file] -> join> -> output{}
```

DOT Language:

```
digraph {
    {
        node [shape=ellipse]
        files
        output
    }

    {
        node [shape=box]
        compile_file
    }

    {
        node [shape=trapezium]
        fork_compile
    }

    {
        node [shape=invtrapezium]
        join_compile
    }

    files -> fork_compile -> compile_file -> join_compile -> output
}
```

An example of explicit parallelism:
FlowScript:

```
{[{"file1": {}}, {"file2": {}}]} -> <fork -> {[compile_file_1], [compile_file_2]}
-> join> -> {output1{}, output2{}}
```

DOT Language:

```
digraph {
    {
        node [shape=ellipse]
```

```
        file1
        file2
        output1
        output2
    }

    {
        node [shape=box]
        compile_file_1
        compile_file_2
    }

    {
        node [shape=trapezium]
        fork_compile
    }

    {
        node [shape=invtrapezium]
        join_compile
    }

    {file1, file2} -> fork_compile -> {compile_file_1, compile_file_2} -> join_compile -> {output1, outp
}
```