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

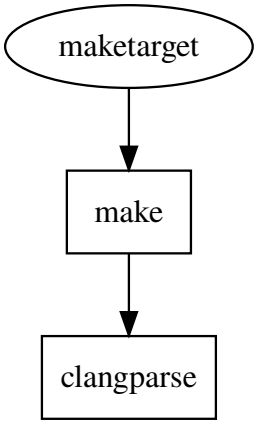
Variables are defined using the default ellipse shape in the DOT language. For variables with static data, such as JSON constants, the "data" attribute may be added, containing the specified JSON constant. Additionally, boxes define process and will be named in the same manner that they are registered in the JobSystem. An example of that would be:

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

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. An example of that would be:

```
digraph {
    {
        node [shape=ellipse]
        maketarget [data="{\"target\" : \"test\"}"]
    }
    {
        node [shape=box]
        make
clangparse
    }
    maketarget -> make -> clangparse
}
```



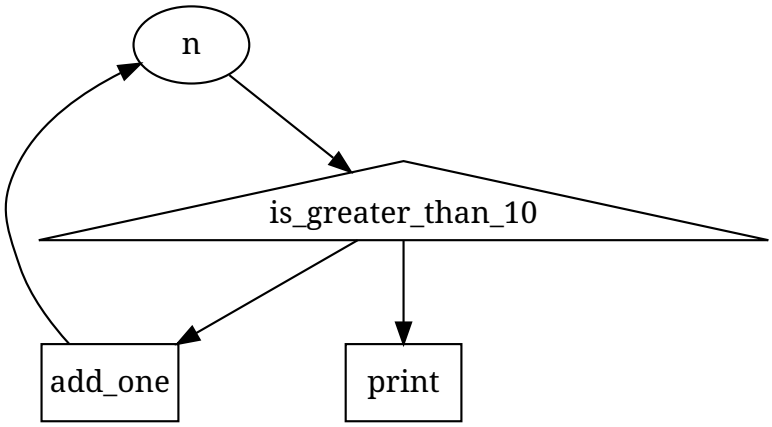
0.3.3 Conditional Statements

Based on a `Process`'s output, the flow of execution may be controlled with a condition statement. Condition statements can be invoked by applying inserting triangle nodes. Conditionals are similar to branches in Assembly programming and may be used to implement loops or if-branching. An example which incorporates both would be:

```
digraph {
    {
        node [shape=ellipse]
        n [data="{\"value\" : \"0\"}"]
    }
    {
        node [shape=box]
        add_one
        print
    }
    {
        node [shape=triangle]
        is_greater_than_10
    }

    is_greater_than_10 -> {print; add_one}
    add_one -> n

    n -> is_greater_than_10
}
```

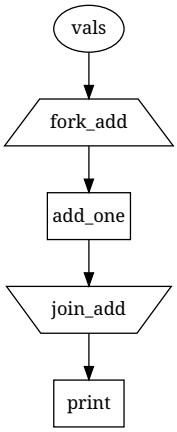


0.4 Parallelism with Fork/Join

Similar to forking on UNIX, fork copies states into separate processes, joining them when they finish. A `fork` may be performed with the trapezium object, while `joining` can be performed with the invtrapezium object. The forked section will only end until it is rejoined. It is invalid to fork without joining. Additionally, the fork operator expects a "values" array. Joining will return the same values array with the mapping done by each Process in the fork. There are 2 variants to forking. One variant, which operates in a data-independent way, where the same processes are called on an array of data. There is also forking in a data dependent manner, where FlowScript code can define which data goes to which branch of the fork. Here is an example of forking in a data independent manner:

```
digraph {
    {
        node [shape=ellipse]
        vals [data="{\"values\" : [0, 0, 1, 1, 2, 2]"]
    }
    {
        node [shape=box]
        add_one
        print
    }
    {
        node [shape=trapezium]
        fork_add
    }
    {
        node [shape=invtrapezium]
        join_add
    }

    vals -> fork_add -> add_one -> join_add -> print
}
```



Here is forking in the data dependent manner. Note that this version will send data to each branch based on the index of the data in the values array and the index of the branch.

```
digraph {
    {
        node [shape=ellipse]
        vals [data="{\"values\" : [{\"dividethis\": 5}, {\"addthis\": 4}]"]
    }
    {
        node [shape=box]
        divide_op
        add_op
        print
    }
    {
        node [shape=trapezium]
        fork_ops
    }
    {
        node [shape=invtrapezium]
        join_ops
    }

    vals -> fork_ops -> {divide_op; add_op} -> join_ops -> print
}
```