FlowScript Interpeter Report

0.1 Rules for lexical and syntax parsing

Both the lexer and parser of the flowscript interpeter are using handmade parsers, with a simple system to parse rules.

0.1.1 Lexer

The lexer is a basic state machine that reads the raw input string, character-by-character. The types of tokens include: Arrow, Open/closed brackets and braces, Comma, Equals, Semicolon, Variable text, Reserved text, made up of dot-reserved keys. The lexer strictly reads character by character, however it saves the state of previous characters to enable features, such as escape characters.

0.1.2 Parser

The parser uses handwritten rules to parse tokens directly from the lexer. The parser takes these tokens and builds out an execution graph using the petgraph library. This stores every possible execution state within a directed graph, and can be read from in parallel. There are multiple states within the parser, however there are 3 main states. First, the parser searches for the `Digraph` (with an optional graph name). Next the parser reads line by line, searching for two types of statements. The first statement type, represents and attribute, for example `make[data="{}"]`, which stores the `data` attribute, storing a JSON within the `make` node. The other type of statement is a directive, which defines what the next state should be, for example: `make -> clang_parse`. Additionally, multiple directives may be added for a single source process. They will be stored in the order they were declared and are jumped to based on the `"status"` code returned by a process/job. Each statement is delimited by a semicolon. Because of the parser's graph based architecture, this enables easy parallelization of graph building, by enabling multiple subgraphs to be generated in parallel and then combined when finished.

0.2 Error types

Errors will occur in the form of an unexpected token along with the upcoming context.

Multiple statements on the same line

```
digraph {
    make[shape=rectangle,data="{\"input\":{\"target\" : \"test\"}}"];

    make->clang\_parse
    make->print\_error;
}
```

This would yield: `Error: "Unexpected token sequence: [Text("make"), Arrow, Text("clang_parse"), Text("make"), Arrow, Text("print_error")]"` Because the parser expects every statement to be delimited by a semicolon token, the upcoming token sequence is printed out to let the programmer know there the error occurred.

To fix this, a semicolon is required after `clang_parse`:

```
digraph {
    make[shape=rectangle,data="{\"input\":{\"target\" : \"test\"}}"];

    make->clang\_parse;
    make->print\_error;
}
```

Incorrect opening to digraph

```
digraph
    make[shape=rectangle,data="{\"input\":{\"target\" : \"test\"}}"];
}
```

This would yield: `Error: "Unexpected token sequence: [Bracket(Open), ReservedText(Shape), Equals, Text("rectangle"), Comma, ReservedText(Data), Equals, Text(""input":"target" : "test""), Bracket(Closed)]"` Because an opening brace is expected, the parser will mistakingly extract all tokens up to thesemicolon, before producing the error.

To fix this, the user must add the opening brace after `digraph`:

```
digraph {
    make[shape=rectangle,data="{\"input\":{\"target\" : \"test\"}}"];
}
```

Silently failing

Most errors in the current state of the FlowScript interpreter will be silently ignored. For example: Unbalanced brackets are often ignored by the interpreter and exist to comply with DOT. Addtionally, errors may occur silently, such as when jobs are not registered with the system. For example:

```
digraph {
    mke[shape=rectangle,data="{\"input\":{\"target\" : \"test\"}}"];
}
```

will silently fail (as `mke` is not registered with the jobsystem). The solution to this type of error is to evaluate the flowscript and find the job that is not registered. The fix would be to rename the job:

```
digraph {
    mke[shape=rectangle,data="{\"input\":{\"target\" : \"test\"}}"];
}
```