Giovanni Boscan 11/13/2023

Lab 3 Report

Lexical and Syntactical Analysis of Flowscript

For Lab 3 I developed a flowscript interpreter in C++ that interprets a digraph so that the jobs specified in the graph can be completed in the way that is intended. This is done by reading the graph line by line and systematically checking for syntactic and lexical errors to ensure that the graph is formatted correctly and so that the code can compile as intended.

While reading the lines of the digraph the interpreter first checks for large formatting, or syntactic, issues in the graph's construction. These large issues include missing conditions, labels created incorrectly, and other issues including missing brackets. By doing this, the interpreter can check first for large issues that might obstruct the proper code from compiling and would always lead to errors.

After the interpreter checks for the large syntactic issues, it then checks the code for smaller lexical issues. While syntactic issues such as missing conditions or labels are checked, issues such as whether these parameters are formatted correctly are not. For example, an if statement might be found to be correct by syntactic measures, as in it might possess the correct structure of true-false, but the condition by which this is decided might not make sense. Potential errors such as these are why a two-step approach to checking for potential errors in the digraph that we use as input.

After the digraph is checked to be error-free it is then formatted in a series of data structures including a vector that lists the proper connections between the nodes of the graph along with an Unordered-Map that is formatted to fully contain the information of the digraph. This data structure, the unordered map, can then process the jobs in the correct order as it works as the intermediate step between the digraph and the rest of the Job system.

Compiler Error Generation

The FlowScript parser has the ability to catch most general syntactical and lexical errors described by the rules of the language. Some of them include the following:

“Statements after brackets are not valid”: The error occurs when there are statements after a bracket expression. For example:

```
digraph {
    input
    output

    input -> sort [label="example"] -> output
}
```

“Process cannot start with a digit”: The error happens when a process name is started with a digit instead of an alphabetic character. For example:

```
digraph {
    input
    output
    1sort [shape="square"]

    input -> 1sort -> output
}
```

“Missing closing bracket”: This error happens when a closing bracket is missing, therefore the argument list is unknown. For example:

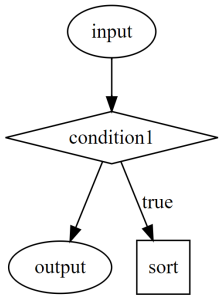
```
digraph {
    input
    output
    sort [shape="square"

    input -> sort -> output
}
```

"Closing bracket before opening bracket": This error occurs when a cloning bracket is detected before an opening bracket. This error makes debugging more intuitive for the programmer instead of just throwing an “Unknown symbol” error. For example:

```
digraph {
    input
    output
    sort] [shape="square"

    input -> sort -> output
}
```

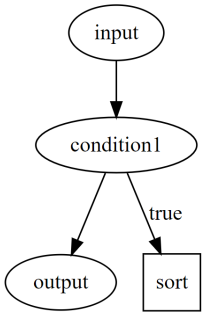
“If statement requires a condition”: The error is thrown when an if condition (a shape=“diamond” is in the argument list) does not contain any label (which indicates the if condition to evaluate). For example:

```
digraph {
    input
    output
    sort [shape="square"]
    condition1 [shape="diamond"]

    input -> condition1
    condition1 -> sort [label="true"]
    condition1 -> output
}
```



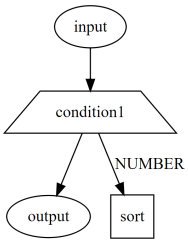
"If declaration cannot be declared with a dependency": In order to simplify the syntax of FlowScript, the if statement has to be declared with no dependencies in order to be valid lexically. Therefore, a compiler error is thrown. For example:

```
digraph {
    input
    output
    sort [shape="square"]
    input -> condition1 [shape="diamond"]

    condition1 -> sort [label="true"]
    condition1 -> output
}
```



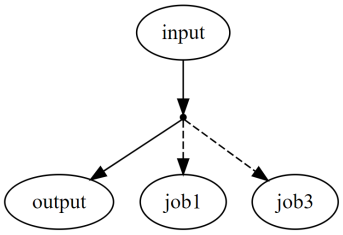
"Switch statement requires a condition": Similarly, if a switch condition does not have a condition, an error will be thrown. For example:

```
digraph {
    input
    output
    sort [shape="square"]
    condition1 [shape="trapezium"]

    input -> condition1
    condition1 -> sort [label="NUMBER"]
    condition1 -> output
}
```



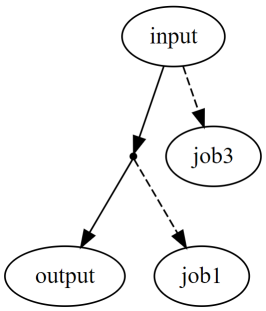
"Point name needs to be split, not xxxxx": Since split is a reserved keyword of FlowScript and to simplify the syntax, a point shape cannot have a different name than “split”, therefore, an error will be generated. For example:

```
digraph {
    input
    notSplit [shape="point"]
    output

    input -> notSplit
    notSplit -> job1 [style="dashed"]
    notSplit -> job3 [style="dashed"]
    notSplit -> output
}
```



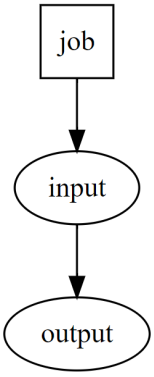
“Dashed property needs to come from a split or another dashed process”: If the style property of a dependency is dashed, and the process is not a child of the split keyword or another threaded job, the compiler will generate an error. For example:

```
digraph {
    input
    split [shape="point"]
    output

    input -> split
    split -> job1 [style="dashed"]
    input-> job3 [style="dashed"]
    split -> output
}
```



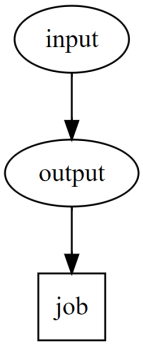
“Input block cannot have a dependency”: Since the nature of an input block, it doesn’t make lexical sense to have a dependency for an input statement, therefore, an error will be thrown. For example:

```
digraph {
    input
    output
    job [shape="square"]

    job -> input -> output
}
```



“Output block cannot be a dependency”: Since the nature of an output block, it doesn’t make lexical sense for the output to be a dependency of another block since it indicates the end of the process chain, therefore, an error will be thrown. For example:

```
digraph {
    input
    output
    job [shape="square"]

    input -> output -> job
}
```



“Unknown symbol”: If there is a symbol that is not recognized by the FlowScript language, an error is generated. For example:

```
digraph {
    input
    output
    job $ [shape="square"]

    input -> job -> outpt
}
```