How to run the program and read the files

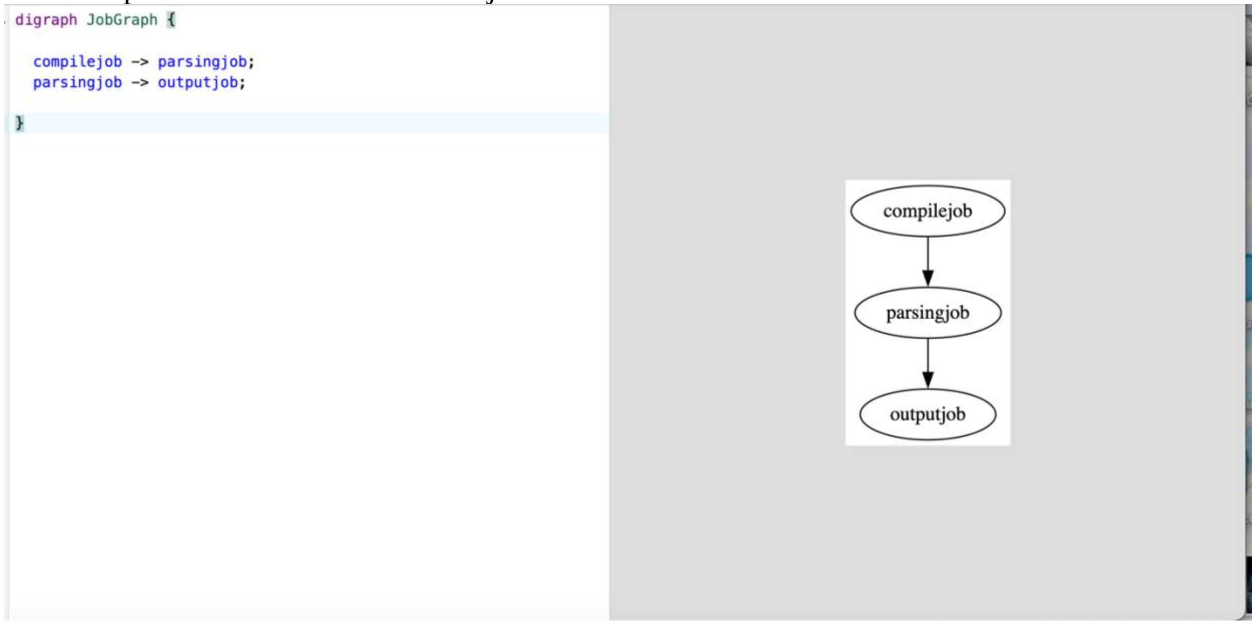
My program uses make to compile and run.

To compile the program: make compile

To run the program: make run

I have three flowscript files that my program reads in:

1. fsp1.txt – this is the base case job flow here

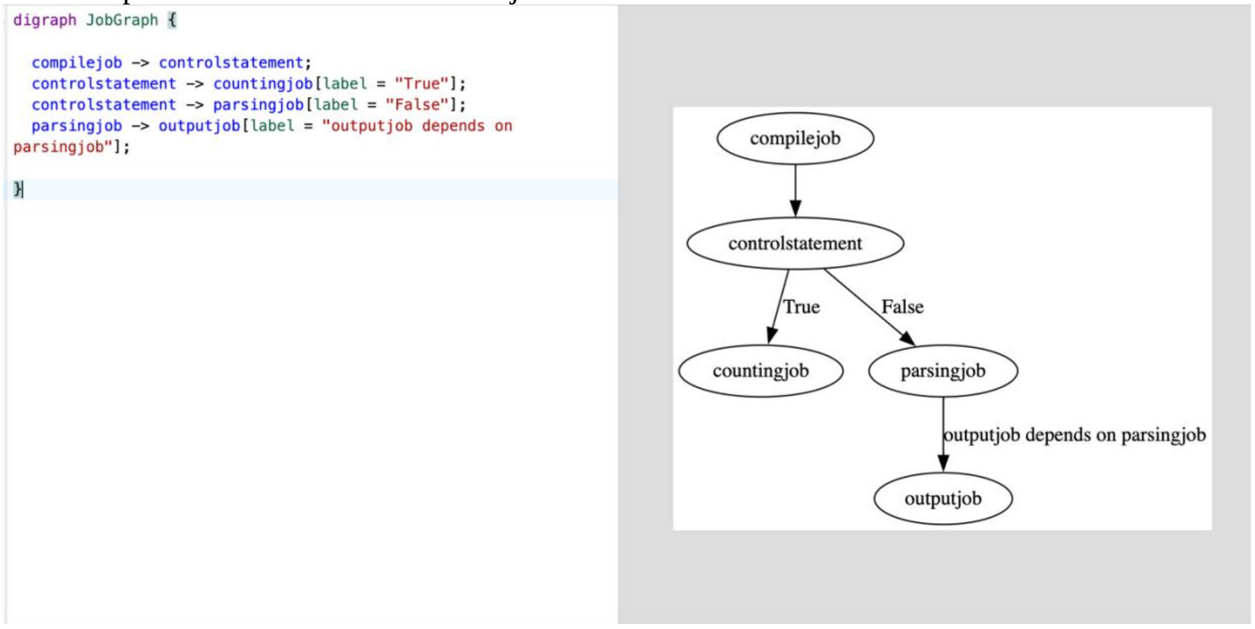


```
digraph JobGraph {

  compilejob -> parsingjob;
  parsingjob -> outputjob;

}
```

2. fsp2.txt – this is the conditional job flow here

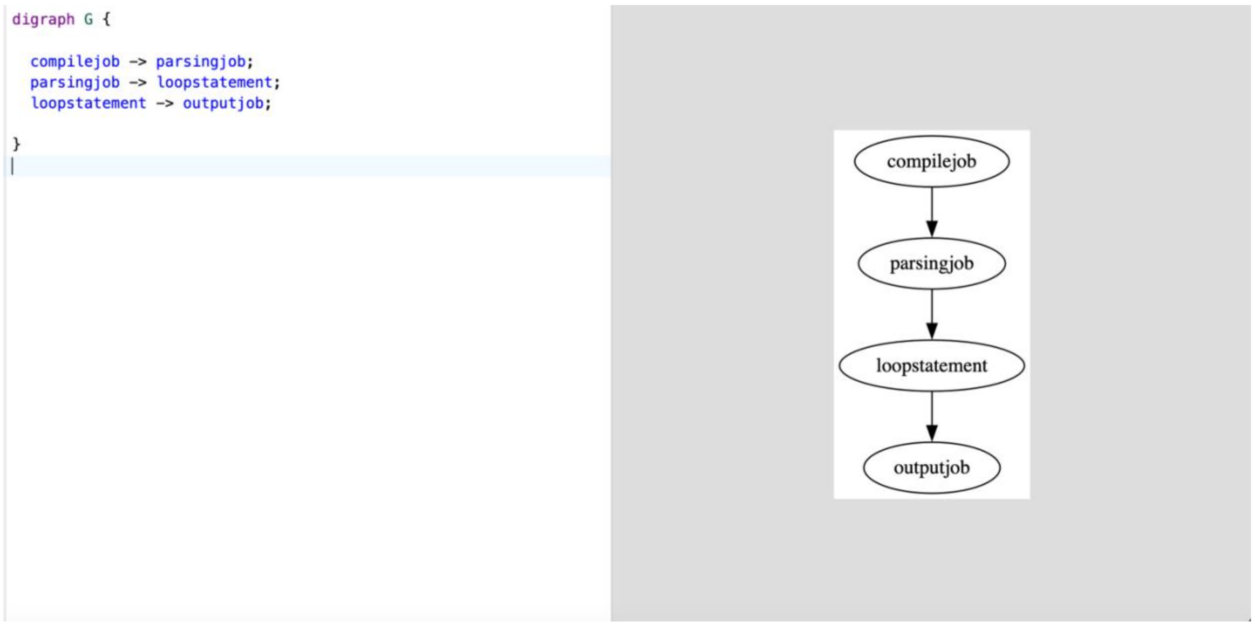


```
digraph JobGraph {

  compilejob -> controlstatement;
  controlstatement -> countingjob[label = "True"];
  controlstatement -> parsingjob[label = "False"];
  parsingjob -> outputjob[label = "outputjob depends on
parsingjob"];

}
```

3. fsp3.txt – this is the loop job flow here

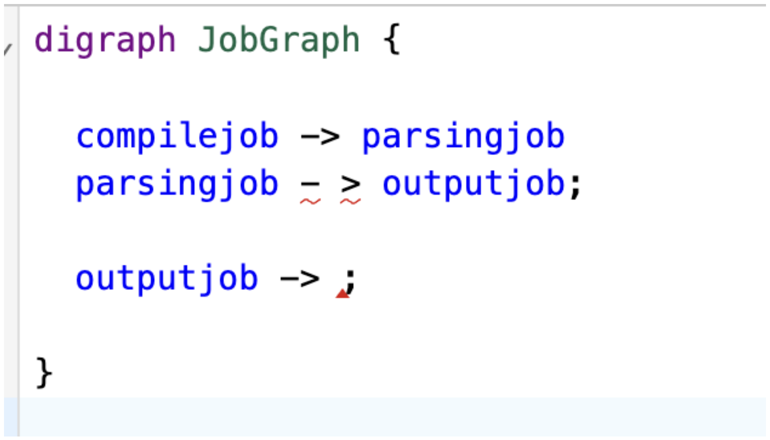


```
digraph G {

  compilejob -> parsingjob;
  parsingjob -> loopstatement;
  loopstatement -> outputjob;

}
```

4. fsp-errors-test.txt – this is the flowscript with intentional errors to test error logging



```
digraph JobGraph {

  compilejob -> parsingjob
  parsingjob - > outputjob;

  outputjob -> ;

}
```

To run each flowscript file that is located in the Data directory, go to line 34 in customjob.cpp:

```
  std::ifstream stream("./Data/fsp-errors-test.txt");
 // Adjust the path location of your .txt file
```

Here you can adjust the path to each of these four files like so:

1. ./Data/fsp1.txt
2. ./Data/fsp2.txt
3. ./Data/fsp3.txt

4. ./Data/fsp-errors-test.txt

Running the different processes will run the jobs in the jobsystem, and the jobs will write to three .json files:

1. compileJobOutput.json – compilejob will write to this json to show its error output from the terminal

2. errors.json – parsingjob will write to this json to show the parsed errors. These are the errors from the compileJobOutput.json, but formatted in a parsed way.

3. Output.json – outputjob will write the errors from the errors.json in an organized format that groups by file name. For example, test-set-one.cpp will be in the same json array as all other test-set-one.cpp's and all the test-set-two.cpp's will be in a different array in the json file separate from test-set-one.cpp, etc.

4. Flowscript-errors.json catalogs lexical and syntactical errors from the flowscript files that antlr cannot parse correctly.

Rules for lexical and syntactical parsing

My grammar rules are able to look for Node_definitions and grab the associated flowscript labels out of the subgraph section of flowscript. For example,

```
digraph JobGraph {

    subgraph cluster_jobs {

    compilejob [label="compilejob", shape="oval"];
    ControlStatement[label="a.outv1>a.outv2", shape="Mdiamond"];
    countingjob [label="countingjob", shape="oval"];
    parsingjob [label="parsingjob",shape="oval"];
    outputjob [label = "outputjob",shape="oval"];
  }

  compilejob -> ControlStatement;
  ControlStatement -> countingjob[label = "True"];
  ControlStatement -> parsingjob[label = "False"];
  parsingjob -> outputjob[label = "outputjob depends on parsingjob"];

}
```

Flowscript like this could be broken down into a Complex Syntax Tree (CST) by antlr, and antlr would then allow you to make customvisitor class that inherits from antlr's generated base visitor class. In this customvisitor class, the visitNode_definition function would be ran everytime antlr visits a node definition in the flowscript, and then take out each node's label. These labels coud then be thrown into a jobQueue vector and written to a file, so for instance the file format would be:

```
compilejob
Oval
a.outv1>a.outv2
Mdiamond
Countingjob
Oval
Parsingjob
Oval
Outputjob
Oval
```

This is a decent start at parsing flowscript and using it to run the jobsystem, but just taking strings out of the subgraph section in the order they appear is not the real job flow that the flowscript illustrates.

I modified my customvisitor class to have another function called VisitEdge_definition. This function visits the edges instead of the node definitions, so the function allows simpler flowscript without a subgraph section. This visitEdge_definition function allows the code to parse the actual flow of the jobs, so the code could take flowscript like this:

```
digraph JobGraph {

  compilejob -> parsingjob;
  parsingjob -> outputjob;

}
```

and place this flow into a jobQueue vector and eventual file without duplicates as:

```
Compilejob
Parsingjob
Outputjob
```

And In more advanced cases, take this flowscript:

```
digraph JobGraph {

  compilejob -> controlstatement;
  controlstatement -> countingjob[label = "True"];
  controlstatement -> parsingjob[label = "False"];
  parsingjob -> outputjob[label = "outputjob depends on parsingjob"];
```

}

and turn it into this flowscript:

Compilejob
Controlstatement
Countingjob
Parsingjob
Outputjob

This simpler flowscript and parsing method is what I settled on for lab-3 as its more adaptable across a wider range of use cases due to its simplicity.

Grammar Rules described in detail:

```
grammar FlowscriptGrammar;


// Parser rules
flowscript              : DIGRAPH ID '{' graph_content '}';
graph_content           : (node_definition | edge_definition)* ;
node_definition         : ID node_attrs ';' ;
node_attrs              : '[' attr_list ']' ;
attr_list               : attr (',' attr)* ;
attr                    : 'shape' '=' STRING {System.out.println($STRING.text); } | ID ;
edge_definition         : ID '->' ID edge_attrs? ';' ;
edge_attrs              : '[' a_list ']' ;
a_list                  : a_list_item (',' a_list_item)* ;
a_list_item             : ID '=' STRING ;
cluster_id              : 'cluster_' ID | ID ;


// Lexer rules
DIGRAPH                 : 'digraph';
ID                      : [a-zA-Z_][a-zA-Z_0-9]* ;
STRING                  : '"' (~["\r\n\\] | '\\' .)* '"';
WS                      : [ \t\r\n]+ -> skip ;
```

Parsing Rules

Flowscript is the root rule that defines the structure of a flowscript file. Flowscript files start with Digraph. This diagraph is followed by an opening {, the graph_content, and a closing }. The opening and closing {} are the borders of the flowscript file.

The graph_content fills up with the content of the flowscript file. Graph_content defines the content inside a flowscript graph. This content can be zero or more node or edge definitions.

Node_definition is the definition of a node within the graph. Node_definition consists of an identifier for the node followed by optional node attributes and a semicolon.

Node_attrs contains properties of a node. This is enclosed in {} square brackets.

Attr_list defines a list of attributes separated by a comma.

Attr represents a single attribute. This attribute can either be a specific shape attribute (with a system output action to print the shape's value) or a generic identifier.

Edge_definition defines an edge between two nodes in a graph. Edge_definition starts with an identifier, followed by an arrow "->", another identifier, optional edge attributes, and a semicolon.
Edge_attrs contains attributes for an edge enclosed in square brackets.

A_list defines a list of edge attributes separated by commas.

A_list_item represents a single attribute for an edge, which is a key-value pair with the value as a string.

Cluster_id is an optional rule that defines identifiers for clusters within the graph, prefixed with 'cluster_' or just regular ID's.

Lexer Rules

Digraph matches the literal string 'digraph'. This string is used to start the definition of a directed graph.

ID defines the pattern for identifiers, which can start with a letter or underscore, followed by any number of letters, underscores, or digits.

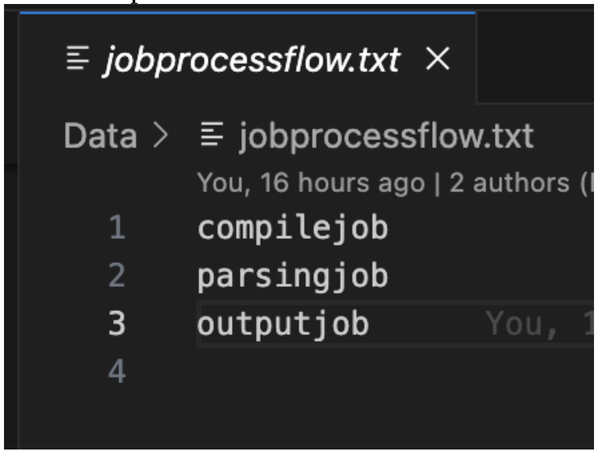
String defines the pattern for string literals, which are enclosed in double quotes.

WS matches whitespace characters. These whitespace characters can be spaces, tabs, carriage returns, and newlines. These whitespace characters are set to be skipped, which means they will not be passed onto the parser.

How each file is broken down by antlr

Antlr takes these grammar rules and breaks each case file (fsp1.txt, fsp2.txt, fsp3.txt) down in different ways based on what is parsed. All files use the visitorEdge_definition custom function in the customVisitor class to pull the edges out of the Concrete Syntax Tree (CST). After this, a vector named jobQueue is filled with parsed strings specific to each case, and these strings are written to a file named jobprocessflow.txt:

1. Fsp1.txt



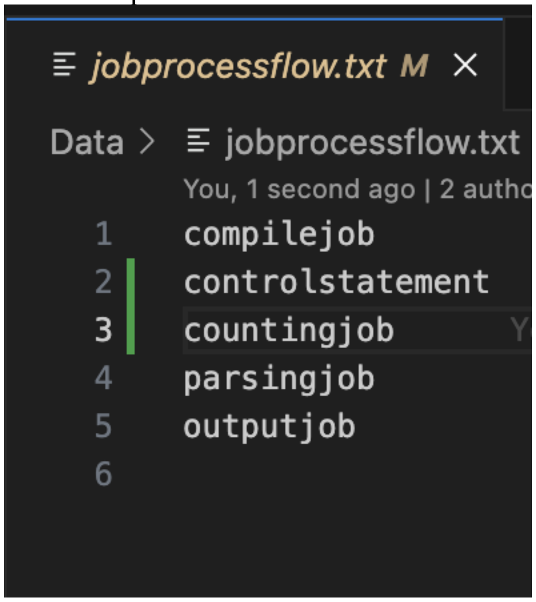
jobprocessflow.txt ×

Data > jobprocessflow.txt

You, 16 hours ago | 2 authors (

```
compilejob
parsingjob
outputjob        You, 1
```

2. Fsp2.txt



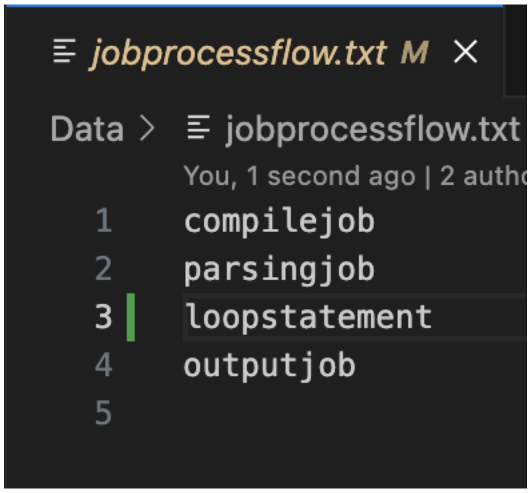
jobprocessflow.txt M ×

Data > jobprocessflow.txt

You, 1 second ago | 2 autho

```
compilejob
controlstatement
countingjob        Y
parsingjob
outputjob
```

3. Fsp3.txt



jobprocessflow.txt M

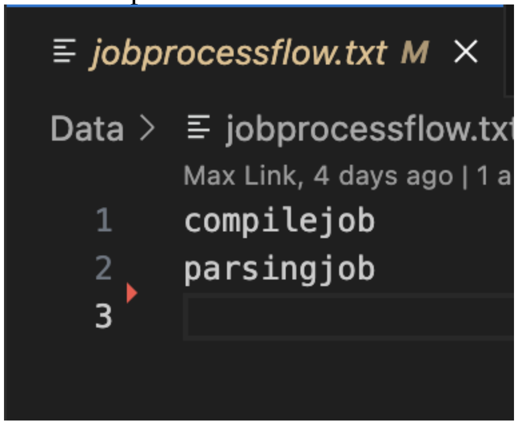
Data > jobprocessflow.txt

You, 1 second ago | 2 auth

```
compilejob
parsingjob
loopstatement
outputjob

```

4. Fsp-errors-test.txt



jobprocessflow.txt M

Data > jobprocessflow.txt

Max Link, 4 days ago | 1 a

```
compilejob
parsingjob

```

No outputjob logged in jobprocessflow.txt for fsp-errors-test.txt because of the specific errors in the flowscript.

Running Case 1: Compilejob→parsingjob→outputjob

If the code is running the fsp1.txt file – the base case – then the if statement conditionals are dodged and the jobs run in this order:

1. Compilejob
2. Parsingjob
3. Outputjob

Running Case 2: Compilejob → countingjob OR compilejob → parsingjob → outputjob

If a controlstatement is read from the jobprocessflow.txt file, then the code knows it should check a conditional statement. The interpreter class then calls the CondFlow() function, and compares its two integers fsCond1 and fsCond2. If fsCond1 is greater than fsCond2, the flow is compilejob→countingjob. If fsCond1 is less than fsCond2, then the flow is compilejob→parsingjob→outputjob. The fsCond1 and fsCond2 integers can be changed in interpreter.h.

Running Case 3: compilejob → parsingjob → parsingjob → parsingjob → outputjob

The loop logic can loop the second job, parsingjob.

Interpreter.cpp logic for looping before the ParsingJob call

If a loopstatement is read from the jobprocessflow.txt file then the code knows it needs to enter a loop for the parsingjob. Interpreter calls the loop function, and this loop function handles the looping of parsingjob with recursion. The loopHit.txt file is updated to alert parsingjob that it has entered the loop statement. A Boolean value is read from the loopbool.txt, if the value taken from this file is 0, then an if statement is entered that calls parsingjob. This if statement then calls the loop function with recursion.

Parsingjob.cpp logic for looping

Everytime parsingjob is executed, in the job completed callback the job reads the loopHit.txt file that interpreter filled out. Interpreter puts a 1 in this file, so that parsingjob's job completed callback finds the 1 value, and sets loopHit to 1. Now an if statement checks if loopHit is 1, and if it is, then the loopParsingJob is called.

loopParsingJob sets loopCond to 1, then reads from parsingJobLoopCounter.txt. ParsingJobLoopCounter.txt holds past values from past calls to the loopParsingJob function. Any past values recorded in this file are stored in the int pastInt variable. pastInt is then appended to loopCond. loopCond is then appended to parsingJobLoopCounter.txt for future run throughs in the loopParsingJob function. This keeps a running counter in essence. Once this running counter loopCond == 3, 1 is written to loopbool.txt.

Interpreter.cpp logic for looping after the parsingjob call

Once loopbool.txt holds 1, loopCond in the interpreter's Loop function reads from the loopbool.txt file and gets initialized to 1. Now that loopCond is 1, the recursion breaks.

ParsingJobLoopCounter.txt is set to zero for future looping cases. Loopbool.txt file is set to 0 for future looping cases.

Documented Flowscript Errors

I created a customerrorlistener class that interfaces with antlrs error reporting to log both lexical and syntactical errors into a json file.

Customerrorlistener.h:

```
#ifndef CUSTOMERRORLISTENER_H
```

```
#define CUSTOMERRORLISTENER_H

#include "antlr4-runtime.h"
#include "./lib/json.hpp"

using json = nlohmann::json;

class CustomErrorListener : public antlr4::BaseErrorListener {
private:
  json errorDetails;

public:
  void syntaxError(antlr4::Recognizer *recognizer, antlr4::Token *offendingSymbol,
           size_t line, size_t charPositionInLine, const std::string &msg,
           std::exception_ptr e) override;

  void writeErrorsToFile(const std::string &filename);

   std::vector<json> getErrors() const {
    if(!errorDetails.empty()){
     return errorDetails;
    }
    else{
      //TODO - logic to handle case with no errors
      // Create a JSON object with a "no errors" message
      json noError = {
        {"type", "No Error"},
        {"message", "No lexical or syntax errors found in flowscript file."}
      };

      // Return a vector containing this single JSON object
      return std::vector<json>{noError};

    }
  }
};
```

```
#endif // CUSTOMERRORLISTENER_H
```

Customerrorlistener.cpp:

```
#include "customerrorlistener.h"
#include <fstream>

void CustomErrorListener::syntaxError(antlr4::Recognizer *recognizer, antlr4::Token *offendingSymbol,
                                      size_t line, size_t charPositionInLine, const std::string &msg,
                                      std::exception_ptr e) {
    // Construct an error object and add it to the errorDetails array.
    json error = {
        {"line", line},
        {"charPositionInLine", charPositionInLine},
        {"offendingSymbol", offendingSymbol ? offendingSymbol->getText() : "null"},
        {"message", msg}
    };

    errorDetails.push_back(error);
}

void CustomErrorListener::writeErrorsToFile(const std::string &filename) {
    // Write the JSON array of errors to a file.
    std::ofstream file(filename);
    if (file.is_open()) {
        file << errorDetails.dump(4);  // Dump the JSON with an indentation of 4 spaces
        file.close();
    } else {
        std::cerr << "Unable to open file for writing JSON errors: " << filename << std::endl;
    }
}
```

I created a customerrorlistener object and attached this object to the antlr parser in customjob.cpp:

```
CustomErrorListener errorListener; //create error listener object

   //TODO -  add parsing here in customJobs execute function

   // Path to flowscript input file taken in
   std::ifstream stream("./Data/fsp-errors-test.txt"); // Adjust the path to the location of your .txt file
   if (!stream) {
       std::cerr << "Cannot open file: ../Data/fsp#.txt" << std::endl;
       return; // Handle the error as you see fit
   }
   antlr4::ANTLRInputStream input(stream);

   // Create a lexer that feeds off of the input stream
   FlowscriptGrammarLexer lexer(&input);

   // Create a buffer of tokens pulled from the lexer
   antlr4::CommonTokenStream tokens(&lexer);

   // Create a parser that feeds off the tokens buffer
   FlowscriptGrammarParser parser(&tokens);

   //attach the customerror listener to the parser
   parser.removeErrorListeners(); //remove default error listener
   parser.addErrorListener(&errorListener); //add custom error listen

   //attach the custom listener to the lexer
   lexer.removeErrorListeners(); //remove default error listener
   lexer.addErrorListener(&errorListener); //add custom error listener
```

I then ran the parsing with this line:

```
// Begin parsing at the root rule (replace 'script' with your actual root rule name)
   antlr4::tree::ParseTree *tree = parser.flowscript();
```

The flowscript errors are derived from this basic flowscript below

digraph JobGraph {

```
  compilejob -> parsingjob
  parsingjob - > outputjob;

  outputjob -> ;

}
```

This flowscript involves these lexical and syntactical errors:

1. Syntax error – missing semicolon in “compilejob→ parsingjob”
2. Lexical error – space between “->” charater in “parsingjob - > outputjob”
3. Syntax error – missing target job in “outputjob->;” line

These errors can be fixed by 1) adding a semicolon to the “compilejob-> parsingjob” line so that it is written as: “compilejob->parsingjob;”. 2) taking away the space in the “- >” character so that the line is: “parsingjob -> outputjob;”. 3) adding a target job to the line “outputjob ->;” so that you have “outputjob -> countingjob” or another type of job registered with the job system as the target.

The errors from the flowscript are recorded into a json file called “flowscript-errors.json” that looks like this:

```
[
    {
        "charPositionInLine": 13,
        "line": 4,
        "message": "token recognition error at: '- '",
        "offendingSymbol": "null"
    },
    {
        "charPositionInLine": 15,
        "line": 4,
        "message": "token recognition error at: '>'",
        "offendingSymbol": "null"
    },
    {
        "charPositionInLine": 2,
        "line": 4,
        "message": "mismatched input 'parsingjob' expecting {';', '['}",
        "offendingSymbol": "parsingjob"
    }
]
```