CMPT 473 | SOFTWARE QUALITY ASSURANCE

Assignment II Project Report

OCTOBER 16, 2015

SCHOOL OF COMPUTING SCIENCE, SIMON FRASER UNIVERSITY
Daphne Ordas, Matthew Chow, Young Yoon Choi (Justin), Wei Wang
Team DJW

Contents

Contact Information

Daphne Ordas	dordas@sfu.ca
Matthew Chow	mkc25 @sfu.ca
Young Yoon Choi (Justin)	yychoi@sfu.ca
Wei Wang	wwa53@sfu.ca

Software Specification

Name:	csv2json & json2csv
Developers:	Darwin, Harrigan, Mibamur
Stable release:	0.3.0 (Mar 18, 2014)
Written in:	Ruby
Operating system:	Windows, Linux
Available in:	English
Type:	Format converter
License:	Free open source license
Website:	https://github.com/darwin/csv2json

“csv2json” and “json2csv” are a set of programs that are format converters which are capable of the bidirectional transformation of CSV and JSON input. In addition, the software is able to acquire user specified separators in the CSV input for customization. The output JSON can be sorted in the CSV column order if requested by the user. Furthermore, the programs can be accessed as a library inside another piece of code to parse data. Besides the executable files, this project also implemented a few basic test cases which focus on random sample data conversion and custom separators. However, further systematical testing shall be conducted.

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

---- *Wikipedia https://en.wikipedia.org/wiki/Comma-separated_values*

In particular, the software is able to support a customized separator. The separator must be passed by an optional parameter -s, surrounded by single quotes. Only one character can be passed at a time.

JSON, (canonically pronounced /ˈdʒeɪsən/ jay-sən;[1] sometimes JavaScript Object Notation), is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is the primary data format used for asynchronous browser/server communication (AJAJ), largely replacing XML (used by AJAX).

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

---- *Wikipedia https://en.wikipedia.org/wiki/JSON*

In particular, the default JSON output has no order. Only in the above Wikipedia example the above format is included i.e. line breaks, spaces, and indentations, to help humans read JSON. The program is able to use -H to sort the output by the same order of the input CSV column.

Input Space Model

Parameter Environment Categories & Values

inputFileName

Funct: Specifies an input file.
Type: Enumeration
Value: {

`EMPTY`: This value will be converted to an empty string later in the script. Calling the program with empty input is regarded as incorrect input. Warning or error is expected;

`thisFileDoesNotExist.csv`: This is a fake file name, error expected;

`empty.csv`: This is an empty input file (the file is empty, but it has a name and can be passed to the program). This shall be accepted as a valid input and corresponding output shall be processed;

`misform.csv`: This is an incorrectly formatted input file. The format here shall be completely different than CSV. An example is changing the extension of a .JPEG file to .CSV as an input. Error is expected;

`misformHFew.csv`: This is an incorrectly formatted input file. This file is a CSV based format but only a few headers are missing. The program shall be able to read it and generate an error;

`misformHMore.csv`: This is an incorrectly formatted input file. This file is a CSV based format but only a few extra headers are added. The program shall be able to read it and generate an error;

`correctNorm.csv`: This is a default correct input file with commas as a separator. The data shall contain different cases of valid CSV format i.e. including spaces as data, comma as data, line break as data, double quotes as data etc.;

`correctCusSep.csv`: This is another correct input, but the only difference is the separator is changed to a user defined character. However, commas and this defined character shall appear as part of this data too;

}

seperatorOption

Funct: Specifies a customized separator option and its potential parameters.
Type: Enumeration
Value: {

`EMPTY`: This value will be converted to an empty string later in the script. It means the customized separator option is inactive. No further parameters related to this option will be added;

`-s '?'`: A valid customized separator is specified. This value also corresponds to the input file format;

`-s ?`: An invalid customized separator is specified. However, the parameter is contained in a pair of single quotes. Thus, an error is expected in this case;

`-s '??'`: An invalid customized separator is specified. However, the parameter supposed to contain a single character. Thus, an error is expected in this case;

`-s ''`: A valid customized separator is specified. In this case, an empty character is passed as a separator which means the input file will be sliced character by character. Everything will be put inside a JSON pair;

}

outputOption

Funct: Specifies an output file.
Type: Enumeration
Value: {

`EMPTY`: This value will be converted to an empty string later in the script. It means the output option is inactive. No further parameters related to this option will be added;

`-o`: An invalid output option is specified. As long as the output option is active, an output file name shall be passed later as the saved location. Thus, a single output option is invalid. An error is expected;

`-o output.json`: An invalid output option is specified. A corresponding file will be generated and potentially rewritten. This file will not be the metric to judge the output result since everything is logged into another designated output file;

`-o !@#$%^&.jpeg`: An invalid output option is specified. The passed file name is neither a valid name nor a correct extension. An error is expected;

}

headerOption

Funct: Specifies a customized header if it is required.
Type: Enumeration
Value: {

`EMPTY`: This value will be converted to an empty string later in the script. It means the customized header option is inactive. No further parameters related to this option will be added. However, an error is expected if ignoreOption is active while no header specified;

`-H`: An invalid header option is specified. This option is incomplete due to more customized headers were supposed to be passed after this. Error expected;

`-H -`: An invalid header option is specified. The minus sign means the headers passed here are fewer than the amount it was supposed to have. Error expected;

`-H =`: A valid header option is specified. Equal sign means the headers passed here are the exact amount that it is supposed to have. However, a standalone header is still regarded as invalid without ignoreOption inactive;

`-H +`: An invalid header option is specified. Plus sign means the headers passed here is more than the amount it was supposed to have. Error expected;

}

inputPath

Funct: Specifies a path added before the input file.
Type: Enumeration
Value: {

`EMPTY`: This value will be converted to an empty string later in the script. It means the path is relative to the current location. Error is expected since all input files are stored inside the input folder;

`/`: An invalid path is specified. It means that the path is root. Error is expected because all input files are stored inside the input folder;

`input/`: A valid path is specified. It direct input files inside the input folder;

`/thisPathDoesNotExist/`: An invalid path is specified. This path completely does not exist. Error is expected;

}

oracle

Funct: Specifies a corresponding oracle to each test case.
Type: Enumeration
Value: {

`emptyOracle.json`: This is an empty file with a JSON extension. Since this program will output nothing when errors occur, this file is designed to detect error;

`emptyJsonOracle.json`: This is an empty JSON file. The file is in JSON format and contains empty JSON data. It is designed to detect test cases with empty input files;

`correctNormOracle.json`: This file is the default correct oracle. Most test cases that expect to perform a correct conversion shall generate an output the same as this one. However, there may be a few rare cases not covered by this oracle that shall be regarded as correct also;

`correctCusSepOracle.json`: This is another correct oracle. The only difference is that the separators are changed to customized ones;

}

wrongOption

Funct: Specifies a completely wrong option to the program.
Type: Boolean
Value: {

A completely wrong option will be passed if it is active. The option is not in the support list and contains weird characters;

}

prettyOption

Funct: Specifies a well formatted output option.
Type: Boolean
Value: {

The corresponding oracle will be generated if pretty option is active. This option only changes the format to human readable output;

}

ignoreOption

Funct: Specifies using customized header accompanied with the headerOption.
Type: Boolean
Value: {

> Set to be true if customized headers are introduced in output. This option shall be accompanied with the -H option. Error is expected if it works in standalone;

}

helpOption

Funct: Specifies to display help information.
Type: Boolean
Value: {

> Set to be true if help information is displayed. The help and version options have the highest priority in this program, so it will ignore other input but display help or version if the options are active. If wrong options are ignored in this case, error is expected;

}

versionOption

Funct: Specifies to display version information.
Type: Boolean
Value: {

> Set to be true if version information is displayed. Help and version option have the highest priority in this program, so it will try to ignore other input but display help or version if the options are active. If wrong options are ignored in this case, error is expected;

}

Other Environments

- Machine Architecture: x86 / x64
- Operating System: Windows / Linux
- Ruby: 2.0 (Old version) / 2.1 (Current Version) / 2.1 Dev (Development version)

These types of environments can be setup before the execution of test cases. In this case, one can simply repeat the same cases (Script may change slightly to fit the OS) and record to analyze the results.

Constraints on Choice Combinations

`(inputFileName = “EMPTY”) => (oracle = “emptyOracle.json”)`

Unspecified input file is expected to be an error.

`(inputFileName = “thisFileDoesNotExist.json”) => (oracle = “EmptyOracle.json”)`

Wrong input file is expected to be an error.

`(inputFileName = “misform.csv”) => (oracle = “emptyOracle.json”)`

Wrong input file is expected to be an error.

`(inputFileName = “misformHFew.csv”) => (oracle = “emptyOracle.json”)`

Wrong input file is expected to be an error.

`(inputFileName = “misformHMore.csv”) => (oracle = “emptyOracle.json”)`

Wrong input file is expected to be an error.

`(separatorOption != “EMPTY”) && (seperatorOption != “-s ‘?’”) => (oracle = “emptyOracle.json”)`

If the separator option is active, there is only one call (-s ‘?’) which will result in a correct output. Thus, the rest are all erroneous options.

`(outputOption != "EMPTY") && (outputOption != "-o output.json") => (oracle = "emptyOracle.json")`

If output option is active, there is only one call (-o output.json) which will result in a correct output. Thus, the rest are all erroneous options.

`(headerOption != "EMPTY") && (headerOption != "-H =") => (oracle = "emptyOracle.json")`

If output option is active, there is only one call (-H =) which will result in a correct output. Thus, the rest are all erroneous options. In addition, there may be other cases where even if such conditions are met, overall output still will be an error. This is addressed in the ignoreOption constraints.

`(inputPath = "EMPTY") => (oracle = "emptyOracle.json")`

Wrong input path is expected to be an error.

`(inputPath = "/") => (oracle = "emptyOracle.json")`

Wrong input path is expected to be an error.

(inputPath = "/thisPathDoesNotExist/") => (oracle = "emptyOracle.json")

Wrong input path is expected to be an error.

wrongOption => (oracle = "emptyOracle.json")

Wrong option is expected to be an error.

!ignoreOption => (headerOption = "EMPTY")

When ignoreOption is inactive, the header option has no meaning. Thus, it shall be set to inactive also.

(inputFileName = "empty.csv") => (oracle = "emptyJsonOracle.json")

When the input is an empty file, all later experiments shall be conducted based on empty JSON outputs despite the correctness of the calling.

(inputFileName = "correctNorm.csv") => (oracle = "correctNormOracle.json")

When the input is a correct file, all later experiments shall be conducted based on the correct JSON output despite the correctness of the calling.

(inputFileName = "correctCusSep.csv") => (oracle = "correctCusSepOracle.json")

When the input is a correct customized separator file, all later experiments shall be conducted based on the correct customized separator JSON output despite the correctness of the calling.

helpOption || versionOption => (oracle = "emptyOracle.json")

Since help and version will override most of settings, the expected output shall be empty. Help and version information will not be logged. In this case, massive expected errors may not be detected (The error still happens, but the type of error may be unknown by the tests).

ignoreOption && (headerOption = "EMPTY") => (oracle = "emptyOracle.json")

ignoreOption expects a corresponding header option. If either option is missing, an error is expected.

(inputFileName = "correctCusSep.csv") && (separatorOption = "EMPTY") => (oracle = "emptyOracle.json")

When customized separator input is passed in, an inactive separatorOption will result in error. This is same as a wrong formatted input.

Input Specification

The following files will be prepared before the test script. Also, the test oracles shall be changed based on the expectations from individual input files. In general, the input shall contain complex structures and representative data. All files shall be stored in the same folder/directory. Since most of the files are from official test cases, correctness will be ensured.

Empty Input

File Name:	empty.csv
Oracle Name:	emptyJsonOracle.json

A file name with CSV extension, and the content of the file is empty.

Correct Input

File Name:	correctNorm.csv
Oracle Name:	correctNormOracle.json

With commas as a separator, the data of the file will contain double quotes, periods, commas (as data itself), line breaks, and normal combinations of strings. Also, the csv2json GitHub test folder shall act as a template. One should reference the existing testing case and modify its data to include more variations of input.

File Name:	correctCusSep.csv
Oracle Name:	correctCusSepOracle.json

The only change of this file is that the separator is changed to '|'. Meanwhile, the data will contain the bar such as Test | "Hello | World" so the output shall be Test: "Hello | World" instead of Test: "Hello", World: "…".

Misformatted Input

File Name:	misform.csv
Oracle Name:	emptyOracle.json

The input will be anything you like with a completely wrong format file. For example, you can manually change a JPEG extension to CSV. The input must be deviated from any potential correct CSV format.

File Name:	misformHFew.csv
Oracle Name:	emptyOracle.json

One must choose a correct input from the previous section and manually delete some headers while keeping the data columns unchanged.

File Name:	misformHMore.csv
Oracle Name:	emptyOracle.json

One must choose a correct input from the previous section and manually insert some headers while keeping the data columns unchanged.

Example Input

```
Daphne,Matthew,Justin,Wei
Ford,E350,"ac, abs, moon",3000.00
Chevy,"Venture ""Extended Edition""","",4900.00
Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

Section Conclusion

This program is designed for mostly reliable input, which implies that the user is expected to deliver a correct command with well-prepared source files as input. The testing is focused on validating the most common combinations of correct input. Meanwhile, common invalid callings are also tested. However, since there are endless possible invalid callings compared with much fewer valid inputs, the tests in this part are more limited.

Reflecting on the keyword selection, there are only limited enumerations of possible faulty input, where each one can stand for a type. For instance, for wrongOption, a random character string is passed to represent a wrong option, and there are potentially much more well-structured but still wrong options. A great example here is "-hell". This option is apparently not supported by the program official documentation. However, if it is in the command, help information will be displayed due to the starting letter "h". The author didn't take into account of this particular case.

Reflecting on the constraints, it was noticed that simple tuples of combinations are well discussed while complex ones are omitted. There are few rare cases that weren't captured by the constraints. One good example is the combination of fewer header inputs and the fewer customized headers option. These two faults combined together actually result in a correct output.

In regards to general discussion and observation, there are multiple ways to represent key attributes. This study initially uses Boolean for all possible attributes and uses more complex constraints to restrict these Booleans. It reduces the complexity of future test case generation since the program can inject simple strings when the attributes are true. However, the increased constraint system canceled these advantages. Thus, the final solution introduced enumeration attributes. Another point of discussion is about set constraints. It is easy to conclude something is incorrect if certain factors are observed, while extremely difficult to conclude a case covers major correct situations. To prove some cases are incorrect, only detecting a few faults is enough. On the other hand, to prove some cases are correct, all possible combination of factors shall be examined. Thus, in constraint setting, the experiment tends to only set up which situation is absolute an error condition.

Test Frames Generation

The test frames are generated by the ACTS tool. After parameter environment and constraints are inputted, ACTS uses the IPOG algorithm with strength 6, scratch mode, forbidden tuples constraint handling, and randomized don't care values. In this case, domain size ranges from 2 to 8 which means the combination of the parameters will be a set with 2 to 8 elements. If there was no such setting, the program would basically be asked to compute all permutations of a set size equal to all values summed up in parameters. This task will easily become impractical to solve even with constraints. Based on this, the program generated 255845 possible combinations in total and 5884 cases are valid from constraints. Execution time is 1.139 seconds to generate these combinations, which is a fairly short time. However, the study failed to compute graph representation to determine percentage coverage due to ACTS becoming unresponsive while computing.

ACTS was able to generate less test frames and still cover a fair confidence level. However, this particular frame will execute all possible valid cases. The generated cases are sorted in the order each parameters are inputted. A strong pattern is observed among all cases. An example is test cases A, B, C, and D will execute in order AXXX, AXXX, … AXXX then BXXX, BXXX, … BXXX and so on. This particular case makes the test results easier to track, because normally if one parameter is expected to generate an error, the other associated ones are more likely to generate an error too. However, due to the limited depth of combinations, there is some randomness introduced. For example, a maximum domain size 8 is restricted, and there is a set with 7 elements already. The last element will be chosen from a set {K_1, K_2, K_3}. Each candidate is chosen randomly instead of certain order. This approach increases the potential coverage in a limited depth setting.

Test Cases Generation

There are three parts included in test case generation: generating the cases, executing the cases, and generating the report. Each part is done by a piece of the program, and human control is only needed to start each process.

Generate Cases

Since the ACTS can export a CSV file which contains all the combinations of the cases, the generating program only needs to read the CSV line by line and put calling parameters together. Java was decided to be the language used, and the following is a major part of the code:

```java
while ((line = bufferReader.readLine()) != null) {
        Scanner lineScanner = new Scanner(line);
        lineScanner.useDelimiter(",");
        counter ++;

        // Read all parameters and store beforehand
        String inputFileName = lineScanner.next();
        String separatorOption = lineScanner.next();
        String outputOption = lineScanner.next();
```

```
            String headerOption = lineScanner.next();
            String inputPath = lineScanner.next();
            String oracle = lineScanner.next();
            String wrongOption = lineScanner.next();
            String prettyOption = lineScanner.next();
            String ignoreOption = lineScanner.next();
            String helpOption = lineScanner.next();
            String versionOption = lineScanner.next();

            // Translate headers
            if (headerOption.equals("-H -"))
                  headerOption = new String("-H Daphne,Matthew,Justin");
            if (headerOption.equals("-H ="))
                  headerOption = new String("-H Daphne,Matthew,Justin,Wei");
            if (headerOption.equals("-H +"))
                  headerOption = new String("-H Daphne,Matthew,Justin,Wei,Rob");

            // Construct calling
            paraString += inputPath.equals("EMPTY") ? "" : inputPath;
            paraString += inputFileName.equals("EMPTY") ? "":inputFileName+" ";
            paraString += wrongOption.equals("true") ? "-!@#$%^&* " : "";
            paraString += prettyOption.equals("true") ? "-p " : "";
            paraString += separatorOption.equals("EMPTY")?"":separatorOption" ";
            paraString += outputOption.equals("EMPTY") ? "" : outputOption +" ";
            paraString += ignoreOption.equals("true") ? "-k " : "";
            paraString += headerOption.equals("EMPTY") ? "" : headerOption +" ";
            paraString += helpOption.equals("true") ? "-h " : "";
            paraString += versionOption.equals("true") ? "-v " : "";
            System.out.println(paraString);

            // Generate scripts for execution and report
            String targetInput = "output\\output" + counter + ".txt";
            String targetOracle = "oracle\\" + oracle;
            writerCompare.println("@echo Case Number:"+counter+">> Report.txt");
            writerCompare.println("@echoCommandLine:"+paraString+">>Report.txt")
            writerCompare.println("start /b FC " + targetInput + " " +
                  targetOracle + " >> Report.txt");
            paraString += " > output/output" + counter + ".txt ";

            writerScript.println("start /b " + paraString);
            lineScanner.close();
            paraString = "csv2json ";
      }
```

Details are changed to fit in one line!

Since ACTS uses enumeration as a parameter type, most input can be directly passed to the calling as a string. However, the input order is not as efficient as expected. For example, the header option is before the ignore option but the former one needs the later one to function. Thus, the program records all input beforehand, so later random access can be achieved. In addition, ACTS has a problem receiving commas as part of parameter data. Therefore, a further translation of header options is done here. Also, part of the parameters are set in ACTS while the other parts are done in Java, where, for example, a wrong file name is specified as !@#$%^&.jpeg in ACTS, while a wrong option -!@#$%^&* is specified in Java. This kind of coupling shall be diminished in the future so ACTS and case generation can work independently. The actual calling generation simply concatenates all parameter strings. After, a few BAT scripts were written accordingly for execution and reporting.

Execute Cases

Execution is separated from reporting to prevent potential process synchronization problems such as two processes accessing a file at the same time. Each line of calling consists of the following format:

```
start /b csv2json /thisPathDoesNotExist/ -!@#$%^&* -p -o output.json
-h > output/output11.txt
```

A new process is started for each case and runs in the background. The corresponding output of each case will be logged with its ID inside a designated folder. This approach makes it much easier for later analysis and report. One single click can execute all test cases while outputs can be well formatted. All test cases take about 10G memory to run and a few minutes are needed to completely execute.

Generate Report

In the current Windows environment settings, the major problem of generating reports is process synchronization. The built in FC command cannot access files in read only mode and since the oracles of this application have a one-to-many relationship, massive FC reading to the same oracle will be denied.

```
@echo Case Number : 666 >> Report.txt
@echo Command Line : csv2json /thisFileDoesNotExist.csv -s '??' -o
output.json -k -H >> Report.txt
```

Each report line consists of the above few parts. The case ID will be logged first, then the comparison between output and designated oracle will be logged. Line difference will be generated if test cases are failed. Otherwise, a line with a no difference encountered statement will be added.

Although the report is executed after actual test execution, the synchronization problem still exists. Since there are only limited cases which are interesting, the final report is done manually by reading the output file and calling parameters. Detailed explanations are added later.

Testing Result

The following are cases that will be analyzed. These two statements determine that there will always be less interesting cases compared to the total:

- The superset of an error case is more likely to be an error case too.
- The subset of a correct case is more likely to be a correct case too.

The nature of this is basically that there will be an overwhelming amount of error cases compared with correct ones. Also, since programs generally reject faulty calling, most error cases will be handled correctly. Thus, the following interesting cases include all calling expected to generate correct output and a few faulty cases that didn't terminate as expected. Later cases can be screened by sorting output file by size, while the majority (over 95%) of cases output the same error message, so the remaining faulty cases will generate various lengths of output.

801 `csv2json input/empty.csv`

Expected Outcome:	emptyJsonOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Pass

802 `csv2json input/empty.csv -p -o output.json`

Expected Outcome:	emptyJsonOracleP.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Pass

803 `csv2json input/empty.csv -p -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Fail. The program should be able to pre-validate header matching.

804 `csv2json input/empty.csv -o output.json -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Fail Fail. The program should be able to pre-validate header matching.

805 `csv2json input/empty.csv -p -s '?'`

Expected Outcome:	emptyJsonOracleP.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Pass

806 `csv2json input/empty.csv -s '?' -o output.json`

Expected Outcome:	emptyJsonOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Pass

807 `csv2json input/empty.csv -s '?' -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Fail Fail. The program should be able to pre-validate header matching.

808 csv2json input/empty.csv -p -s '?' -o output.json -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Fail. The program should be able to pre-validate header matching.

1244 csv2json input/misformHFew.csv -p -k -H Daphne,Matthew,Justin

Expected Outcome:	emptyOracle.json
Actual Outcome:	CORRECT
Conclusion:	Pass. Testing tool design defect.

1260 csv2json input/misformHFew.csv -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail Fail. The program should be able to pre-validate header matching.

1596 csv2json input/misformHFew.csv -s '' -k -H Daphne,Matthew,Justin,Wei,Rob

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail. The program should be able to pre-validate header matching.

1644 csv2json input/misformHMore.csv -p -k -H Daphne,Matthew,Justin

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail Fail. The program should be able to pre-validate header matching.

2009 csv2json input/correctNorm.csv

Expected Outcome:	correctNormOracle.json
Actual Outcome:	correctNormOracle.json
Conclusion:	Pass

2010 csv2json input/correctNorm.csv -p -o output.json

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

2011 `csv2json input/correctNorm.csv -p -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

2012 `csv2json input/correctNorm.csv -o output.json -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	correctNormOracle.json
Actual Outcome:	correctNormOracle.json
Conclusion:	Pass

2013 `csv2json input/correctNorm.csv -p -s '?'`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyOracle.json
Conclusion:	Pass

2014 `csv2json input/correctNorm.csv -s '?' -o output.json`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyOracle.json
Conclusion:	Pass

2015 `csv2json input/correctNorm.csv -s '?' -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyOracle.json
Conclusion:	Pass

2016 `csv2json input/correctNorm.csv -p -s '?' -o output.json -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyOracle.json
Conclusion:	Pass

2017 `csv2json input/correctCusSep.csv -s '?'`

Expected Outcome:	correctCusSepOracle.json
Actual Outcome:	correctCusSepOracle.json
Conclusion:	Pass

2018 `csv2json input/correctCusSep.csv -p -s '?' -o output.json`

Expected Outcome:	correctCusSepOracleP.json
Actual Outcome:	correctCusSepOracleP.json
Conclusion:	Pass

```
2019 csv2json input/correctCusSep.csv -p -s '?' -k -H
     Daphne,Matthew,Justin,Wei
```

Expected Outcome:	correctCusSepOracleP.json
Actual Outcome:	correctCusSepOracleP.json
Conclusion:	Pass

```
2020 csv2json input/correctCusSep.csv -s '?' -o output.json -k -H
     Daphne,Matthew,Justin,Wei
```

Expected Outcome:	correctCusSepOracle.json
Actual Outcome:	correctCusSepOracle.json
Conclusion:	Pass

```
3804 csv2json input/misformHFew.csv -p -k -H -v
```

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail. Header option incomplete.

```
3805 csv2json input/misformHMore.csv -p -k -H -h
```

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail. Header option incomplete. Fail. The program shall validate input format.

```
3818 csv2json input/misformHMore.csv -p -s '' -k -H -h
```

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail. Header option incomplete. Fail. The program shall validate input format.

```
4021 csv2json input/correctCusSep.csv -p -s '?'
```

Expected Outcome:	correctCusSepOracleP.json
Actual Outcome:	correctCusSepOracleP.json
Conclusion:	Pass

```
4022 csv2json input/correctCusSep.csv -p -s '?' -o output.json -k -H
     Daphne,Matthew,Justin,Wei
```

Expected Outcome:	correctCusSepOracleP.json
Actual Outcome:	correctCusSepOracleP.json
Conclusion:	Pass

4023 csv2json input/correctCusSep.csv -s '?' -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	correctCusSepOracle.json
Actual Outcome:	correctCusSepOracle.json
Conclusion:	Pass

4024 csv2json input/correctCusSep.csv -s '?' -o output.json

Expected Outcome:	correctCusSepOracle.json
Actual Outcome:	correctCusSepOracle.json
Conclusion:	Pass

4025 csv2json input/empty.csv -p

Expected Outcome:	emptyJsonOracleP.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Pass

4026 csv2json input/empty.csv -p -s '?' -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Fail Fail. The program should be able to pre-validate header matching.

4027 csv2json input/empty.csv -p -s '?' -o output.json

Expected Outcome:	emptyJsonOracleP.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Pass

4028 csv2json input/empty.csv -p -o output.json -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Fail. The program should be able to pre-validate header matching.

4029 csv2json input/correctNorm.csv -p

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

4030 csv2json input/correctNorm.csv -p -s '?' -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

4031 csv2json input/correctNorm.csv -p -s '?' -o output.json

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

4032 csv2json input/correctNorm.csv -p -o output.json -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

4033 csv2json input/empty.csv -s '?'

Expected Outcome:	emptyJsonOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Pass

4034 csv2json input/empty.csv -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Fail. The program should be able to pre-validate header matching.

4035 csv2json input/empty.csv -o output.json

Expected Outcome:	emptyJsonOracleP.json
Actual Outcome:	emptyJsonOracleP.json
Conclusion:	Pass

4036 csv2json input/empty.csv -s '?' -o output.json -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyJsonOracle.json
Conclusion:	Fail. The program should be able to pre-validate header matching.

4037 csv2json input/correctNorm.csv -s '?'

Expected Outcome:	emptyOracle.json
Actual Outcome:	emptyOracle.json
Conclusion:	Pass

4038 csv2json input/correctNorm.csv -k -H Daphne,Matthew,Justin,Wei

Expected Outcome:	correctNormOracle.json
Actual Outcome:	correctNormOracle.json
Conclusion:	Pass

`4039 csv2json input/correctNorm.csv -o output.json`

Expected Outcome:	correctNormOracle.json
Actual Outcome:	correctNormOracle.json
Conclusion:	Pass

`4040 csv2json input/correctNorm.csv -s '?' -o output.json -k -H Daphne,Matthew,Justin,Wei`

Expected Outcome:	correctNormOracleP.json
Actual Outcome:	correctNormOracleP.json
Conclusion:	Pass

`4203 csv2json input/misformHFew.csv -p -s '' -k -H Daphne,Matthew,Justin,Wei,Rob`

Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail Fail. The program should be able to pre-validate header matching.

`4369 csv2json input/misformHMore.csv -k -H Daphne,Matthew,Justin`

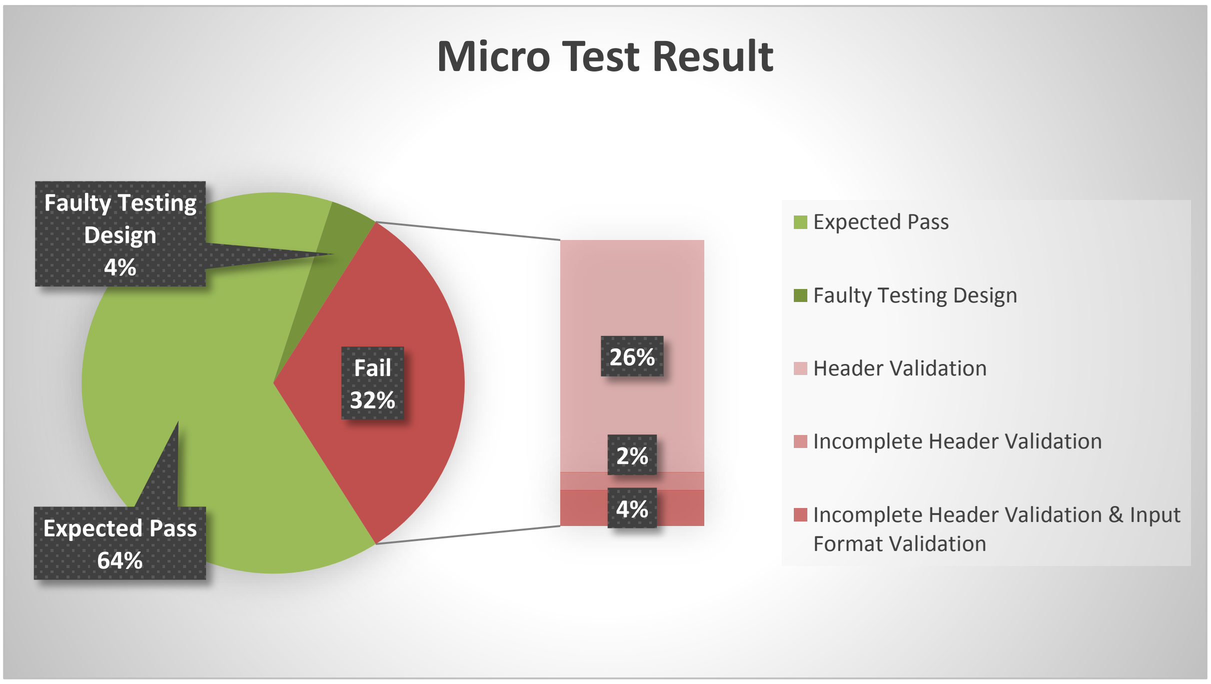
Expected Outcome:	emptyOracle.json
Actual Outcome:	INCORRECT
Conclusion:	Fail. The program should be able to pre-validate header matching.

`4613 csv2json input/misformHFew.csv -s ''`

Expected Outcome:	emptyOracle.json
Actual Outcome:	CORRECT
Conclusion:	Pass. Testing tool design defect.

Observations and Conclusions

	Macro	Micro
Total Cases	5884	50
Passed Cases	%99.73 (5868)	68% (34)
Failed Cases	%0.27 (16)	32% (16)

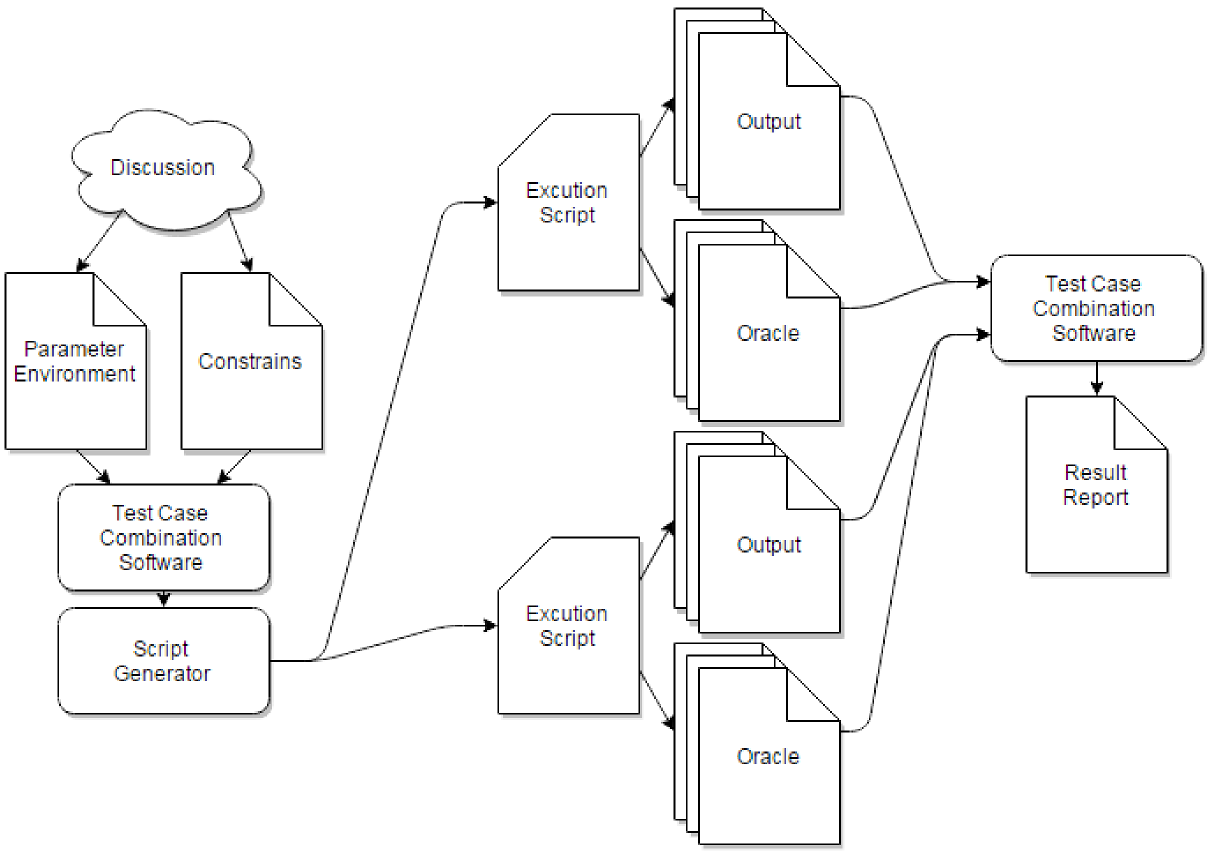


Again, it is restated that this program is designed for mostly reliable input. It cannot be understated that the user is expected to deliver a correct calling with well-prepared source files as input. As a result, the response to incorrect input is not handled well. There are two major concerns in regards to the test results:

- Error Display: The program has no user friendly error output message. It simply returns which part Ruby met the problem and a brief description. What was expected here was a detailed categorized error output with a help instruction and an ID, such that the user will be able to advance by following the instructions and professionals can track the error by ID.
- Header Handling: The program does not check the header matching either in the input file or by user input. If a mismatched header is passed, it will grab the data near the missing (or extra) headers as new header.

Another interesting observation is several combinations of faults can actually generate a correct result. The example here is that both the input file and user inputted header are missing exactly one header for each. Thus, they canceled each other's effect. An error was originally expected, but there was actually no error generated in output file. This is a type of a faulty designed test case, and this special situation shall be included in future study.

One more thing that should be stressed is the nature of test cases. To explain, a massive amount of handled error is observed over the correct execution of cases for a good designed program. This may introduce more difficulties for software testing. Since huge error is handled, the average pass rate will be relatively high. However, it cannot be concluded that the software is well designed. The analysis shall focus on "micro cases" which are the ones expected to pass but not focus on other bizarre cases. These tests are a true matter to ensure a robust system rather than handle combinations of "easy-detected" errors.



In general, the approach above is a well structured approach to generate combinational tests for various environments. The combination software can take care of a specified level of complexity while outputting high level CSV structures which can be applied in different post processors. The script generator can then output accordingly. One advantage here is that the script generator will have highly reusable code for logical parts. Only the format needs to be modified when testing on various environments. A specific example would be switching a BAT script to a SH script. Each test case corresponds to its own output and oracle which makes the later report analysis much easier. Only limited human interaction is needed during the whole process.