

# Exercise 2: Testing with Input Space Partitioning

Cole Stankov (301295209)

Saqib Aziz Dhuka (301338669)

CMPT 473

# Specification of the Program Under Test

For this assignment we decided to test a CLI application written in JavaScript for a CSV to Json converter (Application Github link: <https://github.com/julien-f/csv2json>).

## CSV specification (input file format):

CSV file format reference: <https://tools.ietf.org/html/rfc4180>

While there are various implementations of CSV format there is no exact formal implementation. This allows for a wide variety of interpretations of CSV files. The following specifications will follow the format that is typically implemented:

- For the header there may be one or more fields and they are separated by a comma.
- Each line should contain the same number of fields
- Spaces are included within the field
- The last field in the record shouldn't be followed by a comma
- Fields may or may not be enclosed in double quotes, however, if they are enclosed then the field cannot contain double quotes.
- Fields containing line breaks, double quotes and commas should be enclosed in double-quotes
- If a double quote is used in a double quote enclosed field then the double quote must be escaped by preceding it with another double quote.

ABNF grammar for a CSV file appears as follows:

```
file = [header CRLF] record *(CRLF record) [CRLF]
header = name *(COMMA name)
record = field *(COMMA field)
name = field
field = (escaped / non-escaped)
escaped = DQUOTE *(TEXTDATA / COMMA / CR / LF / 2DQUOTE) DQUOTE
non-escaped = *TEXTDATA
COMMA = %x2C
CR = %x0D ;
```

## JSON specification (output file format):

JSON file format reference: <https://tools.ietf.org/html/rfc8259>

- A JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).
- An object is considered as a collection of zero or more name/value pairs. The name is a string and the value is a string and the value is one of the primitive types mentioned previously.
- An array is an order of sequence of zero or more values.
- The literal names must be in lowercase (ex. true, false, null)

```

<Json> = <Object> || <Array>
<Object> = '{' '}' || '{' <Members> '}' || '{' <Members>, <Members> '}'
<Members> = <name: value>
<Array> = '[' ']' || '[' <Element> ']' || '[' <Element>, <Element> ']'
<Element> = <value>
<value> = string || number || boolean || null || <Array> || <Object>

```

## Parameters for csv2json:

- For this assignment we decided to leave out tests that we felt weren't focused on the processing of a csv file to a json object.
- The program csv2json contains options -d (dynamically typed), -s (separator), -t (tab separator), -help and the input and output fields.
- We decided to exclude the -help for this assignment. The -help was removed due to it not focusing on processing the csv files.
- We will be testing with the -d, -s and -t options and input and output fields.
- For -s we will use " " (space) as the custom delimiter.
- Some tests will contain no options and solely the input and output fields.
- Invalid options do not concern the flags since the program is expected to crash.
- Although csv2json allows for disk and stdin input and output we will only be testing disk for the scope of this assignment.

## Input CSV File:

- The input will be valid or invalid paths depending on the test.
- The input file that will be tested will contain either an empty file or non-empty file and will contain different delimiters as separators to test for -s flag and -t.
- We will not be testing for valid file extensions and it is assumed the input will be a .csv file since that is outside the testing scope.

## Output Json File:

- The output will be valid or invalid paths depending on the test.
- A Json file will be generated to the TestOutput/Files folder and will be compared to expected output files that we created.

## Testing specifications:

The testing platform is a script written in python 3.8.5 which will run the csv2json program with a series of tests that are labeled 1-9. The input files inside TestData/TestFiles will be used as input files. The test will write to output json files located in the TestOutput/File directory. The results are then compared to expected output files located in the TestData/ExpectedOutput directory. A message will also be written to the TestOutput/Messages directory and be compared to TestData/ExpectedMessages directory files. If both the output message and output file match for a test then the test is considered passed. Although test 7 and test 8 pass, they are testing invalid file paths which are expected to generate an output message with the specified error.

- Command to run the test suite: *python3 exercise2.py*
- Machine used: Linux

## Input Space Partitioning

We analyzed the -d, -s and -t option and the input and output field and their corresponding characteristics. Following what was discussed in class regarding input space partitioning, we came to the following conclusion for our input domain model:

### IO Environment

Input\_file : stdin, disk  
 Output\_file : stdin, disk  
 Valid\_input\_path : true, false  
 Valid\_output\_path : true, false

### Command options

Option\_d : true, false  
 Option\_s : true, false  
 Option\_t : true, false

### Input field

Fields\_separator : commas, tabs, none  
 Contents\_of\_file\_length : ZERO\_REC, GREATER\_THAN\_ZERO\_REC  
 Numeric\_and\_boolean\_vals\_present : true, false

## Constraints

1. Input type = "disk"
2. Output type = "disk"
3. option\_t = true => Option\_s = false
4. Option\_s = true => option\_t = false
5. Valid\_input\_path = false => Valid\_output\_path = true
6. Field\_separator = "none" => (Valid\_input\_path = false || Valid\_output\_path = false || Contents\_of\_file\_length = "ZERO\_REC")
7. (Valid\_output\_path = false) => (Numeric\_and\_boolean\_vals\_present = false && Option\_d = false && Option\_s = false && option\_t = false && Field\_separator = "none")
8. (Valid\_input\_path = false) => (Numeric\_and\_boolean\_vals\_present = false && Option\_d = false && Option\_s = false && option\_t = false && Field\_separator = "none")
9. Contents\_of\_file\_length = "ZERO\_REC" => Numeric\_and\_boolean\_vals\_present = false && Option\_d = false && Option\_s = false && option\_t = false && Valid\_input\_path = true && Valid\_output\_path = true && Field\_separator = "none"

Constraints 1 and 2 are needed to limit the input and output type to only disk due to us limiting the testing to only disk and removing stdin from the input space. Constraints 3 and 4 are used due to an overlap in characteristics for -t and -s since they both deal with separating the file by either a given delimiter or tab. Constraint 5 is to remove redundancy in test cases since we do not need to see how the program operates if both input and output paths are invalid at the same time. Constraint 6 is constraining the fact that the only way a file should have a "none" separator is if either the input or output file path is invalid or if the file is empty. Constraints 7 and 8 are used to limit the redundant test cases since if the input and output file paths are invalid the other options are no longer relevant to test. Lastly constraint 9 is used to remove all other characteristics that specify contents of a document or options that act on those contents for empty files.

## Test Generation using ACTS

The ACTS tool was used to generate each testframe characteristics for a given number of tests. It was then converted to actual tests created from scratch in python. The ACTS file can be found within the csv2json.xml file.

## Test Results

The ACTs generated 9 test cases which all worked as expected. The two test cases for invalid input and output were expected to show error messages since paths were invalid, but apart from that all test cases passed successfully. Without pair-wise testing there would have been 27 test cases (using ACT '-Dcombine=all' flag), which includes the constraints as well. Further removing the constraints, there would have been 576 test cases ( $2^6 * 3^2$  permutations, not including input and output type 'stdin' variations).

These are a lot of test cases and although we would've been able to go through each test case and remove redundancy, it would have been nearly impossible to filter out these 9 test cases that we got through ACTs. By working with functionality based techniques we were able to reveal much more important characteristics of parameters themselves. So for example, for our code, one parameter was the input file and instead of looking at the input of the file as a simple input file path (as in interface based approach), we were able to explore its characteristics like the contents of the files which are quite important in testing such programs. Despite this advantage, pairwise testing does have a drawback of not showing how more than three parameters together can affect the program's behaviour and the result. So for programs that work with only three parameters, pairwise testing would perhaps be more suited.

## Reflection

We found this assignment pretty straightforward, especially the ACTS software after understanding the features. After we found our test cases using ACTS we found that setting up the python script from scratch was not that difficult. However, the difficulty arose when we had to decide on the input space partitioning variables. We found ourselves correcting and adjusting the variables and their constraints quite often as we had to make sure that the completeness and disjoint property was maintained. The other surprisingly more difficult task was finding proper documentation for csv and json file specification. Although the ACTS software was easy to get used to, an in class working demo or a video reference would have cleared up some of the confusion we had. Overall, the assignment helped us to understand and implement input space partitioning technique on a real-world program.