Assignment 2 Report

Uzziah Eyee
CMPT 473 SPR 2019

Specification of SUT

csv2json https://github.com/baltimore-sun-data/csv2json

```
csv2json

Inputs:
  -dest string
        Destination file (default: stdout)
  -no-headers
        Return each row as an array
  -src string
        CSV Source file (default: stdin)

Outputs:
    File with JSON contents
    | Error message to Stdout
```

> Input-output specification for the program is sparse because the authors haven't provided more details like specific error messages. I have chosen not to infer unstated behaviour from the source code because that would no longer be black-box-testing.

CSV specification (input file format)

RFC4180 https://tools.ietf.org/html/rfc4180

```
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 ;as per section 6.1 of RFC 2234 [2]

DQUOTE =  %x22 ;as per section 6.1 of RFC 2234 [2]

LF = %x0A ;as per section 6.1 of RFC 2234 [2]

CRLF = CR LF ;as per section 6.1 of RFC 2234 [2]

TEXTDATA =  %x20-21 / %x23-2B / %x2D-7E
```

JSON specification (output file format)

RFC8289 https://tools.ietf.org/html/rfc8259

```
<Json> ::= <Object>
         | <Array>

<Object> ::= '{' '}'
           | '{' <Members> '}'

<Members> ::= <Pair>
            | <Pair> ',' <Members>

<Pair> ::= String ':' <Value>

<Array> ::= '[' ']'
          | '[' <Elements> ']'

<Elements> ::= <Value>
             | <Value> ',' <Elements>

<Value> ::= String
          | Number
          | <Object>
          | <Array>
          | true
          | false
          | null
```

> Source: https://github.com/cierelabs/yaml_spirit/blob/master/doc/specs/json-ebnf.txt

Input-output mapping.

- Since there's no inherent way to tell if a CSV file has a header or not, the program must rely on the `-no-headers` flag: presence => CSV has no header, absence => csv has a header.
- A CSV file => JSON array object

```
...          =>    [ ... ]
```

- With `-no-headers` flag, each CSV record => JSON Array element

```
a, b, c      =>    [ [a, b, c], [x, y, z] ]
x, y, z
```

- Without `-no-headers` flag, each CSV record => JSON Object with CSV header fields as keys

```
a, b, c      =>    [ {a: 1}, {b: 2}, {c: 3}
1, 2, 3              {a: 4}, {b: 5}, {c: 6} ]
4, 5, 6
```

- If csv has no header or records => JSON null element

```
                        =>     null
```

Input space partitioning

```
[System]
Name: csv2json

[Parameter]

-- Environment
Src_File_Exists (boolean) : TRUE, FALSE

-- Command Flags
No_Header (boolean) : TRUE, FALSE
Src (enum) : STDIN, DISKFILE
Dest (enum) : STDOUT, DISKFILE

-- CSV File Spec
File_With_Header (boolean) : TRUE, FALSE
Number_Of_Records (enum) : ZERO, GTZERO
Field_Type_In_Record (enum): ESCAPED, NONESCAPED, DONTCARE
Same_Field_Count_Per_Record (boolean) : TRUE, FALSE

[Constraint]
Src = "DISKFILE"
Dest = "DISKFILE"
Src_File_Exists = true
Field_Type_In_Record = "ESCAPED" => Number_Of_Records = "GTZERO"
Field_Type_In_Record = "NONESCAPED" => Number_Of_Records = "GTZERO"
```

Explaining the constraints

Constraint 1 is necessary because I intend to only use test data from an actual CSV file on disk. Constraint 2

because I only care about the case when the ouput is written to disk…I am using a file diff comparison for my test harness.

With constraint 3, I am assuming that the source CSV file already exists, because I don't care about the case when the file does not exists–I'm more concerned with the CSV-to-JSON parsing functionality of the SUT.

Finally, constraint 4 and 5 eliminate a nonsensical test scenario where there is no record but we test whether the records have an escaped or nonescaped field. In reality this will never happen.

Combinatorial test generation using ACTS

The ACTS tool GUI was used to create a system with parameters and constraints. The generated project files are `./csv2json.xml fireeye.log fireeye.log.lck`. Then the ACTS GUI was used to generate all pair testframes.

> As a much prefered alternative, I specified the system configuration in a file `./specfile` according to the rules in the ACTS manual. Then generated the all pairs test frames using the command below.
>
> ```
> java -Ddoi=2 -jar ./bin/acts_3.0.jar ./specfile ./testframes.txt
> ```

Creating testcases from test frames

Each testframe generated by ACTS was converted to an actual test case following the steps below.

1. A *TestFile* `./TestData/TestFiles/data1.csv` (where 1 is the testframe ID) is created according to the characteristics values in the test frame, i.e number of records, enclosed/unenclosed fields etc.
2. An *ExpectedOutput* file `./TestData/ExpectedOutput/data1.json` is created–it contains the manual conversion of the CSV file in (1) to JSON according the rules in the **Input-Output Mapping** section above.
3. A corresponding *ExpectedMessage* file `./TestData/ExpectedMessages/data1.log` is created–it contains any error messages expected from the conversion process. If no errors are expected, the file is empty.
4. The test harness (written in Go) is run to execute all the tests. Each test basically does the following: `csv2json` is called with the *Testfile* as input. The resulting output and messages are saved in `./TestOutput/Files/` and `./TestOutput/Messages/` respectively. Finally *ExpectedOutput vs. TestOutput* and *ExpectedMessage vs. OutputMessage* are compared using the diff tool. If no difference, then the test passed, otherwise it failed.

```
./bin/csv2json [-no-headers] --src ./TestData/TestFiles/data1.csv 1> ./TestOutput/F

diff ./TestData/ExpectedOutput/data1.json ./TestOutput/Files/data1.json

diff ./TestData/ExpectedMessages/data1.log ./TestOutput/Messages/data1.log
```

Running the test harness

```
./a2 [-h] [-verbose] [-sut="/path/to/csv2json"]
```

All flags are optional. `-h` describes all the options. `-sut="./bin/csv2json"` by default. `-verbose` lets you see exactly which *expected vs. output* difference caused a failure.

If you have Go setup, you can also run the harness from source using `go run main.go`

Questions and answers

How many tests did you generate?

8.

How many of these tests were successful/passing?

3.

How many tests would have been generated if you didn't use pairwise testing?

32.

> Using the ACTS flag -Dcombine=all and optionally -Dchandler=no

What tradeoffs did you make as a result of pairwise testing?

One benefit of pairwise test generation was the reduction in the number of tests generated. The downside was that interactions of t >= 3 parameters are ignored. For example the generated pairwise tests does not cover the case below.

```
Src_File_Exists,No_Header,Src,Dest,File_With_Header,Number_Of_Records,Field_Type_In_Rec
TRUE,FALSE,DISKFILE,DISKFILE,FALSE,GTZERO,DONTCARE,FALSE
```

This implies that bugs that might be caused by a combination of 3 or more factors go untested. In the above example, we would not know what happens with a CSV file containing mixed-field records.

Reflection on experience.

This assignment was faily involved. It covered the full lifecycle of testing a system from program selection, program specification, input partitioning, testframe generation, test implemenation and writing a testing harness, and finally report generation.

The large scope has made me appreciate the complexities in each stage of pairwise black-box testing. Firstly, that choosing system parameters is not always straight forward. It is more of an art than a clear-cut science because I subjectively choose criteria of the system is important to test. Secondly, how to merge theory and practice to get things done. Even though tools like ACTS do the heavy lifting of generating pairwise configurations, it is still up to the test engineer/developer to convert these into actuall test. Hence I had to write actuall code and a test harness.

Once I admitted the subjectivity of defining system parameter criteria, it was easy to specify the system for testframe generation. The hard thing was the program itself `csv2json` was not well documented. For example It did not have specified output error messages for various error inputs. This was one thing that could have helped speed up my specification time.