Processamento de Linguagens e Compiladores (3º Ano LCC)

# Project 1

Project Report

Bruno Dias da Gião
A96544

Maria Filipa Rodrigues
A97536

November 13, 2022

**Resumo**

Um ficheiro do tipo *Comma-Separated Values* é um formato de extrema importância, isto devido ao facto de ser texto plano, no entanto, as suas aplicações são limitadas, motivando a escrita de um programa em Python que, usando expressões regulares, converte qualquer ficheiro deste tipo para um ficheiro do tipo *JavaScript Object Notation*, que, devido à sua legibilidade e capacidade de ser *parsed* diretamente para um objeto de JavaScript ou de ser diretamente usado em contextos web, tem praticalidade mais acrescida. Isto, claro, representa uma conversão extremamente trivial e, devido à natureza de CSV, pode até permitir a conversão de qualquer ficheiro variante de Excel Binary File Format para um ficheiro JSON.

**Abstract**

A Comma-Separated Values file is an extremely important file type, this is due to it being plain text, however its applications are limited, motivating us to write a program in Python that, using regular expressions, converts any file with this file format into a JavaScript Object Notation file, which has more practical uses for both it's readability and being parsed into JavaScript Objects or be used directly into a website context. This, of course, has an extremely trivial conversion and due to CSV's nature, might even allow for the conversion of any XLS file variation into JSON.

# Contents

# Chapter 1

# Report

## 1.1 Introduction

### 1.1.1 CSV to JSON conversion

#### Introduction to the Report

This report is structured by the literate component itself 1 and by the code component which shall contain all the code used for this project 2.
Within the literate component we introduce the project with a historical background of the file formats being studied, why we believe this project is important, the original premise of the exercise and what can be done to expand it in a productive way that complements what is lectured in this class.
After this introduction, we move on towards implementations, design decisions and the finer technical and theoretical aspects of the project in the methodology section 1.2. Here we go into detail on regular expressions, the re module, how the properties of CSV were used to manipulate the file via regex and how the JSON file was created. In this very section there is also a segment dedicated to exemplifying the tests used to verify the proper functioning of the program.
The last section of the this chapter is the conclusion 1.3 where a brief summary of the results are gathered and insight is given on what could've been done and how this project can be expanded on.
Finally, we have the code component of the report which will contain all the code used during this project.
As standard for a report, the last pages are dedicated towards the bibliography.

#### Historical background of CSV and XLS

CSV, or Comma-Separated Values, is a well known file type for data storage, much like XL, or Excel Binary File Format, files in the sense that both represent tabular data, however, the major difference is that CSV is a plain text file, each line representing a row and each comma a column, thus, anything between commas, a cell. CSV has a great historical background as it predates the personal computer being supported by the IBM Fortran under 'OS/360' in 1972. [1] The file type name CSV itself, however, only came into existence in 1983.
XLS however only came into existence in 1987 with Excel's first Windows version, that being Excel 2. [2] In 2007, however, XLS was deprecated and the use of XML versions of Excel spreadsheets was promoted, thus introducing XLSX.

#### Historical background of JavaScript and JSON

JavaScript is a programming language famous for being one of the only capable of being used on the World Wide Web on the client side for web-page behaviour. It was first released in 1995 as an attempt to embed Java and Scheme (a very popular LISP dialect), but was decided it was best to create a new language in itself with syntax more similar to Java than to Scheme.
Most importantly for this report is a specific data structure included in JavaScript, 'objects', which are synonymous to associative arrays in other languages, thus, in the early 2000s, out of a need for stateless, real time server-to-browser communication protocols without Flash or Java applets, JSON files were created, these are code independent, as almost any language can parse it, but due to it's inspired syntax from JavaScript, it's is most popular for use with this programming language.

#### Importance of this project

Given the historical and practical significance of these three file formats, it is understandable how important it is to have a program that can seamlessly convert a CSV file into a JSON file, or, perhaps even, a program that converts a XLS file into a CSV file and thus allow for the previous program to convert it into JSON, that is what we aim to achieve with this project.
Again, considering how readable and easy to produce CSV files are, combined with how practical JSON files are, we can, from a plain text, send data in the text file from a server to a client, or display it on a web page.

#### Background of the Project

In this class, it was asked to solve one of five questions, the fifth, the one we chose, is the conversion of a modified CSV format that includes lists and aggregation functions into a JSON file using regular expressions. These lists can be of fixed size N or a size between N and M, the aggregation functions were left at the students criteria.

#### Expansions of the Project

Considering XLS is nothing more than a zipped file containing XML files, this conversion should be trivial provided the 'renaming' to the zipped file can be done, after which, theoretically, we should only need to find the cell data we need in the files inside the 'xl' directory.
Indeed, this project only requires the conversion of a modified CSV into JSON, however, considering the properties CSV and XLS share, it seemed wise to at least explore the possibilities for the previously mentioned conversion, XLS variations into CSV.
In reality, this endeavour is not as easy as it might appear due to how ambiguous excel cell data is in the xml files that constitute it, however it is still an interesting topic that, despite it there being a functionality in Excel itself, a Python script that could convert XLS to CSV and perform the script created for the project would be of great importance.

## 1.2 Methodology

### 1.2.1 Theoretical Background

#### Regular Expressions

Regular expression are an essential component of Computer Science, both theoretical and practical, this because they were originated in the context of Automaton Theory and Formal Languages. Because regular expressions represent regular languages, we can use these for 'pattern matching', allowing an user to easily find the first instance or all

instances of a given pattern, or instead to replace one, all instances or a given amount of matches, this allows for ease of use for various actions such as, converting file types correctly, converting from a formal language into machine language, sorting, managing data, and much more.

#### Python's re module

In order to work with regular expressions, Python has a built-in module called 're' which allows the use of some powerful functions that take a raw string containing a regular expression, a string to be analysed and produce, very efficiently, the desired result. The most important functions that will be used in this project are:

- `re.search(regex,string)`
- `re.split(regex,string)`
- `re.sub(regex,regex,string)`
- `re.subn(regex,regex,string,count=n)`

There is an increased focus on the sub and split functions as they are the most important functions to be used in this project, carrying into both the parsing of the file, creation of lists, and into the creation of the JSON file itself.

### 1.2.2 Practical component

#### Opening a CSV file

When the program is executed, it will ask the user to input the name of a valid CSV file.
In order to enforce such a prerequisite some defensive code was employed, performing the following instructions

1. Asks the User for an input that ends in .csv
2. Uses a regular expression -

   ```
   ([A-Za-z0-9\_\-]+)\.csv
   ```

   This makes sure that the input refers to a CSV file.
3. Saves the first group from the matched pattern
4. Opens the file, thus checking if it exists

#### Finding all Lists

Finding lists and parsing them so they are ready to be worked with, might be one the aspects of this project, that, alongside the manipulation itself of the lists and the attempt at converting XL files to CSV that required more work than most.
In order to do so however the following structure was adhered to:

1. Uses a regular expression -

   ```
   ([A-Za-z0-9 \_\-]+){([0-9]+)(,([0-9]+))?}(::[A-Z]+)?
   ```

as an argument to the function findall(), in order to locate all lists in the CSV file

2. Iterates through the results saving the first group as N
3. If there is an M, saves that value as the largest
4. Uses a regular expression -

```
(?<=\,)(?=\,)|(?<=\}\,)(?=\,)|(?<=\,\,)(?=)
```

   as an argument to the sub function, thus replacing all ‘,,’ patterns with the name of the list
5. Saves all the groups as flags in an associative array
6. Makes the list of saved groups presentable and in the same format as the list creation
7. Iterates through the results again and removes list creations from the headers

**Writing to a file**

The conversion itself of the CSV file to the JSON file is done via writing the structure of the JSON file to a buffer and plugging in the desired contents, this a very trivial and standard implementation, requiring only to open the file in write mode, creating each element of the JavaScript object and promptly writing it to the JSON file. However, plugging in the contents, in the solution to this problem, the reader may find, in lines 47-50 and in lines 131-148, code that suggests the usage of regular expressions to plug the values and lists into the buffer, that is precisely what happens in the code.
Doing so is nothing more than finding a pattern in how the elements of the JSON file are written, after which we only need to use the re.sub function and thus have a buffer with all the JSON information as required.

**Creating each element of the Object**

To create the object’s elements we need to parse each row of the CSV file, done so with a function in 2.1,*conv_csv_json(content)*, which, through the use of flags and some temporary data structures, we are able to either write to the buffer or create a list as a result.

**Flags and Data Structures Used** This function has the following arguments: *content, headers, and flags*. *Content* contains a line to be parsed, *header* is the first line of the CSV file with the alterations previously mention if lists exist, and the *flags* an associative array containing the minimum amount of elements, the maximum amount of elements, and the aggregation function to be applied, if these do not exist, they are to be ignored.
The content argument is converted into list format as the ‘new’ data structure, *tmp_head* is a copy of the headers variables so we can change the values in it, without compromising the good functioning of the the program on other rows, and *tmp_array* is the data structure that will be used in order to store the values of lists.
This function also uses some flags that assure the proper readability of the code and efficiency during the loops, namely, *flag* which checks if during the iteration of the row, a list was found, *flagM* which checks if the upper bound is M or N, *flagAg* which verifies if after the list is created a function needs to be applied, and *flagErr* that indicates whether or not an error was found during the reading of the CSV file. This program also uses *curr_check* to count the difference between the size of *tmp_array*’s lists and M.

**Iterating and Parsing the List** From line 54 to line 127 of the code 2.1, the program will iterate the contents of *tmp_head* in search of repeated, sequential elements.
When such a pattern is found, the flag variable is updated, the string that is repeated is stored in the *test* variable, *tmp_array* is updated to include *test* as a key and a list associated to that key, finally, the *flags* argument is processed.
Now that all states are set, the program can begin to create the list, which is done via the successive removal of elements until test is no longer in *tmp_head*. By choice, the program only accepts integers as list elements, otherwise it will raise an error by updating the error flag.
However, if the M Flag is set to true, there is a possibility that the element is a NULL string, in which case we must verify if it is so from:

$$N <= el <= M$$

When, finally, we have no instances of *test* in *tmp_head*, we can reset the flags, insert the resulting list to the content data structure new, and reinsert test into *tmp_head*.
Having done so, we need only continue iterating through the list.

**Using Aggregation Functions** After a list is fully created, the program only needs to use that list as an argument to another previously defined function. We chose that it only made sense to allow for SUM, COUNT, AVG, MAX and MIN, easy to implement aggregation functions from SQL. In order to ease this process, we used the built-in python function *eval(string)* to perform these procedures.

### 1.2.3 Testing the code

In view of proving the proper functioning of the program, some exemplifying tests were used.

#### Inputs

- The following input was a CSV 'database' that one of the Co-Authors of this report used in order to coordinate preferences for group and individual tasks in an association's department. In order to protect the identities of the people in the 'database', the names will be replaced with identifying numbers.

Listing 1.1: Database Input1.4

```
1 Name,Group_Pref{3},,,,Task_Pref{3},,,
2 0,0,1,2,0,0,0
3 1,0,0,1,0,1,0
4 2,0,1,0,1,0,1
5 3,1,0,0,0,0,0
6 4,1,0,0,0,0,0
7 5,1,1,2,0,1,0
```

Such that 0 represents no interest in the indexed task, 1 represents interest, and 2 meaning increased interest.

- The following test, however is a more generic input that tests variable size lists.

Listing 1.2: Students input1.5

```
1 Number,Name,Course,Grades{3,6},,,,,,,Area
2 1,Name1,Mathematics,19,20,14,10,,,Computer Science
3 2,Name2,Philosophy,20,19,13,19,20,10,Political Science
4 3,Name3,Philosophy,20,18,16,,,,Political Science
5 4,Name4,Philosophy,20,20,17,18,19,,Natural Languages
```

- The final test consolidates all the previous tests, having both varying sized lists, full lists, and aggregation functions over both of these forms of functions.

Listing 1.3: Students with averages input1.6

```
Number,Name,Course,Grades{3,6},,,,,,,Area,Average{3,6}::AVG,,,,,,
1,Name1,Mathematics,19,20,14,10,,,Computer Science,19,20,14,10,,
2,Name2,Philosophy,20,19,13,19,20,10,Political Science,20,19,13,19,20,10
3,Name3,Philosophy,20,18,16,,,,Political Science,20,18,16,,,
4,Name4,Philosophy,20,20,17,18,19,,Natural Languages,20,20,17,18,19,
```

### Outputs

The program works as expected as can be seen in the output to the previously defined inputs.
The outputs are as follows:

Listing 1.4: Database output1.1

```
[
        {
                "Name": "0",
                "Group_Pref": [0, 1, 2],
                "Task_Pref": [0, 0, 0]
        },
        {
                "Name": "1",
                "Group_Pref": [0, 0, 1],
                "Task_Pref": [0, 1, 0]
        },
        {
                "Name": "2",
                "Group_Pref": [0, 1, 0],
                "Task_Pref": [1, 0, 1]
        },
        {
                "Name": "3",
                "Group_Pref": [1, 0, 0],
                "Task_Pref": [0, 0, 0]
        },
        {
                "Name": "4",
                "Group_Pref": [1, 0, 0],
                "Task_Pref": [0, 0, 0]
        },
        {
                "Name": "5",
                "Group_Pref": [1, 1, 2],
                "Task_Pref": [0, 1, 0]
        }
]
```

Listing 1.5: Students output1.2

```
[
        {
```

```
                "Number": "1",
                "Name": "Name1",
                "Course": "Mathematics",
                "Grades": [19, 20, 14, 10],
                "Area": "Computer Science"
        },
        {
                "Number": "2",
                "Name": "Name2",
                "Course": "Philosophy",
                "Grades": [20, 19, 13, 19, 20, 10],
                "Area": "Political Science"
        },
        {
                "Number": "3",
                "Name": "Name3",
                "Course": "Philosophy",
                "Grades": [20, 18, 16],
                "Area": "Political Science"
        },
        {
                "Number": "4",
                "Name": "Name4",
                "Course": "Philosophy",
                "Grades": [20, 20, 17, 18, 19],
                "Area": "Natural Languages"
        }
]
```


Listing 1.6: Students average output1.3


```
[
        {
                "Number": "1",
                "Name": "Name1",
                "Course": "Mathematics",
                "Grades": [19, 20, 14, 10],
                "Area": "Computer Science",
                "Average": "15"
        },
        {
                "Number": "2",
                "Name": "Name2",
                "Course": "Philosophy",
                "Grades": [20, 19, 13, 19, 20, 10],
                "Area": "Political Science",
                "Average": "16"
        },
        {
                "Number": "3",
                "Name": "Name3",
                "Course": "Philosophy",
                "Grades": [20, 18, 16],
                "Area": "Political Science",
                "Average": "18"
```

```
25          },
26          {
27                  "Number": "4",
28                  "Name": "Name4",
29                  "Course": "Philosophy",
30                  "Grades": [20, 20, 17, 18, 19],
31                  "Area": "Natural Languages",
32                  "Average": "18"
33          }
34  ]
```

### 1.2.4 Preprocessing XL files

Throughout this report it was mentioned that our project would intend to perform efficient conversions of Excel Files to CSV files, a conversion that at first glance seems trivial, however it was unanimously decided it would be a good idea to not go ahead with the development of a preprocessing component to this program. This decision will be explained in a following section 1.2.4

#### How it can be done

If one wishes to convert an Excel file into a Comma-Separated Values file, one would need to find a way to access the XML files that constitute any XL file, normally this can be done via renaming the XL file so it's extension is '.zip', this can be done with a python script that uses two 'os' functions:

- `os.path.splitext()`
- `os.rename()`

Now that one has access to the XML files, we are going to use two of them:

- `./xl/worksheets/sheet1.xml`
- `./xl/sharedStrings.xml`

The keen user can tell that both these files are very notorious since there are some characteristics and rules, such that:

1. Any sequence of digits/numerical values are stored in the worksheets.xml file;
2. Any String is located in the sharedStrings.xml file;
    (a) These strings are stored in worksheets.xml as indexes

In which case, preprocessing XL files seems to be nothing more than the use of some regular expressions, such as one that finds the location and value of each cell and stores that as key and value in an associative array.
After which one would iterate through the sharedStrings file, use regex to save the values via tags, and replace the indexes in the first dictionary with the strings.

#### Complications and why it was not achieved

Even though this was attempted it was scrapped as finding a solution to the problem of recognising whether or not a digit in the sheet file is an index or a numerical value is of extreme difficulty.
Thus we realised this task could only be done via either an intelligent system or access to Excel's source code.
For such a reason this idea was put on hold and it was decided to not implement it within the scope of this project.

## 1.3 Conclusion

This project aptly tested understanding of Python, knowledge of regular expressions, our creativity, our teamwork, and, with the writing of this document, our understanding of what we accomplished and what we could've done to improve it.
Indeed, the reader may find some issues in our code 2.1, namely, as we see it, how unreadable some segments of the code are due to how verbose and compact they are, particularly the segments of code where Lists are being processed, despite the code being well commented, in hopes of lightening the effects of such a 'flaw'.
However, even though we accomplished what we set out to achieve with what was proposed for the project, as was mentioned in 1.2.4 the last section before this conclusion, we were not able to implement preprocessing of XL files in our project. Which is understandable considering the time frame we had to operate under and the requirements such a task really had, however, of course, it is an important analysis to make, knowing that preprocessing is a very important requirement, it is, indeed, an important aspect that can be worked on in the future. Provided there is time and resources to understand in depth how Excel stores and loads data. Not only this, but perhaps with an enhancement to the readability of the code, this could be a useful piece of software, as was shown to be in the data example 1.1 1.4.
Having created a program that correctly converts the required CSV file format into a JSON file format using regular expressions, python, and good programming practices, and by also ellucidating on how complex some seemingly trivial tasks can be, we believe that this project was a major success, having far exceeded our expectations.

# Chapter 2

# Appendix

## 2.1 Code

Listing 2.1: Source Code for the Project's Solution

```
import re
import sys


# SECTION FOR AGGREGATION FUNCTIONS
def SUM(li):
    res = 0
    for el in li:
        res += el
    return res

def COUNT(li):
    return len(li)

def AVG(li):
    return SUM(li) // COUNT(li)

def MAX(li):
    res = li[0]
    for el in li:
        if el > res:
            res = el
    return res

def MIN(li):
    res = li[0]
    for el in li:
        if el < res:
            res = el
    return res

# FUNCTION THAT CREATES THE JSON FILE
def conv_csv_json(content, headers, flags):
    # SECTION 1 - Preamble
    new = re.sub(r'([A-Za-z ]*)\n',r'\1',content)
    new = re.split(r',',new); # NOTE separates content by commas
    json_object = ""; tmp_array = {}
```

```
    tmp_head = headers.copy(); # NOTE safeguarding the headers list
    flag = False; i = 0
    curr_check = 0
    flagM = False; flagAg = False; flagErr = False
    # NOTE flags meanings:
    # flag <- List has been found
    # flagM <- upper bound is M and not N
    # flagAg <- Aggregation function to be applied at end
    # flagErr <- Error was found
    patternreg = "\t\"qwe\": \"yui\",\n\t"
    patternlis = "\t\"asd\": hjk,\n\t"
    patternerg = "\t\"zxc\": \"nm,\"\n\t"
    patterneli = "\t\"123\": 789\n\t"

    # SECTION 2 - List SEARCHING
    # A for does not provide enough control
    if flags:
        while i < (len(tmp_head)-1) and not flagErr:
            if tmp_head[i] == tmp_head[i+1]: # NOTE we found a list
                flag = True; test = str(tmp_head[i]); tmp_array[test] = []
                N = int(flags[test][0])
                try:
                    if (flags[test][1]):
                        try:
                        # Checks if the list is of N,M format
                            M = int(flags[test][1])
                            flagM = True
                            try:
                            # Checks if the list has an agreg function
                                if flags[test][2]:
                                    agreg = re.sub(r'::([A-Z]+)',r'\1',
                                                flags[test][2])
                                    flagAg = True
                            except IndexError:
                            # If there is no agreg function
                                agreg = ''
                                flagAg = False
                        except ValueError:
                        # There is an agreg function
                            M = N
                            agreg = re.sub(r'::([A-Z]+)',r'\1',
                                        flags[test][1])
                            flagM = False; flagAg = True
                except IndexError:
                    flagM = False; flagAg = False;
                    M = N; agreg = ''
            while flag and not flagErr:
                if (test not in tmp_head):
                # NOTE that there must be spaces in varying size lists,
                # TRIVIAL implementation
                # NOTE This is due to removing from tmp_head
                    if flagAg:
                        try:
                            tmp_array[test] = eval(agreg+'('+
                                                str(tmp_array[test])+')')
                        except SyntaxError:
                            flagErr = True
                    # Reset flags
                    flag = False; flagM = False; flagAg = False;
```

```
                    new.insert(i,tmp_array[test])
                    tmp_head.insert(i,test)
                    # Reset Values
                    curr_check = 0
                    del N; del M; del agreg; del elem_test
                if flag:
                    tmp_head.pop(i)
                    # Regular Search
                    if tmp_array[test]:
                            try:
                                elem_test = new.pop(i)
                                tmp_array[test].append(int(elem_test))
                            except ValueError:
                                if (curr_check == 0):
                                    curr_check = len(tmp_array[test])
                                if (flagM and elem_test == '' and
                                 curr_check >= N and
                                curr_check <= M):
                                    curr_check += 1
                                else:
                                    flagErr = True
                # NOTE the try does remove the value from new
                # Thus it is safer to remove to a safe variable
                    else:
                        # Initial Search
                        try:
                            tmp_array[test].append(int(new.pop(i)))
                        except ValueError:
                            flagErr = True

            i= i+1; # NOTE iterate the rest of the content

    # SECTION 3 - Writing to buffer

    if not flagErr:
        for i, v in zip(tmp_head[:-1], new[:-1]):
            if (isinstance(v,list)):
                json_object += re.sub(r'asd\": hjk',
                                      r''+i+'\": '+str(v)+r'',
                                    patternlis,count=1)
            else:
                json_object += re.sub(r'qwe\": \"yui',
                                      r''+i+'\": \"'+str(v)+r'',
                                      patternreg,count=1)
        if (isinstance(new[-1],list)):
            json_object += re.sub(r'123\": 789',
                                  r''+tmp_head[-1]+'\": '+str(new[-1]),
                                  patterneli,count=1)
        else:
            json_object += re.sub(r'zxc\": \"nm,',
                                  r''+tmp_head[-1]+'\": \"'+str(new[-1])+r'',
                                  patternerg,count=1)
    else:
        raise ValueError
    return json_object


# MAIN FUNCTION
def main():
```

```
156
157     # SECTION 1 - Getting the csv file
158     file = input("Insert name of file:\n>> ")
159     file_test = re.search(r'([A-Za-z0-9\_\-]+)\.csv',file)
160     if not file_test:
161         sys.exit("File is not a CSV file")
162     file_name = file_test.group(1)
163     del file_test
164     # NOTE
165     try:
166         f = open(file, 'r')
167     except OSError:
168         raise OSError; exit(1)
169         # NOTE tells the user the file doesn't exist
170     lines = f.readlines(); f.close()
171
172     # SECTION 2 - parsing lists
173     lines[0] = re.sub(r'([A-Za-z ]*)\n', r'\1', lines[0])
174     headers = lines.pop(0)
175     tst = re.findall(r'([A-Za-z0-9 \_\-]+){([0-9]+)'
176                      '(,([0-9]+))?}(::[A-Z]+)?',headers)
177 # 'Notas{3,5}::AVG'
178 # (NOTAS) (3) '' '' ''
179     matches_list = []
180     flags = {}
181     if tst:
182         for x in tst:
183             N = int(x[1])
184             if (x[3]):
185                 N = int(x[3])
186             headers = re.sub(r'(?<=\,)(?=\,)|'
187                              '(?<=}\,)(?=\,)|(?<=\,\,)(?=)',
188                              x[0],
189                              headers,
190                              count=N
191                              )
192             x = list(x)
193             flags[x[0]] = []
194             if x[1]: # N
195                 flags[x[0]].append(int(x[1]))
196             if x[3]: # M
197                 flags[x[0]].append(int(x[3]))
198             if x[-1]: # Function
199                 flags[x[0]].append(x[-1])
200             x.insert(1,'{');x.insert(4,'}')
201             x.pop(5)
202             matches_list.append(''.join(x))
203         headers = re.split(r'(?<!{\d),(?!\d})', headers)
204         for x in matches_list:
205             headers.remove(x) # List creation removed
206     else:
207         headers = re.split(r',', headers)
208     final = lines.pop(len(lines)-1)
209
210     # SECTION 3 - Creating a buffer with JSON file and writing to it
211     with open(file_name+'.json', 'w+') as f:
212         f.write('[\n')
213         for line in lines:
214
```

```
215             js_object = "\t{\n\t"
216             try:
217                 js_object += conv_csv_json(line, headers, flags)
218             except ValueError:
219                 sys.exit("Incorrectly made CSV file\n")
220             js_object += "},\n"; f.write(js_object)
221         js_object = "\t{\n\t"
222         try:
223             js_object += conv_csv_json(final, headers, flags)
224         except ValueError:
225             sys.exit("Incorrectly made CSV file\n")
226         js_object += "}\n]"; f.write(js_object); f.close()
227     return 0
228
229
230 # SCRIPT TO BE EXECUTED
231 if __name__ == '__main__':
232     main()
```

# Bibliography

[1] IBM. Ibm fortran program products for os and the cms component of vm/370 general information (first ed.). http://bitsavers.trailing-edge.com/pdf/ibm/370/fortran/GC28-6884-0_IBM_FORTRAN_Program_Products_for_OS_and_CMS_General_Information_Jul72.pdf, 1972. [Online; Last accessed as verification on 2022/11/11; Info found on page 17].

[2] MICROSOFT. History of microsoft 1987. https://learn.microsoft.com/en-us/shows/history/history-of-microsoft-1987, 2009. [Online; Last accessed as verification on 2022/11/11; Info found in video at timestamp 3:40].