COMPARISON PROGRAM
(VERDICT – VERsatile Difference and Compare Tool)

PURPOSE

To provide the user with a program that performs pair wise comparison of ASCII files. This program is flexible enough for the users to specify at run time how data will be parsed and compared.

HOW TO RUN THE PROGRAM

This program runs as a command line application. From the command line, the user enters 6 arguments exactly in this format:

VERDICT <INI File> <Primary File> <Primary File Type> <Secondary File> <Secondary File Type> <Output File>

- INI File: A file with detailed description of data fields in the file: data names, column position to start reading data values, data types, fields' width, and decimal places. See "HOW TO CREATE AN ENTRY IN THE INI FILE" below.
- Primary File: The file name of the primary file that is used in the comparison.
- Primary File Type: The file type of the *Primary File*. This file type must correspond to an entry in the *INI File*. The user previously has to create this entry in an *INI File* based on a predefined rule.
- Secondary File: The file name of the secondary file that is used in the comparison.
- Secondary File Type: The file type of the *Secondary File*.
- Output File: The file name of the output file that stores the comparison result between the primary and secondary file.

HOW TO CREATE AN ENTRY IN THE INI FILE

Constraint: the two files being compared need to have similar but not identical formats. The tokens for comparison are consistently located and available in both files.

To create an entry for a file type, first make up a unique name for a file type and enclose it in square brackets. Next, enter other entries for the file type such as a Catch string, Discriminator ID line, Discriminator tokens line, Data ID line, Data tokens line, and Unmatching lines limit. In each of these entries, enter the detail descriptions.

The INI File Format:

****
[file type]

Catch string 1 = <start position 1>:"string literal 1" [, <start position 2>:"string literal 2"]
…
Catch string 20 =

Discriminator ID line 1 = <start position 1>:"[!] string literal 1" [, <start position 2>:"string literal 2"]
…
Discriminator ID line 10 =

Discriminator tokens line1 = [!]<field name>:<start position>-<data type><field width>[.<decimal places>], …
…
Discriminator tokens line 10 =

Data ID line 1 = <start position 1>:"string literal 1" [, <start position 2>:"string literal 2"]

…
Data ID line 20 =

Data tokens line 1 = [@]<field name>:<start position>-<data type><field width>[.<decimal places>], …
…
Data tokens line 20 =

Unmatching lines limit:

****

Catch string: the user enters the description of a string that they want the program to catch and record it in the output file.

Discriminator ID line: gives the program the description of a discriminator line so that it can identify and extract data from it.

Discriminator tokens line: If a line matches the description of a discriminator line, its data will be extracted based upon the description of this discriminator tokens line. Its tokenized data are used to synchronize lines between primary and secondary input files.

Data ID line: gives the program the description of a data line so that it can identify and extract data from it.

Data tokens line: If a line matches the description of a data line, its data will be extracted based upon the description of this data tokens line. Its tokenized data are used to compare against the opponents.

The following are components used in an entry:

- **Field name**: the name you want to use for this field. It needs to be a unique name and less than 20 characters
- **Start position**: beginning position where to read the field value
- **Data type**: tell the program of what data type this field is. There are only three data types the program supports.
  - D: double precision floating number (1.7E +/- 308, up to 15 digits)
  - I: integer number (–2,147,483,648 to 2,147,483,647)
  - S: string
- **Field width**: length of the data field. When extracting a data field, the program will start reading from the <start position> and stop at the <start position> + field width
- **Decimal places**: this is only used for floating numbers (data type D) to indicate number of decimal places the field will hold.

For **Catch** string, **Discriminator ID** line, and **Data ID** line, the user basically tells the program how to uniquely identify a line. Either it is a discriminator line, a data line, a text line, or a line that matches the description of a catch string. The user is allowed to enter one or two descriptions that help to identify a line. If a line, at position **<start position 1>**, matches the whole string **<string literal 1>** (and also, if entered, at position **<start position 2>** matches the whole string **<string literal 2>**) then it is of that line type. Then, the line will be tokenized into data values according to the description in the appropriate **Discriminator** tokens line, or **Data** tokens line.

For **Discriminator** tokens line, and **Data** tokens line, you enter the required descriptions to tell the program how to tokenize the discriminator and data lines. You need to supply this information for the program to work.

Special Symbols used in the INI file:

- **Comments**: the user can use semicolon (**;**) or pound (**#**) character at the beginning or in the middle of a line to indicate the string following these symbols are comments.
- **Discriminator line**: the user can use the exclamation character (**!**) right in front of a data token name to indicate that this field needs to be parsed and compared as a data token.

- **Data Line**: if you want to include some fields in the output, you need to add an exclamation character in front of its name. This acts as a discriminator token which means the program needs to match these specified fields first before a comparison takes place.

Note: When you create names for data fields, they need to be unique within an entry or the program will give a wrong result.

HOW DOES THE PROGRAM WORK?

The input files are typically composed of 3 parts:

- discriminator lines,
- data lines, and
- text lines.

Only *discriminator lines* and *data lines* are considered for comparison in the program. *Discriminator lines* are used to synchronize data between the primary and secondary files. After *discriminators lines* have been synchronized, *data lines* between the primary and secondary files will then be parsed and compared. The program only prints out lines that have discrepancies.

1) Synchronizing primary and secondary discriminator lines

a) Find a discriminator line in the primary file; extract data and put them into discriminator tokens
b) Find a discriminator line in the secondary file; extract data and put them into discriminator tokens
c) Compare discriminator tokens between primary and secondary files. If they match then go to the second step. Otherwise, the program searches forward, within "unmatching lines limit", in the secondary file for a match. If the program finds a match in the secondary file, it will print all the extra lines of the secondary file before the match occurs. If the program does not find a match, it will print out the current discriminator line and find the next discriminator line in the primary file. The program then goes back to the beginning of the first step and starts it again.

2) Parsing and comparing data tokens

a) Find a data line from the primary file
b) Find a data line from the secondary file
c) Check to see if the secondary data line is of the same type as the primary. If it is the same, then go to the next step. Otherwise, the program searches forward, within "unmatching lines limit", in the secondary file for a line type match. If the program does not find a match, then it will print out the current primary data line and move to the next data line and start at the beginning of the second step again.
d) Data values from the primary and secondary data lines are extracted and put into data tokens. Data tokens then are pairwise compared. Discrepancies, if exist, are recorded in the output file.
e) In both primary and secondary files, move to the next line. If either one of them is of discriminator line, then go back to step one. If they are of data lines, then go to the beginning of step two again.

Repeat the first step and second step appropriately until either one of them reaches the end of the file. Then the program prints out the comparison result at the end of the output file.

WHAT DOES THE PROGRAM OUTPUT?

After synchronizing discriminator lines between the primary and secondary files, if the program does not find any discrepancies in data tokens, no output will be recorded. Otherwise, the discriminator line will be placed first before the comparing data block. Discrepancies will be shown in this format: name of the data token, level of differences (MD0 – MD9), values of the data token in primary and secondary files. If discrepancy occurs in data tokens that lie on the same line as the discriminator line, both discriminator lines of the primary and secondary files will be shown and the discrepancy is recorded under these two discriminator lines.

The header will print out the primary and secondary file names that are used in comparison program along with their file types.

The program will output any extra discriminator and data lines from two input files.

The first block of summary output:
For each data token, number of discrepancies found will be shown in columns that represents the magnitude of differences (MD0, MD1… MD9). For integer and string typed tokens, discrepancies will be calculated and shown only in the first column (MD0).

The second block of summary output:
All data fields will be shown in the output. For each data token, there are four output columns:

- **Total Errors**: total number of differences found for this data token
- **Absolute Sum**: sum of all absolute values of differences
- **Arithmetic Sum**: sum of all values of differences
- **Average**: calculation result of dividing arithmetic sum by total errors

Degree of differences:

MD0: [1, ∞)	err >= 1
MD1: [0.1, 1)	0.1 <= err < 1
MD2: [0.01, 0.1)	0.001 <= err < 0.01
MD3: [0.001, 0.01)	0.0001 <= err < 0.001
MD4: [0.0001, 0.001)	0.00001 <= err < 0.0001
MD5: [0.00001, 0.0001)	0.000001
MD6: [0.000001, 0.00001)	0.0000001
MD7: [0.0000001, 0.000001)	0.00000001
MD8: [0.00000001, 0.0000001)	
MD9: [-∞, 0.00000001)	

10^(-x) <= MDx < 10^(-x+1)