Title: CRISP#R

Note:

The whole idea is entirely simple, but it is amazingly extendable.
CRISPR is a genome-editing technology, so I choose the name CRISP#R for my idea because it is a tool that can copy and modify a huge amount of members from an existing template simultaneously in a z/OS environment.

Directions:

CRISP#R is a Rexx program. It can be easily specified by a JSON file what it should do.
I wrote 2 external functions for this program. The first is REPLACE which gets 3 parameters, INP_STR, OLD_STR, and NEW_STR.
It replaces OLD_STR with NEW_STR in INP_STR.
Yes I know that is weird, there is no such builtin function. I tried different algorithms and this one is the most efficient.
Example:

```
/* REXX */
SAY REPLACE('HELLO DROW!','DROW','WORLD'); */ says HELLO WORLD! /*
```

The other implemented external function is JSON2STM by using HTWJSON z/OS service. It gets a JSON string and a NAME and returns a string contains instructions to generate an equivalent STEM to the JSON.
If the returned string is interpreted then a STEM with the specified NAME will be available. it just adds a '#' before every JSON object names for security reasons.
Because I need the STEM simply traversable, I put straight rules for creating the STEM.
By observing follow examples you could clearly understand these rules.
Example 1:

```
JSON_FILE:
{ "COLOR" : "BLACK"}

/* REXX */
...
JSON_STR = READ(JSON_FILE);
STEM_STR = JSON2STM(JSON_STR,'MY_VAR');
/*
MY_VAR.0 = 1;
MY_VAR.1.NAME= 'COLOR';
MY_VAR.#COLOR= 'BLACK';
*/
INTERPRET STEM_STR;
SAY MY_VAR.#COLOR; */says BLACK /*
EXIT;
```

Example 2:

```
JSON_FILE:
{ "COLOR" : "BLACK"
, "AGE"   :  24
}
```

```
/* REXX */
...
JSON_STR = READ(JSON_FILE);
STEM_STR = JSON2STM(JSON_STR,'MY_VAR');
/*
MY_VAR.0 = 2;
MY_VAR.1.NAME= 'COLOR';
MY_VAR.#COLOR= 'BLACK';
MY_VAR.2.NAME= 'AGE';
MY_VAR.#AGE= 24;
*/
INTERPRET STEM_STR;
SAY MY_VAR.#COLOR; */says BLACK /*
SAY MY_VAR.#AGE  ; */says 24    /*
EXIT;
```

Example 3:

```
JSON_FILE:
[  "ITEM1"
,  "ITEM2"
,  "ITEM3"
]

/* REXX */
...
JSON_STR = READ(JSON_FILE);
STEM_STR = JSON2STM(JSON_STR,'MY_VAR');
SAY STEM_STR;
/*
MY_VAR.0 = 3;
MY_VAR.1= 'ITEM1';
MY_VAR.2= 'ITEM2';
MY_VAR.3= 'ITEM3';
*/
INTERPRET STEM_STR;
...
EXIT;
```

Example 4:

```
JSON_FILE:
[  [  "ITEM11"
   ]
,  [  "ITEM21"
   ,  "ITEM22"
   ]
]
/* REXX */
...
JSON_STR = READ(JSON_FILE);
```

```
STEM_STR = JSON2STM(JSON_STR,'MY_VAR');
SAY STEM_STR;
/*
MY_VAR.0 = 2;
MY_VAR.1.0 = 1;
MY_VAR.1.1= 'ITEM11';
MY_VAR.2.0 = 2;
MY_VAR.2.1= 'ITEM21';
MY_VAR.2.2= 'ITEM22';
*/
INTERPRET STEM_STR;
...
EXIT;
```

Example 5:

```
JSON_FILE:
{"TASK":
  [ "ITEM1"
  , "ITEM2"
  , "ITEM3"
  ]
}

/* REXX */
...
JSON_STR = READ(JSON_FILE);
STEM_STR = JSON2STM(JSON_STR,'MY_VAR');
SAY STEM_STR;
/*
MY_VAR.0 = 1;
MY_VAR.1.NAME= 'TASK';
MY_VAR.#TASK.0 = 3;
MY_VAR.#TASK.1= 'ITEM1';
MY_VAR.#TASK.2= 'ITEM2';
MY_VAR.#TASK.3= 'ITEM3';
*/
INTERPRET STEM_STR;
...
EXIT;
```

Example 6:

```
JSON_FILE:
[  {"TASK1": "ITEM11"
   }
,  {"TASK2": "ITEM21"
   }
,  {"TASK3": "ITEM31"
   }
]
```

```
/* REXX */
...
JSON_STR = READ(JSON_FILE);
STEM_STR = JSON2STM(JSON_STR,'MY_VAR');
SAY STEM_STR;
/*
MY_VAR.0 = 3;
MY_VAR.1.0 = 1;
MY_VAR.1.1.NAME= 'TASK1';
MY_VAR.1.#TASK1= 'ITEM11';
MY_VAR.2.0 = 1;
MY_VAR.2.1.NAME= 'TASK2';
MY_VAR.2.#TASK2= 'ITEM21';
MY_VAR.3.0 = 1;
MY_VAR.3.1.NAME= 'TASK3';
MY_VAR.3.#TASK3= 'ITEM31';
*/
INTERPRET STEM_STR;
...
EXIT;
```

After all, I designed this JSON to specify how I need CRISP#R to works.
JSON structure:

```
{
  "TASK": [...
   ]
}
```

n the TASK array, I can put as much as needed different TASK objects.

```
{ "INP_ADDRS": "..."
, "OUT_ADDRS": "..."
, "PARM":[...
  ]
, "MEMBER": [...
  ]
}
```

In each TASK object, we should specify the template in INP_ADDRS and the destination path for creating new members in OUT_ADDRS.
In the PARM array, we can specify replace requests for this job. Replace objects would be like this:

```
{ "OLD_STR": "..."
, "NEW_STR": "..."
}
```

Simply the program will change all found OLD_STR with corresponding NEW_STR in template text.
In the MEMBER array, we should specify members we need to create from the template. In addition,

we can indicate particular replace requests for each member in its own PARM array.

```
{ "NAME": "..."
, "PARM": [
    { "OLD_STR": "..."
    , "NEW_STR": "..."
    }
  , ...
  ]
}
```

Example 7:

```
{
  "TASK": [
    { "INP_ADDRS": "Z01878.SOURCE(ADD1JCL)"
    , "OUT_ADDRS": "Z01878.PROJECT.JCL()"
    , "PARM":[
         { "OLD_STR": "ADD1JCL"
         , "NEW_STR": "NEW#JCL"
         }
       , { "OLD_STR": "COMPILE"
         , "NEW_STR": "CMP0000"
         }
       ]
    , "MEMBER": [
         { "NAME": "ADD2CBL"
         , "PARM": [
              { "OLD_STR": "ADD1CBL"
              , "NEW_STR": "ADD2CBL"
              }
            , { "OLD_STR": "OUTLIM=15000"
              , "NEW_STR": "OUTLIM=15002"
              }
            ]
         }
      ,  { "NAME": "ADD3CBL"
         , "PARM": [
              { "OLD_STR": "ADD1CBL"
              , "NEW_STR": "ADD3CBL"
              }
            , { "OLD_STR": "OUTLIM=15000"
              , "NEW_STR": "OUTLIM=15003"
              }
            ]
          }
      ]
    }
  ]
}
```

What CRISP#R will operate due to this JSON file:
It reads the template from “Z01878.SOURCE(ADD1JCL)”.
Then replaces in template all “ADD1JCL” strings by “NEW#JCL” and all “COMPILE” strings by “CMP0000”.
For the member, “ADD2CBL” In assigns the template to a new string and replaces all “ADD1CBL” strings by “ADD2CBL” and all “OUTLIM=15000” strings by “OUTLIM=15002”. Then in creates this member in “Z01878.PROJECT.JCL()”.

For the member, “ADD3CBL” In assigns the template to a new string and replaces all “ADD1CBL” strings by “ADD3CBL” and all “OUTLIM=15000” strings by “OUTLIM=15003”. Then in creates this member in “Z01878.PROJECT.JCL()”.

It is obvious that this tool can be extended for many different proposes and can be used as a task pipeline.