Free Form Programming

Structures and design motivations of Free Form Programming Language.

Author

Dara O Shayda

Why???

A Very Simple Idea

Create a programming language that in one sentence it is part algebraic or programming language like and part loose spoken phrases. We called these Free Form expressions or simply Free Form.

Rather immediately it became obvious such languages cannot possess the classical grammars found in commercial programming languages. Moreover, such languages ought to have myriad of grammars or else it would nearly be impossible to deliver a working simple prototype.

We called these ensembles of grammars **The Grammarian**.

Path Variables

Paths and Options variables

Confidential, unauthorized access forbidden.

CCN Studios internal proprietary software documentation.

□□3.0

Sun 25 May 2025 03:39:23 GMT+1

Synopsis

Path variables are based upon the file paths “/../../..” and “../../..” string formats.

In Free Form Programming Language Path variable is of two kinds:

1. “mesh/value”: **local** to the current program, no starting “/”
2. “/student/dara/script1/mesh/uvcoors” : **external** program thus the path starts with “/”

Specification

1. **Atomic path variable**: “/group/user/script/var”
2. **Composite Path variable**: “/group/user/script/var/value”
3. **Keys**:“/group/user/script/var/value/key”

Key is the identifier or the name of something that consumes computing resources to exist. The latter something is called **Value** as in value for the key. “mesh” is a Key and “mesh/normals” is the Value, note that “mesh” consumes almost 0 computing but to compute the normals of a surface consumes substantial quantities of computing resources e.g. CPU, memory, network bandwidth and of course **recurring consumption** of resources e.g. GPU to render the mesh as a graphical object

30 frames a second on demand!

- Both local and external path variables are indeed URL based and stored in network cloud objects
- The file path address format is familiar to every grammar school student and therefore was selected as the customary way to address variables in complex and distributed structures
- The said complex structures might expand beyond the current program and into many other programs to share their local variables
- Indeed a newbie to Free Form Programming Language can immediately access and use Path variables constructed by other programs before
- Note that by default all variable values are automatically Serialized and if exceed certain water marks then compressed. Compression is one of the best forms of Serialization
- The Path variables are by default available to all third party programming languages, and always in standard simplest JSON form

http: and https: paths

⚠ **The Free Form Grammarian leaves the http paths unprocessed**. They are copied and passed to operators and functions as constant strings with no further interpretations and no further processing**.

Example

```
var = "/ff/free/image1/img";
img = "var/value";
show img;
save as image2;
```

Output:

```
img=![](https://wiki.compclassnotes.com/uploads/images/gallery/2026-05/embedded-image-aonvdpon.png)
```

```
img ⇶ {"var/value"→☁, "(var)"→☁, "(var)"↪var, "var/compressed"→☁, "var/type"→☁, "var/expression"→☁}
```

☁: icon for the persistent cloud object where the value of the key is stored

⇶ : denotes multi-arrow indicting a composite map

↪ : return arrow back to the variable in the program

Options variable

Alternatively the / can be replaced by ‘s for the existing Path variables:

```
var = "/ff/free/image1/img";
img = var's value;
show img;
save as image2;
```

In Free Form Programming Language ‘s is called option as in var’s value as an option of var. Options ‘s made passing a long list of initialized variables to a function much more compact and easier to remember and manage.

⚠ **Options variables with double quote values require more testing.**

Options variables construct path variables from arbitrary option identifiers not in existence! For example, “dara” variable has never had a path variable “hair”:

```
dara's hair is "kind of wavy";
style = "dara/hair";
show style;
save as test3;
```

Output:

```
Style = "kind of wavy"
```

The internal Free Form Grammarian has a function called VarMaker[ ] to construct standard listed typed variables or the arbitrary on-demand never seen before variables and types.

□□ This example clearly indicates the Free Form Spoken nature of the Free Form Programming Language. As the experiments have indicated the youths learn programming from Free Form language much easier and speedier and with motivation!

□□In order to match the English language grammar, for the plural names ending with s the option variables will have the trailing s’:

```
apples' count is 19;
count = "apples/count";
show count;
save as test3;
```

Output:

```
count = 19
```

□□If a path variable or option variable is not available, as in completely missing, then the original identifier, in path form, is returned as value. This avoids dealing with obscure error messages and other unpleasantries of missing values. Therefore

GetSortedVars[ ] always returns a valid value no matter, and the missing value is checked as follows:

```
GetSortedVars \[ name, cred \] == name
```

If GetSortedVars[ ] returns the same value then that variable is missing or missing its value.

Example

The Free Form rocks are constructed by the function rock[ ] which returns a string which is the name of the lhs variable:

```
mesh = rock\["ellipse", 200, 5\*{1, 1, 6}\];
save as rock1;
```

```
rock\[ \] ⇶ {"(mesh)"→☁, "(mesh)"↪mesh }
```

☁ : icon for the persistent cloud object where the value of the key is stored

⇶ : denotes multi-arrow indicting a composite map

↪ : return arrow back to the variable in the program

A third party Free Form program by another Free Former can easily access this rock mesh:

```
var = "/ff/free/rock1/mesh/value";
uv = "var/uvcoors";
show uv;
save as pathvar;
```

Output:

```
uv = {{0.789561, -2.69016}, {-1.67072, 3.75263}, {1.37182, 4.41598}, {0.199764, 4.03556}, {-1.63712, -0.0143999}, {0.230512, -3.25848}, {1.07342, 3.46127}, {1.72891, 1.27304}, {3.60557, 0.308929}, {2.13059, -3.67745}, {2.37085, 2.78642}, {-0.132767, -4.5409}, {-2.19815, -1.07722}, {-0.272674, 4.59587}, {2.96563, -2.22853}, {-0.422547, -4.48889}, {-0.926991, 2.31835}, {0.376589, -0.486291}, {-2.86105, 1.67475}, {1.01256, -3.51811}, {2.76276, 2.33716}, {-4.19068, -0.835185}, {-3.64371, -3.00476}, {0.569355, 2.71332}, {3.30174, 3.03551}, {-2.22201, -0.964461}, {4.5202, -1.39236}, {2.20664, -3.40651}, {4.37338, 1.05228}, {1.03858, -2.69736}, {4.64808, 0.375639}, {3.26256, 3.2405}, {-2.69063, 1.88625}, {-4.5811, 0.34247}, {-3.96512, -0.983707}, {-4.62285, 1.23987}, {-1.38243, -2.3752}, {-0.513229, -4.06979}, {0.602132, 2.14761}, {-2.39837, -3.98407}, {3.70975, -2.5965}, {-2.47709, 3.37107}, {-1.75433, 4.43549}, {1.52735, 1.43025}, {3.46578, -1.03043}, {-0.618612, -0.794119}, {4.68354, -1.12572}, {2.68421, 3.84622}, {-1.4737, 0.439442}, {-4.8006, 0.81873}, {-3.83208, -2.11617}, {3.53052, 1.5748}, {-1.82833, -3.77776}, {3.47874, 0.256054}, {2.02464, 0.0828023}, {-2.66834, -2.55233}}
```

```
var ⇶ {"var/coors"→☁, "var/uvcoors"→☁, "var/vertices"→☁, "var/lines"→☁, "var/triangles"→☁, "var/normals"→☁, "(var)"→☁,"(var)"↪var, "var/compressed"→☁, "var/type"→☁, "var/expression"→☁}
```

⚠ The Free Form Grammarian often compresses the cloud variables under certain criteria of size and contents.

=

= operator in Free Form Programming Language

Confidential, unauthorized access forbidden.

CCN Studios internal proprietary software documentation.

□□3.0

Sat 24 May 2025 14:11:36 GMT+1

Synopsis

- Generally speaking the operation of the operator = in most programming languages is to copy the value/address from the rhs expression to the lhs variable. The said value is singleton and multiple copied values require multiple usage of = operator.
- In commercial programming languages the operator = has merely the lifespan of a program execution session or scope of a function. Upon termination of the program or the return of the function no residues of the = operations remain.

= operator in Free Form language operates on its operands according to the following:

1. **To copy from rhs to lhs is optional for =** . Such a specific rhs copy to lhs is adequate for algebraic or math like procedures e.g. num = cos[x]; num is understood to be a single value and cos[ ] single-value function.
2. = copies certain interim computations in its expression into one or multiple separate **Persistent Cloud Objects** as opposed to simply copying the rhs to lhs, **since both the lhs and rhs are Free to be spread over multiple cloud systems**!
3. For large geometric objects their values are first Serialized and then copied over the network since the local programming space and its memory quota could be limited and often it is more efficient to spread the computing amongst as many cloud objects as possible.

4. Once the program exits and the session ceases, the operator = can still operate on its operands by a number of other processes and processors which will be detailed later.
5. Automated compression performed on variables whose content exceeds a certain watermark on byte size.
6. For many variables the compression has been used to Serialize the content. As it turned out the standard compression algorithms are perfect as Serializers for difficult binary variable values.

Evaluation

The main evaluator for the composite expressions of the form `a = expr;` where the rhs is any customary combinations of operators and functions and variables and constants, is called CCNffEqualExpr[ ] which includes number of processors in its body (+130 grammars/evaluators):

1. Each processor is a separate cloud object which deposits its scripts and data into the Evaluator
2. **The level-0 String Rewriting** occurs where the Free Form language functions and operators are replaced/rewritten by the actual internal functions available to the evaluator
3. At the top of the body it parses and evaluates the double quote strings `"ellipse"` and double-double quotes `"\"ellipse\""` and `"http…"` strings. Those evaluated are added to the program memory subsystem thus accessible by GetSortedVars[ ] cloud function and the rest of the body of the evaluator is bypassed and finally exit. Those strings and double-double quote strings continue with the evaluator for further processing
4. The evaluator also processes string structures which contain grammatical Fee Form expressions e.g. `c = circle of radius 2.01 at center = {x0,y0,z0};`

The said processors, one after the other, in a smart and applicable sequence of grammar parsers evaluate this string to dismiss or process to transform to a new expression or string acceptable to the Free Form Grammarian. These are in part considered as string or term rewriting operations.

5. At strategic locations the evaluator parses the latest version of the parsed expression and isolates symbols.
6. While parsing certain functions and operators are placed in Hold or Inactive states
7. **The level-1 String Rewriting**: finally once all the grammars and parsers and processors are exhausted a repeated string replacement loop runs and until there will be no more string replacements possible. At such an ending the Hold states are released and the held expressions, now re-written, are evaluated one last time.
8. One last time the completed parsed expressions are scanned for symbols
9. Any individual variables with values are added to the Free Form memory subsystems
10. Applicable Path variables are constructed and made available to the GetSortedVars[ ] cloud function

Oddity [Experimental]

A mathematician has no problem writing:

```
5 = sin\[x+pi/6\];
```

Yet in most commercial programming languages this statement returns an error stating that the numeral 5 cannot be reset in value. Indeed in mathematics the = operator is considered commutative in the sense a=b ⟺ b=a .

The Free Form programming language allows without any warnings or anomalies the number 5 set equal to any expression e.g. a trig expression in this case! Therefore, afterwards anywhere in the code the string “(5)” will be replaced by the string “sin[x+pi/6]” ! This was conceived to allow for de novo indexing functions and it is hoped that this feature will be released soon.

Example

Simplest = operator usage:

```
y = 5;
```

transforms to the following variable-to-value map in Serialized string form:

```
"(y)" → "(5)"
```

The Free Form Programming Language’s code symbolic engine exclusively relies on String and Term Rewriting systems:

- All expressions are required to serialize and matched for Rewriting against the variables in the execution history of the program
- In order not to replace the “y” in “boy” by “5” all expressions are parenthesized at post = action on its operands. And in order to replace the “5” in “25” again once operated upon by = the 5 is parenthesized as “(5)” .

Theorem: **The Free Form Programming Language is Turing compatible**.

Proof: Since this language can execute any String Rewriting system, it can also execute any Universal String Rewriting systems which simulate the Universal Turing machines.□

Example

The lhs is sin and the rhs is sin[.] and in Free Form language this is a customary occurrence! Since the = operator and its many grammars do not mix up the sin as function vs. sin as a variable.

```
sin = sin\[pi/6\];
show sin;
sin = sin + sin[pi/3];
show sin;
save as test;
```

Output:

```
sin = 1/2; sin = 1/2 + Sqrt\[3\]/2;
```

The following script models the sin [ ] as function and again “sin” as a return value:

```
sin\[ \] ⇉ {"sin"→☁}
```

For example:

```
num = sin\[pi/6\];
```

“num”→ ☁ denotes the string variable name “num” maps to or points at the value of the variable num and the arrow points at ☁ a specific location in CCN’s cloud systems.

☁ : icon for the persistent cloud object

⇉ : denotes multi-arrow indicting a composite map

↪ : return arrow back to the variable in the program

```
sin\[ \] ⇉ {"num"→☁}
```

Substantial effort was put into this particular aspect of the = operator in theFree Form Programming Language since to most newbies the return value for a function called foo[.] naturally should also be called foo!

This ability is specific to the = operator and it is a part of its construction.

Example

```
rock = rock\["ellipse", 500, {4,1,1}, texture\];
```

The Free Form function rock[ ] depending on its arguments constructs a 3D convex rock.

The = in this case transfers the following computations to their target memory systems:

1. The texture value is copied to a cloud variable named “rock/texture”. This texture might simply be a name e.g. “glazed” or an image path e.g. “student/dara/myduck”

2. Constant value "Mesh_Region3D" is copied to a cloud variable called “rock/type”
3. Constant value True is copied to a cloud object called “rock/compressed”. This indicates the “rock/value” is a compressed form in ascii array of unsigned bytes.
4. The computed convex rock’s triangular mesh is Serialized into ascii format JSON object and copied to cloud variable “rock/value”
5. The string “rock” as a return value and copied to the variable with the same name and this deliberate as it will become clear soon

```
rock ⇶ {"rock/texture"→☁ ,"rock/type"→☁ ,"rock/compressed"→☁ ,"rock/value"→☁ ,"rock"↪rock,"rock"→☁ }
```

☁: icon for the persistent cloud object

"a"→☁: variable called “a”, its value is copied over the net into a cloud variable persisting in a cloud system

"a"→ a : string “a” is copied to the variable also called “a” but shown as a without “” in the program text

```
newmesh = rock\["ellipse", 500, {4,1,1}, texture\];
```

```
newmesh ⇶ {"rock/texture"→☁ , "rock/type"→☁ ,"rock/compressed"→☁ , "rock/value"→☁ , "newmesh"↪newmesh, "newmesh"→☁ }
```

The newmesh without the “” is the return value to the variable currently running the program session called newmesh and the said value is its own name but in string format. This was necessary and unavoidable else the infamous Null return value automatically was assigned by the internal development programs.

The {} denotes a map as a **functional map** and in Free Form internal development language the cloud access API for this map is called

```
GetSortedVars \[ name, cred \];
```

where the cred is the credential path necessary to authorize access e.g. “student/dara/myprogram” with the “group/user/script” format.

□□ Currently GetSortedVars [name, cred] works only for one particular program path cred. In order to access multiple program paths and their session variables multiple calls to GetSortedVars [ ] is required.

Each program has multiple persistent cloud objects each serves as a memory subsystem:

1. Externals: this allows storage and retrieval of variables and data without requiring a particular program session or you might call it session-less cloud object for values external to all programs (these are not global variables)

2. Master archive: cloud objects keep references (maps) to the allocated cloud objects which contain variables and data. Master archive is used for Admin e.g. deleting and freeing space for an existing program in case required for its recurring runs.
3. Chains: linearly chained cloud objects for duplicate nested function compositions
4. GetSortedVars_Expr [name, cred] a sibling of the functional GetSortedVars[ ] which maps the expressional variables e.g. nested functions or linked list-like variables.
5. GetSortedVars_VarMaker [name, cred] another sibling of GetSortedVars[ ] which updates newly created variables as outputs of functions and operators

⚠ Free Form Programming language might consolidate or even add additional cloud memory subsystems per new users' needs and specifications without notice.

Example: Void functions

```
color\[c\];
```

The argument c can be RGB list e.g. the constant list {0,1,1} or named color the constant string "cyan".

It returns a fixed string "ffVoid" .

The argument c is copied to a cloud constant called "Color".

```
color ⇶ {"Color"→☁ ,"rock/type"→☁ , "rock/compressed"→☁, "rock/value"→☁ , "ffVoid"↪void}
```

→void means no value is returned to the program session.

Faces

Symbol extracts human faces from images.

Faces

"?/highlights"

"highlight**s**" in plural with an extra ending "s", configures the neural nets in □□ Symbol to search for and extract all human faces.

Example

```
□□

show "□□highlights";

save as faces;
```

Output

In these particular ensemble of neural nets only the faces partially expose are properly processed, however the back of the heads are not!

Example

Output

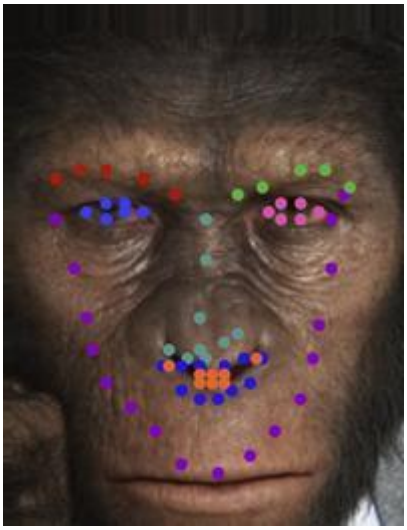
Faces

"?/highlight"

```
□□

show "□□highlight";

save as highlight;
```



Faces

Apes?!

□□ symbol in Free Form language accesses images in variety of ways.

No matter, the following orangutan image was accessed by the □□ symbol or **Face Symbol**:

```
□□:

show "□□/highlight";

save as face;
```

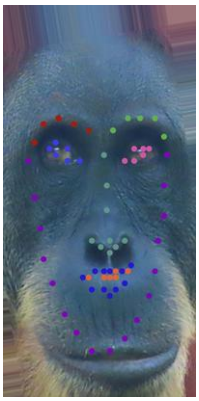
Output

"□□/highlights"

↓

"□□/highlight"

↓

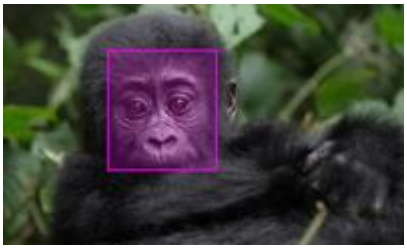


How about this gorilla!

Output

"□□/highlights"

↓



"□□/highlight"

↓

TODO

1. Make □□ Listable