拡張YAMLドキュメント
書きなぐり？

Scriptについて

	メリット	デメリット
対話型 （GUI）	・マウス操作で直感的に操作	・形状寸法の変更や条件を振って計算するための作業が面倒
バッチ処理 （Script）	・自動Scan 物性値、 形状寸法、 配向条件、 駆動条件、etc ・他のアプリとの連動	・直感的に分かりずらい ・基礎的なプログラミングの知識が必要

プログラム知識のない
ユーザーにはハードルが高い

<<< LCDMaster2D,3Dスクリプト >>>
一般ユーザーにはハードルが高い

```
var Init = new Init_File();

Init.bFDM = FALSE; //FDM:TRUE, FEM:FALSE
Init.AnalysisMode = _DC_STATIC;
//  _DYNAMIC        // Default
//  _DC_STATIC      //
//  _AC_STATIC      // AC conductance analysis
//  _BLUEPHASE      // Blue Phase
//  _FREQU_ST       // Frequensy analysis (static)

Init.CellWidth   = 48.000;
Init.CellDivNumX = 480;
Init.LayerN      = 5;
Init.Layer[0].Thickness = 0.300;          //[0]:Bottom layer
Init.Layer[0].DivNumZ   = 10;
Init.Layer[1].Thickness = 0.400;
Init.Layer[1].DivNumZ   = 10;
Init.Layer[2].Thickness = 0.090;
Init.Layer[2].DivNumZ   = 10;
Init.Layer[3].Thickness = 3.000;
Init.Layer[3].DivNumZ   = 150;
Init.Layer[4].Thickness = 0.090;
Init.Layer[4].DivNumZ   = 10;

CreateModel2D(Init.address());

////////////////////////////////////////////////////////
//
// Create structure
//
////////////////////////////////////////////////////////
var Attrib = new ATTRIB_IE();
Attrib_Init(Attrib.address());            //Initialize

Attrib.LayerNo  = 5;//LayerNo.
Attrib.Att_Type = 0;                      //0:Insulator, 1:Electrode
Attrib.IN_Const = 6.000000;
Attrib.ST_OptParams.RefMode  = _REFRACTION;
Attrib.ST_OptParams.order    = _ISO;
Attrib.ST_OptParams.DataName = "1737"

Attrib.pt_N = 4;                          //Number of vertex.

Attrib.pt[0].x = 0.000000;
Attrib.pt[0].z = 0.000000;
Attrib.pt[1].x = 0.000000;
Attrib.pt[1].z = 0.090000;
Attrib.pt[2].x = 48.000000;
Attrib.pt[2].z = 0.090000;
Attrib.pt[3].x = 48.000000;
Attrib.pt[3].z = 0.000000;

CreateStruct2D(Attrib.address());

Attrib.LayerNo  = 1;//LayerNo.
Attrib.Att_Type = 0;                      //0:Insulator, 1:Electrode
Attrib.IN_Const = 6.000000;
Attrib.ST_OptParams.RefMode  = _REFRACTION;
Attrib.ST_OptParams.order    = _ISO;
Attrib.ST_OptParams.DataName = "1737"

Attrib.pt_N = 4;                          //Number of vertex.

Attrib.pt[0].x = 0.000000;
Attrib.pt[0].z = 0.000000;
Attrib.pt[1].x = 0.000000;
Attrib.pt[1].z = 0.300000;
Attrib.pt[2].x = 48.000000;
Attrib.pt[2].z = 0.300000;
Attrib.pt[3].x = 48.000000;
Attrib.pt[3].z = 0.000000;
// Generating of Mesh
var Mesh_Set = new Mesh_Setting();
Mesh_Set.MeshType = _DELAUNAY;    //_DEl
Mesh_Set.BaseDistance = 0.400;    //Base

//Mesh_Set.MeshType = _OR_GRID;   //_DEl
//Mesh_Set.BaseDistance = 120;    //Num
//Mesh_Set.Layer_Mag[0] = 5;
                    //[0]:Bottom layer
//Mesh_Set.Layer_Mag[1] = 5;
//Mesh_Set.Layer_Mag[2] = 5;
//Mesh_Set.Layer_Mag[3] = 50;
//Mesh_Set.Layer_Mag[4] = 5;

RunMesh(Mesh_Set.address());      //Gen

// Transmittance (Bulk) - Set up calculation co
var Trns = new TrnsParam();
TransParam_Init(Trns.address());
Trns.TranRefF = _TRANS;
//  _TRANS          // Transmittance
//  _REFLE          // Reflectance
//  _REF4X4         // Reflectance(4x4)

Trns.AlgorithmF = _ALGO2X2;           //_ALG
_ALGO4X4:4x4,
Trns.LCDataF = TRUE;                  //Usin
Trns.LCDataName = "ZLI-4792";         //LC d
Trns.LightDataF = FALSE;              //Set
Trns.WaveLength = 550.000;            //Wav
Trns.PolF = _LINE;   //Polarizer [deg.]
Trns.Polarizer = -7.000;              //Pola
Trns.Analyzer  = 83.000;              //Anal

SetTrans_Bulk(Trns.address());
```

YAMLとは

- 構造化データを表現するためのデータ形式

  JSON, XMLなどと同じ構造化データフォーマットと同類。

  YAML Ain't Markup Language の略
  YAMLはJSONと類似の規格、JSONの拡張

  [特徴] **記述が簡潔**、 **読みやすい**

  Ruby on Rails をはじめとして、各種フレームワークやツールの設定ファイルやデータ保存用に使われている。

- JSONとYAML (単純な例)

```
// JSON  数値
3.14159
```
=
```
# YAML
3.14159
```

```
// JSON  文字列
"シンテック株式会社"
```
=
```
# YAML
シンテック株式会社
```

```
// JSON オブジェクト
{"name": "John Smith", "age": 33}
```
=
```
# YAML (ブロックスタイル)
name:  John Smith
age:   33
```

```
# YAML（フロースタイル）
{name:  John Smith,  age:   33}
```

```
// JSON 配列
[“milk“,  “bread",  “eggs“]
```
=
```
# YAML (ブロックスタイル)
- milk
- bread
- aggs
```

```
# YAML（フロースタイル）
[milk,  bread,  eggs]
```

2D,3Dスクリプトへのハードルを下げる。
入力データとプログラムの分離

 # 入力データ #
$translator: translator/FEM_STATIC_FFS_2D_tmp

$translatorタグで
入力データとプログラムをつなぐ

YAML形式で
入力データを記述

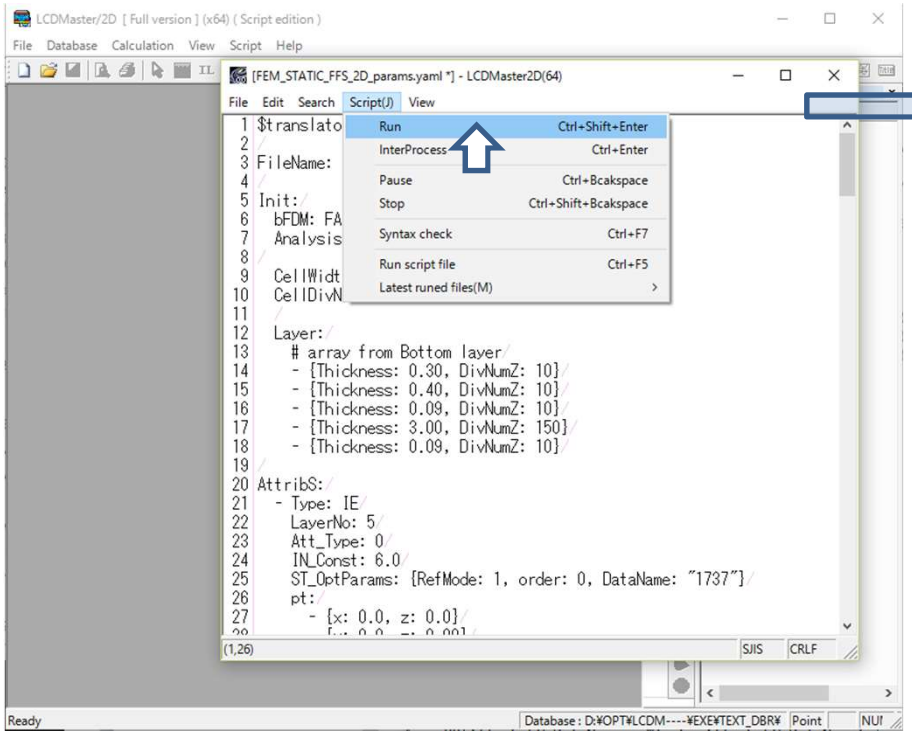
```
FileName: FEM_STATIC_FFS.2d

Init:
 bFDM:
 AnalysisMode: 1

 CellWidth: 48.000
 CellDivNumX: 480

 Layer:
  # array from Bottom layer
  - {id: 5,  Thickness: 0.30, DivNumZ: 10}
  - {id: 4,  Thickness: 0.40, DivNumZ: 10}
  - {id: 3,  Thickness: 0.09, DivNumZ: 10}
  - {id: 2,  Thickness: 3.00, DivNumZ: 150}
  - {id: 1,  Thickness: 0.09, DivNumZ: 10}

Attrib_IEs:
 - Layer: 5
   Att_Type: 0
   IN_Const: 6.0
   ST_OptParams: {RefMode: 1, order: 0, DataName: "1737"}
   pt:
    - {x: 0.0, z: 0.0}
    - {x: 0.0, z: 0.09}
    - {x: 48.0, z: 0.09}
    - {x: 48.0, z: 0.0}

 - Layer: 1
   Att_Type: 0
   IN_Const: 6.000000

   ST_OptParams: {RefMode: 1, order: 0, DataName: "1737"}

   pt:
    - {x: 0.0, z: 0.0}
    - {x: 0.0, z: 0.3}
```

 # プログラム # translator/FEM_STATIC_FFS_2D_tmp.js

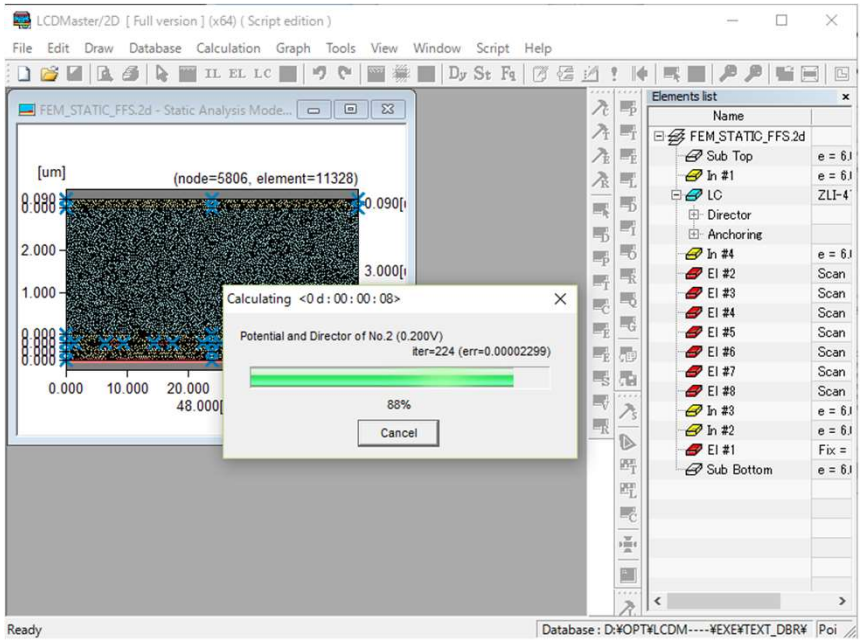
プログラムは
JavaScriptで記述

```
var yobj = this.$yobj;

////////////////////////////////////////////////////////////
//=YAML= Init:
var Init = new Init_File();

Init.bFDM = yobj.Init.bFDM;
Init.AnalysisMode = yobj.Init.AnalysisMode;

Init.CellWidth   = yobj.Init.CellWidth;
Init.CellDivNumX = yobj.Init.CellDivNumX;
Init.LayerN      = yobj.Init.Layer.length;
for (var i=0; i<Init.LayerN; i++) {
 Init.Layer[i].Thickness = yobj.Init.Layer[i].Thickness;
 Init.Layer[i].DivNumZ   = yobj.Init.Layer[i].DivNumZ;
}

CreateModel2D(Init.address());

// Create structure
var Attrib_IE = new ATTRIB_IE();
Attrib_Init(Attrib_IE.address());            //Initialize

for (var i=0; i<yobj.AttribS.length; i++) {
 var attrib = yobj.Attrib_IEs[i];

 if (hasValue(attrib.LayerNo))  Attrib_IE.LayerNo = attrib.LayerNo;
 if (hasValue(attrib.Att_Type)) Attrib_IE.Att_Type = attrib.Att_Type;

 if (hasValue(attrib.IN_Const)) Attrib_IE.IN_Const = attrib.IN_Const;

 if (hasValue(attrib.EL_Type)) Attrib_IE.EL_Type = attrib.EL_Type;
 if (hasValue(attrib.EL_Vs)) Attrib_IE.Vs = attrib.EL_Vs;

 if (hasValue(attrib.EL_Step)) {
  var startVoltage = attrib.EL_Step.start, endVoltage = attrib.EL_Step.end, Vstep
= attrib.EL_Step.vstep;
  Attrib_IE.EL_nStep = ((endVoltage-startVoltage)/Vstep+1) | 0;
  for( var j=0; j<Attrib_IE.EL_nStep; j++ ){
```

入力データの実行方法
OgOで「Open」+「Run」、「Run Script file」



Run



入力値を変更して実行したい
$contextタグに変数表、解釈は$(式), $_(式)

 # 入力データ #
$translator: translator/FEM_STATIC_FFS_2D_tmp

$context:
T: 0.09
_INSULATOR: 0
_ELECTRODE: 1

変数表

FileName: FEM_STATIC_FFS.2d

Init:
bFDM: FALSE
AnalysisMode: $(_DC_STATIC)
CellWidth: 48.000
CellDivNumX: 480

Layer:
array from Bottom layer
- {id: 5, Thickness: 0.30, DivNumZ: 10}
- {id: 4, Thickness: 0.40, DivNumZ: 10}
- {id: 3, Thickness: $(T), DivNumZ: 10}
- {id: 2, Thickness: 3.00, DivNumZ: 150}
- {id: 1, Thickness: $(T), DivNumZ: 10}

Attrib_IEs:
- Layer: 5
Att_Type: $(_INSULATOR)
IN_Const: 6.0
ST_OptParams: {RefMode: $(_REFRACTION), order: $(_ISO), DataName: "1737"}
pt:
- {x: 0.0, z: 0.0}
- {x: 0.0, z: $(T)}
- {x: 48.0, z: $(T)}
- {x: 48.0, z: 0.0}

- Layer: 1

$(式)
変数表の変数
グローバル変数
関数を使った複雑な式が記述可能

 # プログラム # translator/FEM_STATIC_FFS_2D_tmp.js

```
module.exports = function () {
var yobj = this.$yobj;
var {$, $_} = this.$;
var hasValue = this.$hasValue;

var  yobj =  $_(yobj);

var Init = new Init_File();

Init.bFDM = yobj.Init.bFDM;
Init.AnalysisMode = yobj.Init.AnalysisMode;

Init.CellWidth   = yobj.Init.CellWidth;
Init.CellDivNumX = yobj.Init.CellDivNumX;
Init.LayerN      = yobj.Init.Layer.length;
for (var i=0; i<Init.LayerN; i++) {
 Init.Layer[i].Thickness = yobj.Init.Layer[i].Thickness;
 Init.Layer[i].DivNumZ   = yobj.Init.Layer[i].DivNumZ;
}

CreateModel2D(Init.address());

// Create structure
var Attrib_IE = new ATTRIB_IE();
Attrib_Init(Attrib_IE.address());            //Initialize

for (var i=0; i<yobj.AttribS.length; i++) {
 var attrib = yobj.Attrib_IEs[i];

 if (hasValue(attrib.LayerNo))  Attrib_IE.LayerNo = attrib.LayerNo;
 if (hasValue(attrib.Att_Type)) Attrib_IE.Att_Type = attrib.Att_Type;

 if (hasValue(attrib.IN_Const)) Attrib_IE.IN_Const = attrib.IN_Const;

 if (hasValue(attrib.EL_Type)) Attrib_IE.EL_Type = attrib.EL_Type;
 if (hasValue(attrib.EL_Vs)) Attrib_IE.Vs = attrib.EL_Vs;
```

$_(YAMLオブジェクト)
YAML拡張データ内のメタデータ「$式」をすべて評価する

$(メタデータ)
引数のメタデータを評価する

変数値を複数パターンで実行
$repeatタグで繰り返しを実現

 # 入力データ #
$translator: translator/FEM_STATIC_FFS_2D_tmp

$context:
 _INSULATOR: 0
 _ELECTRODE: 1

$repeat:
 T: [0.09, 1.0]

FileName: FEM_STATIC_FFS.2d

Init:
 bFDM: FALSE
 AnalysisMode: **$(**_DC_STATIC**)**
 CellWidth: 48.000
 CellDivNumX: 480

 Layer:
  # array from Bottom layer
  - {id: 5, Thickness: 0.30, DivNumZ: 10}
  - {id: 4, Thickness: 0.40, DivNumZ: 10}
  - {id: 3, Thickness: **$(**T**)**, DivNumZ: 10}
  - {id: 2, Thickness: 3.00, DivNumZ: 150}
  - {id: 1, Thickness: **$(**T**)**, DivNumZ: 10}

Attrib_IEs:
 - Layer: 5
  Att_Type: **$(**_INSULATOR**)**
  IN_Const: 6.0
  ST_OptParams: {RefMode: **$(**_REFRACTION**)**, order: **$(**_ISO**)**, DataName: "1737"}
  pt:
   - {x: 0.0, z: 0.0}
   - {x: 0.0, z: **$(**T**)**}
   - {x: 48.0, z: **$(**T**)**}

```
for (var i=0; i<2; i++) {
   T  = values[i];
   $translatorタグ
  のプログラムを実行
}
```

 # プログラム # translator/FEM_STATIC_FFS_2D_tmp.js

```
module.exports = function () {
var yobj = this.$yobj;
var {$, $_} = this.$;
var hasValue = this.$hasValue;

var  yobj =  $_(yobj);

var Init = new Init_File();

Init.bFDM = yobj.Init.bFDM;
Init.AnalysisMode = yobj.Init.AnalysisMode;

Init.CellWidth   = yobj.Init.CellWidth;
Init.CellDivNumX = yobj.Init.CellDivNumX;
Init.LayerN      = yobj.Init.Layer.length;
for (var i=0; i<Init.LayerN; i++) {
 Init.Layer[i].Thickness = yobj.Init.Layer[i].Thickness;
 Init.Layer[i].DivNumZ   = yobj.Init.Layer[i].DivNumZ;
}

CreateModel2D(Init.address());

// Create structure
var Attrib_IE = new ATTRIB_IE();
Attrib_Init(Attrib_IE.address());            //Initialize

for (var i=0; i<yobj.AttribS.length; i++) {
 var attrib = yobj.Attrib_IEs[i];

 if (hasValue(attrib.LayerNo))  Attrib_IE.LayerNo = attrib.LayerNo;
 if (hasValue(attrib.Att_Type)) Attrib_IE.Att_Type = attrib.Att_Type;

 if (hasValue(attrib.IN_Const)) Attrib_IE.IN_Const = attrib.IN_Const;

 if (hasValue(attrib.EL_Type)) Attrib_IE.EL_Type = attrib.EL_Type;
 if (hasValue(attrib.EL_Vs)) Attrib_IE.Vs = attrib.EL_Vs;
```

繰り返しのために
特別なJavaScriptプログラムの記述はない

入力データタから計算した結果の判定
$postタグで判定する関数を記述

 # 入力データ #
$translator: translator/FEM_STATIC_FFS_2D_tmp

$context:
 _INSULATOR: 0
 _ELECTRODE: 1

$repeat:
T: [0.09, 1.0]

$post: !!js/function |
  function(**result**) {
    // result 使って判定
  }

FileName: FEM_STATIC_FFS.2d

Init:
 bFDM: FALSE
 AnalysisMode: **$(**_DC_STATIC**)**
 CellWidth: 48.000
 CellDivNumX: 480

 Layer:
  # array from Bottom layer
  - {id: 5, Thickness: 0.30, DivNumZ: 10}
  - {id: 4, Thickness: 0.40, DivNumZ: 10}
  - {id: 3, Thickness: **$(**T**)**, DivNumZ: 10}
  - {id: 2, Thickness: 3.00, DivNumZ: 150}
  - {id: 1, Thickness: **$(**T**)**, DivNumZ: 10}

Attrib_IEs:
 - Layer: 5
  Att_Type: **$(**_INSULATOR**)**
  IN_Const: 6.0

$translatorで指定したプログラムの実行結果が$postの関数の引数に渡されて実行

 # プログラム # translator/FEM_STATIC_FFS_2D_tmp.js

```
module.exports = function () {
var yobj = this.$yobj;
var {$, $_} = this;
var hasValue = this.$hasValue;

var  yobj =  $_(yobj);

var Init = new Init_File();

Init.bFDM = yobj.Init.bFDM;
Init.AnalysisMode = yobj.Init.AnalysisMode;

Init.CellWidth   = yobj.Init.CellWidth;
Init.CellDivNumX = yobj.Init.CellDivNumX;
      .
      .
      .
var Ret = new Ret_Param();
Ret.LCDataName = "ZLI-4792";            //LC database name
Ret.WaveLength = 550.000;               //Wavelength [nm]
var Data_R = DataInfo.array(DivX)();    //DataInfo Data_R[DivX];
CalcRet_Bulk(Ret.address(), SampleNo, Data_R);            //Retardation (Bulk)
//  Data_R[0].pt.x    // X[um]
//  Data_R[0].value // Retardation[nm]

Result.Data_R = Data_R;

return Result;

}; // end of module.exports
```

OgO拡張YAML
メタタグ、メタデータ

- メタタグ（Meta-Tag）
  $translator: 変換モジュールまたは変換関数。このタグがあれば 拡張YAML、なければ通常のYAMLと解釈する。
  $context: 変数表
  $repeat: 繰り返し設定
  $pre: 変換前処理関数
  $post: 変換後処理関数
  $prelude: すべての処理前に実行する関数
  $finale: すべての処理後に実行する関数

- メタデータ（Meta-Data）
  $(JavaScript計算式**)**
  ${JavaScript計算式**}**
  YAMLのタグ値、配列要素値

- YAML本体 YAMLデータのメタタグ以外

YAML の拡張化

- ハッシュ(連想配列、マップ)の拡張YAML

```
Init:
  bFDM: FALSE
  AnalysisMode: 1
  CellWidth: 48.000
  CellDivNumX: 480

  Layer:
 - {id: 5,  Thickness: 0.09, DivNumZ: 10}
 - {id: 4,  Thickness: 3.00, DivNumZ: 150}
 - {id: 3,  Thickness: 0.09, DivNumZ: 10}
 - {id: 2,  Thickness: 0.40, DivNumZ: 10}
 - {id: 1,  Thickness: 0.30, DivNumZ: 10}
```

YAML拡張化

```
$translator:  CREATE_STANDARD_2D
$context:
  T: 1.0
##############################
Init:
  bFDM: FALSE
  AnalysisMode: $(_DC_STATIC)
  CellWidth: 48.000
  CellDivNumX: 480

  Layer:
 - {id: 5,  Thickness: $(T), DivNumZ: 10}
 - {id: 4,  Thickness: 3.00, DivNumZ: 150}
 - {id: 3,  Thickness: $(T), DivNumZ: 10}
 - {id: 2,  Thickness: 0.40, DivNumZ: 10}
 - {id: 1,  Thickness: 0.30, DivNumZ: 10}
```

実行

```
Init:
  bFDM: FALSE
  AnalysisMode: 1
  CellWidth: 48.000
  CellDivNumX: 480

  Layer:
 - {id: 5,  Thickness: 1, DivNumZ: 10}
 - {id: 4,  Thickness: 3.00, DivNumZ: 150}
 - {id: 3,  Thickness: 1, DivNumZ: 10}
 - {id: 2,  Thickness: 0.40, DivNumZ: 10}
 - {id: 1,  Thickness: 0.30, DivNumZ: 10}
```

- リスト(配列)の拡張YAML

```
- pt:
   - {x: 6.5, z: 0.0}
   - {x: 9.5, z: 0.0}

- pt:
   - {x: 14.5, z: 0.0}
   - {x: 17.5, z: 0.0}

- pt:
   - {x: 22.5, z: 0.0}
   - {x: 25.5, z: 0.0}
```

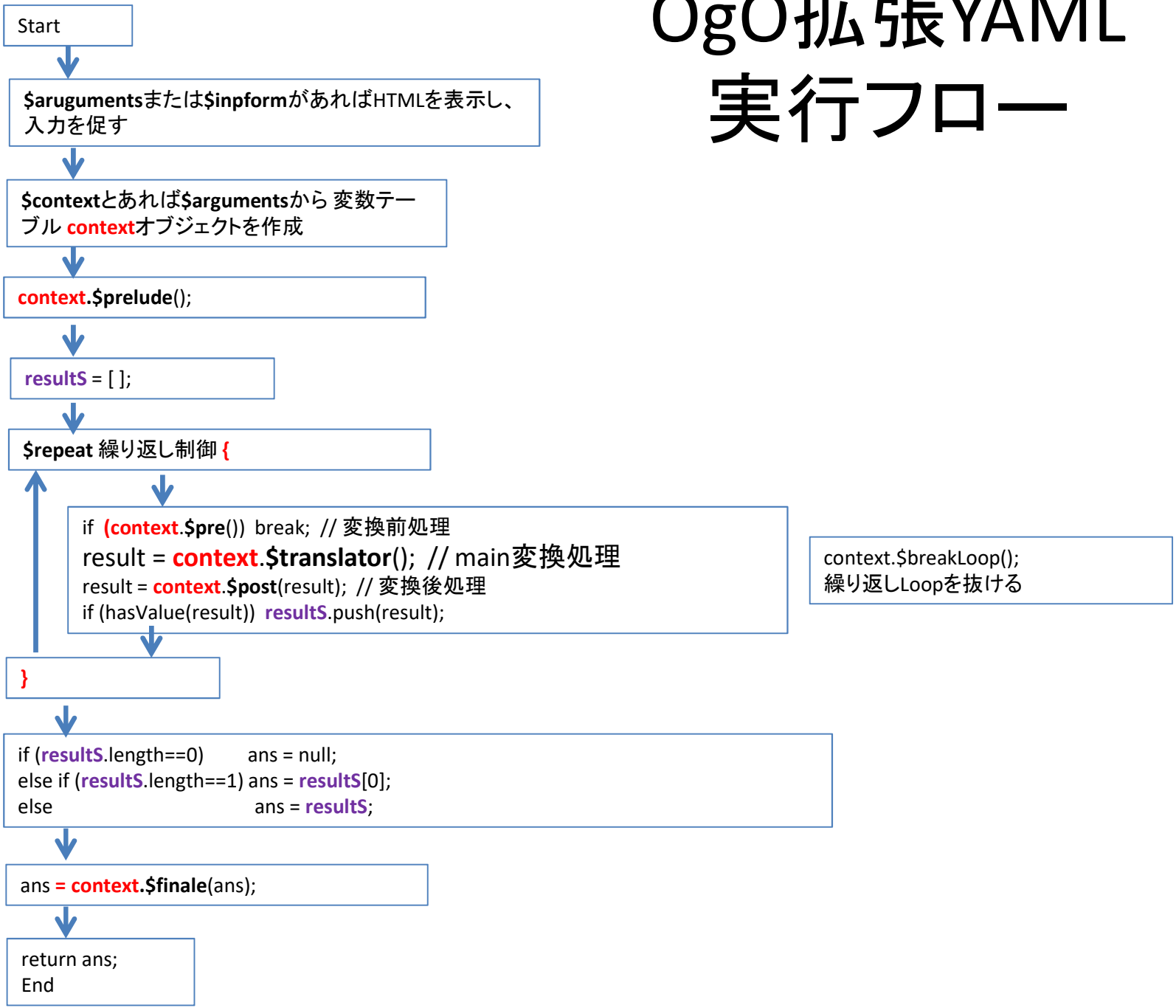
YAML拡張化

```
-
   $translator:  ~
   $context:
      width:  2.5
      pitch:   8.0
##############################
- pt:
   - {x: $(pitch-width/2), z: 0.0}
   - {x: $(pitch+width/2), z: 0.0}

- pt:
   - {x: $(2*pitch-width/2), z: 0.0}
   - {x: $(2*pitch+width/2), z: 0.0}

- pt:
   - {x: $(3*pitch-width/2), z: 0.0}
   - {x: $(3*pitch+width/2), z: 0.0}
```

実行

```
- pt:
   - x: 6.75
     z: 0
   - x: 9.25
     z: 0
- pt:
   - x: 14.75
     z: 0
   - x: 17.25
     z: 0
- pt:
   - x: 22.75
     z: 0
   - x: 25.25
     z: 0
```

OgO拡張YAML 実行フロー



$translator: メタタグ

- **$translator:** module

```
== module.js ==
module.exports = function () {
//---
var yobj = this.$yobj;    // YAMLデータ JavaScriptオブジェクト
var {$, $_} = this;              // Meta-Data解釈関数
var hasValue = this.$hasValue;
var result;

var Init = new Init_File();
Init.bFDM = $(yobj.Init.bFDM);
・・・
・・・
result = ・・・;
return result;
//---
}
```

- **$translator: !!js/function |**

```
  function() {
    var yobj = this.$yobj;   // YAMLデータ JavaScriptオブジェクト
    var {$, $_} = this;              // Meta-Data解釈関数
    var hasValue = this.$hasValue;
    var result;

    var yobj = S_(yobj);

     var Init = new Init_File();
     Init.bFDM = yobj.Init.bFDM;
     ・・・
     ・・・
     result = ・・・;
     return result;
 }
```

- **$translator:**

or

$translator: ~

は、メタタグ以外を評価したYAMLを返します。

$context: メタタグ

- **$context:**　　# 変数、関数定義

```
T :  0.09
_INSULATOR :   0
_ELECTRODE :  1
increment :  !!js/function |
     function (x) {
        return x + 1;
     }
T1:  $( increment(T) )
```

$repeat: メタタグ

- **$repeat:**

```
# for (var i=0; i<values.length; i++ ) {
#     context.x = values[i];
#     //  下記 繰り返し処理(start,end,step)・・・
# }
- x:  [0.1, 0.2, 0.5]

#  for (y=start; y<=end; y+=step) {
#      context.y = y;
#      //  下記 繰り返し処理(until)・・・
#  }
-  y: {start:  0.0,  end:  5.0,  step: 0.1}

# while( ! until() ) {
#     context.$pre();
#     context.$translator();
#     context .$post();
#  }
- !!js/function    |
       function () {  // until関数
          this.cnt++;
          if (this.cnt==10) return  true;   // 繰り返し終了
        }
```

$repeat: メタタグ その２ (values)

- **$repeat:**

```
# for (var i=0; i<values.length; i++ ) {
#     context.x = values[i];
#      pre(i, context.x);
#     //  繰り返し処理(start,end,step)・・・
#      post(i, context.x);
# }
- var:   x
  values:  [0.1, 0.2, 0.5]
  pre:  !!js/function
        function (y) {  //前処理
           //….
        }
  post:  !!js/function
        function (y) {  //後処理
           //….
        }
```

$repeat: メタタグ その3 (start, end)

- **$repeat:**

```
#  for (y=start; y<=end; y+=step) {
#     context.y = y;
#     pre(y);
#     //  下記 繰り返し処理・・・
#     post(y);
#  }
-    var:    y
     start:  0.0
     end:    5.0
     step:   0.1
     pre:  !!js/function
          function (y) {  //前処理
             //….
          }
     post:  !!js/function
          function (y) {  //後処理
             //….
          }
```

$repeat: メタタグ その４ (until)

- **$repeat:**

```
#  while ( ! until() ) {
#     pre();
#     //  下記 繰り返し処理・・・
#     post();
#  }
-    until:   !!js/function   |
      function () {  // until関数
         this.cnt++;
         if (this.cnt==10) return  true;   // 繰り返し終了
       }
    pre:  !!js/function
       function () {  //前処理
           //….
       }
    post:  !!js/function
       function () {  //後処理
          //….
       }
```

$pre: , $post: メタタグ

- **$pre:** !!js/function | # 変換前処理関数

```
function() {
    var  T = this.T;      // thisは$contextから作られたcontrextオブジェクト
    var  increment = this.increment;
    this.cnt = 0;         // contrextオブジェクトに変数(プロパティ)を登録
    //・・・
    // return true; //trueを返したら、変換処理、繰り返しが終了する
}
```

- **$post:** !!js/function | # 変換後処理関数

```
function(result) {   // resultは変換関数の実行結果
   var  T = this.T;     // thisは$contextから作られたcontextオブジェクト
    var  increment = this.increment;
    this.cnt = 0;         // contrextオブジェクトに変数(プロパティ)を登録
    //・・・ resultを加工
    return  result;
}
```

$prelude: , $finale: メタタグ

- **$prelude:** !!js/function| #すべての処理前に実行する関数

```
function() {
    var  T = this.T;      // thisは$contextから作られたcontrextオブジェクト
    var  increment = this.increment;
    this.cnt = 0;         // contrextオブジェクトに変数(プロパティ)を登録
    //この関数内だけ 拡張YAML $translator, $contextタグ以外の内容は変更できます。
    //従って、$repeatタグの内容を変更すれば、繰り返しの構成が変更できます(但し推奨しない処理) 。
    //・・・
    // return true; //trueを返したら、繰り返し、変換処理すべて       実行されません。
}
```

- **$finale:** !!js/function | #すべての処理後に実行する関数

```
function(ans) {  // resultは変換関数の実行結果
   var  T = this.T;     // thisは$contextから作られたcontextオブジェクト
    var  increment = this.increment;
    //・・・ ansを加工
    return  ans;
}
```

- **$finale:** $null_func　#拡張YAMLの実行結果が null値となる。

メタデータ
$JavaScript計算式

- tag: $ T + Math.atan(x,y) #ハッシュ(連想配列、マップ)要素データ
  JavaScript計算式では、contextオブジェクトのプロパティは変数として使用可能

  $context:タグでも、このメタデータを解釈します。
  しかし、YAML本体については解釈を$translatorの実装に任せています。(解釈は基本的にすべき)

- - $ T + Math.atan(x,y) #リスト(配列)要素データ

- 複数行あるメタデータは、「|」を使って記述します。

  ```
  tag :  |
    $if (T>0)
          T+ Math.atan(x,y);
       else
          T- Math.atan(x,y);
  ```

- tag: $(JavaScript計算式)
  - $(JavaScript計算式)

  がサンプルでよく現われるが

  tag: $JavaScript計算式
  - $JavaScript計算式

  と同じです。

$arguments: メタタグ

- **$arguments:** 拡張YAML実行結果のフォーマットを指定するためのメタタグです。

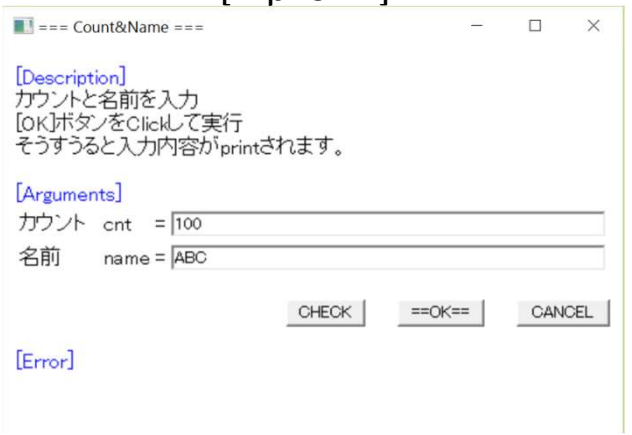
- **$arguments:**　　# 引数変数定義
  - {**var**: str}
  - {**var**: T, **value**: 0.09, comment: 厚み }
  - {**var**: V, **value**: 5, comment: 電圧}
  - {**var**: theta, **value**: 0.3}
  - {**var**: others, from: …}　# 残りの入力値をcollectして 変数othersの値とする

  var:　引数変数名　必須プロパティ
  …変数名は、下記のload,include,translate関数で引数argasからstr,T,V,theta以外の残り引数値を格納する変数となる。
  ただし、InpFormの入力項目には表れない。
  value: 引数値　メタデータ $JavaScript計算式が使用可能
  comment:　引数説明

- DSL_YAML.**load**(yaml[, args]), DSL_YAML.**include**(yaml[, args]), DSL_YAML.**translate**(yobj[, args])
  あるいは
  ypipe(yfile[, args]),　ypipe._(yfile[, args])
  における引数**args**はメタデータ**$arguments**に基づいて解釈される。
  例
  $arguments:
  - {var: x, value: 1.5}
  - {var: y, value: 2.5}
  - {var: z, value: $(x+y)}

  であるとき
  上記**load**関数呼び出しで
  args = { x: 10, y: 20}　または args = [10, 20]
  を引数とすれば、
  拡張YAML内、メタデータ $(x+y+z)は
  x=10, y=20, z=30
  で計算される。

  [InpForm]



上記**load**関数呼び出しで引数argsが渡されない場合、InpFormダイアログが起動して引数変数に値の入力を促す。Argsにundefined値以外の値を渡すと、InpFormダイアログは起動しない。

$checkarg : メタタグ

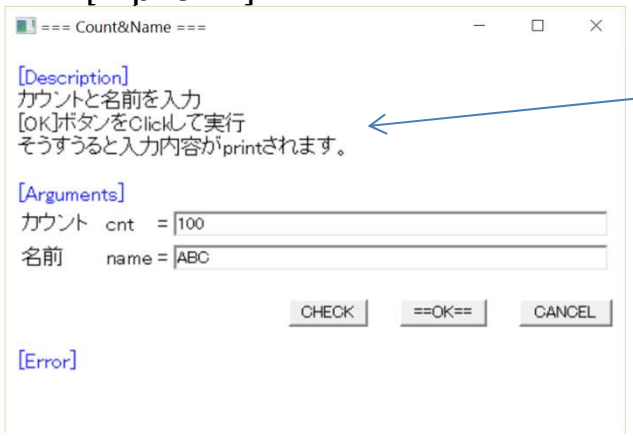
- $argumentsメタタグで定義された引数値をチェックする関数をセット。
- **$checkarg:** !!js/function |

```
function(vname, value) {
  //vname 引数変数名
  //value　引数変数値
  //  値が適正なら、return;
  //  値を変更したいなら、return 変更値
  //値が不適なら、throw new Error(“不適理由などメッセージ”);
}
```

$description: メタタグ

- **$description**メタタグには、拡張YAMLの説明を記述します。
  $argumentsで引数定義がされ、DSL_YAML.load(yaml[, args])のargsなしで実行されると、InpFormが起動しHTMLの<div id="$description> 部分にこの説明文が表示されます。

[InpForm]



拡張YAMLファイル

```
$translator: ~

$description: |
  カウントと名前を入力
  [OK]ボタンをClickして実行
  そうすうると入力内容が printされます。
```

$results: メタタグ

- **$results**メタタグは拡張YAML実行結果のフォーマットを指定します。

- パターン1
  $results:
  - {var: A}
  - {var: B}
  - {var: X, from: …} # … 残り結果を集める。 from: $(式) もok、式内で拡張YAML実行結果や$contextのプロパティが扱える _ansには拡張YAML内部実行結果がセットしてある

  拡張YAMLの内部実行結果が
  （1）{A: 100, B: 200, C: 300}のとき、外部結果は{A: 100, B:200, X:{C:300}}となる。
  （2）[100, 200, 300]のとき、外部結果は{A: 100, B:200, X:[300]}となる

  { var: 変数名, from: origin or $expr, comment: コメント, format: yaml or json などの設定された形式で表示} varタグ以外は省略可能

- パターン2
  $results: {var: X}

  拡張YAMLの内部実行結果が
  ・{A: 100, B: 200, C: 300}のとき、外部結果は{X: {A: 100, B:200, C:300}}となる。
  ・[100, 200, 300]のとき、外部結果は{X: [100, 200, 300]}となる

  { …: [origin or $expr, ・・・], comment: コメント, format: yaml or jsonなどの設定された形式で表示} …タグ以外は省略可能

- パターン3
  **$results**: {…: [A,B,…]} or {…: [2,3,…]} # []内の…は残り結果を集める。また$(式) もok、式内で拡張YAML実行結果や$contextのプロパティが扱える _ansには拡張YAML内部実行結果がセットしてある

  配列結果を作成する。
  拡張YAMLの内部実行結果のプロパティA,B あるいは 要素インデックス 2、3の値を配列の要素とし、…は残りの要素を並る。

  $results: {…: [A,B]}である場合
  ・内部実行結果が{A: 100, B: 200, C: 300}のとき、外部結果は[100, 200] となる。

  $results: {…: [2,3]}である場合
  ・内部実行結果が[10, 20, 30,40]のとき、外部結果は{30, 40]となる

  $results: {…: [2,3, …]}である場合
  ・内部実行結果が[10, 20, 30,40]のとき、外部結果は{30, 40, 10, 20]となる

$inpform: メタタグ

- **$inpform**メタタグには、InpFormダイアログの設定やモードを指定します。

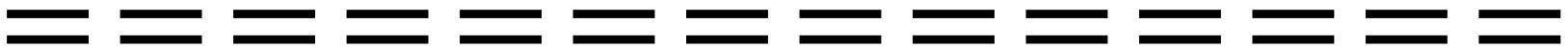
$inpform: HTMLファイル名
あるいは詳しい設定は
$inpform: {
html: htmlファイル名(相対パスならYamlファイルのディレクトリから絶対パスへ修正、省略したらデフォルトHTML(libs/yinpform.htm)が使われる),
title: タイトル名,
width: 幅, height: 高さ,
inputsize: 入力editorの幅(文字数で設定),
modeless: trueならInpFormダイアログをモードレスで動かす,
results: trueなら実行結果表示領域を表示する
}

- InpFormのHTML記述約束

(1) 説明文$descriptionメタタグの内容を表示
(2) <table><tbody id="$arguments"></tbody></table>
$argumentsメタタグの入力項目をリスト表示
各入力変数につき
<input type="text" id="$arg-変数名" size=要素幅 value=初期値>
が、HTMLロード時に展開される
(3) 入力チェックの結果を表示

(4) <input type="button" id="$ok" value="==OK==" onclick="ogo_sendEvent("$ok")>
[==OK==]or[RUN]ボタン クリックで入力チェックし有効な入力ならInpFormを閉じ 拡張YAMLを実行する
(5) <input type="button" id="$cancel" value="CANCEL" onclick="ogo_sendEvent("$cancel")>
[CANCEL]ボタン クリックでInpFormを閉じ 拡張YAMLを中止する.(モードレスの場合非表示)
(6) <input type="button" id="$check" value="CHECK" onclick="ogo_sendEvent("$checjk")>
[CHECK]ボタン クリックで入力変数の値をチェックし、不適なら にそのエラー内容を表示する

(7) <table><tbody id="$results"></tbody></table>
拡張YAMLの実行結果値を表示する表領域です。
各出力変数につき (実行結果が配列であるとき、出力変数は"..."と規定しています。
<td id="$out-出力変数>出力変数の値(YAML形式)</td>
が、実行後にHTML要素として追加される。



tag: $include(yamlファイル[, パラメータオブジェクト])
or
- $include(yamlファイル[, パラメータオブジェクト])

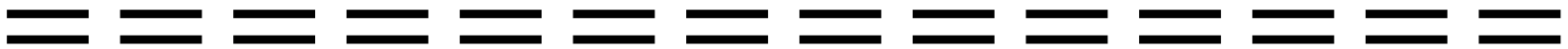
- tag値、配列値に（拡張）YAMLの評価結果をセットします。

```
# == SLID.yaml ===
- $translator: ~
  $context:
      y: 0

- {x: 0,   y: $(y)}
- {x: 100,  y: $(y)}
- {x: 100,  y: $(y+50)}
- {x: 0,  y: $(y+50)}
```

```
# == parts.yaml ===
$translator: ~
$context:
     A:  10
     B:  20

partA:  '$include("SLID.yaml", {y:A})'
partB:  '$include("SLID.yaml", {y:B})'
```

メニューコマンド[Script]-[Run]
or
var DSL_YAML = require('DSL/yaml');
var ans = DSL_YAML("parts.yaml");

```
# == 実行結果をYAML形式で表示 ===
partA:
  - {x: 0, y: 10}
  - {x: 100, y: 10}
  - {x: 100, y: 60}
  - {x: 0, y: 60}
partB:
  - {x: 0, y: 20}
  - {x: 100, y: 20}
  - {x: 100, y: 70}
  - {x: 0, y: 70}
```



context(JS)オブジェクト(変数、関数表)

- **$context:タグ から作成されたオブジェクト**

- **context**オブジェクト＝拡張YAMLの拡張部分＋utility関数群

[1]$yobj --- 拡張タグを除いたYAML生データ
[2]$translator,$context,$repeat,$pre,$post,$prelude,$finale --- 拡張タグの内容

[3]$ --- JS関数、$(YAMLの要素[, deep=0]) で値をget, deep<0ならオブジェクト・配列の要素まで再帰的に$(要素, deep-1)が実行される。$_(YAMLの要素)=S(YAMLの要素,-1)
[4]$hasValue --- JS関数、$hasValue(v), $hasValue(obj, prop)でv, obj.propが値を持つかどうかを判定
[5]$breakLoop --- breakLoop(level)JS関数、$translator,$context,$repeat,$pre,$post関数内で変換処理を中止
[5]$contiLoop --- contiLoop()JS関数、$translator,$context,$repeat,$pre,$post関数内で,現繰返の次ステップに処理を移す

[6]$toYAML --- JS関数、$toYAML(obj) objをYAML形式文字列に変換
[7]$include --- JS関数、$include(filename[,iniContext]) (拡張)YAMLファイルを読んで実行、その結果が返値となる。
[8]$translate --- JS関数、$translate(yobj[,iniContext]) 拡張タグを持つyobjを実行、その結果が返値となる。
[9]$parse --- JS関数、$parse (ytext) YAML文字列を解釈しyobjオブジェクトを返す。

YAMLデータを記述時の注意

- **tag: 値** --- 「:」と「値」には必ずスペースを置く
- **- 値** --- 「-」と「値」には必ずスペースを置く
- インデントを揃えること。揃えないとエラーまたは意図しない値がセットされる。
- メタデータ **$....** 内に「**:**」や「**,**」がある場合、引用符「 **"** 」または「**'**」で囲んだほうがよい.
- TAB文字は使わないこと。思わぬ所でエラーが起こります.

JSスクリプトから拡張YAMLの実行方法

- 引数なし実行

  var ans = **load**("Yamlファイル名.yaml");

- 引数渡し実行

  var DSL_YAML = require(“**DSL/yaml**”);
  var ans = DSL_YAML.**load**(“Yamlファイル名.yaml”,
  {x: 10, y:20, z:50, h:1000);
  ==========
  $translator: TRANSLATOR_MODULE

  $context: #引数なし実行 #引数渡し実行
  x: 100 | ⇒ 100 | ⇒ 10
  y: 200 | ⇒ 200 | ⇒ 20
  z: x + y | ⇒ 300 | ⇒ 50
  追加⇒ h: 1000

「DSL/yaml」モジュール

- 「DSL/yaml」モジュールのロード
  var DSL_YAML = require("**DSL/yaml**");

- **.parse**(ytext) 文字列ytextをYAML形式として評価し、評価結果JSオブジェクトを返す。
- **.load**(YAMLファイル名[, args]):Any YAMLファイルを読み込んで、それが拡張YAMLならargsからメタタグ$argumentsに基づいてcontextオブジェクトを初期化し、実行する。
  .include (YAMLファイル名[, args]):Any も同じ。
- **.load**(csvファイル名[, callback]), **.include**(csvファイル名[, callback]): Array csvファイルの各行をカンマ(,)で分けられた配列要素とした配列を返します。Callback関数が渡された場合、各行毎にその行のデータを配列化したesを引数として 呼ばれたcallback(es)の値を新たな配列要素に変換します。
- **.translate**(yobj[, args]):Any JSオブジェクトyobjがメタタグを持てば、拡張YAMLとして実行。
  メタタグを持たなければ, yobjを返す。
- **.checkSyntax**(YAMLファイル名) YAMLファイルを拡張YAMLとして文法的に正しいかcheckする。文法が間違っていれば、エラー例外がthrowされる。正しければ何も返さない。
- **.toYAML**(obj) objをYAML形式で文字列化する。
- **.hasValue**(value[, prop･･･]) valueがundefinedでなければtureを返し、undefinedであればfalseを返します。prop･･･があれば、同様に value.prop･･･がundefinedであるかどうかを判定する。
- **.ypipe**(yaml[, args]) : YPipe 実行可能な拡張YAMLあるいはオブジェクトをつないげられるYPipeオブジェクトを作成する。YPipeオブジェクトはメソッドを数珠つなぎし、最後尾に.yendで締めくくると実行可能プログラムなYprogオブジェクトが作成されます。
- **.yeval**(y [, args]), **.yexec**(y[, args]) y=yaml_file, yaml_obj or yprog 渡すと実行した結果を返します。
  yaml_objとは $translatorプロパティを持つオブジェクトのこと

YPipeオブジェクトのメソッド、プロパティ

- **ypipe**(y[, args]):YPipe ypipeオブジェクトにy=yaml_file, yaml_objをつなげる。

  ypipe._(y[, args])も同じ。

- **ypipe**(yprog):YPipe yprogは YProgオブジェクトコード(後で説明)です。ypipeにyprogをつなげる。
- **ypipe**(translator):YPipe ypipe.translate(translator)と同じ。
- **ypipe.pname**(name):YPipe ypipeまで実行の結果valueから {name: value}オブジェクトを作成して次のypipeへ渡す。
- **ypipe.rename**(chgmap):YPipe ypipeまで実行の結果をオブジェクトchgmapを連想配列として、プロパティ名を変えるypipeをつなぐ。
- **ypipe.translate**(translator):YPipe 関数function translator(pobj) {.....}(thisはypipe環境オブジェクト, pobjはypipeまでの実行結果)を実行しそれがが返した値を次のypipeに渡す。
  （ **ypipe**(translator)としても同じ動作となる）

  translatorをメタタグ$resultsで指定する出力フォーマットをsetすれば、ここまでのypipe計算結果を指定した出力フォーマットに沿って変換をする。
- **ypipe.proc**(processor):YPipe 関数function processor(pobj) {.....} (thisはypipe環境オブジェクト, pobjはypipeまでの実行結果)を実行し、この関数が値を返さなければ、pobjを次のpipeに渡します。processor() が値を返せば、それを次のypipeに渡します。
- **ypipe.null**: Ypipe ypipeまで実行を無視し、null値を次のpipeに渡します。
- **ypipe.yend**: YProg ypipeでつながれた一連のypipeのつながりをyevalで実行できるオブジェクトコードにまとめる。

YPipeオブジェクトのメソッドその2

- **ypipe.if**(condA)･A･･**.elseif**(condB)･B･･**.else**･C･･ **.endif** :YPipe
  条件引数function condA(pobj) {…}(thisはthisはypipe環境オブジェクト)がtrueを返すなら･A･･部分だけを実行する。condA関数がfalseを返し、function condB(pobj) {…}(this･･･)がtrueを返したら･B･･部分だけを実行する。condB関数がfalseを返したら･C･･部分だけを実行する。pobjはypipeまでの実行結果。
- **ypipe.for**(repeat_cond)･S･･**.end**:YPipe 繰り返し引数repeat_condで･S･･を繰り返す。
  repeat_condは[1] (varname, [a,b,c])または ([a,b,c])は配列要素分繰り返し、varnameがあればypipe環境オブジェクト内でその名前のプロパティ値となる、[2] (varname, [start:s, end:e, step: stp])または([start:s, end:e, step: stp])はインクリメントに値が変わり、varnameがあればypipe環境オブジェクト内でその名前のプロパティ値となる、[3] (function cond(pobj) {･･･}) {.....}(thisはypipe環境オブジェクト, pobjはypipeまでの実行結果)はfalseを返せば繰り返し終了。
- **ypipe.until**(cond)･･･**.end**:YPipe :YPipe 繰り返し判定関数function cond(pobj) {･･･} (thisはypipe環境オブジェクト, pobjはypipeまでの実行結果)がtrueを返せば繰り返しを終了する、それまで･S･･をpipeが流れる。
- **ypipe.block**(name)･S･･**.end**:YPipe 一連のpipeをブロック化しまとめ名前付けする。
- **ypipe.break**(), (cond), (label) or (label, cond):Ypipe
  for()…endfor, until()…enduntil, block()…endblock 内に置き(繰り返し)ブロックを抜け出ます。function cond(pobj){…} } (thisはypipe環境オブジェクト, pobjはypipeまでの実行結果)がfalseを返せば抜け出ない。labelがあれば、その名前付けされたブロックを抜け出ます
- **ypipe.continue**(), (cond), (label) or (label, cond):Ypipe
  for()…endfor, until()…enduntil, block()…endblock 内に置き(繰り返し)ブロックの先頭にジャンプします。function cond(){…} (thisはypipe環境オブジェクト, pobjはypipeまでの実行結果)がfalseを返せばジャンプせず次のpipeに移動します。labelがあれば、その名前付けされたブロックの先頭にジャンプします。

YProgオブジェクト

- つないだYPipeオブジェクトを最後に**.yend**を置くことでDSL_YAML.yeval関数で実行できるYprogオブジェクトがまとめられます。

- YProgオブジェクトの作成方法

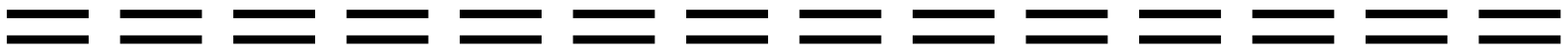
```
var  DSL_YAML = require("DSL/yaml");
var yprog = DSL_YAML.
      ypipe(yamlfile).
        If(cond).
           _(Ayml).
        else.
           _(Byml).
         endif.
      yend;
```

- YProgオブジェクトの実行方法

```
var ans = DSL_YAML.yeval(yprog, {x:20, y:30});
```

- YProgオブジェクトの再利用

```
var more_yprog = DSL_YAML.ypipe(Ayml).
   for({start:1, end:3}).
        _(Xyml)(yprog).
        _(Yyml)(yprog).
    yend.
```



C/C++からScript関数を実行

- json = apOgO->**JS_call**(funcname, json_or_yaml, asOneArg=false);

  引数:

funcname ： String	**グローバル**関数名 (func)または objname.mfunc
json_or_yaml : String	jsonまたはyamalデータ (data)
asOneArg:Bool(=false)	falseでdataが配列なら各要素funcの引数値とする。 その他の場合、dataを1引数としてfuncを実行する。

  返値:

  func(data)の実行結果を**json**フォーマットに変換して値を返す

- yaml = apOgO->**JS_ycall**(funcname, json_or_yaml , asOneArg=false);

  引数:

funcname :String	同上
json_or_yaml :String	同上
asOneArg:Bool(=false)	同上

  返値:

  func(data)の実行結果を**yaml**フォーマットに変換して値を返す

```
//== JavaScript =
function  add(x,y) {
  return  x + y;
}
var  Creator = {
  pt2: function(x,y) {
     return {x:x, y:y};
  }
};
```

```
//== C/C++ =
var  ans = apOgO->JS_call("add", [ 10, 20]);
print(ans);  //30   ans:String

var  ans = apOgO->JS_call("Creator.pt2", [ 10, 20]);
Print(ans);  // {"x": 10, "y":20}  ans:String

var  ans = apOgO->JS_ycall("Creator.pt2", [ 10, 20]);
Print(ans);  // x:   10
             // y:  20
```

C/C++からScriptコードを実行

- json = apOgO->**JS_eval**(scriptcode, filename=null, lineno=1);

  引数:

  scriptcode：String　　Scriptコード

  filename : String(=null)　ファイル名。エラー時にScriptコード名を表示する。Nullの場合”<exe::eval>”が引数値となる。

  lineno:Integer(=1)　　開始行番号。

  返値:

  Scriptコード最後に実行された式の値を**json**フォーマットで返す

- yaml = apOgO->**JS_yeval**(scriptcode, filename=null, lineno=1);

  引数:

  scriptcode :String　　同上

  filename :String　　同上

  lineno:Integer　　同上

  返値:

  func(data)の実行結果を**yaml**フォーマットに変換して値を返す

C/C++からScriptファイルを実行

- apOgO->**run**(scriptname);

  引数:

  scriptname：String　　Scriptファイル名

  返値:

  起動できたらtrueを返す。できなければfalseを返す。