Roboticsware

Automation シリーズ

スクリプトマニュアル

RWMN-5-21-960

本マニュアルの目的 3
1章　スクリプト VER1 (SC1 構文) 4
概要 4
構文 4
演算子 5
2章　スクリプト VER2 (SC2 構文) 8
概要 8
2－1　変数と値 9
変数について 9
値のタイプについて 10
時刻型の値の書式 12
値の型変換について 14
配列変数 15
多次元配列変数 16
キーワード「c」で配列を作成する 17
配列のインデックスについて 18
不定値（#N/A#）と空値（#NOTHING#）について 21
2－2　オペレータ 22
スクリプトで使用できるオペレータ 22
オペレータの特徴と使用例 23
数値のビット演算 25
2－3　ステートメント 26
条件文（if - else） 26
条件文（switch-case） 28
繰り返し文（while） 29
繰り返し文（for） 31
2－4　ユーザー定義関数 32
ユーザー定義関数について 32
ユーザー定義関数の構文 32
2－5　コメント 34
スクリプト中にコメントを書く 34
3章　演算式構文 35
概要 35
構文 36
変数とは 37
変数の禁則文字について 37
使用できる値 37
使用できるオペレータ 38
使用できる関数 39

本マニュアルの目的

弊社パッケージでは様々な場面でスクリプトが利用されます。本マニュアルではスクリプトの構文について説明します。

スクリプト構文は３つの種類があります。

・スクリプト Ver1 (SC1 構文)
・スクリプト Ver2 (SC2 構文)
・演算式構文

本マニュアルではそれぞれのスクリプトの構文を章毎に説明します。

１章 スクリプト Ver1(SC1 構文)	最もシンプルなスクリプトです。略して SC1 構文と呼びます。 下記の２箇所で使用されます。 ・スクリプト Ver1 アクション ・アクション詳細設定タブのスクリプト実行
２章 スクリプト Ver2(SC2 構文)	本格的なスクリプトです。略して SC2 構文と呼びます。 下記の２箇所で使用されます。 ・スクリプト Ver2 アクション ・Panel Browser
３章 演算式構文	演算式を表現するための構文です。バインド書式、アラームマスタの発生条件、メールマスタの送信条件など様々な場面で利用されます。

1章　スクリプト Ver1 (SC1 構文)

概要

スクリプト Ver1（以後 SC1 構文と呼ぶ）は下記の２箇所で使用されます。

・スクリプト Ver1 アクション
・アクション詳細設定タブのスクリプト実行

SC1 構文は+ - * / < > などの VisualBasic 互換の演算子を利用し、タグ同士の演算を行う簡易的なスクリプトです。手軽に使用できることを目的としており、if 文（条件分岐）や for 文（繰り返し処理）などは使用できません。より高度な処理を行いたい場合はスクリプト Ver2(SC2 構文)をご使用下さい。



Ver4 ではタグスクリプトと呼ばれていましたが、Ver5 からスクリプトの構文体系を整理するため、スクリプト Ver1（略して SC1 構文）と名称が変わりました。



実際のアクションの設定方法については「アクションマニュアル」「スクリプト Ver1 アクション」を参照して下さい。

構文

SC1 構文は下記の構文で記述します。

```
タグパス１ = 演算式１
タグパス２ = 演算式２
タグパス３ = 演算式３
...
```

スクリプトが実行されると、演算式の結果がタグへ書き込まれます。演算式の部分にタグパスを含めることができます。複数の式を記述したい場合は、改行して記述していきます。計算はすべて数値で行われ、TRUE は 1 で FALSE は 0 として扱われます。文字は数値に変換されます。

例）

```
U01.F01.T01 = 1
U01.F01.T01 = U01.F01.T02 * (U01.F01.T03 + 1)
U01.F01.T01 = (U01.F01.T02 And U01.F01.T03) Or (U01.F01.T04 And U01.F01.T05)
```



SC1 構文では文字を扱うことができません。文字型のタグの場合、文字が数値に変換され処理されます。例えば文字列の合成など、文字として値を扱いたい場合はスクリプト Ver2(SC2 構文)を使用して下さい。

演算子

使用できる演算子は下記になります。

算術演算子			
	指数演算	^	数値のべき乗を求めます。
	マイナス符号	-	式の符号を反転した値を指定するために使用されます。
	乗算と除算	* /	2 つの数値の積・商を計算します。
	整数除算	¥	2 つの数値の商を計算し、結果を整数で返します。
	剰余演算	Mod	2 つの数値の除算を行い、その剰余を返します。
	加算と減算	+ -	2 つの数値の和、差を求めます。
比較演算子			
	等しい	=	2 つの数値の比較し、等しければ TRUE を返します。
	等しくない	<>	2 つの数値の比較し、等しくなければ TRUE を返します。
	より小さい	<	2 つの数値の比較し、左値が右値より小さければ TRUE を返します。
	より大きい	>	2 つの数値の比較し、左値が右値より大きければ TRUE を返します。
	以下	<=	2 つの数値の比較し、左値が右値以下であれば TRUE を返します。
	以上	>=	2 つの数値の比較し、左値が右値以上であれば TRUE を返します。
論理演算子			
	論理否定	Not	式の論理否定を求めます。
	論理積	And	2 つの式の論理積を求めます。
	論理和	Or	2 つの式の論理和を求めます。
	排他的論理和	Xor	2 つの式の排他的論理和を求めます。
	論理等価演算	Eqv	2 つの式の論理等価演算を行います。
	論理包含演算	Imp	2 つの式の論理包含演算を行います。
論理演算子			
	論理積	AndN	2 つの式の論理積を求めます（数値としてビット単位の比較をします)。
	論理和	OrN	2 つの式の論理和を求めます（数値としてビット単位の比較をします)。

下記は各演算子ごとの具体的な計算の例です。

算術演算子			
	指数演算	^	2 ^ 3 結果 3
	マイナス符号	-	3 – 2 結果 1
	乗算と除算	* /	2 * 1.5 結果 3　　5 / 2 結果 2.5
	整数除算	¥	15 ¥ 2 結果 7
	剰余演算	Mod	15 Mod 2 結果 1 (15 / 2 = 7 余り 1 であるため)
	加算と減算	+ -	1 + 0.5 結果 1.5　 2 – 4 結果-2
比較演算子			
	等しい	=	12 = 12 結果 1　 34 = 12 結果 0
	等しくない	<>	12 <> 12 結果 0　 34 <> 12 結果 1
	より小さい	<	12 < 12 結果 0　 12 < 34 結果 1
	より大きい	>	12 > 12 結果 0　 34 > 12 結果 1
	以下	<=	12 <= 12 結果 1　 34 <= 12 結果 0
	以上	>=	12 >= 12 結果 1　 12 >= 34 結果 0

＜次頁につづく＞

＜前頁のつづき＞

<table>
<tr><th colspan="2">論理演算子</th><th></th><th></th></tr>
<tr><td></td><td>論理否定</td><td>Not</td><td>Not a
<table><tr><th>a</th><th>結果</th></tr><tr><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td></tr><tr><td>123</td><td>0</td></tr></table></td></tr>
<tr><td></td><td>論理積</td><td>And</td><td>a And b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>12</td><td>123</td><td>1</td></tr></table></td></tr>
<tr><td></td><td>論理和</td><td>Or</td><td>a Or b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>12</td><td>123</td><td>1</td></tr></table></td></tr>
<tr><td></td><td>排他的論理和</td><td>Xor</td><td>a Xor b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>12</td><td>123</td><td>0</td></tr></table></td></tr>
<tr><td></td><td>論理等価演算</td><td>Eqv</td><td>a Eqv b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>12</td><td>123</td><td>1</td></tr></table></td></tr>
<tr><td></td><td>論理包含演算</td><td>Imp</td><td>a Imp b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>12</td><td>123</td><td>1</td></tr></table></td></tr>
</table>

＜次頁につづく＞

＜前頁のつづき＞

<table>
<tr><th colspan="2">拡張論理演算子</th><th></th><th></th></tr>
<tr><td></td><td>論理積</td><td>AndN</td><td>a And b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>12</td><td>123</td><td>8</td></tr></table></td></tr>
<tr><td></td><td>論理和</td><td>OrN</td><td>a And b
<table><tr><th>a</th><th>b</th><th>結果</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>12</td><td>123</td><td>127</td></tr></table></td></tr>
</table>



1 つの式の中で複数の演算子が使われているとき、一定の順序で計算されます。まず、算術演算子、比較演算子、論理演算子の順序で評価されます。算術演算子および論理演算子は、上表に示す順序で評価されます。比較演算子の各演算子の優先順位はすべて同じです。括弧「()」 を使用すると、強制的に優先順序を入れ替えることができます。括弧の内側の演算は外側の演算よりも常に優先的に実行されます。



仮想デバイスは PLC をシミュレートしたデバイス（ビットデバイス、ワードデバイス）なので、整数しか扱えません。小数を扱うには、メモリデバイスの数値タグを利用します。



ネットワークタグをスクリプト中で用いる場合、下記のように先頭に「//Network/」を付加して下さい。

```
//Network/N01/T01
```

2章　スクリプト Ver2 (SC2 構文)

概要

スクリプト Ver2（以後 SC2 構文と呼ぶ）は高度な処理を目的とした本格的なスクリプト構文です。本マニュアルでは SC2 構文の基本構文について説明します。

SC2 構文は下記の２箇所で使用されます。

・スクリプト Ver2 アクション
・FA-Panel (Panel Browser)

SC2 構文の仕様は上記２箇所で共通ですが、実際に使用できる関数やオブジェクトなどはそれぞれ異なります。
まずは、下記のマニュアルを参照の上、本マニュアルにお進み下さい。

スクリプト Ver2 アクション	「アクションマニュアル」「スクリプト Ver2 アクション」章
FA-Panel (Panel Browser)	「FA-Panel ユーザーズマニュアル」「スクリプト」章

２－１　変数と値

変数について

スクリプトでは、変数を定義して使用することができます。変数とは、数値、文字、ブール値（TRUE/FALSE）などを保管するための箱のようなものです。

変数は「var」で宣言します。宣言の右側に「 = 値」を指定すると、その値で初期化されます。下記は宣言方法の例です。

```
var a;
var a, b;
var a = 123;
var a = "Hello";
```

変数には下記のように値を何度でも代入できます。

```
a = 1;            // a に 1 が代入される
a = "hello";      // a に"hello"が代入される
a = 1 + 2 + 3;    // a に 6 が代入される
a = a + 1;        // a に 1 を加えた値が代入される
```

下記は変数を実際に使用した例です。

例１）

```
var a = 1;                          // a に数値 1 を設定
var b = 2;                          // b に数値 2 を設定
var x = a * 2 + b;                  // x に演算結果である数値 4 が設定される
```

例２）

```
var a = "Hello ";                   // a に文字"Hello "を設定
var b = "World";                    // b に文字"World "を設定
var x = a + b;                      // x に演算結果である" Hello World "が設定される
```



変数を宣言する場合、スクリプトの予約語は使用することができません。特に**「c」「T」「F」**は 1 文字ですが予約語となっており、下記のように「c」を変数として使用することはできません。

var c;

値のタイプについて

SC2 構文で扱える値のタイプは下記の５種類です。

値のタイプ	説明
ブール	ブール値（T or F）です。“T”は On の状態を表し、“F”は Off の状態を表します。
数値	数値です。小数点も扱えます。
文字	文字列です。
時刻	時刻（日時）です。
オブジェクト	オブジェクトを変数に格納することができます。

ブール型

T もしくは F と記述します。T は On の状態をあらわし、F は Off の状態を表します。TRUE もしくは　FALSE と書くこともできます（※全て大文字で記述してください）。

例）

```
var a = T;
var b = F;
var d = a && b;                    // d は F になります。
var e = a || b;                    // e は T になります。
var f = T && F || F;               // f は F になります。
```

数値型

数値です。通常の整数はもちろん倍精度浮動小数も扱えます。

例）

```
var a = 123.45;
var b = 23.45;
var d = a + b;                     // d は 146.9 になります。
var e = 0xFF23;                    // 0x を頭に付けると数値を 16 進数で表現できます
```



16 進数表記で扱える値の範囲は 0x00000000 ～ 0xFFFFFFFF になります。

文字型

文字です。文字は必ず、「"」で囲みます。

例）

```
var a = "Hello";
var b = a + "World";                // b は"Hello World"になります。
```

時刻型

時刻です。時刻の表現には時刻書式を用います。時刻書式については次頁を参照下さい。

例）

```
var a = #2005/1/2 3:4:5#;           //2005年1月2日3時4分5秒
var a = #2005/1/2 3:4:5.123#;       //2005年1月2日3時4分5秒123ms
var a = #now#;                      //現在の時刻
var a = #now - 1D#;                 //現在の時刻から1日前（２４時間前）の時刻
```

オブジェクト型

オブジェクトを変数に格納することができます。

例）

```
var a = this;                       //自分自身のオブジェクトを代入する
a.FillColor = "#FF0000";            //自分自身の背景を赤にする
a = parent;                         //親オブジェクトを代入する
a.FillColor = "#00FF00";            //親の背景を緑にする
```



上記以外に変数の状態として不定値と空値があります。詳しくは「不定値（#N/A#）と空値（#NOTHING#）について」を参照して下さい。

時刻型の値の書式

本節では、時刻型の値の書式について解説します。時刻型のデータでは、直接的な時刻指定のほか、例えば今現在から１０分後などといった相対時刻を求めることができます。また、コントロールの中には時刻型のプロパティを持つものがあり、それらに対してスクリプトからアクセスする場合には、時刻型の変数によって行います。

絶対時刻による指定

絶対時刻で時刻表現する場合のフォーマットは下記のとおりです。

```
#yyyy/mm/dd H:M:S#
```

例）

```
#2005/1/2 3:4:5#
```

ms まで含めたい場合は下記の書式になります。

```
#yyyy/mm/dd H:M:S.ms#
```

例）

```
#2005/1/2 3:4:5.123#
```

相対時刻による指定

時刻型では相対時刻表現も可能です。相対時刻とは、現在の日時を基準として 1 日後の日時、などの様に指定する方法です。

例）現在が「2003/01/15 12:12:12」（水曜）の場合

```
#now#            現在。つまり「2003/01/15 12:12:12」。
#now-1d#         現在-１日。つまり「2003/01/14 12:12:12」。
#now+1d#         現在+１日。つまり「2003/01/16 12:12:12」。
#now-1mo#        現在-１ヶ月。つまり「2002/12/15 12:12:12」。  ※下記の注意を参照
#month#          今月。つまり「2003/01/01 00:00:00」。
#month-1d#       先月の末日。つまり「2002/12/31 00:00:00」。
#month-1d+9h#    先月の末日の朝９時。つまり「2002/12/31 09:00:00」。
#day-1d#         昨日。つまり「2003/01/14 00:00:00」。
#day+8h+30m#     今日の８時３０分。つまり「2003/01/15 08:30:00」。
#week+2d#        今週の火曜日。つまり「2003/01/14 00:00:00」。
```

相対時刻のフォーマットは下記のようになります。

```
基準時刻[+時間間隔] [-時間間隔]…
```

上記フォーマットの基準時刻の部分には下記の基準時刻が指定できます。

基準時刻	説明	現在が「2003/06/18 12:12:12」（水曜）とすると
now	現在。	2003/06/18 12:12:12
year	今年。	2003/01/01 00:00:00
month	今月。	2003/06/01 00:00:00
week	今週。	2003/06/15 00:00:00
day	今日。	2003/06/18 00:00:00
hour	今の時刻（時まで有効）。	2003/06/18 12:00:00
minute	今の時刻（分まで有効）。	2003/06/18 12:12:00
future	未来。	はるか未来の時刻。
past	過去。	はるか過去の時刻。

時間間隔の前には「+」か「-」及び数値を必ず指定します。「+」であれば未来へ、「-」であれば過去になります。例えば「+10m」であれば基準時刻から 10 分後を指します。

時間間隔	説明
y	年
mo	月
d	日
w	週
h	時
m	分
s	秒



時刻型の値で計算を行った場合下記のようになります。

```
var a = #2004/1/1 0:0:0#;
var b = #2004/1/1 0:1:0#;
var d = b - a;      // d は 60（秒）になる。
var e = a + 10;   // e は a に 10 秒加算されて#2004/1/1 0:0:10#になる。
```



ルートメソッド「GetRelativeTime」は、ある特定の時刻（現在の時刻ではなく、指定した時刻）を基準に相対時刻の書式を用いて、相対時刻を求めることができます。

例 1）結果は「2002/10/9 10:10:10」になります。

```
var a = ::GetRelativeTime(::CTime("2002/10/10 10:10:10"), "now-1d");
```

例 2）結果は「2002/10/9 0:0:0」になります。

```
var a = ::GetRelativeTime(::CTime("2002/10/10 10:10:10"), "day-1d");
```



現在が 31 日の場合、NOW に対して月の引き算を含めた相対時刻（例　#NOW-1MO#等）を使用できません。理由は、31 日の 1 ヶ月前が 30 日までしかない月の場合、該当する日が存在しないためエラーが発生しますので注意してください。

値の型変換について

ルートメソッドには下記の変換メソッドが用意されており、スクリプトなどから呼び出す事で型変換を行うことができます。それぞれのメソッドの詳細は、「コントロールリファレンス」マニュアルのルートメソッドを参照下さい。

CBool	値をブール型に変換する。
CNum	値を数値型に変換する。
CStr	値を文字列に変換する。
CTime	値を時刻型に変換する。
CNumStr (NS)	フォーマットに従った数値文字列を作成する。
CTimeStr (TS)	値を時刻文字列に変換する。
CNumToHex	値を 16 進数の文字列に変換する。
CNumToOct	値を 8 進数の文字列に変換する。
CNumToBin	値を 2 進数の文字列に変換する。
CHexToNum	16 進数の文字列を値に変換する。
COctToNum	8 進数の文字列を値に変換する。
CBinToNum	2 進数の文字列を値に変換する。

例えば下記の場合、変数 c には“Data1:123 Data2:456”という文字列が代入されます。

```
var a = 123;
var b = 456;
var c = "Data1:" + ::CStr(a) + " Data2:" + ::CStr(b);
```



CNumStr (別名 NS)は小数点や桁数を調整した数値文字列を生成する便利な関数です。第一引数で数値を、第二引数で小数点以下の桁数を指定します。

例）下記の例では、a には"12.00"という文字が代入されます。
var a = ::CNumStr(12, 2);　// a には文字列“12.00”が代入される

配列変数

スクリプトは配列変数を扱うことができます。配列とは、値の集合のことです。配列を使うと、例えば 100 個のデータがある場合、それを 1 つの変数でまとめて扱うことができます。大量のデータをまとめて処理したいときなどに利用します。

配列変数の宣言

配列変数は下記のように宣言します。

例）100 個の配列の例

```
var a[100];                          // 100 個の配列の宣言

a[0] = 123;                          // 123 を 1 番初めの要素として代入する
a[1] = 234;                          // 234 を 2 番初めの要素として代入する
…
a[99] = 345;                         // 345 を 100 番初めの要素として代入する
var b = a[0] + a[1] ･･･ + a[99];     // 配列 a の全要素を合計して b に代入する
var d = ::Math.Sum(a);               // 配列 a の全要素を合計して d に代入する（Sum 関数を使用した例）
```



配列要素の添え字（インデックス）は 0 から始まります。



::Math.Sum は合計値を求めるメソッドです。Math オブジェクトには算術演算を行うための便利な関数がまとめられています。配列と組み合わせて利用する事によって、合計値、最大値、最小値、平均値などを簡単に求めることができます。詳しくは「コントロールリファレンス」マニュアルのルートメソッドを参照下さい。

値の型の混在

配列では複数の値の型を混在させることができます。配列内で統一する必要はありません。

```
a[0] = 123;
a[1] = "hello";
a[2] = T;
```

多次元配列変数

前述にて解説した配列変数「var a[100];」は、正確には 1 次元配列と呼びます。SC2 構文では多次元配列も扱えるようになっており、配列の次元数には制限はありません。

多次元配列変数の宣言

多次元配列は下記のように宣言します。

例 1）2 次元配列(10 x 100 個)の例

```
var a[10,100];
a[0, 0] = 1;              // 1 を[0, 0]に代入する
a[0, 1] = 2;              // 2 を[0, 1]に代入する
…
a[9, 99] = 999;           // 999 を[9, 99]に代入する
var b = ::Math.Ave(a);    // 2 次元目の要素の平均値を求めます（10 個の 1 次元配列で返ってくる）
```



上記「var b = ::Math.Ave(a);」は、各列の平均値を１０個の 1 次元配列として一括で返してくれます。
つまり、
b[0] ･･･ a[0]の 100 行分のデータの平均
b[1] ･･･ a[1]の 100 行分のデータの平均
: : :
といった戻り値となります。



2 次元配列はデータベースのテーブルのようなマトリックスとして考えるとイメージが浮かびやすいかもしれません。例えば「var a[10,100];」は 10 列、100 行のテーブルとして考えることができます。尚、データベースを扱うコントロールから値を取得した場合、取得結果は 2 次元配列として扱われます。

例 2）3 次元配列(10 x 100 x 100 個)の例

```
var a[10, 100, 100];
a[0, 0, 0] = 1;           // 1 を[0, 0, 0]に代入する
a[0, 0, 1] = 2;           // 2 を[0, 0, 1]に代入する
…
a[9, 99, 99] = 99999;     // 99999 を[9, 99, 99]に代入する
var b = ::Math.Ave(a);    // a の 3 次元目の要素の平均値を求めます（10x100 個の 2 次元配列で返ってくる）
var d = ::Math.Ave(b);    // b の 2 次元目の要素の平均値を求めます（10 個の 1 次元配列で返ってくる）
var e = ::Math.Ave(d);    // d の 1 次元目の要素の平均値を求めます（1 個の値として返ってくる）
```

キーワード「c」で配列を作成する

キーワード「c」を用いると、配列を簡単に作成することができます。

このキーワードの書式は下記のとおりです。

```
c([value1], [value2], …)
```

例）

```
var a = c(1,2,3,4);                                  //「var a[4]」と同意。各要素に「1,2,3,4」が代入される
var a = c(1 + 2, 3 + 4, 4 + 5);                      // 計算式を含められます
var a = c(::GetVal("TAG1"), ::GetVal("TAG2"));       // メソッドを含められます
var a = c(T, 123, "Hello", #2005/1/1 0:0:0#);        // 異なるデータタイプの値を含められます
```

また上記を組み合わせて多次元配列を作成することもできます。

例）

```
var a = c(c(1,2), c(3,4), c(4,5));     //「var a[3,2]」と同じ。a[0,0]は 1、a[0,1]は 2、a[2,1]は 5。
var a = c(c(c(1,2), c(3,4)), c(c(5,6), c(7,8)))      //「var a[2, 2, 2]」と同じ。
```



キーワード「c」は combine（結合）の略です。

配列のインデックスについて

1 次元配列のインデックス指定

配列の要素を取得するには[ ]の中に配列のインデックスを指定します。インデックスは 0 から(要素数-1)までの数値です。

a[Index]

インデックスの指定方法には下記の方法があります。

1) インデックスに数値 i を指定すると i 番目の要素を取り出します。
2) インデックスに c(i, j, k, l, m....)を指定すると i, j, k, l, m...番目の要素を取り出します。i,j,k,l,m...は小さい順に指定する必要はありません。また、重複した値を指定することもできます。例）c(3,4,4,4,2,1,0)
3) インデックスに要素数と同じブール値の配列を指定すると、T となっている要素のみ抜き出します。
4) インデックスに i : j を指定すると、i から j 番目の連続した要素を取り出します。
5) インデックスになにも指定しないと、全要素を取り出します。

上記の各指定方法について、「var a = c(1,2,3,4,5,6,7,8,9,10)」を元に考えると、

1) a[5] は先頭から 6 つ目の要素を返します。つまり 6 を返します。
2) a[c(2, 5, 3)] は先頭から 3 つ目、6 つ目、4 つ目の各要素を返します。つまり c(3,6,4)を返します。
3) a[c(T, T, F, F, F, T, F, F, F, T)] は T となっている要素のみを抜き出します。つまり c(1,2,6,10)を返します。
4) a[2 : 5] は先頭から 3 つ目から 6 つ目までの連続した要素を返します。つまり c(3,4,5,6) を返します。
5) a[]は配列の全要素を返します。つまり c(1,2,3,4,5,6,7,8,9,10)をそのまま返します。

1 次元配列のインデックス指定の応用例

上記の応用により、配列から指定した条件を満足する要素のみを抜き出す事ができます。

例）

```
var a = c(1,2,3,4,5,6,7,8,9,10)
var b = a[a > 6];
```

上記を実行すると、b には 6 以上の要素、つまり c(7,8,9,10)が代入されます。
a > 6 は c(F,F,F,F,F,F,T,T,T,T)を返し、結果として a[c(F,F,F,F,F,F,T,T,T,T)]は c(7,8,9,10)を返すからです。

例）

```
var a = c("タンク異常", "タンク正常", "タンク開", "ポンプ異常", "ポンプ正常", "ポンプ開")
var b = a[::IsLike(a, "%異常%")];
```

上記を実行すると、"異常"の文字が含まれた要素、つまり c("タンク異常", "ポンプ異常")が代入されます。
::IsLike(a, "%異常%")は c(T,F,F,T,F,F)を返し、結果として a[c(T,F,F,T,F,F)]は c("タンク異常", "ポンプ異常")を返すからです。

多次元配列のインデックス指定

配列の要素の値を取得するには[ ]の中に配列の各次元のインデックスを指定します。

a[Index1 , Index2, ...]

インデックスの指定方法は 1 次元配列と同じです。

「var a = c(c(1,2), c(3,4), c(4,5))」 を元に考えると

1）a[2, 0] は、1 次元目が先頭から 3 つ目、2 次元目が先頭の要素、つまり 4 を返します。
2）a[2,] は、1 次元目が先頭から 3 つ目、2 次元目は省略なので全要素、つまり c(4,5)を返します。
3）a[, 1] は、1 次元目は省略なので全要素、2 次元目が先頭から 2 つ目の要素、つまり c(2,4,5)を返します。
4）a[c(T, F, T),] は、1 次元目が T となっている要素、2 次元目は省略なので全要素、つまり c(c(1,2), c(4,5))を返します。
5）a[1 : 2,] は 1 次元目が先頭から 2 つ目から 3 つ目の連続した要素、2 次元目は省略なので全要素、つまり c(c(3,4), c(4,5))を返します。

多次元配列のインデックス指定の応用例

上記の応用により、以下のような比較による要素の抜き出しが可能です。

例）

```
var a = c(c("Japan", "England", "France" , "China", "Singapore"),
                                    c("Asia", "Europ", "Europ", "Asia", "Asia"));
var b = a[0, a[1, ] == "Asia"];
```

上記を実行すると、b には"Asia"の国だけ抜き出され、c("Japan", "China", "Singapore")が代入されます。
理由は、a[1,] == "Asia"は c(T,F,F,T,T)を返し、a[0, c(T,F,F,T,T)]は c("Japan", "China", "Singapore")を返す為です。

２次元配列をテーブルとして考えてみる

ここでは、２次元配列をデータベースのテーブルに見立てて、インデックスの応用を考えてみます。

1	Test1	T	2005/1/1 12:30:00
2	Test2	F	2005/1/1 14:25:00
3	Test3	T	2005/1/1 15:15:00

例えば、上記のような３行４列のテーブルを、キーワード「c」を用いて表現すると下記の様になります。

```
a = c(   c(1,2,3),
         c(“Test1”,”Test2”,”Test3”),
         c(T,F,T),
         c(#2005/1/1 12:30:00#, #2005/1/1 14:25:00#, #2005/1/1 15:15:00#)   );
```

この例では、下記のようなインデックスの使い方が可能です。

1. a[0,]は０列目、つまり c(1,2,3)になります。

2. ::Math.Ave(a[0,])は０列目の平均値、つまり２になります。

3. a[c(1,2),]は１列目と２列目、結果は下記になります。

```
c(c(“Test1”,”Test2”,”Test3”),
   c(T,F,T) );
```

4. a[c(1,1) ,]はいずれも１列目、結果は下記になります。

```
c(c(T,F,T),
   c(T,F,T) );
```

5. a[, a[3,] > #2005/1/1 14:00:00#]は、「2005/1/1 14:00:00」以降のレコードを取得します。

```
c(c(2,3),
   c(”Test2”,”Test3”),
   c(F,T),
   c(”#2005/1/1 14:25:00#”,”#2005/1/1 15:15:00#”) );
```

6. ::Math.Ave(a[0 , a[3,] > #2005/1/1 14:00:00#])は、「2005/1/1 14:00:00」以降のレコードの、０列目の平均値を求めます。つまり c(2,3)の平均値、結果は 2.5 になります。



データベースからテーブルデータを取得したい場合は、コントロールリファレンスマニュアルの「ダイレクトクエリーデータベースコントロール」もしくは「リモートクエリーデータベースコントロール」を参照して下さい。

不定値（#N/A#）と空値（#NOTHING#）について

不定値（値がなにも設定されていない状態）や空値（要素数 0 の状態）は下記のように表現されます。

不定値

不定値（値がなにも入っていない状態）の状態です。不定値は#N/A#で表現します。

例 1）

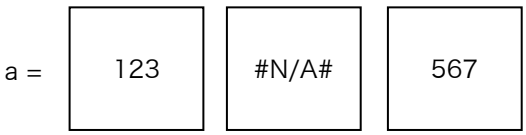
```
var a = #N/A#;                    // 要素数 1 で値が不定値。
var b = (a == #N/A#);             // T になる
```

例 2）

```
var a = c(123, #N/A#, 567);       // 要素数 3 で a[1]が不定値。
var b = (a == #N/A#);             // c(F,T,F)になる
```



不定値とは、言い換えると箱はあるが箱になにも入っていない状態です。
例）var a = c(123, #N/A#, 567)



空値

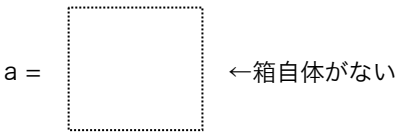
全要素数が 0 の状態です。空値は#NOTHING#で表現します。
下記のようなケースで空値の状態になります。

例）

```
var a = c(12, 23, 34);
var b = a[a < 10];          // 10 より小さい要素が 1 つもないため、要素数 0 の状態になる。
if (b == #NOTHING#)
{
        // 空値の時の処理
        ...
}
```



空値とは、言い換えると箱自体がない状態です。
例）var a = #NOTHING#



2－2　オペレータ

スクリプトで使用できるオペレータ

スクリプトで使用できるオペレータは以下のとおりです。

算術演算子			
	マイナス符号	-	式の符号を反転した値を指定するために使用されます。
	乗算と除算	* /	2 つの数値の積・商を計算します。
	加算と減算	+ -	2 つの数値の和・差を計算します。
比較演算子			
	等しい	==	2 つの数値を比較し、等しければ TRUE を返します。
	等しくない	!=	2 つの数値を比較し、等しくなければ TRUE を返します。
	より小さい	<	2 つの数値を比較し、左値が右値より小さければ TRUE を返します。
	より大きい	>	2 つの数値を比較し、左値が右値より大きければ TRUE を返します。
	以下	<=	2 つの数値を比較し、左値が右値以下であれば TRUE を返します。
	以上	>=	2 つの数値を比較し、左値が右値以上であれば TRUE を返します。
論理演算子			
	論理否定	!	式の論理否定を求めます。
	論理積	&&	2 つの式の論理積を求めます。
	論理和	||	2 つの式の論理和を求めます。
ビット演算子			
	And 演算子	&	ビットごとの And 演算子（整数としてビット単位の比較をします)。 尚、左式が文字列の場合、文字列として結合させます。
	Or 演算子	|	ビットごとの Or 演算子（整数としてビット単位の比較をします)。

オペレータの特徴と使用例

暗黙の文字列合成と数値演算

オペランドの内容がいずれも文字列の場合、加算は文字列の合成として処理されます。但し、片方のオペランドが数値でもう片方のオペランドが文字列の場合には、文字列の内容が数字の場合は数値同士の演算として加算が行われます。

```
var a = "Hello" + " World";        //変数 a には文字列"Hello World"が代入される
var b = "1" + 2;                   //変数 b には数値 3 が代入される
var d = 1 + "2";                   //同様に数値 3 になる
```

四則演算におけるブール値の解釈

ブール値を四則演算しようとすると、T は 1、F は 0 に変換されます

```
var a = T + T + T + F + T;         //数値 4 になる（1+1+1+0+1）
var b = T * 123 + F * 456;         //数値 123 になる（1*123 + 0*456）
```

時刻の加算、減算

時刻の場合、下記の値の組み合わせで加算、減算が行えます。

1） [時刻] + [数値]　または　[時刻] - [数値]の場合
数値は秒として解釈され、秒単位で時刻の値を進めたり戻したりできます。

```
var a = #2004/1/1 0:0:0#;
var b = a + 10;            // b は a に 10 秒加算されて#2004/1/1 0:0:10#になる。
var d = a + 10.123;        // b は a に 10 秒 123ms 加算されて#2004/1/1 0:0:10.123#になる。
```

2） [時刻] - [時刻] の場合
時刻の差を秒として数値で返します。

```
var a = #2004/1/1 0:0:0#;
var b = #2004/1/1 0:1:0#;
var d = b - a;              // d は 60 になる（1 分の差＝60 秒）。
```

配列に対する演算

配列の場合、各要素すべてに一括で演算を行うことができます。

■四則演算（ +, -, *, / ）オペレータ

```
a = c(1,2,3);
b = a + 1;                          // b は c(2, 3, 4)になる
d = a - 1;                          // d は c(0, 1, 2)になる
e = a * 2;                          // e は c(2, 4, 6)になる
f = a / 2;                          // f は c(0.5, 1, 1.5)になる
g = a + c(2, 3, 4);                 // g は c(3, 5, 7)になる
```

■比較演算（ <, <=, >, >=, == ）オペレータ

```
a = c(1,2,3);
b = a < 2;                          // b は c(T, F, F)になる
d = a <= 2;                         // d は c(T, T, F)になる
e = a > 2;                          // e は c(F, F, T)になる
f = a >= 2;                         // f は c(F, T, T)になる
g = a == 2;                         // f は c(F, T, F)になる
```

多次元配列に対する演算

多次元配列の場合も、要素すべてに一括で演算が行われます。

```
a = c(c(1,2,3), c(4,5,6));
b = a + 2;                          // b は c(c(3,4,5), c(6,7,8))になる
```



「c(1,2,3,4,5,6,7,8) + c(1,2,3)」のように異なる要素数の配列の演算をした場合は、右側 c(1,2,3)が繰り返し演算に使用されます。つまり「c(1,2,3,4,5,6,7,8) + c(1,2,3)」は「c(1,2,3,4,5,6,7,8) + c(1,2,3,1,2,3,1,2)」と同じ結果、つまり c(2,4,6,5,7,9,8,10)となります。

数値のビット演算

2 進数で i 番目のビットを取得したり、i 番目のビットだけを ON/OFF したりする場合には、ビット演算子（「&」及び「|」）を使用します。

例）

```
var b = (a & 0x1) > 0;              // a の数値の 1 ビット目が 1 であれば b は T になります。
var b = (a & 0x2) > 0;              // a の数値の 2 ビット目が 1 であれば b は T になります。
var b = (a & 0x4) > 0;              // a の数値の 3 ビット目が 1 であれば b は T になります。
var b = (a & 0x8) > 0;              // a の数値の 4 ビット目が 1 であれば b は T になります。

var a = a | 0x1;                    // a の数値の 1 ビット目を 1 にします。
var a = a | 0x2;                    // a の数値の 2 ビット目を 1 にします。
var a = a | 0x4;                    // a の数値の 3 ビット目を 1 にします。
var a = a | 0x8;                    // a の数値の 4 ビット目を 1 にします。

var a = a & 0xFFFE;                 // a のワード値の 1 ビット目を 0 にします。
var a = a & 0xFFFD;                 // a のワード値の 2 ビット目を 0 にします。
var a = a & 0xFFFB;                 // a のワード値の 3 ビット目を 0 にします。
var a = a & 0xFFF7;                 // a のワード値の 4 ビット目を 0 にします。
```

2－3　ステートメント

条件文（if - else）

■ if 文

if 文を使うと、条件式の判定によってスクリプトを実行するかどうかを切り替えることができます。条件式の判定はブール値で行われ、判定が TRUE の時にブランケットで囲まれたスコープ内のスクリプトが実行されます。尚、スクリプトが 1 行だけの場合はブランケットを省略することができます。

```
if ( <条件式> )
{
    <スクリプト>;
}
```

■ if-else 文

if-else 文は、条件式の判定によってスクリプト 1 又はスクリプト 2 のどちらかを実行するように処理の流れを分岐させる事ができます。判定結果が TRUE の時にスクリプト 1 が実行され、FALSE の時にスクリプト 2 が実行されます。尚、各スクリプトが 1 行だけの場合はブランケットを省略することができます。

```
if ( <条件式> )
{
    <スクリプト 1>;
}
else
{
    <スクリプト 2>;
}
```

■ if- else if - else if - else 文

if- else if - else if - … - else 文は、複数の else if 文と最後の else 文で構成されます。最後の else 文はオプションで必要がなければ記述しなくてもかまいません。上から下の順で条件式の評価が行われ、条件が TRUE になったスコープ内のスクリプトが実行されます。すべての条件を評価後、いずれの条件も TRUE でない場合には最後の else に記述したスクリプトが実行されます。尚、各スクリプトが 1 行だけの場合はブランケットを省略することができます。

```
if ( <条件式> )
{
    <スクリプト 1>;
}
else if ( <条件式> )
{
    <スクリプト 2>;
}
else
{
    <スクリプト 3>;
}
```

if 文の使用例は以下のとおりです。

例 1）条件式に式やメソッドを含めた例

```
if (a == 1 || (b > d * 123) && ::GetVal("TAG01"))
{
    ::WriteVal("TAG01", 1);
}
else
{
    ::WriteVal("TAG01", 2);
}
```

例 2）スクリプトが 1 行のときはブランケットを省略できる

```
if (a == 1) ::WriteVal("TAG01", 1);
```

例 3）各ステートメントが 1 行のときもブランケットを省略でき

```
if (a == 1)
    ::WriteVal("TAG01", 1);
else  if (a == 2)
    ::WriteVal("TAG01", 2);
else  if (a == 3)
    ::WriteVal("TAG01", 3);
}
```



条件文「if - else」の構文は C 言語にほぼ準拠しています。

条件文（switch-case）

switch 文も if 文と同様に条件式による処理の分岐が行えます。条件式の多い多分岐の処理をシンプルに記述できます。switch 文では<式>と<値>が一致したスクリプトが実行されます。条件値は「case」の後ろに記述し、行末に「:」を一文字書きます。続いてスクリプトを記述し、処理のブロックの最後には break キーワードを書きます。スクリプトは複数行記述することも可能です。また、いずれの条件も満足しなかった時の処理として、<デフォルトスクリプト>を記述することができます。<デフォルトスクリプト>は省略することもできます。

```
switch ( <式> )
{
    case <値 1>:
        <スクリプト 1>;
        break;
    case <値 1>:
        <スクリプト 2>;
        break;
        .....
    case <値 n>:
        <スクリプト n>;
        break;
    default:
        <デフォルトスクリプト>;
}
```

例）

```
switch (a)
{
    case 1:
        b = ::GetVal("TAG01");
        break;
    case 2:
        b = ::GetVal("TAG02");
        break;
    case 3:
        b = ::GetVal("TAG03");
        break;
}
```



条件文「switch - case」の構文は C 言語にほぼ準拠しています。

繰り返し文（while）

while 文は、条件式が TRUE の間スクリプトを繰り返し実行します。スクリプトが 1 行だけの時はブランケットを省略することができます。

```
while ( <条件式> )
{
    <スクリプト>;
}
```

■break キーワード

繰り返し処理の実行中に、スクリプトの途中でループを抜けたい時には break キーワードを用います。break キーワードは任意の場所で使うことができます。break は複数個書くこともできます。通常 if 文と組み合わせて記述し、if 条件が TRUE になった時に break を呼びだすようにします。

```
while ( <条件式 1> )
{
    <スクリプト 1>;
    if ( <条件式 2> ) break;
    <スクリプト 2>;
    if ( <条件式 3> ) break;
    <スクリプト 3>;
}
```

■continue キーワード

繰り返し処理の実行中に、途中でループの先頭に戻って実行したい場合には、continue キーワードを用います。continue キーワードは任意の場所で使うことができます。continue は複数個書くこともできます。通常 if と組み合わせて記述し、if 条件が TRUE になった時に continue を呼びだすようにします。continue が呼び出されると while 条件が評価され、TRUE であればブランケット内のスクリプトの先頭に戻って処理が実行されます。

```
while ( <条件式 1> )
{
    <スクリプト 1>;
    if ( <条件式 2> ) continue;
    <スクリプト 2>;
    if ( <条件式 3> ) continue;
    <スクリプト 3>;
}
```

尚、break キーワードと continue キーワードは混在させて使用する事もできます。

例 1)

```
var a = 0;
while (a < 100)
{
    ....
    a = a + 1;
}
```

ブランケット内のスクリプトは 100 回実行されます。

例 2）

```
var a = 0;
while (1)
{
    ...
    a = a + 1;
    if (a == 100) break;
}
```

前の例と同じようにブランケット内のスクリプトは 100 回実行されます。ここで、while (1)として指定しているループ条件はずっと TRUE になることが無いため、break の if 文が成立するまではループを抜けることはありません。



繰り返し文「while」の構文は C 言語にほぼ準拠しています。

繰り返し文（for）

for 文は、while 文と同じように条件式が TRUE の間スクリプトを繰り返し実行します。while 文と異なる点は<初期化式>と<再初期化式>です。<初期化式>は for 文を評価する前に 1 度だけ実行されるもので、通常はループ条件判定のための変数初期化に用います。<再初期化式>はブランケット内のスクリプトが最後まで実行された後に呼び出されるもので、ループ条件のカウンタ変数をカウントアップする際などに使用します。尚、<初期化式>と<再初期化式>にはなにも書かずに省略することもできます。また、while 文と同じように break/continue キーワードを使用することが可能であり、スクリプトが 1 行だけの時はブランケットを省略することもできます。

```
for ( <初期化式> ; <条件式> ; <再初期化式> )
{
    <スクリプト>;
}
```

for 文の処理順序は下記のとおりです。

1) <初期化式>
2) <条件式>でもし TRUE なら<スクリプト>を実行、FALSE なら終了
3) <再初期化式>
4) <条件式>が FALSE になるまで 2 と 3 を繰り返す

例）スクリプトを 10 回繰り返して実行する例

```
for (var a = 0; a < 10; a = a + 1)  // a は 0,1,2...と変化する。
{
    <スクリプト>;
}
```

例）同様に、スクリプトは 10 回実行される

```
for (var a = 0; a < 20; a = a + 2)  // a は 0,2,4...と変化する。
{
    <スクリプト>;
}
```



繰り返し文「for」の構文は C 言語にほぼ準拠しています。

２－４　ユーザー定義関数

ユーザー定義関数について

ユーザー定義関数とは、開発者が独自に作成することができる関数のことです。ユーザー定義関数はオブジェクト毎に記述することができます。オブジェクトに記述した関数は、メソッドやプロパティと同じように他のオブジェクトから絶対パス参照 又は 相対パス参照によって呼び出して実行する事ができます。

ユーザー定義関数は function キーワードの後に関数名を書き、「( )」の中に引数を「,」（カンマ）区切りで定義します。関数名や引数名は自由に名前が付けられます。そして、「return」キーワードで関数を終了させて、返値を返すことができます。以下は、ユーザー定義関数の記述例です。

```
function Total(a, b)
{
    return a + b;
}
```

この例の場合、関数名は「Total」です。a と b という 2 つの引数を渡すと、「a+b」の計算結果を返値として返します。

ユーザー定義関数の構文

ユーザー定義関数の構文は以下のとおりです。

```
function FunctionName(<パラメータ 1>, <パラメータ 2>, …<パラメータ n>)
{
    <スクリプト>
    return <返値>;
}
```

1） パラメータは必要な数だけ指定する事ができます。
2） パラメータが必要ないときは省略可能ですが、関数名の後の「()」は付ける必要があります（「function FunctionName()」などとします）。
3） return キーワードは関数を終了させて、呼び出し元に対して返値をかえす事ができます。
4） return の返値は省略することも可能です。返値を省略した場合、関数は「return」の位置で終了します。
5） return キーワードはスクリプト中、何個でも書くことができます。また、return 自体を書かないこともできます。そのような場合、関数は最後までスクリプトを実行した時点で終了します。

例）return が無い場合

```
function SampleFunction()
{
    <スクリプト>
}
```

例）return が 1 つある場合

```
function Total(a, b)
{
    return a + b;
}
```

例）return が複数ある場合

```
function ErrMessage(a)
{
    switch (a)
    {
        case 1: return "Error1";
        case 2: return "Error2";
        case 3: return "Errro3";
    }
    return "Not Found Error code";
}
```

6） パラメータの先頭に「&」を付けて記述することにより、パラメータを介して値を返すことができます。

例）引数に値を返す

```
function SampleFunction()
{
    var Param;
    GetParam(Param);  //Param には 100 が返る。
}
function GetParam(&vParam)
{
    vParam = 100;  //引数で渡された変数に値を代入。
}
```



ユーザー関数の再帰呼び出しには対応しておりません。



関数名や変数名の一番先頭には数字は使用できません。また、スクリプトで予約されている for や if、オペレータで使用している「+-*/&|=<>!(),」などの文字も使用できません。

２－５　コメント

スクリプト中にコメントを書く

スクリプト中にコメントを書きたい場合は「//」を先頭に書きその後にコメントを書きます。

```
// コメント
function XXX(a, b)
{
    // コメント
    …
    //コメント
    a = a + 1;     //コメント
    b = b + 1;     //コメント
}
```

複数行のコメントを書きたい場合、下記のようにします。

```
// コメント１
// コメント２
// コメント３
```

また、「/*」と「*/」キーワードで囲むと、その間はすべてコメントとして認識されます。

```
/*  コメント１
    コメント２
    コメント３ */
```



コメントの構文については c 言語に準拠しています。

3章　演算式構文

概要

演算式を表現するための構文です。
バインド書式、アラームマスタの発生条件、メールマスタの送信条件など様々な場面で利用されます。

●ブラウザのバインド書式の「IN: / OUT: / INOUT:」以降の書式
例）

```
INOUT : U01.F01.T01
IN: U01.F01.T01 * 100 + 50
IN:CNumStr((#D0 + #D1) / 2, 2)
IN: {$M0}{"ON"},{default}{"OFF"}
```

●アラームアクションのメール送信条件
例）

```
[CM0] == "重故障"
[CM0] > 10 && [CM1] == 1
```

構文

演算子構文には値（T/F、123、"abc"など）、オペレータ（+ - + / & | など）、変数（U01.F01.T01、[CM0]など）及び関数（CNumStr など）が使用できます。

演算子構文の仕様は下記になります（本マニュアルでは抽象化のため、変数を VAR1、VAR2、VAR3、、と表現します）。

構文 1

演算式

構文 1 は演算式のみを記述します。

例）

```
VAR1 * 100
VAR2 * VAR3 + 100
CNumStr((VAR1 + VAR2) / 2, 2)
...
```

構文 2

{条件式 1}{条件値 1},{条件式 2}{条件値 2 }, … ,{default}{デフォルト値}

構文 2 は条件式とその条件が成立したときの条件値を記述します。条件式を左から順番に評価し、条件が満たされたときの値（条件値）が結果値になります。もしどれにも当てはまらない場合は、右端のデフォルト値が結果値となります。

例）

```
{VAR1}{"ON"},{default}{"OFF"}
{VAR1>=80}{"High"},{VAR2>20}{"Normal"},{default}{"Low"}
{VAR1>=80}{"#FFFFFF"},{VAR3>20}{"#FF0000"},{default}{"#0000FF"}
...
```

変数とは

演算子構文は様々な場面で利用されますが、その際、演算式の中に変数を利用することができます。
変数として何が指定できるかについては機能毎に異なります。ブラウザのバインド書式の場合、タグパス（U01.F01.T01 など）が変数として利用できます。アラームアクションのメール送信条件の場合、フィールド名（[CM0]、[CM1]など）を変数として利用できます。変数としてなにが使用できるかについては、各機能毎のマニュアルを参照して下さい。

変数の禁則文字について

演算子構文で使用される変数に下記の文字を含めることはできません。

1．下記「使用できるオペレータ」に定義されているオペレータ記号と同じ文字は使用してはならない。但し、下記のように変数の前後を「'」（シングルクオーテーション）で囲む事によって使用する事ができます。
例）

```
'U01.F01.TAG-01'                          //「-」記号が含まれる場合
'U01!Alive'                               //「!」記号が含まれる場合
'U01.F01.#SD0000+FIL=LIN:0|1|0|100'       // アクティブタグのフィルタを使用した場合
```

2．空白文字。 例：「TAG　01」
3．「"」ダブルクオーテーション及び「'」シングルクオーテーション　例：「TAG"02"」
4．先頭の文字には数値は使用できない。　例：「123TAG」
5．先頭の文字にはアンダーバーは使用できない。
例：「_TAG01」
6．全角文字。　例：「U01.F01.ポンプ１」 ※すべての全角文字が使用できないわけではありませんが、文字コードにより使用できない場合があるため、全角文字はなるべく避けてください。

使用できる値

演算子構文で扱える値のタイプは下記の４種類です。

値のタイプ	説明
ブール	ブール値（T or F）です。“T”は On の状態を表し、“F”は Off の状態を表します。
数値	数値です。小数点も扱えます。
文字	文字列です。
時刻	時刻（日時）です。

各値の仕様ついては SC2 構文と同様です。詳しくは「スクリプト VER2 (SC2 構文)」「値のタイプについて」を参照してください。

使用できるオペレータ

下記は演算子構文で扱えるオペレータです。各オペレータの動作の仕様ついては SC2 構文と同様です。詳しくは「スクリプト VER2 (SC2 構文)」を参照してください。

算術演算子			
	マイナス符号	-	式の符号を反転した値を指定するために使用されます。
	乗算と除算	* /	2 つの数値の積・商を計算します。
	加算と減算	+ -	2 つの数値の和、差を求めます。
比較演算子			
	等しい	==	2 つの数値を比較し、等しければ TRUE を返します。
	等しくない	!=	2 つの数値を比較し、等しくなければ TRUE を返します。
	より小さい	<	2 つの数値を比較し、左値が右値より小さければ TRUE を返します。
	より大きい	>	2 つの数値を比較し、左値が右値より大きければ TRUE を返します。
	以下	<=	2 つの数値を比較し、左値が右値以下であれば TRUE を返します。
	以上	>=	2 つの数値を比較し、左値が右値以上であれば TRUE を返します。
論理演算子			
	論理否定	!	式の論理否定を求めます。
	論理積	&&	2 つの式の論理積を求めます。
	論理和	||	2 つの式の論理和を求めます。
ビット演算子			
	And 演算子	&	ビットごとの And 演算子（整数としてビット単位の比較をします)。 左式が文字列の場合、文字列として結合させます。
	Or 演算子	|	ビットごとの Or 演算子（整数としてビット単位の比較をします)。

使用できる関数

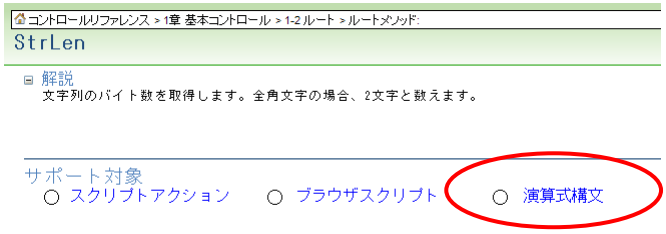
演算子構文から関数（ルートメソッド）を呼び出すことができます。詳しくは「コントロールリファレンス（オンラインマニュアル)」のルートメソッドを参照下さい。



コントロールリファレンスにてルートメソッドの一覧表の「演算式構文」に○が付いているメソッドが利用できます。



また、説明のページで「演算式構文」に○が付いているかどうかでも判断できます。



演算子構文で使用できる関数の例	
String	
StrLen	文字列の文字数を取得する。
LeftStr	文字列の左側の部分を抽出する。
RightStr	文字列の右側の部分を抽出する。
MidStr	文字列の中間の部分を抽出する。
FindStr	文字列内の文字を検索する。
IsLike	文字列内の文字を類似検索し一致するかどうかを判別する。
CombineStr	文字列を結合する（特に配列の場合に使用される。通常は「&」や「+」で文字を結合できます)。
TrimLeftStr	文字列の先頭にあるスペース（もしくは指定文字）を取り除く。
TrimRightStr	文字列の末尾にあるスペース（もしくは指定文字）を取り除く。
UpperStr	文字列を大文字に変換する。
LowerStr	文字列を小文字に変換する。
ReplaceStr	指定された文字を他の文字に置きかえる。
CStrToArray	セパレータ文字列を区切りとして、文字列を配列に変換する。
CArrayToStr	配列を、セパレータ文字列を付加した文字列に変換する。
Chr	指定したアスキーコードに対応する文字を返す。

＜次頁につづく＞

＜前頁のつづき＞

Value	
GetQuality	値の品質フラグを取得する（"GOOD" or "BAD" or "UNCERTAIN"）。
IsQualityGood	値の品質フラグが GOOD かどうか調べる。
IsQualityBad	値の品質フラグが BAD かどうか調べる。
IsQualityUncertain	値の品質フラグが UNCERTAIN かどうか調べる。
GetValueDimention	値の次元数を求める。
GetValueElement	値の各次元ごとの要素数を求める。
GetValueTotalElement	値の全要素数を求める。
GetValueType	値のデータ型を求める。
GetRemainder	値１/ 値２の余りを求める。
IsBool	値がブール型かどうか調べる。
IsNum	値が数値型かどうか調べる。
IsStr	値が文字型かどうか調べる。
IsTime	値が時間型かどうか調べる。
IsObj	値がオブジェクト型かどうか調べる。
CBool	値をブール型に変換する。
CNum	値を数値型に変換する。
CStr	値を文字列に変換する。
CTime	値を時刻型に変換する。
CTimeStr (又は、TS)	値を時刻文字列に変換する。
CTimeSpanStr	値を時間幅文字列に変換する。
CNumStr (又は、NS)	フォーマットに従った数値文字列を作成する。
CNumToHex	値を 16 進数の文字列に変換する。
CNumToOct	値を 8 進数の文字列に変換する。
CNumToBin	値を 2 進数の文字列に変換する。
CNumToExp	値を指数表記（1.23e+001 等）の文字列に変換する。
CHexToNum	16 進数の文字列を値に変換する。
COctToNum	8 進数の文字列を値に変換する。
CBinToNum	2 進数の文字列を値に変換する。
CNumToComma	数値を３桁毎のカンマ「,」で区切られた数値文字にする
CCommaToNum	３桁毎のカンマ「,」で区切られた数値文字を数値にする。
CScript	c(1,2,3,4）などスクリプトで使用される値の表記文字列に変換する。
Nums	連続した数値の配列を生成する。
IntStrs	連続した数値文字（整数）の配列を生成する。
SortValue	値を昇順もしくは降順に並び替える。
GetUniqueValue	配列から重複した値を削除し昇順もしくは降順に並び替える。
IsExist	比較リスト（配列）内に値が存在するかどうかを求める。
SelectVal (又は、SV)	条件に従い値を選択する。

＜次頁につづく＞

<前頁のつづき>

Time	
GetTickTime	Windows 起動からの経過時間を、ミリ秒単位で取得する。
GetClientTime	現在時刻を取得する。
IsCurrentTime	現在時刻（指定時刻）になったかどうか調べる。
GetYear	時刻から年を取得する。
GetMonth	時刻から月を取得する。
GetDay	時刻から日を取得する。
GetHour	時刻から時を取得する。
GetMinute	時刻から分を取得する。
GetSecond	時刻から秒を取得する。
GetDayOfYear	時刻から年の通算日を求める（1 から 366）
GetDayOfWeek	時刻から曜日を求める（1-日曜日 2-月曜日 3-火曜日 4-水曜日 5-木曜日 6-金曜日 7-土曜日）
GetRelativeTime	ある時刻から相対的に時刻を求める。

■Math メソッド	
Math.LinearTrans (Math.LT)	工学値変換を行う。
Math.Abs	絶対値を求める。
Math.Ceil	切り上げした数を求める。
Math.Floor	切り捨てした数を求める。
Math.Sum	配列から合計値を求める。
Math.Ave	配列から平均値を求める。
Math.Max	配列から最大値を求める。
Math.Min	配列から最小値を求める。
Math.Random	乱数を求める。
Math.Log / Math.Log10	対数を求める。
Math.Pow	x の y 乗を求める。
Math.Sqrt	平方根を求める。
Math.Nthroots	べき乗根を求める。
Math.AddingUp	積算値を求める。
Math.StDev	標準偏差を求める。
Math.Correl	相関係数を求める。
Math.LineSt	線形回帰分析（最小二乗法）で求められる直線の傾きとオフセットを求める。
Math.Sin/Cos/Tan/Asin/Acos/Atan	サイン/コサイン/タンジェント/アークサイン/アークコサイン/アークタンジェントを求める。
Math.Exp	指数を求める。
Math.Summary	サマリー（合計値、平均値、最大値、最小値）を求める。

「スクリプトマニュアル」Copyright © 2010 Roboticsware, Inc. All rights reserved.
2010 年 6 月 1 日 発行　著作・発行 株式会社ロボティクスウェア
〒331-0811 埼玉県さいたま市北区吉野町 1-11-1　TEL:048-782-9861 FAX:048-782-9862

■本書は著作権上の保護を受けています。本書の一部または全部について、株式会社ロボティクスウェアの
文章での許諾を得ずに、いかなる方法においても無断で複写・複製することは禁じられています。
■記載されている会社名、製品名、ブランド名は、対応する法人または個人の（登録）商標です。
■このソフトウェアの仕様及び本書に記載されている事項は、将来予告無しに変更することがあります。