【コメントの記述方法、4つのパターン】

【行末コメント】

サンプルコード

```
ソースコード部分です。
|| これは１行コメントです
ソースコード部分です。|| 行末にもコメントが書けます
ソースコード部分です。
```

【複数行のコメント】

サンプルコード

```
ソースコード部分です。|# これもコメントです。１行でも書けます。 #| ソースコード部分です。
|#
 これは
 複数行コメント
 です。
 #|
ソースコード部分です。
```

【タグ付きコメント】

サンプルコード

```
ソースコード部分です。
|test#
このコメントは、短いコメントを含むソース コード部分をコメントアウトする場合に使用できます。
ソースコードの部分ですがコメントアウトされてます。|| 行末コメント
|# 複数行
   コメント #|
|other-tag# 他のタグが付与されたコメント #other-tag|
#test|
ソースコード部分です。
```

【長さ指定のコメント】

サンプルコード

```
||長さ指定コメントを使えば、コメント終了識別子も１つの文字として使えます。
ソースコード部分です。 |20# please#|&#@|ignore #20| ソースコード部分です。
```

【変数の使い方は？】

【変数の使用】

サンプルコード

```
||変数は、値をメモリに格納する 1 つの方法です。
||変数にはそれぞれ名前があり、これを使用して値にアクセスしたり、新しい値を代入します。
||変数を作成して初期値を与えるには let 式を使用します。

{let x = 13}  || 初期値を１３として宣言しています。


||set 式を使用して、変数の値を変更することもできます。

{set x = 200}  || 値を２００に変更してます。
```

【valueとdoの使い方は？】

【value と do 式の使用】

サンプルコード

```
|| 変数を（初期値１３）定義する
{let current-floor = 13}

|| valueで囲まずに使った場合のサンプルです。
|| 実行結果を確認してください。current-floorがそのまま出力されています。
current-floor

|| valueで囲んだ場合のサンプルです。
|| 実行結果を確認してください。current-floorの値が出力されています。
{value current-floor}


{do
    || 変数を（初期値１３）定義する
    let current-floor = 13

    || １３＋１の結果を設定する
    set current-floor = current-floor + 1

    || さらに１を加算する
    set current-floor = current-floor + 1

    || do 式は 1 つ以上の Curl 言語式を実行しますが値を生成しません。
    || current-floorと書いても画面には何も出力されません。
    || 実行結果確認してください。
    current-floor
}
```

実行結果

```
current-floor
13
```

【演算式の記述方法は？】

【演算式】

サンプルコード

```
{value
    ||変数を定義する
    let a = 5
    let b = 1
    let c = 4
    let d = 2

    ||計算結果を表示する
    ||実行結果を確認してください。
    (a * d) - (b * c)
}
```

実行結果

```
6
```

【Curlのリテラルの記述方法、7つのパターン】

【整数リテラル】

サンプルコード

```
||以下のコードは、有効な整数リテラルを示したものです。
{VBox
    12345
}
{VBox
    0000012345
}
{VBox
    -12345
}
{VBox
    0xFFFFFFFFu
}
```

【浮動小数点リテラル】

サンプルコード

```
||以下のコードは、有効な浮動小数点リテラルを示したものです。
{VBox
    12345.0
}
{VBox
    123.45
}
{VBox
    .12345
}
{VBox
    -12345.0
}
```

【ブール値リテラル】

サンプルコード

```
||以下のコードは、ブール値リテラルを示したものです。
{VBox
    true
}
{VBox
    false
}
```

【文字リテラル】

サンプルコード

```
||文字リテラルは 1 つの文字です。次のいずれかのフォームを使用します。
||一重引用符で囲まれた 1 つの文字。例：'a'
||一重引用符で囲まれたエスケープ シーケンス。例：'¥n'
'a'
```

【文字列リテラル】

サンプルコード

```
{VBox
    || 文字列リテラルです。空白を含めることができます。
    "string literal text",

    || 空文字列です。
    ""
      ,

    || ダブルクォーテーションを含める場合です。
    "¥"",

    || 複数行にも書けます
    "This is a
    multiple line
    string",

    || 複数行に余分な空白を含めた書いた場合です。
    || 実行結果を確認してください。
    "This is a
    multiple line

    string with extra         whitespace",

    || 文字列結合の方法です。
    "¥"" & "string literal text" & "¥""
}
```

実行結果

```
string literal text
"
This is a multiple line string
This is a multiple line string with extra whitespace
"string literal text"
```

【数量リテラル】

サンプルコード

```
||数量リテラルは、数字リテラルと後に続く測定単位で構成され、間に空白は入りません。
{VBox
    5pt
}
{VBox
    5cm
}
{VBox
    5(m^2)
}
{VBox
    10(ft/s)
}
{VBox
    5degrees
}
```

【nullリテラル】

サンプルコード

```
||null 値は、拡張クラス タイプ（ヌル型を許容）の変数
||およびプロシージャ タイプの変数がオブジェクトを参照しない場合に
||プレースホルダーとして使用される特殊な定数です。
null
```

【式の記述方法、アクセス式や代入、値の増減方法は？】

【フィールド アクセス式】

サンプルコード

```
||以下のコードは、フィールド アクセス式を使ったサンプルです。
{value
    || 文字列の定義と初期値を設定する
    let str:String = {String "Hello World!"}

    || 文字列のサイズを確認する
    || Stringクラスのsizeアクセサ
    || 実行結果を確認してください。
    str.size
}
```

実行結果

```
12
```

【配列アクセス式】

サンプルコード

```
||以下のコードは、配列アクセス式を使ったサンプルです。

|| intを要素に持つ配列の定義する
{let arr:{Array-of int} = {{Array-of int} 98, 99, 100, 101, 102, 103}}

|| インデックス：3 の値を表示する
{value arr[3]}

|| インデックス：3 に値を設定する
||  （インデックスは、ゼロ始まり）
{set arr[3] = 13}

|| インデックス：3 の値を表示する
{value arr[3]}
```

実行結果

```
101
13
```

【代入ステートメント】

サンプルコード

```
||以下のコードは、代入に関するサンプルです。

||変数
set var = 30
||フィールド
set obj.field = true
||オプション
set obj.option = true
||配列要素
set a[1] = "gloves"
||StringBuf 要素
set sb[5] = 'e'
||ハッシュテーブル要素
set h["cat"] = 37
||複数の値
set (x, y) = {return-2-values}
```

【値の増減】

サンプルコード

```
||代入ステートメントの代わりに、inc と dec を値の増減に使用できます。
||inc は指定した量の値を増加させます。
||同様に、dec は指定した量の値を減少させます。

{value
    let a:int = 1
    {inc a}
    a
}
```

実行結果

```
2
```

【演算子の記述方法、10のサンプル】

【演算子の優先順位】

サンプルコード

```
||以下は、演算の優先順を確認するためのサンプルです。
{text
    3 + 4 * 5 = {value 3 + 4 * 5}

    (3 + 4) * 5 = {value (3 + 4) * 5}
}
```

実行結果

```
3 + 4 * 5 = 23
(3 + 4) * 5 = 35
```

【四則演算子】

サンプルコード

```
||以下は、四則演算子を使ったサンプルです。
{value
    || 変数を定義する
    let x:int = 28
    let y:double = 3.0
    let z:int = 3

    {text
        || "x / y" の値を出力し、結果のデータ型を出力する
        || ルールにより、結果は double 型になる
        x / y is ... {value x / y}, which is a {type-of x / y}

        || "x / z" の結果を出力し、結果のデータ型を出力する
        x div z is ... {value x div z}, which is a {type-of x div z}
    }
}
```

実行結果

```
x / y is ... 9.33333, which is a double
x div z is ... 9, which is a int
```

【リレーショナル演算子】

サンプルコード

```
||以下は、リレーショナル演算子を使ったサンプルです。
{value
    || 変数を定義する
    let a:int = 5
    let b:int = 6

    || 各比較演算子を使い、"a" と "b" を比較する
    {text a is {value a}, b is {value b}
        {br}a == b is ... {value a == b}
        {br}a != b is ... {value a != b}
        {br}a > b is ... {value a > b}
        {br}a < b is ... {value a < b}
        {br}a >= b is ... {value a >= b}
        {br}a <= b is ... {value a <= b}
    }
}
```

実行結果

```
a is 5, b is 6
a == b is ... false
a != b is ... true
a > b is ... false
a < b is ... true
a >= b is ... false
a <= b is ... true
```

【論理演算子】

サンプルコード

```
||以下は、論理演算子を使ったサンプルです。
{value
    || 変数 "a" と "b" を宣言し初期化する
    let a:bool=true
    let b:bool=false

    || "a" と "b" と共に論理演算を行う
    {text a is {value a}, b is {value b}
        {br}a and b is ... {value a and b}
        {br}a and true is ... {value a and true}
        {br}a and (5 > 2) is ... {value a and (5 > 2)}
        {br}a or b is ... {value a or b}
        {br}not a is ... {value not a}
        {br}not b is ... {value not b}
    }
}
```

実行結果

```
a is true, b is false
a and b is ... false
a and true is ... true
a and (5 > 2) is ... true
a or b is ... true
not a is ... false
not b is ... true
```

【文字列演算子】

サンプルコード

```
||以下は、文字列演算子を使ったサンプルです。
{value
    || int、char、double、bool、String 型の変数を宣言し初期化します。
    let i:int = 3
    let c:char = '='
    let d:double = 3.7
    let b:bool = false
    let s:String = "a horse"

    || 文字列を連結します。
    let temp1:String = s & " is " & s & " ... "
    let temp2:String = i & c & d & " is " & b
    let result:String = temp1 & temp2

    || "result" の文字列を返します。
    result
}
```

実行結果

```
a horse is a horse ... 3=3.7 is false
```

【強制変換演算子】

サンプルコード

```
||以下は、強制変換演算子を使ったサンプルです。
{value
    || 変数 "d" を double 型で宣言し、
    || 37.7 で初期化する
    let d:double = 37.7

    || int 型に変換した変数 "d" の値を表示する
    d asa int
}
```

実行結果

```
37
```

【NULL 許容の拡張演算子】

サンプルコード

```
||"NULL 許容" 演算子は変数の宣言で使用され、変数のデータ型名の前に付けて、
||その変数値が指定したデータ型または null のいずれかになることを示します。
||この演算子が必要な場合とその使用方法については、Curl開発者ガイドを確認してください。
null
```

【value 式】

サンプルコード

```
||以下は、value 式を使ったサンプルです。

||value式に演算を含める
I am {value 8 * 12} months old.

||文字列を結合する
My name is {value "Barbra " & 'J' & " Streisand"}.

|| 数量算術演算（単位が違うことに注意）を行う
I am {value 1m + 80cm} tall.

|| 論理演算を行う
Are these statements {value true or false} or {value not true}?
```

実行結果

```
I am 96 months old.
My name is Barbra J Streisand.
I am 1.8m tall.
Are these statements true or false?
```

【do 式】

サンプルコード

```
||以下は、do 式を使ったサンプルです。

||変数を定義する
{let a:int = 1}
||do式にてコード ブロック内の各ステートメントや式を評価（ここでは、変数aの値を増加）する
{do
    {inc a}
}
||value式にて変数aの値を表示する
{value a}
```

実行結果

```
2
```

【コード ブロック値の受け取り】

サンプルコード

```
||以下は、コード ブロック値を利用したサンプルです。

|| 0と1の間の乱数値を生成する
{let random-number:float = {{Random}.next-float}}
||
|| "if-else"のコードブロックを評価する
|| 実行結果を確認してください
A coin was tossed and came up
{if random-number < 0.5 then
    "heads"
 else
    "tails"
}
```

実行結果

```
A coin was tossed and came up heads
```

【条件式の記述方法、7つのパターン】

【if 式】

サンプルコード

```
{value
    || 変数を定義する
    let x:int=1
    let y:int=2
    let message:VBox={VBox}

    || "x"は、"y"より大きいまたは "y"に等しい場合に以下を実行する
    {if x >= y then
        {message.add "x is greater than or equal to y!"}
    }

    || "x"は、"Y"より小さい場合に以下を実行する
    {if x < y then
        {message.add "x is less than y!"}
    }

    || "message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
x is less than y!
```

【if ... else 式】

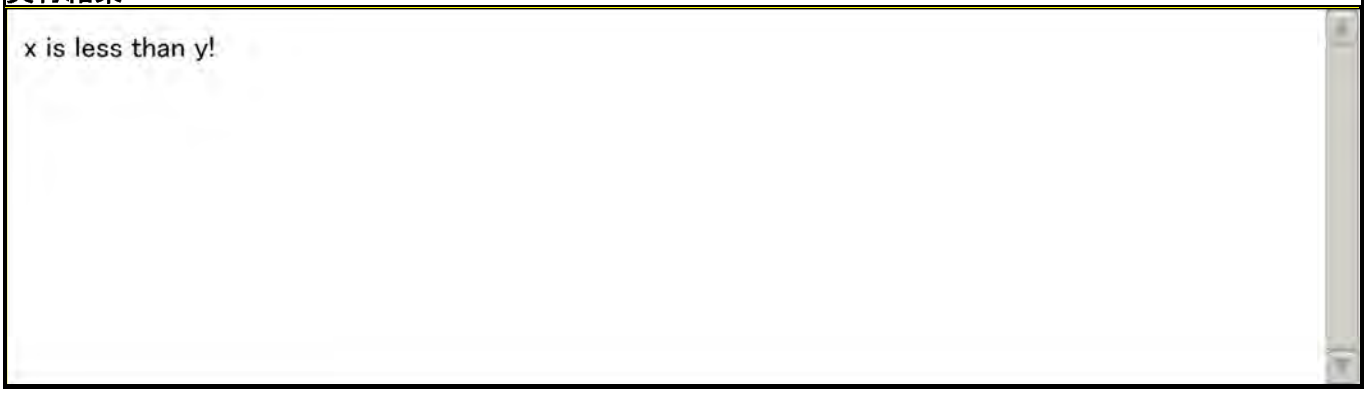
サンプルコード

```
{value
    || 変数を定義する
    let x:int=1
    let y:int=2
    let message:VBox={VBox}

    || "x"は、"y"より大きいまたは "y"に等しい場合に以下を実行する
    {if x >= y then
        {message.add "x is greater than or equal to y!"}
    || それ以外の場合に以下を実行する
     else
        {message.add "x is less than y!"}
    }

    || "message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果



【if ... elseif 式】

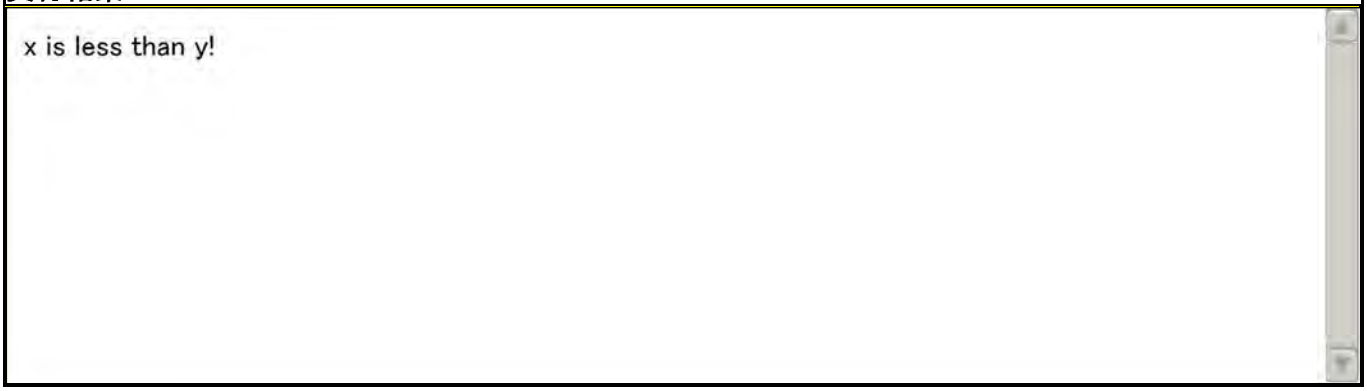
サンプルコード

```
{value
    || 変数を定義する
    let x:int=1
    let y:int=2
    let message:VBox={VBox}

    || "x"は、"y"に等しい場合に以下を実行する
    {if x == y then
        {message.add "x is equal to y!"}
        || "x"は、"y"より大きい場合に以下を実行する
     elseif x > y then
        {message.add "x is greater than y!"}
        || 上記以外の場合に以下を実行する
     else
        {message.add "x is less than y!"}
    }

    || "message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果



【if-non-null 式】

サンプルコード

```
||以下は、if-non-null式を使ったサンプルです。

||サンプル１
{let msgx:#String}
{value
    let x:String="1"
    let y:#String
    ||x が non-null の場合、trueになります。
    {if-non-null x then
        {set msgx = "if-non-null x"}
    }
    ||実行結果を確認してください
    msgx
}

||サンプル２
{let msgy:#String}
{value
    let x:String="1"
    let y:#String
    ||x が non-null の場合、trueになります。
    {if-non-null y then
        {set msgy = "if-non-null y"}
    ||x が null の場合、elseに分岐されます。
    else
        {set msgy = "if-non-null else y"}
    }
    ||実行結果を確認してください
    msgy
}
```

実行結果

```
if-non-null x

if-non-null else y
```

【unless 式】

サンプルコード

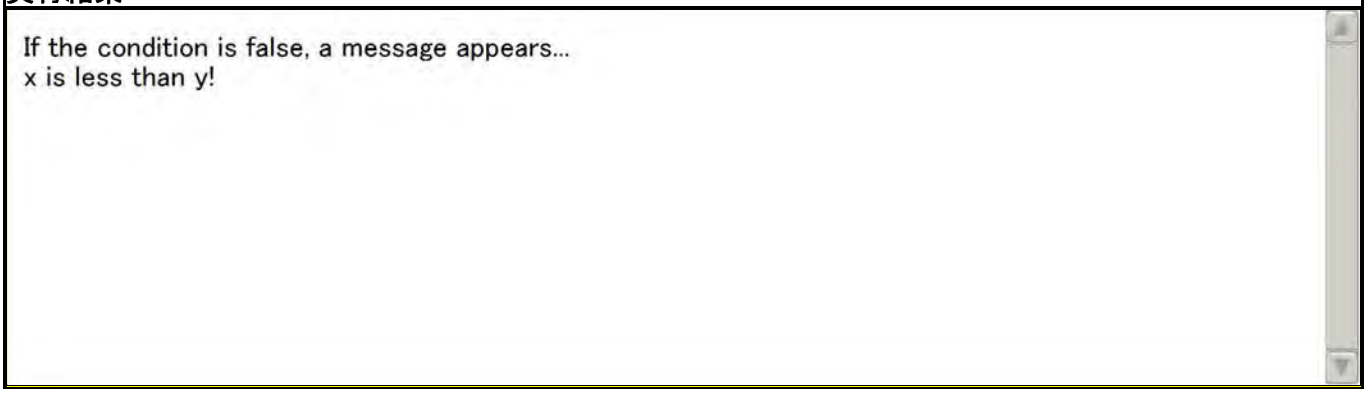
```
{value
    || 変数を定義する
    let x:int=1
    let y:int=2
    let message:VBox={VBox}

    {message.add "If the condition is false, a message appears..."}

    || 条件が true でない場合、1 つまたは複数の式を評価する
    {unless x >= y do
        {message.add "x is less than y!"}
    }

    || "message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果



【switch 式】

サンプルコード

```
||以下は、switch 式を使ったサンプルです。
{value
    let message:VBox={VBox}

    {for y:int = 0 to 3 do
        ||switch 式では、与えられた式を一連の値と比較することができます。
        {switch y
         case 0 do
            {message.add "y is equal to 0!"}
         case 1 do
            {message.add "y is equal to 1!"}
         case 2 do
            {message.add "y is equal to 2!"}
         else
            {message.add "y is greater than 2!"}
        }
    }
    || 実行結果を確認してください
    message
}
```

実行結果

```
y is equal to 0!
y is equal to 1!
y is equal to 2!
y is greater than 2!
```

【type-switch 式】

サンプルコード

```
||以下は、type-switch 式を使ったサンプルです。
{value
    let y:int
    let message:Frame = {Frame}

    ||type-switch 式を利用することで値のデータ型を使用して
    ||実行するコードブロックを決定することができます。
    {switch y using isa
     case bool do
         {message.add "y is a Boolean!"}
     case char do
         {message.add "y is a character!"}
     case int8, int16, int32, int64, int, uint8, uint16 do
         {message.add "y is an integer!"}
     case float, double do
         {message.add "y is a floating-point number!"}
     else
         {message.add "y is not a simple primitive value!"}
    }
    || 実行結果を確認してください
    message
}
```

実行結果

```
y is an integer!
```

【ループ式の記述方法、9つのパターン】

【while 式】

サンプルコード

```
{value
    ||変数を定義する
    let start:int=1
    let message:HBox={HBox}

    ||10未満では、次の手順を実行します。
    ||  メッセージのstartの値を設定する
    ||  インクリメントの開始（その値に1を追加）する
    {while start < 10 do
        {message.add start}
        set start = start + 1
    }

    || "message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
123456789
```

【until 式】

サンプルコード

```
{value
    ||変数を定義する
    let start:int=1
    let message:HBox={HBox}

    ||スタートが10に等しくなるまで、次の操作を行います。
    || メッセージ内のstartの値を配置する
    || インクリメントの開始（その値に1を追加します）する
    {until start == 10 do
        {message.add start}
        set start = start + 1
    }

    || 実行結果を確認してください
    message
}
```

実行結果

```
123456789
```

【for … to … step … do …】

サンプルコード

```
{value
    ||変数を定義する
    let message:HBox={HBox}

    ||xの開始値を0とし、xが8になるまで（8含む）繰り返す
    ||xは1ずつカウントアップする
    {for x:int=0 to 8 do
        {message.add x}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
012345678
```

【for … below … step … do …】

サンプルコード

```
{value
    ||変数を定義する
    let message:HBox={HBox}

    ||xの開始値を0とし、xが7になるまで（8含まない）繰り返す
    ||xは1ずつカウントアップする
    {for x:int=0 below 8 do
        {message.add x}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
01234567
```

【for … downto … step … do …】

サンプルコード

```
{value
    ||変数を定義する
    let message:HBox={HBox}

    ||xの開始値を8とし、xが0になるまで（0含む）繰り返す
    ||xは1ずつカウントダウンする
    {for x:int=8 downto 0 do
        {message.add x}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
876543210
```

【for … above … step … do …】

サンプルコード

```
{value
    ||変数を定義する
    let message:HBox={HBox}

    ||xの開始値を8とし、xが1になるまで（0含まない）繰り返す
    ||xは1ずつカウントダウンする
    {for x:int=8 above 0 do
        {message.add x}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
87654321
```

【for ... in ... do ...】

サンプルコード

```
{value
    ||変数を定義する
    let message:VBox={VBox}
    let a:{Array-of int}={{Array-of int} 10, 20, 30}

    || 配列"a"内の各要素"v"全てに対して繰り返す
    {for v:int in a do
        {message.add v}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
10
20
30
```

【for key ... in ... do ...】

サンプルコード

```
{value
    ||変数を定義する
    let message:VBox={VBox}
    let a:{Array-of int}={{Array-of int} 10, 20, 30}

    || 配列"a"内の各要素のキー"k"全てに対して繰り返す
    {for key k:int in a do
        {message.add k}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
0
1
2
```

【for ... key ... in ... do ...】

サンプルコード

```
{value
    ||変数を定義する
    let message:VBox={VBox}
    let a:{Array-of int}={{Array-of int} 10, 20, 30}

    || 配列"a"内の各要素全てに対して繰り返す（キーを"k"、要素を"v"に代入）
    {for v:int key k:int in a do
        {message.add {text {value k} ... {value v}}}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
0 ... 10
1 ... 20
2 ... 30
```

【分岐式の記述方法、3つのパターン】

【continue 式】

サンプルコード

```
{value
    ||変数を定義する
    let i:int=0
    let message:VBox={VBox}

    || iが2より小さい間は、次の手順を実行する
    || ・カウントアップI（その値に1を追加する）
    || ・メッセージ内の適切な文字列を配置する
    || ・forループを実行する
    || ・メッセージ内の適切な文字列を配置する
    {while i < 2 do
        set i = i + 1
        {message.add "outer loop: start iteration for i = " & i}

        ||xの開始値を0とし、xが5になるまで（5含む）繰り返す
        ||xは1ずつカウントアップする
        {for j = 0 to 5 do
            |||jが3の場合、現在のループの繰り返しの最後に制御を移す
            {if j == 3 then
                {continue}
            }
            {message.add "inner loop: j = " & j}
        }
        {message.add "outer loop: end of iteration"}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
outer loop: start iteration for i = 1
inner loop: j = 0
inner loop: j = 1
inner loop: j = 2
inner loop: j = 4
inner loop: j = 5
outer loop: end of iteration
outer loop: start iteration for i = 2
inner loop: j = 0
inner loop: j = 1
inner loop: j = 2
inner loop: j = 4
inner loop: j = 5
outer loop: end of iteration
```

【break 式】

サンプルコード

```
{value
    ||変数を定義する
    let i:int=0
    let message:VBox={VBox}

    || iが2より小さい間は、次の手順を実行する
    || ・カウントアップI（その値に1を追加する）
    || ・メッセージ内の適切な文字列を配置する
    || ・forループを実行する
    || ・メッセージ内の適切な文字列を配置する
    {while i < 2 do
        set i = i + 1
        {message.add "outer loop: start iteration for i = " & i}

        ||xの開始値を0とし、xが4になるまで（4含む）繰り返す
        ||xは1ずつカウントアップする
        {for j = 0 to 4 do
            ||jが3の場合、ループ外に制御を移す
            {if j == 3 then
                {break}
            }
            {message.add "inner loop: j = " & j}
        }
        {message.add "outer loop: end of iteration"}
    }

    ||"message"を表示する
    || 実行結果を確認してください
    message
}
```

実行結果

```
outer loop: start iteration for i = 1
inner loop: j = 0
inner loop: j = 1
inner loop: j = 2
outer loop: end of iteration
outer loop: start iteration for i = 2
inner loop: j = 0
inner loop: j = 1
inner loop: j = 2
outer loop: end of iteration
```

【return 式】

サンプルコード

```
||プロシージャコールの値として引数×２の値を返す
{define-proc public {my-proc1 i:int}:int
    {return i*2}
}

|| 実行結果を確認してください
{value
    {my-proc1 2}
}
```

実行結果

```
4
```

【グローバルプロシージャの使い方は？】

【複数の戻り値のあるプロシージャの呼び出し】

サンプルコード

```
||以下は、複数の戻り値のあるプロシージャを説明するためのサンプルです。
||-------------------------------------------------------------------------
||数値計算ライブラリの round プロシージャは、
||2 つの引数の指数 (2 番目の引数は既定では 1) を最も近い整数値に丸めた値と、
||丸められた指数から残された剰余の 2 つの値を返します
||-------------------------------------------------------------------------

|| 最初の戻り値を直接使用する
The first value returned by |"{round 4.7}"| is {round 4.7}.

||最初の戻り値にて変数を初期化する
{value
    let rounded:double = {round 4.7}
    {text The first return value is {value rounded}.}
}

||二つの変数、戻り値ごとに初期化する
||実行結果を確認してください
{value
    let (rounded:double, leftover:double) = {round 4.7}
    {text The rounded value is {value rounded} and the remainder
        is {value leftover}.
    }
}
```

実行結果

```
The first value returned by {round 4.7} is 5.

The first return value is 5.

The rounded value is 5 and the remainder is -0.3.
```

【引数の記述方法、8つのパターン】

【プロシージャのデータ型】

サンプルコード

```
||プロシージャを定義する（文字列が文字 'a' で始まる場合、falseを返す）
{define-proc {eliminate-a str:String}:bool
    {return str[0] != 'a'}
}

{value
    ||変数を定義する
    let fruits:{Set-of String} =
        {new {Set-of String}, "apple", "banana", "cherry"}

    ||'a' で始まる要素をフィルタリングする
    {fruits.filter eliminate-a}

    ||配列要素を表示する
    ||実行結果を確認してください
    {String {splice fruits}}
}
```

実行結果

```
cherrybanana
```

【キーワード引数の使用】

サンプルコード

```
||プロシージャ（2つの数値の和を返す）を定義する
{define-proc public {add-two-numbers number1:int=12, number2:int=13}:int
    {return number1 + number2}}

||引数を指定し、プロシージャを呼び出す
||実行結果を確認してください
{add-two-numbers}
{add-two-numbers number1=12}
{add-two-numbers number2=13}
{add-two-numbers number1=12, number2=13}
{add-two-numbers number2=13, number1=12}
{add-two-numbers number2=69, number1=12, number2=13}
```

実行結果

```
25 25 25 25 25 25
```

【位置引数とキーワード引数の使用】

サンプルコード

```
||プロシージャ（2つの数値の和を返す）を定義する
{define-proc public {add-two-numbers number1:int, number2:int=13}:int
    {return number1 + number2}
}

||引数を指定し、プロシージャを呼び出す
||実行結果を確認してください
{add-two-numbers 12, number2=13}
{add-two-numbers number2=13, 12}
{add-two-numbers number2=69, 12, number2=13}
```

実行結果

```
25 25 25
```

【残余引数】

サンプルコード

```
||プロシージャ（残余引数を受け取りStringBufを返す）を定義する
{define-proc {my-proc ...}:StringBuf
    ||変数を定義する
    let result:StringBuf = {StringBuf}

    ||残余引数コンテナの各値を連結する
    {for v in ... do
        {result.concat v}
        {result.append ' '}
    }
    ||値を返却する
    {return result}
}

{value
    ||変数を定義する
    let s1:String = "Here"
    let s2:String = "comes"
    let s3:String = "Curl!"

    ||引数を指定し、プロシージャを呼び出す
    ||実行結果を確認してください
    {my-proc s1, s2, s3}
}
```

実行結果

```
Here comes Curl!
```

【splice1】

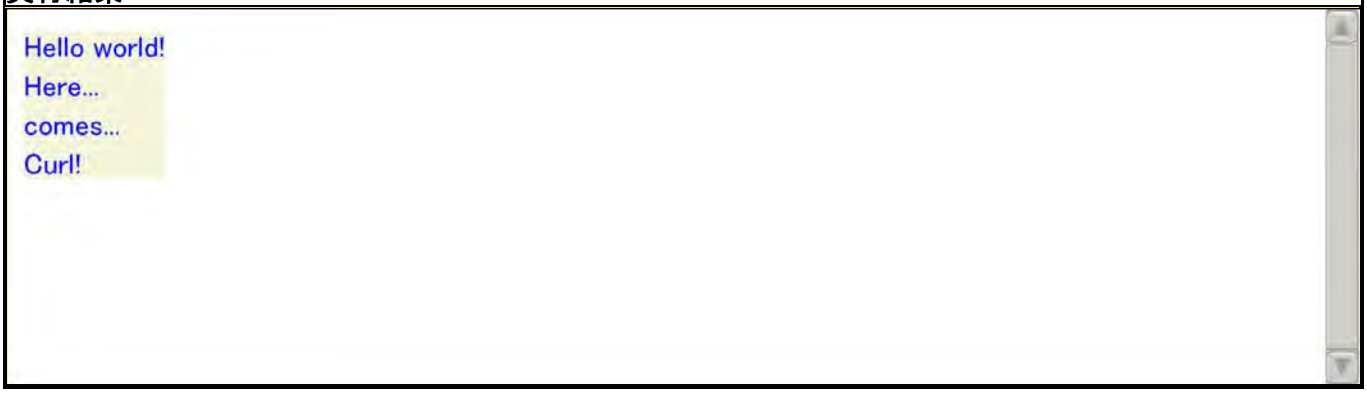
サンプルコード

```
||以下のコードは、spliceを使ったサンプルです。
{value
    ||Argumentsクラスを定義する
    let args:Arguments = {Arguments background = "beige",
                                color = "blue",
                                "Here...",
                                "comes..."}

    ||spaced-vboxオブジェクトの呼び出しとして利用する
    let vb:VBox = {spaced-vbox
                        "Hello world!",
                        {splice args},
                        "Curl!"}

    ||"vb"を表示する
    ||実行結果を確認してください
    vb
}
```

実行結果



【splice2】

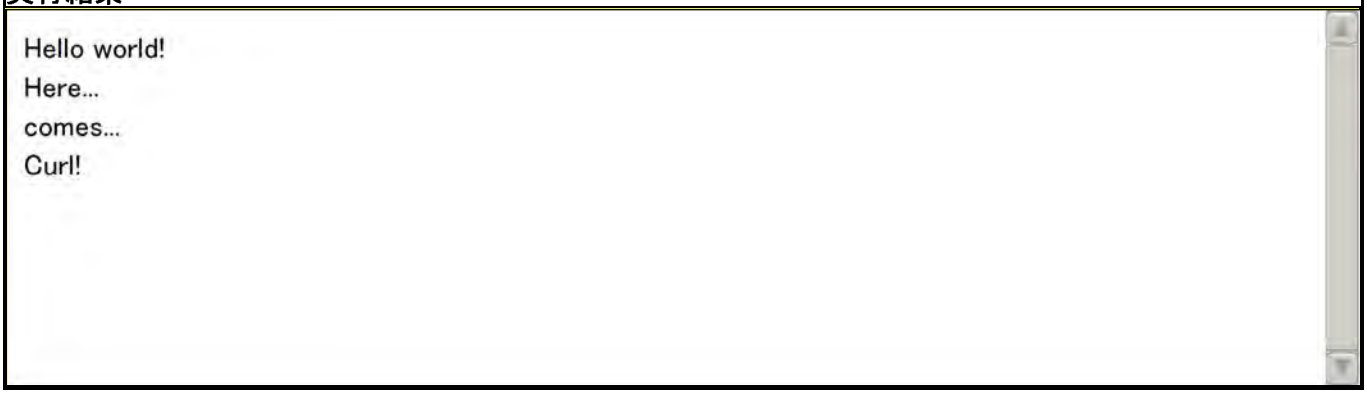
サンプルコード

```
||以下のコードは、spliceを使ったサンプルです。
{value
    ||変数を定義する
    let some-strings:StringArray =
        {new StringArray, "Here...", "comes...", "Curl!"}

    ||spaced-vboxオブジェクトの呼び出しとして利用する
    let vb:VBox = {spaced-vbox
                        "Hello world!",
                        {splice some-strings}}

    ||"vb"を表示する
    ||実行結果を確認してください
    vb
}
```

実行結果



【プロシージャのデータ型】

サンプルコード

```
||以下のコードは、プロシージャのデータ型を説明するサンプルです。
{value
    ||引数：int、戻り値：intを持つ変数（プロシージャ）を定義する
    let my-proc:#{proc-type {int}:int}
    ||変数を初期化する
    set my-proc =
        {proc {x:int}:int
            {return 3 * x}
        }

    ||2を指定して実行する
    ||実行結果を確認してください
    {my-proc 2}
}
```

実行結果

```
6
```

【引数として匿名プロシージャを渡す】

サンプルコード

```
||以下のコードは、引数として匿名プロシージャを渡しているサンプルです。
{value
    ||変数を定義する
    let fruits:{Set-of String} =
        {new {Set-of String}, "apple", "banana", "cherry"}

    ||'a'で始まる要素をフィルタリングする
    {fruits.filter {proc {str:String}:bool
                       {return not str.empty? and str[0] != 'a'}
                   }
    }

    ||フィルタリングの結果を表示する
    let display:StringBuf = {StringBuf ""}
    {for s:String in fruits do
        || 要素を文字列結合する
        {display.concat s}
        || 空白を追加
        {display.append ' '}
    }

    ||実行結果を確認してください
    display
}
```

実行結果

```
cherry banana
```

【匿名プロシージャの使い方は？】

【匿名プロシージャとスコーピング】

サンプルコード

```
||-----------------------------------------------------
||以下のコードは、匿名プロシージャのスコープを説明するためのサンプルです。
||詳細は、Curl開発者ガイドを確認してください
||-----------------------------------------------------
|| プロシージャを定義する
|| tripleでは、引数で指定された結果の3倍の値を返却する
{define-proc {triple x:int}:int

    || 変数を定義（匿名プロシージャ）する
    let my-proc:{proc-type {int}:int} =
        ||tripleの引数として指定された値と
        ||my-procの引数として指定される値を
        ||乗算した結果を返却する
        {proc {b:int}:int
            {return x * b}
        }

    || my-procに3を指定した結果をtripleの戻り値として返却する
    {return {my-proc 3}}
}

||実行結果を確認してください
{value
    || 9を指定してtripleを実行する
    {triple 9}
}
```

実行結果

```
27
```

【戻り値としての匿名プロシージャ】

サンプルコード

```
||------------------------------------------------------
|| 以下のコードは、匿名プロシージャを戻り値として利用したサンプルです。
||------------------------------------------------------
||  プロシージャを定義する
|| adderでは、匿名プロシージャを返却する
{define-proc {adder x:int}:{proc-type {int}:int}
    || 戻り値として匿名プロシージャを返却する
    {return
        {proc {y:int}:int
            {return x + y}
        }
    }
}

{value
    || 変数（指定した引数に3を加算して返却するプロシージャ）を定義する
    let add3:{proc-type {int}:int} = {adder 3}

    || add3に5を指定して実行する
    {add3 5}
}
```

実行結果

```
8
```

【匿名プロシージャの高度な利用】

サンプルコード

```
||-------------------------------------------------------
||以下のコードは、匿名プロシージャを駆使したサンプルです。
||実行結果とソースコードを確認しながらトレースしてください
||-------------------------------------------------------

|| プロシージャを定義する
|| new-counterでは、複数の匿名プロシージャを返却する
{define-proc {new-counter val:int}:(any, any)
    {return
        || 第一戻り値
        {proc {}:int {return val}},
        || 第二戻り値
        {proc {new:int}:void {set val = new}}
    }
}

|| 以下のコードと実行結果を見ながら確認してください

||変数を定義する
{let (my-getter:any, my-setter:any) = {new-counter 2}}
{let (your-getter:any, your-setter:any) = {new-counter 3}}


||変数（匿名プロシージャ）に対して操作する
{my-getter}       ||実行結果を表示する(2)

{your-getter}     ||実行結果を表示する(3)

{my-setter 7}

{my-getter}       ||実行結果を表示する(7)

{your-getter}     ||実行結果を表示する(3)

{your-setter -1}

{your-getter}     ||実行結果を表示する(-1)

{my-getter}       ||実行結果を表示する(7)
```

実行結果

```
2
3
7
3
-1
7
```

【例外の記述方法、作成／発生／キャッチ】

【例外クラスの作成】

サンプルコード

```
||例外：Exception を継承するクラスを作成します
{define-class public MyException {inherits Exception}

  {constructor public {default message:String}
    {construct-super message}
  }
}
```

【例外の発生】

サンプルコード

```
||例外：Exception を継承するクラスを作成します
{define-class public MyException {inherits Exception}

  {constructor public {default message:String}
    {construct-super message}
  }
}

||throw 式を使用して発生させます
{throw MyException "mysample"}
```

【キャッチの使用例】

サンプルコード

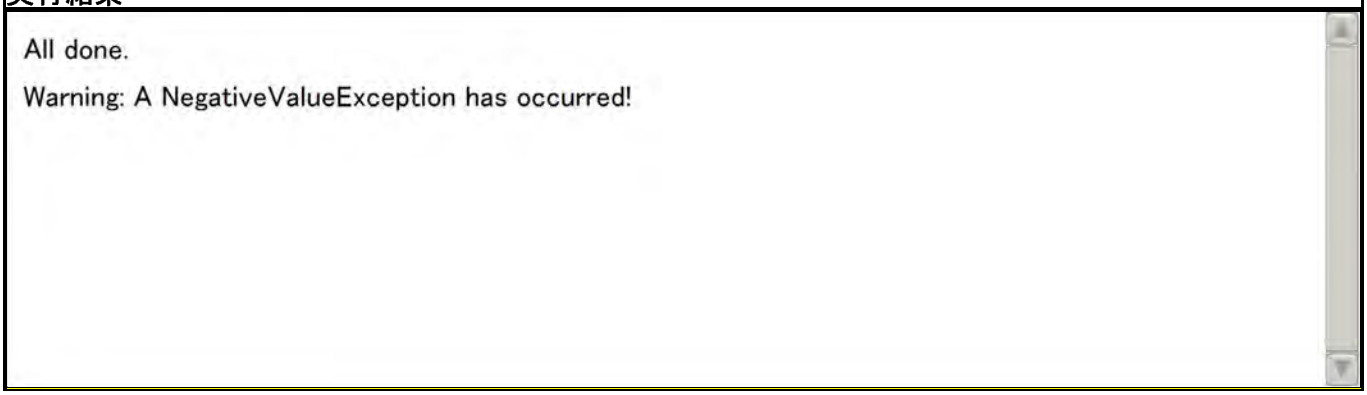
```
||------------------------------------------------------
||以下のコードは、try-catchを利用したサンプルです。
||------------------------------------------------------
{let results:VBox = {VBox}}
{value results}

||例外クラスを定義する
{define-class public NegativeValueException {inherits Exception}
  {constructor public {default message:String}
    {construct-super message}
  }
}

||try 式を catch 句および finally 句と使用します
{try
    ||例外を発生させる
    {throw {NegativeValueException "mysample"} }

 catch e:NegativeValueException do
    ||例外をキャッチする
    {results.add
        "Warning: A NegativeValueException has occurred!"
    }
 finally
    {results.add "All done."}
}
```

実行結果



【例外の例】

サンプルコード

```
||------------------------------------------------------
||以下のコードは、例外を利用した際の一連の流れを示すためのサンプルです。
||------------------------------------------------------
|| 変数を定義する
{let current-exception:Visual = {text-part}}
{let results:VBox = {VBox}}
{value results}

|| 例外クラスを定義する
{define-class public NegativeValueException {inherits Exception}
  {constructor public {default message:String}
    {construct-super message}
  }
}

|| クラスを定義する
{define-class public MyClass
  field value:int

  || コンストラクタを定義する
  {constructor public {default arg:int}
    ||引数チェックを行う
    {if arg < 0 then
        ||例外を発生させる
        {throw
            {NegativeValueException
                {format "The value %i is negative!", arg}
            }
        }
     else
        set self.value = arg
    }
  }
}

{try
    ||例外を発生させるためのコード
    let var:MyClass = {MyClass -1}

 catch e:NegativeValueException do
    || 例外 (NegativeValueException) をキャッチした場合の処理を記述する
    {results.add
        "Warning: A NegativeValueException has occurred!"
    }
 catch e:Exception do
    || その他の例外をキャッチした場合の処理を記述する
    {results.add
        "An unexpected exception has occurred: " & e
    }
 finally
    ||finallyは、例外がスローされたかどうかにかかわらず処理されます
    {results.add "All done."}
}
```

実行結果

```
Warning: A NegativeValueException has occurred!
All done.
```

【マクロの使い方は？】

【簡単なマクロの定義】

サンプルコード

```
||使用上の特別な制限として、使用される場所以外のパッケージでCurl マクロを定義する必要があります
{import * from SWAP-MACROS,
    location = "./swap-macros.scurl"
}
||----------------------------------------------------------------
|#上記ファイルには以下を定義しています
{define-macro public {swap ?a:identifier, ?b:identifier}
    {return
        {expand-template
            {do
                let tmp:{compile-time-type-of ?a} = ?a
                set ?a = ?b
                set ?b = tmp
            }
        }
    }
}
#|
||----------------------------------------------------------------

||変数を定義する
{let first:double = 1.0,
    second:double  = 2.0
}

||作成したマクロを実行する
{swap first, second}

||実行した結果を表示する
{spaced-vbox
    background = "silver",
    {text first: {value first}},
    {text second: {value second}}
}
```

実行結果

【expand-template の使用】

サンプルコード

```
||以下のコードは、expand-templateを利用したサンプルです。
||詳細は、Curl開発者ガイドを確認してください
{value
    ||expand-templateは渡されたリテラル テキストを表す Curl ソース コードを返します
    ||expand-template は主に define-macro と共に使用します
    ||Curl ソースコードを表すテキストを渡します
    let x = {expand-template 5}
    let exp = {expand-template {output ?x + y}}

    ||生成されたソースコード（CurlSource オブジェクト）を表示します
    ||実行結果を確認してください
    {exp.get-text}
}
```

実行結果

```
{output 5 + y}
```

【マクロのパターン マッチング】

サンプルコード

```
||使用上の特別な制限として、使用される場所以外のパッケージでCurl マクロを定義する必要があります
{import swap-expressions from SWAP-MACROS,
    location = "./swap-macros.scurl"
}
||------------------------------------------------------------
|#上記ファイルには以下を定義しています
{define-macro public {swap-expressions ?a:expression, ?b:expression}
    {return
        {expand-template
            {do
                let tmp:{compile-time-type-of ?a} = ?a
                set ?a = ?b
                set ?b = tmp
            }
        }
    }
}
#|
||------------------------------------------------------------
||変数を定義する
{let arr:StringArray = {StringArray "one","two","three"},
    var1:String = "foo",
    var2:String = "bar"
}
||マクロを実行する
{swap-expressions var1, var2}
{swap-expressions var2, arr[2]}

||結果を表示する
||実行結果を確認してください
Now var1 has value {value var1},
var2 has value {value var2},
and arr[2] has value {value arr[2]}.
```

実行結果

```
Now var1 has value bar, var2 has value three, and arr[2] has value foo.
```

【ガベージコレクションの使い方は？】

【ガーベッジ コレクションの利用】

サンプルコード

```
||以下のコードは、ガーベッジ コレクションの挙動を説明するためのサンプルです。
||ガーベッジ コレクションに関する詳細はCurl開発者ガイドを確認してください
{value
    ||ボタンを押すとメモリを消費するコードです
    let make-garbage:CommandButton =
        {CommandButton label = "Make Garbage",
            {on Action do
                {for x = 1 to 1000 do
                    let a:{Array-of char} = {new {Array-of char}}
                    {a.set-size 1024, 'x'}
                }
            }
        }
    ||ボタンを押すとメモリが解放されるコードです
    let clean-garbage:CommandButton =
        {CommandButton label = "Clean Garbage",
            {on Action do
                ||garbage-collect プロシージャを使用して、
                ||ガーベッジ コレクションがプログラム内の一定の場所で実行されていることを
                ||システムに示します
                {garbage-collect}
            }
        }
    {HBox make-garbage, clean-garbage}
}
```