資産管理電卓 LineCalc 説明書

Ver 1.178

・ はじめに

資産管理電卓 LineCalc は自分の資産を計算する際に株価などを WEB 等からコピペで数値を取ってきてそのまま計算できるように、数値の中に “,”(カンマ)を含むことのできる電卓として始め開発されました(例.123,456)。

, 区切りが引数の区切りに使えず、前にスペースを置く必要がありますがあえてそのままにしています(関数名(1, [スペース]2, [スペース]3)スペースは , を入力すると自動で挿入されます)。

その後株の値、自分の現金残高等を関数として定義できるようにして簡易プログラム可能な簡易言語機能(LCS(LineCalcScript))を付け資産を関数の形で管理できる電卓に発展しました。

具体例としては積立における将来想定される出費などをプログラムしておけば、より正確な積立目標をシミュレーションすることができます。

簡易言語(LCS)では資産の今後の経緯や、投資をして行った場合の今後の資産配当額などの計算をプログラムでき、簡単にグラフとして表示することができます。

また、簡易言語(LCS)ではネット関連の組み込み関数が整備されていて WEB からの情報取得のプログラムを書くことが可能です。

・ インストールアンインストール

・インストール

LineCalcSetup.zip を適当なディレクトリに展開してください。
setup.exe を起動してください。
インストーラーが走りますので指示に従ってください。
Edge 版 web を使用する場合は、WebView2 runtime のインストールがされていない場合は、マイクロソフトのサイトより WebView ランタイムを取得してインストールしてください。

LineCalc のアイコンと取扱説明書がデスクトップ上とメニューに作成されますので起動してご使用ください。

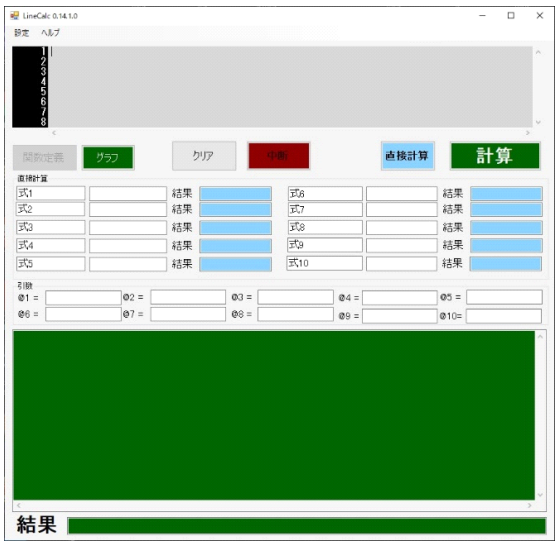
・アンインストール

setup.exe を起動するとインストール済みの場合修復か、削除を効いてきますので削

除を選択して実行してください。
もし、できない旨の表示があった場合は、Windows メニューの設定->アプリから LineCalcSetup を選択してアンインストールしてください。

・ スタートアップ

起動すると以下のような画面が開きます。



上の右に数字で行番号が示してある部分に試しに

(

123456+789123

)*1.03

と改行込みで打ち込み計算ボタンを押してください。

すると結果に、

939956.37

と表示されます。

123,456

と入力して計算ボタンを押すと 123456 と表示されます。

,を入力するとスペースが自動で挿入されますが CTRL を押しながらだと , だけ入力できます(もしくはメニューから機能を無効にできます)。

WEB などでコピーしてきた改行、, 込みの数値をそのままコピペして貼り付けて自由フォームで計算が可能です。

次に、クリアボタンを押して入力を消して、

print(

"合計:", [スペース] 123+456

)

と入力して計算ボタンを押してください。
下の広い緑のエリアに
合計:579
と表示されました。
次に、クリアボタンを押して入力を消して、

```
print(
 "合計:",
 {
  sum=0 ,
  for(i, [スペース] 1, [スペース] 100, [スペース] 1,
   sum=sum+i
  )
 }
)
```

と入力して、計算ボタンを押してください。
for の数字の後の , の前と sum=0 の後ろにはスペースを入れてください。
for(I, [スペース]1, [スペース]100, [スペース]1,
sum=0,
でないと 1,100,1 という数値と認識されてエラーになります。

合計:5050
と表示されました。
区切りは違和感なければ : でも代用できます。
Ex)

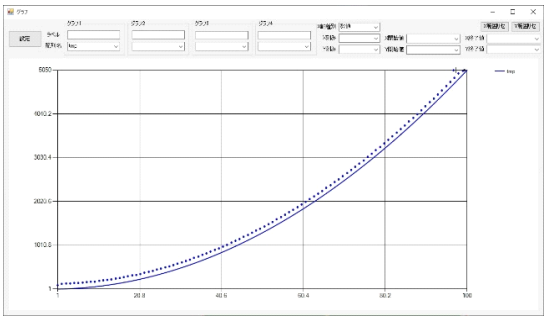
```
 for(cnt:1:1000:1:tmp[cnt]=cnt)
```

次に、クリアボタンを押して入力を消して、

```
    {
     sum=0 ,
     for(i, [スペース] 1, [スペース] 100, [スペース] 1,
      {
       sum=sum+i ,
       tmp[i]=sum
      }
     ),
     grph(tmp)
```

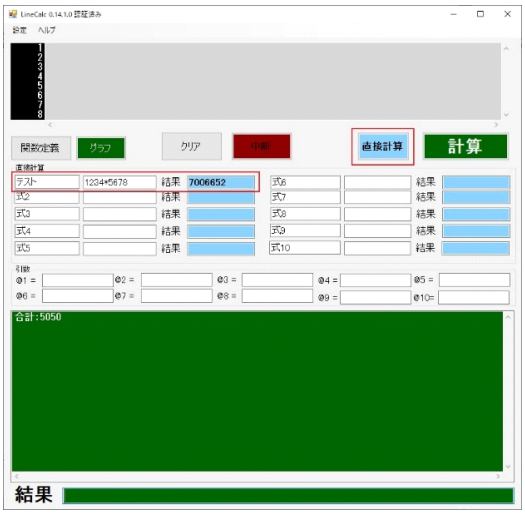
}

と入力して、計算ボタンを押してください。



上記のグラフが表示されました。

このように、単純な計算からプログラムによるシミュレーションまでファイルとしてプログラムを保存することなく LineCalc 内のエディタだけで行う事ができます。



次に”直接計算”と書かれてある下の上図の位置に“テスト”、”1234*5678”と入力して”直接計算”ボタンを押してください。

すると水色のエリアに左に記載した計算内容の結果が表示されます。

計算ボタンでは１つの項目しか計算できませんが、直接計算では良く行う項目を計算することができます。

そして、このまま右上の×ボタンで終了してください。

そして、また立ち上げなおしてください。

すると先ほど入力した式がそのまま残っています。

直接計算の値、関数定義ボタンで登録した内容、メインダイアログの入力内容は終了時に自動保存され立ち上げなおすと前回の終了時の状態が復帰されます。

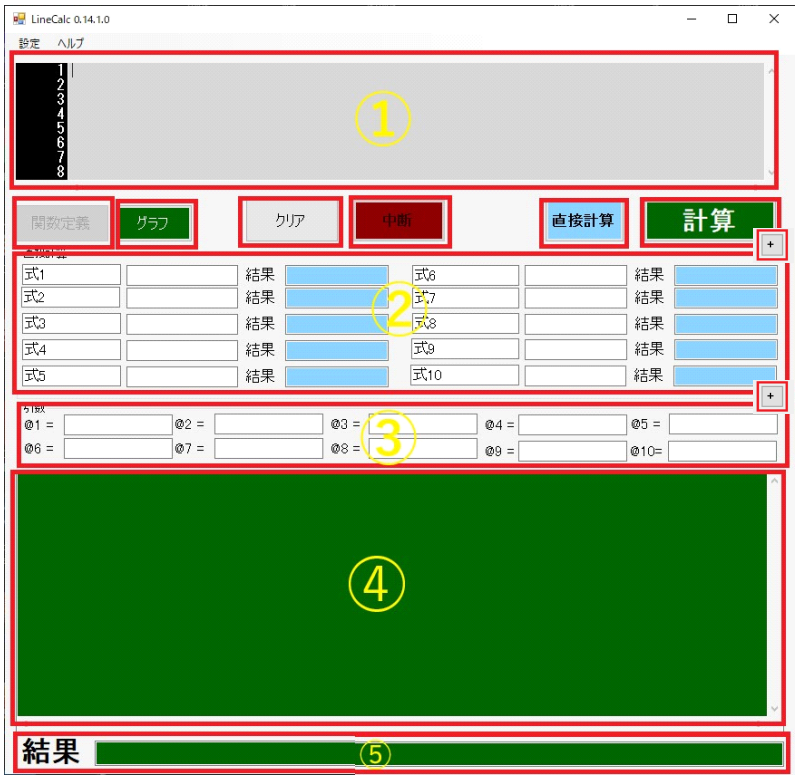
メモのように資産の変化や現在の現金などを記載しておくとそのままファイルとして

保存されます。

・　画面構成

各画面構成を説明します。

画面構成は以下のようになっています。



・　計算式エリア

“計算”ボタンで計算する内容を記載します。

文字列を選択して右クリックすると右クリックメニューを選択できます。

“計算”ボタンを押すごとに履歴が登録されます。

履歴に同じものがあれば１つだけが残されます。

CTRL+カーソルキーUP で履歴を１つ戻ります。

CTRL+カーソルキーDOWN で履歴を１つ進めます。

ALT+カーソルキーUP でカーソル手前までの文字と一致する履歴まで１つ戻ります。

ALT+カーソルキーDOWN でカーソル手前までの文字と一致する履歴を１つ進めます。

CTRL+マウスホイール上で履歴を１つ戻ります。

CTRL+マウスホイール下で履歴を１つ進めます。

ALT+マウスホイール上でカーソル手前までの文字と一致する履歴まで１つ戻ります。

ALT+マウスホイール下でカーソル手前までの文字と一致する履歴まで１つ進めま

す。

履歴位置は”計算”ボタンを押すかキー入力で最後にリセットされます。
関数を前半一部入力して F1 キーを押すと後半が一致する候補が出ます。
カーソルか、上下キーで選択してしばらくするとヘルプが表示されます。
クリック、enter キーで選択した候補が入寮されます。

- 直接計算エリア
  コメント、計算式、結果からなる小計算エリアが複数個あり”直接計算”ボタンで個別に計算できます。
  起動時に再計算されます。
  右上の + のボタンにより表示、非表示を切り替えられます。
- 引数エリア
  計算式は LCS(LineCalcScript)のスクリプトとなり @1,@2…@10　まで引数エリアの値が渡され使用することができます。
  @1=100
  @2=200
  として計算式エリアに
  @1*@2
  と入力して”計算”ボタンを押すと結果に 20000 と表示されます。
  毎日変化していく残金などを記録しておけば直接計算で自動でその他項目を計算する場合などに使用します。
  右上の + のボタンにより表示、非表示を切り替えられます。

- 情報エリア
  LCS の print,lprint により任意の文字列を表示できます。
  株価や配当率現在の資産の状況を LCS でプログラムして表示させることができます。
- 結果エリア
  計算式エリアで入力した LCS のスクリプトの結果を表示します。
  結果エリアには数値しか表示されず、文字はできるだけ数値に変換して表示されます。

- “計算”ボタン
  計算式エリアの LCS を評価して結果エリアに出力します。

・　“直接計算”ボタン

直接計算内の LCS を評価して結果を直接計算内に表示します。

・　“中断”ボタン

LCS の実行を中断します。

・　“クリア”ボタン

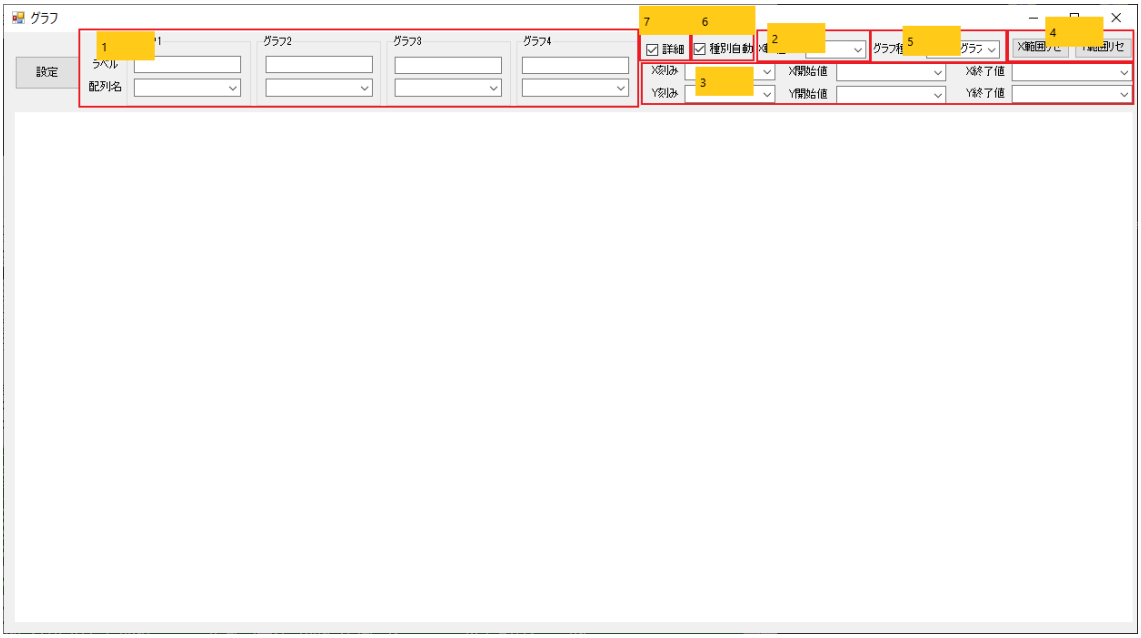
計算式エリアをクリアします。

・　“グラフ”ボタン

グラフボタンを押すと以下のようなダイアログが開きます。

グラフは配列に対して表示でき、配列のインデックスを X 軸、値を Y 軸としてグラフを描画します。

インデックスはソートされますが文字列の場合はソートされません。



・　①グラフソース設定エリア

配列名のドロップダウンリストでグラフ表示する配列を指定します。

複数設定すると同じグラフ上で色を変えて表示します。

グラフの●、▲などはマウスオーバー表示で具体的な値が表示されます。

・　②X 軸種類指定

X 軸の種類を数値か日付で選択します。

・　③軸範囲指定

X,Y 軸の軸の刻み、開始、終了値を設定します。空で自動計算。

X 軸が日付の場合 X 軸は日付指定に切り替わり、X 軸の刻みは設定不能になります。

日付の場合の X 軸の自動計算は開始値、終了値を同じ日付にします。

・　④範囲リセット

範囲をリセットします。

・　⑤グラフ種類

グラフ種類を設定します。

・　⑥種別自動

チェックが付いている場合 X 軸の種別を自動判定します。

・　⑦詳細

チェックが付いている場合マウスオーバーでポイントごとの詳細情報を表示します。

グラフ内ではマウス操作が可能です。

・グラフ内で左クリックドラッグ

　グラフを平行移動します

・グラフ内で右クリックドラッグ

左右方向で X 軸の拡大縮小、上下方向で Y 軸の拡大縮小を行います。

・グラフ内でホイール回転

マウスポインタの位置からグラフの全体拡大縮小を行います。

・グラフ X 軸ラベル表示エリアでの左クリック左右ドラッグ

X 軸移動を行います。

・グラフ Y 軸ラベル表示エリアでの左クリック上下ドラッグ

Y 軸移動を行います。

・グラフ X 軸ラベル表示エリアでの右クリック左右ドラッグ

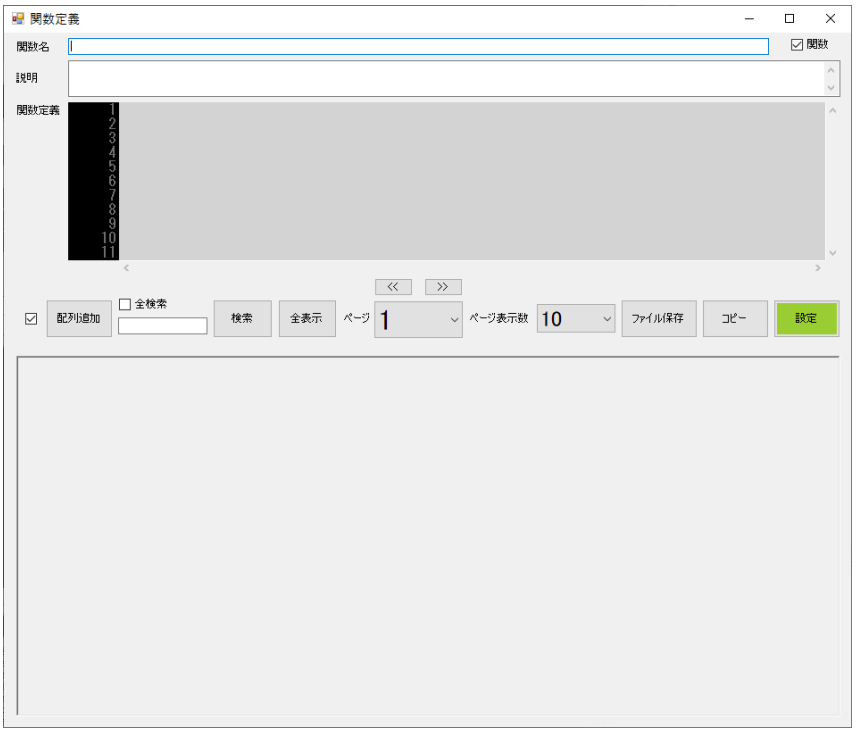
X 軸拡大縮小を行います。

・グラフ Y 軸ラベル表示エリアでの右クリック上下ドラッグ

Y 軸拡大縮小を行います。

・　“関数定義”ボタン

関数定義ダイアログを開きます。



上記のダイアログが開きます。
複数開くことができます。
関数を新規に作成する場合は、関数名を入力し、説明は任意で入力します。
関数定義に LCS で関数の定義を書き設定ボタンを押下します。
新規に配列を追加する場合は、配列追加ボタンを押して、配列名、説明、
配列のインデックス、値を入力して、設定ボタンを押します。
関数定義部分では右クリックメニューを表示できます。

- “設定”ボタン
  現在の関数名、説明、関数定義を設定します。
- “コピー” ボタン
  関数定義を計算エリアにコピーします

- “検索” ボタン
  文字列で関数を名前から検索します。正規表現が使えます
  全検索チェックボックスをチェックすると名前、説明、定義内容全てが検索対象になります。

- “全表示”ボタン
  検索文字列をクリアして全項目を表示します。

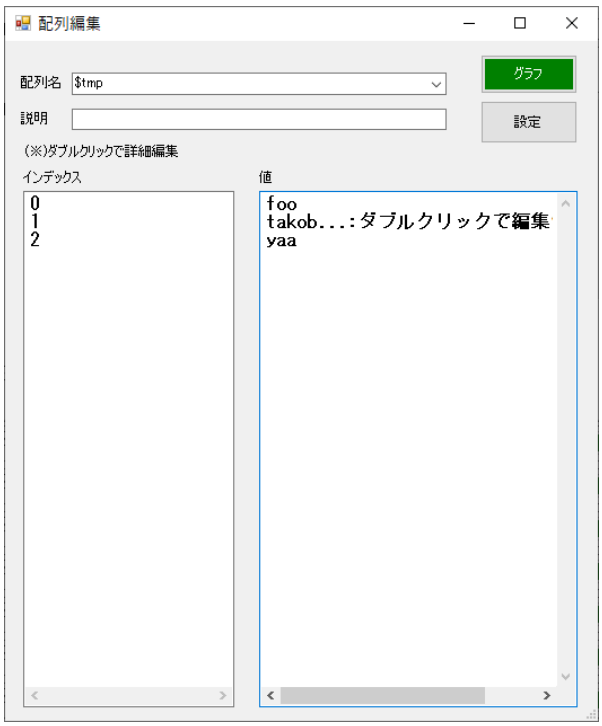
- “ページ”ドロップダウンリスト
  表示ページを指定します。
  フォーカスが当たった状態だとマウスホイールでページを切り替えられます。

- “ページ表示数”ドロップダウンリスト
  1 ページに表示する項目数を設定します。

- “<<”ボタン
  ページを 1 つ戻します。

- “>>”ボタン
  ページを 1 つ進めます。

・ “配列追加” ボタン
  新規に配列を追加します。
  押すと以下のダイアログが開きます。



  配列名、説明、インデックスは数値および任意文字列が設定できます(文字列は 100 文字まで改行は削除されます。指数表現数値および文字列はグラフ表示できません)。
  値は数値、文字列が使用できます。

設定を押すと現在の内容で設定します。
グラフボタンを押すと設定している配列名でグラフを表示します。
設定後でしかグラフは表示できません。
行をダブルクリックすることで詳細編集画面が開きます。
1000 文字以上または複数行の場合は省略表示となりますのでダブルクリックして詳細編集画面で編集してください。

・ “編集”ボタン
配列なら配列の設定画面、関数名なら内容を関数名、説明、関数定義にコピーします。
SHIFT キーと同時に押すと別ダイアログで開きます。

・ “⇈”ボタン
関数の位置を一番上に移動します。

・ “↑”ボタン
関数の位置を一つ上に移動します。

・ “⇊”ボタン
関数の位置を一番下に移動します。

・ “↓”ボタン
関数の位置を一つ下に移動します。

・ “削除”ボタン
関数、配列を消去します。

・ “ファイル保存”ボタン
チェックが付いている現在の検索結果の関数定義をファイルに保存します。

・ “関数”チェックボックス
チェックされていると関数として登録されます。チェックがされていない場合評価されず文字列が返されます。

・右クリックメニュー
関数定義ダイアログの関数定義テキストボックス、メインダイアログの計算式エ

リアでは右クリックメニューが表示できます。
文字を選択して、マウスの右クリックを押すとメニューが開きます。

・定義を別ウインドウで編集
選択中の文字列を新規関数定義ダイアログで検索して表示します。
変数が見つからない場合全検索の結果を出します。
・定義を全検索
洗濯中の文字列の全検索の結果を出します。
・補完
エディタ内でカーソル直前か選択中の文字の補完候補を表示します。候補は内部関数、グローバル変数、グローバル関数、グローバル配列、コンテキストリソース ID、バインドされたグローバル変数のコンテキストリソース ID です。候補をクリックするか、キーボードのカーソルで選択してエンターキーで入力できます。候補の上にマウスを置いておくかカーソルを動かさずしばらくすると説明が表示されます。
ドットアクセスの場合 ID がグローバル変数でコンテキストリソースにバインドされている場合のみコンテキストリソース内の候補を表示します。
$lib_util.(ここで補完)
上記では$lib_util にコンテキストリソース ID がバインドされていればコンテキストリソース内の候補が表示されます。
左端の ID がグローバルなら連続でドットアクセスが並んでいる場合は候補を表示します。全てコンテキストリソースにバインドされている必要があります。
$lib_util.disp.(ここで補完)
$lib_util.disp がコンテキストリソース ID にバインドされているなら補完候補を表示します。
コンテキストリソース ID を直接指定したドットアクセスも補完されます。
0(スペース). (ここで補完)
0 がコンテキストリソース ID の場合補完候補が表示されます。

・メニューエリア
・　設定
・　設定保存
設定保存では、現在の設定を終了時以外に任意のファイルで保存することができます。

・　設定読み込み(初期化)
設定読み込みでは、任意のファイルから設定を読み込んで復元します。

現在の設定は初期化されます。

・　設定読み込み

設定読み込みでは、任意のファイルから設定を読み込んで復元します。
現在の設定は初期化されず上書きされます。

・　リソース全削除

使われていないリソースが沢山出来てしまった場合など現在あるリソースを全て削除します。

・　IE 使用

選択するとチェックが入り、Web リソースに IE を使用します。
チェックが入らないと Edge(chromium ベース)を使用しますが、事前にインストールしておく必要があります。

・　,でスペースを入れる

選択するとチェックが入り、”,”を入力すると自動でスペースが入力されます。
CTRL を押しながらだとスペースは挿入されません。
チェックが入っていない場合は CTRL を押しながらだと,にスペースが挿入されます。

・WEB リソースキャッシュクリア

WEB リソースのキャッシュをクリアします。
Edge 版の WEB リソースではログフォルダ内にキャッシュファイルを作成します(環境は資産管理電卓個別です)。
このキャッシュファイルを削除します。
キャッシュが大きくなってきた場合や動作が不安定になった場合にはキャッシュをクリアすると動作が安定する場合があります。

・ワークフォルダ設定

WEB リソースのテンポラリファイルやロググローバル変数が保存される、ワークフォルダを設定します。
ワークフォルダを変更すると指定フォルダの環境で初期化されます。

・ユーザーエージェント設定

net , web(edge) 内部コマンドのデフォルトユーザーエージェント文字列を指定します。

“”(空)文字を指定するとシステムのデフォルトを使用します。
web 内部コマンドは使用するブラウザが edge の場合以外は反映されません。

・　ヘルプ
・　ライセンス認証
ライセンスの認証を行い機能制限を解除します。
改行などはいれずライセンス文字列を１行で入れ OK を押してください。

・　ログ表示
logprint で出力するシステムログ、その他システムのログを表示します。
.　ログフォルダ表示
ログが作成されているフォルダがエクスプローラーで開きます。
ログは大きくなると自動で分割されます。
・　バージョン情報
現在のバージョンライセンス関係の表示を行います。
・　リソース情報
　　リソース情報ダイアログを表示します。

・エディタコマンド
　計算式エリア、関数ダイアログ、配列編集ダイアログ内のエディタでは特殊キーにより各種コマンドを実行できます。

　Ctrl+F ： 検索ダイアログを開きます。検索文字列は正規表現を使います。
　Ctrl+S ： 関数定義ダイアログ、配列編集ダイアログで編集中データを保存します(設定ボタンを押すのと同等)。
　F3 ： 検索ダイアログで検索した文字列をさらに検索します。
　F1 ： 右クリックメニューの補完を呼び出します。

・ LCS(LineCalcScript)について

LCSは資産管理電卓(LineCalc)内で使用できるスクリプト言語です。
構文は簡単で式と関数しかありませんので構文に関してとても簡単になっています。
繰り返しや条件判断なども行う事ができます。
言語としての機能を実現する組み込み関数が定義されており組み込み関数群を使う事で複雑なプログラムが実現できます。

・ コメント

/* */ で囲まれた文字はコメントとなります。
例)

```
 /* -----
     コメント
 ------- */
```

// から改行までもコメントとなります。
例)

```
// コメント
```

・ 数値

数値は正規表現で
∞|[0-9][0-9,]*(¥.[0-9]+)?(e|E¥-?+?[0-9]+)?
上記で定義されます。
∞は無限大数値
+-{数値}e+-{指数数値}
数値には ,(コンマ)を含めることができます。
例)
123,456e10
+123e-10
123,456,789
-123,456
値の範囲は C# の double の範囲です。

値としては日付も値として扱います。
年(4桁)月(2桁)日(2桁)時(2桁)分(2桁)秒(2桁)
として、日付を表します。

例)

20190722161600
(2019 年 7 月 22 日 16 時 16 分 00 秒)

・ 文字列

文字列は”か’でくくられ、文字列そのものを表します

例)

“abcdef”

‘ああいいううええ’

¥はエスケープ修飾子で次に来る文字をそのまま使用することを示します。

¥n は改行に置き換わります(内部的には CR LF と 2 文字に置き換わります)。

¥t はタブに置き換わります。

文字列の中に改行をそのまま入れても問題ありません。

例)

“aaa¥”bbb”

‘ccc¥’ddd’

@を前につけると ¥ は無効になります(¥は¥)。

@を付けた場合“のなかでは””(“が２つ)が使え¥”と同じ意味になります。

‘のなかでは’’(‘が２つ)が使え¥’と同じ意味になります。

例)

@”aaa””bbb”

@’ccc’’ddd’

・ 数値と文字列の相互変換

LCS では型としては数値と文字列を持ちますが、相互に自動的に変換されます。

```
{
 a=”123$”,
 a*3
}
```

上記は a に文字列 “123$” を代入しています。そこに３をかけていますが、自動的に 123 と解釈され計算されます。

```
{
 a=123 ,
 print(a & “$”)
}
```

上記では a に数値 123 を代入しています。その後 “$” と文字列の連結を行って

いますが自動的に文字列”123”と解釈され 123$ が表示されます。

・ 演算子

演算子として

+, -, /, *, &, =, <, <=, >, >=, ==, !, &&, ||, ~, [], <-, =>

があります。

+:数値を足します。

1+2

-:数値を引きます。

1-2

*:数値を掛けます

2*3

/:数値を割ります

3/4

&:文字列を結合します。数値でも文字列に変換して結合しますが ‘,’ は削除されます。配列同士だと右側の 1 次元目を数値に置き換えて連結します。どちらかだけ配列の場合はその配列を返します。

“abc”&123,456

(結果)abc123456

=:変数を代入します

a=123

<:左辺より右辺が小さいか返します(1:真,0:偽)。文字列も比較できます。

a<123

<=:左辺より右辺が小さいか同じか返します(1:真,0:偽)。文字列も比較できます。

a<=123

>:左辺より右辺が多きいか返します(1:真,0:偽)。文字列も比較できます。

a>123

>=:左辺より右辺が大きいか同じか返します(1:真,0:偽)。文字列も比較できます。

a>=123

==:左辺と右辺が等しいか返します(1:真,0:偽)。文字列も比較できます。

a==123

!=:左辺と右辺が等しくないか返します(1:真,0:偽)。文字列も比較できます。

a!=123

<, <=, >, >=, ==, != は後ろに “&” をつけると文字列比較となります。

つけないと文字列か数値かを自動判定します。

Ex)
　“150.50a” ==& 150.5
0 が返ります。
　“150.50a” == 150.5
1 が返ります。

!:与えられた式の真偽を逆にします(1:真,0:偽)。
　!(a<123)
||:論理和を返します。どれか正になれば残りは評価しません。
　a<123 || b<123 || c<123
&&:論理積を返します。どれか偽になれば残りは評価しません。
　a<123 && b<123 && c<123
~:右に指定した式の値を評価します。eval と同等ですが関数内変数にアクセス、作成可能です。
　{ tmp = 2 , ~"{foo=tmp+2}",lprint(foo) }
[]:配列初期化文字列を作成します。[{キー値::}式,…] で配列を作成できます。
 tmp = [1 , 2, 3]
<-:文字列評価を行います。文字列 <- (引数１、引数２,…) と記載すると文字列を関数として引数で呼び出します。eval(文字列,引数１,引数 2,…) と同等です。
　“@1*@2” <- (2, 5)
=>:任意の LCS から無名関数を作成します。
無名関数は変数に代入すると変数が関数として定義されます。
文字列で関数を渡し <- や eval で明示的に評価するより、構文解析が行われるので構文エラーが発見されやすくなります。
また、=> {} の中では定義元の関数内の変数にアクセス可能になります(クロージャ)。
定義元変数は削除できません。
 例)
sort(tmp,=>{if(@1 < @2 ,-1 ,if(@1 > @2,1 , 0)) })
{tes = =>{@1,@2},tes(2 , 3)}
=>{@1*@2}(2, 3)
{ tmp = 1 ,tes = => {tmp} , tes } // 定義元変数 tmp の 1 が表示される

演算子の優先順位は
() ,~,[],<-,=>
*,/

+,-,&
<,>,<=,>=,==,!=,!
||,&&
=
となっています。
下に行くほど優先順位が低くなります。

式の値としては数値か文字列を持つことができ、配列そのものは文字列に変換された値となります。

・ シーケンス

連続して式を実行する場合 { と }で数式を ‘,’ か ‘:’で区切り並べると先頭から順次実行されます。

ex)

```
{
 sum=0 ,
 for(cnt , 1 , 100 , 1 ,
  sum = sum + cnt
 )
}
```

（注）

{…} は 互換性を維持するため (…) seq(…) でも記述できます。

・ 変数と関数

LCS では変数を定義できます。
変数は値を入れると定義され、宣言は必要ありません。
値は数値か、文字列です。

変数名=値
として、変数名に値を保持することができます。
例）

```
{
 tmp=0 ,
 print(“値:”,tmp)
 }
```

値:0 が表示されます。

特殊変数として引数があり、

@1,@2,…

として定義されます。
これはこれは関数呼び出しの引数を表します。

変数のほかに関数があり、

fnc（関数名、関数定義）
def（関数名、関数定義文字列）

として定義するか、関数定義ダイアログから定義します。
関数は評価した値を返します。

変数と何が違うかというと関数は中身が評価(計算)されます。
変数は中身は評価されず文字列で “1+1” と定義すると “1+1” の文字列が返されますが、関数では計算され 2 が帰ります。
関数の呼び出しは以下のようにし、引数を渡すことができます。

関数名(引数１,引数 2,引数 3,…)

引数は関数内で @1,@2,…として使用できます。
引数の数は任意となります。
任意の位置で引数を指定しない場合スキップすることができます（内部関数は任意の位置の引数スキップはできません)。

例)
{
fnc(foo, @1+@2),
foo(1 ,2)
}

3 が表示されます。

```
{
 fnc(tes, print(isdef(@1) & isdef(@2))),
 tes(,1)
}
```
 01 が表示されます。

関数の引数の区切りには , を使いますが、数値の区切りとしても , は使用します。
ですので、上記では foo(1 , 2) と 1 と, の間にスペースを入れて関数の区切りの , を明示しないといけません、これを忘れてスペースを忘れるとエラーとなります。
代替えとして、関数の区切りには : も使用できます。
例)
```
 foo(1:2)
```

違和感がなければ、上記の記法も可能です。

関数は資産管理で使用する株価、配当等を記録するのに使う基本構造です。
各種資産の記録は全て関数内に定義していくことになります。

関数呼び出しの中で関数、変数を定義すると関数中だけで有効になります(ローカル変数、関数)。
関数呼び出しを行うと関数、変数定義領域が作成されます。
この関数呼び出し時に作られる関数、変数の定義領域の事をコンテキストと定義します。
どのような関数名、変数名を使っても他の関数からは見えませんし、影響を与えません。関数が終了すると自動で消去されます。
但し、コンテキストリソース内以外では内部関数名は関数名として使用できません。

$変数名,$関数名
＄を先頭につけるとグローバル変数、関数として登録され、どの関数からも見えるようになり、関数定義ダイアログに現れるようになります。
また、終了時にグローバル変数、関数は自動保存されます。
ファイルとして保存され、起動時に読み込まれますので作成するグローバル変数、配列、関数の数に注意してください。

例）
foo=1 として計算、クリアを押して
foo とすると foo は定義されておらず 0 と出る

$foo=1 として計算、クリアして
$foo とすると 1 が表示される

アプリを終了させて、立ち上げなおし $foo とすると 1 が表示される。

・ 配列

配列は数字の添え字（インデックス）でアクセスできる変数、関数です。
宣言は必要なく、値を入れると作成されます。
数値と文字列を入れることができます。
配列を代入するとコピーされます。
配列の要素は添字を含んだ文字として扱われる１変数として扱われています。
ford 等の繰り返し指定で使うことが可能です。

配列変数は
配列変数名[添え字]… = 値
または
配列変数名= [{インデックス::}値 , …]
で定義します。
[式,…] の形式は配列を一度に初期化することができます。
例）
tmp = [1 , 2 , 3]
tmp = [“adr”::1 , “age”::80]
配列のインデックスを :: の左側に書くことで指定できます。
指定しない場合は数値が自動で割り当てられます。
値としては配列変数も指定することができます。

[添え字]は複数指定することもできます。
foo[2][3]=2
値を呼び出すには、
配列変数名[添え字]
とします。
[添え字]は複数指定することもできます。

例）

```
{
 foo[2]=3 ,
 foo[“yaa”]=4 ,
 foo[“yaa”][2]=4 ,
 foo[2],
 foo[“yaa”][2]
}
```

添え字は数値および任意の文字列が指定できますが、文字列は 100 文字まで、指数表現及び文字列はグラフ表示できません。

```
{
 tmp[0] = 1,
 tmp[0][0] = 1,
}
```

とすると配列 tmp にインデックスが一次元と 2 次元の要素を作ることができます。

あまり使わないですが、配列の関数も定義できます。
fnc（配列関数名[添え字]…,関数定義）
def(配列関数名[添え字]…,関数定義文字列)
で定義します。
配列関数を呼び出すには
配列関数名[添え字]…(引数,…)
とします。
配列関数呼び出しでは、配列関数の中身は評価され返されます。
ex)

```
{
 fnc(fn [0], @1*2),
 fnc(fn [1], @1*4),
 sum=0 ,
 fordim(tag,fn,sum=sum+fn[tag](100))
}
```

配列は引数として渡すこともできます。
例）

```
{
 fnc(tes,
```

```
  {
   print(@1[1]) // 引数として渡された配列にアクセス
  }
 ),
 tmp = [ 1 , 2 , 3],
 tes(tmp) // 配列を引数として渡す
}
```

配列名[::式] とすることでインデックスではなく配列内の位置で配列にアクセスすることが出来ます。

Ex)

```
{
 tmp = ["foo"::1, "taa"::2, "yaa"::3],
 print(tmp[::1]) // 2 番目の値を表示する
}
```

2 が表示されます。

sort 内部関数などではインデックスではつけ変わらず位置によりソートされますのでソートの結果を得る場合などに使用できます。

位置は０が先頭になります。

マイナスの値を指定すると-1 で最後尾となり最後から逆位置を指定できます。

:: は多次元配列に使うとその次元内の全ての要素の位置を指定することになります。

Ex)

```
{
 tmp = [[1, 2, 3], ["foo"::1, "taa"::2, "yaa"::3]],
 print(tmp[::4]) // tmp 内の 5 番目の値を表示する
}
```

tmp 内の全ての要素の並びの５番目 2 が表示されます。

::は配列指定の中で１回しか出現できません。

```
{
 tmp = [[1, 2, 3], ["foo"::1, "taa"::2, "yaa"::3]],
 print(tmp[1][::1]) // tmp[1] 内の 2 番目の値を表示する
}
```

2 が表示されます。

上記のように部分配列内の要素位置を指定することも出来ます。

追記)
配列は関数呼び出しでは値としてコピーされ渡されます。
演算子 = の右側に配列の結合 & を書いた場合は配列はいったん文字列に変換されます。
この場合速度が遅くなりメモリも消費します。
1000 以上の要素を持つなどの大きな配列結合には appenddim を使用するようにしてください。
配列を返す場合はいったん文字列に変換されるので大きな配列を返すと速度が低下し、メモリを消費します。
大きな配列を関数の値として返す場合は、コンテキストリソースを使うか無名関数を使うと高速化できます。

配列を初期化する場合内部に配列を指定することもできます。
例)

```
{
 tmp = [1 ,2]
 tmp2 = [tmp , tmp]
}
```

上記では、tmp2 は tmp2[0][0] = 1 , tmp2[0][1] = 2 , tmp2[1][0] = 1 , tmp2[1][1] = 2 と設定されます。

- リソース

グラフ、並列プロセスなどリソースとして実態が管理されます。
リソースは作成されると独自に動作し始め、作成関数は直ぐに帰ってきます。
その時、リソース ID を返します。

```
id=grph(tmp) /* id にグラフリソースのリソース ID を返します */
```

この ID を使ってリソースを操作できます。
リソースは、種別を持ち reskind 内部関数で取得できます。

“GRPH” : グラフリソース
“PRC” : プロセスリソース
“CNT” : コンテキストリソース
“WEB” : WEB リソース
“STR” : ストリングバッファリソース

解放されたリソース ID では 0 が返ります。
LineCalc が起動されてからリソース ID は 1 つずつ増やされて割り当てられます。
リソース ID は 2147483647 まで割り当てられそれ以上作ると 0 に戻ります。

delres でリソースを削除できます。
bindres でバインドしておけば変数消去時に削除されます。
context(コンテキストリソース),stringbuffer(ストリングバッファリソース)は = の代入だけでも変数にバインドされます。

リソース削除には明示的な処理が必要ですが変数にバインドしておけばガベージコレクタの対象になります。

中断すると実行で作成されたリソースは全て削除されます。

関数内で作成されたリソースは関数が正常終了して削除されない状態だとそのまま残ったまま関数から帰ってきます。
グラフ、WEB、PRC リソースはウインドウを閉じる、実行が完了すると削除されます。

同一関数呼び出し内で作成されたリソースは waitresall で一括で終了を待つことができ,delresall で一括で削除できます。

```
{
 prc( ‘lptint(net(^html,″ http://google.com″)’), /* html 取得 */
 prc( ‘lprint(net(^html,″ http://yahoo.com″)’), /* html 取得 */
 waitresall /* 取得完了まで待ちます */
}
```

waitresall は引数でどの種類のリソースを待つか指定できます。

```
waitresall(“PRC”)
```

waitresall で対象になるリソースは同一関数呼び出し内で作られたリソースのみです。
別関数呼び出し内で作られたリソース、prc コマンド内で作られたリソースは対象になりません。

リソースは reskind で 0 が返れば終了しています。

プロセスリソースは並列に実行され、html の取得など時間がかかる処理を並列で行う事ができます。

prc の終了値を得るには無名関数を使用して変数に値を返すか、コンテキストを渡してその中に値を返すこともできます。
ex)
```
{
 ret = 0, // 帰り値用変数定義
 prc(=>{ ret = 123 }),
 waitresall,
 print(ret)
}

{
 ctx = context,
 prc(“@1.ret = 123”,ctx),
 waitresall(“prc”),
 print(ctx.ret)
}
```

グラフリソースはグラフ表示リソースです。
結果の値は０です。

コンテキストリソースはコンテキスト(関数、変数領域)を作成します。
コンテキストは変数、関数を定義しておけるエリアの事です。
コンテキストリソースを入れた変数名.(ドット)変数名(関数名) でアクセスします。
コンテキストリソースは明示的に消去する必要があります。
コンテキスト内関数では this で関数定義のあるコンテキストリソース ID を示します。
コンテキスト内の変数、関数からクラスを作成することができます。
明示的に消去しない限り残るのでグローバル変数にリソース ID を残しておくとグローバルクラスとしてどこからでも使用できます。
グローバル変数に設定しておいてもコンテキストはアプリ終了時に保存されません

(コンテキスト ID として数値が設定されている為)。
グローバル変数に全体で使用するコンテキストを設定して内部に共通関数を定義しておけば、個別で定義する必要が無くなり便利です。
この場合は初期化関数 $sys_init 内にコンテキストを作成してグローバル変数に登録する LCS を記載しておきます。
コンテキストにまとめず、グローバル関数群として定義することもできます。

WEB リソースは WEB ページを表示します。
URL を指定する以外に HTML を直接設定することもできます。

ストリングバッファリソースは巨大な文字列に高速に追加できるリソースです。
& で文字列追加を行うと大きくなると遅くなっていきます。
stringbuffer で作成したストリングバッファリソースに追加していくと高速に追加できます。
ストリングバッファリソースを文字列に戻すには rdump を使います。

・リソースの寿命について
グラフ、WEB リソースは消去操作されるか bindres や delres で消されると消滅します。
プロセスリソースは実行が終了すると消滅します。
ストリングバッファ、コンテキストリソースは明示的に消去しない限り残り続けます。
変数に代入するか、bindres を使うと関数が終了して変数が消去されるのと同時にリソースが削除されます。
以下では bindres を明示していますが context,stringbuffer は=代入だけでもバインドされます。
例)

```
{
 bindres(cnt,context), /* コンテキストを作成します(変数 cnt にバインドして関数を出ると削除) */
 // ctx = context, // これでも OK
 cnt.tmp=2, /* コンテキスト内に変数を作成します */
 fnc(cnt.foo, @1*this.tmp), /* コンテキスト内に関数を作成します */
 cnt.foo (123) /* コンテキスト内関数を呼び出します */
}
```

246 が表示されます。

bindres した時点で変数が既に bindres されている場合、変更前のリソースのバインドが解除後新たしいリソースにバインドされます。
バインドされた変数を他の変数に代入すると代入された変数はリソースにバインドされます。
例)

```
{
 bindres(ctx, context),
 ctx2 = ctx // ctx2 は ctx のリソースにバインド
}
```

関数の帰り値としてバインドしたコンテキスト変数を返すと、呼び出し元では帰ってきたリソースの破棄は必要なくなります。

```
{
 fnc(tes, {bindres(ctx, context), ctx.foo = 123, ctx}),  // context は ctx にバインドされ帰り値として返される
 ctx2 = tes,  // ctx2 は帰り値のバインドごとコピーされリソースにバインドされる
 lprint(ctx2.foo),
 lprint(tes.foo) // これでも OK
}
```

bindres で一度変数にバインドしておけば参照がなくなればガベージコレクタにより破棄されるためリソース破棄管理は必要なくなります。

(注記)
.(ドット演算子)は左の式の値をコンテキストリソース ID として、右の値が変数名、関数名ならそのコンテキストリソース内で評価します。
リソースは複数の変数にバインドできます。
バインドされた変数が全て削除された後リソースは削除されます。

```
{
 bindres (ctx, context),
 ctx.yaa = 123,
 fnc(tes, { bindres (ctx2, @1), lprint(ctx2.yaa) }),
 tes(ctx)
}
```

上記では関数 tes に渡したコンテキストリソース ID を tes 内でさらにバインドしています。
tes が終了した時点で tes の ctx2 変数は消去されバインドは解除されますが、もう一つバインドされているためリソースは削除されず変数 ctx が削除された後に削除されます。

```
{
 bindres (ctx, context),
 ctx.yaa = 123,
 fnc(tes, { bindres (ctx2, @1), prc(=>{delay(1000), lprint(ctx2.yaa)})}),
 tes(ctx)
}
```

上記では関数 tes 内で ctx2 にリソースがバインドされ prc 内の無名関数 =>{} に関数 tes の変数 ctx2 が拘束されます。prc 内で ctx2 を使用後 ctx2 が削除されリソースが削除されます。

bindres は１つのリソースを変数にバインドします。
コンテキストリソース内で参照するコンテキストリソースはバインドしておく必要があります。
例）

```
{
 bindres(ctx, context),
 bindres(ctx.ctx2, context),
 bindres(ctx.ctx2.ctx3, ctx),
}
```

上記では自己参照によるループが発生していますが、ガベージコレクタにより削除されリソースリークは発生しません。
説明のため bindres を明記しましたが、context,stringbuffer に関しては代入だけで済み、ガベージコレクタが消去するので bindres や delres によるリソース管理を気にする必要はありません。
例）

```
{
 ctx = context,
 ctx.ctx2 = context,
 ctx.ctx2.ctx3 = ctx
}
```

```
{
 ctx = context,
 ctx.yaa = 123,
 fnc(tes, { ctx2 = @1, prc(=>{delay(1000), lprint(ctx2.yaa)})}),
 tes(ctx)
}
```

上記で OK です。

- クラス作成

コンテキストを利用してクラスを作成できます。

```
{
 // クラス Tes のコンストラクタ
 fnc(newTes,
  {
   cls = context,
   cls.yaa = 123, // メンバ変数定義
   fnc(cls.foo, this.yaa), // メンバ関数定義
   cls // 作成コンテキストを返す
  }
 ),
 te = newTes, // クラス Tes を作成
 te.foo // メンバ関数 foo を呼び出し
}
```

cls.foo 内で定義元コンテキストをアクセスするため this を使用しています。

継承はベースコンテキストを作成して後で加工すればできます。

```
{
 mctx = context,

 // クラス Tes のコンストラクタ
 fnc(mctx.newTes,
  {
   cls = context,
```

```
   cls.yaa = 123, // メンバ変数定義
   fnc(cls.foo, this.yaa), // メンバ関数定義
   cls // 作成コンテキストを返す
  }
 ),

 // クラス Tes を継承
 fnc(mctx.newTes2,
  {
   cls = this.newTes,
   cls.taa = 321, // メンバ変数定義
   fnc(cls.foo2, this.foo * this.taa), // メンバ関数定義
   cls // 作成コンテキストを返す
  }
 ),

 te = mctx.newTes2, // クラス Tes2 を作成
 te.foo2 // メンバ関数 foo2 を呼び出し
}
```

上記では newTes を参照するために mctx に new 関数をまとめるコンテキストを作成しています。

上記の記載を簡潔にするための内部関数 class が利用できます。

```
{
 // クラス Tes のコンストラクタ
 class(Tes,
  {
   cls.yaa = 123, // メンバ変数定義
   fnc(cls.foo, this.yaa), // メンバ関数定義
  }
 ),
 te = newTes, // クラス Tes を作成
 te.foo // メンバ関数 foo を呼び出し
}
```

```
{
 mctx = context,

 // クラス Tes のコンストラクタ
 class(mctx.Tes,
  {
   cls.yaa = 123, // メンバ変数定義
   fnc(cls.foo, this.yaa), // メンバ関数定義
  }
 ),

 // クラス Tes を継承
 class (mctx.Tes2,
  {
   cls = this.newTes, // 継承元作成(this で mctx にアクセス)
   cls.taa = 321, // メンバ変数定義
   fnc(cls.foo2, this.foo * this.taa), // メンバ関数定義
  }
 ),

 te = mctx.newTes2, // クラス Tes2 を作成
 te.foo2 // メンバ関数 foo2 を呼び出し
}
```

上記はコンテキスト mctx を使わないと継承元クラスにアクセスできないので統括クラスを導入すると、

```
{
 class(CMain, // 統括クラス
  {
   // クラス Tes のコンストラクタ
   class(cls.Tes,
    {
     cls.yaa = 123, // メンバ変数定義
     fnc(cls.foo, this.yaa), // メンバ関数定義
```

```
    }
   ),

   // クラス Tes を継承
   class (cls.Tes2,
    {
     cls = this.newTes, // 継承元作成(this で mctx にアクセス)
     cls.taa = 321, // メンバ変数定義
     fnc(cls.foo2, this.foo * this.taa), // メンバ関数定義
    }
   ),
   fnc(cls.main, // メイン関数
    {
     te = this.newTes2, // クラス Tes2 を作成
     te.foo2 // メンバ関数 foo2 を呼び出し
    }
   )
  }
 ),
 ma = newCMain, // 統括クラス作成
 ma.main // メイン関数呼び出し
}
```

上記となります。

クラスメンバの定義には cls.メンバ名 を使用してください。
クラス定義関数の ID には”new”が自動で追加されます。

- 動的 ID 生成

変数、関数の ID を()でくくった式から動的に作成することができます。

```
ex)
{
 a=”foo”,
  (a & “yaa”) = 2, /* 括弧内の式から代入する変数名 fooyaa を作成 */
 print(fooyaa) /* 2 が表示*/
}
```

上記では、変数 a に “foo” という文字列を入れ、その文字列に “yaa”を追加

して fooyaa という変数名を作成し値を代入しています。
コンテキストリソースに対しても可能です。
ex)
```
{
 ctx = context,
 a = "foo",
 ctx.(a & "yaa") = 2 , /* ctx.fooyaa に 2 を代入 */
 print(ctx.(a & "yaa")) /* ctx.fooyaa をアクセス */
}
```
上記ではコンテキストリソースのメンバ名を動的に文字列から作りアクセスしています。
動的 ID の後ろには配列指定も可能です。
ex)
```
{
 ctx = context,
 a = "foo",
 ctx.(a & "yaa")[1][2] = 2 ,
 print(ctx.(a & "yaa")[1][2])
}
```
また、最後にパラメータリストを配置すると関数呼び出しが可能です。
ex)
```
{
 ctx = context,
 a = "foo",
 def(ctx.(a & "yaa")," @1+@2") ,
 print(ctx.(a & "yaa")(1 , 2))
}
```
コンテキスト内(.(ドット)連結 ID)では上記のように生成文字は評価されますが、コンテキスト外(.(ドット) 連結していない単体 ID)は () を後ろにつけない限り評価されません。
ex)
```
{
 ctx = context,
 a = "foo",
 def((a & "yaa"),"@1+@2") ,
 print((a & "yaa")), /* fooyaa という文字列が表示 */
```

```
 lprint((a & "yaa")(1 , 2)), /* fooyaa(1 , 2) が評価され 3 が表示 */

 (a & "taa") = 2 ,
 lprint((a & "taa")), /* footaa という文字列が表示 */
 lprint((a & "taa")()), /* footaa() が評価され 2 が表示 */

 def(ctx.(a & "yaa"),"2+3"),
 lprint(ctx.(a & "yaa")), /* fooyaa が評価され 5 が表示 */
 lprint(ctx.(a & "yaa")()), /* fooyaa が評価され 5 が表示 */

}
```

上記では明示的にパラメータが指定されたので関数が評価されます。変数名でも()を書くと評価されます。コンテキスト内の ID はパラメータが指定されなくても評価されています。

isdef 等の ID 指定としても使用可能です。

```
ex)
{
 ctx = context,
 a = "foo",
 (a & "yaa") = 1 , /* fooyaa に 1 を代入 */
 print(isdef((a & "yaa"))), /* 1 が返る */

 ctx.(a & "yaa") = 1 , /* ctx.fooyaa に 1 を代入 */
 lprint(isdef(ctx.(a & "yaa"))) /* 1 が返る */
}
```

動的 ID の無名関数を得るには # は使えませんので、lambda を使用してください。

```
ex)
{
 fnc(("tmp"), lprint(123)),
 fn = lambda(("tmp")),
 fn
}
```

- 評価修飾子

変数名、関数名は始めに@か英字(a-zA-Z_)そのあとに数値か英字(a-zA-Z_)を続けら

れます。
例）
A123abc
@1

変数名、関数名の前には ^ か # の修飾文字を指定することができます。
^ を指定すると関数名をそのまま文字列として返します。
例）
```
 print(^xom)
```

xom が表示されます。
“” の短縮形になります。
コンテキストリソース内の指定はできません。
例）
```
 ^ctx.foo
```

エラーになります。
ctx.^foo とするとコンテキストリソース内に ^foo という関数にアクセスします。

#を前につけると関数の場合中身を評価せず定義内容の無名関数を返します。

例）
```
{
 fnc (foo, 1+1),
 print(#foo)
}
```
foo の無名関数を表す文字列が表示されます。
無名関数は変数に代入することで関数を定義できます。
例）
```
{
 fnc(tes,1+1),
 fnc2 = #tes
}
```
fnc2 は tes と同じように使用できます。
関数定義した変数を関数の帰り値として返す場合や引数として渡す場合に使用します。

^ と # の後ろにはスペースは入れられません。
コンテキストリソース内では先頭に指定します。
例)

```
 {
  ctx = context,
  fnc(ctx.foo, 1+1),
  print(#ctx.foo)
 }
```

1+1 の関数定義が表示されます。
ctx.#foo とするとコンテキストリソース内の #foo という関数にアクセスすることになります。
コンテキストリソース内の関数内で this が使われていた場合、無名関数を他の変数に代入しても this は元のコンテキストリソース ID を示します。

```
ex)
{
 cnt = context,
 cnt2 = context,
 cnt.yaa = 123,
 fnc(cnt.foo, {this.yaa}), // cnt 内の関数定義

 fnc(cnt2.setcallbk, this.cb = #@1),  // cnt2 にコールバック関数を設定する
 fnc(cnt2.callbk, this.cb),  // cnt2 に設定されたコールバック関数の呼び出し

 cnt2.setcallbk(#cnt.foo), // cnt 内の foo をコールバックに設定(this 参照)
 print(cnt2.callbk), // コールバック呼び出し this は cnt を示す
 f = #cnt.foo, // f に cnt.foo から関数を定義
 lprint(f & "¥n"), // f 内の this は cnt を示す
}
```

・文字列評価
~式
を使用して文字列を評価できます。

例)

```
{
 numvar1 = 1,
 numvar2 = 2,
 numvar3 = 3,
 sum = 0,
 for(cnt , 1, 3 , 1 , sum = sum + ~("numvar" & cnt))
}
```

number1,number2,number3 の文字列を評価して合計しています。

eval との違いは eval では式の評価はローカル環境が新たに作成して行われますが、~ ではローカル環境内で行われますので関数内の変数を作成、変更可能です。

<- (式,…)

上記を使って文字列を eval 評価できます。

引数の区切りは , か : が使用できます。

“123+456”<- ()

()には引数を記載できます。

“@1*@2”<- (2 , 3) // 6 が表示

・無名関数

=>{式,…}

を使用して無名関数を作成することができます(無名関数を表す文字列を作成します)。

式の区切りは , か : が使用できます。

無名関数は関数定義名を持たない関数定義の事で変数に代入されると変数に関数が定義され、評価されると関数が実行されます。

```
=> {@1 * @2} (2 , 3) // 6 が表示

{
 tes = => {@1 * @2},
 tes(2, 3) // 6 が表示
}
```

無名関数は引数として関数を渡したり、関数を帰り値として返す場合に使用できます。

```
{
 fnc(tes,@1(2 , 3)), // 引数として渡された関数を実行
 tes(=>{@1*@2}) // 6 が表示
}
```

```
{
 fnc(tes,=>{@1*@2}), // 関数を返す
 tes2 = tes, // 返された関数を tes2 に定義
 tes2(2 , 3) // 6 が表示
}
```
変数に定義された関数は # を付けることで無名関数を得る事が出来ます。

```
{
 fnc(tes,
  {
   fnc(tes,@1*@2),
   #tes // tes の無名関数を得て返す
  }
 ),
 tes2 = tes, // 返された無名関数を tes2 に定義
 tes2(2 , 3) // 6 が表示される
}
```
=> の中では定義元の変数にアクセス可能ですが作成、削除はできません。
```
{
 tmp = 123,
 => {lprint(tmp),tmp=1, tmp2 = 123,lprint(tmp)}(), // tmp=1 は外部参照の tmp が変更される,tmp2 は無名関数内のローカルコンテキストに作成される
 lprint(tmp), // 無名関数内で変更され 1 が表示
 lprint(tmp2) // エラー
}
```
12311 と表示されますが、tmp2 は無名関数内で作成され外部では参照できない為エラーになります。
```
{
 tmp = [0],
 => { tmp[0] = 100,appenddim([101 , 102, 103],tmp),tmp2 = [1 ,2] }(),// tmp は外部参照追加、tmp2 はローカルで作成される
 ford(tag,tmp,lprint(tmp[tag])), // 無名関数内で変更追加された値が表示
 lprint(tmp2[0]) // エラー
}
```
100101102103 と表示されます。

配列は一部だけ外部で作成しておけば無名関数内でも変更、追加が可能です。
引数 @数値 は外部参照できません。

```
{
 fnc(tes,
  {
   lprint(@2),
   =>{lprint(@1),@2}(4) // 4 が表示され,@2 は未定義エラー
  }
 ),
 tes(1, 2)
}
```

24 と表示され、@2 は未定義エラーになります。
引数の値として無名関数を渡した場合でも定義元の変数にアクセス可能です。

```
{
 fnc(tes,@1(2 , 3)),
 tmp = 4,
 tes(=>{ tmp * @1 * @2}) // 定義された環境内の変数は評価先によらずアクセス可能
}
```

def と組み合わせると外部変数にアクセス可能な関数が作成できます。
ex)

```
{
 tmp = 123,
 def(tes, =>{tmp}),  // 外部変数にアクセス
 tes
}
```

tes は無名関数で定義され外部の変数にアクセス可能になります。

= でも可能です。

```
{
 tmp = 123,
 tes = =>{tmp},  // 外部変数にアクセス
 tes
}
```

```
{
 tmp = 123,
```

```
 def(tes,
  =>{
   foo = 456,
   def(tes2,
    =>{
     tmp+foo // 外部変数にアクセス
    }
   ),
   tes2
  }
 ),
 tes
}
```

上記のように関数定義の中に更に関数定義を無名関数で登録すると外側の変数 foo,tmp にアクセスでき、579 が帰ります。

・システム関数

　システム定義のグローバル関数があり、特定の名前のグローバル関数を定義すると特別な意味を持ちます。

・$sys_init

初期化システム関数です。

起動時に実行される LCS を記載します。

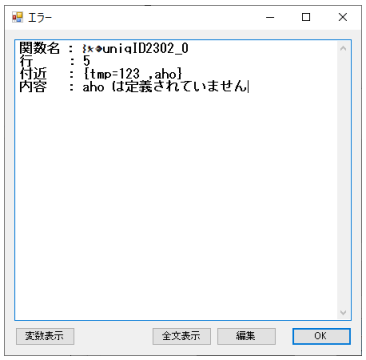
・$sys_final

終了システム関数です。

終了時に実行する LCS を記載します。

・エラーダイアログ

エラーが発生すると以下のようなダイアログが開きます。



エラー:
 エラーの種類を示します。
関数名 :
 発生関数名を表示します。動的に定義された関数の場合どの関数内で定義されたかも表示されます。
行:
 定義された関数内の何行目でのエラーか表示します。
付近:
 エラーが発生した付近のコードを表示します。
内容 :
 エラーの具体的な内容を表示します。

編集ボタンが表示されている場合は押すとエラー箇所の行の関数定義ダイアログでの編集画面が表示されます。
メイン画面での編集の場合はメイン画面の行がジャンプします。
関数定義を文字列で行っている場合変数などで関数定義を合成している場合は行表示は不完全になり編集ボタンでも正確な位置にはジャンプしません。

全文表示ボタンを押すと文字列を合成して実行している場合等ソースがない場合も評価中文字列全体を表示するダイアログを開きます。
エラー箇所にカーソルを移動して表示します。
スクロールなどした場合でエラー箇所をまた見たい場合は、行を戻すボタンでエラー箇所にカーソルを移動します。

変数表示ボタンを押すとエラーが起こった場所でのデバッグダイアログが開きます。
変数の内容が確認できます。

・ 内部関数

制御構造などは内部関数として実装されています。

LCS は全ての制御構文を関数として実装している完全関数型言語です。

構文は関数呼び出しのみとなり構文自体は非常に単純で、内部関数を使えるようになればプログラム可能です。

・ 内部関数の定義

以下

式は任意の LCS を示し、それ以外は明示します。

[] は省略可能を示します。

…は任意の繰り返しを示します。

関数名:

let

説明:

（非推奨 互換性維持関数）

変数に値を代入します。$を先頭に付けるとグローバル変数になります。

グローバル変数は終了時ファイルに自動保存され起動時に復帰します。

= と同じです。

引数:

let(ID(変数名),式(設定値))

例:

let(foo,100) /* ローカル変数 foo に 100 を設定 */

let($foo,100) /* グローバル変数$foo に 100 を設定 */

関数名:

def

説明:

関数を文字列から定義します。$を先頭に付けるとグローバル関数になります。

グローバル関数は終了時ファイルに自動保存され起動時に復帰します。

引数:

def(ID(関数名), 式(設定値))

例:

def(foo, “1+1 “) /* ローカル関数 foo を定義 */

def($foo, “1+1 “) /* グローバル関数 $foo を定義 */

関数名:

letdim

説明:

（非推奨 互換性維持関数）

配列変数に値を設定します。$を先頭に付けるとグローバル変数になります。

グローバル変数は終了時ファイルに自動保存され起動時に復帰します。

[] と = に同等です。

引数:

letdim(ID(配列変数名), 式(数値添え字), 式(設定値))

例:

```
letdim(foo,1 ,100) /* 配列変数 foo に添え字 1 で値 100 を設定 */
letdim($foo,1 ,100) /* グローバル配列変数$foo に添え字 1 で値 100 を設定
*/
```

関数名:

defdim

説明:

（非推奨 互換性維持関数）

配列関数を文字列から定義します。

$を先頭に付けるとグローバル変数になります。

グローバル関数は終了時ファイルに自動保存され起動時に復帰します。

引数:

defdim(ID(配列関数名), 式(数値添え字),式(定義値))

例:

```
defdim(foo,1 ,”1+1”) /* 配列関数 foo に値 1+1 を設定します */
defdim($foo,1 ,”1+1”) /* グローバル配列関数 $foo に値 1+1 を設定します
*/
```

関数名:

if

説明:

条件判定して処理を分岐します。

条件は数値で判定し 0 より上で真(true)、以外は偽(false)です。

引数:

if(式(条件),式(真時実行)[,式(偽時実行)])

例:

```
if(tmp>1 , /* 条件 tmp が 1 より上 */
 /* true 時実行 */
 print(“1 より上”),
 /* false 時実行 */
```

```
  print(“1 以下”)
 )
```

関数名:

seq

説明:

（非推奨 互換性維持関数）

式を卓次評価します。

2 つ以上の式を評価（実行）したい場合に使用します。

最後の式の値を帰り値として返します。

(…),{…}と同等です。

引数:

seq(式[,式,…])

例:

```
seq(
 let(tmp,1),
 print(tmp+1)
)
```

関数名:

while

説明:

条件が 0 より上の間繰り返します。

最後に実行した式の内容を返します。

引数:

while(式(条件),式(繰り返し内容))

例:

```
 {
  cnt=0 ,
  while(
   /* 条件 */
   cnt<100,
   /* 繰り返し内容 */
   {
```

```
    lprint(":",cnt),
    cnt=cnt+1
   }
  )
 }
```

0 から 100 まで表示します

関数名:

eq

説明:

（非推奨 互換性維持関数）

値が等しい場合 1 を等しくない場合 0 を返します。

== と同じです。

引数:

eq(式(比較対象 1),式(比較対象 2))

例:

```
eq(1 ,1) /* 1 が返る */
eq(0 ,1) /* 0 が返る */
```

関数名:

not

説明:

（非推奨 互換性維持関数）

式の値の真偽を反転します。0 より上なら 0、以外は 1 を返します。

!と同じです。

引数:

not(式)

例:

```
not(eq(1 ,1)) /* 0 が返ります */
```

関数名:

gr

説明:

（非推奨 互換性維持関数）
より大きいか判定する(=含まない)。比較値 1>比較値 2 なら１、以外は 0 を返す
＞と同じです。

引数:

gr(式(比較値 1),式(比較値 2))

例:

```
gr(10 ,5) /* 1 を返す */
gr(5 ,10) /* 0 を返す */
gr(10 ,10) /* 0 を返す */
```

関数名:

ge

説明:

（非推奨 互換性維持関数）
以上か判定する(=含む)。比較値 1>=比較値 2 なら１、以外は 0 を返す
>=と同じです。

引数:

ge(式(比較値 1),式(比較値 2))

例:

```
ge(10 ,5) /* 1 を返す */
ge(5 ,10) /* 0 を返す */
ge(10 ,10) /* 1 を返す */
```

関数名:

lt

説明:

（非推奨 互換性維持関数）
より小さいか判定する(=含まない)。比較値 1<比較値 2 なら１、以外は 0 を返す
<と同じです。

引数:

lt(式(比較値 1),式(比較値 2))

例:

```
lt(10 ,5) /* 0 を返す */
```

```
lt(5 ,10) /* 1 を返す */
lt(10 ,10) /* 0 を返す */
```

関数名:

le

説明:

（非推奨 互換性維持関数）

以下か判定する(=含む)。比較値 1<=比較値 2 なら１、以外は 0 を返す

<=と同じです。

引数:

le(式(比較値 1),式(比較値 2))

例:

```
le(10 ,5) /* 0 を返す */
le(5 ,10) /* 1 を返す */
le(10 ,10) /* 1 を返す */
```

関数名:

or

説明:

（非推奨 互換性維持関数）

式が０より上になるまで実行します。どれか 1 つの式が真か返します。

どれか真になればそれ以降は実行されません。

||と同じです。

引数:

or(式 1[,式 2,…])

例:

```
if(
 or(
  ge (tmp,10),
  le(tmp,20)
 ),
 lprint("10 以上か２０以下")
)
```

関数名:

and

説明:

（非推奨 互換性維持関数）

式が 0 以下になるまで実行します。すべての式が真か返します。

どれか偽になればそれ以降は実行されません。

&&と同じです。

引数:

and(式１[,式 2,…])

例:

```
if(
  and(
   ge (tmp,10),
   le(tmp,20)
  ),
  lprint("10 以上かつ２０以下”)
 )
```

関数名:

rnd

説明:

乱数を発生します。0～式の整数部分までの乱数を発生します。

引数:

rnd(式)

例:

```
rnd(10) /* 0～10 までの乱数を発生 */
```

関数名:

for

説明:

繰り返し実行します。指定変数名に初期値から最終地まで進める値ずつ足されながら実行される内容が毎回実行されます。最終値より上になると終了します。

引数:

for(変数名,式(数値初期値),式(数値最終値),式(進める値),式(実行される内容))

例:

```
for(cnt,1 , 100 , 1 , lprint(cnt))
```

1 から 100 まで表示します。

関数名:

valdim

説明:

（非推奨 互換性維持関数）

配列の値を取得します。配列が関数の場合は配列の内容は評価されます。

[]でも配列要素にアクセスできます。

引数:

valdim(ID(変数名),式(インデックス))

例:

```
(
 defdim(tmp,1 ,“1+1”),
 valdim(tmp,1)
)
```

2 が返ります。

```
(
 letdim(tmp,1 ,” 1+1”),
 print(valdim(tmp,1))
)
```

“1+1”が表示されます。

関数名:

deldim

説明:

配列を消去します（※配列用の消去命令が別にあるのは、[]が実装されるまでの名残りです)。

配列は 配列名_セパレータ_インデックス という変数が生成されます。配列名_セパレータ_* を全て消去します。

インデックス指定がある場合特定の部分を消去します。

引数:

deldim(ID(配列名)[,式(インデックス)])

例:

```
(
 letdim(tmp,1 , 1),
 deldim(tmp)
)
```

関数名:

del

説明:

変数、関数、配列を消去します。

引数:

del(ID(変数、関数、配列名))

例:

```
del($tmp) /* 変数、関数を削除 */
del(tmp[2]) /* 配列の要素を削除 */
del(foodim) /* 配列を削除 */
```

関数名:

fordim

説明:

配列の繰り返し処理。インデックスを指定変数に入れながら最後までループします。

配列が多次元だった場合次元ごとのインデックスの値が配列として入ります。

引数:

fordim(ID(変数名(インデックスが入る)),ID(配列名),式(実行式))

例:

```
fordim(idx,dat,lprint(dat[idx]))

{
a[1][1] = 1 ,
a[1][2] = 2 ,
a[1][3] = 3 ,
fordim(tag,a,lprint(a[tag[0]][tag[1]]))
}
```

関数名:

datetime

説明:

日付を作成します。引数がない場合現在時刻を返します。

日付は数値で

yyyymmddhhmmss

yyyy ： 年

mm ： 月

dd ： 日

hh ： 時

mm ： 分

ss ： 秒

（文字数は桁数）

で表されます。

日付文字列(2020-1-2 10:11:12 等)を指定しても作成できます。

時分秒を省略すると 00:00:00　が指定されます。

引数:

datetime[(式(年,日付文字列)[,式(月),式(日)[,式(時),式(分),式(秒))]]]

例:

datetime

20190730110500 等が返ります

datetime(2019,7,3,8,0,0)

20190703080000 が返ります。

datetime(“2020-10-10 10:11:12”)

20201010101112 が返ります。

関数名:

year

説明:

年を取得します。引数なしで現在の時刻から取得します。

引数に日時を指定すると年を返します。

引数:

year[(式(日時))]

例:

year

2019 等が返ります

year(datetime)

2019 等が返ります。

関数名:

mon

説明:

月を取得します。引数なしで現在の時刻から取得します。

引数に日時を指定すると月を返します。

引数:

mon[(式(日時))]

例:

mon

7 等が返ります

mon(datetime)

7 等が返ります。

関数名:

day

説明:

日を取得します。引数なしで現在の時刻から取得します。

引数に日時を指定すると日を返します。

引数:

day[(式(日時))]

例:

day

30 等が返ります

day(datetime)

30 等が返ります。

関数名:

hour

説明:

時を取得します。引数なしで現在の時刻から取得します。

引数に日時を指定すると時を返します。

引数:

hour (式(日時))]

例:

hour

11 等が返ります

hour(datetime)

11 等が返ります。

関数名:

min

説明:

分を取得します。引数なしで現在の時刻から取得します。

引数に日時を指定すると分を返します。

引数:

min (式(日時))]

例:

min

11 等が返ります

min(datetime)

11 等が返ります。

関数名:

sec

説明:

秒を取得します。引数なしで現在の時刻から取得します。

引数に日時を指定すると秒を返します。

引数:

sec (式(日時))]

例:

sec

35 等が返ります

sec(datetime)

35 等が返ります。

関数名:

week

説明:

日付から週を得ます。

(0->日,1->月,2->火,3->水,4->木,5->金,6->土)

引数:

week(式(日付))

例:

week(datetime)

2 等が返ります。

関数名:

timeadd

説明:

時間に秒数を加算します。

引数:

timeadd(式(時間),式(加算秒))

例:

timeadd(datetime , 60 * 60) /* 現在から 1 時間後の時間を得る */

関数名:

timesub

説明:

時間同士を引き算して秒数を返します。日付１－日付２

引数:

timesub(式(日付 1),式(日付 2))

例:

timesub(timeadd(datetime,60),datetime)

60 が返ります。

関数名:

refvaldim

説明:

（非推奨 互換性維持関数）

関数配列を中身を評価せず返します。fncdim,defdim で作成した配列の値が対象になります。

引数:

refvaldim(ID(配列名),式(インデックス))

例:

```
(
 defdim(tmp,1 ,"1+1"),
 print(
  valdim(tmp,1),
  ":",
  refvaldim(tmp,1)
 )
)
```

2:1+1 が表示されます。

関数名:

stradd

説明:

文字列を結合します。& の関数版です。

引数:

stradd(式[,式])

例:

stradd(“abcd”,”def”,”efg”)

“abcddefefg” が返ります。

関数名:

strlen

説明:

文字の長さを返します。

数値を返す。

式２で１を指定すると全角を２文字として返す。

引数:

strlen(式 1,[式 2(0(default)->全角を 1 文字として返す 1->全角を２文字として返す)])

例：

strlen(“abcde”)

5 が返ります。

strlen (″あ aa″,1)

4 が返ります

関数名:

substr

説明:

文字の部分文字を返します。位置は 0 始まりでマイナスで後ろから取ってきます。-1 で一番後ろの文字になります。長さは 1 で 1 文字です、位置がマイナスの場合は後ろから長さを数えます。

式 4 に１を指定すると全角を２文字として処理します。指定しないか 0 で全角も１文字とします。

全角の半分の位置を指定された場合半角スペースに置き換わります。

引数:

substr(式 1(文字列),式 2(開始位置),式 3(長さ)[,式 4(全角２文字とするか 0(default)->１文字 1->２文字)])

例：

substr(“abcdef”,2 , 2)

“cd” が返ります。

substr(“abcdef”,-2 , 2)

“ef” が返ります。

print(substr(″oi あ iu″, 2, 3, 1))

“あ i”が表示されます

関数名:

upper

説明:

文字列を大文字にします。全角文字にも有効です。

引数:

upper(式)

例：

lpper(“ab ａ ｂ”)

“ABＡＢ”が返ります。

関数名:

lower

説明:

文字列を小文字にします。全角文字にも有効です。

引数:

lower(式)

例:

lower(“ABＡＢ”)

“ab ａ ｂ”が返ります。

関数名:

isdef

説明:

変数（関数）または配列が定義されているか返します。

0->定義されていない 1->定義されている

引数:

isdef(ID(変数、関数、配列名))

例:

```
{
 tmp=1 ,
 isdef(tmp),
}
{
 foo[1]=1 ,
 isdef(foo)
}
{
 foo[1]=1,
 isdef(foo[1])
}
```

１が返ります。

関数名:

isdefdim

説明:

（非推奨 互換性維持関数）

配列が定義されているか返します。0->定義されていない 1->定義されている。

式は省略すると ID が配列として定義されているか返し、指定すると指定の

インデックスの定義があるか返します。

引数:

isdefdim(ID(配列名)[,式（インデックス)])

例:

```
(
 tmp[1]=1 ,
 isdefdim(tmp)
)
```

1 が返ります

関数名:

print

説明:

情報エリアの内容をクリアして設定しなおします。

改行は ¥n。

引数:

print(式[,式…])

例:

print(“aaa”,”:”,”bbb”)

aaa:bbb が表示されます

関数名:

net

説明:

ネットから値を取得します。

区別名:

^html : WEB HTML 値の URL の内容を取得します。

第 3 引数にエンコード文字列を渡します””(空文字)か省略した場合は自動

判定になります(“SJIS”等指定)。エンコード文字列は .NET の Encoding クラスに準じます。

第４引数に UserAgent を指定します。省略した場合はデフォルトを指定します。

^post : WEB HTML 値の URL の内容を取得します。

第３引数にパラメータが入ったコンテキストリソース ID を渡します。

第 4 引数にエンコード文字列を渡します省略した場合は “UTF-8”になります(“SJIS” 等指定)。エンコード文字列は .NET の Encoding クラスに準じます。

^bin :式(値)を URL として、式３のファイル名にダウンロードする。

^val : 株価 を Google 検索から取得します。値はティカーコードでも会社名でも検索で使えるものを指定してください。net(^val,^Kawase) で現在の円ドル為替レートを取ります。**(※機能解除版のみ)**

^div : 配当を得ます。値はティカーコード(日本株は数値)。**(※機能解除版のみ)**

^ocfm : 営業キャッシュフローマージンを得ます。値はティカーコード(日本株は数値)。**(※機能解除版のみ)**

^name : 日本株の数値コードから企業名を得ます。**(※機能解除版のみ)**

^eps : 一株利益を得ます。値はティカーコード(日本株は数値)。**(※機能解除版のみ)**

^dpr : 配当性向性(Dividend Payout Ratio)を得ます。値はティカーコード(日本株は数値)。**(※機能解除版のみ)**

^shi : 機関保有率 (Share Held by Institutions)を得ます。値はティカーコード(米株のみ)。**(※機能解除版のみ)**

^per : PER を得ます。値はティカーコード(日本株は数値)。**(※機能解除版のみ)**

^pbr : PBR を得ます。値はティカーコード(日本株は数値)。**(※機能解除版のみ)**

引数:

net(式(区別名),式(値)[,式 3[,式 4]])

例:

print(net(^html,@”http://google.com/”))

html 取得して文字列を返します。

```
{
 ctx = context,
```

```
 ctx.a = "foo",
 ctx.b = "yaa",

print(net(^post,"http://ポストサイトURL",ctx))
}
```

A=foo&b=yaa を引数として URL を POST で呼び出します。

関数名:

delay

説明:

処理を待ちます。単位はmsec(1/1000)です。

引数:

delay(式)

例:

```
delay(1000)
```

1秒ウエイトします。

関数名:

mod

説明:

割り算の余りを返します。式は整数化されます。

引数:

mod(式(割られる値),式(割る値))

例:

```
mod(10 ,3)
```

1が返ります。

関数名:

rinput

説明:

リアルタイムキー入力を返します。値としてキーを識別できる文字列を指定します(大文字小文字区別しません)。押されていれば 1,押されていなければ 0 を返しま

す。

up : Key.Up

down : Key.Down

left : Key.Left

right : Key.Right

sps : Key.Space

a : Key.A

b : Key.B

c : Key.C

d : Key.D

e : Key.E

f : Key.F

g : Key.G

h : Key.H

i : Key.I

j : Key.J

k : Key.K

l : Key.L

m : Key.M

n : Key.N

o : Key.O

p : Key.P

q : Key.Q

r : Key.R

s : Key.S

t : Key.T

u : Key.U

v : Key.V

w : Key.W

x : Key.X

y : Key.Y

z : Key.Z

t0 : Key.NumPad0

t1 : Key.NumPad1

t2 : Key.NumPad2

t3 : Key.NumPad3

t4 : Key.NumPad4
t5 : Key.NumPad5
t6 : Key.NumPad6
t7 : Key.NumPad7
t8 : Key.NumPad8
t9 : Key.NumPad9
ent : Key.Enter
per : Key.OemPeriod

引数:
rinput(式(識別文字列))
例:
rinpt(^sps)
sps が押されたかリアルタイムで返します。

関数名:
input
説明:
入力ウインドウを開きます。
ダイアログを出してキー入力を得る。
式１はキャプション。
式２は 0 で enter で OK 、1(デフォルト)で enter は改行入力となります。
引数:
input[(式 1(プロンプト)[,式 2(改行を許すか))]
例:
input(“数値を入力してください”,0)

関数名:
logprint
説明:
ログファイルに文字列を出力します。メニュー->ヘルプ->ログ表示 から内容を確認することができます。
引数:
logprint(式[,式…])
例:

logprint(“株価更新”,”aaa”)
“株価更新 aaa”が出力されます

関数名:
int
説明:
整数部分を取り出します。
引数:
int(式)
例:
int(3.123)
3 が返ります

関数名:
pow
説明:
値１^値２のべき乗を計算します。
引数:
pow(式(値 1),式(値 2))
例:
pow(2 ,0.5)
2 の 0.5 乗を計算します

関数名:
xpath
説明:
XML の文字列を受けてそれに XPath を適用し添え字番号(0 から)の文字列を抽出します。添え字は省略出来て 0 を示します。
Chrome から得られる XPath 等とは互換性がない場合があります。
C# の HtmlAgilityPackXPath の XPath を使用しています。

引数:
xpath(式(XML 文字列),式(xpath 文字列)[,式(何番目の値を取り出すか添え字)])

例:

```
print(
 xpath(
  net(^html,"http://google.com"),
  '//a'
 )
)
```

関数名:

regex

説明:

指定した文字列に正規表現を適用して文字を返します。()（括弧）でくくったグループの何番目の文字列を返すか添え字で指定します。エラーは””（空文字）を返します。.Net の Regex クラスに準じます。

引数:

regex(式(マッチング文字列),式(正規表現)[,式(インデックス)])

例:

```
print(
 regex("abcde",@"(.*)(cd)(.*)",1)
)
```

cd が返ります。

関数名:

regedidx

説明:

指定した文字列に正規表現を適用してマッチング文字列のマッチング位置インデックス(0 始まり)を返します。()（括弧）でくくったグループの何番目の文字列を返すか添え字で指定します。エラーは-1 を返します。

引数:

regexidx(式(マッチング文字列),式(正規表現)[,式(インデックス)])

例:

```
print(
 regexidx("abcde",@"(.*)(cd)(.*)",1)
)
```

2 が返ります

関数名:

fprint

説明:

ファイルに出力します。ファイル名にファイルのフルパス。値に出力文字列を指定するとファイルの最後に追加出力します。失敗した場合 0 を返します。成功した場合は１を返します。

UTF-8 BOM 無しで出力します。

ファイルパスが相対パスだった場合 workfld からのパスになります。

引数:

fprint(式(ファイル名),式(出力内容),…)

例:

fprint(@”c:¥doc¥foo.txt”,”aaa”,”bbb”)

“aaabbb”が c:¥doc¥foo.txt の最後に追加されます。

関数名:

fload

説明:

ファイルをロードして文字列を返します。失敗した場合 “”(空)を返します。

ファイルパスが相対パスだった場合ドキュメントフォルダからのパスになります。

読み込むファイルのエンコードは UTF-8,UTF-16 は自動で判定されますが、文字化けする場合は、第二引数にはエンコード文字列を指定できます(“SJIS”等)。

エンコード文字列は .NET の Encoding クラスに準じます。

引数:

fload(式(ファイル名)[,式(エンコード文字列)])

例:

fload(@”c:¥doc¥foo.txt”)

関数名:

delfile

説明:

ファイル名を削除します。

ファイルパスが相対パスだった場合 workfld からのパスになります。

引数:

delfile(式(ファイル名))

例:

```
delfile(@”c:¥doc¥foo.txt”)
```

関数名:

existfile

説明:

ファイル名があれば 1 なければ 0 を返します。

ファイルパスが相対パスだった場合 workfld からのパスになります。

引数:

existfile(式(ファイル名))

例:

```
existfile(@”c:¥doc¥foo.txt”)
```

関数名:

frdlg

説明:

読み込みファイル名選択ダイアログを開きます。タイトル、デフォルトファイル名、拡張子(*.txt の形“” ですべてのファイル)を指定します。キャンセルされた場合場合 “”(空)を返します。

引数:

frdlg(式(タイトル),式(デフォルトファイル名),式(拡張子))

例:

```
print(
 frdlg("入力ファイルを指定してください","foo.txt","*.txt")
)
```

関数名:

fwdlg

説明:

書き込みファイル名選択ダイアログを開きます。タイトル、デフォルトファイル名、拡張子(*.txt の形“” ですべてのファイル)を指定します。指定なしの場合

“”(空)を返します。

引数:

fwdlg(式(タイトル),式(デフォルトファイル名),式(拡張子))

例:

```
print(
 fwdlg("出力ファイルを指定してください","foo.txt","*.txt")
)
```

関数名:

chgret

説明:

ネットや外部ファイルの “¥r” の場合 “¥r¥n” に変換します。

引数:

chgret(式(文字列))

例:

```
chgret(fload(@”c:¥doc¥foo.txt”))
```

関数名:

replace

説明:

文字列を置換します。.Net の Regex.Replace に準じます。

引数:

replace(式(対象文字列),式(置換対象正規表現),式(置換文字列))

例:

```
replace("ab123456cdefch",@"[^¥d]"," ")
```

関数名:

lprint

説明:

情報エリアの内容に追加で文字を出力します。

改行は ¥n。

引数:

lprint(式[,式…])

例:

lprint(“aaa”,”:”,”bbb”)

aaa:bbb が追加で表示されます

関数名:

reskind

説明:

指定リソース ID のリソースの種類を文字列で得ます。リソースでない(終了済み含む)場合は 0 を返します。

“PRC” : プロセスリソース

“GRPH” : グラフリソース

“CNT” : コンテキストリソース

“WEB” : WEB リソース

“STR” : ストリングバッファリソース

引数:

reskind(式(リソース ID))

例:

reskind(grph(tmp))

関数名:

delres

説明:

指定リソース ID のリソースを削除します。

GC を待たず強制的に削除されます。

引数:

delres(式(リソース ID))

例:

delres(tmpid)

関数名:

grph

説明:

グラフリソースを作成します。

配列名の入れいつのグラフを作成します。

リソース ID を返します。
右上の×で消すか、delres すると消去されます。
同時に複数作成可能です。
X 軸種類は
-1:自動判定(デフォルト)
0 : 数値
1:日付(X 軸最小、最大、間隔無効)

グラフ種類は
0 : 折れ線グラフ (デフォルト)
1:棒グラフ
配列のインデックスが文字列だとラベルが設定されます。
X 軸種別の自動判定はインデックスが全て 19000101000000 以上であれば日付
と判定され、それ以外は数値と判定されます。
reskind で “GRPH” を返します。

引数:
grph(ID(配列名)[,式(ラベル文字列)[,式(X 軸種類) [,式(グラフ種類) [,式(X 最小値),式(X 最大値)[,式(X 区切り間隔)[, 式(Y 最小値),式(Y 最大値)[,式(Y 区切り間隔)]]]]]])
例:
grph(tmp,” tmp”)

関数名:
prc
説明:
プロセスリソースを作成します。
文字列,無名関数を与え並列に実行します。
リソース ID を返します。
グローバル変数にアクセスできますが排他処理が入り、同時アクセスする
と速度は落ちます。
引数を指定できます。
同時並列数はスーパーユーザー以外は制限され、10 となります。
引数:
prc(式(実行文字列、無名関数) [,式(引数)...])
例:

```
prc(“(delay(100),net(^html,@’ http://foo.com/’ ))”)
prc(“1+@1”,123)
{ tmp = 1 , prc(=>{ lprint(tmp) }),waitresall }
```

関数名:

argval

説明:

引数を値で指定して返します。1 だと@1 を示します。

引数:

argval(式(引数のインデックス値))

例:

```
argval(1)
```

関数名:

waitresall

説明:

関数内で作成したリソースを待ちます。

同一関数内で作成したリソース ID は関数と紐づけされて保持されています。

waitresall では関数内で作成したリソースを個別に待たずに複数対象に待つことができます。

リソース種別を指定することで特定のリソースに対して待つことができます。

省略するとすべてのリソース種別が対象になります。

引数:

waitresall[(式(リソース種別))]

例:

```
{
 prc(“net(^html,@’ http//foo.com’ )”),
 prc(“net(^html,@’ http//foo2.com’ )”),
 prc(“net(^html,@’ http//foo3.com’ )”),
 waitresall
}
```

関数名:

return

説明:

関数を式の値で終了します。式は省略すると０

値としては数値か、文字列を返せます。

引数:

return[(式(帰り値))]

例:

```
 {
  sum=0 ,
  for(cnt , 1 , 1000 , 1 ,
    {
     if(cnt > 100 , return(cnt)),
     sum=sum + cnt
    }
   )
 }
```

関数名:

break

説明:

ループを式の値で終了します。式は省略すると０

引数:

break [(式(ループの終了値))]

例:

```
 for(cnt , 1 , 100 , 1 ,
  if(cnt>50,break)
 )
```

関数名:

continue

説明:

ループの先頭に戻ります。最終計算値を式の値にします。式は省略すると０

引数:

continue [(式(ループの計算値))]

例:

```
for(cnt , 1 , 100 , 1 ,
 {
  if(cnt<=50,continue),
  lprint(cnt)
 }
)
```

関数名:

context

説明:

コンテキストリソースを作成します。リソース種別は”CNT”を返します。コンテキストは変数、関数を定義しておけるエリアの事です。コンテキストリソースを入れた変数名.変数名(関数名) でアクセスします。明示的に消去する必要があります。関数定義内では this で関数定義のあるコンテキストリソース ID を示します。クラスを作成できます。= の代入だけで変数にバインドされます。

引数:

context

例:

```
 {
  cnt=context , /* コンテキストリソース作成 */
  cnt.foo=123 , /* コンテキストリソース cnt に変数 foo を作成 */
  fnc(cnt.yaa, lprint(this.foo)) /* コンテキストリソース cnt に関数 yaa を作成。関数内で関数定義のあるコンテキストリソース this の変数 foo をアクセス */
 }
```

関数名:

delresall

説明:

関数内で定義したリソースを全て削除します。式でリソースの種類を指定できます。

“ALL”でトップの関数呼び出しからの全てのリソースを削除します。終了前にす

べてのリソースを削除するときに使用します。
　強制的に削除されます。
引数:
　delresall(式(リソース種別))
例:
```
delresall
delresall(“CNT”)
```

関数名:
　use(廃止)
説明:
　(※)スレッドセーフにできない為廃止します
コンテキストリソースにアクセスを切り替えます。
　コンテキストリソース内では変数の定義は $ID を含めすべてコンテキスト内に行われますが未定義変数は外部ローカル変数、グローバル変数へのアクセス可能です。
　関数呼び出し先で callcnt のアクセス先は use の呼ばれた関数となり、use で指定されたコンテキストリソースではありません。
引数:
　use(式(コンテキストリソース ID),式(実行分))
例:
```
{
print(""),
tmp = 123, // 外部ローカル変数定義
$yaa = 543, // グローバル変数定義
bindres(ctx,context), // コンテキスト作成
use(ctx, // ctx を参照
 {
  lprint(tmp), // 外部定義した変数を参照できます
  tmp2 = 321, // 変数作成は use で指定したコンテキストに行われます
  lprint($yaa), // コンテキスト内に定義されていない変数は外部ローカルに対して行うのでグローバル変数にもアクセス可能です
  $foo = 543 // グローバルでなくコンテキスト内に定義されます
 }
),
```

```
 lprint(rdump(ctx)),
}
```

関数名:

sysinfo

説明:

システムの情報を文字列で返します。確保されているリソースなどを表示します。

引数:

sysinfo

例:

```
print(sysinfo)
```

関数名:

len

説明:

配列の長さを返します。

引数:

len(ID(配列名)[,開始インデックス[,終了インデックス]])

例:

```
len(dimvar)
```

関数名:

amean

説明:

配列の算術平均値(Arithmetic mean)を返します。

引数:

amean(ID(配列名)[,開始インデックス[,終了インデックス]])

例:

```
amean(dat)
```

関数名:

gmean

説明:

配列の幾何平均値(Geometric mean)を返します。

引数:

gmean(ID(配列名)[,開始インデックス[,終了インデックス]])

例:

gmean(dat)

関数名:

hmean

説明:

配列の調和平均値(Harmonic mean)を返します。

引数:

hmean(ID(配列名)[,開始インデックス[,終了インデックス]])

例:

hmean(dat)

関数名:

med

説明:

配列のメディアン(中央値)を返します。

配列の値を昇順にソートして奇数なら配列の中央の要素を返し、偶数なら
中央の 2 要素の平均を返します。

引数:

med(ID(配列名)[,開始インデックス,終了インデックス])

例:

med(dat)

関数名:

hmed

説明:

ヒストグラム配列(インデックスが値で値が度数)のメディアン(中央値)を返しま

す。
　配列の値を昇順にソートして奇数なら配列の中央の要素を返し、偶数なら
　中央の 2 要素の平均を返します。
引数:
　hmed(ID(配列名)[,開始インデックス[,終了インデックス]])
例:
　med(dat)

関数名:
　dis
説明:
　配列の分散(dispersion)を返します。
引数:
　dis(ID(配列名)[,開始インデックス[,終了インデックス]])
例:
　dis(dat)

関数名:
　hamean
説明:
　ヒストグラム配列の算術平均値(Arithmetic mean)を返します。
引数:
　hamean(ID(配列名)[,開始インデックス[,終了インデックス]])
例:
　hamean(dat)

関数名:
　hgmean
説明:
　ヒストグラム配列の幾何平均値(Geometric mean)を返します。
引数:
　hgmean(ID(配列名)[,開始インデックス[,終了インデックス]])
例:

hgmean(dat)

関数名:

hhmean

説明:

ヒストグラム配列の調和平均値(Harmonic mean)を返します。

引数:

hhmean(ID(配列名)[,開始インデックス[,終了インデックス]])

例:

hmean(dat)

関数名:

hdis

説明:

ヒストグラム配列の分散(dispersion)を返します。

引数:

hdis(ID(配列名)[,開始インデックス,終了インデックス])

例:

hdis(dat)

関数名:

cov

説明:

配列名１と配列名２の共分散(Covariance)を返す
インデックスが同じ組から共分散を計算します。
同じインデックスとする範囲は先頭から範囲内に見つかったものを
ペアにするので全く同じ値以外に範囲外のペアが先に見つかった
場合は違う値のペアになることがあります。
配列２のインデックスにはオフセットを指定できます。

引数:

cov(ID(配列名 1),ID(配列名 2),[,式(同じインデックスとする範囲),[開始インデックス[,終了インデックス[,配列名２オフセット]]]])

例：

cov(dat1,dat2)

関数名:

cor

説明:

配列の相関係数(Correlation coefficient)を返します。
インデックスが同じ組から相関係数を計算します。
同じインデックスとする範囲は先頭から範囲内に見つかったものを
ペアにするので全く同じ値以外に範囲外のペアが先に見つかった
場合は違う値のペアになることがあります。
配列２のインデックスにはオフセットを指定できます。

引数:

cor(ID(配列名 1),ID(配列名 2) [,式(同じインデックスとする範囲),[開始インデックス[,終了インデックス[,配列名 2 オフセット]]]])

例：

cor(dat1,dat2)

関数名:

waitres

説明:

リソースの終了を待ちます

引数:

waitres(式(リソース ID)[,式(リソース ID)...])

例：

waitres (resid1)

関数名:

sqr

説明:

ルートを計算します

引数:

sqr (式)

例:

```
sqr (2)
```

関数名:

rdump

説明:

リソースコンテキストの内容をダンプします。

読み込みできる文字列を返します。

引数:

rdump(式(リソース ID))

例:

```
{
 ctx=context,
 ctx.foo=1 ,
 delfile(“data.txt”), /* fprint は追記になるので事前にファイル消去 */
 fprint(“data.txt”,rdump(ctx)), /* data.txt に ctx の内容を保存 */
 delres(ctx) /* リソースは残り続けるので消す */
}
```

関数名:

getrcom

説明:

リソースのコメント文字列を得る

引数:

getrcom(式(リソース ID))

例:

```
{
 ctx=context ,
 setrcom(ctx,” コメント”),
 print(getrcom(ctx)),
 delres(ctx) /* リソースは残り続けるので消す */
}
```

関数名:

setrcom

説明:

リソースのコメント文字列を設定する

引数:

setrcom(式(リソース ID),式(コメント文字列))

例:

```
{
 ctx=context,
 setrcom(ctx,”コメント”),
 print(getrcom(ctx)),
 delres(ctx) /* リソースは残り続けるので消す */
}
```

関数名:

cpdim

説明:

配列をコピーする

コピー先配列は最初に初期化される。

コピーされた配列の要素数を返す

関数があれば自己参照 this はコピー先を示す(代入ではそのまま代入され this は元参照)。

引数:

cpdim(ID(コピー元配列名),ID(コピー先配列名))

例:

```
cpdim(foo,foo2)
```

関数名:

mgdim

説明:

配列をマージする

コピー先配列は最初に初期化されない。

コピーされた配列の要素数を返す

引数:

mgdim(ID(マージ元配列名),ID(マージ先配列名))

例:

mgdim(foo,foo2)

関数名:

cload

説明:

rdump で出力した文字列をコンテキストに読み込む。

読み込んだ数を返す。

読み込み先のコンテキストリソースは初期化されず上書きされる。

エラーで 0 を返す

引数:

cload (式(コンテキストリソース ID),式(読み込む文字列))

例:

```
{
ct=context,
cload (ct,fload ("data.txt")), /* data.txt の内容を読み込み */
print(rdump(ct)),
delres(ct) /* リソースは残り続けるので消す */
}
```

関数名:

solve

説明:

式の解を求める。 @1 を未知数として求める。

引数が 1 つだけの場合、計算式だけを与え"式=式"の形となり = で式をつなげる文字列を渡す。

この場合 = は文字列内に 1 回しか出現できず、無名関数(=>{})は使えない。

許容誤差はブレント法の場合関数の値の誤差で|f(X)|の値となる。

デフォルトでは 1e-10

ニュートン法の開始点はデフォルトは 1.0。

範囲の最大値を与えるとアルゴリズムがブレント法になり、それ以外はニュートン法となる。

ブレント法では f(最小範囲) と f(最大範囲) の符号が逆でないと計算できないので 左辺-(右辺) の値の符号が逆になる範囲を与える必要がある。

引数:

solve(式(左辺式文字列)[,式(右辺式文字列)[,式(求める範囲最小値)[,式(求める範囲最大値)[,式(許容誤差)]]]])

例:

```
solve(“pow(1.02 ,@1)=2”) /* ニュートン法*/
solve(“pow(1.02 ,@1)”,”2”, 0.1) /* ニュートン法で開始点指定*/
solve(“pow(1.02 ,@1)”,”2”, 0.1 ,50) /* ブレント法で計算範囲指定 */
solve(“pow(1.02 ,@1)”,”2”, 0.1 ,50 ,1e-5) /* ブレント法で計算範囲、許容誤差指定 */
```

関数名:

eval

説明:

文字列,無名関数として作成したLCSを評価する。

引数を指定できます。

引数:

eval(式(式文字列,無名関数)[,式(引数1)...])

例:

```
eval(“@1*@2”,1 , 2)
eval(“isdef($”& valname & “)”)
```

関数名:

workfld

説明:

作業フォルダを返す。

ログが出力されるフォルダとなる。

引数:

workfld

例:

```
lprint(workfld)
```

関数名:

bindres

説明:

リソースを変数に結び付ける。

変数が削除されるとリソースも一緒に削除される。

１つの変数のみにバインドされ削除された後リソースは強制削除される。

コンテキストリソースの場合リソース内変数にバインドされたリソースも全て強制削除される。

バインドした時点で変数が既にバインドされている場合、

既にバインドされていたリソースは削除され、新しいリソースに

再バインドされます。

引数:

bindres(ID(変数名),リソース ID)

例:

bindres(ctx,context) // ctx 破棄と同時にコンテキストリソースも破棄される。

関数名:

setcom

説明:

変数にコメントを付ける。

配列が指定された場合全ての配列要素に同じコメントを付ける。

引数:

setcom(ID(変数名),式(コメント文字列))

例:

setcom(foo,”コメント”)

関数名:

getcom

説明:

変数のコメントを得る。

配列が指定された場合先頭の要素のコメントを得る。

引数:

getcom(ID(変数名))

例:

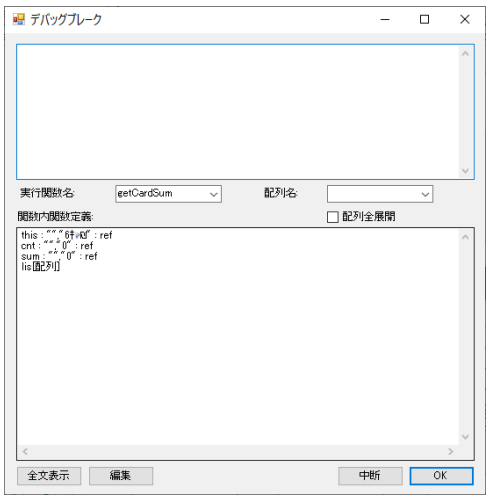
getcom(foo)

関数名:

dbg

説明:

プログラムを中断してデバッグ情報を表示する。



引数のメッセージを表示し、関数内の変数を表示します。

ドロップダウンリストに呼び出し順に関数が設定されており選択することで任意の関数のローカル変数を表示できます。

全文表示ボタンが押せる場合押すと実行されている全ソースが表示され実行中行にカーソルが移動します。

編集ボタンを押すとエディタで止まっている箇所が開きカーソルが移動します。

コンテキストリソース内の関数呼び出しで実行された場合 this がリストの最後に追加され、選択すると定義されているコンテキストリソースの内容を表示します。

配列全展開のチェックを入れると配列が展開されます、チェックが入っていない場合配列は配列 ID のみ表示され、配列名コンボボックスで選択した場合配列の内容が表示されます。

変数表示部で左ダブルクリックすると変数の内容をコンテキストリソース ID として内容を表示します。

コンテキストリソース ID でなかった場合は変数の内容が表示されます。

展開されていない配列の行をダブルクリックすると展開された配列が表示されます。

中断を押すとその場で中断されます。

OK を押すか enter キーでダイアログを閉じて実行が再開されます。

引数:

dbg[(式(表示メッセージ))]

例:

```
{
 tes=123 ,
 dbg( “tes=” & tes)
}
```

関数名:

xpathdim

説明:

xpath で html 解析。

配列に返す。

式 1 の文字列に式 2 の XPath を適用して InnterText を返す。

成功すると見つかった要素数,エラーは-1 を返す。

ID2 には属性を返す配列名を指定する。

ID2[値の該当タグ][属性名] に属性が帰る。

引数:

xpathdim(ID(値を返す配列 ID),式 1,式 2[,ID2(属性を返す配列 ID)])

例:

xpath(ret,net(^html,https://foo.com),” //a”)

関数名:

regexdim

説明:

正規表現を適用する。

マッチグループを配列に返す。

文字列（式 1）に正規表現（式 2）を適用してグループ文字列を配列名に返す。

式３で開始文字位置を設定でき、帰り値を設定していけば連続でマッチング

できる。

成功するとマッチした文字列の次の文字インデックス、エラーは-1 を返す。

引数:

regexdim(ID(値を返す配列 ID),式 1,式 2[,式 3(スタートインデックス)])

例:

```
{
 str="<foo><taa><yaa>",
 st = 0 ,
 while((st = regexdim(ret,str,@"<(.+?)>",st)) > 0 ,
  fordim(tag,ret,lprint(tag & ":" & ret[tag] & "¥n"))
```

```
 )
}
```

関数名:

getpos

説明:

変数の定義位置を得る。

配列の要素も 1 変数なので全体での位置を返す。

エラーは-1 を返す。

引数:

getpos(ID(変数名))

例:

```
{
  foo = 1 ,
  taa[1] = 1 ,
  yoo = 1 ,
  taa[2] = 2 ,
  print(getpos(taa[1]) & ":" & getpos(taa[2]))
}
```

21:23 が表示される。

関数名:

mvpos

説明:

変数の位置を変える。変更前の位置を返す

エラーは-1 を返す。

引数:

mvpos(ID(変数名),式(位置))

例:

```
{
 foo[1] = 1 ,
 foo[2] = 2 ,
 foo[3] = 3 ,
mvpos(foo[3],getpos(foo[2])), /* foo[2] の前に foo[3]を移動 */
fordim(tag,foo,lprint(tag & ":" & foo[tag] & " "))
}
```

1:1 3:3 2:2 と表示される。

関数名:

htmldecode

説明:

HTML の特殊文字を変換して返す。

引数:

htmldecode(式(文字列))

例:

print(htmldecode(“a&b”))

a&b と表示される

関数名:

htmlencode

説明:

文字列を HTML の特殊文字に変換して返す

引数:

htmlencode(式(文字列))

例:

print(htmlencode(“a&b”))

a&b と表示される

関数名:

urldecode

説明:

URL の特殊文字を変換して返す

引数:

urldecode(式(文字列))

例:

print(urldecode(“https://foo.com/search?q=a%26b%26c”))

https://foo.com/search?q=a&b&c と表示される

関数名:

urlencode

説明:
　文字を URL エンコードして返す
引数:
　urlencode(式(文字列))
例:
　print(urlencode(“a&b&c”))
a%26b%26c と表示される

関数名:
　rchgret
説明:
　“¥r¥n” を“¥n” に変換します。
引数:
　rchgret(式(文字列))
例:
　rchgret(“a¥nb”)

関数名:
　sort
説明:
　配列の順序をソートする。
　判定式を与えない場合数値か文字列を自動判定してソートする。
　ソートは配列の fordim での順番に反映される。
　判定式を与えた場合は、@1 , @2 に比較する配列の候補２つの値
　name に配列名, ind1,ind2 に比較する配列の候補のインデックス
　(多次元の場合配列 ind1[0],... ind2[0],...)
　番号 1,2 は比較する配列の要素１つめ、２つめを示す。
　配列がコンテキストリソース参照の場合 this にコンテキストリソース ID が
　入る。
　無名関数の場合は this は設定されません。
　1 つめ < 2 つめ で 0 より下(0 含まない)
　1 つめ > 2 つめ で 0 より上(0 含まない)
　1 つめ == 2 つめ で 0
　を返すような判定式を与える。
　ソートした配列の要素数を返す。
　判定式内では呼び出し元変数の参照が可能です。また判定式は => {} でも書く

ことが可能です。

引数:

sort(ID(配列名)[,式(判定式文字列)])

例:

```
{
 print(""),
 a[0] = 3 ,
 a[1] = 1 ,
 a[2] = 0 ,
 sort(a,"if(@1 < @2 , -1 , if(@1 > @2 , 1 , 0))"), /* sort(a) と同じ */
 fordim(tag,a,lprint(tag & ":" & a[tag] & "¥n"))
}
```

```
 2:0
 1:1
 0:3
```

と出力。

関数名:

reverse

説明:

配列の順序を逆転する。

引数:

reverse (ID(配列名))

例:

```
 {
  a[0] = 3 ,
  a[1] = 2 ,
  a[2] = 1 ,
 reverse (a),
 fordim(tag,a,lprint(tag & ":" & a[tag] & "¥n"))
}
```

```
2:1
1:2
0:3
```

と表示される。

関数名:

split

説明:

文字列を分割して配列に返す。

分割文字は正規表現。分割数を返す。

該当しない場合も全体が入った 1 要素の配列を返す。

引数:

split(ID(値を返す配列名),式(文字列),式(分割文字正規表現文字列))

例:

```
 {
 print(""),
 split(tmp,"foo,taa:yaa",@",|:"),
 fordim(tag, tmp,lprint(tag & ":" & tmp[tag]&"¥n"))
}
0:foo
1:taa
2:yaa
```

が返る。

関数名:

web

説明:

web リソースを作成します。

リソース ID を返します。

web リソースは指定された WEB ページを表示します。

reskind は “WEB” を返します。

サイズ X は 0 で画面外、マイナスでデフォルト、プラスでサイズ指定となります。

引数:

web[(式 1(URL)[,式 2(サイズ X),式 3(サイズ Y)])]

例:

```
waitres(web( “https://google.com” ))
```

関数名:

seturl

説明:

web リソースに URL を設定します。

リソース ID を返します。

引数:

seturl(式(WEB リソース ID),式(設定する url 文字列))

例:

```
{
 id = web,
 seturl(id,” https://google.com” ),
 waitres(id)
}
```

関数名:

sethtml

説明:

web リソースに HTML を設定します。

リソース ID を返します。

引数:

sethtml (式(WEB リソース ID),式(設定する HTML 文字列))

例:

```
{
 id = web,
 sethtml (id,net(^html,” https://google.com” )),
 waitres(id)
}
```

関数名:

gethtml

説明:

web リソースから HTML を取得します。

引数:

gethtml (式(WEB リソース ID))

例:

```
{
 id = web( “https://google.com” ),
 print(gethtml(id)),
 delres(id)
}
```

関数名:

aline

説明:

配列の移動平均線を求めます。指定配列に値を返します。
インデックスは数値か日付のみで文字列は使用できません。
指定されたインデックス前までの要素の平均を取り配列を作成します。
どこまでの要素との平均を取るかはインデックス範囲により指定されます。
配列の個数からではないので注意必要です。
平均化する範囲は日付の場合秒数で指定します。
平均計算対象インデックスは
“(作成インデックスーインデックス範囲)”(含む)から
”作成インデックス”(含む)
までの範囲のインデックスの要素を平均化します。
インデックスは数値か、日付かを指定できますが、無指定または０で自動判定されます。
全てのインデックスが 19000101000000 以上の場合日付と判定されそれ以外は数値インデックスと判定されます。
スタートインデックス、終了インデックスは作成するインデックスの範囲を指定します。

引数:

aline(ID(帰り値を入れる配列 ID),ID(配列),式(平均をとるインデックス範囲(日付の場合は秒数))[,式(インデックス種別(0->自動判定 1->数値インデックス 2->日付インデックス))[,式(スタートインデックス)[,式(終了インデックス)]]])

例:

```
{
 a[3] = 3 ,
 a[2] = 2 ,
 a[0] = 0 ,
 aline(tmp,a,1),
 waitres(grph(tmp,"",0))
}
```

関数名:

callcnt

説明:

関数呼び出し元のコンテキストリソース ID を得ます。

関数呼び出し元の変数にコンテキストリソース経由でアクセスでき、

作成も可能です。

パラメータの配列渡し、配列として値を返したい場合などに使用します。

callcnt で出られたコンテキストリソース ID は bindres や delres で削除する必要はありません。

callcnt は .(ドット)アクセスで ID がグローバル変数と一致するとグローバル変数をアクセスします(呼び出し元と同じ環境)。

引数:

callcnt

例:

```
{
 fnc(foo,
{
 callcnt.tes[0] = callcnt.yaa , /* 呼び出し元の変数 yaa を参照
                                   呼び出し元の配列 tes に書き込み */
 callcnt.tes[1] = callcnt.yaa + 1
}
),
 yaa = 123 ,
 foo,
 fordim(tag,tes,lprint(tag & ":" & tes[tag] & "¥n"))
}
```

```
0:123
1:124
```

と表示。

関数名:

fmax

説明:

最大値を求めます。引数でスキップ判定式を書きます。

デフォルトですべてが対象。

@1 に値,name に配列名,ind , ind[] にインデックスの値が入るので計算に含める

なら 0 以上 含めないなら 0 以下を返す。
配列がコンテキストリソース参照の場合 this にコンテキストリソース ID が入る。
判定式内では呼び出し元変数の参照が可能です。また判定式は => {} でも書くことが可能です。

引数:

fmax(ID(配列 ID)[,式(スキップする判定式)])

例:

```
{
 tmp[0][1] = 1 ,
 tmp[2][3] = 2 ,
 tmp[4][5] = 3 ,
 fmax(tmp,"if(ind[0] > 1 , 1 , 0)")
}
```

上記では tmp[2][3],tmp[4][5] が計算の対象になります。

関数名:

fmin

説明:

最小値を求めます。引数でスキップ判定式を書きます。
デフォルトですべてが対象。
@1 に値,name に配列名,ind , ind[] にインデックスの値が入るので計算に含めるなら 0 以上 含めないなら 0 以下を返す。
配列がコンテキストリソース参照の場合 this にコンテキストリソース ID が入る。
判定式内では呼び出し元変数の参照が可能です。また判定式は => {} でも書くことが可能です。

引数:

fmin(ID(配列 ID)[,式(スキップする判定式)])

例:

```
{
 tmp[0][1] = 1 ,
 tmp[2][3] = 2 ,
 tmp[4][5] = 3 ,
 fmin(tmp,"if(ind[0] > 1 , 1 , 0)")
}
```

上記では tmp[2][3],tmp[4][5] が計算の対象になります。

関数名:

fnc

説明:

関数を式のまま登録します。

配列変数も登録できます。

引数:

fnc(ID(関数名),式(関数定義内容))

例:

```
{
 fnc(foo,
  {
   sum = 0 ,
   for(cnt , 1 , 100 , 1 , sum = sum + cnt)
  }
 ),
 foo
}
```

関数名:

fncdim

説明:

（非推奨 互換性維持関数）

配列関数を式のまま登録します。

fnc で代用できます。

引数:

fncdim(ID(関数名),式(インデックス),式(関数定義内容))

例:

```
{
 fncdim(foo, 1 ,  // fnc(foo[1] , としても同じ
  {
   sum = 0 ,
   for(cnt , 1 , 100 , 1 , sum = sum + cnt)
  }
 ),
```

```
 foo[1]
}
```

関数名:

isbind

説明:

変数がバインドされているか返します。

0:バインドされていない 1:バインドされている

引数:

isbind (ID(変数名))

例:

```
 {
  bindres(ctx,context),
  isbind(ctx)
 }
```

関数名:

json

説明:

json 文字列をコンテキストリソースに変換します。

入れ子の json はコンテキストリソース内の変数にバインドされた

コンテキストリソースとして実現されます。

true,false,null は文字列として読み込まれます。

コンテキストリソース ID を返します。

消去するには帰ってきたコンテキストリソースを削除すれば、入れ子の

バインドされたコンテキストリソースは自動で削除されます。

引数:

json(式(json 文字列))

例:

```
{
 jsonctx = json('
{
 "a":{"b":123 , "c":[false,true,null]}
}
 '),
// jsonctx.a.b jsonctx.a.c[1] 等でアクセスする
```

```
 print(rdump(jsonctx) & "¥n" & rdump(jsonctx.a) & ":" & jsonctx.a.c[1])
}
```

関数名:

mkjson

説明:

コンテキストリソースの内容を json 文字列に変換します。

入れ子のコンテキストリソースはバインドされている必要があります。

引数:

mkjson(式(コンテキストリソース ID))

例:

```
{
 jsonctx = context,
 jsonctx.a = 123 ,
 jsonctx.b[0] = 1 ,
 jsonctx.b[1] = 2 ,
 jsonctx.b[2] = 3 ,
 bindres(jsonctx.c,context),
 jsonctx.c.d = 321 ,
 print(mkjson(jsonctx))
}
```

関数名:

unbindres

説明:

変数のリソースバインドを解除する。

引数:

unbindres(ID(変数名))

例:

```
{
 bindres(we,web),
 unbindres(we)
}
```

関数名:

xml

説明:

xml 文字列をコンテキストリソースに変換します。

属性は @ID の変数として定義されます。

属性が定義された場合は値は #text という変数に入ります。

消去するには帰ってきたコンテキストリソースを削除すれば、入れ子の

バインドされたコンテキストリソースは自動で削除されます。

引数:

xml (式(xml 文字列))

例:

```
{
 xmlctx = xml('
<a><b>1</b><b>2</b><c>foo</c></a>
'),
// xmlctx.a.c xmlctx.a.b[1] 等でアクセスする
 print(xmlctx.a.c & ":" & xmlctx.a.b[1])
}
```

関数名:

mkxml

説明:

コンテキストリソースの内容を xml 文字列に変換します。

入れ子のコンテキストリソースはバインドされている必要があります。

xml で読み込まれたコンテキストリソースの形式以外は変換できません。

ルートの XML ノードを 1 つ作り、その下にノードを作成します。

ノードはコンテキストリソースにバインドされた変数として表します。

ノードが属性を持つ場合は@ID という変数で属性を定義できます。

属性を定義した場合値は #text という変数に入れます。

引数:

mkxml (式(コンテキストリソース ID))

例:

```
{
 xmlctx = context,
 xmlctx.a = context, // ルートノード
 xmlctx.a.b = 321 , // 値
 xmlctx.a.c[0] = 1 , // 配列
 xmlctx.a.c[1] = 2 ,
```

```
 xmlctx.a.c[2] = 3 ,
 xmlctx.a.d = context, // 子ノードd
 xmlctx.a.d.("@name") = "foo" , // 属性 name
 xmlctx.a.d.("#text") = "yaa" , // 値
 print(mkxml(xmlctx))
}
```

<a>321<d name="foo">yaa</d><c>1</c><c>2</c><c>3</c>
が表示される。

関数名:
　cpcnt
説明:
　コンテキストリソースをコピーする。マージ先 ID が指定されていればマージする。
　コンテキストリソースにバインドされた変数があればコピーされる。
 関数があれば自己参照 this はコピー先を示す。

引数:
　cpcnt(式(コンテキストリソース ID)[,式(マージするコピー先コンテキストリソース ID)])
例:

```
{
 ctx = context,
 ctx.foo = 123 ,
 ctx2 = cpcnt(ctx),
 print(rdump(ctx2)),

 ctx3 = context,
 ctx3.yaa = 123 ,
 cpcnt(ctx2,ctx3),
 lprint(rdump(ctx3))
}
```

関数名:
　try
説明:

エラーダイアログをスキップして実行。
実行文の帰り値を返します。
エラーが発生した場合 ID の変数にエラーメッセージが入り、指定された式が
実行されます。
prc は別処理で実行されるため、渡す文字列内にエラーがある場合
渡す文字列にも try を書く必要があります。

引数:

try(式(実行文),ID(エラーメッセージを入れる変数名),式(エラー発生時実行文))

例:

```
ctx = xml(try(net(^html,"foo.com"),msg,net(^html,"yaa.com")))
```

関数名:

throw

説明:

エラーを任意文字で発生させます。
throw が実行されると関数の実行は終了しエラーが直ちに返されます。

引数:

throw(式(エラーメッセージ))

例:

```
{
 fnc(foo,
  throw("foo エラー")
 ),
 try(foo,msg,print(msg))
}
```

関数名:

lock

説明:

並列処理をロックする。ロックオブジェクトに使う変数を指定する。
lock 内の実行文はロック変数 ID が同じ lock がされていれば終了を待つ。
実行文の値が返る。

引数:

lock(ID(ロック変数 ID),式(実行文))

例:

```
{
 $tes = 1 ,
 for(cnt , 1 , 10 , 1 , prc("lock($tes,$tes = $tes + 1)")),
 waitresall,
 print($tes)
}
```

関数名:

setrpos

説明:

ウインドウを持つリソースの位置、サイズを設定します。

sizeX は 0 以下でサイズ無指定となります。

引数:

setrpos(式(リソース ID),式(sizeX),式(sizeY)[,式(X),式(Y)])

例:

```
{
 bindres(we,web("",0 , 0)), // 画面外表示
 setrpos(we, 64 , 128 , 128 ,64), // 画面表示
 waitres(we)
}
```

関数名:

addgrph

説明:

グラフを追加する。

位置は無指定で開いているところに追加、ラベル名は無指定で配列の

コメントが表示される。

引数:

addgrph(式(グラフリソース ID),ID(配列名)[,式(位置(0～))[,式(ラベル名)[,式(X 最小値),式(X 最大値)[,式(X 区切り単位)[,式(Y 最小値),式(Y 最大値)[,式(Y 区切り単位)]]]]]])

例:

```
{
 grres = grph($dat_all),
 addgrph(grres,$dat_sub)
}
```

関数名:

mkdir

説明:

ディレクトリを作成する。

相対パスは workfld 相対となる。

引数:

mkdir(式(ディレクトリ名))

例:

mkdir(“tmp”)

関数名:

rmdir

説明:

ディレクトリを削除する。

相対パスは workfld 相対となる。

引数:

rmdir(式(ディレクトリ名))

例:

rmdir(“tmp”)

関数名:

fprintenc

説明:

ファイルにエンコード指定で出力します。ファイル名にファイルのフルパス。値に出力文字列を指定するとファイルの最後に追加出力します。失敗した場合 0 を返します。成功した場合は１を返します。

UTF-8 を指定した場合 BOM 付で出力します。

ファイルパスが相対パスだった場合 workfld からのパスになります。

エンコード文字列は .NET の Encoding クラスに準じます。

引数:

fprintenc(式(ファイル名),式(エンコード文字列),式(出力内容),…)

例:

fprintenc(@”c:¥doc¥foo.txt”,”SJIS”,”aaa”,”bbb”)

“aaabbb” が c:¥doc¥foo.txt に SJIS で最後に追加されます。

関数名:

files

説明:

ディレクトリ内のファイル、ディレクトリリストを得る。相対パスは workfld 相対。ファイルリスト数を返す。

引数:

files(ID(ファイルリストを返す配列名),式(ディレクトリ名))

例:

```
files(tmp,” work”)
```

関数名:

isdir

説明:

指定パスがディレクトリか返す。相対パスは workfld 相対。1->フォルダ 0->フォルダでない

引数:

isdir(式(フォルダ名))

例:

```
isdir(“workfld”)
```

関数名:

appenddim

説明:

配列をつなげる(ID2 = ID2 + ID1)。マージではなく ID2 の配列に ID1 の１次元目のインデックスが被らないように数字インデックスを再割り当てする。

ID2 & ID1 として & 演算子を使用するのと同じですが、こちらはいったん文字列に変換しないので高速です。

ID2 の配列定義がない場合 ID1 の配列をコピーします。

ID1 には直接配列定義を記述できます。

式を省略か 0 で後ろにつなげる、1 で前につなげる。

引数:

appenddim(ID1(繋げる配列 ID),ID2(繋げられる配列)[,式(0->後ろにつなげる 1->前につなげる)])

例:

```
{
 for(cnt , 1, 1000 , 1 , appenddim([cnt,cnt*2,cnt*3]),tmp),
```

```
 for(cnt , 1, 1000 , 1 , { tmp2 = [1 , 2, 3],appenddim(tmp2,tmp)})
}
```

関数名:

ford

説明:

配列の繰り返し処理。インデックスを指定変数に入れながら最後までループします。

指定配列名の 1 次元目だけがループの対象になり、1 次元目のインデックスだけが変数名にセットされます。

引数:

ford (ID(変数名(インデックスが入る)),ID(配列名),式(実行式))

例:

```
{
 print(""),
 tmp = [[1, 2] , [3, 4], [5 ,6]],
 ford (tag,tmp,lprint(tag & ":"))
}
```

tmp の最初の次元だけが対象になり 0:1:2 が表示されます。

関数名:

timetosec

説明:

日付をグレゴリオ暦の 0001 年 1 月 1 日の午前 0 時 12:00:00 からの経過時間を表す秒に変換する

引数:

timetosec(式(時間))

例:

```
 timetosec(datetime)
```

関数名:

sectotime

説明:

グレゴリオ暦の 0001 年 1 月 1 日の午前 0 時 12:00:00 からの経過時間を表す秒を日付に変換する

引数:

sectotime(式(秒))

例:

```
sectotime(timetosec(datetime))
```

関数名:

stringbuffer

説明:

ストリングバッファリソースを作って ID を返す。文字列を & で結合するより高速。リソース識別文字列は "STR",rdump で文字列に変換する。= の代入だけで変数にバインドされます。

引数:

stringbuffer[(式(初期化文字列))]

例:

```
{
 strres = stringbuffer,
 appendstrbuf("foo",strres),
 print(rdump(strres)),
}
```

関数名:

appendstrbuf

説明:

ストリングバッファソースに文字列を足す。& より高速。位置は省略で末尾、その他は0始まりの指定位置。追加後の長さを返す

引数:

appendstrbuf(式(追加文字列),式(ストリングバッファリソース ID)[,式(位置)])

例:

```
{
 strres = stringbuffer,
 for(cnt , 1, 1000 , 1 ,
  {
   appendstrbuf(cnt,strres),
  }
 )
}
```

関数名:

dimidx

説明:

ID の配列の先頭を 0 とした数値位置からインデックス文字列を得る。

マイナスの値を指定すると-1 で最後尾となり最後から逆位置を指定する。

引数:

dimidx(ID(配列名),式(位置))

例:

```
{
 tmp = ["foo"::1, "taa"::2, "yaa"::3],
 lprint(dimidx(tmp, 0)),  // tmp の先頭のインデックスを表示
 tmp[dimidx(tmp, 0)] = 123,  // tmp の先頭を変更
 lprint(tmp[dimidx(tmp, 0)]) // tmp の先頭を表示
}
```

foo123 と表示される

関数名:

posinfo

説明:

情報画面を任意位置にスクロールする。

引数:

posinfo(式(Y 位置))

例:

```
posinfo(0)
```

関数名:

execscript

説明:

web リソース(Edge のみ)上でスクリプトを実行して結果を返す。

スクリプト最終行の実行帰り値を返します。

引数:

execscript(式(web リソース ID),式(スクリプト文字列))

例:

```
{
```

```
  bindres(we, web("https://google.com")), // web はバインド明記
  ret = execscript(we, "document.documentElement.outerHTML;"),
  print(ret)
 }
```

関数名:

winput

説明:

キー入力を待つ。キー判定は直後に rinput で行う。

引数:

winput

例:

```
{
 winput, // キー入力待ち
 print(rinput(^up)), // 上キー
 while(rinput(^up) == 1, 1) // 上キーが離されるまで待つ
}
```

関数名:

fcpy

説明:

ファイルをコピーする。

引数:

fcpy(式(コピー元ファイル名),式(コピー先ファイル名))

例:

```
fcpy(@"c:¥tmp¥foo.txt",@"c:¥tmp¥yaa.txt")
```

関数名:

fattr

説明:

ファイル属性を取得。取得できた場合 1、できなかった場合 0 を返す。

配列に属性を返す。

Directory : ディレクトリなら 1 以外 0

Hidden : 隠しファイルなら 1 以外 0

ReadOnly : 読み込み専用なら 1 以外 0

Name : ディレクトリを含まないファイル名

FullName ： ディレクトリを含むフルパス名
CreationTime ： 作成日時
LastWriteTime ： 最終更新日時
CreationTimeUtc ： 作成日時(UTC)
LastWriteTimeUtc ： 最終更新日時(UTC)
Length ： ファイル長さ(bytes)。ディレクトリの場合 0

引数:

fattr(ID(結果配列名),式(ファイル名))

例:

fattr(retatr, @”c:¥tmp¥foo.txt”)

関数名:

setuseragent

説明:

web リソースの UserAgent を指定する。成功すると 1 失敗 0 を返す。edge のみ対応

引数:

setuseragent(式(web リソース ID),式(UserAgent 文字列))

例:

```
{
 re = web,
 setuseragent(re, "Mozilla/5.0 (iPhone; CPU iPhone OS 14_5 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) CriOS/91.0.4472.80 Mobile/15E148 Safari/604.1"),
 seturl(re, "https://google.com"),
 waitresall
}
```

関数名:

getuseragent

説明:

web リソースの UserAgent を返す。edge のみ対応

引数:

getuseragent(式(web リソース ID))

例:

```
{
 re = web,
 print(getuseragent(re)),
 waitresall
}
```

関数名:

setbasicauth

説明:

web リソースにベーシック認証を設定する。成功すると 1 失敗 0 を返す。edge のみ対応

引数:

setbasicauth(式(web リソース ID),式(ユーザ名文字列),式(パスワード文字列))

例:

```
{
 re = web,
 setbasicauth(re, "test", "password"),
 seturl(re, "https://foo.com/foo.html"),
 waitresall
}
```

関数名:

cmpstr

説明:

式１と式２を文字列として比較します。

式１ < 式２ : -1
式１ == 式２ : 0
式１ > 式２ : 1

上記が返ります。

引数:

cmpstr(式 1,式 2)

例:

cmpstr(“150.50a”,150.5) // 1 が返ります

関数名:

cmpval

説明:

式１と式２を数値として比較します。

数値に変換できない文字列は 0 に変換されます。

式１ < 式 2 : -1

式１ == 式 2 : 0

式１ > 式 2 : 1

上記が返ります。

引数:

cmpval(式 1,式 2)

例:

cmpval (“150.50a”,150.5) // 0 が返ります

関数名:

cmp

説明:

式１と式２を文字列と数値を自動変換して比較します。

数値文字列の後ろに文字列がある場合は数値とみなされます。

式１, 式２のどちらかが文字列の場合文字列比較が行われます。

==,>,>=,<,<=,!= で比較する場合と同じです。

比較演算子では数値か文字列かを自動判定されますので、

“150.50a” == 150.5 は 1 が返ります。

式１ < 式 2 : -1

式１ == 式 2 : 0

式１ ＞ 式２ ： １

上記が返ります。

引数:

cmp（式 1,式 2）

例：

cmp（“150.50a”,150.5）// 0 が返ります

関数名:

lambda

説明:

ID の無名関数を得ます。

引数:

lambda(ID)

例：

```
{
 fnc(("tmp"), lprint(123)),
 fn = lambda(("tmp")),
 fn
}
```

関数名:

class

説明:

クラスの作成関数を定義します。

クラスメンバのアクセスには cls を使用してください。

クラス作成関数には自動で new が追加されます。

引数:

class(クラス作成関数 ID,定義内容)

例：

```
{
 ctx = context,
 class(ctx.CTes, // クラス作成関数指定
```

```
  {
   cls.foo = 123, // メンバ変数定義
   fnc(cls.fn, lprint(this.foo)), // メンバ関数定義
  }
 ),
 cl = ctx.newCTes,
 cl.fn
}
```

関数名:

ref

説明:

変数の定義文字列を返す。関数の場合も評価せず文字列を返す。

配列に対しても代入できる文字列を返しますが、クロージャ、this 参照は削除されます。

引数:

ref(ID)

例:

```
{
 fnc(tes,{123}),
 delfile("tes.txt"),
 fprint("tes.txt",ref(tes)), // tes の定義内容をファイル tes.txt に保存
 def(tes2,fload("tes.txt")), // ファイル tes.txt を読み込み tes2 を定義
 tes2
}

{
 tmp = [1, 2],
 fnc(tmp[2],123),
 delfile("tes.txt"),
 fprint("tes.txt", ref(tmp)),  // 配列 tmp の代入可能な文字列を得る
 tmp2 = fload("tes.txt"), // 配列をファイルから読み込み設定
 tmp2[2]
}
```

・ 動作環境

Windows10,Windows8.1 の動作確認を行っています。

C# の .NET Framework 4.7.2 のインストールが必要です。

Edge(chromium ベース)のインストールが必要です。

WebView2 runtime のインストールが必要です。

変更履歴

Ver 1.22 : use のサンプルを修正。

Ver 1.23 : grph を ID 化。len 対応

Ver 1.24 : amean,gmean,hmean 対応

Ver 1.25 : med 対応

Ver 1.26 : hmed 対応

Ver 1.27 : dis 対応(v.0.16.29.0)

Ver 1.28 : hamean,hgmean,hhmean 対応(v.0.16.30.0)

Ver 1.29 : hdis 対応(v.0.16.31.0)

Ver 1.30 : 設定読み込み(部分),関数部分保存対応対応(v.0.16.32.0)

Ver 1.31 : cov 対応 (v.0.16.33.0)

Ver 1.32 : cor 対応。関数定義ダイアログ更新高速化 (v.0.16.34.0)

Ver 1.33 : waitres,sqr 対応(v.0.16.35.0)

Ver 1.34 : reskind で CNT 追加 (v.0.16.35.0)

Ver 1.35 : rdump,setrcom,getrcom 追加 (v.0.16.40.0)

Ver 1.36 : cpdim 追加 (v.0.16.41.0)

Ver 1.37 : ログフォルダ表示、分割対応(v.0.16.43.0)

Ver 1.38 : cload,solve 対応 (v.0.16.47.0)

Ver 1.39 : mgdim 対応(v.0.16.48.0)

Ver 1.40 : 配列インデックス任意文字列対応 (v.0.16.53.0)

Ver 1.41 : 棒グラフ対応(v.0.16.55.0)

Ver 1.42 : 関数定義ダイアログページ対応(v.0.16.56.0)

Ver 1.43 : エディタ内コマンド対応 (v.0.16.58.0)

Ver 1.44 : eval 対応 (v.0.16.59.0)

Ver 1.45 : 初期化、終了関数追加(v.0.16.60.0)

Ver 1.46 : workfld 対応。Fload,fprint,delfile,existfile 相対パス作業フォルダ指定対応 (v.0.17.1.0)

Ver 1.47 : 配列アクセス[]対応(v.0.17.2.0)

Ver 1.48 : bindres 対応(v.0.17.3.0)

Ver 1.49 : =代入対応(v.0.17.4.0)

Ver 1.50 : (…,…)シーケンス対応(v.0.17.5.0)

Ver 1.51 : 比較演算子対応 (v.0.17.6.0)

Ver 1.52 : 関数定義ダイアログ複数表示対応。 (v.0.17.7.0)

Ver 1.53 :右クリックメニュー対応(v.0.17.8.0)

Ver 1.54 :prc 引数対応 (v.0.17.9.0)

Ver 1.55 :エラーダイアログ編集対応 (v.0.17.10.0)

Ver 1.56 :setcom,getcom 対応 (v.0.17.11.0)

Ver 1.57 :dbg 対応 (v.0.17.12.0)

Ver 1.58 :xpathdim,regexdim 対応 (v.0.17.14.0)

Ver 1.59 :getpos,mvpos 対応 (v.0.17.15.0)

Ver 1.60 : htmlencode,htmldecode,urlencode,urldecode,rchgret 対応 (v.0.17.16.0)

Ver 1.61 :多次元配列 fordim 対応(v.0.17.17.0)

Ver 1.62 :sort,reverse 対応(v.0.17.18.0)

Ver 1.63 :split 対応(v.0.17.19.0)

Ver 1.64 :web,seturl,sethtml,gethtml 対応(v.0.17.20.0)

Ver 1.65 :誤記修正 (v.0.17.20.0)

Ver 1.66 :誤記修正 (v.0.17.20.0)

Ver 1.67 :改行修正 (v.0.17.26.0)

Ver 1.68 :数値と文字列の自動変換記載(v.0.17.26.0)

Ver 1.69 :bindres 再バインド時削除対応 (v.0.17.27.0)

Ver 1.70 :sort 誤記修正(v.0.17.27.0)

Ver 1.71 :aline 対応 (v.0.17.30.0)

Ver 1.72 :グラフ種別自動判定対応 (v.0.17.31.0)

Ver 1.73 :callcnt 対応(v.0.17.33.0)

Ver 1.74 :動的 ID 生成対応。代入=での左辺が関数の注意事項記載。(v.0.17.34.0)

Ver 1.75 :fmax,fmin 対応。 (v.0.17.37.0)

Ver 1.76 :未定義参照エラー対応。 (v.0.17.38.0)

Ver 1.77 :代入前関数非評価対応。 (v.0.17.39.0)

Ver 1.78 ://コメント対応。内部関数名使用不可能追記。 (v.0.17.42.0)

Ver 1.79 :コンテキスト内評価修飾子対応(v.0.17.44.0)

Ver 1.80 :sort,fmax,fmin this 対応 (v.0.17.49.0)

Ver 1.81 :fnc,fncdim 対応 (v.0.17.51.0)

Ver 1.82 :isbind 対応 (v.0.17.52.0)

Ver 1.83 :エディタ補完対応 (v.0.17.54.0)

Ver 1.84 :エディタ補完グローバル変数内コンテキストリソース対応 (v.0.17.55.0)

Ver 1.85 :json,mkjson 対応(v.0.17.57.0)

Ver 1.86 :unbindres 対応(v.0.17.58.0)

Ver 1.87 :datetime 文字列対応(v.0.17.59.0)

Ver 1.88 :xml,mkxml 対応 (v.0.17.62.0)

Ver 1.89 :net ^post 対応 (v.0.17.63.0)

Ver 1.90 :cpycnt 対応 (v.0.17.64.0)

Ver 1.91 :try 対応 (v.0.17.66.0)

Ver 1.92 :throw 対応 (v.0.17.68.0)

Ver 1.93 :リソース情報メニュー対応,PRC 制限解除 (v.0.17.69.0)

Ver 1.94 :lock 対応 (v.0.17.71.0)

Ver 1.95 :edge 対応 (v.1.0.0.0)

Ver 1.96 :IE 使用メニュー対応 (v.1.0.1.0)

Ver 1.97 :setrpos,web 画面外表示対応 (v.1.0.5.0)

Ver 1.98 :xpathdim 属性対応(v.1.0.8.0)

Ver 1.99 :エリア表示非表示対応 (v.1.0.9.0)

Ver 1.100 :addgrph 対応 (v.1.0.11.0)

Ver 1.101 :addgrph 修正(v.1.0.14.0)

Ver 1.102 :grph,addgrph パラメータ省略対応 (v.1.0.15.0)

Ver 1.103 :net ^bin ,mkdir,rmdir 対応(v.1.0.16.0)

Ver 1.104 :fload エンコード対応 (v.1.0.17.0)

Ver 1.105 :fprintenc 対応 (v.1.0.18.0)

Ver 1.106 :files,isdir 対応 (v.1.0.19.0)

Ver 1.107 :[] 配列初期化,配列値返し対応 (v.1.0.20.0)

Ver 1.108 :配列引数対応 (v.1.0.21.0)

Ver 1.109 :文字列エスケープタブ対応 (v.1.0.23.0)

Ver 1.110 :配列帰り値直接参照対応 (v.1.0.25.0)

Ver 1.111 :シーケンス {} 対応(v.1.0.27.0)

Ver 1.112 :配列&連結対応,appenddim 対応(v.1.1.0.0)

Ver 1.114 :ford 対応 (v.1.1.2.0)

Ver 1.115 :~対応 (v.1.1.7.0)

Ver 1.116 :<- 対応 (v.1.1.8.0)

Ver 1.117 :timetosec,sectotime 対応 (v.1.1.9.0)

Ver 1.118 :=>対応 (v.1.1.10.0)

Ver 1.119 :# 定義を変更(v.1.1.11.0)

Ver 1.120 :fmin,fmax,sort,<-,eval,=> 呼び出し元変数参照可能,無名関数追記(v.1.1.12.0,v0.18.12.0(IE))

Ver 1.121 :use 廃止(v.1.1.15.0,v0.18.15.0(IE))

Ver 1.122 :stringbuffer,appendstrbuf 対応 (v.1.1.16.0,v0.18.16.0(IE))

Ver 1.123 :グラフマウス操作対応(v.1.1.18.0,v0.18.18.0(IE))

Ver 1.124 :IE 版廃止(v.1.2.0.0)

Ver 1.125 :input ENTER 決定対応 (v.1.2.3.0)

Ver 1.126 :グラフマウス移動説明追加(v.1.2.4.0)

Ver 1.127 :,スペース自動挿入追加 (v.1.2.5.0)

Ver 1.128 :全体の古い記法を更新(v.1.2.7.0)

Ver 1.129 :resval 廃止 (v.1.2.9.0)

Ver 1.130 :cor,cov オフセット指定対応(v.1.2.11.0)

Ver 1.131 :履歴対応 (v.1.2.14.0)

Ver 1.132 :関数定義ダイアログ参照定義対応(v.1.2.15.0)

Ver 1.133 :WEB リソースキャッシュクリア対応(v.1.2.16.0)

Ver 1.134 :誤記修正 (v.1.2.17.0)

Ver 1.135 :dbg 配列展開対応(v.1.2.18.0)

Ver 1.136 :関数呼び出しスキップ対応 (v.1.2.23.0)

Ver 1.137 :appenddim 前に追加対応,dimidx 追加(v.1.2.24.0)

Ver 1.138 :配列内::指定対応 (v.1.2.25.0)

Ver 1.139 :dbg ダブルクリックコンテキスト表示対応(v.1.2.28.0)

Ver 1.140 :posinfo 対応。sort 無名関数 this 未設定追記 (v.1.2.32.0)

Ver 1.141 :ワークフォルダ変更対応(v.1.2.33.0)

Ver 1.142 :dbg 変数内容表示追加 (v.1.2.35.0)

Ver 1.143 :execscript 対応(v.1.2.36.0)

Ver 1.144 :グラフポイントラベル非表示対応(v.1.2.37.0)

Ver 1.145 :マウスホイール履歴移動対応 (v.1.2.39.0)

Ver 1.146 :winput 対応。Webview2 インストーラ付属廃止 (v.1.2.45.0)

Ver 1.147 :コンテキスト内関数の無名関数 this 対応 (v.1.2.47.0)

Ver 1.148 :fattr,fcpy 対応(v.1.2.50.0)

Ver 1.149 :strlen,substr 全角を２文字とする対応 (v.1.2.55.0)

Ver 1.150 :ワークフォルダ変更で初期化対応(v.1.2.56.0)

Ver 1.151 :net(^html)で UserAgent 指定対応 setuseragnet,getuseragent,setbasicauth 対応(v.1.2.71.0)

Ver 1.152 :配列編集画面で詳細編集対応 (v.1.2.78.0)

Ver 1.153 :solve 無名関数対応(v.1.2.92.0)

Ver 1.154 :無名関数のコンテキスト参照永続化対応 (v.1.2.93.0)

Ver 1.155 :bindres 複数変数対応。unbindres 廃止(v.1.2.94.0)

Ver 1.156 :cmpstr 追加 (v.1.2.96.0)

Ver 1.157 :比較演算子&対応(v.1.2.97.0)

Ver 1.158 :cmpval,cmp 対応(v.1.2.98.0)

Ver 1.159 :delres,delresall リソース強制削除追記 (v.1.2.106.0)

Ver 1.160 :bindresref 対応(v.1.2.107.0)

Ver 1.161 :bindresref,callcnt 削除、バインド多変数デフォルト化 GC 対応 (v.2.0.0.0)

Ver 1.162 :文脈修正(v.2.0.3.0)

Ver 1.163 :バインド変数代入でバインド対応 (v.2.0.12.0)

Ver 1.164 :バインド説明追加(v.2.0.16.0)

Ver 1.165 :context,stringbuffer 代入でのバインド説明追加(v.2.0.17.0)

Ver 1.166 :context,stringbuffer 代入でのバインド変更修正 (v.2.0.17.0)

Ver 1.167 :lambda,class 追加。lambda,class 説明追加 (v.2.0.24.0)

Ver 1.168 :代入なしコンテキスト GC 対応(v.2.0.25.0)

Ver 1.169 :class サンプル追加 (v.2.0.25.0)

Ver 1.170 :callcnt 対応(v.2.0.31.0)

Ver 1.171 :unbindres 対応(v.2.0.32.0)

Ver 1.172 :cpdim,cpycnt 自己 this 参照コピー先明示 (v.2.0.41.0)

Ver 1.173 :ref 対応 (v.2.0.42.0)

Ver 1.174 :cpycnt を cpcnt に変更(v.2.0.43.0)

Ver 1.175 :エラーダイアログ全文表示対応 (v.2.0.46.0)

Ver 1.176 :エラーダイアログデバッグダイアログ表示対応 (v.2.0.47.0)

Ver 1.177 :dbg 全文表示ボタン対応(v.2.0.53.0)

Ver 1.178 :dbg 編集ボタン対応(v.2.0.54.0)