萌系スクリプト言語

ツインテール de エンジェルモード !!
スクリプト言語のまにゅある

2up
印刷

（ こすちゅーむ２３０～対応版 ）

©Kasugaさん

目次

0．ツインテール de エンジェルモード !! とは

まずは、簡単な自己紹介から…。

真面目で堅いイメージのある電子計算機プログラミングの世界に、ほんの少しだけ萌要素を取り入れたオープンソース萌系スクリプト言語です。

でも、そんな見かけと違って、中身はとっても真面目なスクリプト言語となっています。

インタプリタのソースプログラムを見なくても、その内部処理（＝Ｃ言語での記述）を人間が想像しやすくなっていて、そのうえ、シンプルでスモールなので、ご主人様との距離は急接近します。

元々は、航空宇宙ミッション解析用に設計された特定用途スクリプトエンジンから派生したものでした。（←前世）　なので、少しだけ理系エンジニア向けのスクリプト言語となっています。

言語の特徴を３行で表現すると…。

1、　伝統的なＵＮＩＸ系スクリプト言語の流れ…。
2、　自然・基本・簡潔、文字タイプ量が少ない…。
3、　言語自体のハッキングが容易。（仕様改変を標準サポート）

気に入ってくれるとうれしいにゃん♪

著作者と連絡先

- 著作権表示と作者連絡先：
  ツインテールdeエンジェルモード!!
  Copyright (C) 2011.08.15-2018 "ねこみみ(♀)"<twintail@angelmode.net>

- ライセンス：
  GPL か LGPL の自由選択

- 利用の画像：
  http://ja.wikipedia.org/wiki/Wikipedia:ウィキぺたん © Kasuga さん

１．スクリプトの全体構成

用語の説明：

- スクリプトとは、この言語で記述されたプログラムのことです。
- スクリプトは、「コード領域」と「データ領域」から構成されます。

（ａ） コード領域

コード領域とは、スクリプトを記述する領域です。デフォルトでは、入力ファイルの全体がコード領域となります。インタプリタ tt は、コード領域に記述されたスクリプトを評価＆実行します。

コード領域の開始位置を明示する場合は、:code 又は __code__ の単独行で指定します。終了位置は、データ領域の開始位置、又は、ファイル末となります。

【例】何も指定しなければ、スクリプト全体がコード領域となります。

```
print("こんにちは、ご主人様♪\n")
```

（ｂ） データ領域

データ領域とは、埋め込みデータを記述するためのオプション領域です。インタプリタ tt は、データ領域に記述されたテキストを評価＆実行しません。

データ領域の開始位置は、:data 又は __data__ の単独行で指定します。終了位置は、ファイル末となります。ファイルポインタ :data を利用すると、データ領域を読み込むことができます。

【例】データ領域の指定例です。データ領域には必ず指定が必要です。

```
print("こんにちは、ご主人様♪\n")
:data                           # データ領域の開始 ← この行の記述は必須です。
10
20
30
```


【例】コード領域とデータ領域を指定したスクリプトの例です。
　ここでは、埋め込みデータの合計値を計算＆表示するスクリプトを記述してみます。

```
:code                                   # コード領域の開始 ← この行の記述は任意です。
      sum=0                             # ←（最初に実行される行）
      while( line=gets(:data) )         # 埋め込みデータを１行づつ読み込む（ファイル末まで）
          sum+=int(line)                # 各行を整数化して変数 sum に加える
      sum                               # 合計値 sum を簡易表示する

:data                                   # データ領域の開始 ← この行の記述は必須です。
      10                                # ←（最初に読込される行）
      20
      30
```

【実行】例示コードをテキストファイル "example" に保存して実行してみます。
　埋め込みデータの合計値が表示されます。

```
% tt example
60
```

（ｃ） コメント行

文字 # から行末までと、文字列 // から行末まではコメントとなります。同様に、/* ～ */ の内側も、コメントとなります。（シェル方式＋Ｃ言語方式）

また、データ領域もコメントとして利用できます。

２．データ型

用語の説明：

- 基本データ型とは、この言語が取り扱う基本的なデータ型のことです。（５種類）
- 派生データ型とは、基本データ型を組み合わせたデータ型のことです。（配列型と構造体）
- カッコ内の記号（ 緑色１文字 → U/S/I/D/P/X/A ）は、そのデータ型の略称です。

（ａ） 基本データ型［５種類］

文 字 列（S）　【例】 x = "こんにちは、ご主人様!!\n"
　　【例】 x = <<RAW文字列>>
　　【例】 x = /[a-z]+/

技術メモ：

- " 文字列 " では、Ｃ言語と同じエスケープシーケンスが使えます。　　（ 例： "\n"　）
  又、"\xFF" のパターンで、任意の１バイト HEX データを埋め込みます。（ 例： "\x0a" ）
- <<文字列>> では、複数行にわたるRAW文字列（無変換文字列）が記述できます。
- /正規表現/ では、POSIX正規表現を保持します。（そのまま、文字列としても使用できます。）

整 数 型（I）　【例】 x = 1024　or　'A'　　（ 参考ビット幅 = 64[bit]　）
実 数 型（D）　【例】 x = 3.14　or　+1.000E-99　　（ 参考ビット幅 = 64[bit]　）
ポインタ（P）　【例】 x = NULL　or　stdin　　（ 参考ビット幅 = 64[bit]　）

技術メモ：

- 文字列とポインタの内部表現は、それぞれ char* と void* となります。又、整数型と実数型の内部表現は、それぞれ intptr_t と double となります。
- 整数の表現では、"0[bB]"、"0[oO]"、"0[dD]"、又は、"0[xX]" のプリフィックスが指定可能です。（それぞれ、{ 2| 8|10|16}進数を表現します。）

関 数 型（X）　【例】 x = &print()　or　print

技術メモ：

- 変数又は関数 x の定義の判定には、関数 isdef(x) が使用できます。又、それらを未定義化するには、関数 erase(x) が使用できます。
- 変数又は関数 x の型を調べるには、関数 type(x) が使用できます。なお、初期化されてない場合は、未定義（U）となります。

（ｂ） 派生データ型［２種類］

配 列 型（A）の構文：配列名 [添字,添字,添字,,,] = 値
構 造 体（A）の構文：構造体名 . メンバ　　= 値

- 配　列は、要素に値を代入することで自動的に定義されます。配 列 名は、配　列実体へのリファレンスを保持する変数となります。
- 構造体は、要素に値を代入することで自動的に定義されます。構造体名は、構造体実体を保持する変数となります。
- 配列の添字は、基本データをコンマ区切りで指定します。又、構造体のメンバは、英数字（先頭は英字のみ）をドット区切りで指定します。いずれも、次元数（区切りの数）は任意です。

【例】いろいろな値で、配列と構造体を定義してみます。（代入する値は、何でもOKです。）

```
a[0] = "This is Array"              # 文字列
a[1] = 100                          # 整数型
a[2] = 3.14                         # 実数型
a[3] = stderr                       # ポインタ
a[4] = argv                         # 他の配　列
pos.x=640 ; pos.y=480 ; a[5]=pos    # 他の構造体
a[999] = &print()                   # 関数型
```

【例】多次元配列 c を定義してみます。次元数の分だけ、添字をコンマ区切りで指定して下さい。
ここでは、最初2次元配列を定義して、次に3次元配列の成分も追加してみます。（異次元共存）

```
# 2次元
   loop( i<1000 )                   # i = 0,1,2, ～ ,999
         c["tt", i] = NULL          # 各要素を NULL で初期化します。
# 3次元
   loop( i<10 )                     # i = 0,1,2, ～ ,9
         loop( j<10 )               # j = 0,1,2, ～ ,9
               c["tt", i, j] = i*j  # 各要素を i*j で初期化します。
```

コピー時の動作

配列をコピーすると、データ本体へのリファレンス情報のみがコピーされます。よって、コピー後はデータ本体の共有状態となります。（２つの配列名から同じ要素にアクセスできます。）

一方、構造体をコピーすると、リファレンス情報とデータ本体の一式全てがコピーされます。

配列リテラルの記述方法

記述方法１：　添字の　０・１・２・３…　に相当する値を、順番に記述します。
　　{ と } をネストすることにより、多次元配列も記述可能です。
　　→　　{ 値０ , 値１ , 値２ , 値３ , … }

記述方法２：　添字と値のペアを => 記号でつなげて、順番に記述します。
　　→　　{ 添字=>値 , 添字=>値 , 添字=>値 , … }

【例】配列リテラルを使い、配列を定義してみます。
　次に示す文は、いずれも右辺で配列が生成され、それが左辺の変数に代入されます。

```
a = { "Red", "Green", "Blue" }            # 記述方法1：一次元配列
b = { { 1, 2 }, { 3, 4 }, { 5, 6 } }      # 記述方法1：二次元配列
c = { "Jan"=>31, "Feb"=>28, "Mar"=>31 }   # 記述方法2
```


ベクトルと行列

　ベクトルと行列は、それぞれ1次元配列と2次元配列で表現します。通常の配列としての取り扱いに加えて、四則演算、積算(@)、内積(<>)、外積(><)、及び、転置(~)の演算ができます。もちろん、スカラー（通常の値）との演算も可能です。

【例】ベクトルと行列を定義して、各種演算を行ってみます。（スカラーは、全要素に作用します。）

```
x = { 1, 2, 3 }                   # ベクトルの定義
y = { 4, 5, 6 }                   # ベクトルの定義

ret = x+3                         # ret = {  4,  5,  6 }
ret = x-3                         # ret = { -2, -1,  0 }
ret = x*y                         # ret = {  4, 10, 18 }
ret = x/y                         # ret = {  0,  0,  0 }

ret = x<>y                        # ret = 32             ← 内積
ret = x><y                        # ret = { -3,  6, -3 } ← 外積

m = { { 1, 2, 3}, { 4, 5, 6} }    # 2x3 行列の定義
n = { { 2, 4}, { 6, 8}, {10,12} } # 3x2 行列の定義

ret = m@n                         # ret = {{44,56},{98,128}} ← 行列の積
```

３．変数

（ａ） システム変数

用語の説明：

- システム変数とは、システムによって事前に定義された global変数 のことです。
  詳しくは、別冊マニュアル「**システム変数とシステム関数**」をご覧下さい。

（ｂ） ユーザー変数

定　義：

- 変数に値を代入することで、自動で定義されます。
- 変数名は、英数字（先頭は英字）に限ります。

型と値：

- 型と値に制限はありません。全ての変数は、任意型のデータの保持と上書きができます。

属　性：

- 定義時にキーワード {global|static|local} を前置すると、スコープと記憶クラス（変数の寿命）の指定ががができます。（デフォルトでは、local変数となります。）
- 各変数の属性（スコープと記憶クラス）は、次表の通りとなります。

変数の種類	スコープ	記憶クラス（変数の寿命）
global 変数	グローバルスコープ	静的（スクリプト終了まで）
static 変数	ローカル　スコープ	静的（スクリプト終了まで）
local　変数	ローカル　スコープ	動的（Local 関数終了まで）

【例】グローバル変数・スタティック変数・ローカル変数を明示的に定義してみます。

```
global x=10                    # グローバル　変数 x の定義
static y=20                    # スタティック変数 y の定義
local  z=30                    # ローカル　　変数 z の定義
```

４．関数

（ａ） システム関数

用語の説明：

- システム関数とは、システムによって事前に定義された 組み込み関数 のことです。
  詳しくは、別冊マニュアル「**システム変数とシステム関数**」をご覧下さい。

（ｂ） ユーザー関数

関数の定義： def 関数名(仮引数リスト){ スクリプト本文 } ← 単文の場合は、{} を省略可

- 仮引数リストは、仮引数名をコンマ区切りで記述します。
- 戻り値は retn() 文を用いてコンマ区切りで記述します。

関数の実行： 戻り値 = 関数名(実引数リスト)

- 実引数リストは、実引数値をコンマ区切りで記述します。
- 戻り値が複数ある場合は、左辺を () で括ってコンマ区切りで記述します。

【例】階乗を計算するユーザー関数 fact() を定義＆実行してみます。
　ここでは、再帰呼び出しで定義してみます。

```
def fact(x){                              # ユーザー関数の定義（パラメータ１つ）
       retn((x==0||x==1) ? 1:x*fact(x-1)) # 再帰呼び出し
}

loop(i<10)                                # ユーザー関数の実行 ( i = 0,1,2, 〜 ,9 )
       print("%d => %f\n", i, fact(i))    # 結果の表示
```

【実行】例示コードをテキストファイル "example" に保存して実行します。

```
% tt example
0 => 1.000000
1 => 1.000000
2 => 2.000000
3 => 6.000000
...省略...
```


【例】２つの数値の和差積商（４つ）を返すユーザー関数 cal4() を定義＆実行してみます。

```
def cal4(x,y){                            # ユーザー関数の定義（パラメータ２つ）
       retn(x+y,x-y,x*y,x/y)              # 戻り値（４つ）は、コンマ区切りで指定します。
}

(add,sub,mul,div) = cal4(2,3)             # ユーザー関数の実行
print("%d %d %d %d\n",add,sub,mul,div)    # 結果の表示
```

【実行】例示コードをテキストファイル "example" に保存して実行します。

```
% tt example
5 -1 6 0
```

技術メモ：

- ユーザー関数は、TopLevel でのみ定義が可能です。又、全て global関数 となります。
- ユーザー関数は、任意データの {値渡しと値戻し} ／再帰呼び出し／前方参照と後方参照／任意個数の戻り値…、などに対応しています。※
  ※ 注意：配列型はリファレンスなので、結果的に参照渡しと参照戻しになります。
- 引き数や戻り値の個数が不一致の場合は、余った値は無視され不足する値は未定義となります。

変数と関数の関係

変数に関数を代入する時は、`変数 = &関数()` 又は `変数 = 関数` の形式で記述します。また、変数に代入された関数を実行する時は、`変数(引数)` の形式で記述します。

変数の定義時に同名の関数があった時は、デフォルトでは（同名の関数への上書きは行われずに）ローカル変数が新規で作成されます。関数への上書き行う時は、明示的にグローバル変数として定義して下さい。

【例】ローカル変数 v に指数関数を代入して実行してみます。

```
v = &exp()                              # 関数の代入
print("%f\n",v(1))                      # 代入した関数の実行と表示
```

【実行】例示コードをテキストファイル "example" に保存して実行します。

```
% tt example
2.718282
```

【例】三角関数を別の値（文字列、整数、関数）で上書きしてみます。

```
global sin = "Hello, Master!!\n"        # 関数への上書き（文字列）
global cos = 100                        # 関数への上書き（整数型）
global tan = &exp()                     # 関数への上書き（関数型）
```


関数の再定義

関数を再定義する時には、同名の関数で再度定義して下さい。再定義された「元の関数」は、アンダースコア "_" を前置した名前で参照や実行が可能となります。

【例】指数関数を再定義してみます。

```
def exp(x){ retn(_exp(x)+1000) }        # 関数の再定義
print("%f\n",exp(1))                    # 再定義した関数の実行
```

【実行】例示コードをテキストファイル "example" に保存して実行します。

```
% tt example
1002.718282
```

５．文法

（ａ） 演算子と型変換

- 【参考】演算子と型変換を一言で説明すると…、Ｃ言語とほぼ同等です。
  （注意箇所を赤字で表示しています。）

数値　演算子：＋[加算] －[減算] * [乗算] / [除算] % [余剰] **[累乗] ++[Incr] --[Decr]
　　：@ [行列の積] <> [ベクトルの内積] >< [ベクトルの外積]
ビット演算子：~ [反転] & [AND] | [IOR] ^ [XOR] <<[左シフト] >>[右シフト]
論理　演算子：！[否定] &&[AND] ||[IOR] ^^[XOR]
比較　演算子：> [値大] < [値小] >=[以上] <=[以下] ==[等しい] !=[異なる]
マッチ演算子：~~[マッチ肯定判定]　!~[マッチ否定判定]　@~[マッチ位置検出]
その他演算子：= [代入演算] ?:[３項演算] ()[まとめ括弧] , [コンマ演算]
OP= 型演算子：+=　-=　*=　/=　など

【例】マッチ位置を検出します。

```
(s,e) = "ABCDEFG" @~ /CDE/
# s=2  マッチ開始位置
# e=5  マッチ終了位置（＝非マッチ開始位置）
```

技術メモ：　文字列の演算について

- 文字列＋文字列　→ 文字列の連結
- 文字列＋整数値　→ 文字列と文字（整数値）の連結
- 文字列＊整数値　→ 文字列の伸長（整数倍）
- 文字列［整数値］→ 配列アクセス（整数値番目の文字を表します。先頭＝０、末尾＝－１）

技術メモ：　数字同士の演算について

- 整数 と 整数　　→ 整数
- 片方でも実数　　→ 実数

技術メモ：

- NULL は数　値との演算時には、整数値の０として扱われます。
- NULL は文字列との演算時には、文字列 ""（空文字列）として扱われます。

（ｂ） 条件分岐

```
if()  , elif(), else      { 説明略 }
swch(), case: , deft:     { 説明略 }     別名 switch(), default:
```

技術メモ：

- swch() や case: のパラメータとして、任意の基本型データをコンマ区切りで複数個指定可能です。
  又、記号 ＊ がワイルドカードとして使用可能です。

【例】swch() 文の例。２つのパラメータを一度に判別します。パラメータの型は自由です。

```
swch( OS , Lang ){
      case "Linux","cc": result="燃え"; break;   # OS=="Linux" && Lang=="cc" の時
      case    *   ,"tt": result="萌え"; break;   #                Lang=="tt" の時
      deft:              result="不明";          # デフォルト
}
```

（ｃ） 繰り返し

```
for( ; ; )              { 説明略 }
while()/do_while()      真の間、本文を実行 [ do_while() は、本文実行後の条件判定 ]
until()/do_until()      偽の間、本文を実行 [ do_until() は、本文実行後の条件判定 ]
each( X = A  )          配列Ａの各要素を、順次ループ変数 X に代入して本文を実行
loop( X< VAL )          for( X=0 ; X< VAL ; X++ ) に等価 [ ループ変数 X は自動定義 ]
loop( X<=VAL )          for( X=0 ; X<=VAL ; X++ ) に等価 [ ループ変数 X は自動定義 ]
```

【例】コマンドライン引数の一覧を表示します。

```
each( x = argv ){               # 配列 argv[] の各要素を順次 x に代入してループ
      p(x)                      # 変数の簡易表示
}
```

```
redo  redo()            現ループを再度実行する
next | next()           現ループの次に進む ( 別名 = continue )
last | last()           現ループを終了する ( 別名 = break    ) 、又は、swch() 文を終了する
retn | retn()           ユーザ関数から戻る ( 別名 = return   )
```

6 . APPENDIX - ハッキング

スクリプト言語「ツインテール de エンジェルモード !!」の改変方法について、具体例を用いて簡単に説明します。ここでのハッキング対象は、キーワード（予約語）・システム変数・システム関数とします。

いずれの場合であっても、変更後はインタプリタ本体の再コンパイルが必要となることに注意してください。

(a) キーワード（予約語）

0. 前提知識 { 例：elif }

キーワードは、字句解析フェーズにおいて、変数名や関数名を認識する直前のタイミングで、関数 chk_rsvdkeywd() によって判定されます。(syntax/fsup.c)

例えば、キーワード elif ならば、同関数内の次の if 文によって判定が行われます。

```
if( strcmp(str,"elif")==0 ){ f("[ELIF]"); return(ELIF); }
```

関数 chk_rsvdkeywd() の戻り値は、認識したキーワードの種別を表すトークン番号（整数値 ELIF ）となります。構文解析フェーズにおいては、その整数値によってキーワードの種別が区別＆認識されます。

なお、…

関数 f() & 関数 b() は、インタプリタ本体のデバッグ用メッセージ出力関数です。

オプション -F 指定時に、字句解析フェーズのデバッグ用メッセージが出力されます。【 関数 f() 】
オプション -B 指定時に、構文解析フェーズのデバッグ用メッセージが出力されます。【 関数 b() 】

これらの関数は、インタープリタの通常動作には影響を与えませんので、コードリーディング時には考慮する必要はありません。（削除してしまっても問題ありません。）

1. 名前の変更 { 例：elif → elseif }

次の様に、判定文を変更します。

```
//      if( strcmp(str,"elif"  )==0 ){ f("[ELIF]"); return(ELIF); }     // 削除
        if( strcmp(str,"elseif")==0 ){ f("[ELIF]"); return(ELIF); }     // 変更
```

2. 別名の登録 { 例：elseifの登録 }

次の様に、判定文を追加します。

```
        if( strcmp(str,"elif"  )==0 ){ f("[ELIF]"); return(ELIF); }
        if( strcmp(str,"elseif")==0 ){ f("[ELIF]"); return(ELIF); }     // 追加
```

3. 内容の変更 { 例：elifの変更 }

文法を変更したい時は、ファイル syntax/bison.y 内の ELIF 使用箇所を変更します。
→ これは、結構難しいです。

処理を変更したい時は、ファイル icode/lang.flow/x_ifswch.c 内の関数 x_elif() を変更します。
→ これは、意外と簡単です。

4. 新規の登録 {}

他のキーワードに倣って、ファイル syntax/fsup.c 及び syntax/bison.y に記述を追加します。
前者はキーワード自体の設定で、後者はそれに対応する文法の設定です。

又、新規のキーワードに対応する処理関数を icode/lang.flow 追加し、その宣言を admin/extern.h に追加します。これは、実際の処理内容の記述となります。

最後に、exec/toplvl.c に処理関数を実行する記述を追加します。これらの記述によって、「キーワードの認識→文法の認識→抽象構文木の生成→処理の実行」という処理ができることになります。

5. 登録の削除 { 例：elifの削除 }

次の様に、判定文を削除します。

```
//      if( strcmp(str,"elif"  )==0 ){ f("[ELIF]"); return(ELIF); }     // 削除
```

これで十分なのですが、関連する定義や宣言も消した方が、よりスッキリします。

(b) システム変数

0. 前提知識 { 例 : M_PI }

システム変数の実体は、定義済みのグローバル変数です。その定義は、次のインストール文によって行われます。(ファイル admin/inst-syscnst.c 関数 inst_syscnst())

```
wr_dtab(GL_DTAB,"M_PI",'D',0,(tdbl)M_PI);
```

この文の意味は次の通りです。すなわち…

「スクリプトのGL_DTAB {グローバルスコープ領域} に、"M_PI"という識別子をインストールする。 なお、識別子の型を'D' {実数型} 、識別子の属性を 0 {大文字小文字の区別有り} 、識別子の値を (tdbl)M_PI とする。」

ここで、tdbl は、インタプリタ内部で使用している実数型の型名称となります。同様に、tint は、インタプリタ内部で使用している整数型の型名称となります。又、M_PI の値ですが、/usr/include/math.h の中で 3.14159265358979323846 と定義されています。(環境依存)

これにより、スクリプトから識別子 M_PI が参照されると、実数型の値 3.14159265358979323846 が返されることとなります。(これは、円周率です。)

1. 名前の変更 { 例 : M_PI → PI }

次の様に、インストール文を変更します。

```
//      wr_dtab(GL_DTAB,"M_PI",'D',0,(tdbl)M_PI);               // 削除
        wr_dtab(GL_DTAB,"PI"  ,'D',0,(tdbl)M_PI);               // 追加
```

2. 別名の登録 { 例 : PIの登録 }

次の様に、インストール文を追加します。

```
        wr_dtab(GL_DTAB,"M_PI",'D',0,(tdbl)M_PI);
        wr_dtab(GL_DTAB,"PI"  ,'D',0,(tdbl)M_PI);               // 追加
```

3. 内容の変更 { 例 : 実数 → 整数 }

次の様に、インストール文を変更します。

```
//      wr_dtab(GL_DTAB,"M_PI",'D',0,(tdbl)M_PI);               // 削除
        wr_dtab(GL_DTAB,"M_PI",'I',0,(tint)M_PI);               // 変更
```

4. 新規の登録 { 例 : Authorの登録 }

まず、追加する識別子(システム変数の名前)と、データ型と、データ値を決定します。識別子の構成文字が正しいこと、名前の衝突が発生しないこと、データ型とデータ型の間に矛盾がないことに注意して下さい。

ここでは、新しいシステム変数 Author に、文字列 "NekoMimi(F)" を割り当ててみます。
次の様に、インストール文を追加します。

```
        wr_dtab(GL_DTAB,"Author",'S',0,"NekoMimi(F)");          // 追加
```

5. 登録の削除 { 例 : M_PIの削除 }

次の様に、インストール文を削除します。

```
//      wr_dtab(GL_DTAB,"M_PI",'D',0,(tdbl)M_PI);               // 削除
```

(c) システム関数

0. 共通知識 { 例 : tan() }

スクリプトから見えるシステム関数名 { この場合は、tan() } を外部名と呼び、 外部名に対応した、インタプリタ内部のC言語関数名 { この場合は、i_tan() } を内部名と呼びます。

すなわち、システム関数 tan() は、C言語関数 i_tan() が実際の処理を行っています。

内部名のC言語関数は、外部名に "i_" を前置したものとなりスクリプトからは見えません。又、全て非破壊的に動作し入力を書き換えません。なお、入力データ＆出力データは全て構文解析木の形式にて受け渡しが行われます。

内部名と外部名のつながりは、次のインストール文によって関連付けが行われます。(ファイル admin/inst-sysfunc.c 関数 inst_sysfunc())

```
wr_dtab(GL_DTAB,"tan",'X',0,i_tan,1);
```

この文の意味は次の通りです。すなわち…、

「スクリプトのGL_DTAB {グローバルスコープ領域} に、"tan"という識別子（外部名）をインストールする。
なお、識別子の型を'X' {システム関数} 、識別子の属性を０ {大文字小文字の区別有り} 、識別子に対応するC言語関数（内部名）を i_tan()、識別子が期待する引数の個数を１個とする。」

※ 以下の説明にある、「宣言部」と「定義部」と「インストール部」の具体的な箇所は以下の通りです。

- 宣言部： ファイル admin/extern.h
- 定義部： ディレクトリ icode 以下の各ファイル
- インストール部： ファイル admin/inst-sysfunc.c （外部名と内部名の関連付け）

また、内部名のC言語関数が利用する、構文解析木の定義箇所は以下の通りです。

- 解析木： ファイル admin/mdtype.h 内の struct ctree
- ノード： ファイル admin/mdtype.h 内の struct dtab

1. 名前の変更 { 例 : sin() → sine() }

外部名を sin() -> sine() に変更します。インストール部の１ヶ所。
内部名を i_sin() -> i_sine() に変更します。宣言部と定義部とインストール部の３ヶ所。

```
//      wr_dtab(GL_DTAB,"sin" ,'X',0,i_sin ,1);          // 削除
        wr_dtab(GL_DTAB,"sine",'X',0,i_sine,1);          // 変更
```

2. 別名の登録 { 例 : cosine()の登録 }

外部名が cosine() で、内部名が i_cos() となるインストール文を追加します。（インストール部）

```
        wr_dtab(GL_DTAB,"cos"   ,'X',0,i_cos,1);
        wr_dtab(GL_DTAB,"cosine",'X',0,i_cos,1);         // 追加
```

3. 内容の変更 { 例 : tan()の変更 }

外部名 tan() に対応する、C言語関数 i_tan() の処理内容（定義部）を変更します。
具体的なコードは、希望する変更内容に依存します。※

※ 類似する既存関数のコードを参照して下さい。

4. 新規の登録 { 例 : vvv() }

外部名 vvv() に対応する、C言語関数 i_vvv() の処理内容（定義部）を記述します。
具体的なコードは、希望する変更内容に依存します。※

※ 類似する既存関数のコードを参照して下さい。

内部名 i_vvv() に対応する宣言を追加します。（宣言部）
外部名が vvv() で、内部名が i_vvv() となるインストール文を追加します。（インストール部）

```
        wr_dtab(GL_DTAB,"vvv",'X',0,i_vvv,-1);           // 新規追加（可変引数の例）
```

5. 登録の削除 { 例 : rad()の削除 }

定義部にて i_rad() 関数本体を削除し、宣言部とインストール部の記述も削除します。

```
//      wr_dtab(GL_DTAB,"rad",'X',0,i_rad,1);            // 削除
```