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

（ こすちゅーむ２４０～対応版 ）

2up
印刷

© Kasugaさん

目次

０．ツインテール de エンジェルモード !! とは

（ａ） まずは、超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。

（ｂ） 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）

（ｃ） 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:ウィキペたん © Kasuga さん

気に入ってくれるとうれしいにゃん♪

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．データ型とデータ値

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。（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。） また、IPアドレスが記述可能です。
  【例】ip = 127.0.0.1 （これは、ip = 0x7F000001 と同じです。）

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．文法規則と制御構文

（ａ） スクリプトの記述と実行

スクリプトはコード領域領域に書かれた「文(statement)」を、先頭から順に実行します。

文は、原則として1行に1文づつ記述しますが、セミコロン ； を記述することで、1行に複数の文を記述することもできます。逆に、行末にバックスラッシュ ¥ を記述することで、1つの文を複数の行に渡って記述することもできます。

なお、行末のセミコロン ； は不要です。（あっても、問題はありません。）

（ｂ） 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。（先頭位置は０となります。）

```
(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
- なお、整数又は実数が期待される場所に数値（整数又は実数）が与えられると、期待される型（整数又は実数）に自動型変換されます。

技術メモ： NULL について

- 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。

（ｃ） 条件分岐と繰り返し

```
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
}
```

```
for(;;)                   { 説明略 }
loop(X<V) | loop(X<=V)    for(X=0;X<V;X++) | for(X=0;X<=V;X++) に等価 [ X は自動定義 ]
while()   | do_while()    真の間、本文を実行 [ do_while() 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
```

【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
}
```

```
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  ）
```


（ｄ） 変数の文字列展開

変数の値を文字列内に展開する場合は、文字列中において "${変数名}" と記述します。

【例】変数の文字列展開を行います。

```
x = 3.14
n = NULL
s = "x=${x} n=${n}"             # 文字列 s 内に変数 x n の値を展開

s                               # 文字列 s の簡易表示
```

技術メモ：

- 展開時の書式は、システム変数 {FMT_S2S | FMT_I2S | FMT_D2S | FMT_P2S | FMT_N2S} の設定値に従います。（これらは、自由に変更可能です。）

４．ユーザー変数とユーザー関数

用語の説明：

- ユーザー変数とは、ユーザーによって定義された変数です。
- ユーザー関数とは、ユーザーによって定義された関数です。（全てglobal関数となります。）

（ａ） 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
```


（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．システム変数とシステム関数

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 です。
- システム関数とは、システムによって事前に定義された global関数 です。
  → システム変数とシステム関数は、その値や内容を自由に変更することができます。

（ａ） 凡例と注釈

記号の説明

このマニュアルで使用する記号です。

- S,I,D,P,X,A → データ型を表します。
- FILE → ファイル名を表す文字列、又は、ファイルポインタ を表します。
- /RE/ → 正規表現 を表す文字列、又は、コンパイル済み正規表現を表します。
- ＊ → 任意のデータ型を表します。
- [X] → X が省略可能であることを表します。（配列記号とは文脈で区別します。）
- X|Y → X 又は Y が選択可能であることを表します。

共通の性質

「システム関数」には、次の共通した性質があります。

- 常に非破壊的に動作します。（入力データを破壊しません。）
- 常に全 自 動で動作します。（メモリ管理やファイル管理は、自動で行います。）
- エラー戻り値は NULL で統一されています。

（ｂ） 基本パラメーター

＜システム変数＞

基本の定数

名前	型	意味	備考
NULL	P	ポインタ NULL	関数のエラー戻り値
TRUE	I	真の値（実体は整数１）	
FALS	I	偽の値（実体は整数０）	
VER	I	インタプリタのバージョン番号	= こすちゅーむ番号
PID	I	自プロセスIDの値	= $$
※ VERとPIDは参照用です。変更してもスクリプトの動作に影響を与えません。			

変数のビット幅

名前	型	意味	備考
INT_SIZE	I	整数のビット数	例：64［bit］
DBL_SIZE	I	実数のビット数 ［ = DBL_MANTSIZE + DBL_EXPOSIZE + 1 ］	例：64［bit］
PTR_SIZE	I	ポインタのビット数	例：64［bit］
DBL_MANTSIZE	I	実数の仮数部のビット数	例：52［bit］
DBL_EXPOSIZE	I	実数の指数部のビット数	例：11［bit］
※ 全て参照用です。変更してもスクリプトの動作に影響を与えません。			

＜システム関数＞

型の種別と型の判定：

名前	説明
I = type（＊）	引数の型を1文字で返します。 戻り値={'U'|'S'|'I'|'D'|'P'|'X'|'A'}
I = isnull(＊)|istrue(＊)|isfals(＊)| isstr (＊)|isint (＊)|isdbl (＊)| isptr (＊)|isfunc(＊)|isarry(＊)|	引数が {NULL｜TRUE｜FALS｜文字列｜整数｜実数｜ポインタ｜関数|配列} であるかどうか判定します。 戻り値 ={TRUE|FALS}

定義判定と未定義化：

名前	説明
I = isdef(＊)	引数が定義済みであるかどうか判定します。 戻り値={TRUE|FALS}
U = erase(＊)	引数を未定義化します。 戻り値=無し

（ｃ） コマンドライン引数

＜システム変数＞

コマンドライン引数：

名前	型	意味	備考
argc	I	コマンドライン引数の数　（コマンド名も含む）	
argv	A	コマンドライン引数の配列（コマンド名も含む）	
argr	I	現時点でshift()可能なコマンドライン引数の残数	自動更新
args	S	argv[1] からargv[argc-1]まで連結した文字列	
cmd	S	コマンド名の文字列	= argv[0]
env	A	環境変数の配列※	

※ 例えば、環境変数が PATH=/usr/bin であった場合は、env["PATH"]="/usr/bin" となります。

＜システム関数＞

コマンドライン引数：

名前	説明
S = shift　([*])	コマンドライン引数を、呼び出し毎に argv[1] から順に取得する
S = opshift([*])	（ 〃 ）　ただし、対応する引数が option の場合にのみ取得する
I = unshift()	次の取得位置を１つ前に戻す（戻り値＝取得位置）

※ shift() 及び opshift() は、取得するコマンドライン引数が無い場合は、指定されたパラメータ ＊（又は、その指定も無い場合 NULL ）が戻り値となります。
※ オプションとは '-' 文字で始まるコマンドライン引数のことです。

（ｄ） ファイル入出力

＜システム変数＞

ファイルポインタ：

名前	型	意味	備考
stdin	P	標準入力	
stdout	P	標準出力	
stderr	P	標準エラー出力	
:code	P	コード領域アクセス用	= __code__
:data	P	データ領域アクセス用	= __data__

※ :code 及び :data は、通常ファイルに記述されたスクリプト内でのみアクセス可能です。

＜システム関数＞

１文字、又は、１行の入出力：

名前	説明
I = getc　([FILE]) S = gets　([FILE]) S = getn　([FILE,]　Max)	指定されたファイルから、1文字、又は、1行を読み込みます。 （ FILE 省略時は、標準入力を対象とします。）
I = putc　([FILE,] Chr) I = puts　([FILE,] Str) I = putn　([FILE,] Str,Max)	指定されたファイルへ、　1文字、又は、1行を書き込みます。 （ FILE 省略時は、標準出力を対象とします。）
I = ungetc([FILE,] Chr) I = ungets([FILE,] Str) I = ungetn([FILE,] Str,Max)	指定されたファイルへ、　1文字、又は、1行を戻します。 （ FILE 省略時は、標準入力を対象とします。）

※ 文字はChr(I)、文字列はStr(S)、最大文字数はMax(I)で指定します。

バイナリーデータの入出力：

名前	説明
(Ptr,Cnt) = read ([FILE,]　Cnt)	指定バイト数の読み込み。 （ FILE 省略時は、標準入力を対象とします。）
Cnt　= write([FILE,] Ptr,Cnt)	指定バイト数の書き込み。 （ FILE 省略時は、標準出力を対象とします。）

※ Cnt(I)は、要求バイト数と実行バイト数です。又、Ptr(P)は、対象データの保存場所アドレスです。

各種制御関数：

名前	説明
P = open　(FILE,Mode)	ファイルの明示的オープンです。（通常は不要です。）
P = close (FILE　)	ファイルの明示的クローズです。（通常は不要です。）
P = flush (FILE　)	ファイルバッファのフラッシュです。（NULL指定=全ファイル）
I = getpos(FILE　)	ファイル位置の取得です。（現在の位置、先頭=00　　）
I = setpos(FILE,I)	ファイル位置の設定です。（絶対値指定、先頭=00／末尾=-1）
I = movpos(FILE,I)	ファイル位置の移動です。（相対値指定、進行=正／後退=負）
I = rewind(FILE　)	ファイル先頭位置(00)への移動です。 [= setpos(FILE,0)]

※ Mode(S) は、Ｃ言語 fopen() 互換のモード文字列です。

（ｅ） データ表示（印刷出力）

＜システム変数＞

文字列の変換書式：

名前	型	意味	備考
FMT_S2S	S	文字列　→文字列 への変換書式	初期値 = "%s"
FMT_I2S	S	整数型　→文字列 への変換書式	初期値 = "%d"
FMT_D2S	S	実数型　→文字列 への変換書式	初期値 = "%+f"
FMT_P2S	S	ポインタ→文字列 への変換書式（NULL以外）	初期値 = "%p"
FMT_N2S	S	NULL　→文字列 への変換書式（NULLのみ）	初期値 = "NULL"
※ 全てデフォルトの変換書式です。 p() 関数や、"${変数}" 構文による変数展開などで利用されます。			

＜システム関数＞

簡易表示と詳細表示：

システム関数	説明
void = p(*, ...)	変数、又は、式の値を簡易表示します。
void = d(*, ...)	変数、又は、式の値を詳細表示します。　← ダンプ表示
x　← 変数名のみを記述した場合。	変数 x の値を簡易表示します。　← p(x) に同じ
※ 詳細表示では、識別ＩＤ｛スコープ種別(G=Global|S=Static|L=Local)＋識別番号｝、名前、データ型、データ属性、データ値、といった内部管理情報を表示します。	

書式付き出力：

システム関数	説明
I = print (FMT,...)	標準　　出力用の print 文です。
I = eprint(FMT,...)	標準エラー出力用の print 文です。 = fprint(stderr,FMT,...)
I = fprint(FILE,FMT,...)	ファイル　出力用の print 文です。
S = sprint(FMT,...)	文字列　　出力用の print 文です。
void = dying(FMT,...)	この関数は { eprint(FMT,...); exit(1); } と等価です。
※ FMT(S) は、Ｃ言語 printf() 互換の書式文字列です。拡張機能として2進数出力 %b %B、及び、数値出力（整数又は実数を、その型に応じて %d %f %e 又は %E を自動選択する）%v %V が指定できます。	

（ｆ） 文字列操作

＜システム関数＞

文字列の比較：

システム関数	説明
I = strcmp (S1,S2) I = strcmpi(S1,S2)	文字列 S1 と S2 の大小比較です。
I = strncmp (S1,S2,I) I = strncmpi(S1,S2,I)	文字列 S1 と S2 の大小比較です。（最大 I 文字まで）
※ 戻り値は、Ｃ言語互換です。又、i 付き関数は Ignore Case 版（大文字小文字の区別なし）です。	

文字列の検索：

システム関数	説明
I = strchr(S,Chr) | strrchr(S,Chr)	文字　Chr の｛順方向｜逆方向｝検索です。
I = strstr(S,Str) | strrstr(S,Str)	文字列 Str の｛順方向｜逆方向｝検索です。
(Is,Ie) = strreg(S,/RE/)	正規表現 /RE/ のマッチ位置検索です。 {Is=始点、Ie=終点}
※ いずれの関数も、文字列 S 内の位置を戻り値とします。（先頭＝０）	

文字列の置換；

システム関数	説明
S = ssub(S,V1,V2) | gsup(S,V1,V2)	文字列 S 内の V1 を V2 に｛１回｜全部｝置換します。
S = evalesc(S)	ESCシーケンスを順方向変換します。（例："¥n"→0x0a）
S = rvalesc(S)	ESCシーケンスを逆方向変換します。（例：0x0a→"¥n"）
S|I = uc(S|I) | lc(S|I)	引数を｛大文字化｜小文字化｝します。
※ パラメータ V1 には、文字、文字列、又は、正規表現が指定できます。 ※ パラメータ V2 には、文字、文字列　　が指定できます。	

改行等の削除：

システム関数	説明
S = chop　(S)	{　　行末　　} の改行コードを１組削除します。
S = hstrip(S)| tstrip(S)| strip(S)	{行頭｜行末｜両方} の空白コードを全部削除します。
※ 改行コードは、Win/Mac/Unix 形式に対応します。空白コードは、isspace() で判定します。	

部分文字列等：

システム関数	説明
I = subchr(S,Ic)	文字列 S の位置 Ic の文字を戻します。（ = S[Ic] ）
I = substr(S,Is,Ie)	文字列 S の位置 Is～Ie の部分文字列を戻します。
※ 位置の指定方法（先頭から指定する場合）： 先頭 = 0, 1, 2, … ※ 位置の指定方法（末尾から指定する場合）： 末尾 = -1,-2,-3, …	

文字列の配列化：

システム関数	説明
A = split(S,V [,I])	文字列 S を V で区切った各要素で配列化します。
A = psplit(S,V [,I])	（ 〃 ） ただし、空要素はスキップします。（ Packed形式 ）
A = scan (S,V [,I])	文字列 S を V でマッチさせた各要素で配列化します。
A = pscan (S,V [,I])	（ 〃 ） ただし、空要素はスキップします。（ Packed形式 ）
※ いずれの関数も、パラメータ V には、文字、文字列、又は、正規表現が指定できます。又、戻り値 A は、新規に生成された１次元配列です。（添字は０～） 第３パラメータ(I)指定時は、その添字値に対応する要素(S)のみを戻します。	

（ｇ） データ操作

＜システム関数＞

データの変換：

システム関数	説明
I　= int (＊)	パラメータを整数値化します。【 別名 atoi() 】
D　= dbl (＊)	パラメータを実数値化します。【 別名 atof() 】
I|D = atov(＊)	パラメータを数　値化します。（戻り値の型は自動判定）
S　= str (＊)	パラメータを文字列化します。
P　= ptr (＊)	パラメータをポインタ化します。
I = bin(S)| oct(S)| dec(S)| hex(S)	文字列 S を{2|8|10|16}進数と見なして整数化します。
/RE/ = reg(S)	文字列 S の正規表現をコンパイルします。

データの判定：

システム関数	説明
I = isalnum(I) | isalpha(I) | isascii(I) | isblank(I) | iscntrl(I) | isdigit(I) | isgraph(I) | islower(I) | isprint(I) | ispunct(I) | isspace(I) | isupper(I) | isxdigit(I)	Ｃ言語互換の文字種別判定関数です。　戻り値={TRUE|FALS}

データの消去：

システム関数	説明
U = erase(＊)	定義済み変数を消去します。（未定義化）

データの複製：

システム関数	説明
＊ = dup(＊)	パラメータを複製します。なお、配列又は（静的変数を持つ）関数以外では、代入文と同じ効果となります。

サイズの計測：

システム関数	説明
I = size(＊)	パラメータのサイズを求めます。
I = strlen(S)	文字列 S の長さを求めます。（バイト数）
※ 関数 size() の戻り値は、文字列→長さ（バイト数）、整数型＆実数型＆ポインタ→表現ビット数、関数型→演算ノード数、配列型→要素数となります。	

ソートの実行：

システム関数	説明
A =　sort([X,] V,...)	パラメータ V を昇順にソートします（戻り値＝ソート済み配列）
A = rsort([X,] V,...)	パラメータ V を**降順**にソートします（戻り値＝ソート済み配列）
※ 比較関数 X を指定しない場合は、パラメータの単純ソートとなります。この場合、パラメータ V は、{**文字列**｜**整数**｜**実数**} 又はそれらを含む配列が指定できます。比較関数 X を指定した場合は、比較関数による一般ソートとなります。この場合、パラメータ V は自由に設定出来ます。なお、比較関数 X は、２つの引数を取り比較結果を数値の符号で戻す関数となります。（Ｃ言語 qsort(3) の比較関数互換。）	

配列化と配列値：

システム関数	説明
Ao = vals(Ai)	配列 Ai の各要素値を要素とする、１次元配列 Ao を生成します
Ao = keys(Ai)	配列 Ai の各添字値を要素とする、１次元配列 Ao を生成します
I　= ndim (Ai)	（ハッシュ要素を含まない）配列 Ai の次元数を求めます
Ao = shape(Ai)	（ハッシュ要素を含まない）配列 Ai の形　状を求めます

（ｈ） ネットワーク

0. ＵＤＰ

＜典型的な処理フロー＞

1、ソケットの作成とバインド

→ 省略可能です。省略時はソケットが必要になったタイミングで自動で作成され、アドレス＆ポート番号も自動で設定されます。ただし、サーバー用プログラムではポート番号を指定するのが一般的だと思います。

```
sock = udp_socket(アドレス,ポート番号)
```

2、回線の接続

→ UDPなので回線の接続は不要です。

3、送受信の実行

→ 送信時は、送信先のアドレス＆ポート番号と送信データを指定して送信します。

```
tx_udp( ip, port, "Hello, Master!!" )
```

→ 受信時は、送信元のアドレス＆ポート番号と受信データが得られます。

```
( ip, port, buf ) = rx_udp()
```

4、その他（ソケットの指定について）

→ 省略可能です。UDPの送受信関数は、ソケット値が指定された場合はその値を利用しますが、指定が省略された場合は UDP_SOCKET の値（デフォルトソケット値）を利用します。なお UDP_SOCKET は、最後に作成されたUDPソケットの値が自動で設定＆更新されます。

＜サンプルプログラム＞

【例sv】UDPエコーサーバーを構築します。具体的には、UDPのポート７番（echo）で待ち受け受信をして、そこで受信したデータをそのまま送り元に送信（返答）します。

```
s = udp_socket( 0, "echo" )              # ポート番号を指定してソケットを作成します。
do_while( buf!=NULL ){
       ( ip, port, buf ) = rx_udp()      # UDPで受信します。
       ret = tx_udp( ip, port, buf )     # UDPで送信（返答）します。
}
```

★解説：ローカルポート番号を指定するために、関数 udp_socket() を実行します。
なお、作成されたUDPソケットの値 s は、UDP_SOCKET にも自動で設定されます。

関数 rx_udp() は、ソケット値が省略されているため UDP_SOCKET の値を利用します。
関数 tx_udp() も、同様です。

【例cl】UDPエコークライアントを構築します。具体的には、ローカルホスト上のUDPエコーサーバーに挨拶メッセージを送信して、サーバーからの送信（返答）メッセージを受信して表示します。

```
ip   = 127.0.0.1
port = 7
tx_udp( ip, port, "Hello, UDP!!" )       # UDPで挨拶メッセージを送信します。
( ip, port, buf ) = rx_udp()             # UDPで返答メッセージを受信します。

print("[%s]¥n",buf)                      # 受信したメッセージを表示します。
```

★解説：関数 tx_udp() は、ソケット値が省略されているため UDP_SOCKET の値を利用します。
しかし、UDP_SOCKET は初期値０（無効）であるため、利用直前にUDPソケットが自動で作成され UDP_SOCKET に設定されます。（なお、関数 tx_udp() は設定後の値を利用します。）

関数 rx_udp() は、ソケット値が省略されているため UDP_SOCKET の値を利用します。

【実行】例示コードをテキストファイル "sv" と "cl" に保存して実行します。
なお、サーバーの動作を確認するためクライアントを3回連続で実行します。

```
% tt sv &
[1] 1234
% tt cl ; tt cl ; tt cl
[Hello, UDP!!]
[Hello, UDP!!]
[Hello, UDP!!]
```

＜システム変数＞

UDP関連のパラメーター：

名前	型	意味	備考
UDP_SOCKET	I	UDPソケット省略時に使用するUDPソケット番号	自動設定＆自動更新
UDP_TIMEOUT	D	UDP送受信関数のタイムアウト値	単位［秒］ ※
※ UDP_TIMEOUT の初期値は INF（無限ブロッキング）です。例えば、３を設定すると３秒後にタイムアウトします。また、０を設定するとノンブロッキング動作となります。			

＜システム関数＞

UDPソケットの作成：

名前	説明
sock = udp_socket([ip,port])	指定された ip(I|S) & port(I|S) にて、UDP送受信用のソケットを作成します。なお、数値0を指定すると自動設定となります。 戻り値は、作成されたソケット番号 sock(I) です。
※ 作成されたソケット番号は、デフォルトソケット UDP_SOCKET(I) に設定されます。	

UDP送信とUDP受信：

名前	説明
len = tx_udp([sock,]ip,port,buf[,len])	ソケット sock(I) を利用して、送信先 ip(I|S) & port(I|S) に、buf(S) のデータを len(I) バイト、UDPパケットで送信します。なお、len(I) 省略時は buf(S) の全データが送信されます。 戻り値は、送信したUDPデータ部のバイト数 len(I) です。 ※
(ip,port,buf,len) = rx_udp([sock])	ソケット sock(I) を利用して、自分宛のUDPパケットを受信します。 戻り値は、送信元 ip(I) & port(I)、受信したUPDデータ buf(S)、及び、そのバイト数 len(I) です。 ※
※ ソケット sock(I) 省略時には、デフォルトソケット UDP_SOCKET(I) が使用されます。	

1. TCP

＜典型的な処理フロー＞

1、ソケットの作成とバインド

→ 省略可能です。省略時はソケットが必要になったタイミングで自動で作成され、アドレス＆ポート番号も自動で設定されます。ただし、サーバー用プログラムではポート番号を指定するのが一般的だと思います。

```
sock = tcp_socket(アドレス,ポート番号)
```

2、回線の接続

→ サーバー側は listen() で待ち受けします。

```
sock = listen ()
```

→ クライアント側は connect() で接続します。

```
sock = connect()
```

3、送受信の実行

→ 送信時は、送信データを指定して送信します。

```
tx_tcp( "Hello, Master!!" )
```

→ 受信時は、受信データが得られます。

```
buf = rx_tcp()
```

4、その他（ソケットの指定について）

→ 省略可能です。TCPのコネクト関数と送受信関数は、ソケット値が指定された場合はその値を利用しますが、指定が省略された場合は TCP_SOCKET の値（デフォルトソケット値）を利用します。なお TCP_SOCKET は、最後に作成されたTCPソケットの値が自動で設定＆更新されます。

→ 一方、TCPのリスン関数は、ソケット値が指定された場合はその値を利用しますが、指定が省略された場合は前回リスンしたソケットの値を利用します。もし、それも無い場合は、TCP_SOCKET の値を利用します。

＜サンプルプログラム＞

【例sv】TCPエコーサーバーを構築します。具体的には、TCPのポート７番（echo）で待ち受け受信をして、そこで受信したデータをそのまま送り元に送信（返答）します。

```
s = tcp_socket( 0, "echo" )              # ポート番号を指定してソケットを作成します。
while( s_new=listen(s) ){                # クライアントからの接続要求をリスンします。
       buf = rx_tcp()                    # TCPで受信します。
       ret = tx_tcp(buf)                 # TCPで送信（返答）します。
}
```

★解説：ローカルポート番号を指定するために、関数 tcp_socket() を実行します。
なお、作成されたTCPソケットの値 s は、TCP_SOCKET にも自動で設定されます。

関数 listen() は、ソケット s にて接続要求を待ち受けます。クライアントからの接続要求が到着してTCP回線の接続が確立すると、接続済みの新しいソケットが生成され、その値を戻り値とします。また、この新しいソケット値 s_new は、TCP_SOCKET にも自動で設定されます。

関数 rx_tcp() は、ソケット値が省略されているため TCP_SOCKET の値を利用します。
関数 tx_tcp() も、同様です。

【例cl】TCPエコークライアントを構築します。具体的には、ローカルホスト上のTCPエコーサーバーに挨拶メッセージを送信して、サーバーからの送信（返答）メッセージを受信して表示します。

```
ip   = 127.0.0.1
port = 7
connect(ip,port)                         # TCPサーバーに接続します。
tx_tcp( "Hello, TCP!!" )                 # TCPで挨拶メッセージを送信します。
buf = rx_tcp()                           # TCPで返答メッセージを受信します。

print("[%s]¥n",buf)                      # 受信したメッセージを表示します。
```

★解説：関数 connect() は、ソケット値が省略されているため TCP_SOCKET の値を利用します。
しかし、TCP_SOCKET は初期値０（無効）であるため、利用直前にTCPソケットが自動で作成され TCP_SOCKET に設定されます。（なお、関数 connect() は設定後の値を利用します。）

関数 rx_tcp() は、ソケット値が省略されているため TCP_SOCKET の値を利用します。
関数 tx_tcp() も、同様です。

【実行】例示コードをテキストファイル "sv" と "cl" に保存して実行します。
なお、サーバーの動作を確認するためクライアントを3回連続で実行します。

```
% tt sv &
[1] 1234
% tt cl ; tt cl ; tt cl
[Hello, TCP!!]
[Hello, TCP!!]
[Hello, TCP!!]
```

＜システム変数＞

TCP関連のパラメーター：

名前	型	意味	備考
TCP_SOCKET	I	TCPソケット省略時に使用するTCPソケット番号	自動設定＆自動更新
TCP_TIMEOUT	D	TCP回線接続関数＆TCP送受信関数のタイムアウト値	単位［秒］　※
※ TCP_TIMEOUT の初期値は INF（無限ブロッキング）です。例えば、３を設定すると３秒後にタイムアウトします。また、０を設定するとノンブロッキング動作となります。			

＜システム関数＞

TCP回線の接続：

名前	説明
sock = listen([sock])	ソケット sock(I) を利用して、TCP回線の接続を待ち受けします。 戻り値は、新たに生成された接続済みのソケット番号 sock(I) です。　※1
sock = connect([sock,]ip,port)	ソケット sock(I) を利用して、指定された ip(I|S) & port(I|S) に、TCP回線の接続を行います。 戻り値は、接続済みのソケット番号 sock(I) です。　※2
※1 ソケット sock(I) 省略時には、前回リンスしたソケットが再び使用されます。もし、それが無い場合は、デフォルトソケット TCP_SOCKERT(I) が使用されます。 ※2 ソケット sock(I) 省略時には、デフォルトソケット TCP_SOCKET(I) が使用されます。	

TCPソケットの作成：

名前	説明
sock = tcp_socket([ip,port])	指定された ip(I|S) & port(I|S) にて、TCP送受信用のソケットを作成します。なお、数値0を指定すると、自動設定となります。 戻り値は作成されたソケット番号 sock(I) です。
※ 作成されたソケット番号は、デフォルトソケット TCP_SOCKET(I) に設定されます。	

TCP送信とTCP受信：

名前	説明
len = tx_tcp([sock,]buf[,len])	ソケット sock(I) を利用して、接続先に buf(S) のデータを len(I) バイト、TCPパケットで送信します。なお、len(I) 省略時は buf(S) の全データが送信されます。 戻り値は送信したTCPデータ部のバイト数 len(I) です。　※
(buf,len) = rx_tcp([sock])	ソケット sock(I) を利用して、自分宛のTCPパケットを受信します。 戻り値は、受信したTCPデータ buf(S)、及び、そのバイト数 len(I) です。　※
※ ソケット sock(I) 省略時には、デフォルトソケット TCP_SOCKET(I) が使用されます。	

2. RAW（PING）

＜典型的な処理フロー＞

1、ソケットの作成とバインド

→ 省略可能です。省略時はソケットが必要になったタイミングで自動で作成され、アドレスも自動で設定されます。

```
sock = raw_socket(アドレス)
```

3、ping() の実行

→ 送信先のアドレスを指定して ping() を実行します。

```
ping( ip )
```

4、その他（ソケットの指定について）

→ 省略可能です。ping() 関数は、ソケット値が指定された場合はその値を利用しますが、指定が省略された場合は RAW_SOCKET の値（デフォルトソケット値）を利用します。なお RAW_SOCKET は、最後に作成されたRAWソケットの値が自動で設定＆更新されます。

＜サンプルプログラム＞

【例】www.google.com に ping を実行します。

```
ip = "www.google.com"                    # 送信先を設定します。
tm = ping( ip )                          # ping を実行します。

print("%f [ms]¥n",tm*1000)               # 往復時間を表示します。
```

【実行】例示コードをテキストファイル "cl" に保存して実行します。

```
% tt cl
9.119000 [ms]
```


＜システム変数＞

RAW（PING）関連のパラメーター：

名前	型	意味	備考
RAW_SOCKET	I	RAWソケット省略時に使用するRAWソケット番号	自動設定＆自動更新
RAW_TIMEOUT	D	RAW送受信関数のタイムアウト値	単位［秒］ ※

※ RAW_TIMEOUT の初期値は INF（無限ブロッキング）です。例えば、３を設定すると３秒後にタイムアウトします。また、０を設定するとノンブロッキング動作となります。

＜システム関数＞

PING送信とPING受信：

名前	説明
ret = ping(ip)	指定された ip(I|S) に ICMP ECHO を送信し、戻って来た ICMP ECHOREPLY を受信します。 戻り値は、往復時間 ret(D) です。[単位=秒]
※ ソケット sock(I) 省略時には、デフォルトソケット UDP_SOCKET(I) が使用されます。	

3. その他

＜システム関数＞

ソケットの情報：

名前	説明
(lo_ip,lo_sip,lo_port,lo_sport,lo_type,re_ip,re_sip,re_port,re_sport,rer_type) = sockinfo(sock)	ソケット sock(I) の情報を戻り値とします。情報には以下のものが含まれます。ローカルIPアドレス(I)と文字列(S)、ローカルport(I)と文字列(S)、ソケットタイプ(S)、並びに、リモートIPアドレス(I)と文字列(S)、リモートport(I)と文字列(S)、ソケットタイプ(S)。なお、ソケットタイプは、{"RAW"|"UDP"|"TCP"} となります。

IPアドレスとPORT番号：

名前	説明
S = ip2str(I|S)	IPアドレス又はホスト名(I|S)を文字列に変換します。
I = ip2int(I|S)	IPアドレス又はホスト名(I|S)を整数値に変換します。
S = port2str(I|S)	ポート番号又はサービス名(I|S)を文字列に変換します。
I = port2int(I|S)	ポート番号又はサービス名(I|S)を整数値に変換します。

（ｉ） 日付と時刻

【 参 考 】
・UNIX時刻 とは、1970-01-01 00:00:00 からの経過時刻［単位＝実数秒］のことです。
・UTC とは、協定世界時のことです。すなわち、世界時間のことです。
・LTZ とは、ローカルタイムゾーン設定に基づく地方時間のことです。（例：日本標準時）

＜システム変数＞

タイムゾーンと時差：

名前	型	意味	備考
UTC	S	協定世界時を表す ”UTC” の文字列	
LTZ	S	ローカルタイムゾーンを表す文字列	日本の例：”JST”
LTZ_OFFSET	I	UTCを基準としたLTZ時刻の時差（秒）	日本の例：+32400
※ LTZ と LTZ_OFFSET の初期値は、インタプリター起動時に OS より自動設定します。LTZ_OFFSET の値を変更することにより、スクリプト内の時差を自由に設定することが出来ます。なお、これらの値を変更したとしても OS には影響を与えません。			

デフォルトの変換書式：

名前	型	意味	備考
FMT_SDATE	S	{年月日への｜年月日からの} 変換書式	初期値 = ”%Y-%m-%d”
FMT_STIME	S	{時分秒への｜時分秒からの} 変換書式	初期値 = ”%H:%M:%S”
※全て、Ｃ言語 {strftime(3)｜strptime(3)} 互換の変換書式となっています。			

＜システム関数＞

現在時刻（=UNIX時刻）の取得：

システム関数	説明
D = time()	現在の UNIX時刻 を取得します。【単位＝実数秒】
※ UNIX時刻は、国や地域などに依存しない世界共通の時刻となります。（時差調整や夏時間調整が存在しません。）又、単一の数値で表現できる利便性もあるため、ツインテールdeエンジェルモード!!に含まれる「日付と時刻」の処理関数の多くは、このUNIX時刻を処理対象としています。	

プロセス時刻の取得：

システム関数	説明
A = times()	Process時間,UsrCPU時間,SysCPU時間を取得します。
※ 戻り値 A は構造体です。メンバー名と設定値の意味は以下の通りです。【単位＝実数秒］ A.ptime = スクリプト開始 ～ 関数 times() 実行時点までの経過時間（Process時間） A.utime = Process時間のうちユーザーモードの実行時間（UsrCPU 時間） A.stime = Process時間のうちカーネルモードの実行時間（SysCPU 時間）	

UNIX時刻の操作（分解と合成）：

システム関数	説明
(year,mon,day[,hour,min,sec]) = t2ymd (D)	UNIX時刻(D)から、UTC 年月日時分秒を求めます。
(year,mon,day[,hour,min,sec]) = t2lymd(D)	UNIX時刻(D)から、LTZ 年月日時分秒を求めます。
D = ymd2t (year,mon,day[,hour,min,sec])	UTC 年月日時分秒から、UNIX時刻(D)を求めます。
D = lymd2t (year,mon,day[,hour,min,sec])	LTZ 年月日時分秒から、UNIX時刻(D)を求めます。
※ 上記４関数において、{year|mon|day|hour|min} は整数型、{sec}は実数型です。 関数 ymd2t() 及び lymd2t() では、年月日[時分秒]を表す１つの文字列でも日時の指定が可能です。	

時刻文字列の操作：

システム関数	説明
S = sdate(D) ｜ lsdate(D)	UNIX時刻(D) から ”YYYY-MM-DD” {UTC|LTZ} 文字列を生成します。（※1）
S = stime(D) ｜ lstime(D)	UNIX時刻(D) から ”hh:mm:ss” {UTC|LTZ} 文字列を生成します。（※1）
S = sftime(FMT,D) ｜ lsftime(FMT,D)	UNIX時刻(D) から {UTC|LTZ}で表現された時刻文字列(S) を生成します。（※2）
D = sptime(FMT,S) ｜ lsptime(FMT,S)	{UTC|LTZ}で表現された時刻文字列(S) から UNIX時刻(D) を生成します。（※3）
※1 生成される文字列は、システム変数 {FMT_SDATE｜FMT_STIME} で変更可能です。 ※2 FMT は、変換書式の文字列です。Ｃ言語 strftime(3) 関数の変換書式と同一です。 ※3 FMT は、変換書式の文字列です。Ｃ言語 strptime(3) 関数の変換書式と同一です。	

うるう年の判定：

システム関数	説明
I = isleap(I)	西暦年(I)のうるう年判定をします。（戻り値＝TRUE｜FALS）

時刻構造体の操作：

システム関数	説明
A = t2utc(D) ｜ t2ltz(D)	UNIX時刻(D) から、{UTC|LTZ} 時刻構造体 を求めます。
D = utc2t(A) ｜ ltz2t(A)	{UTC|LTZ} 時刻構造体(A) から、UNIX時刻(D) を求めます。
A = utc2ltz(A)	UTC 時刻構造体(A) から、LTZ 時刻構造体(A) を求めます。
A = ltz2utc(A)	LTZ 時刻構造体(A) から、UTC 時刻構造体(A) を求めます。
A = tm_norm(A)	時刻構造体を正規化します。（例：3時65分→4時05分）

時刻構造体の書式（フォーマット）

時刻構造体のメンバー	有効性	型	備　考
t.year = 西暦年４桁	○	I	
t.mon　= 月　（０１～１２）	○	I	
t.day　= 日　（０１～３１）	○	I	
t.hour = 時　（００～２３）	○	I	
t.min　= 分　（００～５９）	○	I	
t.sec　= 秒　（００～６０）	○	D	実数型
t.wday = 曜日（００～０６）	△	I	日曜＝0、月曜＝1 ～ 土曜＝6
t.yday = 年日（０～３６５）	△	I	正月＝0、大晦日＝364 又は 365
t.isdist = 夏時間（TRUE/FALS）	○	I	

注意１）○の項目は、常に有効かつ常に必須です。
注意２）△の項目は、入力時には設定不要（未利用）ですが、出力時には利用可能となります。

（ｊ）　数学関連（整数と実数）

＜システム変数＞

基本の数学定数：

名前	型	意味	備考
INF	D	無限大	isinf() で検査可能
NAN	D	非数（ Not a Number ）	isnan() で検査可能
M_E	D	自然対数の底	
M_PI	D	円周率	

最大値と最小値：

名前	型	意味	備考
INT_MAX INT_MIN	I	整数の最大値 と 整数の最小値	正の値 と 負の値
DBL_MAX DBL_MIN	D	実数の最大値 と 実数の最小値	正の値 と 負の値
INT_QUANT	I	整数の正の最小値	
INT_EPSILON	I	整数のうち X!=X+1　となる正の最小値	イプシロン値
DBL_QUANT	D	実数の正の最小値	
DBL_EPSILON	D	実数のうち X!=X+1.0 となる正の最小値	イプシロン値

角度の単位：

名前	型	意味	備考
SYS_ANGLE	D	システムの角度単位（ RAD 又は DEG ）	初期値 = RAD

＜システム関数＞

実数の処理：

システム関数	説明
D = ceil(D) | floor(D) | trunc(D)	実数の丸め込み｛切り上げ｜切り下げ｜０の方向｝
D = rint(D) | round(D)	実数の丸め込み｛偶数丸め｜四捨五入｝
D = abs (D)	絶対値
(Dint,Dfra) = modf (D)	実数 D から｛整数=Dint＆少数=Dfra｝への分解
D　　= Dint + Dfra	｛整数=Dint＆少数=Dfra｝から実数 D への合成
(Dman,Iexp) = frexp(D)	実数 D から｛仮数=Dman＆指数=Iexp｝への分解
D　　= ldexp(Dman,Iexp)	｛仮数=Dman＆指数=Iexp｝から実数 D への合成

指数と対数：

システム関数	説明
D = log(D) | log2(D) | log10(D)	｛底ｅ｜底 2｜底10｝の対数
D = exp(D) | exp2(D) | exp10(D)	｛　ｅ｜　2｜　10｝の累乗
D = pow(Dx,Dy) | mod(Dx,Dy)	DxのDy乗 | Dx／Dyの余剰
D = sqrt(D) | cbrt(D)	平方根 | 立方根
D = lin (D) | dbi (D)	リニア値 | デシベル値

角度の単位

システム関数	説明
D = rad2deg(D) | deg(D)	Rad → Deg 変換
D = deg2rad(D) | rad(D)	Deg → Rad 変換
(Deg,Min,Sec) = rad2dms(D)	Rad → 度分秒 変換
(Deg,Min,Sec) = deg2dms(D)	Deg → 度分秒 変換
D = dms2rad(Deg,Min,Sec)	度分秒 → Rad 変換
D = dms2deg(Deg,Min,Sec)	度分秒 → Deg 変換

三角関数：

システム関数	説明
D = sin (D) | cos (D) | tan (D)	{正弦|余弦|正接} 関数
D = asin (D) | acos (D) | atan (D)	{正弦|余弦|正接}逆関数
D = atan2(Y,X)	{正接}逆関数（２変数指定）
角度単位の初期値は rad です。（システム変数 SYS_ANGLE で変更できます。）	

双曲関数：

システム関数	説明
D = sinh(D) | cosh(D) | tanh(D)	双曲線{正弦|余弦|正接} 関数
D = asinh(D) | acosh(D) | atanh(D)	双曲線{正弦|余弦|正接}逆関数

ベッセル関数：

システム関数	説明
D = j0(D) | j1(D) | jn(D,D)	第一種ベッセル関数
D = y0(D) | y1(D) | yn(D,D)	第二種ベッセル関数

座標変換：

システム関数	説明
(X,Y,Z) = xrot(x,y,z,θ)	Ｘ座標軸を +θ 回転した後の座標を求める
(X,Y,Z) = yrot(x,y,z,θ)	Ｙ座標軸を +θ 回転した後の座標を求める
(X,Y,Z) = zrot(x,y,z,θ)	Ｚ座標軸を +θ 回転した後の座標を求める
※ 角度単位の初期値は rad です。（システム変数 SYS_ANGLE で変更できます。）	

統計関数：

システム関数	説明
D = max(V) | med(V) | min(V)	{最大値|中央値|最小値} を求める
I = argmax(V) | argmin(V)	{最大値|最小値} のインデックスを求める
(Imin,Imax) = argmed(V)	中央値のインデックスを求める（戻り値は２つ。）
D = sum (V) | ave (V)	{合計値|平均値} を求める
D = svar(V) | uvar(V)	標本分散 (Sample)|不偏分散 (Unbias) を求める
D = sdev(V) | udev(V)	標本標準偏差(Sample)|不偏標準偏差(Unbias) を求める
D = lgamma(D) | tgamma(D)	{loge |True } ガンマ関数
D = erf(D) | erfc(D)	{誤差 |余誤差} 関数
※ パラメーター V は {整数、実数、又は、それらの配列} の任意並びです。 ※ 関数 argmed() の戻り値は、引数が奇数個の時 Imin=Imax 、偶数の時 Imin!=Imax となります。 ※ 標本分散と標本標準偏差は 1/N の式、不偏分散と不偏標準偏差は 1/(N-1) の式です。	

乱数発生：

システム関数	説明
I = rand(MAX)	0 <= x < MAX(I) の整数乱数を発生します。（0以上MAX未満）
D = urand()	0.0 < x < 1.0 の一様乱数を発生します。
D = nrand([AVE,DEV])	正規乱数を発生します。{平均値=AVE(D)／標準偏差=DEV(D)}
void = srand(I)	乱数のシードを手動設定します。（デフォルトでは、自動設定）
※ 関数 nrand() の引数を省略した場合は、標準正規分布（AVE=0.0／DEV=1.0）となります。	

無限大と非数：

システム関数	説明
I = isinf(D)	無限大かどうか判定します。(戻り値＝TRUE/FALS)
I = isnan(D)	非 数かどうか判定します。(戻り値＝TRUE/FALS)

（ｋ） プロセスと割り込み

＜システム定数＞

プロセス関連の定数：

名前	型	意味	備考
$$	I	自プロセスIDの値	= PID
$?	I	（直近）外部コマンドの終了値	

シグナル関連の定数：

名前	型	意味	備考
SIG_DFL	P	割り込み動作設定用（既定動作）	rxsig()で使用
SIG_IGN	P	割り込み動作設定用（受信無視）	rxsig()で使用
SIG_ERR	P	割り込み設定時のエラー戻り値 [= NULL]	rxsig()で使用
NSIG	I	シグナル番号の定義数	0 ～ (NSIG-1) が有効
シグナル番号 (I) = SIGHUP / SIGINT / SIGQUIT/ SIGILL / SIGABRT/ SIGFPE / SIGKILL/ SIGSEGV/ SIGPIPE/ SIGALRM/ SIGTERM/ SIGUSR1/ SIGUSR2/ SIGCHLD/ SIGCONT/ SIGSTOP/ SIGTSTP/ SIGTTIN/ SIGTTOU/			

＜システム関数＞

プロセス関数：

システム関数	説明
I = pid() | ppid()	{自 | 親} プロセスＩＤを取得します。
D = sleep (D)	スリープ状態に Ｉ[秒] 入ります。（戻り値＝残時間[秒]）
void = pause()	ポーズ　状態に入ります。
void = exit(I)	スクリプトを終了値 Ｉ で終了します。

外部コマンド：

システム関数	説明
I = system(S)	外部コマンド S を実行します。（戻り値＝コマンドの終了値）
So = `S` [< Si]	外部コマンド S を実行します。（戻り値＝標準出力）
※ 記号 ` ` にて実行する場合は、オプションで標準入力 Si(S) の指定が出来ます。又、標準出力は戻り値 So(S) としてキャプチャされます。（標準エラー出力はキャプチャされません。）なお、終了値は $? にセットされます。	

割り込み関数：

システム関数	説明
I = txsig(Isig,Ipid)	シグナル（Isig番）をプロセス（Ipid番）に送信します。
I = rxsig(Isig,Func)	シグナル（Isig番）受信時の処理関数を登録します。
※ Func(X) は、１つの引数を持つ任意のユーザー関数です。シグナル受信時に割り込み実行されます。仮引数には受信したシグナル番号がセットされます。（仮引数名は任意です。）なお、Func(X) の代わりに {SIG_DFL | SIG_IGN} を指定すると、シグナル受信時の動作が、それぞれ {既定動作 | 受信無視} となります。	

（ｌ） ガーベージコレクション

＜システム関数＞

明示的実行：

システム関数	説明
void = gc_collect(void)	ガーベージコレクションを明示的に実行します。
※ 通常は、完全自動でメモリ管理がされていますので、当関数を明示的に実行する必要はありません。しかし、大量のデータを連続的に使用＆破棄する場合などにおいて、適切なタイミングで当関数を実行することによって、メモリー使用量を削減することができます。	

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
```