Toy-lang Language Manual

Mitsuhide SATO

Jul 26, 2009

- 目 次 -

1. はじめに

この度は toy-lang に関心いただきましてありがとうございます。

この文書は、プログラミング言語 toy-lang について解説したものです。toy-lang は、個人的に開発している小さなスクリプト言語です。作者である私自身が気の向くまま、楽しみながら開発しているプログラミング言語です。

言語の方針としては、あくまでも簡単に、シンプルに仕様を抑えつつ作っているつもりですが、開発を始めてからほぼ一年、色々な機能も増えてきたため、一度この辺で仕様を整理する意味で本文書を用意することにしました。

まだまだ色々なアイデアを試してみたいと思っておりますので、もしこの文書が目に留まり何か考えるところがあればご意見をいただけると幸いです。

それでは、どうぞお楽しみ下さい。

2. 概要

2.1 特徴

toy-lang は、以下のような特徴を持ったプログラミング言語です。

- インタプリタ言語
- シンプルな文法
- プロトタイプベースのオブジェクト指向
- C 言語による拡張性
- ファーストクラスとしてのクロージャ
- キーワード引数
- その他

(1) インタプリタ言語

toy-lang はインタプリタ言語です。現在、いくつかの UNIX ライクな OS の上で動作します。UNIX のスクリプト形式もしくは、toy-lang 組み込みのコマンドラインインタプリタで対話的に動作させることができます。

(2) シンプルな文法

toy-lang の全ての文法は、コマンド呼び出しもしくはメソッド呼び出しに引数を伴ったものとして定義されます。一般的な言語にあるような文法やキーワード、予約語などは存在しません。

例えば、以下は toy-lang の if 文です。

```
if {$i = 1} then: {do-something} else: {do-otherwise};
```

toy-lang では、if という構文が用意されているわけではありません。この文は、if コマンドの引数として、コンディションを示す一パラメータである “`{$i = 1}`” 、コンディションが真の時に実行される “then: {do-something}”キーワード引数、コンディションが偽の時に実行される “else: {do-otherwise}” キーワード引数、という 3 つのパラメータを持つコマンドの実行と考えることができます。

各文の終端は “;”記号で示します。 “;”が現れた時点で、その文を評価するという意味になります。ただし、ブロック(ブロックについては後述しますが、“{}”で囲まれた文の集まりのこと)内の最後の文については、 “;”を省略することが可能です。

(3) プロトタイプベースのオブジェクト指向

toy-lang は、プロトタイプベースのオブジェクト指向機能を持っています。オブジェクトを生成する際には、クラスを指定しますが、クラスの実態は、toy-lang のオブジェクトと等価です。新たに生成されたオブジェクトは、生成時に指定したクラスにメッセージを委譲することでメソッドが選択され実行されます。

また、実行時に動的にクラスやオブジェクトに対してメソッドを定義することが可能です。

(4) C 言語による拡張性

toy-lang は C 言語により記述されており、比較的簡単に C 言語による拡張が可能です。

現時点では、実用的なプログラムを作成するために必要な機能はまだまだ足りないですが、必要な機能については比較的簡単に追加することができると思います。

(5) ファーストクラスとしてのクロージャ

toy-lang はクロージャ(実行時の環境を持つデータ型)を持ちます。クロージャの表現は、ソーススクリプト上で “{}”で囲まれた部分です。また、クロージャは、ファーストクラスのデータ型であるため、関数の内外への持ち出しや変数への保持、任意の時点での評価が簡単にできます。

クロージャは、一般的には実行時の環境を持つデータ型で、言語により意味や実装方法は様々ですが、toy-lang の場合は、実行時のローカル変数およびカレントの実行オブジェクトを保持する構造です。

(6) キーワード引数

toy-lang は、引数の表現が 2 種類あります。ひとつは、多くのの言語で一般的な位置パラメータです。もうひとつは、オプショナルなパラメータを指定する際に便利なキーワードパラメータです。

キーワードパラメータは、 “name: value”のように、キーワードを名前 +“:”で指定し、その後に渡したい値を指定します。また、キーワードパラメータは省略可能です。

(7) その他

その他、リフレクション、例外処理、遅延評価の仕組みがあります。また、個人的に興味のある機能なども随時追加されるかもしれません。

2.2 外観

それでは、toy-lang という言語はどのような外観をしているでしょうか。簡単なサンプルプログラムを以下に示します。

toy-lang の外観:

```
defun grep (pat file) {
    set f [new File];
    try {
       $f open mode: i $file;
       set n 1;
       while {set r [$f gets]} do: {
           if {$r =~ $pat} then: {
              print $file ":" $n ": " $r;
           };
           $n ++;
       };
    }
    fin: {
       $f close;
    };
};
```

上記サンプルですが、UNIX の grep コマンドを真似た関数の定義です。この関数の起動の仕方は、プロンプトに続いて以下のように入力します。

```
grep 'pattern' "file-name";
```

外観から読み取れる特徴としては、

1. “{” と “}” で囲まれた C 言語のような処理ブロック
2. while や if による制御構造
3. try による例外処理
4. “=~” による正規表現のパターンマッチング
5. よくわからない “;” による文の終端(!?)
6. UNIX シェルのような “$” 記号による変数の参照
7. C 言語っぽい演算子

などがあると思います。でも、ある程度の言語の経験のある方はそれほど違和感は無いのではないかと思います。

2.3 影響を受けた言語

toy-lang は、以下の言語の影響を受けています。

- C 言語
- Lisp (実は作者はよく知らないのですが)
- Tcl
- Ruby (こちらも実はほとんど存じ上げませんが)
- その他

言語の外観からすると、Tcl に一番近いかもしれません。これは、作者がしばらくの期間 Tcl のプログラマであった影響によると思われます。また、内部的な動作や言語の基本的な機能は Lisp から来ている部分が多いでしょう。Ruby からはコマンド名などのいくつかのアイデアを借用しています。その他、作者が過去に使用した言語から、知らずのうちに影響を受けているかもしれません。

目指したいところとしては、一番外側の括弧が無く目に優しい Lisp といった感じでしょうか。

3. 準備

3.1 プラットフォーム

現時点で動作を確認しているプラットフォームは以下のとおりです。

- Ubuntu 8.04
- Ubuntu 9.04
- FreeBSD 7.1R

実行形式を作るために、コンパイラは gcc、GC ライブラリとして Boehm GC および、正規表現ライブラリとして鬼車が必要です。その他のライブラリとしては、UNIX の標準ライブラリのみですので、一般的な UNIX 環境であればビルドは可能かと思います。

3.2 コンパイル方法

(1) 必要な外部ライブラリの用意

(1-1) GC に Boehm GC ライブラリを使います。以下の URL よりバージョン 7.1alpha3-080224 を入手してインストールしてください。

http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/

(1-2) 正規表現ライブラリに鬼車を使わせていただきました。以下の URL よりバージョン 5.9.1 を入手してインストールしてください。

http://www.geocities.jp/kosako3/oniguruma/index_ja.html

(2) ソースの入手

以下の URL から、toy-lang の tar ball を入手します。

http://www31.atwiki.jp/toy-lang/pages/12.html

(3) ビルド

(3-1) 展開したディレクトリに入ります。

```
cd toy
```

(3-2) 最初に、Makefile の PREFIX を調整してください。

(3-3) make を実行します。

```
make
```

とすると実行ファイルができます。

(3-4) インストールします。

```
make install
```

とすると、PREFIX にインストールされます。

(4) テストを実行します。

```
cd test
./testall
```

とすると、簡単なテストを実行します。全てのテストでOKと出れば大丈夫ですが、リリースバージョンによってはNGとなる項目がある場合もあります。

3.3 処理系の構成

toy-lnag を標準的な構成でインストールした際の処理系の構成について説明します。

3.3.1 ファイル構成

$PREFIX

/bin/toysh	…	インタプリタ本体
/lib/toy/setup.toy	…	起動時セットアップスクリプト
/lib/toy/lib/	…	ライブラリスクリプトディレクトリ

$HOME

/.toyrc	…	ユーザ定義のセットアップスクリプト

$PREFIX は、make 時に指定するインストール先ディレクトリです。
$HOME は、ユーザのホームディレクトリです。

3.3.2 環境変数

toy-lang では、以下のシェル環境変数を参照します。

$HOME	…	ユーザのホームディレクトリです。

3.3.3 グローバル変数

toy-lang が起動したときにに定義されるグローバル変数です。

HOME	…	ユーザのホームディレクトリです。
ENV	…	シェル環境変数の Hash オブジェクトです。
LIB_PATH	…	ライブラリスクリプトが格納されたディレクトリのリストです。
ARGV	…	インタプリタ起動時の引数リストです。
VERSION	…	処理系のバージョンです。
CWD	…	現在のカレントディレクトリです。

3.4 起動

toy-lnag のインタプリタを起動するには、UNIX 上のシェルから、toysh コマンドを実行します。

インタプリタの起動:

```
$ toysh
*** Start toy-lang interpreter version 0.0.27.
>
```

3.5 最新情報

toy-lang についての最新情報は、今後も Web ページで細々と発信してゆく予定です。以下の URL にてご確認ください。

http://www31.atwiki.jp/toy-lang/

4. 言語仕様

4.1 データ型

ここでは、toy-lang のデータ型について説明します。

toy-lang の型は、以下の説明で特に断りの無い限り、そのほとんどがファーストクラスオブジェクトとしての性質を持ちます。つまり、ほとんどの型が、変数への代入、引数への指定、戻り値としての使用が可能です。

4.1.1 nil

toy-lang で偽値を表します。ソース上では、 “nil”で表現します。

toy-lang では、nil 以外の全ての値は真として扱われ、nil は常に偽となります。

4.1.2 シンボル

toy-lang のシンボルは、変数名、クラス名、メソッド名、ハッシュのキー値を表す名前です。名前に使用可能な文字は以下のとおりです。

- A-Z、a-z
- 0-9 (ただし、0-9 のみで構成される場合は、整数値とみなされます)
- ! % & - _ = ^ ~ | @ + * < > . / ?

上記の文字の一文字以上の組み合わせが、toy-lang で有効なシンボルとなります。

```
例:     name
        _123
        my-name
```

4.1.3 参照

シンボルの先頭に “$”をつけたものは、シンボルが示す値の参照となります。

```
例:     $name
        $?
```

文中に参照表現が現れた場合、その参照は、現在の環境の値(ローカル変数、オブジェクト変数または、グローバル変数)に置換されてから関数もしくはメソッドが実行されます。

参照は、厳密には実行時の型ではないため、従ってファーストクラスオブジェクトではありません(参照が現れた時点で実際のデータ型へ置換されてしまうため)。

4.1.4 整数

toy-lang の整数は、64bit 符号付バイナリ値です。スクリプト内では 10 進と 16 進での表現が可能です。

例:
```
123
0
-123
0x0000ffff
```

4.1.5 浮動小数点

toy-lang の浮動小数点は、64bit 浮動小数点です。内部形式については、プラットフォームに依存します。スクリプト内での表現は、仮数表示および指数表示の組み合わせにより行います。

例:
```
.1
1.
.0
3.141592
-123.0
1E10
-1E-10
-.123E3
```

指数表示がない場合は、仮数表示部に小数点(“.”)が必要となります。小数点が無い場合には、整数と判断されます。ただし、“.” のみの場合、浮動小数点ではなく、シンボルとして判断されます。また、指数表示が存在する場合は小数点は不要です。

4.1.6 文字列

toy-lang の文字列は、ダブルクォート “"”で囲まれた文字の列です。また、文字列内では、エスケープ記号(\)で特定の文字コードを表現することが可能です。

例:
```
"Hello World"
""
"End\n"
```

文字列内で使用可能なエスケープ記号は、以下のとおりです。

- \\ →エスケープ記号
- \t →タブ
- \n →ラインフィード
- \r →キャリッジリターン
- \" →ダブルクオート

4.1.7 正規表現文字列

toy-lang では正規表現パターンを表すための型を用意しました。正規表現文字列は、文字列と似ていますが、シングルクォート “'”で囲まれた文字列であるところが異なります。また、利用可能なエスケープ記号も異なります。

例:
```
'[A-z0-9].*'
'\(.*\)'
```

文字列内で使用可能なエスケープ記号は、以下のとおりです。

- \\ →エスケープ記号
- \' →シングルクォート

文字列とは異なり、上記以外の組み合わせ以外で単独で現れたエスケープ記号は、エスケープ記号そのものを表します。

4.1.8 リスト

toy-lang のリストは、“(” と “)” で囲まれたデータの列です。要素間は空白文字(スペース、タブおよび、改行)により区切ります。また、各要素は任意のデータ型を指定できます。もちろん、リストの中の要素としてリストを指定することも可能です。

また、Lisp 処理系におけるドット対の表記も可能です。この場合は、Lisp と同様 cons セルの car 部、cdr 部のそれぞれの要素を記述することが可能です。

例:
```
()
(a b c)
("a" b 0 (1 2 3))
("a" . {do-something})
```

4.1.9 クロージャ(ブロック)

クロージャを説明する前に、まず、toy-lang のブロックについて説明します。ブロックとは、“{” と “}” で囲まれた文の集まりです。通常の使い方としては、例えばループの処理部や関数の本体などの一連の処理のかたまりを表すために使用します。

ブロックの例:
```
set i 0;
while {$i < 10} do: {
    # ループ本体のブロック
    $i ++;
};
```

スクリプト上に現れる “{” と “}” で囲まれた部分は全てブロックです。もちろん、ブロックの入れ子も可能です。

これだけだと、見た目では C 言語のブロックとはあまり違いはありません。

もうひとつ説明すると、toy-lang のブロックは、静的な構造であるということです。それでは、ブロックに対応する動的な構造とはなんでしょうか。それがクロージャです。

クロージャは、スクリプトが実行され、ブロックが含まれる文に処理が到達したときに生成されるデータ型です。クロージャを構成するデータ型は、ブロックが現れたときの実行環境とそのそのブロック自身を含みます。toy-lang のクロージャが持つ実行環境とは、クロージャが生成されたときのローカル変数とインスタンス変数です。

クロージャを使うと、処理のかたまりと環境のセットを簡単に受け渡したり、別の環境であとから実行するといったことができます。

クロージャによるブロックの持ち出しの例:

```
defun foo (x) {return {println $x " world"}};
set x [foo "hello"];
println $x                  # → {println $x " world"}
$x eval;                    # → hello world
```

4.1.10 評価ブロック

toy-lang は、文の評価を明示的に示す必要があります。評価ブロックは、“[”と “]”で囲まれた文の集まりです。表記自体はブロックと似ています。スクリプト中で、評価ブロックを含む文に到達したとき、そのコマンドやメソッドの実行に先立ち “[”と “]”とで囲まれたスクリプトが実行され、評価ブロックが記述された部分にその値が埋め込まれます。評価ブロックの実行環境は、実行時の環境のままです。従って、評価ブロックの中では、評価ブロックの外と同じ変数が参照可能です。

Lisp では、評価は自動的に実行されますが、toy-lang の場合は、プログラマが明示的に示す必要があることが Lisp とは大きく異なる部分です。

評価ブロックによる置換の例:

```
set i 10;
println [$i ++];            # → 11
```

評価ブロックは、厳密には実行時の型ではないため、従ってファーストクラスオブジェクトではありません(評価ブロックが現れた時点で文の評価後のオブジェクトへ置換されてしまうため)。

4.1.11 関数

toy-lang の関数もまたファーストクラスオブジェクトの性質をもちます。関数は、fun コマンド、defun コマンドおよび、Object クラスの method メソッドにより生成されます。

fun コマンド、defun コマンドおよび method メソッドにより生成されるのはすべて関数型の型であり、生成のされかたによりその関数がどのように管理されるかが異なります。

fun コマンドでは、名前の無い関数を生成することが可能です。

defun コマンドでは、名前付きの関数を生成し、グローバルなスコープで関数を名前で呼び出すことができるようになります。

method メソッドでは、オブジェクトおよびクラスに対してオブジェクトのメンバ関数を定義することができます。オブジェクのメソッド呼び出しによりオブジェクトのメンバ関数を呼び出すことができるようになります。

4.1.12 オブジェクト

toy-lang のオブジェクトは、new コマンドにより生成されます。オブジェクトは、その構成要素として、委譲先のクラス、メンバ変数、メンバ関数を含みます。

オブジェクトに対して委譲先のクラスを複数指定することができ、これにより多重継承の仕組みを実現しています。

4.1.13 制御

toy-lang の制御型は、関数からの戻り値を指定して関数を終了したり、ループの実行を中断、再開したりするためのものです。

制御は、以下のコマンドにより生成されます。

- return
- break
- continue
- retry
- redo

return は、関数の実行を中断し、return の引き数の値を結果として呼び出し元へ返します。

break は、ループの処理を中断します。引数を指定することにより、結果として値を返すことが可能です。

continue は、現在のループの処理を中断し、次の要素から処理を再開します。

retry は、現在のループの処理を中断し、ループの最初から処理をやり直します。

redo は、現在のループの処理を中断し、もう一度同じ要素の処理を再開します。

4.1.14 例外

例外は、toy-lnag の処理系の様々な部分で発生します。発生の原因は、スクリプトの記述間違い、引数の数の不一致、スクリプト中での明示的記述などで、たくさんの要因があります。

例外が発生すると、関数の呼び出しを遡って例外が伝播されます。例外を補足するためには、try コマンドを使用します。関数の呼び出しを遡る途中で、try コマンドにより例外が補足されない場合はトップレベルまで伝播し、最終的にインタプリタによりエラーが報告されます。

try コマンドが例外を補足した際に、発生した例外を参照するには、try コマンドの catch: ブロックに渡されるバインド変数を使用する必要があります。

スクリプト中で例外を明示的に発生させるためには、throw コマンドを使用します。

4.1.15 バインドリスト

バインドリストとは、イテレータを構成するブロックに渡されるバインド変数のリストです。バンドリストは、“|” と “|” とで囲まれたシンボルのリストです。以下は、バインドリストの使用例です。

バインドリストの例:

```
(1 2 3) each do: {| i |              # | i | はバインドリスト
       println $i                    # i はバインド変数
};
```

toy-lan の文は、基本的に “;” により終端しますが、バインドリストに関しては例外的に “;” は不要となっています。

注意事項 “|” 記号は、シンボル名の一部としても使用可能です。従って、バインドリストを記述する際の “|” の前後には空白文字が必要です。空白が無い場合、前後の文字と合わせてシンボルと判断されてしまいます。

4.2 クラス

toy-lang のクラス構成は以下のとおりです。

Object	全てのオブジェクトの基底クラスです。
Integer	整数型の wrapper クラスです。
Real	浮動小数点型の wrapper クラスです。
List	リスト型の wrapper クラスです。
String	文字列型の wrapper クラスです。
RQuote	正規表現型の wrapper クラスです。
Block	ブロック(クロージャ)型の wrapper クラスです。
Hash	Hash クラスです。キー・値のペアでデータを管理するクラスです。
Array	Array クラスです。任意の型をインデックス番号で管理するクラスです。
Functions	名前付き関数が便宜的にメンバとして登録されるクラスです。
File	ファイル入出力のためのクラスです。

toy-lang における wrapper クラスは、基本的なデータ型(整数、浮動小数点、リスト、文字列、正規表現文字列および、ブロック)に対してクラスとしての性質を付与するためのものです。ソースコード中に、これらの基本的なデータ型がオブジェクトの位置に現れた際には、自動的に対応する wrapper クラスで wrap されたオブジェクトが生成され、wrapper クラス内のメンバ関数が呼び出されます。

各クラスの詳細については、「5 章　リファレンス」を参照して下さい。

4.3 プログラムの構成

toy-lang のプログラムはスクリプトと文により構成されます。以下の例は、小さなスクリプトと文の例です。ひとつのスクリプトに二つの文が含まれています。最初の文は、"defun" で始まるもので、もうひとつは、"[addl 100];" です。defun で始まる文は、さらにブロック内に文が現れています。

プログラムの例:

```
defun add1 (x) {
    fun () {$x ++ 1};
};
[add1 100];          # → 101
```

(1) スクリプト

スクリプトは、toy-lang で記述されたプログラムです。スクリプトは、通常ファイルにテキスト形式で記述し、load コマンドによりインタプリタに読み込まれ、評価されます。
スクリプトファイルの拡張子は、“.toy” です。

また、スクリプトは、toy-lang の文の集まりです。先に説明したブロックや評価ブロックの中も文の集まりであり、再帰的にスクリプトの構造が現れます。

(2) 文

文は、toy-lang の評価の最小単位であり、スクリプト内の文が先頭から順に実行することでプログラムのアルゴリズムを実際に実現してゆきます。スクリプト中の各文は、“;” で終端します。

(3) コメント

toy-lang のコメントは、 “#” で始まり、その行の終わりまでとなります。

コメントの例:

```
# この行はコメントです。
set i 0; # シャープ記号(#)から行の最後までがコメントです。
```

(4) スクリプト/関数の値

スクリプトの値は、スクリプトの最後の文が実行された結果となります。また、“result”コマンドで明示的に示すことも可能です。文がイテレータブロックの場合は、“break”コマンドの引数も値となります。

さらに、関数を実行した場合は、“return”コマンドにより戻り値を指定することも可能です。

(5) 文の実行結果

文を実行した後は、特殊なローカル変数である、“?”にその結果が設定されます。参照する際には、“$?”とします。

文の実行結果:

```
set i 100;
$i + 1;
println $?;          # → 101
```

4.4 構文規則

toy-lang の文は、以下の規則に従い評価されます。

- 関数 ［引数 ...］； … 構文規則 1
- オブジェクト メソッド ［引数 ...］； … 構文規則 2

(1) 関数呼び出し

構文規則 1 は、組み込み関数、ユーザ定義関数および、カレントオブジェクトのメソッドを呼び出すための構文です。

文の最初のキーワードが評価され、組み込み関数、ユーザ定義関数もしくは、メソッドである場合に、その関数が実行されます。

関数呼び出しの例:

```
set x 100;          # → 組み込み関数の呼び出し
foo "a string";     # → ユーザ定義関数の呼び出し
```

オブジェクトに定義したメソッドの呼び出しに関して、同一オブジェクトのコンテキストでは、オブジェクトの指定は省略可能であり、その際は、関数呼び出しの形式で自オブジェクトのメソッドが起動されます。以下は、暗黙のメソッド呼び出しの例です。

暗黙のメソッド呼び出しの例:

```
class F;
F method m1 (x) {
    m2 $x;          # F::m2 の呼び出し
};
F method m2 (x) {
   println $x;
};
```

(2) メソッド呼び出し

構文規則 2 は、オブジェクトに対してメソッドを適用するための構文です。

文の最初のキーワードが評価され、それがオブジェクトの場合、次のキーワードを評価し、オブジェクトのメソッドの探索が行われます。メソッドが見つかった場合、そのオブジェクトの環境が作られ、メソッドが実行されます。

メソッド呼び出しの例:

```
class F;
F method m1 (x) {
    println $x;
};
set o [new F];
$o 100;       # → 100
```

この呼び出し規則は、組み込みのデータ型でも良く使用します。例えば、整数型に対する演算などは、この呼び出し規則の典型的な例となります。

整数データによる Integer クラスメソッドの呼び出しの例:

```
set i 0;
$i + 1;       # Integer::+ の呼び出し
```

(3) クラスメソッド呼び出し

構文規則 2 のもうひとつの場合として、文の最初のキーワードがクラス名の場合、そのクラスのメソッドが呼び出されます。

クラスメソッドの呼び出しの例:

```
Integer method foo  ()  {
    # do something
    ...
};
Integer foo;         # クラスメソッドの呼び出し
```

この例のように、組み込みクラスおよび、ユーザ定義のクラスに対して、動的にメソッドを定義することが可能です。

4.5 変数

toy-lang には、プログラム中で設定したり参照することが可能な変数があります。変数のクラスとしては、ローカル変数、インスタンス変数および、グローバル変数があります。

変数の参照は、プログラムのテキスト中で、変数のシンボル名の先頭に “$” を付与することにより行います。プログラムのコンテキストで、各変数クラスに同名の変数が存在した場合、

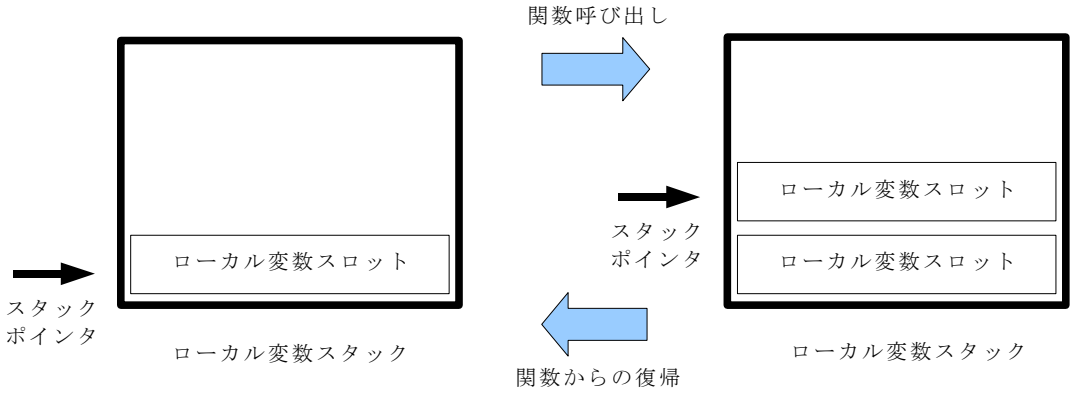
ローカル変数　→　インスタンス変数　→　グローバル変数

の順序で変数の検索が行われます。

toy-lang の変数は動的なものです。スクリプトの任意の時点で任意の型のデータを設定することが可能です。

(1) ローカル変数

ローカル変数は、関数の呼び出し毎に生成されるスタック環境です。



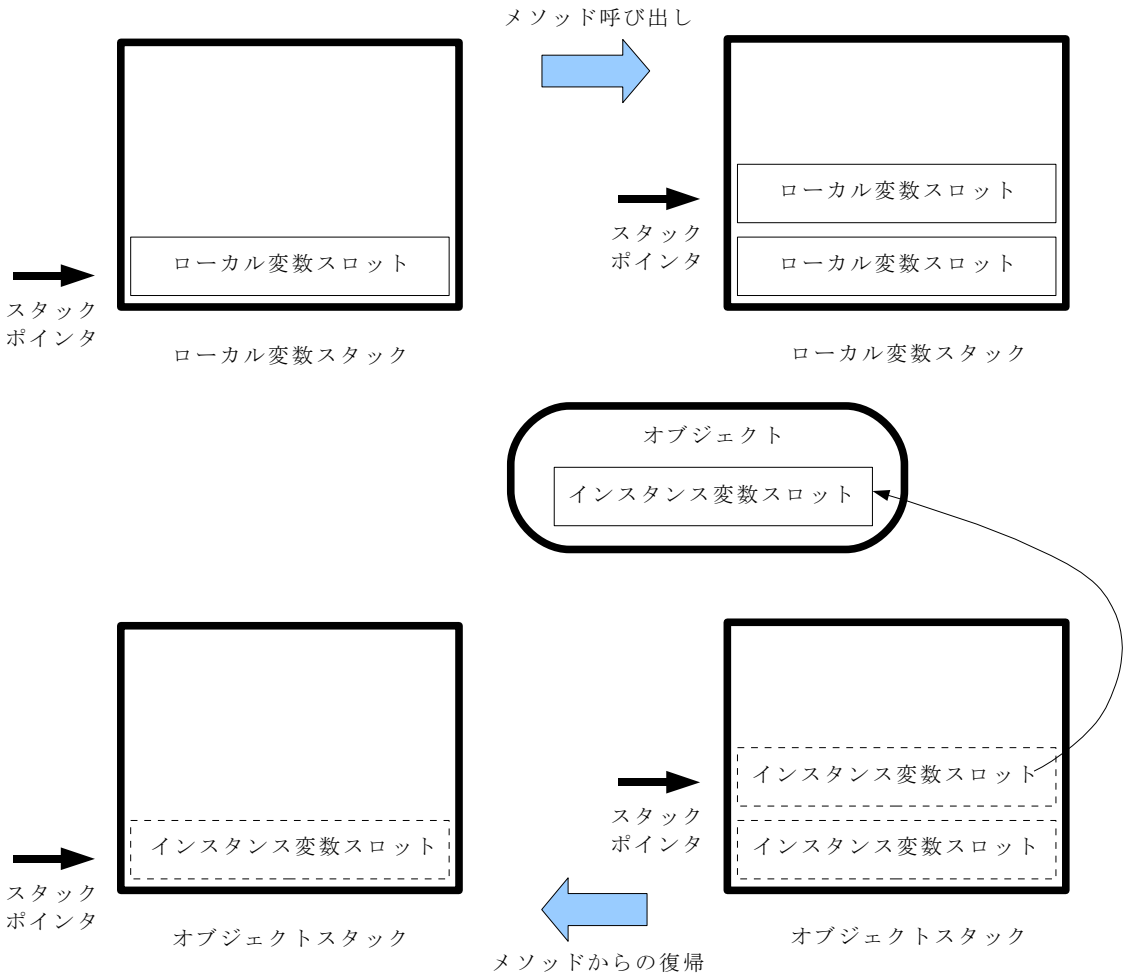
ローカル変数スロットは、実行中の関数内で参照可能なシンボルの辞書構造です。ローカル変数の参照の際は、スタックポインタにあるローカル変数スロットからシンボルが参照さ、値が読み出されます。

関数の呼び出しにより、新たなローカル変数スロットが生成され、関数の終了とともにそのスロットは消滅します。ただし正確には、関数内にてクロージャが生成された際には、そのクロージャに構成の一部としてその時のローカル変数スロットが保持されます。

ローカル変数の設定には、“set” コマンドを使用します。また、ローカル変数がローカル変数スロットに存在するかを調べるには、“set?” コマンドを使用します。

(2) インスタンス変数

インスタンス変数は、オブジェクトが持つ変数です。インスタンス変数は、オブジェクトの生成により、そのオブジェクトの内部に生成されるインスタンス変数スロットに保持されます。



toy-lang には、ローカル変数スタックのほかに、オブジェクトスタックというものもあり、オブジェクトのメソッド呼び出しに伴い、ローカル変数スタックに加えてオブジェクトスタック上にも新たな環境が生成されます。オブジェクトスタック上のインスタンス変数スロットは、オブジェクトが持つインスタンス変数スロットへの参照になっています。

インスタンス変数の設定には、“sets”コマンドを使用します。また、インスタンス変数がインスタンス変数スロットに存在するかを調べるには、“sets?”コマンドを使用します。

(3) グローバル変数

toy-lang のグローバル変数は、スクリプト中の全ての場所からの設定、変更が可能です。

グローバル変数の定義には、“defvar”コマンドを使用します。また、内容の変更には、“setvar”コマンドを使用します。二重にグローバル変数を定義しようとすると、例外が発生します。グローバル変数が存在するかを調べるには、“defvar?”コマンドを使用します。

4.6 関数

toy-lang の関数は、名前なし関数と名前つき関数があります。関数の実態は、名前なし関数であり、先に説明した関数データ型のデータがその本体です。名前つき関数は、名前なし関数が Functions クラスのインスタンス変数に登録されたものです。

名前なし関数の生成は、“fun” コマンドを使用します。名前なし関数の呼び出しは、変数の参照や、　関数からの戻り値を実行するなどの方法があります。

名前なし関数呼び出しの例:

```
set foo [fun (i) {$i + 1}];
$foo 100;            # 名前なし関数の呼び出し
```

名前つき関数の定義は、“defun” コマンドを使用します。呼び出しは、関数名を指定することにより行います。

名前つき関数呼び出しの例:

```
defun foo (i) {$i + 1};
foo 100;             # 名前つき関数の呼び出し
```

4.7 クラス

toy-lang のクラスは、オブジェクトをクラスの辞書へ名前をつけて登録したものに過ぎません。クラスという言葉は、便宜的なものであり、実態はオブジェクトそのものです。

クラス(としてのオブジェクト)の目的は、他のクラスを定義する際のテンプレートとして名前を参照するためと、オブジェクトを生成するためのデレゲートクラスを指定するためにあります。

クラスは、“class” コマンドで生成します。本コマンドでクラスを定義する際に、デレゲートクラスを指定することができます。

デレゲートクラスとは、普通の言葉で言うところの親クラスのことで、toy-lang の場合はこれを複数指定することが可能であり、多重継承のようなことも可能です。

クラス定義の例:

```
# 単純なクラスを定義(デフォルトで Object クラスがデレゲートクラスとなる)
class bar;
# クラス foo を定義(bar, baz クラスにデレゲートする)
class foo bar delegate: (baz);
```

4.8 オブジェクト

toy-lang は、標準で簡単なオブジェクトシステムをサポートしています。オブジェクトの意味合いは、他の言語と似ていて、オブジェクトとそれに包括されるメンバ関数、メンバ変数を含むデータの集合です。また、オブジェクトはデレゲートクラスを持ち、継承と似た動作も行えます。

toy-lang のオブジェクトは動的なもので、実行中にメソッドの定義が可能であり、mix-in のような事を簡単に実現できます。

オブジェクトは、“new” コマンドにより生成されます。オプションとして既定のクラスを指定することにより、クラスからオブジェクトを生成する仕組みと似た動作をさせることも可能です。

オブジェクト生成の例:

```
# 単純なオブジェクトの生成(デフォルトでは Object クラスのインスタンスとなる)
set o [new];
# クラスを指定してオブジェクトを生成
set o [new foo];
```

4.9 メソッド

toy-lang のメソッドは、オブジェクトのメンバとして登録された関数のことです。メソッドは、クラスおよびオブジェクトに対して定義することができます。

メソッドの定義は、メソッドを定義したいクラスもしくはオブジェクトに対して“method”メソッドを呼び出すことにより行います。ちなみに“method”メソッド自身は、全ての基底オブジェクトである Object クラスに定義されたメソッドです。

メソッド定義の例:

```
# ユーザ定義クラスにメソッドを定義する
class MyClass;
MyClass method foo(x) {println $x};

# オブジェクトに対してメソッドを定義する
set x [new MyClass];
$x method bar(x) {println $x};

# wrapper 組み込みのクラスに独自のメソッドを定義する
List method print () {println [self]};
(a b c) print;      # → ”(a b c)”
```

メソッドの呼び出しは、オブジェクトにメソッド名を指定することにより行うか、もしくは、そのオブジェクトのメソッドが実行中の場合は、直接メソッド名を指定することにより行います。

メソッド呼び出しの例:

```
# クラス/メソッドを定義し、オブジェクトを生成
class MyClass;
MyClass method foo(x) {println $x};
set x [new MyClass];
# メソッドの呼び出し
$x foo ”abc”;       # → ”abc”
```

4.10 引数

toy-lang では他の言語と同様に、関数やメソッドを定義する際に引数を定義することができます。関数やメソッド定義に指定する引数のことを仮引数と言います。仮引数定義は、関数名、メソッド名、もしくは fun コマンドの場合には fun コマンドの直後に、“(”と“)”で囲まれた部分に記述します。

また、関数やメソッドを呼び出す際には、その関数やメソッドに与えるための引数を指定しますが、これを実引数と呼びます。実引数の指定は、関数名やメソッド名に続いて、関数に与えたい値を列挙することにより行います。

仮引数と実引数の例:

```
# 関数定義における仮引数
defun foo (x y z) {...};   # x, y, z が仮引数
# 関数呼び出しにおける実引数
foo 1 ”abc” (a b c);       # 1, ”abc”, (a b c) が実引数
```

toy-lang での引数の指定の方法にはいくつかのルールがあります。

1. 位置引数
2. キーワード引数
3. スイッチ引数
4. 可変長引数
5. 遅延評価引数

上記の各引数のそれぞれは、ひとつの関数およびメソッドの定義内で組み合わせて指定することが可能です。以下、それぞれの引数について説明します。

(1) 位置引数

位置引数とは、その名の通り仮引数と実引数の対応を引数の位置により指定するものです。仮引数部分では、引数のシンボル名をリストの形で順に列挙します。実引数として渡す際には、関数名、メソッド名に続いて関数に渡す値を順に列挙します。位置引数の場合、仮引数と実引数の数は一致しなければなりません。

位置引数の例:

```
# 位置引数の定義
defun foo (x y z) {...};   # x, y, z が仮引数
# 位置引数を指定して関数を呼び出す
foo 1 ”abc” (a b c);       # 1, ”abc”, (a b c) が実引数
```

上記の例の場合、関数 foo が呼び出された際には、x に 1 が、y に "abc" が、そして、z にリストとして (a b c) がそれぞれ渡されます。

(2) キーワード引数

キーワード引数とは、呼び出し側と関数定義側との引数の受け渡しをキーワードにより指定するものです。位置引数とは異なり、呼び出し時には仮引数側で指定されたすべてのキーワード引数を指定する必要はありません。呼び出し時には、必要な引数のみ指定することが可能です。また、指定の順序についても制限はなく、呼び出し側で任意の順序でキーワード引数を指定することができます。

キーワード引数の定義は、仮引数側の定義は、「**キーワード： シンボル名**」となります。また呼び出し時の指定は、「**キーワード： 値**」となります。仮引数側のキーワードと呼び出し側のキーワードが一致した際に、呼び出し時の値がキーワード仮引数のシンボル名に設定されます。

呼び出し時にキーワード引数が指定されなかった場合は、関数側のシンボルは設定されません。そのため関数側では、キーワード引数が与えられているかどうかを知るために、“set?” コマンドを使って調べる必要があります。

キーワード引数の例:

```
# キーワード引数の定義
defun if (cond then: then-body else: else-body) {...};
# キーワード引数を指定して関数を呼び出す
if $x then: {set i 0} # then: キーワード引数を指定、else: は未設定
```

上記の例は、if コマンドを模擬的に示したものです。関数呼び出しにおいて thne: キーワードを指定した場合、if コマンドが呼び出された際には、シンボル名 then-body に値(ここで

は、{set i 0})が設定されます。else: キーワードは設定されていないため、else-body シンボルには値は設定されません。また、この例では、仮引数の定義で、第 1 引数には cond というシンボルが指定されていますが、これは、位置引数とキーワード引数を混在して指定可能であることを示しています。

(3) スイッチ引数

スイッチ引数は、キーワード引数の実引数を指定する際の糖衣構文です。キーワード引数の実引数を指定する際に「**:キーワード**」とすることにより、値を省略することができます。仮引数には、非 nil 値が設定されます。スイッチ引数の目的は、パラメータの有無のみが意味を持つような引数の与え方を簡単にすることです。

スイッチ引数の例:

```
# キーワード引数の定義
defun foo (nocase: case-switch) {...};
# スイッチ引数を指定して関数を呼び出す
foo :switch;  # foo の case-switch には非 nil が設定される
foo;          # foo の case-switch には値が設定されない
```

(4) 可変長引数

位置引数は、仮引数と実引数の数が一致する必要があります。実引数として任意個数の引数を渡したい場合は、キーワードパラメータとして「args: **シンボル名**」を指定します。このようにすると、位置引数より多い実引数の残りすべては、関数が呼び出された際に args: キーワード引数のシンボル名にリストとして設定されます。

可変長引数の例:

```
# 可変長引数を利用した関数の呼び出し
foo 1 2 3;
# 可変長引数の定義と動作
defun foo (a args: rest-parameters) {
  println $a;                    # → 1
  println $rest-parameters       # → (2 3)
};
```

可変長引数と位置引数を同時に指定する際には、実引数の個数は最低限仮引数で定義した個数が必要です。

(5) 遅延評価引数

遅延評価引数は、クロージャブロックの評価を仮引数の参照時まで遅延するための機能です。遅延評価引数を指定するには、関数定義の仮引数シンボル名を「**&シンボル名**」と記述することにより行います。呼び出し側の実引数は、遅延評価引数に対してクロージャブロックを指定します。

遅延評価引数の例:

```
# 遅延評価引数の定義
defun foo (&a) {
  println $a; # → 1 (初回の参照なのでクロージャ{$i ++}が評価される)
  println $a; # → 1 (2回目以降の参照は、最初の評価結果となる)
};
# 遅延評価引数を用いた関数の呼び出し
set i 0;
foo {$i ++};
```

遅延評価引数にクロージャブロックを渡すと、関数内で、最初にそのシンボルを参照した際にクロージャが実行されます。クロージャの実行環境は、実引数の環境となります。遅延評価引数が2度目以降に参照された場合は、最初のクロージャの実行結果(スクリプトの最後の値)が記憶されますので、その記憶された値となります。複数回クロージャが実行されることはありません。

遅延評価引数に対して、実引数としてクロージャブロック以外を渡した場合は、通常の変数の参照と同等になります。

4.11 マクロ

toy-lang には、パーサに組み込みのマクロ的な構文があります。ただし、Lisp のようにユーザがマクロを定義することはできません。

4.11.1 get マクロ

get マクロは、Object に対して get メソッドを適用するための糖衣構文です。もともとは、Hash の要素を参照する際の簡易な構文として考えましたが、1 引数の get メソッドを持つオブジェクトに対してはどのオブジェクトに対しても使用可能です。

get マクロの記法は、文の中で以下のように記述します;

オブジェクト, パラメータ

このように記述した際、実行時には以下のように置換され、オブジェクトの get メソッドが呼び出されます。

［**オブジェクト** get **パラメータ**］

get マクロの例:

```
# Hash の生成と要素の代入
set o [new Hash];
$o set ”foo” 1;
# 要素の参照(通常の記法)
println [$o get ”foo”];     # → 1
# 要素の参照(get マクロによる記法)
println $o,”foo”;           # → 1
```

4.11.2 init マクロ

init マクロは、オブジェクトの生成の際に実行されるコンストラクタ(クラスの init メソッド)を簡易に呼び出すための糖衣構文です。

init マクロの記法は、文の中で以下のように記述します;

`クラス名 パラメータ

このように記述した際、実行時には以下のように置換され、クラスの init メソッドが呼び出されます。

[new **クラス名** init: (**パラメータ**)]

init マクロの例:

```
# クラス定義
class Foo;
Foo method init (a) {....};
# オブジェクトの生成(通常)
set o [new Foo init: (1)];
# オブジェクトの生成(init マクロによる記法)
set o `Foo 1;
```

4.12 関数のオートロード

toy-lang には、関数のオートロード機能があります。

未知の関数名が呼び出された際にインタプリタはグローバル変数 LIB_PAHT に記述されたサーチパスの順に関数が定義されたファイルを探し、自動的にスクリプトファイルをロードし、その後関数の呼び出しを行います。

スクリプトファイルの命名規則は、「**関数名** + ".toy"」です。本スクリプトファイル中には、当該関数の定義(defun)がある必要があります。もしスクリプトファイル中に当該関数の定義が無い場合、関数呼び出し毎に毎回ファイルがロードされることになります。

5. リファレンス

本章は、組み込みのコマンドおよびメソッドのリファレンスとなります。
本リファレンス中での書式の凡例を以下に示します。

凡例:

command	(正体) スクリプト中でそのものを表記することを表します。
value	(斜体) この位置にパラメータを記述することを表します。
val ...	直前の引数を繰り返し指定可能なことを表します。
[*val*]	省略可能な引数であることを表します。 （評価ブロックの意味ではないことに注意）
val1 | *val2*	val1 または val2 どちらかを選択することを表します。
(...)	引数にリストを与えることを表します。
{ ... }	引数にブロックを与えることを表します。
{| *val* | ... }	引数にバインド変数が受け取り可能なブロックを指定可能であることを表します。
"*val* "	引数が文字列であることを表します。
'*val* '	引数が正規表現文字列であることを表します。
class::*method*	クラスに定義されたメソッドを明示する際の表記です。

5.1 コマンドリファレンス

!

説明

論理否定を返します。

書式

! *val*

詳細

val の値の論理否定を返します。

戻り値

val が非 nil の場合は nil を、nil の場合は t を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
if {! $a} then: {
  # $a is false
} else: {
  # $a is true
};
```

alias

説明

変数の別名を作成します。

書式

alias [up: *uplevel*] *orig-var new-var*

詳細

orig-var で指定したローカル変数を *new-var* でも参照できるようにします。*orig-var* は存在しなくても構いません。*new-var* が設定された時点で、*orig-var* も生成されます。

up: *uplevel* を省略した場合は、現在のスタック上の *orig-var* の別名を作成します。*uplevel* に正の整数を指定した場合には、指定した数分スタックを呼び出し元の方向へ辿り、*orig-var* の参照をローカルスタックへ作成します。

戻り値

成功した場合、t を返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrStackUnderflow: uplevel で指定したスタックが存在しません。
ErrLinkAlias: すでに別名が存在するか別名参照がループしています。

使用例

```
# ローカル変数のエイリアスを生成
set i 0;
alias i x;
println $x;   # → 0

# シンボル名で引数を与える
defun foo (s) {
  alias up: 1 $s x;
  println $x;
};
set i 1;
foo i;        # → 1
```

and

説明

引数の真偽値による論理積を返します。

書式

and *val* ...

詳細

val の値が nil である場合偽と判断します。全ての引数についての真偽値を調べ、それら全ての値の論理積を返します。

戻り値

t または nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
and 0 1 2;          # → t
and 0 1 2 nil;      # → nil
and ();             # → t
```

break

説明

イテレータを脱出します。

書式

break [*value*]

詳細

イテレータを脱出します。*value* を指定した際には、そのイテレータコマンドの戻り値となります。
break が有効なコマンドおよびメソッドは次の通りです:
while / try / cond / Integer::each / Integer::list / List::each / Array::each

戻り値

制御が呼び出し元に戻るため、break 自身の戻り値はありません。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set i 0;
while {true} do: {
  if {$i > 10} then: {break};
  $i ++;
};
```

call

説明

関数を呼び出します。

書式

call *fun* (*var* ...)

詳細

関数 fun を呼び出します。関数の引数は、1 引数のリストで指定します。
スクリプトの中で、引数を組み立てて関数を呼び出す際に使用します。

戻り値

関数の戻り値を返します。

例外

ErrSyntax: 書式に誤りがあります。
呼び出された *fun* で例外が発生した際には、その例外が返されます。

使用例

```
defun foo (x y z) {# do-something};
set l ();
$l + 1;
$l + 2;
$l + 3;
call foo $l;         # same as: foo 1 2 3;
```

case

説明

値により処理ブロックを選択します。

書式

case *val pat1* { *body1* } [*pat2* { *body2* }] ... [* { *default-body* }]

詳細

val で与えられたデータの文字列表現が、*pat1, pat2 ...* の文字列表現に完全一致した場合、次の引数の処理ブロックを実行します。
マッチするパターンが無い場合、* パターンが定義されていれば *default-body* を実行します。
注意: case コマンドのブロック中では、制御コマンドはそのまま呼び出し元へ返されます。

戻り値

実行された処理ブロックの値となります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
case $i
  0 {println "zero"}
  1 {println "one"}
  * {println "other"};
```

cd

説明

カレントワーキングディレクトを変更します。

書式

cd ["*directory*"]

詳細

directory にカレントワーキングディレクトリを変更します。パラメータを省略した場合、ユーザのホームディレクトリに変更します。
ディレクトリの変更に成功した場合、グローバル変数 CWD に現在のディレクトリが設定されます。

戻り値

コマンド実行後の絶対パス文字列を返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrSysCall: システムコールの呼び出しでエラーが発生しました。エラー詳細は Exception のエラー文字列を参照のこと。

使用例

```
cd "/";
```

class

説明

クラスを定義します。

書式

class *name* [*parent-class*] [delegate: (*additional-class ...*)]

詳細

クラス名 *name* を新規に定義します。
parent-class には、親クラスを指定します。省略した場合は、既定クラスとして Object クラスが親クラスとなります。
さらにいくつかの親クラスを指定するためには、*additional-class* に列挙することにより行います。
parent-class および、*additional-class* を複数指定した場合のメソッドの探索順は、

- 自オブジェクトのメソッド
- *parent-class* およびその親クラスのメソッド
- 最初の *additional-class* およびその親クラスのメソッド
- 次の *additional-class* およびその親クラスのメソッド
- ...

となります。

戻り値

生成されたクラスオブジェクトを返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrNoClass: 親クラスが存在しません。

使用例

```
class foo;          # Object が親クラス
class foo bar;      # bar が親クラス
class foo bar delegate: (x y);
                    # bar / x / y が親クラス
```

cond

説明

条件により処理ブロックを選択します。

書式

cond *exp1 block1* [*exp2 block2*] ...

expX: *value* | { *block*}

blockX: *value* | {| *result* | *block*}

詳細

expX を評価し、非 nil の場合 *blockX* を実行します。評価順序は *exp1* → *exp2* ... となり、順に全ての *exp* と *block* の評価、実行を行います。ただし 、*block* の中で *break* が呼ばれた際は、評価はその時点で終了します。

exp は、値もしくは処理ブロックを指定します。処理ブロックの場合は、そのブロックを実行した結果が nil かどうかを判定します。

block は、値もしくは処理ブロックを指定します。処理ブロックを指定した場合は、そのブロックを実行します。また、バインドリストを指定すると、直前に評価した *exp* の結果を *result* として受け取ることができます。

戻り値

最後に評価された *block* の値となります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
1 .. 100 do: {| i |
  cond {[$i % 15] = 0} {break "FizzBuzz”}
       {[$i % 3] = 0}  {break "Fizz"}
       {[$i % 5] = 0}  {break "Buzz"}
       t               $i;
};
```

continue

説明

イテレータの次の要素に制御を渡します。

書式

continue

詳細

イテレータの現在の要素の処理を終了し、次の要素に制御を渡します。
continue が有効なコマンドおよびメソッドは次の通りです:
while / Integer::each / Integer::list / List::each / Array::each

戻り値

制御が呼び出し元に戻るため、continue 自身の戻り値はありません。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set i 0;
while {true} do: {
  $i ++;
  if {[$i % 2] = 0} then: {continue};
  println $i;
};
```

defun

説明

関数を定義します。

書式

defun *name* (*argspec*) { *body* }

argspec : [*posargs* ...] [*keyword*: *keyargs* ...] [args: *restarg*]

posargs : *symbol* | &*symbol*

keyargs : *symbol* | &*symbol*

restarg : *symbol*

詳細

名前つき関数 *name* を定義します。引数定義 *argspec* に関しては、「4.10 引数」を参照ください。

body には、関数本体のスクリプトを記述します。

戻り値

定義された関数を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
defun fact (n) {
    if {$n < 1}
    then: 1
    else: {
        $n * [fact [$n - 1]];
    };
};
```

defvar

説明

グローバル変数を定義します。

書式

defvar *var* [*val*]

詳細

グローバル変数 *var* を定義し、値を *val* とします。*val* が省略された場合は、変数 *var* の値を返します。
すでに *var* が定義されている場合はエラーとなります。

戻り値

var の値を返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrNoSuchVariable: 変数が存在しません。
ErrAlreadyExists: 既に変数が存在します。

使用例

```
defvar FOO "X";      # → "X"
defvar FOO;          # → "X"
```

defvar?

説明

グローバル変数が定義されているか調べます。

書式

defvar? *var*

詳細

グローバル変数 *var* が設定されているかを調べ、t または nil を返します。

戻り値

グローバル変数 *var* が定義されている場合 t を返します。定義されていない場合 nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
defvar FOO "X";
defvar? FOO;         # → t
```

exists?

説明

変数が定義されているか調べます。

書式

exists? *var*

詳細

変数 *var* が現在のスコープに定義されているかを調べ t または nil を返します。ローカル変数→インスタンス変数→グローバル変数の順番に調べ、変数が見つかった時点で t を返します。

戻り値

変数 *var* が定義されている場合 t を返します。定義されていない場合 nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
exists? foo;
```

exit

説明

toy-lang インタプリタを終了します。

書式

exit [*status*]

詳細

toy-lang インタプリタを終了します。*status* を 0-255 の値の範囲で指定すると、プロセスの終了ステータスとなります。省略した場合は 0 となります。

戻り値

ありません。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
exit 1;
```

false

説明

nil を返します。

書式

false

詳細

nil を返します。

戻り値

常に nil を返します。

例外

ありません。

使用例

```
false;
```

file

説明

ファイルに関する調査を行います。

書式

file ?

file exists? *"name"*

file dir? *"name"*

file read? *"name"*

file write? *"name"*

file exec? *"name"*

file list [*"directory"*]

詳細

ファイルの調査を行います。第 1 引数により目的の動作を指定します。以下、第 1 引数の値による動作を説明します。

? ... ヘルプを表示します。

exists? ... 第 2 引数で指定したファイルが存在する場合、t を返します。

dir? ... 第 2 引数で指定したファイルがディレクトリの場合、t を返します。

read? ... 第 2 引数で指定したファイルが読込み可能な場合、t を返します。

write? ... 第 2 引数で指定したファイルが書込み可能な場合、t を返します。

exec? ... 第 2 引数で指定したファイルが実行可能な場合、t を返します。

(上記、いずれも t 以外の場合は nil を返します。)

list ... 第 2 引数で指定したディレクトリのエントリ一覧のリストを返します。第 2 引数を省略した場合は、"." (カレントディレクトリ)となります。

戻り値

詳細を参照のこと。

例外

ErrSyntax: 書式に誤りがあります。

ErrFileAccess: ファイルアクセス権限がありません。

使用例

```
file exists? "foo.txt";
             # → t or nil
file list "/";
             # → return list of root directory files
```

fork

説明

プロセスを生成します。

書式

fork

詳細

UNIX プロセスを生成します。プロセスは、fork を呼び出した次の処理から親プロセス、子プロセスに分岐し、それぞれが同じ状態(ただし、fork の戻り値を除く)次の命令から実行されます。

戻り値

親プロセスの場合は子プロセスの id が返ります。子プロセスの場合は 0 が返ります。

例外

ErrSyntax: 書式に誤りがあります。
ErrSysCall: システムコールでエラーが発生しました。

使用例

```
set pid [fork];
if [$pid = 0] then: {# for parent} else: {# for child};
```

fun

説明

無名の関数を定義します。

書式

fun (*argspec*) { *body* }

argspec : [*posargs* ...] [*keyword*: *keyargs* ...] [args: *restarg*]

posargs : *symbol* | &*symbol*

keyargs : *symbol* | &*symbol*

restarg : *symbol*

詳細

名前無し関数を定義します。引数定義 *argspec* に関しては、「4.10 引数」を参照ください。

body には、関数本体のスクリプトを記述します。

戻り値

生成された関数を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
defun foo (n) {
  fun (i) {$n ++ $i}
};
set x [foo 100];
$x 10;              # → 110
$x 10;              # → 120
```

if

説明

条件により処理を分岐します。

書式

if *exp* then: *then-block1* [else: *else-block*]

exp : *value* | { *block* }

then-block : *value* | { *block* }

else-block : *value* | { *block* }

詳細

exp を評価し、非 nil の場合 *then-block* を実行します。nil の場合は *else-block* を実行します。

exp は、値もしくは処理ブロックを指定します。処理ブロックの場合は、そのブロックを実行した結果が nil かどうかを判定します。

then-block および *else-block* は、値もしくは処理ブロックを指定します。処理ブロックを指定した場合は、そのブロックを実行します。

戻り値

実行された *then-block* もしくは *else-block* の値となります。実行すべきブロックが指定されていなかった場合は、nil が返ります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
# 通常の使用例
if {$i = 0} then: {
  # do-something
} else: {
  # do-otherwise
};

# 変数の有無によりオプションを設定するようなケース
defun foo (sort-order: order) {
  if [set? order] else: {set order >};
  # ...
};
```

info

説明

インタプリタの情報を返します。

書式

info local | closure | self | func | class | global | script

詳細

インタプリタに関する情報をリストで返します。第1引数の内容により動作が変わります。以下、第1引数の内容による機能を示します。
local ... 現在のローカル変数のシンボルリストを返します。
closure ... 現在のクロージャのスコープにあるローカル変数のシンボルリストを返します。
self ... カレントオブジェクトのインスタンス変数のシンボルリストを返します。
func ... 組み込みコマンド、ユーザ定義関数のシンボルリストを返します。
class ... クラスのシンボルリストを返します。
global ... グローバル変数のシンボルリストを返します。
script ... ロードされたスクリプトのリストを返します。

戻り値

指定した引数によります。詳細参照のこと。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
# 全ての変数の一覧をリスト化
[info local] concat [info closure] [info self] [info global];

# 関数の一覧
info func;
```

lazy

説明

遅延評価ブロックの生成

書式

lazy { *block* }

詳細

遅延評価ブロックを生成します。lazy コマンドにより生成された *block* は、次に参照された際に評価されます。そのため、lazy コマンドの戻り値は変数に設定する必要があります。次に、lazy コマンドの戻り値が設定された変数が参照された際は、*block* がクロージャとして評価されます。また、クロージャの実行結果の値は、最初に lazy コマンドの戻り値が設定された変数に再設定されます。二度目以降の変数の参照は、最初のクロージャの評価結果が使用されることになります。

戻り値

遅延評価ブロックが返ります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set foo [lazy {println "foo"; result 123}];
println $foo;       # → display "foo" and 123
println $foo;       # → display 123
```

load

説明

スクリプトをロードします。

書式

load "*script*"

詳細

script で示されるスクリプトファイルをロードし、実行します。

戻り値

スクリプト id が返されます。

例外

ErrSyntax: 書式に誤りがあります。
ErrFileNotOpen: ファイルがオープンできません。
WarnParseString: 文字列の終端記号がありません。
WarnParseClose: リスト、ブロックもしくは、評価ブロックの終端記号がありません。
ErrParseClose: 括弧の対応が間違っています。

使用例

```
load "lib/ls.toy";
```

new

説明

オブジェクトを生成します。

書式

new [*parent-class*] [delegate: (*additional-class ...*)]
[init: (*construct-parameters ...*)]

詳細

オブジェクトを生成します。
parent-class には、親クラスを指定します。省略した場合は、既定クラスとして Object クラスが親クラスとなります。
さらにいくつかの親クラスを指定するためには、*additional-class* に列挙することにより行います。
parent-class および、*additional-class* を複数指定した場合のメソッドの探索順は、

- 自オブジェクトのメソッド
- *parent-class* およびその親クラスのメソッド
- 最初の *additional-class* およびその親クラスのメソッド
- 次の *additional-class* およびその親クラスのメソッド
- ...

となります。
クラスに init メソッドが定義されている場合は、*construct-parameters* を伴っての init メソッドの呼び出しが行われます。

戻り値

生成されたクラスオブジェクトを返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrNoClass: 親クラスが存在しません。

使用例

```
new;                    # Object が親クラス
new foo;                # foo が親クラス
new foo delegate: (x y);
                        # foo / x / y が親クラス
new Hash init: (((key1 . 1) (key2 . 2)));
                        # Hash オブジェクト生成の例
```

or

説明

引数の真偽値による論理和を返します。

書式

or *val* ...

詳細

val の値が nil である場合偽と判断します。全ての引数についての真偽値を調べ、それら全ての値の論理和を返します。

戻り値

t または nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
or 0 1 2;           # → t
or 0 1 2 nil;       # → t
or ();              # → t
or nil nil;         # → nil
```

print / println

説明

文字列を印字します。

書式

print [file: *file-object*] args ...

println [file: *file-object*] args ...

詳細

args を文字列に変換し、ファイルへ出力します。

出力先のファイルの指定は以下の通りです。

- *file-object* が指定されていれば、このオブジェクトの puts メソッドを呼び出します。
- ローカル変数またはインスタンス変数に@out 変数があれば、このオブジェクトの puts メソッドを呼び出します。
- グローバル変数の stdout 変数があれば、このオブジェクトの puts メソッドを呼び出します。

print と println の違いは、print が改行コードを出力しないのに対して 、purintln は、改行コードを出力し、ファイルオブジェクトの flush メソッドを呼び出します。

戻り値

nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

ErrNoSuchVariable: @out 変数または、stdout 変数が見つかりません。

使用例

```
println "Hello world.";
              # → Hello world.
1 each to: 10 do: {| i | print $i " "}; println;
              # → 1 2 3 4 5 6 7 8 9 10
```

pwd

説明

カレントワーキングディレクトリを返します。

書式

pwd

詳細

現在のワーキングディレクトリを返します。また、グローバル変数 "CWD" にも同じ値が設定されます。

戻り値

カレントワーキングディレクトリを返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
pwd;
```

rand

説明

乱数を返します。

書式

rand

詳細

乱数を返します。

戻り値

0.0 以上 1.0 未満の乱数を返します。

例外

ありません。

使用例

```
rand;
```

redo

説明

イテレータの現在の要素に再度制御を移します。

書式

redo

詳細

イテレータの現在の要素に制御を移し、再び同じ要素を実行します。
redo が有効なコマンドおよびメソッドは次の通りです:
while / Integer::each / Integer::list / List::each / Array::each

戻り値

制御がイテレータに戻るため、redo 自身の戻り値はありません。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set i 0;
while {true} do: {
  println $i;
  if {$i > 5} then: {redo};
  $i ++;
};            # → 0 1 2 3 4 5 6 6 6 6 6 6 6 6 6 ...
```

result

説明

引数を返します。

書式

result [*val*]

詳細

与えられた引数 *val* をそのまま返します。*val* が指定されなかった場合は nil を返します。

関数の最後の行で関数の値を指定したいときなど、return コマンドの代わりに使うことができます。また、if や case、cond などの評価関数のブロックの中で、評価関数自身の値を明示する際にも使えます。

戻り値

val を返します。*val* の指定が無い場合は nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set x [if $cond then: {
  do-something;
  result 0;
} else: {
  do-otherwise;
  result 1;
}];
```

retry

説明

イテレータの最初の要素に再度制御を移します。

書式

retry

詳細

イテレータの最初の要素に制御を移し、再び同じイテレータを最初から実行します。try コマンドの場合、try ブロックもしくは catch ブロック中で retry が実行された場合、再度 try ブロックの実行を行います。
retry が有効なコマンドおよびメソッドは次の通りです:
while / try / Integer::each / Integer::list / List::each / Array::each

戻り値

制御がイテレータに戻るため、retry 自身の戻り値はありません。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
1 .. 5 do: {| i |
  println $i;
  if {$i >= 5} then: {retry};
};            # → 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 ...
```

return

説明

関数を終了し呼び出し元に戻ります。

書式

return [*val*]

詳細

現在の関数呼び出しを終了し、呼び出し元に戻ります。
val が指定されている場合、関数の戻り値は *val* となります。指定されていない場合は、関数の戻り値は nil となります。

戻り値

呼び出し元に戻るため、このコマンドの戻り値はありません。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
defun foo (n) {
  set i 0;
  set s 0;
  while {true} do: {
    $s ++ $i;
    $i ++;
    if [$i > $n] then: {return $s};
  };
};
foo 100;            # → 5050
```

sdir

説明

ロード済みスクリプトの一覧を返します。

書式

sdir

詳細

load コマンドおよび、unknown によりロードされたスクリプトの一覧を、スクリプト id、ファイル名のリストの形式で返します。

戻り値

スクリプト id、ファイル名のリストのリストを返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
sdir;
```

self

説明

現在のオブジェクトを返します。

書式

self

詳細

現在のオブジェクトスタックのスタックトップにあるオブジェクトを返します。

戻り値

現在のオブジェクトを返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
# 自クラスのメソッド内から、自クラスの foo メソッドを呼ぶ
[self] foo;          # → same as: foo;
```

set

説明

ローカル変数に値を設定します。

書式

set *var* [*val*]

詳細

ローカル変数 *var* に、値 *val* を設定します。*val* を省略した場合は、*var* の内容を返します。

戻り値

var に設定した内容を返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrNoSuchVariable: 変数が定義されていません。

使用例

```
set i 0;                    # i に整数を設定
set m "Hello world."        # m に文字列を設定
set s sym                   # s にシンボルを設定
set o [new];                # o にオブジェクトを設定

set i;                      # → 0
```

set?

説明

ローカル変数が定義されているか調べます。

書式

set? *var*

詳細

ローカル変数 *var* が設定されているかを調べ、t または nil を返します。

戻り値

ローカル変数 *var* が定義されている場合 t を返します。定義されていない場合 nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set foo "X";
set? foo;            # → t
```

sets

説明

インスタンス変数に値を設定します。

書式

sets *var* [*val*]

詳細

インスタンス変数 *var* に、値 *val* を設定します。*val* を省略した場合は、*var* の内容を返します。

戻り値

var に設定した内容を返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrNoSuchVariable: 変数が定義されていません。

使用例

```
sets i 0;                  # i に整数を設定
sets m "Hello world."      # m に文字列を設定
sets s sym                 # s にシンボルを設定
sets o [new];              # o にオブジェクトを設定

sets i;                    # → 0
```

sets?

説明

インスタンス変数が定義されているか調べます。

書式

sets? *var*

詳細

インスタンス変数 *var* が設定されているかを調べ、t または nil を返します。

戻り値

インスタンス変数 *var* が定義されている場合 t を返します。定義されていない場合 nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
sets Foo "X";
sets? Foo;          # → t
```

setvar

説明

グローバル変数に値を設定します。

書式

setvar *var* [*val*]

詳細

グローバル変数 *var* を定義し、値を *val* とします。*val* が省略された場合は、変数 *var* の値を返します。
var が定義されていない場合はエラーとなります。

戻り値

var の値を返します。

例外

ErrSyntax: 書式に誤りがあります。
ErrNoSuchVariable: 変数が存在しません。

使用例

```
defvar FOO "X";      # → "X"
defvar FOO;          # → "X"
setvar FOO "Y";
setvar FOO;          # → "Y"
```

show-stack

説明

スタック情報を返します。

書式

show-stack

詳細

stack-trace を呼んだ時点のスタック情報を返します。

戻り値

スタック情報のリスト。
返されるリストの書式は以下の通りとなります:
((stack_size . x) (cur_obj_stack . y) (cur_func_stack . z))
x ... スタックサイズ
y ... 現在のオブジェクトスタック消費数
z ... 現在の関数スタック消費数

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
show-stack
```

sid

説明

関数が定義されたスクリプトのスクリプト ID を返します。

書式

sid *func*

詳細

func で指定した関数が定義されているスクリプトを調べるために、関数のスクリプト ID を返します。

戻り値

スクリプト ID を返します。*func* が存在しない場合、nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
sid foo;
```

sleep

説明

プロセスをスリープします。

書式

sleep *msecs*

詳細

msecs で指定したミリ秒時間、プロセスをスリープします。

戻り値

t を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
sleep 1000;  # → sleep 1 sec
```

stack-trace

説明

スタックトレース情報を返します。

書式

stack-trace

詳細

stack-trace を呼んだ時点のスタックトレース情報を文字列データで返します。

戻り値

文字列によるスタックトレース情報。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
stack-trace;
```

symbol

説明

文字列からシンボルを生成します。

書式

symbol *str*

詳細

文字列 *str* で指定したシンボルを生成します。動的にシンボル名を生成する際に使用します。

戻り値

シンボルを返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set i 0;
set [symbol ["sym" . $i]] 1;
       # → same as: set sym0 1;
```

throw

説明

例外を発生します。

書式

throw *exp* [*message*]

throw (*exp* . [*message*])

詳細

例外 *exp* を発生します。詳細メッセージとして *message* を指定します。

2 番目の書式は、try コマンドで例外をキャッチした際にバインド変数で渡される例外を throw する際に使用できます。

戻り値

例外を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
try {
    # do-something
} catch: {| e |
    throw $e
};
```

time

説明

ブロックを実行し、実行時間を計測します。

書式

time { *block* }

詳細

block を実行し、実行時間を出力します。

戻り値

block の実行結果が返ります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
time {tarai 12 6 0};
```

trace

説明

スクリプトのトレースを出力します。

書式

trace [*level*] … (書式 1)
trace [out: *fd*] { *block* } … (書式 2)

詳細

1 番目の書式で、スクリプトのトレースを出力します。*level* に 0 を指定するとトレースを停止します。*level* が 1 以上でトレースを出力を開始します 。*level* を省略すると、現在のトレースレベルを返します。
2 番目の書式では、指定した *block* のトレースを出力します。*fd* を指定することにより、指定のファイルへトレース結果を出力できます。

戻り値

書式 1 の場合は、トレースレベルを返します。
書式 2 の場合は、*block* の結果を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
trace 1;       # 以降のスクリプトの実行をトレースする
trace {do-something}; # do-something の実行をトレースする
```

trap

説明

シグナルの処理方法を定義します。

書式

trap *signal* [{ *block* }]

詳細

プロセスが *signal* を受信した際の動作を *block* として定義します。*block* が省略された場合は、現在の *block* を返します。未定義の場合は nil を返します。
signal としては、以下を指定可能です:
SIGHUP / SIGINT / SIGQUIT / SIGPIPE / SIGALRM / SIGTERM / SIGURG / SIGCHLD / SIGUSR1 / SIGUSR2
スクリプト実行中にシグナルを受信した際は、現在実行中のスクリプトの実行が中断され、*block* が実行されます。その後、*block* の実行結果がそれまで実行していたスクリプトの実行結果となります。

戻り値

signal に設定された *block* を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
# ^C を押された際に ReceiveSignal 例外を throw する
trap SIGINT {throw ReceiveSignal "Recieve SIGINT"};
```

true

説明

t を返します。

書式

true

詳細

t を返します。

戻り値

常に t を返します。

例外

ありません。

使用例

```
true;
```

try

説明

例外を補足します。

書式

try { *body* } [catch: { | *exp* | *catch-body* }] [fin: { f*in-body* }]

詳細

body を実行し、例外が補足された場合、*catch-body* を実行します。*fin-body* は定義されている場合には、try コマンドの終了時の最後に必ず実行されます。
catch: ブロックに *exp* を定義した場合は、*catch-body* 内にて発生した例外を参照できます。
body および *catch-body* 内では、制御コマンドとして retry が使用可能で 、retry コマンドが実行された場合には *body* を再実行します。
catch-body 内でさらに例外が発生した場合は、その例外が呼び出し元に伝播します。
catch: ブロックが定義されていない場合は、*body* 内で発生した例外はそのまま呼び出し元に伝播します。例外を単に無視する場合は、catch: ブロックに空のブロックを記述します。

戻り値

body もしくは *catch-body* の結果のうち、最後に実行された値を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set f [new File];
try {
  $f open "foo.txt";
}
catch: {| e |
  throw $e;
}
fin: {
  $f close;
};
```

type?

説明

オブジェクトの型名シンボルを返します。

書式

type? *val*

詳細

*val*の型名シンボルを返します。

型名シンボルには次のものがあります:

NIL / SYMBOL / LIST / INTEGER / REAL / STRING / NATIVE / OBJECT / CLOSURE / FUNC / RQUOTE / BIND

戻り値

型名シンボルを返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
case [type? $o]
INTEGER {# do-something }
REAL    {# do-something }
OBJECT  {# do-something }
*       {# do-otherwise };
```

unknown

説明

コマンド未定義時のオートロードドライバです。

書式

unknown *func* [*args* ...]

詳細

コマンド呼び出しにおいて、*func* が未定義の際に上記の書式により unknown が呼び出されます。
unknown が呼び出された際には、グローバル変数 LIB_PATH のパス順に *func* が定義されているファイルを順次ロードし、*func* を実行します。

戻り値

func の戻り値になります。

例外

func の戻り値になります。

使用例

ユーザスクリプトより呼び出すことはありません。

unset

説明

ローカル変数を未定義とします。

書式

unset *var*

詳細

ローカル変数 *var* を未定義とします。

戻り値

var が存在する場合、*var* の値を返します。*var* が存在しない場合は nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
unset foo;
```

unsets

説明

インスタンス変数を未定義とします。

書式

unsets *var*

詳細

インスタンス変数 *var* を未定義とします。

戻り値

var が存在する場合、*var* の値を返します。*var* が存在しない場合は nil を返します。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
unsets foo;
```

while

説明

繰り返し処理を行います。

書式

while { *cond-block* } do: { *block* }

詳細

cond-block の評価結果が非 nil の間、*block* を繰り返し実行します。
block の中では、以下の制御コマンドが使用できます:
break / continue / redo / retry

戻り値

block の最後の実行結果が返ります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
set i 0;
while {$i < 10} do: {
  # do-something
  $i ++;
};
```

yield

説明

イテレータブロックを実行します。

書式

yield { | *var* | *block* } [*arg* ...]

詳細

block をイテレータブロックとして実行します。その際 *arg* をバインドリストとして渡します。

戻り値

block の実行結果が返ります。

例外

ErrSyntax: 書式に誤りがあります。

使用例

```
defun foo-loop (body times) {
  set i 0;
  while {$i < $times} do: {
    yield $body $i;
    $i ++;
  };
};
foo-loop {| i | println $i} 10;
```

5.2 クラスリファレンス

Array::+

説明

書式

詳細

戻り値

例外

使用例

Array::append

説明

書式

詳細

戻り値

例外

使用例

Array::each

説明

書式

詳細

戻り値

例外

使用例

Array::get

説明

書式

詳細

戻り値

例外

使用例

Array::init

説明

書式

詳細

戻り値

例外

使用例

Array::last

説明

書式

詳細

戻り値

例外

使用例

Array::len

説明

書式

詳細

戻り値

例外

使用例

Array::list

説明

書式

詳細

戻り値

例外

使用例

Array::set

説明

書式

詳細

戻り値

例外

使用例

Array::string

説明

書式

詳細

戻り値

例外

使用例

Block::>>

説明

書式

詳細

戻り値

例外

使用例

Block::eval

説明

書式

詳細

戻り値

例外

使用例

File::close

説明

書式

詳細

戻り値

例外

使用例

File::eof?

説明

書式

詳細

戻り値

例外

使用例

File::flush

説明

書式

詳細

戻り値

例外

使用例

File::gets

説明

書式

詳細

戻り値

例外

使用例

File::init

説明

書式

詳細

戻り値

例外

使用例

File::open

説明

書式

詳細

戻り値

例外

使用例

File::puts

説明

書式

詳細

戻り値

例外

使用例

File::ready?

説明

書式

詳細

戻り値

例外

使用例

File::set!

説明

書式

詳細

戻り値

例外

使用例

File::stat

説明

書式

詳細

戻り値

例外

使用例

Hash::each

説明

書式

詳細

戻り値

例外

使用例

Hash::get

説明

書式

詳細

戻り値

例外

使用例

Hash::init

説明

書式

詳細

戻り値

例外

使用例

Hash::keys

説明

書式

詳細

戻り値

例外

使用例

Hash::len

説明

書式

詳細

戻り値

例外

使用例

Hash::pairs

説明

書式

詳細

戻り値

例外

使用例

Hash::set

説明

書式

詳細

戻り値

例外

使用例

Hash::set?

説明

書式

詳細

戻り値

例外

使用例

Hash::string

説明

書式

詳細

戻り値

例外

使用例

Hash::unset

説明

書式

詳細

戻り値

例外

使用例

Integer::!=

説明

書式

詳細

戻り値

例外

使用例

Integer::%

説明

書式

詳細

戻り値

例外

使用例

Integer::*

説明

書式

詳細

戻り値

例外

使用例

Integer::+

説明

書式

詳細

戻り値

例外

使用例

Integer::++

説明

書式

詳細

戻り値

例外

使用例

Integer::-

説明

書式

詳細

戻り値

例外

使用例

Integer::--

説明

書式

詳細

戻り値

例外

使用例

Integer::..

説明

書式

詳細

戻り値

例外

使用例

Integer::/

説明

書式

詳細

戻り値

例外

使用例

Integer::<

説明

書式

詳細

戻り値

例外

使用例

Integer::<=

説明

書式

詳細

戻り値

例外

使用例

Integer::=

説明

書式

詳細

戻り値

例外

使用例

Integer::>

説明

書式

詳細

戻り値

例外

使用例

Integer::>=

説明

書式

詳細

戻り値

例外

使用例

Integer::each

説明

書式

詳細

戻り値

例外

使用例

List::+

説明

書式

詳細

戻り値

例外

使用例

List::->>

説明

書式

詳細

戻り値

例外

使用例

List::.

説明

書式

詳細

戻り値

例外

使用例

List::<<

説明

書式

詳細

戻り値

例外

使用例

List::<<-

説明

書式

詳細

戻り値

例外

使用例

List::>>

説明

書式

詳細

戻り値

例外

使用例

List::add

説明

書式

詳細

戻り値

例外

使用例

List::append!

説明

書式

詳細

戻り値

例外

使用例

List::car

説明

書式

詳細

戻り値

例外

使用例

List::cdr

説明

書式

詳細

戻り値

例外

使用例

List::concat

説明

書式

詳細

戻り値

例外

使用例

List::delete

説明

書式

詳細

戻り値

例外

使用例

List::each

説明

書式

詳細

戻り値

例外

使用例

List::eval

説明

書式

詳細

戻り値

例外

使用例

List::filter

説明

書式

詳細

戻り値

例外

使用例

List::get

説明

書式

詳細

戻り値

例外

使用例

List::insert

説明

書式

詳細

戻り値

例外

使用例

List::item

説明

書式

詳細

戻り値

例外

使用例

List::join

説明

書式

詳細

戻り値

例外

使用例

List::last

説明

書式

詳細

戻り値

例外

使用例

List::len

説明

書式

詳細

戻り値

例外

使用例

List::map

説明

書式

詳細

戻り値

例外

使用例

List::next

説明

書式

詳細

戻り値

例外

使用例

List::null?

説明

書式

詳細

戻り値

例外

使用例

List::seek

説明

書式

詳細

戻り値

例外

使用例

List::split

説明

書式

詳細

戻り値

例外

使用例

Object::delegate?

説明

書式

詳細

戻り値

例外

使用例

Object::eq

説明

書式

詳細

戻り値

例外

使用例

Object::get

説明

書式

詳細

戻り値

例外

使用例

Object::method

説明

書式

詳細

戻り値

例外

使用例

Object::method?

説明

書式

詳細

戻り値

例外

使用例

Object::set!

説明

書式

詳細

戻り値

例外

使用例

Object::string

説明

書式

詳細

戻り値

例外

使用例

Object::type?

説明

書式

詳細

戻り値

例外

使用例

Object::var?

説明

書式

詳細

戻り値

例外

使用例

Object::vars

説明

書式

詳細

戻り値

例外

使用例

Real::!=

説明

書式

詳細

戻り値

例外

使用例

Real::*

説明

書式

詳細

戻り値

例外

使用例

Real::+

説明

書式

詳細

戻り値

例外

使用例

Real::-

説明

書式

詳細

戻り値

例外

使用例

Real::/

説明

書式

詳細

戻り値

例外

使用例

Real::<

説明

書式

詳細

戻り値

例外

使用例

Real::<=

説明

書式

詳細

戻り値

例外

使用例

Real::=

説明

書式

詳細

戻り値

例外

使用例

Real::>

説明

書式

詳細

戻り値

例外

使用例

Real::>=

説明

書式

詳細

戻り値

例外

使用例

String::!=

説明

書式

詳細

戻り値

例外

使用例

String::+

説明

書式

詳細

戻り値

例外

使用例

String::.

説明

書式

詳細

戻り値

例外

使用例

String::<

説明

書式

詳細

戻り値

例外

使用例

String::<=

説明

書式

詳細

戻り値

例外

使用例

String::=

説明

書式

詳細

戻り値

例外

使用例

String::=~

説明

書式

詳細

戻り値

例外

使用例

String::>

説明

書式

詳細

戻り値

例外

使用例

String::>=

説明

書式

詳細

戻り値

例外

使用例

String::eval

説明

書式

詳細

戻り値

例外

使用例

String::len

説明

書式

詳細

戻り値

例外

使用例

String::sub

説明

書式

詳細

戻り値

例外

使用例