プログラミング言語

NewtonScript

Ver. 1.0.0

原題 “The NewtonScript Programming Language”

翻訳　　齋藤匡弘

brown@ga2.so-net.or.jp

本書の配賦条件

本書は Apple の著作物 “The NewtonScript Programming Language” をベースにして翻訳・変更・加筆された物であり、Apple はこの 2 次著作物の公開を許可する。また、個人で使用される場合に限りその使用を許可する。但し、Apple は翻訳文の内容について責任を負う物ではない。

訳者のことば

まず最初に、訳者は10年以上プログラミングに関わってはいるものの、Newton プログラミングに関してはまだ経験が浅い初心者であることをここで白状しておく。さらに学生時代英語のテストで赤点を取った事もあるほどの寂しい英語力を誇っている :-) だから、本書のいたるところで間違いをやらかしているかもしれないことをご了承いただくとともに、もしそういった所を見つけたら、是非お知らせ願いたい。訳者の email アドレスは、brown@ga2.so-net.or.jp である。

訳者がこのマニュアルを日本語に訳そうと思い立った動機は次の通りである:

- 日本語で書かれた Newton プログラミングの書籍があまりに少ないこと
- 文法を理解していると、プログラミング言語をよりよく理解できること
- 日本語のドキュメントがあると、潜在的プログラマを開拓でき、これがひいては Newton プラットフォームの活況につながると思ったこと。

原書は日本語にするにはやや回りくどい部分が有ったので、訳者が意訳・変更している部分が多少ある。また、訳注として、訳者自身の考えも書き込んでいるが、その部分は訳註とした。

それではこのアイデアに満ちた小さな言語、NewtonScirpt の世界を見ていきたいと思う。

目次

PREFACE　本書について xi

想定される読者　xi
関連する本　xi
サンプルコード　xii
本書での規約　xiii
　テキスト中の特殊フォント　xiii
　構文規約　xiv
デベロッパー製品とサポート　xv
　日本での連絡先　xvi
文書化されないシステムソフトウェアオブジェクト　xvi

CHAPTER 1　概要　1-1

イントロ　1-1
意味的な概要　1-2
　式 1-2
　オブジェクトモデル　1-2
　データ型とクラス　1-3
　スコープ　1-5
　有効範囲　1-7
　ガベージコレクション　1-7
　NewtonScript は、どのくらい動的か？　1-8
基本構文　1-9
　セミコロンセパレータ　1-9
　インラインオブジェクト構文　1-10
　文字セット　1-11
　コメント　1-11

コードのサンプル　1-12
互換性　1-13

CHAPTER 2　オブジェクト、式、演算子 2-1

オブジェクトとクラスシステム　2-1
クラスとサブクラス　2-4
イミディエイト値と参照値　2-6
NewtonScript オブジェクト　2-9
文字　2-9
論理値(ブール値)　2-11
整数　2-11
実数　2-12
シンボル　2-13
文字列　2-14
配列　2-16
配列アクセッサ　2-17
フレーム　2-18
フレームアクセッサ　2-20
パス式　2-22
式　2-24
変数　2-24
ローカル　2-24
定数　2-27
Constant 予約語　2-27
クォーテッド定数　2-29
演算子　2-30
代入演算子　2-30
算術演算子　2-32
等価、関係演算子　2-33
論理演算子(ブール演算子)　2-34

単項演算子　2-35
文字列演算子 2-36
Exists　2-36
演算子の優先順位　2-38

CHAPTER 3　制御構文3-1

複式 3-1
If...Then...Else　3-2
繰り返し　3-4
For　3-4
Foreach　3-7
Loop　3-11
While　3-12
Repeat　3-13
Break　3-14
例外処理　3-14
例外の使用　3-15
例外の定義　3-16
例外シンボルパート　3-17
例外フレーム　3-17
Try ステートメント 3-19
例外の発行(throw)　3-20
他のハンドラへの例外再発行(rethrow)　3-21
例外の捕獲　3-21
例外への対応 3-25

CHAPTER 4　関数とメソッド　4-1

関数とメソッドについて　4-1
関数コンストラクタ　4-2
Return　4-3

関数起動　4-3
メッセージ送信演算子　4-3
Call With 構文 4-6
グローバル関数宣言　4-6
グローバル関数起動　4-7
パラメタ渡し　4-8
関数オブジェクト　4-8
関数コンテキスト　4-10
字句環境　4-10
メッセージ環境 4-11
関数オブジェクトの例4-12
抽象データ型実装のための関数オブジェクトの利用　4-14
ネイティブ関数 4-16

CHAPTER 5　継承と探索　5-1

継承 5-2
プロトタイプ継承　5-2
プロトタイプフレームの生成　5-2
プロトタイプ継承ルール　5-3
ペアレント継承　5-4
ペアレントフレームの生成 5-4
ペアレント継承ルール5-5
プロトタイプとペアレント継承の組み合わせ　5-6
スロットとメッセージ探索の継承ルール　5-8
スロットの存在をテストするための継承ルール　5-10
スロットの値をセットするための継承ルール 5-10
オブジェクト指向な例　5-12

CHAPTER 6　組み込み関数6-1

互換性　6-2

新しい関数　6-2
新しいオブジェクトシステム関数 6-2
新しい文字列関数　6-3
新しい配列関数 6-3
新しいソートされた配列のための関数 6-3
新しいメッセージ送信関数 6-4
新しいデータ詰め込み関数 6-4
新しいグローバルなものを取得したり、セットする関数 6-4
新しいその他の関数　6-5
互換性の為に残っている既に時代遅れな関数　6-5
オブジェクトシステム関数 6-5
文字列関数　6-18
ビット演算関数 6-26
配列関数　6-26
ソートされた配列用の関数 6-38
整数演算関数　6-49
浮動小数演算関数　6-52
浮動小数環境の管理　6-71
財務関数　6-76
例外関数　6-77
メッセージ送信関数　6-80
データ抽出関数 6-82
データ詰め込み関数　6-87
グローバルな変数・関数の取得及びセット　6-92
その他の関数　6-95
関数とメソッドの要約6-98

APPENDIX A　予約語　A-1

APPENDIX B　文字コード表　B-1

APPENDIX C　クラスベースのプログラミング C-1

クラスのどこがいいのか?C-1
クラス: 簡単な復習　C-2
NewtonScriptでの継承　C-3
基本的なアイデア　C-4
実践的な情報　C-7
クラス変数　C-8
スーパークラス C-9
スープエントリをカプセル化するためにクラスを使う　C-10
ROM インスタンスプロトタイプ　C-11
インスタンスを背後に置く　C-11
最後に　C-11
作者について　C-12

APPENDIX D　NewtonScript 構文定義　D-1

文法について　D-2
構文文法　D-2
字句文法　D-12
演算子の優先順位　D-16

APPENDIX E　クイックリファレンス E-1

用語集　GL-1

訳者あとがき　EP-1

索引　IN-1

図、表、リスト

CHAPTER 1　概要 1-1

リスト 1-1　簡単なフレーム　1-3
図 1-1　データ構造のサンプル　1-4
リスト 1-2　動的なサンプル　1-12

CHAPTER 2　オブジェクト、式、演算子　2-1

図 2-1　NewtonScript 組み込みクラス　2-3
図 2-2　NewtonScript コードサンプル　2-7
図 2-3　C コードのサンプル　2-8
表 2-1　特別な意味を持つ文字　2-10
表 2-2　文字列内での特殊文字　2-15
表 2-3　特殊なスロット名と意味 2-20
表 2-4　定数への置換　2-28
表 2-5　演算子の優先順位と、結合規則 2-38

CHAPTER 3　制御構文　3-1

図 3-1　データオブジェクトとそれらの関係　3-10
表 3-1　foreach と foreach deeply の結果比較　3-11
表 3-2　例外フレームのデータスロット名と内容　3-17
表 3-3　例外フレームサンプル　3-18
リスト 3-1　例外シンボルの例　3-16
リスト 3-2　Throw()の使用例　3-20
リスト 3-3　順番の正しくないいくつかの onexception 節 3-22
リスト 3-4　正しく並んだ onexception 節　3-23
リスト 3-5　正しくない try ブロックのネスト　3-24
リスト 3-6　begin と end(bold で示す)を使って、ネストした try ブロックの問題を解消した例　3-24
リスト 3-7　スープストア例外　3-25

リスト 3-8　例外フレームをチェックする例外ハンドラ　3-25

CHAPTER 4　関数とメソッド 4-1

図 4-1　関数オブジェクトのパーツ　4-9
図 4-2　functionObject1 の解剖　4-13

CHAPTER 5　継承と探索　5-1

図 5-1　プロトタイプフレーム　5-3
図 5-2　プロトタイプチェーン　5-4
図 5-3　親子関係　5-5
図 5-4　プロトタイプとペアレント継承影響順序　5-7
図 5-5　継承構造　5-13

CHAPTER 6　組み込み関数　6-1

表 6-1　浮動少数例外　6-71
表 6-2　例外フレームデータのスロット名と内容　6-78

APPENDIX B　文字コード表　B-1

表 B-1　Macintosh 文字コード順の文字コード表　B-1

本書について

この “*The NewtonScript Programming Language*” は、プログラミング言語NewtonScriptを学ぶ全ての人のための最も完全なリファレンスである。

もし、Newtonプラットフォーム用のアプリケーションを開発しようとしているなら、まずこの本を読む必要がある。NewtonScript言語に慣れたら、実装の詳細を知るために “*Newton Programmer's Guide*” を読み、Newton Toolkitのインストール方法と使用方法を知るため “*Newton Toolkit User's Guide*” を読むとよい。Newton Toolkitは、Newtonプラットフォーム用プログラムを書くための開発環境である。

想定される読者

この本は、CやPascal等の高級言語の経験があり、オブジェクト指向プログラミングのコンセプトを理解しているプログラマのためのものである。

もしオブジェクト指向プログラミングに関してあまり通じていないなら、近所の本屋に行けば、このテーマに関する本がたくさんあるのでそれを読むように。

関連する本

この本は、Newton開発環境であるNewton Toolkitに含まれる本のセットのうちの一つである。

セット中の以下の本も参照する必要がある:

- *Newton Programmer's Guide: System Software.* この本のセットは通信以外のNewtonプログラミングトピックスに関するガイド及び参考書の決定版である。

- *Newton Programmer's Guide: Communications.* この本は、Newton 通信プログラミングに関するガイド及び参考書の決定版である。
- *Newton Toolkit User's Guide.* この本では、Newton プログラミング環境への導入と、Newton Toolkit を使用しての Newton アプリケーション開発の方法が収録されている。Newton アプリケーション開発の初心者なら、この本を最初に読むと良い。
- *Newton Book Maker User's Guide.* この本では、Newton デジタルブック及び、Newtonアプリケーションに追加されるオンラインヘルプを作るに当たっての、Newton Book Maker と、Newton Toolkit の使い方を説明している。Book Maker 込みで Newton Toolkit を購入したときだけこの本が付いてくる。
- *Newton 2.0 User Interface Guidelines.* この本には、Newtonデバイスとユーザーの対話を最適にするNewtonアプリケーションのデザインを手助けするガイドラインを収録している。

サンプルコード

Newton Toolkit の製品版には沢山のサンプルコードプロジェクトが含まれている。それらのサンプルを検証し、そこから学習し、それらを試して、自分自身のアプリケーションの出発点とすればよい。それらサンプルコードプロジェクト群は、この本でカバーされるトピックスのほとんどを例示する。これらは、この本で語られるトピックスを理解するための非常に貴重なリソースであると同時に、Newtonプログラミングの世界への長い旅をたやすくするためのものである。

Newton開発者サポートチームは過去のサンプルをアップデートし、新しいサンプルを作る作業を継続している。eWorld の Newton 開発者エリアには、最新のサンプルコードのコレクションがある。Newtonデベロッパーサポートプログラムに参加することで、サンプルコードにアクセスする事もできる。ページxvの「デベロッパー製品とサポート」には、Newton デベロッパーサポートプログラムに関して Apple にコンタクトする方法が示されている。

本書での規約

本書では、情報を表すためいくつかの規約を用いる。

テキスト中の特殊フォント

次の特殊フォントが使われる:

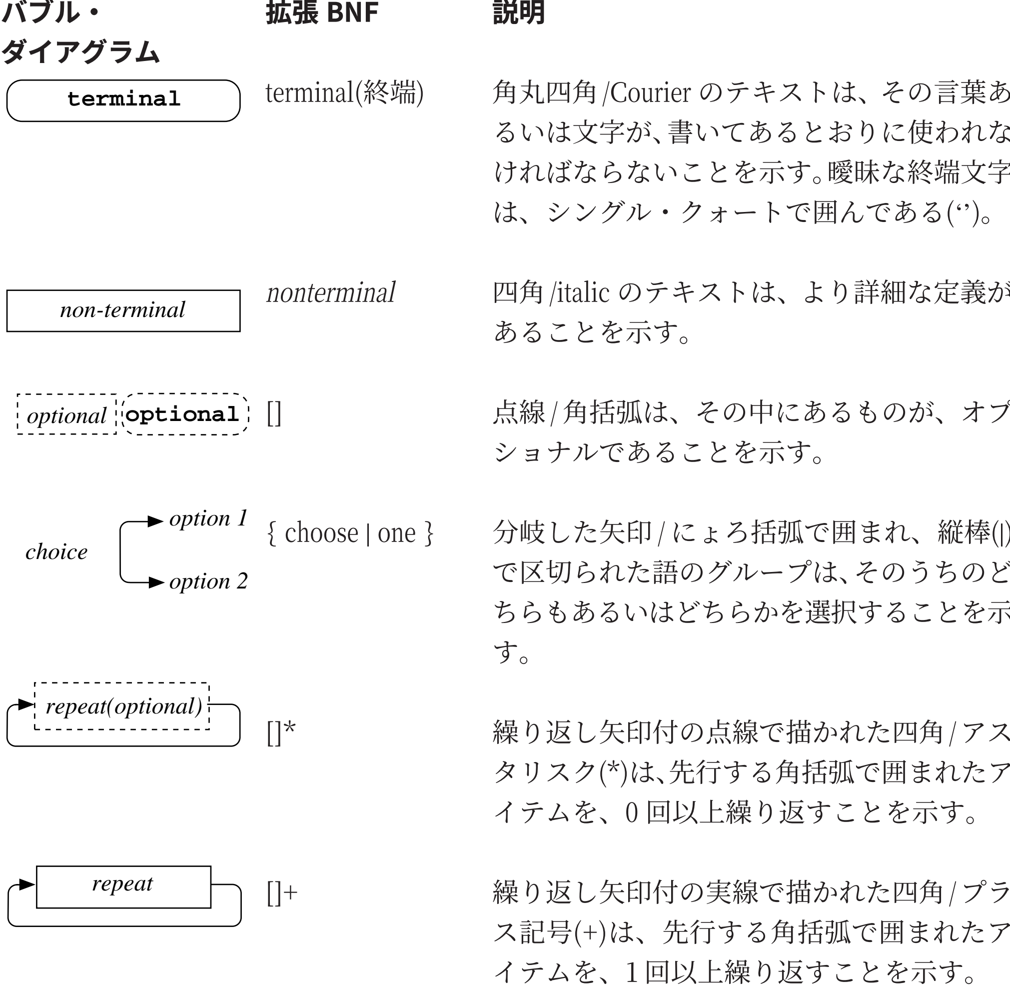
Boldface 体 キーとなる用語及びコンセプトが最初に現れた時に、boldface又はゴシックで表示される。これらの用語は用語集でも定義されている。

`Courier` コードリスティング、コードの断片、テキスト中の特別な識別子(定義済みシステムフレーム名、スロット名、関数名、メソッド名、シンボル、定数)は、通常のテキスト本体と区別するためにCourier書体で表示される。プログラム内では、Courierで表示されるアイテムは、その通りにタイプしないといけない。

Italic 斜体は、独自の名前に置き換えないといけないような関数パラメタの名前のような置換可能なアイテムを表示するためにコード中で使われる。書名もまたitalic書体で表示される。まれに強調のために使われることもある。

構文規約

このドキュメントでは定義を拡張 BNF と、Pascal 構文チャートで示す。それぞれ以下のように定義される。



バブル・ダイアグラム	拡張 BNF	説明
terminal	terminal(終端)	角丸四角/Courier のテキストは、その言葉あるいは文字が、書いてあるとおりに使われなければならないことを示す。曖昧な終端文字は、シングル・クォートで囲んである(‘’)。
non-terminal	*nonterminal*	四角/italic のテキストは、より詳細な定義があることを示す。
optional optional	[]	点線/角括弧は、その中にあるものが、オプショナルであることを示す。
choice *option 1* *option 2*	{ choose | one }	分岐した矢印/にょろ括弧で囲まれ、縦棒(|)で区切られた語のグループは、そのうちのどちらもあるいはどちらかを選択することを示す。
repeat(optional)	[]*	繰り返し矢印付の点線で描かれた四角/アスタリスク(*)は、先行する角括弧で囲まれたアイテムを、0 回以上繰り返すことを示す。
repeat	[]+	繰り返し矢印付の実線で描かれた四角/プラス記号(+)は、先行する角括弧で囲まれたアイテムを、1 回以上繰り返すことを示す。

デベロッパー製品とサポート

APDA(現在はADC: Apple Developer Catalogと改称)は、Appleコンピュータプラットフォームのためのアプリケーション開発に興味のある人々のために、無数の開発ツール、テクニカルリソース、トレーニング製品と情報を提供する世界規模の情報源である。ADCは顧客にアップル及びポピュラーなサードパーティによる開発ツールの最新バージョンを収録したApple Developer Catalogを配布している。ADCは、サイトライセンシングを含む、支払・出荷をより便利にするオプションをも提供している。

製品もしくはApple Developer Catalog(無償)を注文するには、以下の宛先を利用すること:

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, New York 14207-0319
U.S.A.

U.S 1-800-282-2732
Canada 1-800-637-0029
International .... (716) 871-6555
Fax .... (716) 871-6511

E-Mail: order.adc@apple.com
WorldWide Web: http://www.devcatalog.apple.com

もし商業製品とサービスを供給したいので有れば、Appleのデベロッパーサポートプログラムに関する情報を得るため、408-974-4897に電話すること。

日本での連絡先

株式会社バイス
DeveloperDepot Japan(DDJ)
Tel.03-5802-0755
Fax.03-5802-0756
営業時間帯　平日 10:00 ～ 17:00
E-Mail:　　devdepotj@byse.co.jp
URL: http://www.byse.co.jp/adc/

Developer Depot 及び Developer Depot Japan は、Xplain Corporation の登録商標です。

文書化されないシステムソフトウェアオブジェクト

NTKのインスペクタウィンドウをブラウズしていくと、この本には書かれていない関数、メソッド、データオブジェクトを発見することと思う。文書化されていない関数、メソッド、データオブジェクトは、サポートされないかまたは、将来の Newton デバイスで動作することを保証されているものである。それらを使うことは、現状のNewtonデバイスでは予期せぬ効果を生ずることがある。

CHAPTER 1

概要

NewtonScript は Newton プラットフォーム用に開発された最先端の、動的な、オブジェクト指向プログラミング言語である。

イントロ

NewtonScript の目標は、開発者が早く、スマートなアプリケーションをたやすく作れるようにすることにある。そのため、言語には以下の性質が必要となった:

- 表現性があり、柔軟であり、使用が簡単である
- コンセプトと構造を再使用可能にするための十分な一貫性
- 異なるアーキテクチャーの探求を容認する可搬性
- 限られた容量の RAM で動くための十分なコンパクト性

Newton システムの制約は、メモリを効果的に使い、ガベージコレクションを自動的に行う再利用可能なコードライブラリが生成できるといった能力を言語に要求した。

NewtonScript は、Smalltalk や LISP で最初に使われた原理に基いている。そしてそれら自身も、スタンフォード大学で開発された言語に影響を受けている。

意味的な概要

このセクションでは NewtonScript 言語の、特殊ないくつかの機能を簡単に紹介する。

式

他の多くのプログラミング言語が文(ステートメント)ベースの言語であるのに対して、NewtonScript は式ベースの言語である。NewtonScript 中のほとんど全てのものは、値を返すので、このマニュアルではステートメントとかコマンドではなく、むしろ「式」について話をする。

訳注: NewtonScript では、`if` や `while` 等の制御構造ですら値を返す式である。

オブジェクトモデル

NewtonScript は、オブジェクトモデルに基づいている。全てのデータは**オブジェクト**あるいは、型付けされたデータの断片として格納される。これは C++ とか Object Pascal のような(データが、オブジェクトと通常の型付きデータのあいのこのようになっている)他のオブジェクト指向言語とは異なっている。

NewtonScript はまた、Smalltalk とも違ってはいるが、Smalltalk のように、全てのデータをオブジェクトとして表現する。NewtonScript において、メッセージを受け取ることが出来るのは、フレームという形態のデータただ一つである。

Newton の オブジェクトモデルは、オブジェクトを表現するために、二種類の 32bit 値を用いてデータを構成する。その値とは次のものである:

- **イミディエイト(即値)**–32bit の中に直接値が入っている
- **リファレンス(参照)**–32bit ポインタで、間接的にオブジェクトを参照する

これは Chapter 2 「オブジェクト、式、演算子」、page 2-6 から始まるセクション 「イミディエイト値・参照値」で、より詳細に説明される。

訳注: **イミディエイトと、リファレンスについては訳語の点でかなり悩んだ。イミディエイトについては「即値」という従来からの言葉があるが、訳者としてはこの言葉があまり良い言葉であるように思えないので、そのまま「イミディエイト」とさせてもらった。また、リファレンスについては、「参照値」・「参照型」などの言葉を使っている。**

データ型とクラス

NewtonScript での**クラス**は、特定の意味をデータ型に与えるものだ。Newton プラットフォームは、クラスをシステムの一部分として使う。たとえば Intelligent Assistant(これは *Newton Programmer's Guide* で説明する)では、オブジェクトの性質を実行時に決定する。こうして、特定のタイプのオブジェクトは興味深くかつ異なる方法で取り扱われる。

一つの例として、リスト 1-1 の個人情報を表すデータ型を考えてみる。これはにょろ括弧(カーリーブラケット)で囲まれているので、「フレーム」と呼ばれるデータ構造であることがわかる。フレームの中には、name, company, phones という名前のスロットが存在する。

リスト 1-1　　簡単なフレーム

```
{       name:           "Walter Smith",
        company:        "Apple Computer",
        phones:["408-996-1010", "408-555-1234"]
}
```

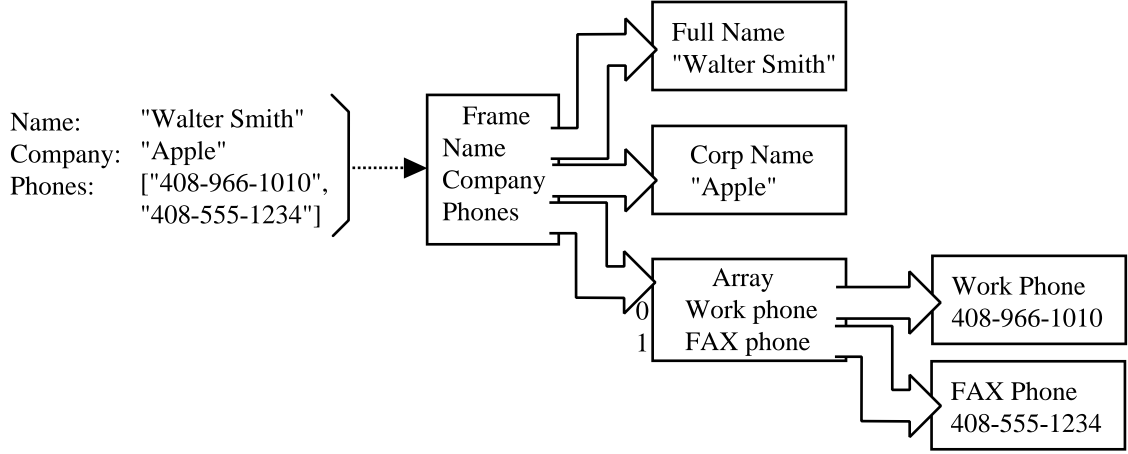
この場合、オブジェクトは型付けされ、name と company の値は文字列型となるが、さらに phones は、workPhone や faxPhone 型としても定義できる。こうしたユーザー定義型の導入によって、ただの文字列を別な方法で取り扱うことが出来る。例えば、だれかが自分の作ったアプリケーションを使って faxPhone を使うとき、普通の電話番号に対して発生するアクションの集合とは異なるアクションが起こる。

リスト 1-1 で構成されたデータオブジェクトを図 1-1 に示す。

上の例では name, company, phones に対して(int や char や char[ ][ ] のような)型指定の構文は現れていないが、暗黙の内に型が決まっている(latent typing: 隠れた型付け)。NewtonScript の変数は表面上型を持たないが、中に入るオブジェクトによって、動的に暗黙の型付けがなされる。つまり、変数は必要に応じていろんな種類のオブジェクト、異なる型のオブジェクトを自由に格納出来ることを意味する。

page 2-1 から始まる「オブジェクトとクラスシステム」で、クラスシステムの詳細を説明する。

図 1-1　　データ構造のサンプル



注意
先にも述べたが、NewtonScript のクラスと、Smalltalk のクラスは全然別物である。ただ、Smalltalk におけるクラスベースのプログラミングのコンセプトを部分的に使うことは出来る。その詳しい方法は、Appendix C に書かれている。◆

スコープ

ある変数を使用できるプログラムの部分のことを、変数のスコープという。通常、変数はそれが宣言された関数内で有効であるが、変数のように使うことのできるスロットは、プロトタイプやペアレントフレームから継承してくることができる。

フレームと**継承**についてはそれぞれ page 2-18 から始まるセクション「フレーム」と、Chapter 5 「継承と探索」を参照のこと。

変数の値が探索されるとき、NewtonScript はまず**ローカル**変数を探し、次に**グローバル**変数を探し、最後にプロトタイプ・ペアレント継承チェーンによる継承された変数を探す。

次のようなコードセグメントを考えてみて欲しい:

```
aFrame:= {
        foo: 10,
        bar: func(x) begin
                if foo then Print ("hello");
                if x > 0 then begin
                        local foo; //local variable to function
                        foo:= 42;
                end;
                return foo;
        end;
}
```

この場合、ローカル変数 `foo` のスコープは、C ののりで考えれば、`if x > 0 ... end` のブロック中になるような気がするが、NewtonScript ではそうした制限はなく、`foo` のスコープは、関数 `bar()` の中である。

さらに C ののりで考えるなら、`local foo;` という行が出現する前に `if foo then...` で `foo` を評価しているので、コンパイルエラーが生じるか、その外側にあるスロット `foo` が参照されるように思うが NewtonScript ではそうではない。このケースでは、`foo` が `if` 文で参照された時、既に `nil` という値を持つローカル変数としてコンパイラによって宣言されてしまっているのである。

では、この場合 `aFrame` フレーム内で立派に宣言されている初期値 10 のスロット `foo` を参照するには、次のように明示的に、`foo` がフレーム内のスロットであることを指定すればよい:

```
if self.foo then Print ("hello");
```

`self.foo` でなく、`foo` とだけ書いた場合は、先に述べたようにローカル変数→グローバル変数→プロトタイプ・ペアレント継承チェーン内(それは自分自身も含む)のスロットという順番で探索が行われる。

上記 `aFrame` の例では、何も印字されない。なぜなら、`foo := 42` の部分が実行されるまで、メソッド `bar` のローカル変数 `foo` の値は `nil` としてコンパイラによって生成されたままだからだ。

`aFrame` のメソッド `bar` を、パラメタの値 10 で呼び出してみよう:

```
aFrame:bar(10)
```

この場合は、値 42 が戻ってくる。

```
aFrame:bar(-5)
```

の呼び出しに対しては、`nil` が戻ってくる。

メッセージ転送の詳細については、Chapter 4 の「関数の起動」を参照のこと。

有効範囲

変数の有効範囲(というか生存期間)はCやPascal等の多くの言語では通常スコープと同じである。しかし、NewtonScriptでは、実行中のコードのどこででも、そのオブジェクトが参照できる限り、オブジェクトが死ぬことはない。オブジェクトがもうどこからも参照されなくなってはじめて、自動的なガベージコレクタ(ごみ収集)によって回収される。よって、データ構造のためのメモリ領域は、それへの参照がなくなるまで有効である。

もはや不要な大きなデータ構造への参照をそのまま残さないように注意したい。もしそれをそのままにしておくと、NewtonScriptはそのデータ構造に関連するメモリを回収できなくなってしまう。

Newtonのアプリケーションにおいて、このオブジェクトの生存期間について考慮しないといけないポイントはどこか？その一つの例はアプリケーションの終了時である。アプリケーションが終了するときに、ベースビューの中の値を`nil`にする事で、アプリケーションが動いてないときでも無駄なメモリを確保してしまうのを抑えることが出来る。

ガベージコレクション

ガベージコレクタは、システムによって自動的に起動され、誰も参照していないメモリブロックを回収して回る。NewtonScriptでは、再現性のないバグを生じる「宙ぶらりんのポインタ」は作り出せない(Cではこれが実に簡単に作り出せ、悩みの種となる)ので、読者はメモリマネジメントについてはあまり心配しなくても良い。

もはや不要になったオブジェクトを回収してもらうには、そのオブジェクトを参照している全てのスロットや変数の値を`nil`にしてやるだけでよい。こうすることで、ガベージコレクタがずかずかとやってきて、そのオブジェクトが占有していたメモリ領域を回収してくれる。

ガベージコレクションは、システムがメモリを使いきったときに勝手に行われるので、ユーザーがそれを明示的に行う必要はないが、どうしてもやりたければ `GC()` という関数を呼び出せばよい。この関数について詳しくは、*Newton Toolkit User's Guide* の「デバッギング」を参照のこと。

訳注: これは人から聞いた話だが、OMP → NMP100 → NMP110 → NMP120 と、OS や CPU が変わらないにも関わらず処理速度が向上しているのは、内部メモリ(ヒープ)が増加したため、ガベージコレクションの頻度が下がり、結果として処理速度が向上している為だという。

NewtonScript は、どのくらい動的か？

「動的」という言葉はいろいろな局面で使用されるが、NewtonScript の場合は、「実行時にオブジェクトの属性をどの程度変更できるか」という意味合いで使われる。実際、必要ならアプリケーション実行中にオブジェクトを別の種類のオブジェクトに変えてしまうこともできる。

アプリケーションの使用中に新しいデータをオブジェクトに追加することもできる。たとえば、実行中に動的に新しい変数を、実行オブジェクトに追加し、それを使い、同じオブジェクトにメソッドを追加してそれを使い、しまいには別のオブジェクトへの特別な参照を追加することによって、継承構造を変更してしまうような NewtonScript コードを書くこともできる。(この「特別な参照」は、page 1-12 内、リスト 1-2 の中の `_parent` として示されるものである)

これらのオペレーションは、静的な言語では不可能なものだし、また、動的言語においてもかなりのな思考と、訓練を必要とするものである。しかし、このパワフルな機能は静的言語のやり方では不可能な対話的プログラムを可能にしてくれるが、この機能は控えめに、かつ注意して使うべきだ。

基本構文

全く新しい文法を創作するのではなく、NewtonScript は、Pascal を念頭にデザインされた。可能な限り、そのシンタックスは、Pascal のそれと近くなるように作られている。

訳注: これはちょっと誤解を招く表現で、読者(特に Pascal プログラマ)に余計な先入観を与えてしまう可能性があると訳者は思っている。細部まで見ていくとそれほど Pascal には似ていないことがわかる。

セミコロンセパレータ

各式はセミコロンで区切る。C ではなく、Pascal のような区切りかたをすると思えばよい。セミコロンはあくまで、式を区切るものだ。

NewtonScript においてもコードの入力はフリーフォーマットだが、やっぱり次のようなインデントはしておいた方が可読性は向上すると思う。

```
if condition then
       expression1
else
       expression2;
```

空白(タブ、改行なども含む)はそれが置けるところならどこでも好きなだけ置いて良い。それはコンパイラによって無視されるからである。

注意

NewtonScript のインタープリタは「式」というものを好意的に解釈しようとするので、もしセミコロンが抜けているとそれ以降のものをも式の一部として取り込もうとしてしまう。よって「エラーにはならないが、変な値を返す式」というものを構成することがあるため注意すること。◆

インラインオブジェクト構文

(ここで説明するのは早すぎるような気もするが、)NewtonScriptには、オブジェクトリテラルと、オブジェクトコンストラクタという二つの構文機能がある。

オブジェクトリテラル構文を使うと、複雑なオブジェクトを、まるで整数を扱うように簡単にプログラムに組み込むことができる。この構文モードは、次の文字列を二つ含むフレームの簡単な例に見られるように、シングルクォートによって開始される。

```
x := '{name: "xxx", phone: "yyy"};
```

この場合、オブジェクトはコンパイル時に構築されるので、同一オブジェクトへの参照結果は、常にそのオブジェクトリテラルの評価結果となる。

一方オブジェクトコンストラクタ構文は、オブジェクトリテラルと似ているが、シングルクォートを使わない。これは、実行時にオブジェクトを構築するのに使う。

オブジェクトコンストラクタの場合は、それへの評価が起こる度に新たなオブジェクトが生成されるので、生じる結果は毎回別のオブジェクトとなる。

オブジェクトコンストラクタの例を示す:

```
x := {name: first && last, tax: wage * taxRate};
```

このオブジェクトコンストラクタの例を、別の書き方でやってみると次のようになり、オブジェクトコンストラクタの方が読みやすい事がわかるだろう。

```
x := {};
x.name := first && last;
x.tax := wage * taxRate;
```

文字セット

NewtonScript の文字セットは、Mac における拡張アスキー文字でなく、7bit アスキーである。これはコードがどのNewton開発環境でも動くことを意味する。

コメント

NewtonScript のコメントは、C++ のコメントと同じである。つまり、複数行にわたるものは、/* と */ の間に書く。ただし入れ子には出来ない。

```
/* this is a long
  type comment */
```

短いコメントには、// を使える。// が出現するとそこから改行までがコメントとみなされ、無視される。

```
x := 6; // this is a short type comment
```

ただし、// タイプのコメントを、/*...*/ タイプのコメントの中で使うことは出来る。

コードのサンプル

このマニュアルを読む前にコードを理解してみたいなら、このセクションを続けて読んで欲しい。そうでなければここは飛ばして次の章に行くこと。

注意すべきは、にょろ括弧({ })はフレームオブジェクトを意味し、コロン–等号(:=)は代入演算子を意味し、コロン(:)は、その前に示されるフレーム式に対してコロンの後に続くメッセージを送るメッセージ送信演算子を意味することである。

リスト1-2に示されるコードは、page 1-8から始まる「NewtonScriptはどのくらい動的か?」で少し触れられている。

リスト 1-2　動的なサンプル

```
y := { YMethod: func () print("Y method"), yVar: 14 };
x := { Demo: func () begin
       self.newVar := 37;
       print(newVar);
       self.NewMethod := func () print("hello");
       self:NewMethod();
       self._parent := y;
       print(yVar);
       self:YMethod();
       end
};
x:Demo();
37
"hello"
14
"Y method"
#2        NIL
```

互換性

NewtonScript バージョン 2.0 には、次の二つの主要な拡張がなされた:

- 新しいサブクラス機構
- ネイティブ関数

新しいサブクラス機構については page 2-4 のセクション「クラスとサブクラス」で述べる。このサブクラス機構により、ユーザー定義クラスはより正確な意味の定義ができるし、カテゴリの論理的な構造を保存できる。

Page 4-16 のセクション「ネイティブ関数」で述べられるネイティブ関数は、インタープリタを通さず、直接Newtonのプロセッサ上で実行される。これは、自分の関数のスピードを上げることも有れば、低下させてしまうこともある。

さらに、ネイティブ関数の処理スピードをアップするために、二つの型識別子 `int` と `array` が言語に追加された。これら型識別子は、ローカル変数定義と、関数の引数リスト定義の2つの場所で使用される。そのシンタックス情報については、page 2-24 の「ローカル」セクションと、page 4-2 の「関数コンストラクタ」を参照のこと。ネイティブ関数と型識別子の使用に関する詳細は、*Newton Toolkit User's Guide* の 「パフォーマンスのチューニング」の章を参照のこと。

言語のバージョン 2.0 はまた、新しい組み込み関数を持っている。それらのリストに付いては、Chapter 6 「組み込み関数」 の page 6-2 から始まる「互換性」セクションを参照のこと。

空のページ

CHAPTER 2

オブジェクト、式、演算子

この章では、オブジェクト、式、演算子について述べる。

オブジェクトとクラスシステム

オブジェクトの意味的な型は、**クラス**(しつこいようだが、Smalltalk のそれとは違う)により指定される。Newton のオブジェクトシステムは、4 つの組み込みプリミティブクラスを持ち、オブジェクトの基本的な型を示す。それらは以下の通りである:

- イミディエイト(即値)
- バイナリ
- 配列
- フレーム

オブジェクト obj のプリミティブクラスは関数 PrimClassOf(obj) で調べることができる。同様に、オブジェクト obj のクラスは ClassOf(obj) で調べることができる。

オブジェクトが個々の型のものかどうかチェックする関数もいくつか用意さ

れている。それは IsArray, IsFrame, IsInteger, IsSymbol, IsCharacter, IsReal, IsString というもので、PrimClassOf, ClassOf より動作が早い。これら及び、その他の Newton 組み込み関数については、Chapter6 「組み込み関数」に述べられている。

プリミティブクラスは、イミディエイト(即値)と参照型(リファレンス)の二つのカテゴリから構成される。参照オブジェクトカテゴリは**バイナリ**、配列、フレームからなる。これら二つのカテゴリ間の違いについては、Page 2-6 「イミディエイトと参照値」に詳しく述べられている。

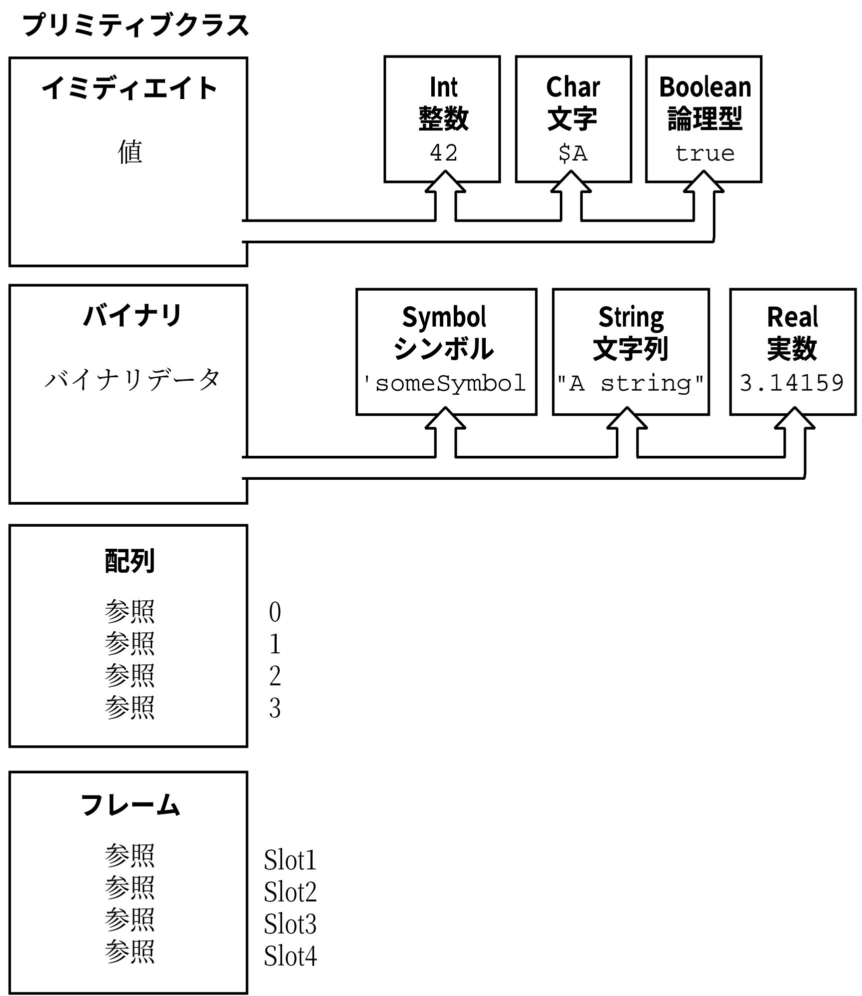
イミディエイトをプリミティブクラスに持つオブジェクトは、クラス Int, Char, Boolean に分類される。バイナリをプリミティブクラスに持つシステム定義のオブジェクトは Symbol, String, Real のいずれかに属する。

参照オブジェクトの場合は、ユーザー定義クラスも指定可能である。

NewtonScript クラス構造は図 2-1 に示す。

NewtonScript でのクラス機能は、あくまで意味的なもので、システムに対して、オブジェクトのデータの意味を通知するものだ。たとえばクラス 'string は、そのオブジェクトが文字列格納のためのバイナリオブジェクトであることを示し、クラス 'phoneNumber は、それが電話番号を格納する文字列であることを示す。こうしたクラス分けによって、Newton デバイスは、文字列と電話番号文字列を別の形態で(例えば電話ならダイヤルしたりして)扱うことになる。

図 2-1 NewtonScript 組み込みクラス



クラスとサブクラス

`SetClass(`*obj, classSymbol*`)`関数を使うと、参照オブジェクトにクラス*classSymbol*を適用することが出来る。配列とフレームには、ユーザー定義クラスをセットするための内部機構があるが、バイナリオブジェクトに関しては、`SetClass`を使わないといけない。これらのメカニズムは、page 2-16「配列」と、page 2-18「フレーム」で説明される。

クラスシンボルは階層的に配置される。つまり、あるクラスはサブクラスを持つという事で、これにより、オブジェクトがそれらの定義の論理的な構造を保持しながら、より正確な意味定義を持つ事が可能となる。

サブクラスを生成するには、クラス名にピリオド(`.`)とサブクラス名を追加する。たとえば、`'|rectangle.square|`は、`'rectangle`のサブクラスである。XとYが全く同じものである場合、あるいはY.Xのような場合はクラスXはクラスYのサブクラスであるという。あらゆるものは空のシンボル`||`のサブクラスである。

ピリオドの後に追加するシンボルは、当然それ自身がピリオドを含んではならない。また、シンボルはセミコロン(`;`)も含んでいてはいけない(将来の拡張のため予約されている)。

さらに、ピリオドを含むクラスシンボルは、縦棒(`|`)に囲まれてないといけない。これはシンボルの構文(Page2-13 「シンボル」で述べられている)上必要なものである。

組み込み関数`IsSubclass(x, y)`は、`x`が`y`のサブクラスかどうかを調査する。また、`IsInstance(obj, x)`は、オブジェクト`obj`がクラス`x`あるいはクラス`x`のサブクラスのオブジェクトかどうかを調査する。これは`IsSubclass(ClassOf(obj), x)`を短くしたものである。これらの関数の詳細についてはChapter 6 「組み込み関数」を参照の事。

注意

ピリオドによるサブクラス生成手法は、NewtonScriptの新しい機能であるため、バージョン1.xのNewtonデバイスではサポートされない。よって、OS1.xのマシンで動くアプリケーションを作る場合は、この機構は使わないこと ◆

オブジェクトにクラスを追加すると、アプリケーションの複雑さが増加するので、どうしても必要な場合以外はオブジェクトにクラスを追加しないほうがいいと思う。

さらに注意すべきこととしては、このセクションで説明された組み込みプリミティブクラスと、そこから派生するクラスの間には、何のサブクラス関係も無いと言うことがある。たとえば、`String`は、バイナリオブジェクトのサブクラスではない。

そのサブクラスがNewtonScirpt組み込み関数にとって重要なのは`'string`クラスだけである。その引数として、`'string`クラスまたは`'string`クラスのサブクラスを要求する文字列処理関数がいくつか存在する。

自動的に`'string`のサブクラスであるとNewtonシステムによって理解されるクラスシンボルがいくつかある。それは、`'company`, `'address`, `'title`, `'name`, `'phone`, `'homePhone`, `'workPhone`, `'faxPhone`, `'otherPhone`, `'carPhone`, `'beeperPhone`, `'mobilePhone`である。

たとえば`'firstName`の様なものを`'string`のサブクラスとして生成したい場合、明示的に`'|string.firstName|`と定義する。

イミディエイト値と参照値

先に述べたように、値は32ビットで格納され、その内2ビットがクラス情報として使用される。**イミディエイト**オブジェクト(整数、文字、論理値)は、残り30ビットの中にその値を格納する。一方**参照**オブジェクト(バイナリ、配列、フレーム)はデータが存在するメモリエリアへの「参照」を格納する。

変数への値の代入において重要な違いが有ることを覚えて置いて欲しい。変数にイミディエイトオブジェクトを代入するとき、値は直接変数にコピーされる。変数に参照オブジェクトを代入するとき、そのオブジェクトへの参照だけがコピーされるに過ぎない。

このことは、時に混乱を引き起こす。次のコードを考えてみる:

```
local a := {x: 1, y: 3};
local b := a;
a.x := 2;
// at this point b.x = 2
```

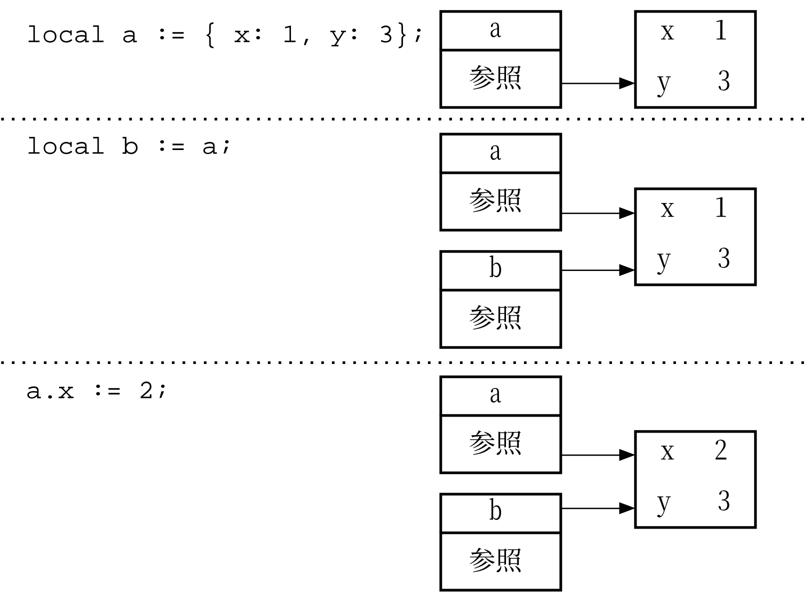
最初の行でローカル変数`a`が定義され、二つのスロット(`x`, `y`という名前で値はそれぞれ`1`と3)を持つフレームが代入される。

次の行では、もう一つのローカル変数`b`が生成され、そこに`a`の値(それは、最初の行で生成されたフレームオブジェクトへの参照である)が代入されている。こうして、変数`a`と`b`は同じメモリエリアを参照することになる。

3行目(`a.x := 2`)では、`x`スロットの値を変え、`a`と`b`は同じフレームを参照しているので、`b.x`の値も、それが明示的に代入されていないにも関わらず`2`となる。

この結果は図2-2に見ることが出来る。

図 2-2 NewtonScript コードサンプル



C のコードについて考えてみよう。

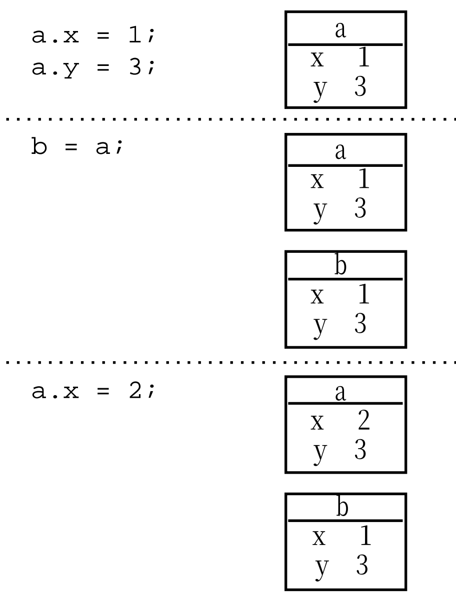
```
struct foo {
       int x,y;
};
struct foo a;
struct foo b;
a.x = 1;
a.y = 3;
b = a;
a.x = 2;
// at this point b.x = 1
```

訳註: C++ じゃないのか？

この例では、a, b という二つの別の構造体オブジェクトが別のメモリエリアに存在する。各構造体は二つの整数 x, y を含み、a.x には 1 が代入される。

構造体ａの値がｂに代入され、a.xが２にセットされても、期待通りb.xの値は１のまま変化することはない。この結果は図 2-3 に示す。

図 2-3　C コードのサンプル



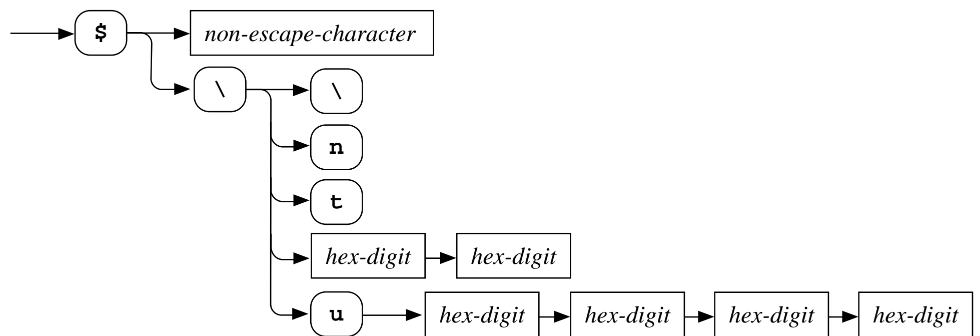
注意すべきは、NewtonScirpt での参照オブジェクトの代入は、C における配列や文字列(C ではそれらはポインタである)の代入と同じだと言うことである。

NewtonScript オブジェクト

このセクションでは、Page2-3 の「NewtonScript 組み込みクラス」に示される、文字、論理型、整数、実数、シンボル、文字列、配列、フレームといった個々のオブジェクトについて説明する。

文字

character:
$ { *non-escape-character* |
\ { \ | n | t | *hex-digit hex-digit* | u *hex-digit hex-digit hex-digit hex-digit* } }



標準文字コードの文字は、ドルマーク($)のあとに、以下のいずれかを続けて指定する。

- バックスラッシュエスケープ文字(\)の後に \ , n , t あるいは 2 個の 16 進文字等の特殊文字指定が付いたもの
- バックスラッシュエスケープ文字(\)の後に、u (unicode の意味)と、4 つの 16 進文字が付いたもの。
- 非エスケープ文字

Newton の文字セットは 2 バイトの unicode で格納される。デザイン的には、最初の 128 文字は ASCII 文字セットとマッチするが、それ以外は unicode 文字

を使わなければならない。

文字は、イミディエイトオブジェクトである。(イミディエイトオブジェクトについて、より詳しくは、Page2-6「イミディエイト値と参照値」参照。)

nonEscapeCharacter コード 32-127 の ASCII 文字のセット。ただし、バックスラッシュ(\)は除く。

hexDigit { 0|1|2|3|4|5|6|7|8|9 |a|b|c|d|e|f|A|B|C|D|E|F }から構成される。

たとえば、`$a` と `$7` は、それぞれ文字 “a” と “7” を意味する。

“π” のような特別な文字は、`$\u` の後に 4 つの 16 進数が続く unicode(16-bit)で指定する必要がある。たとえば、“π” は unicode では `$\u03c0` と同じである。

改行は `$\n` を文字列に埋め込むことで指定できる。特殊文字の要約は、表 2-1 に示す。

表 2-1 特別な意味を持つ文字

文字コード	意味
`$\n`	改行
`$\t`	タブ
`$\\`	バックスラッシュ
`$\` *hexDigit hexDigit*	16 進数
`$\u` *hexDigit hexDigit hexDigit hexDigit*	Unicode

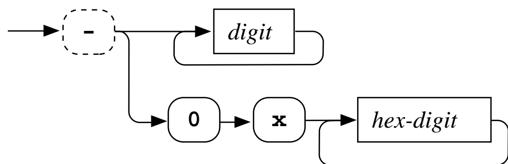
特殊文字と、Unicode の対応については、Appendix B 「特殊文字コード」を参照のこと。

論理値(ブール値)

論理値定数としては`true`だけが定義されている。関数と制御構造にとってのfalseはnilであり、nil以外のものは全てtrueとして扱われる。trueとして使える適当なものがない場合は、特殊なイミディエイトtrueを使うこと。

整 数

[-] { [*digit*]+ | 0x [*hex-digit*]+ }



全ての整数は10進数あるいは16進数で記述できる。もし、10進数に0xが先行している場合は、それは16進数と解釈される。オプションのマイナス符号(–)を10進数の前に付けると、負の整数を意味する。整数の例をいくつかあげる。

```
13475    -86  0x56a
```

整数の範囲は536870911から–536870912までである。上限下限を越えた場合の動作は未定義である。整数はイミディエイトオブジェクトである。

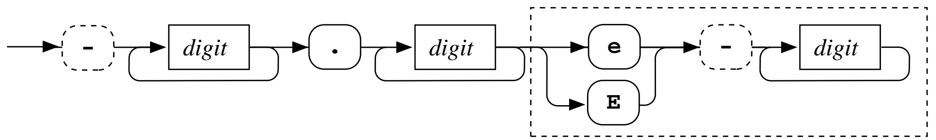
digit { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }

注意

整数の–536870912はリテラルとしては指定できないが、計算は出来る ◆

実数

[-] [*digit*]+ . [*digit*]* [{ e | E } [-] [*digit*]+]



実数は一つ以上の数字に小数点が付いたものか、それにさらに複数の数字が付いたもので構成される。最初の位置に付くオプションのマイナス(–)は、それが負の値であることを意味する。

最後の数字の後に文字 e(または E)を付け、科学的表記もでき、その乗数表現範囲は –308 から +308 の範囲である。

digit { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }

浮動小数形式の実数は内部的には 64bit 倍精度で表現され、およそ 15 桁の精度である。実数の例をあげる:

```
-0.587   123.9    3.141592653589
```

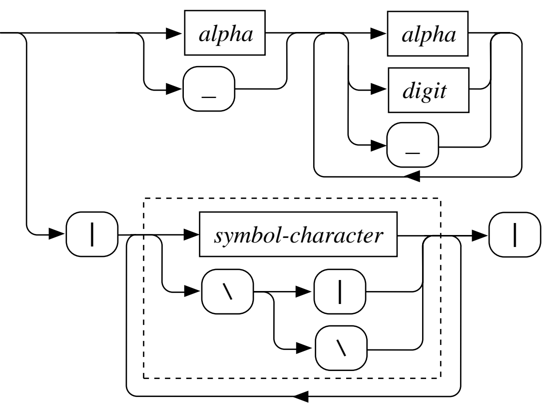
また、指数表記のサンプルも示す:

```
763.112e4    87.3789E-45   -34.2e6  69.e-5
```

実数はバイナリオブジェクトとして格納され、クラス real となる。

シンボル

{ { *alpha* | _ } [{ *alpha* | *digit* | _ }]* | '|' [{ *symbol-character* | \ { '|' | \ }]* '|' }



シンボルは識別子として使用されるオブジェクトである。シンボルは変数、クラス、メッセージ、フレームスロットに名前を付けるために使用する。単に値を指定して、他言語では列挙型として知られるような使い方もできる。

シンボル名の長さは254文字までで、プリント可能なASCII文字から出来ていないといけない。たとえば`|Weird%Symbol!|`というシンボルは正しい。シンボルが英字もしくはアンダースコアから始まり、英字・アンダースコア・数字だけを含むような形式で有れば、縦棒で囲まずにそれだけで書いても良い。大文字小文字の区別はされないが、大文字小文字の情報は保存する。

alpha	{`A-Z` and `a-z`}
digit	{ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` }
symbolChar	32-127 までのアスキー文字全部。ただし | と \ は除く。

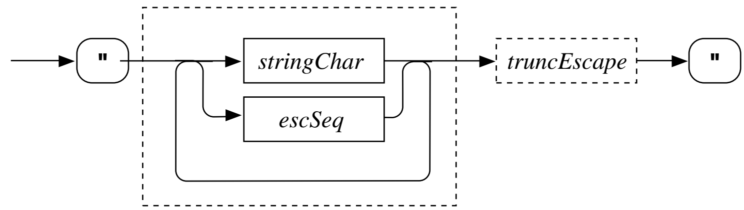
シンボルが必要な場所を一つあげるとすれば、それは例外処理である。例外シンボルは例えば次のようなものである`|evt.ex.fr.intrp|`。

シンボルの中にドットが含まれる場合は、縦棒が必要である。Page3-16「例外定義」で、より詳しく述べる。

式の中に登場するシンボルは、通常変数参照として評価されるが、シングルクォート(')を前に付けることで、この評価を抑えられる。クォーテッドシンボルは、シンボルそのものとして評価される。Page2-29「クォーテッド定数」を参照。

文字列

" [{ *stringChar* | *escSeq* }]* [*truncEscape*] "



文字定数は、ダブルクォーテーション(二重引用符)に挟まれた文字の集まりである。

stringChar 32-127 までのアスキー文字からなる。ただし、ダブルクォート(")と、バックスラッシュ(\)は除く

escSeq 特殊文字あるいは unicode の列から構成される。特殊文字は、`\"`, `\\`, `\n`, `\t` 等である。unicode のシーケンスは `\u` から始まり、いくつかの 16 進数が続き、`\u` で終わるものである。

truncEscape 短い unicode 指定列からなる。これは `\u` のあとに任意の数の 4 桁の 16 進数が続く。

簡単な文字列の例をあげる:

```
"pqr"    "Now is the time"""
```

文字列の中に、標準文字セットの中にはない、Unicode 文字を \u エスケープコードを挿入し、unicode 16 進数モードを開始することによって含むことが出来る。\u の後には任意の数の 4 桁で特殊文字を指定するグループを続けることが出来る。また、もう一つの \u を追加することで、unicode 16 進モードを終わらせ、通常文字セット状態に戻すことが出来るが、特に 16 進モードを終了させる必要はない。(特殊文字と、Unicode の対応については、Appendix B「特殊文字コード」を参照のこと。)

文字列中で指定できる特殊文字は、表 2-2 に要約する。

また、文字列に対して、配列アクセッサ構文を使うことが出来る。より詳しくは Page2-17「配列アクセッサ」参照のこと。

たとえば、文字列を次のように指定する:

```
aString := "ABCDE";
```

そして、配列アクセッサで文字 B を参照したいとすると ...

```
aLetter := aString[1];
```

のような書き方をすればよい。

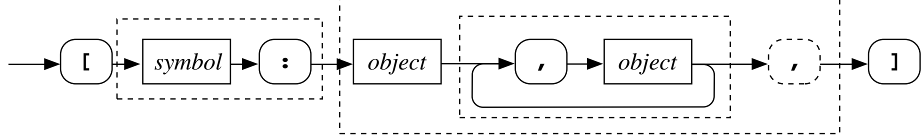
aString へのインデックスが 1 である事に注意。配列要素は添え字 0 から始まる。

表 2-2 文字列内での特殊文字

文字コード	意味
\n	改行
\t	タブ
\u	Unicode モードのスイッチ
\\	バックスラッシュ
\"	ダブルクォート

配列

‘[’ [*symbol* :] [*object* [, *object*]* [,]] ‘]’



配列は、角括弧の中に、0 個以上のオブジェクトをカンマで区切って集めたものである。配列の最初に、シンボルを書いてそのすぐ後にコロン(:)を書くと、配列にクラスを与えることが出来る。

symbol 構文上シンボルとなる識別子で構成される。もしこれが存在すれば、配列にユーザー定義クラスをセットすることができる。

object NewtonScript のどのオブジェクトで構成しても良い。NewtonScript は、配列要素を 0 から番号付ける。オブジェクトが一つ以上ある場合は、コンマで区切る。

注意

‘[’ *symbol* : *symbol* (... は曖昧である: 最初のシンボルは配列のクラスまたは、メッセージのレシーバにも見えるが、NewtonScript では最初の方の解釈を取る。(メッセージ送信は、Chapter4「関数とメソッド」で説明される。)◆

意味的には、配列はオブジェクトの順序付き集まりで、添え字は 0 から始まる。フレーム同様、配列もメソッド・フレーム・配列を含むどんなオブジェクトでも格納できる。他の参照オブジェクトのように、配列はユーザー指定クラスを持つこともできるし、サイズを動的に変更することもできる。

リテラル配列の簡単な例をあげる:

```
[1, 2, 3]
```

最初の配列要素の前に、適当なクラスを指定する識別子を置き、セミコロンを続けることで、次のように、クラス名を配列に適用できる:

```
[RandomData: [1, 2, 3], 0, "Lastelement" ]
```

RandomData クラスを持つこの配列は、オブジェクトのミックスとなっている。それはもう一つの配列を最初の要素とし、整数 0 を二番目に、そして文字列を三番目の要素としている。

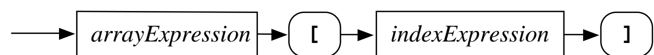
次のように、最後の要素の後に、余分なカンマを付けても良い。

```
[RandomData: [1, 2, 3], 0, "LastElement", ]
```

カンマの存在がプログラムに影響を与えることはないが、こうすることで、ソースコードを編集するときに、配列要素の移動がずっと簡単になるはずである。

配列アクセッサ

arrayExpression '[' *indexExpression* ']'



配列要素は、配列として評価される式と、角括弧の中に入れられた整数として評価されるインデックス式でアクセスできる。

arrayExpression 配列として評価される式 .

indexEpression 整数として評価される式。*indexExpression* はアクセスしたい配列要素に対応している。配列の添え字は 0 から始まることに注意。

たとえば、配列 myArray を次のように宣言したとして、

```
myArray := [123, [4, 5, 6], "Alice's Restaurant"];
```

二番目の要素に、この式を用いてアクセスするには、次のようにする。

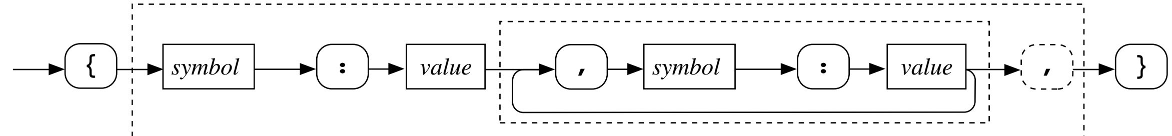
```
myArray[1];
```

この式の評価結果は`[4, 5, 6]`となる。

配列要素に、パス式を使ってアクセスすることもできる。それについてはpage 2-22「パス式」セクションを読むこと。配列アクセッサは実は演算子であり、ここに含まれているのは読者の便宜を図るためである。NewtonScriptの他の演算子については、Page2-30「演算子」を参照。

フレーム

'{' [*symbol* : *value* [, *symbol* : *value*]* [,]] '}'



フレームはにょろ括弧に囲まれ、0個以上の**スロット**をカンマで区切って配置したものである。スロットは、シンボルと、その後につづくコロン、そして、スロット式で構成される。シンボルはスロット式の値を参照する。

symbol シンボルはスロットの名前を与える。アンダースコアで始まるスロットはシステム予約なので、スロットの名前をアンダースコアで始めてはいけない。

value 他のフレームやメソッドなどのどんなオブジェクトでも良い。

フレームは名前と値のペアで構成されるスロットの集合である。配列と同様、スロットの値はメソッドやフレーム、配列を含むどんなオブジェクトであっても良い。

フレームはPascalとかCの構造体のような、データの貯蔵庫であるが、メッセージを受信できるオブジェクトとしても使用できる。よって、フレームはNewtonScriptにおける基本のプログラミングユニットである。

名前、電話を含む簡単なレコードに似たフレームは次のように書ける。

```
{name:"Joe Bob", phone:"4-5678", employee:12345}
```

次に、最初の3つのスロットに整数を含み、4番目のスロットにメソッドを持つフレームを例示する:

```
Jupiter := {
       size:491,
       distance:8110,
       speed: 34,
       position: func(foo) speed*foo/3.1416
}
```

オプションで、スロットclassを使うことによって、フレームのクラス名を指定することもできる。クラススロット

```
class : 'planet
```

をJupiterフレームに追加すれば、該当するクラス名が与えられる。配列クラスのように、フレームのクラスは特殊なプロパティではなく型を与える。しかし、クラスplanetの特殊な性質とか機能を表す全てのオブジェクトを与えたいなら、NewtonScriptの継承構造を使い、他のオブジェクトのデータを継承することを可能にする関係をセットアップできる。(Chapter 5 「継承と探索」参照)

他のフレームとの関係を指定するには、それらを_proto, _parentという名前のスロットを通して参照すればよい。それらのスロットが確立する関係により、オブジェクト指向アプリケーションを構築するための二重の継承機構というNewtonScriptの利点を得ることが出来る。Chapter 5 「継承と探索」では、そうしたコンセプトの説明を行っている。

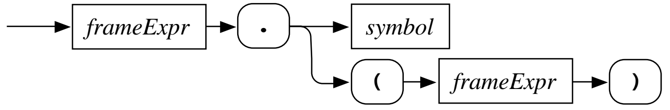
システムによって理解され、特殊用途に使われるスロット名がいくつかある。そうしたオプショナルなスロット全てを表 2-3 に示す。

表 2-3 特殊なスロット名と意味

スロットの名前	意味合い
`class`: *identifier*	特別なスロット名 `class` により意味的な型をフレームに付けられる。ユーザーオブジェクトのクラスにはシンボルを指定しないといけない。
`_parent`: *frame*	特別なスロット名 `_parent` を使い他のフレームをこのフレームの親として指定できる。ペアレント継承チェーンを構築するため、この手順を繰り返し他のフレームに使える。
`_proto`: *frame*	特別なスロット名 `_proto` を使い他のフレームをこのフレームのプロトタイプとして指定できる。プロトタイプ継承チェーンを構築するため、この手順を繰り返し他のフレームに使える。

フレームアクセッサ

frameExpr . { *symbol* | (*pathExpr*) }



フレームの値は、フレームとして評価される式と、シンボルまたは(パス式として評価される)括弧の中に入った式により、アクセスできる。フレームアクセッサ式は、指定したスロットの内容を返すが、スロットが無ければ `nil` を返す。

frameExpr 評価結果がフレームとなる式

symbol スロットを参照するシンボル。シンボルの構文について

は、Page2-13 「シンボル」参照。

pathExpr パス式となる式。*pathExpr* は、アクセスしたいフレームのスロットに対応する。配列の添え字は 0 から始まることに注意。

特定のフレーム内のスロットには、ドット(.)の後にシンボルを書いてアクセスする。たとえば次の式を見てみる:

```
myFrame.name;
```

この式の評価結果は、変数 `myFrame` で参照されるフレームから得られる `name` スロットの内容となる。

この構文を使って指定されたフレームにスロットが見つからない場合は、継承チェーンを通ってスロットの探索が継続される。NewtonScript が探す次の場所はプロトタイプフレームで、この探索はプロトタイプチェーンがつきるまで繰り返される。スロットが見つからなければ、検索は停止し、親フレームまでは検索が続けられない。

訳注: この記述は誤解を招くかもしれない。スロット参照において、どちらの継承が起こり、また起こらないかについては、後のセクションで詳しく説明されるので、この説明だけが全てだと思わない方がよい。

スロットが存在しなければ、式の評価結果は `nil` となる。

組み込み関数 `GetVariable()` と `GetSlot()` は、似たような種類のスロットアクセスを提供するが、継承の振る舞いは異なる。より詳しくは Chapter 6「組み込み関数」参照。

継承機構に関してより詳しくは Chapter 5 「継承と探索」参照。

フレームアクセッサは実際には演算子であるが、ここに書かれたのは読者の便宜のためである。残りの NewtonScript 演算子については、Page 2-30 「演算子」参照。

フレームのスロットには、パス式を使ってもアクセスできる。それについて

は、セクション「パス式」参照のこと。

パス式

パス式のオブジェクトは、複数のオブジェクトを経由したアクセスパスをカプセル化する。これらのオブジェクトは配列またはフレームを必要とする、なぜならNewtonScriptの中で、他のオブジェクトを含むことが出来るのは、配列とフレームだけだからである。

パス式は、次の3形態の内のどれかを取る

- 整数
- シンボル
- pathExpr クラスの配列

整数であるパス式は配列要素への参照を必要とする。フレームのスロットはシンボルでなくてはならないからである。以下のコードで配列要素をアクセスするための整数式の使用例を示す:

```
anArray := ["zero", "one", "two"];
aPathExpression := 1;
anArray.(aPathExpression);
"one"
```

同じく、シンボルパス式は次のコードに見られるように、フレームスロットへの参照を必要とする。

```
aFrame := {name: "Fred", height: 6.0, weight: 150};
aPathExpression := 'height;
aFrame.(aPathExpression);
6.0
```

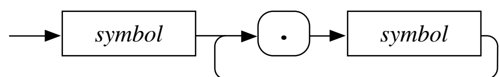
3番目のパス式はどんなオブジェクトも参照できる。それが配列、フレーム、その両方の中にネストしていてもである。以下のサンプルコードはパス式が

どのように、配列とフレームの中にあるオブジェクトへのアクセスパスをカプセル化できるかを示す:

```
myFrame:={name:"Matt",info:{grade:"B",tests:[87,96,73]}};
myPath:='[pathExpr: info,tests,1];
myFrame.(myPath);
96
```

パス式がシンボルしか含んでいなければ、次の構文が使える:

symbol [. *symbol*]+



symbol 任意のシンボル。

この構文は実際にはpathExprクラスの配列を作る。そしてこの構文で書かれたパス式はインスペクタ上ではpathExprクラスの配列としてプリントされる。

以下のコードは、この構文の使い方を示す:

```
myFrame := {kind:"Cat",type:{hair:"Long",color:"Black"}};
myPath := 'type.color;
myFrame.(myPath);
"Black"
```

スロットの値をセットする時にもパス式は使えることに注意。たとえば、上の例で、catのcolorを変更したければ、次の式を使う。

```
myFrame.(myPath):= "White";
```

式

簡単な式は、次のコードのように値と演算子からなる:

```
12 + 3;
```

一つの行に登場する値(12 と 3)とその間のプラス演算子は、評価されて戻り値 15 となる。変数はしばしば値をストアするための名前付きコンテナとして式で使用される。たとえば、代入演算子の左辺で代入のために変数を次のように使うことが出来る:

```
currentScore := 12;
```

変数 `currentScore` は、値の識別子となる。代入演算子(`:=`)についてより詳しくはこの章の「演算子」セクションを参照のこと。

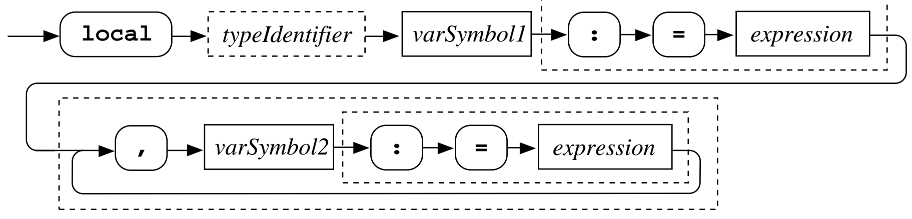
変数

変数はシンボルによって名前付けられる。変数はあらゆるオブジェクトの参照に使用できる。実行中のメソッドが変数参照に行き当たると、システムは変数の値を評価する。これは、NewtonScript の二重継承機構のルールに従う参照である。変数探索は Chapter 5 「継承と探索」で述べられている。

次のセクションでは、ローカル変数定義のための、キーワード `local` の使用方法について述べる。

ローカル

local [*typeIdentifier*] *varSymbol1* [:= *expression*] [, *varSymbol2* [:= *expression*]]*



ローカル宣言は予約語 `local` と、いくつかの初期化部分 — オプションの型指定子、シンボル、オプショナルの代入演算子(`:=`)に式が後続したもの — からなる。

varSymbol 構文的にシンボルとなる識別子からなる。シンボルは変数の名前となり、それはオプショナルな式で初期化できる。シンボルについてより詳しくはPage2-13「シンボル」参照。

typeIdentifier 予約語 `int` または `array`。ローカルな配列または整数をネイティブ関数の中で宣言する場合は、パフォーマンスを向上させることが出来るので、これは重要である。ネイティブ関数についてより詳しくはPage4-16「ネイティブ関数」参照。

expression 任意の式。ローカル変数が明示的に初期化されない場合、NewtonScript はそれを `nil` に初期化する。

`local` 予約語の使用はオプショナルである。もしそれが省略されても、同名の変数がなければ宣言され、初期化される。だが、この予約語は次の理由から省略すべきではない:

- パフォーマンス向上;システムはローカル変数宣言前にグローバルや継承構造を探索しないといけない
- 見つけるのが困難なバグを未然に防ぐ。もし同名のグローバル変数か継承スロットが存在すると、その値が変更されてしまい、新しい変数は宣言されない。グローバル変数や継承されたスロットがその後アクセスされると、予期しない結果が起こる。
- 明示的なローカル変数宣言はプログラムを読みやすくし、保守性を高める
- ネイティブコンパイラは宣言されてないローカル変数を取り扱えない

ローカル変数のスコープは、それが宣言された関数の中に制限される。

変数をローカルなものとするために、ローカル式を初期化なしで次のように使うこともできる:

```
myFunc: func (x)
        begin
                local myVar, counter;
                ...
        end
```

このサンプルは、変数 myVar と counter をローカル変数として宣言し、nil に初期化している。そして、myFunc の関数定義が実行される度、新しいローカル変数が生成され、その関数に対してユニーク(一意)なものになる。

オプションで、次の例のように、ローカル式を一つ以上の代入文と一緒に使うことが出来る。

```
local x:=3, y:=2+2, z;
```

この式は3つのローカル変数 x, y, z を作り、それぞれの値をそれぞれ 3, 4, nil に初期化する。

ローカル変数の宣言はコンパイル時に処理されるが、値の代入は実行時、その式が登場したときに行われる。

```
x := 10;
local x, y := 20;
```

これによって x の値は 10 となり y の値が 20 となる。ローカル定義は関数のどこにあっても働くのである。

しかし、次の例では実行時エラーが生じる:

```
x := y + 10;
local x, y := 20;
```

なぜなら、x, y はコンパイル時に宣言され、nil に初期化される。代入は実行時に行われるので、y は nil と評価され、nil+10 の計算でエラーが生じる。

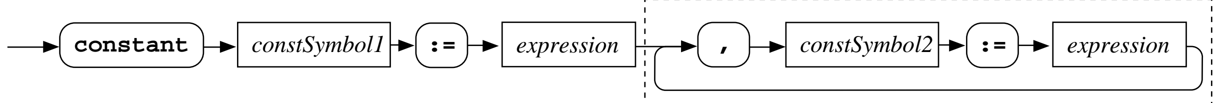
定数

NewtonScript で、変更されない値を作る方法は次の通り:

- 予約語 constant を使う
- オブジェクトリテラルの前にシングルクォートを置く
- 変数をリテラル値に初期化する

Constant 予約語

constant *constSymbol1* := *expression* [, *constSymbol2* := *expression*]*



定数宣言は予約語 constant と、一つ以上の初期化節(シンボルと代入演算子と式からなる)からなる

constSymbol 構文的にシンボルとなる識別子。詳しくは Page2-13「シンボル」参照。

expression 演算子と定数からなるあらゆる式。この式の値が *constSymbol* の値となる。

定数が値として使われる時、NewtonScript は効果的に定数をリテラル値に置き換える。このことは、次の式で定数を宣言したときに、

```
constant kConst := 32;
```

NewtonScript のコード中で kConst を値として使ったとき、それは自動的に 32 に置き換わることを意味する。もし次のように書けば、

```
sum := kConst + 10;
```

次のように書いたのと全く同じである、

```
sum := 32 + 10;
```

しかし、同じ識別子を次のように値としてではなく使った場合、

```
x:kConst(42);
kConst(42);
x.kConst;
```

定義した値は**置換されない**。これは、定数を、それ自身の値として関数やメソッドを持つように定義した場合に問題となる。こうしたケースでは、表2-4に見られるような組み込み関数を、置き換えのために使用する。

表 2-4 定数への置換

置き換えない	置き換えるやりかた
`x:kConst(42);`	`Perform(x,kConst,[42]);`
`kConst(42);`	`call kConst with (42);`
`x.kConst;`	`x.(kConst);`

また、定数の値は次のようなクォーテッド式内では置換されないことに注意:

```
'{foo: kConst};
'[kConst];
```

シングルクォート構文については、次のセクションを参照のこと。

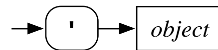
▲ 警告

ローカル定数を、関数定義の中で宣言することもできる。こうして作られた定数は、コンパイラ中ではローカル変数と同じ名前ス

ペースに入れられるので、同名のローカル変数は、定数を書き換えてしまうし、その逆もまた真である。▲

クォーテッド定数

'object



シングルクォート(')は、クォーテッド定数と呼ばれる種類の式を開始する。

クォートを使うと、リテラルオブジェクトを作ることができる。オブジェクトはコンパイル時に作られ、オブジェクトリテラルが評価される度に同じリテラルへの参照が起こる。

以下に、クォーテッド定数構文によって作られたリテラルオブジェクトのいくつかの例をあげる:

```
'{name: "Joe Bob", income: yearTotal};
'myFrame.someSlot;
'[foo, 1234, "a string"];
```

ブラケットやブレースの外側でクォートが現れたら、配列またはフレームの全要素にそれが適用されるので、配列またはフレーム内のシンボルに個々にクォートをする必要はない。たとえば、次のような、一見正しいフレームを作ろうとすれば:

```
storyFrame := {Bear1: Mama, Bear2: Papa, Bear3: Baby};
```

Mama が未定義の変数として翻訳される場合にはエラーが起こるだろう。これを避ける一つの方法は、各名前の前にクォートを置くか、より単純には次の式のようにフレーム全体の前にクォートを一つ置くかである。

```
storyFrame := '{Bear1: Mama, Bear2: Papa, Bear3:Baby};
```

このクォートされたフレームリテラルを、必要なときにオブジェクトを適用するためのパラメタとして渡すことができる。

演算子

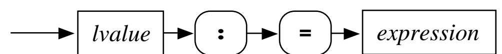
NewtonScript の演算子は次の通り:

- 代入
- 算術
- 論理
- 等価
- 関係
- 単項
- メッセージ送信
- 配列・フレームアクセッサ

メッセージ送信演算子と配列・フレームアクセッサを除く全ての演算子は、このセクションで説明される。メッセージ送信演算子については Chapter 4「関数とメソッド」で述べる。アクセッサに関しては、Page 2-20 「フレームアクセッサ」及び Page 2-17 「配列アクセッサ」で述べている。

代入演算子

lvalue := *expression*



代入式では、左辺値(シンボル、フレームアクセッサ、配列アクセッサ)に、代入演算子 := の右側に現れる式の値が代入される。代入式の評価結果は代入演算子の右側の式の値となる。

lvalue シンボル、配列要素への参照、フレームスロットへの参照からなる。左辺値ともいう。

expression 任意の式。

変数とスロットの値を変更するため、代入演算子を使うことができる。簡単な代入式は、次のようなものである:

```
a := 10;
```

代入式の右側には、NewtonScript のどんな式でも使うことができ、それが評価されるときに左辺値への代入が起こる。例をあげると、次の式では、変数 `x` には、`if` 式の戻り値がセットされる:

```
x := if a > b then a else b;
```

次にフレームを変数に代入する例をあげる:

```
myFrame := {name: "", phone: "123-4567"}
```

こうすると、次のように `name` に値を代入できる:

```
myFrame.name := "Julia"
```

注意したいのは、フレームへの代入式の振る舞いに NewtonScript の継承ルールが最終的に影響するという事である。継承と、スロットへの値の代入についてより詳しくは、Chapter 5 「スロットの値を設定するための継承ルール」を参照のこと。

同じやり方で、配列スロットへの代入が出来る。下のコードの 2 行目で、値を 789 から 987 に変更している:

```
myArray := [123, 456, 789, "a string"];
myArray [2] := 987;
```

変数への参照オブジェクトの代入は、単に変数にオブジェクトへのポインタのコピーが渡されるだけである。(Page 2-6 「イミディエイトと参照値」参照)

組み込み関数 `Clone`, `DeepClone`, `TotalClone` によって、この動作を変えることが出来る。これらの関数の動作については Chapter 6 「組み込み関数」参照。

代入式 *lvalue* := *expression* の評価結果は、*expression* の値である。さらに、代

入演算子は、右から左に結合されるので、式を次のように書いて良い:

```
aVariable := anotherVariable := anExpression;
```

これは次のように解釈される:

```
aVariable := (anotherVariable := anExpression);
```

(anotherVariable := anExpression) の部分は anExpression として評価され、aVariable と anotherVariable の値はどちらも anExpression の値となる。

注意

代入式の中で、代入演算子(:=)の代わりに、等価演算子(=)を間違って書いてしまうと、式はただの比較式になってしまう。たとえば、x=5 という式の評価は true または nil となってしまい、x の値は変わらないままとなる ◆

算術演算子

```
{ + / - / * / / / div / mod / << / >> }
```

NewtonScript では、標準の 2 項算術演算子を用意している。それは、加減乗除(+-*/)で、剰余計算には div と mod があり、左右ビットシフト(<<, >>)演算子も用意されている。

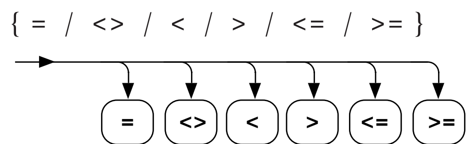
演算子	説明
+-*/	見ての通り、両側の数値を加減乗除する。
div	左の数字を右の数字で整数除算したときの商を返す。
mod	左の数字を右の数字で整数除算したときの余りを返す。
<<	左の数字を右の数字で指定したビット数だけ左にシフトする。1<<2 は 4 となる。

`>>` 左の数字を右の数字で指定したビット数だけ右にシフトする。`8>>1` は `4` となる。

演算子の優先順位については、Page 2-38 の表 2-5 を参照のこと。

等価、関係演算子

{ = / <> / < / > / <= / >= }



NewtonScript は、標準の等価・関係演算子を提供する。それは等価(=)、不等(<>)、より小さい(<)、より大きい(>)、以下(<=)、以上(>=)である。

`=` イミディエイトと実数の等価テストをし、両方とも等しければ `true`, さもなくば `nil` を返す。

`<>` イミディエイトと実数の等価テストをし、両方とも等しければ `nil`, さもなくば `true` を返す。

`<` 数値、文字、文字列のテストを行い、左の方が右より小さければ `true` さもなくば `nil` を返す。フレームや配列を比較するとエラーになる。

`>` 数値、文字、文字列のテストを行い、左の方が右より大きければ `true` さもなくば `nil` を返す。フレームや配列を比較するとエラーになる。

`<=` 数値、文字、文字列のテストを行い、左右が等しいか、左の方が右より小さければ `true` さもなくば `nil` を返す。フレームや配列を比較するとエラーになる。

`>=` 数値、文字、文字列のテストを行い、左右が等しいか、左の方が右より大きければ `true` さもなくば `nil` を返す。フレームや配列を比較するとエラーになる。

次の式は、二つの参照値を比較している:

```
"abc"<> [1, 2];
```

これは、二つのオブジェクトが違うものなので、true となる。

同じように、等価演算子を二つの配列オブジェクトに適用してみる。次のコードを実行してみると ...

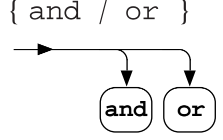
```
[1,2] = [1,2];
```

両辺とも等しく見えるにも関わらず、結果は nil となる。なぜなら [1,2] の式は常に新しいオブジェクトとして生成されるためである。

関係演算子は、数字、文字、文字列に対して働く。これらを配列やフレームと一緒に使おうとすると、エラーが発生する。

論理演算子(ブール演算子)

{ and / or }



論理演算子 and, or は、二つの式を比較する、論理的演算子である。

and	and 演算子は、両側の二つの論理的な値をテストし、両側とも true なら true を返し、さもなくば nil を返す。
or	or 演算子は、両側の二つの論理的な値をテストし、どちらかが true かどっちも true なら true を返し、両方とも nil なら nil を返す。

論理演算子を含む式は C における && や || とよく似ていて、式の真理値が決定したところで評価を打ち切る。たとえば、

```
x < length(someArray)
```

が次の式の中で nil に評価されるとすると、

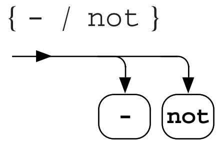
```
if x < length(someArray) and someArray[x]
then doSomething
else doSomethingElse;
```

評価は直ちに停止し、else 節の方に制御が移行する。and 操作のいずれかのパートが true でなければ、式全体が not true となる。よって、最初の半分が true でない時は、後の半分を評価する必要がないのだ。この事象を評価のショートカットという。同じく or 操作の最初の半分が true なら式全体も true となるので、2 番目のパートは評価されない。

論理演算子を使う式の戻り値は、nil(false を意味する)か、それ以外(true を意味する)のどちらかである。

単項演算子

{ - / not }



単項前置演算子マイナス(-)と not は、一つの式の前に置かれる。

- これが先行している式の値を正負反転して返す。

not 式の前の not 演算子は、その式の論理値の否定を返す。

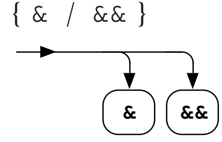
これらの式を使った例を示す:

```
-x;      not x;   -(1 + 5);
not(a and -f(12) > 3);
```

Exists は、もう一つの NewtonScript における単項演算子(ただし後置形式)である。それは Page2-36 「Exists」セクションで述べられる。

文字列演算子

{ & / && }



二つの文字列演算子 && と & は、二つの文字列から新しい文字列を生成する。& の方は文字列の間にスペースを挟まないが、&& の方はスペースを挟んで文字列を生成する。

文字列でないオブジェクトを && や & に適用した場合、オブジェクトを文字列に変換し新しい文字列生成に使う。これは、シンボル、文字、数値に適用される。

& と && 式は、新しい文字列を返す。

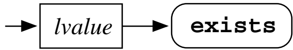
&	& 式は、左右の式から生ずる文字列を連結して返す。
&&	&& 式は、左右の式から生ずる文字列をスペースを挟んで連結して返す。

例をあげると、

"Lucky" & 7; と、"Lucky"&& 7; はそれぞれ、"Lucky7" と "Lucky 7" という新しい文字列を返す。

Exists

lValue exists



exists 演算子は、単一の値(それはシンボルや、フレームアクセッサやメッセージ送信であってよい)の後に付く特別な後置演算子である。

exists 演算子は変数、スロット、メソッドの存在をチェックするために使

用する。変数のような識別子に `exists` 演算子を適用すると、変数が現在のコンテキスト中で定義されていれば `true` を返し、そうでなければ `nil` を返す。(変数のスコープについてより詳しくは、Page 1-5「スコープ」参照)

lValue シンボル、フレームアクセッサ、メッセージ転送に評価される式から構成される。

以下は、全て `exists` 式の正しい例である。

```
x exists;
x.y exists;
x.(y) exists;
x:m exists;
```

`if...then` 構造の `exists` を使った例をあげる:

```
if myVar exists then myVar else 0;
```

フレームアクセッサに `exists` を適用すれば、`exists` はフレームまたはそのプロトタイプのどれかにそのスロットが有れば `true` を返し、さもなくば `nil` を返す。(プロトタイプフレームと、継承については Chapter 5「継承と探索」を参照。)`If...then` 式については、Chapter 3「制御構文」を参照。

この演算子はプロトフレームにスロットが存在するかどうかをチェックするのに便利である。

```
if myFrame.aSlot exists then
       if not hasSlot(myFrame, 'aSlot) then
              print("'aSlot slot is in a prototype of myFrame")
```

組み込み関数 `HasSlot()` も `exists` 演算子と似たような機構を提供する。しかし、それはスロット検索に使用する継承ルールが異なる。

Chapter 5 の「スロットの存在を確認するための継承ルール」及び、Chapter 6「組み込み関数」も参照のこと。

注意

`exists` 演算子のローカル変数への適用結果は保証されない ◆

演算子の優先順位

表 2-5 には、NewtonScript の全演算子の優先順位と、高いものから低いものへ、上から下に並べて示す。一緒のグループに入っている演算子は、等価な優先順位を持つことに注意。

表 2-5 演算子の優先順位と、結合規則

演算子	意味	結合方向
`.`	スロットアクセス	→
`: :?`	(条件付)メッセージ送信	→
`[]`	配列要素	→
`-`	単項マイナス	→
`<< >>`	左シフト 右シフト	→
`* / div mod`	乗算、除算、整数除算、余り	→
`+ -`	加算、減算	→
`& &&`	文字列合成、文字列スペース入り合成	→
`exists`	変数・スロットの存在確認	なし
`< <= > >= = <>`	比較	→
`not`	論理否定	→
`and or`	論理 AND, 論理 OR	→
`:=`	代入	←

CHAPTER 3

制御構文

この章では、NewtonScript の制御構造について述べる。制御構造は、`if then else`, `while`, `for`, `foreach`, `loop`, `repeat until`, `begin…end` による複式などがある。

また、NewtonScript が例外的な状況やエラーを制御するために使う、非標準の制御機構である例外処理についても述べる。

複式

```
begin
    expression1;
    expression2;
    ...
    expressionN[;]
end
```

NewtonScript では、予約語 begin と end は式のグループ化に使用されるが、ある種の言語に見られるような、変数のスコープを定義する構造ブロックを生成するためのものではない。

この構造は、条件式、ループ本体、関数定義等では有用なものである。複式は、単一の式が置けるところならどこに置いても良い。式の戻り値は、一番最後に実行された式の値となる。

expression NewtonScript の任意の式でよいが、式と式の間はセミコロンで区切らなければならない。ただし、最後の式の後にセミコロンを置く必要はない。後続する end が、区切りの代わりとなるからである。

訳注: 訳者としては、妙なバグを避けるためにも、区切るべき所は区切るように推奨したい。

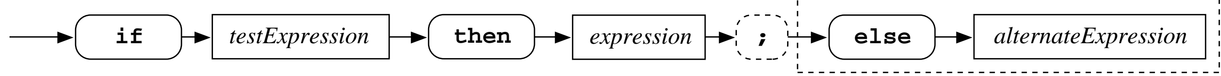
条件付き構文の中で、一つ以上の式を実行したい場合は複数の式をグルーピングするため、予約語 begin と end をつかう。

使用例を一つお目に掛ける:

```
if x=length(myArray) then begin
        result := self:read(x);
        print(result)
end
```

If...Then...Else

if *testExpression* then *expression* [;] [else *alternateExpression*]



if…then…else 構文は、条件判定を用いた、プログラマティックな流れの制御を可能にする。

他のプログラミング言語同様、if 式を使って、テストの条件が満たされれば

ある処理を行い、テストの結果が`nil`になればまた別の処理を行える。

`if` 式の戻り値は、*expression* 又は *alternateExpression* の戻り値である。テスト条件が`true`にならず、`else`節が存在しない場合は、`nil`が戻り値となる。

testExpression 条件の真偽をテストする式で構成される。テスト式の結果が`nil`以外のものであれば後続する *expression* が実行される。

expression NewtonScript の任意の式。複式を使ってもよい。(詳細はセクション「複式」参照。)

この式はテストの結果が`true`の場合実行される。この式の戻り値が`if`式の戻り値となる。

alternateExpression NewtonScirpt の任意の式。複式を使ってもよい。(詳細はセクション「複式」参照。)

`else`節に従属する *alternateExpression* は、テスト式の結果が`nil`の場合実行される。

NewtonScript に置ける`if then else`構文は式として取り扱われる。だから、

```
maxone := if x > y then x else y;
```

のような書き方が出来る。

`else`節は、もっとも近い位置にある、`else`節を持たない`if-then`節に結合される。

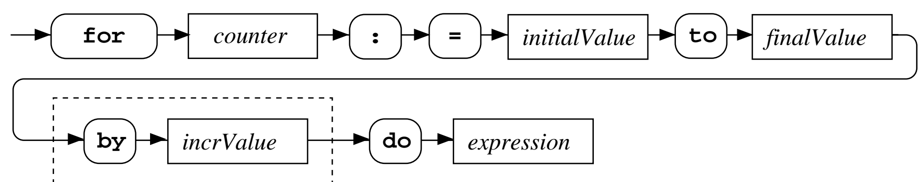
繰り返し

繰り返し構文は次の通り:

- For
- Foreach
- Loop
- While
- Repeat
- Break

For

`for` *counter* `:=` *initialValue* `to` *finalValue* [`by` *incrValue*] `do` *expression*



`for` ループは一連の式を、ループカウンタ変数の値が最終値に到達するまで、あるいは `break` 式に到達するまで繰り返される。カウンタ変数はループが実行された回数を保持する。カウンタ変数の最初の値と、最後の値を指定してこの構文を使う。オプションの予約語 `by` の後にカウンタの増(減)分値を指定できるが、これを指定しない場合は、デフォルトでカウンタは1づつカウントアップされる。

`for` ループの戻り値は `nil` もしくは、(`break` がループを終了させた場合は) `break` 式の戻り値となる。

counter 　このシンボルは `for` ループ開始時に *initialValue* の値にセットされる。ループが実行される度に *counter* 変数は *incrValue* の値だけ増加される。*incrValue* を指定していな

い場合は、デフォルトの増分値は1である。

counter シンボルは自動的にローカル変数としてコンパイラによって定義される。

ループ終了時の *counter* の値は未定義である。この値をループ本体で変更するのは間違いである。それを行ったときの動作は未定義である。

initialValue この式の結果は整数でなければならない。これはループ開始前に一度だけ評価され、カウンタの初期値として使用される。

finalValue この式の結果は整数でなければならない。これはループ開始前に一度だけ評価され、カウンタの最終値として使用される。

incrValue この式は予約語 `by` の後に続ける。これはループ開始前に一度だけ評価され、ループ実行中にカウンタ変数を増加(減少)させるために使用される。

増分値式の結果は整数でなければならない。(正でも負でもよい)値0は実行時エラーを生じる。予約語 `by` の後に増分値式を指定しない場合にはカウンタのデフォルト増分値は1である。

expression NewtonScript の任意の式。複式を使ってもよい。(詳細はセクション「複式」参照。)

次の書き方をすると、`print(j)` は 7 回繰り返され、出力結果は以下のようになる。

```
for j := 0 to 6 do print(j);
0
1
2
3
4
5
6
```

次の書き方なら、`print(j)` が 2 回繰り返され、出力結果は以下のようになる。

```
for j := 0 to 6 by 4 do print(j);
0
4
```

`by` *incrValue* を指定しない場合の、デフォルトのカウンタ増分は 1 である。*incrValue* は負の値であっても良い。

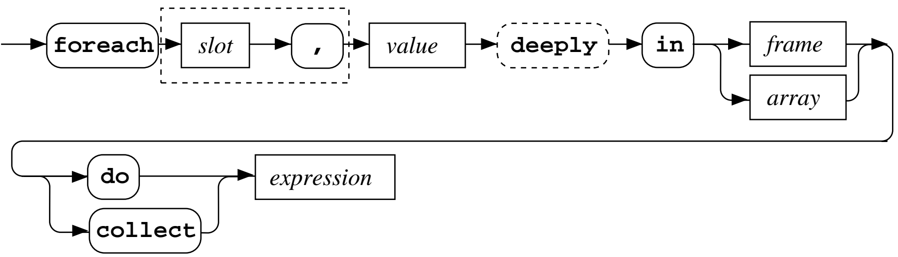
```
for j := 10 to -5 by -3 do print(j);
10
7
4
1
-2
-5
```

`break` を使って強制的にループを中断した場合は、`break` の戻り値が `for` ループの戻り値となる。(下の例では `nil` が返る)

```
for j := 0 to 10 do begin
        ...
        if x > 30 then break;
end;
```

Foreach

`foreach` [*slot* ,] *value* [`deeply`] `in` {*frame* | *array*} {`collect` | `do`} *expression*



`foreach` を使うと、配列やフレームの個々の値に対して式を適用したり、データを集めたりというような事が出来る。

対象が配列の場合、添え字 0 である第一要素から順に繰り返しが起こる。フレームの場合は、どのスロットからどの順に起こるかは予測できない。(訳者は、スロットの並びはソースコードで書いた順番だと思っていたが、実際その限りではないらしい)

この繰り返しは、フレームや配列内の要素に対する「浅い」ものとなるが、`deeply` オプションを付けることによって、繰り返しがターゲットフレーム内の値だけでなく、プロトチェーンのフレームにまで拡張される。ただ、`deeply` オプションを付けて処理しているフレームが `_proto` スロットを持っていなくても、別にエラーにはならない。プロトタイプ継承については Chapter 5 「継承と探索」を参照のこと。

`foreach` の戻り値は `nil` だが、ループ停止のために `break` を使うと、`break` の戻り値が `foreach` の戻り値となる。

slot このシンボルの値には、配列又はフレームの各要素に対し繰り返し処理を行うときの次のスロットの名前または添え字(インデックス)がセットされる。*slot* 変数の使用はオプションである。*slot, value* ペアに、一つしか変数を指定していない場合は、*value* が指定されたと仮定される。

slot シンボルはコンパイラによって自動的にローカル変数として宣言される。

value このシンボルの値には、配列又はフレームの各要素に対し繰り返し処理を行うときの次の配列要素の値またはフレームスロットの値がセットされる。ループ終了字のこの変数の値は未定義である。*value* 変数の使用は必須である。*slot, value* ペアに、一つしか変数を指定していない場合は、*value* が指定されたとみなされる。

value シンボルはコンパイラによって自動的にローカル変数として宣言される。

array 配列として評価される式。

frame フレームとして評価される式。

expression NewtonScript の任意の式。複式を使ってもよい。(詳細は Page3-1 のセクション「複式」参照。)

`deeply` オプション。この予約語が含まれていると、繰り返しはまず指定されたフレームに対して起こり、続いて `_proto` フレーム群に対して同様に行われる。(プロトタイプ継承については Chapter 5 「継承と探索」を参照のこと。) `deeply` オプションと一緒に指定したフレームが `_proto` フレームを持っていない場合でもエラーは生じない。その代わり、スロットの値は、現在のフレームのものとして評価される。

Pascal のレコードや C の構造体の様に機能するフレームのデータにアクセスするには `foreach` を使う。次の例で使用されるデータは `namedFrame` というフレームで、次のように定義される:

```
namedFrame := {
        name:"Carol",
        office:"San Diego",
        phone:"123-4567"
};
```

namedFrame内のスロットの名前とその値をプリントするメソッドは次のように書ける。

```
reporter:= {
reportEm: func(frameName)
              foreach slot, value in frameName do
                     print(slot && ":" && value);}
```

メソッドreportEmを引数namedFrameで呼び出してみよう:

```
reporter:reportEm(namedFrame);
```

結果は次のようになる:

```
"name : Carol"
"office : San Diego"
"phone : 123-4567"
```

次に、collectをforeachと一緒に使うと、データの収集が簡単になるという例を見て見よう。次のように定義されたnumbersFrameを考えて見る:

```
numbersFrame := [1,3,5,7,9]
```

foreach…collectを使い、numbersFrameの各値を二乗したものを配列にして、結果をプリントしてみよう。

```
result := foreach value in numbersFrame collect value*value;
print(result);
```

収集されたデータは次のようになる:

```
[1,9,25,49,81]
```

結果が配列に集められている(collectされている)ことに注意。

注意

foreachの動作は、ループ本体でフレームや配列が変更された場合には不定となる。ただし、現在の要素が削除された場合は例外である。この場合、期待通り次の要素にループは継続する。◆

最後に、deeply オプションの例を見てみる。まず次の _proto チェーンを持つフレーム群を構築してみよう。

```
x := {one:1, two:2, three:3};
y := {four:4, five:5, combo:x};
z := {six:6, _proto:y};
```

このデータは図 3-1 に図示する。同時に、表 3-1 には、foreach 単独と deeply オプションが付いたときの異なる結果を示す。

図 3-1　　データオブジェクトとそれらの関係

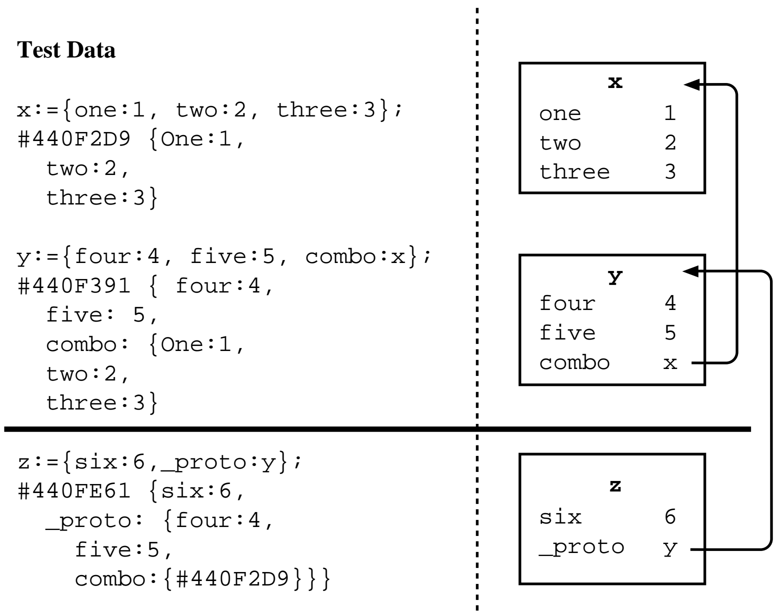


表 3-1 で、shallowList と deepLsit の二つの関数が図 3-1 のデータに適用された場合の結果の比較ができる。

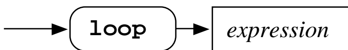
訳註:　表内のプログラムを試す前に、a := {} という空のフレームを作っておくとよい。

表 3-1 foreach と foreach deeply の結果比較

<table>
<tr><th>foreach</th><th>foreach deeply</th></tr>
<tr><td><pre>a.shallowlist := func (param) begin
     foreach tempItem in param
               collect tempItem;
     end;</pre></td><td><pre>a.deeplist := func (param) begin
     foreach tempItem deeply in param
               collect tempItem;
     end;</pre></td></tr>
<tr><td><pre>a:shallowlist(x)
#4413441 [1, 2, 3]</pre></td><td><pre>a:deeplist(x)
#44137D9 [1, 2, 3]</pre></td></tr>
<tr><td><pre>a:shallowlist(y)
#4413A11 [4,
           5,
           {One: 1,
            two: 2,
            three: 3}]</pre></td><td><pre>a:deeplist(y)
#4413C49 [4,
           5,
           {One: 1,
            two: 2,
            three: 3}]</pre></td></tr>
<tr><td><pre>a:shallowlist(z)
#4416E29 [6,
           {four: 4,
            five: 5,
            combo: {#4415D79}}]</pre></td><td><pre>a:deeplist(z)
#4416FE1 [6,
           4,
           5,
           {One: 1,
            two: 2,
            three: 3}]</pre></td></tr>
</table>

Loop

`loop` *expression*



`loop` は無限ループを形成する機構であるため、`break` によるループ停止は必須のものとなる。

よって、`loop` の戻り値は、`break` の戻り値となる。

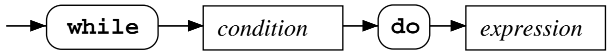
expression NewtonScript の任意の式。複式を使ってもよい。(詳細はPage3-1 のセクション「複式」参照。)式はループ繰り返しの間、毎回実行される。

次の例をお目に掛ける。これは、変数 x の値が 0 になるまで x の値をプリントし続ける。

```
local x:=4;
loop
       if x = 0 then
              break
       else
              begin
                     print(x);
                     x:=x-1
              end
4
3
2
1
```

While

while *condition* do *expression*



while ループはまず条件式を評価する。それが非 nil(true もしくは nil でないあらゆる値)の値を返すなら、予約語 do の後の式が実行される。この流れは条件式が nil に評価されループが終了するまで繰り返される。

while の戻り値は nil だが、ループ停止を break でやった場合は break の戻り値が while の戻り値となる。

condition 条件の真偽をテストするための式から構成される。テス

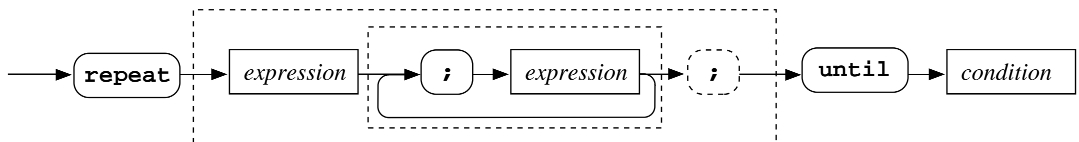
ト式が nil に評価されるとループ実行は終了する。

expression NewtonScript の任意の式。複式を使ってもよい。(詳細は Page3-1 のセクション「複式」参照。)式はループ繰り返しの間、毎回実行される。

訳註: 訳者は、条件判定に数値以外のものを使えるという点で while の方が好きだが、配列に単純な繰り返し操作を行うような場合は for の方が実行時の効率が良い様である。

Repeat

```
repeat
    expression1;
    expression2;
    ...
    expressionN[;]
until condition
```



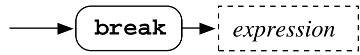
repeat ループは、ループの最初から、until *condition* による条件テストまでの間にある各式を実行する。while と違うのは、ループ内の式が必ず一回実行されることだ。ループの最後で *condition* が非 nil を返せば、ループは停止する。repeat の戻り値は非 nil だが、break を使っていたら、break の戻り値が repeat の戻り値となる。

condition 条件の真偽をテストするための式から構成される。テスト式が nil 以外に評価されるとループ実行は終了する。

expression NewtonScript の任意の式。

Break

`break` [*expression*];



`break`は、繰り返し構造の中断に用いる。`loop`(ループ停止条件を持たない)の場合、`break`の使用は必須である。`break`の後の式(*expression*)は、`break`の戻り値となり、それは同時にループの戻り値となることを意味する。
`break`の後に式を付けなければ、`nil`が返される。

expression NewtonScriptの任意の式。複式を使ってもよい。(詳細はPage3-1のセクション「複式」参照。)この式の結果が、`break`の戻り値として返される。

訳注: 個人的には、continue(ループを終了させず、whileやforのcondition部分にジャンプする)がないというのが非常に残念である。

例外処理

このセクションではNewtonScriptの例外処理について述べる。例外処理とは、NewtonScriptがNewtonシステムソフトウェアから受け継いだ、標準的ではないフロー制御機構のことである。

NewtonScriptの例外処理によって、プログラム実行中に発生する例外的な状況に対応することができる。例外的な状況あるいは**例外**とは、Newtonシステムソフトウェア又はユーザーのプログラムが、その実行中に何か予期しない事や、エラーに行き当たったときに発生させるものである。(変数が見つからないとか、数を0で除算したとか)

例外が実行中に持ち上がると、システムは制御を**例外ハンドラ**に渡す。例外ハンドラとは、状況をうまく取り繕い、アプリケーションのクラッシュを回避するための、システムあるいはユーザーが書いたコードブロックである。例外ハンドラはエラーメッセージを表示したり、処理を復帰させたりといったさまざまなアクションを起こす。

例外の発行は、例外の投げかけ(throw)として知られる。例外ハンドラは例外を捕獲し、特定の方法でそれに対応する。各例外は一意な名前を持っていて、各例外ハンドラがその例外クラスあるいは特定の例外に対して対応する。

Newton システムソフトウェアはいくつかの組み込み例外を投げ、捕獲する。プログラマは自分の例外を定義して投げたり捕獲したりできるし、組み込み例外を捕獲する事もできる。

例外の使用

例外の取り扱いにおいて、つぎのことができる。

- 特定の例外あるいは例外クラスに対する例外シンボルの定義
- `try` ステートメントによる、例外捕獲ブロックの実装
- `onexception` ステートメントによる特定の例外の捕獲
- `CurrentException()` 関数による、現在の例外内容の検証
- 例外検出時の例外発行(`throw`)
- 例外処理時の、他の例外処理ブロックへの再発行(`rethrow`)

例外処理時でも、通常の NewtonScript のコードを何でも使用できる。さらに、例外処理に構造を持たせ、複数のハンドラをネストさせることもできる。

各例外ハンドラはそれが処理する例外のシンボル(またはそのプレフィックス)を示すことで、処理する例外や例外クラスを指定できる。例外ハンドラはまた、例外を再発行(`rethrow`)することもでき、これによって一連の例外処理の中で、他の例外ハンドラに処理の機会を与える。

例外処理を使う基本的なやり方は、次の通りである:

1. 定義・処理しようとしている例外の名前を決める。
2. プログラムコードの適切な場所で、例外発行のため `throw()` 関数を使う。
3. 各例外に対応する `onexception` 節を記述する。各節には例外を指定し、例外を処理する文を書く。
4. 例外を発行し、処理するコードブロックを、`try` ステートメントで囲む。

例外の定義

各例外は、**例外シンボル**で名前付けされる。例外シンボルを定義するときには、次のフォーマットルールに従わなければならない。

各シンボルは:

- 縦棒(|)で囲む
- セミコロンで区切れば、複数のパートを含めることが出来る
- ただし、どれかのパートに、最初に `evt.ex` で始まる例外名を持ってないといけない
- 127 文字まで書ける

リスト 3-1 に、いくつかの例外シンボルの例をあげる。

リスト 3-1 例外シンボルの例

```
|evt.ex|
|evt.ex.fr.intrp|
|evt.ex.div0|
|evt.ex.msg;type.ref.frame|
```

重要

縦棒の中のものは、すべて(空白ですら)意味があると見なされるので、例外シンボルのどの部分にも空白を残してはならない。

例外処理シンボルに含まれるプレフィックスは、例外処理の階層を指定するために使われる。このことについては、page 3-21 「例外の捕獲」参照。例外のプレフィックスもまた、例外のタイプを定義するために重要なものである。これについては Page3-17「例外フレーム」参照。

例外シンボルパート

各例外シンボルは、複数のパートを縦棒の中に含むことができる。たとえば、シンボル

```
|evt.ex;type.ref.something|
```

は、二つのパートを含む。例外シンボルの各パートはセミコロンで区切らないといけない。

例外シンボルが複数のパートを含んでいても、それは一つの例外として扱われる。つまり、各パートのどれかを捕獲する最初の例外ハンドラが例外を処理するのだ。そのハンドラからは、rethrow を使ってさらに別のハンドラに例外を再発行出来る。

例外フレーム

例外には、**例外フレーム**という構造が付属している。例外フレームは二つのスロットを含むが、その内の name というスロットは常に存在していて、例外シンボルを格納している。もう一つのスロットの名前と内容は、例外シンボルの構造によって変わってくる。それについては表 3-2 を参照されたい。

表 3-2 例外フレームのデータスロット名と内容

例外シンボル	スロット名	スロットの内容
type.ref プレフィックスを持つパートを含む	data	データオブジェクト
ext.ex.msg プレフィックスを含むパートを含む	message	メッセージ文字列
その他	error	整数のエラーコード

表 3-3 には、例外と、そこに付属するフレームの例を示す。

表 3-3 例外フレームサンプル

例外シンボル	例外フレーム
`|evt.ex;type.ref|`	`{name: '|evt.ex;type.ref|, data: {type: 'inka, size: 42}}`
`|evt.ex.msg|`	`{name: '|evt.ex.msg|, message: "there seems to be a problem"}`
`|evt.ex|`	`{name: '|evt.ex|, error: -48666}`

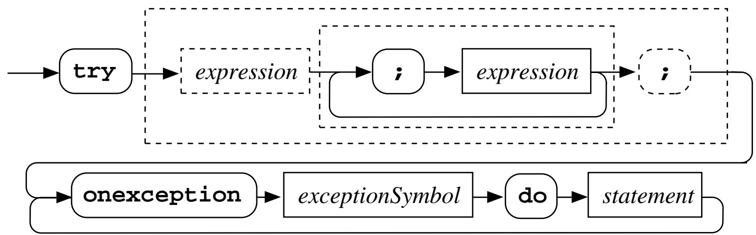
組み込み関数 `CurrentException` を呼ぶことで、現在の例外ハンドラ内から、例外に結び付けられているフレームにアクセスすることができる。この関数については Chapter6「組み込み関数」参照。

`CurrentException` 関数は、現在の例外に結び付けられたフレームを返すので、処理中の例外ハンドラでこの関数から返ってきたフレームを検証して、どのタイプの例外を処理しているか特定できる。たとえば、`HasSlot` 関数を呼び、`error` という名前のスロットがそのフレームにあるかどうか調べ、適切な処理をすることが出来る。

Try ステートメント

例外を処理したいと思うステートメントのリストを、try ステートメントで囲む。try ステートメントの構文は次の通り:

try
    expression1 ;
    expression2 ;
    ...
    expressionN
onexception *exceptionSymbol* do
    statement
onexception *exceptionSymbol* do
    statement...



try ステートメントは、*expression1* から *expressionN* を囲み、例外発生時は、onexception 節の一つに制御を移す。例外が発生しない場合は、try ステートメントの戻り値は、最後の式(この例では *expressionN*)の戻り値となる。例外発生時は、実行されたいずれかの onexception 節のステートメントの戻り値となる。

expression いかなる式でも良い

exceptionSymbol セミコロンで区切られた複数パートを持つ例外シンボル。このシンボルは縦棒で挟まれ、127 文字までを含む。

例外シンボルについては Page3-17 「例外シンボルパート」で説明される。try, onexception ステートメントの例は、Page3-23 のリスト 3-4 にある。

例外の発行(throw)

NewtonScriptで例外を発行するには、例外名とデータをパラメタにして関数`Throw`を呼ぶ必要がある。パラメタとして送るデータの形態は発行する例外に依存する。

`throw(`*name*, *data*`)`関数により、指定された*name*の例外を発生させることが出来る。*data*パラメタは、その例外に付属する例外フレームのデータとなるが、その内容はPage 3-17の表3-2の対応表に見られるとおり例外のシンボルによって変化する。`Throw`関数についてはChapter 6「組み込み関数」参照。

`try`ステートメント内の式から`throw()`を呼び出すと、*name*と一致する`onexception`節に制御が移る。リスト3-2にその使用例を示す。

リスト3-2　Throw()の使用例

```
Throw('|evt.ex.foo|, -12345); //error number required
Throw('|evt.ex.msg|, "This is my message"); //message string required
Throw('|evt.ex;type.ref.something|, ["a", "b", "c"]); //data object
required
```

`Throw`関数への最初のパラメタとして渡す例外シンボルは、二番目のパラメタとして渡すデータの種類を決定してしまうことに注意:

- リスト3-2の最初の文は、二番目のパラメタとしてエラーナンバーを必要とする
- 第二の文は、`evt.ex.msg`プレフィックスを持つので、二番目のパラメタは文字列を必要とする
- 第三の文は、`type.ref`プレフィックスを持つので、データオブジェクト(この例では配列)を二番目のパラメタとして必要とする。

他のハンドラへの例外再発行(rethrow)

例外ハンドラから、さらに別の例外ハンドラへ制御を移すには、throw() でなく、rethrow() を使う。この関数は Chapter6「組み込み関数」で説明される。

rethrow() 関数は、現在の例外を再発行し、別の try ステートメントにそれを処理する機会を与える。最初の throw() 呼び出しに渡したのと同じパラメタを受け渡していくので、rethrow() にはパラメタを指定しない。以下に rethrow の使用例を示す:

```
onexception |evt.ex.msg| do
        if StrEqual (CurrentException().message, someString)
                then self:doSomething();
        else Rethrow()
```

重要

Rethrow 関数は、onexception 節の中からだけ呼べる ◆

例外の捕獲

プログラム実行中に例外が発行されると、制御は、発行された例外とマッチする最初のイベントハンドラに直ちに移行する。各例外ハンドラは、リスト 3-3 及び 3-4 に見られるように、try ステートメントに囲まれた onexception 節である。

予約語 onexception の後には、処理対象となる例外あるいは例外クラスのシンボルを指定している。発行された例外にマッチしたシンボルを持っている最初の例外ハンドラが起動される。このメカニズムは次のようなものだ。

1. 例外が起こると、OS が現在アクティブな try ステートメントの onexception 節を調べていく。onexception 節は、定義された順に上から下へ調査される。
2. 最初にマッチした onexception 節が実行され、その戻り値が、try ス

テートメントの戻り値となる。マッチする onexception 節は、その例外シンボルが、実際に発行された例外シンボル内のどれかのパートのプレフィックスとなっている onexception 節である。

3. 現在の try ステートメントが、マッチする onexception 節を持たない場合、例外は次に外側の try ステートメントに渡される。
4. 例外は処理されるまで次々と try ステートメントを渡り歩く。アプリケーション内のハンドラに、全く onexception 節がない場合、例外はシステムによって処理され、エラーアラートが表示される。

例外ハンドラを使うに当たって考慮しないといけない二つのポイントがある。

ポイントその１：

例外シンボルに含まれるパートの中のプレフィックスを指定している最初の onexception 節によって例外が処理されるから、onexception 節を、詳しいものからおおざっぱなものへと並べていかないと意味がないということだ。リスト 3-3 の例を見てもらいたい。これは、onexception 節の間違った並べ方の例である。

リスト 3-3　　順番の正しくないいくつかの onexception 節

```
try
       c := x:myFunc(p, q);
       :anotherFunc(c)
onexception |evt.ex.pgm.fnerr| do
       begin
              print("function error");
              c := nil;
       end
onexception |evt.ex.pgm| do
              print("program error")
onexception |evt.ex.pgm.dataerr| do
```

```
            print("data error");
```

evt.ex.pgm をプレフィックスに持つ例外は、全部最初の onexception で実行されてしまうので、evt.ex.pgm.dataerr の onexception 節は決して実行されることがない。解決策は、例外シンボルの並びを詳しいものからおおざっぱなものに、リスト 3-4 のように並べ替えることだ。

リスト 3-4　　正しく並んだ onexception 節

```
try
    c := x:myFunc(p, q);
    :anotherFunc(c)
onexception |evt.ex.pgm.fnerr| do
    begin
        print("function error"); do
        c := nil;
    end
onexception |evt.ex.pgm.dataerr| do
        print("data error")
onexception |evt.ex.pgm| do
        print("program error");
```

ポイントその 2:

onexception 節は、もっとも近い try ステートメントにマッチするという事だ。else 節が最も近い if-then 節にマッチするのと似ているが、if-then との違いは、try ステートメントが複数の onexception 節と結合するということである。リスト 3-5 は、このことがネストされた try ブロックにおいてどのような問題を起こすかを示す。そして、リスト 3-6 には、予約語 begin と end を用いてこの問題を解消する例を示す。

リスト 3-5　　正しくない try ブロックのネスト

```
func f()
begin
	try
		try
			self:doSomething()
		onexception |evt.ex| do
			print( CurrentException() );
		self:doSomethingElse()
	onexception |evt.ex| do
		print( "There was a problem." );
end
```

リスト 3-6 begin と end(bold で示す)を使って、ネストした try ブロックの問題を解消した例

```
func f()
begin
	try
		try
		begin
			self:doSomething()
		onexception |evt.ex| do
			print( CurrentException() );
		end
		self:doSomethingElse()
	onexception |evt.ex| do
		print( "There was a problem." );
end
```

重要

onexception 構文は、よけいなセミコロンを許容しないので、onexception 節の前に、セミコロンを置かないこと。◆

訳註: リスト 3-6 は間違っている。どこが間違っているかは、読者自身で検証されたい。(第一、例としても適切でないように思う)

例外への対応

このセクションでは、アプリケーションプログラムにおける例外ハンドリングのいくつかの例を示して説明する。

リスト 3-7 では、Newton システムソフトウェアが DateBook スープに新しい日付のデータをストアするのに十分なメモリを持っていない場合に発行する例外を捕獲する例を示す。

リスト 3-7　　スープストア例外

```
onException |evt.ex.fr.store| do
       :Notify(kNotifyAlert,
"Dates", "Not enough memory to save changes.");
```

リスト 3-8 に示すのは、例外が特定のエラーを示すかどうか確定するため例外フレームを検証する例外ハンドラの例を示す。もしそうなら、ハンドラは何らかのアクションを起こすし、そうでない場合、ハンドラは例外を再発行するので、他のハンドラにより捕獲される。

リスト 3-8　　例外フレームをチェックする例外ハンドラ

```
onException |evt.ex| do
       if HasSlot(CurrentException(), 'error) then begin
              if CurrentException().error = -48211 then
                     Print("The string you entered is too
large")
              else Rethrow();
       end else Rethrow();
```

空のページ

CHAPTER 4

関数とメソッド

この章では、関数とメソッドの使い方と、次の項目について説明する:

- メソッドと関数の定義
- メッセージ
- パラメタ渡し
- 関数オブジェクト
- ネイティブ関数

関数とメソッドについて

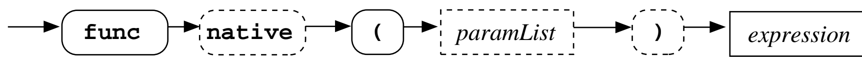
NewtonScriptで提供される多くの関数は実際には**メソッド**である。メソッドとは、フレームの中に定義されて、メッセージにより起動される関数のことだ。

他のオブジェクト指向言語と同じく、オブジェクトに**メッセージ**を送ってメソッド実行を実行することになるが、NewtonScriptにおいて、メッセージを受信できる唯一のオブジェクトはフレームである。(Page2-18の「フレーム」セクション参照)

また、NewtonScriptはシステムの一部でもある組み込みのグローバル関数を持っている。それらはChapter6 「組み込み関数」で述べられる。

関数コンストラクタ

func [native] (*paramList*) *expression*



関数コンストラクタの構文は、予約語 func とオプショナルな予約語 native に、カンマで区切った 0 個以上のパラメタリストを括弧で囲んだものと、単一の式からなる関数本体を後に続けたものである。

キーワード native は、その関数がネイティブ関数であることを示す。Page 4-16「ネイティブ関数」参照。

関数コンストラクタは、**関数オブジェクト**を返す。関数コンストラクタの戻り値は、*expression* の戻り値である。もし *expression* が複式なら、実行された最後の式がこれに該当する。

paramList 0 個以上のパラメタ識別子をカンマで区切って並べ、括弧で囲んだもの。パラメタがなくとも、予約語 func 又はオプションの native のあとに空の括弧を続ける必要がある。ネイティブ関数の場合、各パラメタの名前の前に、int あるいは array という型指定子を付け、自動的に個々の型として変数を宣言することができる。

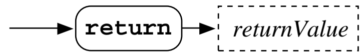
expression NewtonScript の式から構成される。複式でもよい。(Page 3-1 のセクション「複式」参照)

関数が実行されると、*expression* の結果が返される。次の例は、パラメタ p1 と p2 の差を返す関数 myFunc を定義した例でその戻り値は if の値である。

```
myFunc := func(p1, p2)
       if p1 > p2 then p1 - p2; else p2 - p1
```

Return

`return` [*returnValue*]



`return` は、関数から脱出し、戻り値を返す。

式が予約語 `return` のあとに出現すると、その式の評価結果が関数の戻り値として返される。*returnValue* を指定しない場合は、関数の戻り値は `nil` となる。

returnValue オプション。NewtonScirpt の式あるいは複式。この式の評価結果が `return` の戻り値となり、同時に関数の戻り値となる。*returnValue* を省略すると、`return` の戻り値は `nil` となる。

関数起動

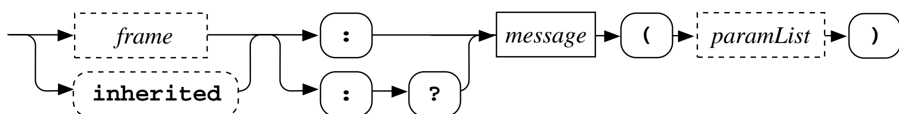
関数オブジェクトは次の３つの方法で実行できる。

- メッセージ送信
- call with 構文
- グローバル関数起動

このセクションではそれぞれの方法について、説明する。

メッセージ送信演算子

[{`inherited` | *frame*}] {`:`|`:?`} *message* (*paramList*)



コードの多くは、フレームにメッセージを送った結果として実行される。メッ

セージはメッセージ送信演算子 : または、条件付きメッセージ送信演算子 `:?` を使って送ることができる。両演算子の違いは、`:?` の方がメッセージ送信前に、フレームの継承チェーンの中にメソッドが存在するかどうかをチェックすることにある。

オプションのフレーム式 *frame* を演算子の前に指定すれば、メソッドを送るべきフレームを特定することもできる。指定されたフレームにメッセージはダイレクトに送られ、そのフレームが**レシーバ**となる。

*メッセージ送信*演算子の前に何も指定されていない場合、メッセージは疑似変数 **`self`** で参照できる現在のレシーバに送られる。メッセージの前に `self` を置いて、現在のレシーバへの送信であることを明確にする事もできる。(Page5-8 の「注意」には、`self` の利用に関する得失が述べられている)

予約語`inherited`を、メッセージ送信演算子の前に置くと、現在のレシーバーである`self`がオーバーライドしているメソッドではなく、継承されたメソッドを呼ぶことが出来る。これにより、現在のレシーバーはバイパスされ、プロトタイプチェーンの中のメソッドが探索され、呼び出される。`inherited` を指定した場合は、探索はプロトタイプチェーンの終わりと共に終了し、ペアレントチェーンに関しては行われないことに注意。より詳しくは、Chapter 5「継承と探索」参照。

メッセージ送信演算子のあとに続くメッセージはシンボルである。メッセージ送信演算子は、その名前を持つフレームスロットを探索する。フレームスロットは、関数を参照していなくてはならず、またその引き数の数は、メッセージ送信に渡した引き数の数と同じでなければならない。

frame　　フレームとして評価される NewtonScript の式。*frame* に指定されたものが、メッセージのレシーバとなる。以下の式のように、*frame*がコロンの前に現われない場合、メッセージは現在のレシーバに送信される。

```
:message(argList);
```

`inherited`　　プロトタイプチェーンのどこかにある継承されたバージョンのメソッドを呼び出すことを指定する予約語。予約語

inherited を使うことで、メソッドの探索はレシーバではなく、強制的にプロトタイプチェーンから始められる。

message 実行時に、レシーバを起点として、標準の継承ルールによってメソッドを探索するために使用されるシンボル。より詳しくは、Chapter 5「継承と探索」参照。

paramList 0 個以上のパラメタをカンマで区切って並べ、括弧で囲んだもの。パラメタの数は、メソッドが必要とする数と一致していなければならない。

次の呼び出しは、フレーム fram4 にメッセージ msg1 を送る。

```
frame4:msg1();
```

次の呼び方だと、self に msg1 を送る。

```
:msg1();
```

だから次のように書いても同じである。

```
self:msg1();
```

メソッドが存在しているかどうかわからない場合は、:?を使ってメッセージを送る。この場合は、メソッドが存在するときだけ実際にメッセージが送られる。次の 2 つの式の意味は同じである。

```
if frameName:messageName exists
        then frameName:messageName();
frameName:?messageName();
```

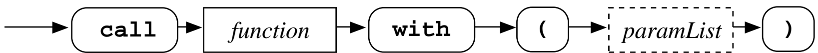
最初の書き方は、exists による探索と、メッセージ送信による探索の合計 2 回の探索が起こるので、後の書き方の方がCPUリソースの無駄使いにならなくて良い。

実行時に組み立てた引数でメッセージを送信するために、Perform() 関数がある、また、PerformIfDefined() という :? に似た関数もある。

ProtoPerform()と、ProtoPerformIfDefined()という、その名前から想像されるように、プロトタイプ継承チェーンだけを探す関数もある。これらはChapter 6「組み込み関数」で説明される。

Call With 構文

call *function* with (*paramList*)



call with 構文によって、関数オブジェクト(後で述べる)を、それが生成された時に取り込まれた環境値を使って、実行できる。lispで言うところのクロージャ的な使い方である。より詳しくはPage 4-9 の「関数オブジェクト」参照。

組み込み関数Apply()を使えば、実行時に組み立てた引数リストで関数を呼び出せる。

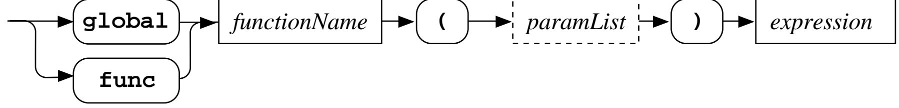
function 評価結果が関数となる式を書く。

paramList 0個以上のパラメタをカンマで区切って括弧で囲む。パラメタの個数は、関数が必要とするパラメタの個数と同じでなければならない。

call with の戻り値は、実行された関数の戻り値である。

グローバル関数宣言

{ global | func } *functionName* (*paramList*) *expression*



ある種のNewtonScript 実装において、グローバル関数を定義するための構文を使うことができる。予約語globalを除いては通常の関数定義構文と同じ構文であるが、この構文でグローバル関数を定義できる。(通常の関数定義文につい

ては Page4-2「関数コンストラクタ」参照)

グローバル関数は、どこかの関数の中でなく、トップレベルでだけ定義でき、そのスコープは、NewtonScript の関数全部である。

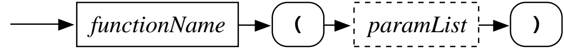
注意

NTK 環境でプログラミングしているときは、予約語`global`は、NTK の NewtonScript 環境の中だけでグローバルな関数を作る。もし、Newton デバイス上の、NewtonScript環境で動いている他の関数からこの関数を呼び出そうとすると、“unknown global function” エラーが生成される。◆

訳注: 本当にグローバルな関数を作る為の関数も用意されている。

グローバル関数起動

functionName （*paramList*）



グローバル関数起動は、`call with`と同様の効果を持つ。それは*functionName*で指定された関数オブジェクトとそのパラメタを関数オブジェクトが生成されたときに捕獲されたメッセージ環境を使って実行する。ただし、関数は式の評価結果としてではなく名前によって特定される。

グローバル関数呼び出しは、実行した関数オブジェクトの戻り値を返す。

functionName グローバル関数の名前シンボル。

paramList 0 個以上のパラメタをカンマで区切って括弧で囲む。このパラメタの個数は、関数が必要とするパラメタの個数と同じでなければならない。

NewtonScirpt には、たくさんのグローバル関数定義が組み込まれている。(Chapter 6 「組み込み関数」にそのリストを示す)

パラメタ渡し

NewtonScriptでは、パラメタは値渡し(call by value)で渡される。参照渡し(call by reference)構文はない。通常、他の言語では、値渡しで渡したパラメタを、相手の関数内で変更しても自分側は影響を受けないことが多いが、NewtonScriptにおいてはこれは当てはまらない。

NewtonScriptにおけるデータ型のある種のものは参照型であるため、これをパラメタとして渡すと、参照によって呼び出し元のデータも変更されてしまうという事に注意すること。

NewtonScript のイミディエイトと参照値についてより詳しくは、Page 2-5 のセクション「イミディエイトと参照値」を読むこと。

関数オブジェクト

関数オブジェクトの構築は、以下の構文で行う:

`func(`*paramList*`)` *expression*;

関数はNewtonScript に置ける最高のオブジェクトである。それは、ローカル変数・配列要素・フレームスロットに代入できる。それはまたスープにも格納でき、関数への引数としても渡すことが出来る。スープに関しては、*The Newton Progarammer's Guide* を参照のこと。

単に「関数」と呼ばず、「関数オブジェクト」という呼び方をしているのは、同一の`func`ステートメントから、たくさんの異なる関数オブジェクトが作り出せると言うことを強調するためである。関数オブジェクトが実行時に生成されるとき、その時点での環境を保存する。だから、ただ一つの関数コンストラクタから作られる関数オブジェクトであっても、その時の環境によっては全く違うものとなりうるのだ。

関数オブジェクトは主要な二つのパートからなる。一つはコード、もう一つは関数コンテキストである。関数コンテキストとは、関数オブジェクトが生成さ

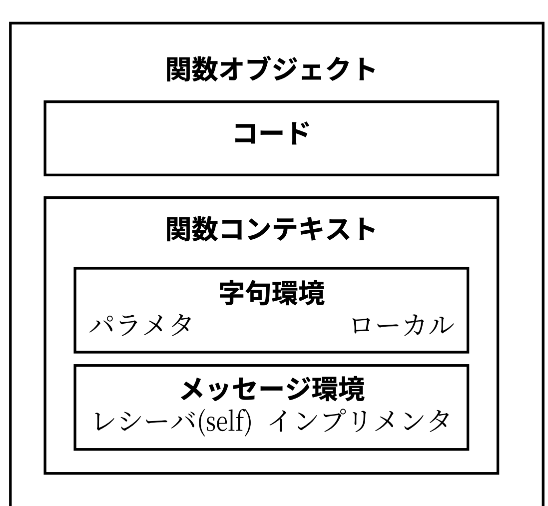
れたときに存在した環境が保存されたものであり、字句環境とメッセージ環境の二つからなる。

- 字句環境は、その関数内のローカル変数とパラメタのリスト及び、その関数内に含まれるあらゆる関数のローカル変数とパラメタのリストである
- メッセージ環境は関数オブジェクトが生成されたフレーム(**インプリメンタ**)への参照、それが送られるフレーム(レシーバ)への参照

環境の保存によって、それが生成されたときに存在したローカル変数やパラメタリストにアクセスすることが出来る。さらに、それが生成されたときの`self`の値により、フレームの継承チェーンの中にある変数にアクセス出来る。NewtonScriptの継承機構についてはChapter 5「継承と探索」に述べられている。

関数オブジェクトのパーツについては、図4-1に示す。

図4-1　関数オブジェクトのパーツ



関数コンテキスト

NewtonScriptは、関数内では定義されてはいないが、使用はしている変数の値を確立するために、コンテキスト(関数の字句及びメッセージ環境)を使う。

字句環境

字句環境に含まれるものは、関数内及び関数内のあらゆる関数の、ローカル変数とパラメタのリストである。たとえば、以下の例に見られる関数の字句環境は関数が呼ばれた時点でのパラメタ`e`の値である。

```
frame1:={
task1: func(e) begin
              //do something
       end
}
```

以下にboldで示される関数`task2`では、字句環境はローカル変数`total`, パラメタ`e`, 関数が呼ばれた時点での`a`の値である。

```
frame2:={
task1: func(e) begin
              local total := e;
              e := 20;
              task2 := func(a) ... ;
              total
       end
}
```

注意

ある種のNewtonScript実装においては、関数本体内で実際に使用された変数だけを字句環境に保存することによって、メモリ使用量の最適化を行っていることがある。◆

メッセージ環境

メッセージ環境は、メッセージのインプリメンタと、メッセージのレシーバからなる。

インプリメンタとは、メソッドが定義されているフレームの事である。メソッドは、フレームの継承チェーンの複数の場所で定義されうることに注意。インプリメンタは、Chapter 5「継承と探索」で説明されている継承ルールを使ってメソッドが見つかる、継承チェーン中の場所のことである。

レシーバは、メッセージを受信したフレームである。メソッドはフレームの継承チェーンの中で探されるから、レシーバとインプリメンタが必ずしも同じものにはならないことに注意されたい。

この点を説明するため、次の frame1 と frame2 を考えてみよう:

```
frame1 := {
greeting : "HI!",
sayHi : func() print(greeting)
};
frame2 := {
greeting: "Hello!",
_proto : frame1
};
```

以下の式で、frame1 は、レシーバとインプリメンタの両方となる:

```
frame1:sayHi();
"HI!"
```

次の式では、frame2 はレシーバ、frame1 がインプリメンタとなる。

```
frame2:sayHi();
"Hello!"
```

変数greetingの値が、レシーバに依存し、インプリメンタには依存していな

いことに注意されたい。この問題に関しては、Page 5-8「スロットとメッセージの探索の継承ルール」を参照のこと。

関数を call with 構文で起動すると、self(レシーバ)の値は、関数のメッセージ環境に保存された値にセットされる。これは、メッセージ送信においてレシーバがメッセージ送信式で指定されたフレームとなるのとは対照的である。

Self

疑似変数 self の値は、常にレシーバと等しい。だから、レシーバを参照するために、self を使用することができる。ただし、self はあくまで「疑似」変数なので、そこに任意の値を代入することはできない。

次の呼び出しでは、self は frame1 となる。

```
x := frame1:sayHi();
```

関数オブジェクトの例

以下の例は、関数内での変数値参照に、関数オブジェクトのコンテキストがどのように使われるかを示す。この例では、関数はネストしており、継承機構が利用されているので複雑である。これは、関数オブジェクトの各部分の動作を実演するためにそうしている。とりあえずこのセクションを飛ばし、Chapter 5「継承と探索」を読んでからここに戻ってきてもよい。

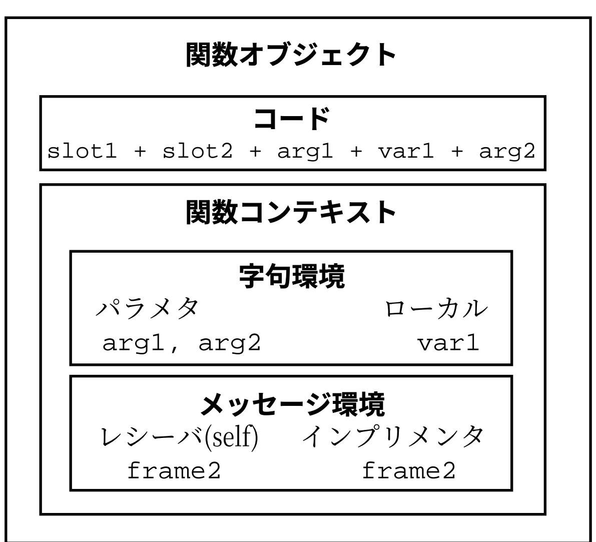
```
frame1 := {slot1 : 5};
frame2 := {
_parent : frame1,
slot2 : 40,
outerMethod : func (arg1) begin
        local var1 := 2000;
        local nestedMethod := func (arg2)
        slot1 + slot2 + arg1 + var1 + arg2;
        nestedMethod;
end;
}
```

outerMethodを、次の方法で呼び出すと ...

```
functionObject1 := frame2:outerMethod(300);
```

outerMethodの戻り値は、関数オブジェクトnestedMethodとなり、それは図4-2に見られる様に、functionObject1に代入される。(nestedMethod自身はまだ実行されていないことに注意)

図 4-2 functionObject1 の解剖



nestedMethodは、その時点の環境を保存している。コンテキスト情報は、引数arg1(300), ローカル変数var1(2000)を持っている。また、メッセージのレシーバであるselfの値も格納しているので、slot2(40), 継承されたslot1(5)の値が格納される。

functionObject1を以下の形で呼び出すと、必要な情報を全て探索できる。

```
call functionObject1 with (10000);
```

戻り値は12345となるであろう。

注意

しかし、次の方法ではうまくいかない。

```
aFrame := {aSlot : functionObject1}
aFrame:aSlot(10000);
```

なぜなら、2行目のメッセージ送信でレシーバ(self)がaFrameになってしまうので、slot1とslot2の値は参照できなくなってしまうのである。(aFrameにはslot1もslot2もなく、さらに継承チェーンも持っていない)だが、次の呼び出しではちゃんと機能する。

```
call aFrame.aSlot with (10000)
```

(selfの値が、aFrameでなく、functionObject1が保存している、メッセージ環境のレシーバに変更されるためである)◆

抽象データ型実装のための関数オブジェクトの利用

オブジェクト指向プログラミングの概念の一つに、「データ隠蔽」というものがある。「データ隠蔽」を簡単に言うと、あるデータ構造中のデータには、特定のメソッドからだけアクセスできるようにし、他からは直接アクセスできないように隠してしまうというものだ。これにより、プログラムの安全性は向上する。

しかし、NewtonScriptにおいてデータ隠蔽は単なるお笑いぐさで、フレームのスロットにはどこからでもばんばん直接アクセスできてしまう。ただ、関数オブジェクトを使って、似たような事をすることは出来る。

次の会計ジェネレータを考えてみよう:

```
MakeAccount := func() begin
       local balance := 0;
       local d := func(amount) begin
              balance := balance + amount;
       end;
       local c := func() begin
              balance := 0;
       end;
       {Deposit: d, Clear: c};
end;
myAccount := call MakeAccount with ();
```

最後の行でのMakeAccount呼び出しによって返ってくるのは、関数オブジェクトd, cを含むフレームである。このフレームの関数オブジェクトは、ローカル変数balanceを、MakeAccountから参照できる。MakeAccountがすでに実行されなくなっていても、balance変数は存在している。ネストした関数DepositとClearがそれを参照しているからである。よって、myAccountを呼ぶことで、隠れた変数balanceを、以下のインスペクタ出力に見られるごとく、変更することができる。

```
call myAccount.Deposit with (50);
#C8       50
call myAccount.Deposit with (75)
#1F4      125
call myAccount.Clear  with ();
#0        0
```

今回は、DepositもClearも、selfに依存する要素は使っていないので、以下のメッセージ送信でも、上のcall/with構文同様上手く動く:

```
myAccount:Deposit (20);
#50       20
```

ネイティブ関数

予約語`native`を関数コンストラクタに指定すると、ある種のコンパイラはその関数をNewtonのCPUが直接実行できる機械語形式にコンパイルする。これをネイティブ関数という。詳しくは*The Newton Programmer's Guide*を参照のこと。

どの関数を`native`に宣言するかについては、考慮すべき点がいくつかある。この問題に関する詳細については、*Newton Toolkit User's Guide*の「Tuning Performance」を参照のこと。

CHAPTER 5

継承と探索

NewtonScript では、オブジェクト指向の機能やコンセプトを二重**継承**機構によってサポートしている。フレームは NewtonScript における基本データ構造である。フレーム間の継承構造は、`_parent`，`_proto` という名前のスロットによりセットされる。この章では、**ペアレント**継承とプロトタイプ(proto)継承について述べる。さらに、以下のことにも触れる:

- フレーム同士の関係のセットアップ方法
- ペアレント・プロトタイプ継承に関するルール
- 継承がスロットとメソッドの探索に与える影響
- 継承がスロットへの値設定(代入)に与える影響
- ペアレント・プロトタイプ継承の使用

継承

NewtonScript には、二つの継承がある。一つはプロトタイプ継承、もう一つはペアレント継承である。

プロトタイプ継承

フレームは、プロトタイプフレームを持ちうる。プロトタイプフレームは、_proto という名前のスロットを通して参照される他のフレームのことだ。フレームは、自分が持たないスロットでも、プロトタイプを通じて継承できる。もし、フレームが、プロトタイプフレームのスロットと同名のスロットを既に持っていれば、プロトタイプの方のスロットはオーバーライドされ、無視される。

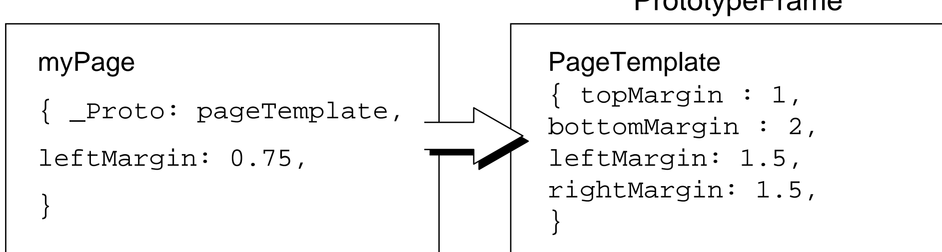
プロトタイプフレーム(protos とか、プロトと略される)からの継承を使うのは次のような理由による:

- Newton では、ユーザーインターフェースの要素はシステムプロトという形で格納されている。それを自分のアプリケーションで継承して使ったり、ある部分をオーバーライドして好みのものに変えたりする事が出来る。
- そうしたデータは通常 ROM または PCMCIA カードに格納されている。

プロトタイプフレームの生成

_proto という名前の特別なスロットにより、フレーム間のプロトタイプ関係を構築できる。例えば、pageTemplate という名前のフレームを myPage という名前のフレームのプロトタイプとして使う場合、myPage の中には、pageTemplate フレームへの参照を行う _proto スロットを組み込む必要がある。このことを図 5-1 に示す。

図 5-1 プロトタイプフレーム



プロトタイプ継承ルール

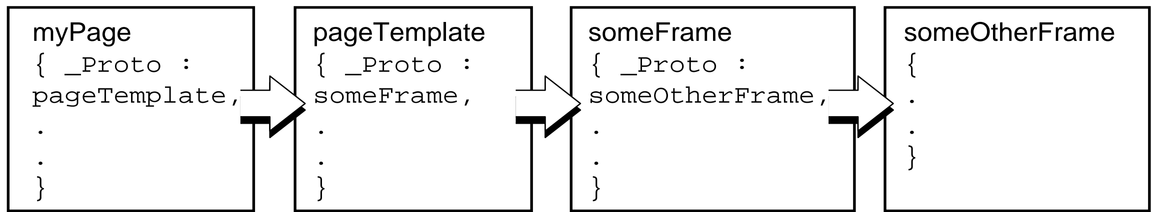
仮に、図5-1に見られるように、myPage 内のメソッドから topMargin スロットを参照したとすると、インタープリタは topMargin スロットの探索を、myPage フレームから始める。myPage フレームに topMargin が見つからないと、次に _proto スロットをたどって pageTemplate を探しに行き、そして値 1 の topMargin が見つかる。これを、「フレームがスロットを**継承**した」と言う。また、leftMargin スロットを myPage 内のメソッドから参照すると、それは myPage フレーム内にすぐ見つかるので(この図の場合は値 0.75)、pageTemplate の方にある leftMargin スロットは参照されない。これを「フレームがスロットをオーバーライドした」という。

メソッドもまたスロットに格納されるので、オーバーライド出来ることに注意すること。

この例では、プロトタイプ参照を一つたどるだけで目的のスロットが見つかったが、このプロトタイプ参照が長い長いチェーンの様にずっと続いていて、目的のスロットが見つからなければ、 _proto スロットをたどった探索が延々と繰り返される。まず最初にスロットを現在のフレームで探索し、それがなければプロトタイプフレームへ、そしてまた見つからなければそのまたプロトタイプへ行き ... という具合に、プロトタイプチェーンの終わりまで探索が続く。

プロトタイプチェーンの例は、図 5-2 に示す。この図では、継承チェーンは現在のフレーム myPage から始まり、矢印方向に右へ進んでいく。

図 5-2 プロトタイプチェーン



ペアレント継承

フレームのプロトタイプ継承とは別に、階層的な親子関係もセットアップできる。

ペアレント継承は次の目的で使用される。

- オブジェクト間の情報共有(動作と、データオブジェクト両方)
- Newton アプリケーションにおけるビュー階層の様な階層形成

ペアレントフレームの生成

フレームの親子リンクは、子供フレーム内の _parent という特別な名前を持つスロットにより提供される。このスロットは、直接コード内でセットできる。また、読者の皆さんが NewtonToolkit を使ってビューを配置しているとき、ペアレント階層は自動的に作られている。図 5-3 に、2 つのフレーム間の親子関係の例を示す。

親フレームは別のフレームの子供になることもできるので、親子関係チェーンもまた、プロトタイプチェーンのように、どんどんのばしていくことが出来る。

図 5-3 親子関係

ペアレント継承ルール

ペアレント継承はプロトタイプ継承と似ている。スロットの継承や、オーバーライドの仕組みはプロトタイプ継承と同じである。

しかし、ペアレント継承では探索されないがプロトタイプ継承では探索されるインスタンスがあったり、代入の時の動作が異なるという、ちょっとした違いがある。

プロトタイプとペアレント継承の組み合わせ

プロトタイプ参照とペアレント参照を使うフレームでは(Newton のアプリケーション上のフレームの多くがこのタイプである)、スロットやメソッドが実行時に参照されるとき、ペアレント継承とプロトタイプ継承が影響し合う。

基本的なルールは以下のようなものだ。

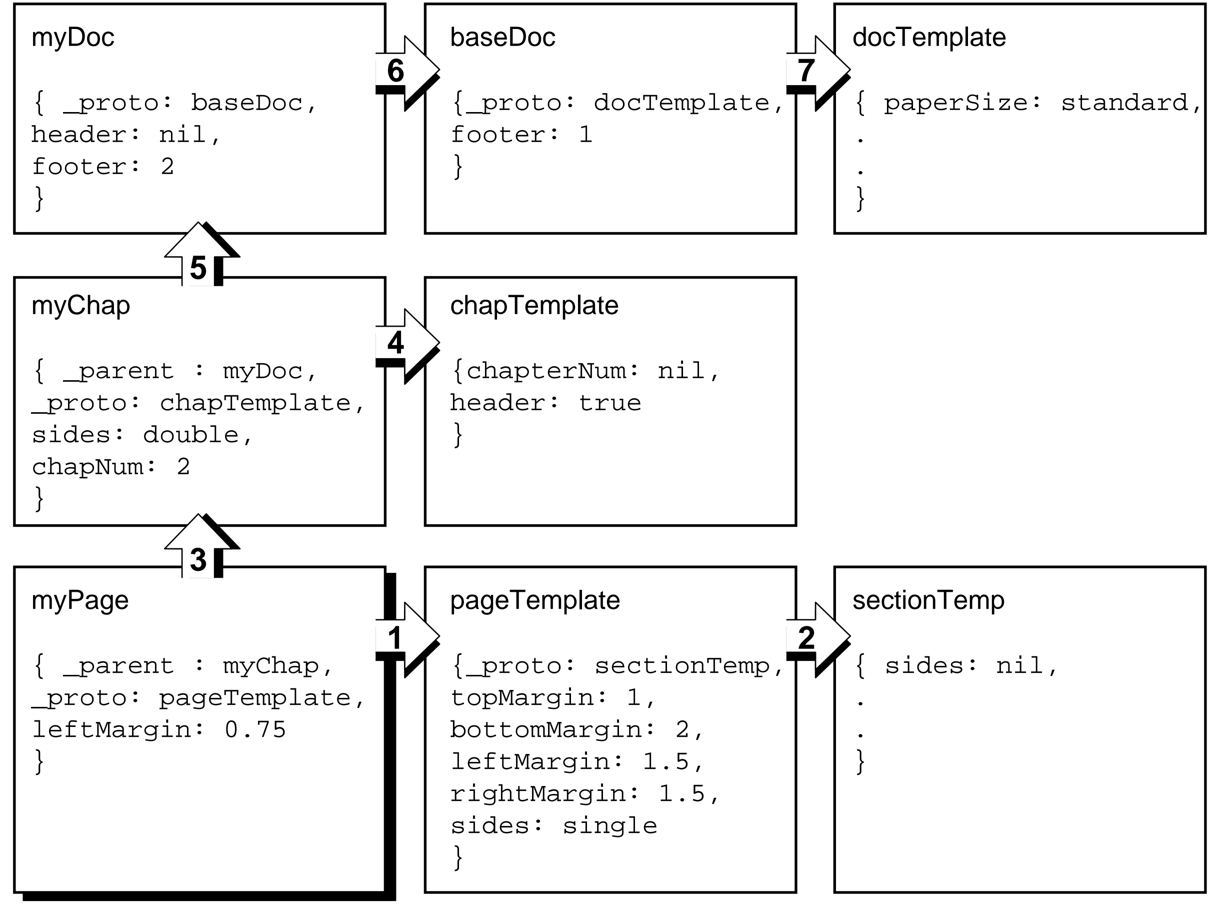
1. 最初に、スロット参照が行われた最初のフレームを探索する。変数探索では、最初のフレームはまず、スロット参照が起こった最初のスロットとなる。最初のフレームは、変数参照の場合は現在のレシーバで、メッセージ送信の場合は指定されたレシーバである。
2. スロットがなければ、最初のフレームのプロトタイプチェーンが探される。
3. 最初のフレームのプロトタイプチェーンの中にスロットが見つからなかった場合は、検索は最初のフレームの一つ上のレベルの親フレームに進み、その親フレームが探され、そこになければ親のプロトタイプチェーンが探される。こうして検索は親の親のレベル、親の親の親の ... とスロットが見つかるまで続く。

この参照順序を、図 5-4 に、番号付きの矢印で示す。

基本的に、プロトタイプ継承はペアレント継承よりも優先する。一つ上の親フレームと、そのプロトタイプフレームを探索にいく前に、同一レベルにあるプロトタイプフレーム全部が探索される。

次のルールが変数の探索ルールに影響することを覚えておいてもらいたい。変数探索時、スロットを探す前に、まず現在のメソッド内でのローカル変数を探し、次にグローバル変数を探し、最後に継承構造を探す。

図 5-4 プロトタイプとペアレント継承影響順序



スロットとメッセージ探索の継承ルール

スロットへのアクセス方法は何通りか用意されている。ある方法では両方の継承チェーンを探すし、ある方法ではプロトタイプチェーンだけを探し、またあるものはどちらも探さない。

スロット名がそれだけで現れた場合、両方の継承チェーンが参照される。例として以下の式では、`chapterNum` と `rightMargin` は図 5-4 で示された順序で(もちろん、ローカルとグローバルどちらも探索された後で)探索される。

```
presentChapter := chapterNum;
if rightMargin > 1.0 then …
```

しかし、以下の例の *frame*`.slot` または *frame*`.(` **パス式** `)` 構文では、プロトタイプチェーンだけが探される。

```
if myChap.header then ...
x:= self.topMargin;
```

メッセージ送信は、*frame*`:`*Message*`()`, `:`*Message*`()` のどちらの書き方をしても、両方の継承チェーンを探索する。`inherited:`*Message*`()` のように予約語 `inherited` を付けた場合は例外で、検索は現在のフレームのプロトタイプから始まりプロトタイプチェーンだけを探索する。

注意

`self:`*Message*`()` にも、対照的な `:`*Message*`()` にも、それぞれ長所と短所がある。`self:`*Message*`()` は `self.`*slot* と見ためが似ているにも関わらず、ペアレント継承チェーンをたどるので、これは混乱を生じうる。一方 `self:`*Message*`()` は論理的にはより読みやすく、常にこの形式を取ることで、一般的なバグを防ぐことが出来る。以下のぱっと見には正しい 2 行のリストを考えて欲しい:

```
:Message1()
:Message2()
```

これはMessage1()の評価結果が何であってもそこにMessage2() を送るが、それには何の期待もできない。self を含めることで、このバグは解消される。このタイプのバグを消すもう一つの方法は、式の最後に必ずセミコロン(;)を置くことである ◆

NewtonScript はまた、フレームスロットへのアクセスに使える 2 つの組み込み関数、GetVariable と GetSlot を持っている。SetVariable は両方の継承チェーンを探し、GetSlot はどちらも探さない。Chapter 6「組み込み関数」を参照のこと。

訳注： 長々と文章で書くより、表の方がわかりやすいので表にした。

アクセス形態	プロトタイプ継承	ペアレント継承
slot	○	○
frame.slot	○	—
frame.（パス式）	○	—
GetVariable(*frame*, *slot*)	○	○
GetSlot(*frame*, *slot*)	—	—
frame:*message*()	○	○
frame:?*message*()	○	○
:*message*()	○	○
inherited:*message*()	○	—
inherited:?*message*()	○	—
symbol exists	○	○
frame.*slot* exists	○	—
frame.（パス式） exists	○	—
frame:*message* exists	○	○
:*message* exists	○	○
HasVariable(*frame*, *slot*)	○	○
HasSlot(*frame*, *slot*)	—	—

スロットの存在をテストするための継承ルール

スロットの存在をテストする継承ルールは、基本的にはスロットの探索と同じである。*slot* `exists` のように、スロット名が単独で現れた場合は、完全に継承チェーンが探される。*frame*`.`*slot* `exists` とか self`.`slot `exists` のように、フレームが明示的に指定された場合は、プロトタイプチェーンだけが探される。

メソッドの場合は、コロンの前にフレーム名が現れようが現れまいが、(*frame*:*message* `exists` とか :*message* `exists` みたいに)両方の継承チェーンが探される。

また、`HasVariable` と `HasSlot` という二つの組み込み関数が、スロット存在確認の為に存在する。`HasVariable` は両方のチェーンを探すが、`HasSlot` はどちらも探さない。

スロットの値をセットするための継承ルール

スロット参照時の継承ルールと、スロット代入時の継承ルールはわずかに異なる。

基本的な違いは、親フレームにあるスロットは書き換えられるが、プロトタイプフレームにあるスロットの値は変えることが出来ないと言うことだ。なぜなら、プロトタイプフレームは、ROM の中に仕込まれていることが多いからである。(もちろん、プロトタイプフレームを最初に作ったときはスロットの値はセットできるが、実行時は無理である)

どのスロットに代入が行われるかを決めるルールを示す：

1. スロットが現在実行中のフレームにある場合、値はそこにセットされる
2. スロットがプロトタイプチェーン内に有る場合、新しいスロットが、現在実行中のフレームに作られ、値がセットされる
3. スロットが現在実行中のフレームの親で見つかると、値はその親のフレー

ムにセットされる。

4. スロットが親のプロトタイプチェーンに見つかると、新しいスロットが、そのスロットが見つかったのと同じレベルの親のスロットにセットされる。

関数の中で変数を作った場合、それはローカル変数となり、関数のスコープの中に制限される。もし、確実にレシーバーの中にスロットを作りたいので有れば、そのことを self を使って、次のように明確に示さないといけない。

```
self.slotName := aValue;
```

もし、現在のフレームの親に確実にスロットをセットしたいのなら、self._parent.theSlot のようにして、強制的に親フレームにスロットを作ることが出来る。

注意

_parent とだけ書いて、親を見に行くのは安全ではない。
ちょっと違うやり方がある、それは ビューシステムのメッセージ :Parent() を使うことで、これはレシーバの親フレームを返す。
また、*frame*._parent とか self._parent もこの問題を解消する。要約すると次のようになる：

_parent	危険
frame:Parent()	OK
self._parent	OK (:Parent() も意味は同じ)
frame._parent	OK (*frame*:Parent() も意味は同じ)

:Parent() メソッドについては、*Newton Programmer's Guide* を参照のこと。◆

オブジェクト指向な例

実験できる継承構造を構築すれば、継承をもっと良く理解できるだろう。次のコードを使ってみる:

```
frame1 := {
       slot1: "slot1 from frame1",
       slot6: 99 };

frame2 := {
       _parent: frame1,
       slot1: "slot1 from frame2",
       slot2: "slot2 from frame2" };

frame3 := {
       slot3: "slot3 from frame3",
       slot5: 42};

frame4 := {
       _parent: frame2,
       _proto: frame3,
       msg1: func() begin
              //show slot from parent inheritance
              Print(slot1);

              //show slot from proto inheritance
              Print(slot3);

              //show slot from parent inheritance - again -
              //but doesn't work because
              // self.slot1 only searches proto chain
              Print(self.slot1);

              //show slot from proto inheritance - again
              Print(self.slot3);
       end }
```

これは、図 5-5 に示す継承構造を生成する。メッセージ `frame4:msg1()` が送られたとき、以下の出力が作られる:

```
"slot1 from frame2"
"slot3 from frame3"
NIL
"slot3 from frame3"
```

図 5-5 継承構造

frame1 — Parent Frame

```
{
slot1: "slot1 from frame1"
slot6: 99
}
```

frame2 — Parent Frame

```
{
_parent: frame1
slot1: "slot1 from frame2"
slot2: "slot2 from frame2"
}
```

frame4

```
{
_parent : frame2
_proto: frame3
msg1: func()...
}
```

frame3 — Prototype Frame

```
{
slot3: "slot3 from frame3"
slot5: 42
}
```

空のページ

CHAPTER 6

組み込み関数

NewtonScript は、多数の組み込み関数をサポートする。これらは、以下の関数グループに分類される:

- オブジェクトシステム
- 文字列
- ビット演算
- 配列とソートされた配列
- 演算
- 浮動小数演算
- 浮動小数演算の制御
- 財務
- 例外処理
- メッセージ転送
- データ抜き取り
- データ詰め込み
- グローバル変数・関数の取得とセット
- その他

注意:

このドキュメントで使用されているインスペクタのサンプルでは、#4945 のような、シャープマークと数字組み合わせが印刷されている。この情報は、データへのシステム内部のポインタなので無視してよい。◆

互換性

このセクションでは、Newton システムソフトウェア 2.0 で行われた組み込み関数への変更について述べる。

新しい関数

以下の新しい関数が、このリリースから追加された。

新しいオブジェクトシステム関数

以下の新しいオブジェクトシステム関数が追加された。

`GetFunctionArgCount`
`IsCharacter`
`IsFunction`
`IsInteger`
`IsNumber`
`IsReadOnly` (1.0 時代から存在したが、今回文書化された)
`IsReal`
`IsString`
`IsSubclass` (1.0 時代から存在したが、今回文書化された)
`IsSymbol`
`MakeBinary`

SetVariable
SymbolCompareLex

新しい文字列関数

以下の新しい文字列関数が追加された。

CharPos
StrExactCompare
StrTokenize
StyledStrTruncate

新しい配列関数

以下の新しい配列関数が追加された。

ArrayInsert
InsertionSort
LFetch
LSearch
NewWeakArray
StableSort

新しいソートされた配列のための関数

ソートされた配列を取り扱うための以下の新しい関数が追加された。これらの関数はバイナリサーチアルゴリズムに基づいているので、関数名の最初にBが付いている。

BDelete
BDifference
BFetch
BFetchRight

```
BFind
BFindRight
BInsert
BInsertRight
BIntersect
BMerge
BSearchLeft
BSearchRight
```

新しいメッセージ送信関数

以下の、直接メッセージを送信するユーティリティ関数が追加された。

```
PerformIfDefined
ProtoPerform
ProtoPerformIfDefined
```

新しいデータ詰め込み関数

以下のデータ詰め込み用の関数が追加された。

```
StuffCString
StuffPString
```

新しいグローバルなものを取得したり、セットする関数

以下の、グローバルな変数及び関数を取得・セット・存在チェックするための関数が追加された。

```
GetGlobalFn
GetGlobalVar
GlobalFnExists
GlobalVarExists
```

DefGlobalFn
DefGlobalVar
UnDefGlobalFn
UnDefGlobalVar

新しいその他の関数

以下の関数が追加された。

BinEqual

互換性の為に残っている既に時代遅れな関数

以前の *NewtonScript Programming Language* で文書化されていたいくつかの関数はすでに廃止された。しかし、これらは古いアプリケーションとの互換性のために残されている。将来のシステムソフトウェアではおそらくサポートされないので、これからはプログラミングに以下の関数を使用しないこと。

ArrayPos（LSearch を使うこと）
StrTruncate（StyledStrTruncate を使うこと）

オブジェクトシステム関数

このセクションで述べられている関数は、NewtonScript のオブジェクトを操作する。これらは、スロットの削除、フレームのクローズなどを行う。

ClassOf

ClassOf(*object*)

オブジェクトのクラスを返す。

object クラスを返したいオブジェクト。

戻り値はシンボルである。共通のオブジェクトクラスは次の通り: 'int, 'char, 'boolean, 'string, 'array, 'frame, 'function, 'symbol。これはオブジェクトのプリミティブクラスと同じである必要がないことに注意。バイナリ、配列、フレームオブジェクトに関しては、プリミティブクラスと異なるクラスが指定されていることがある。

明示的にクラス指定のされてないフレームまたは配列のプリミティブクラスは、それぞれ 'frame と 'array である。フレームが class というスロットを持っていれば、そのスロットの値が返される。以下にいくつかのサンプルを示す:

```
f:={multiply:func(x,y) x*y};
classof(f);
#1294    Frame
f:={multiply:func(x,y) x*y, class:'Arithmetic};
classof(f);
#1294     Arithmetic
s:="India Joze";
classof(s);
#1237    String
```

page 6-14, PrimClassOf も参照のこと。

Clone

Clone(*object*)

オブジェクトの「浅い」コピーを返す。つまり、オブジェクト中の参照値はコピーされるが、その参照値が参照しているデータはコピーされない。

object コピーするオブジェクト。

以下に例を示す：

```
SeaFrame :=  {Ocean: "Pacific", Size: "large" , Color: "blue"};
```

```
seaFrameCopy := clone(seaFrame);
seaFrameCopy.Deep := true;
seaFrame
#441896D  {Ocean: "Pacific", size: "large", Color: "blue"}
seaFrameCopy
#4418B0D  {Ocean: "Pacific", size: "large", Color: "blue",
       Deep: TRUE}
```

DeepClone

`DeepClone(`*object*`)`

オブジェクトの「深い」コピーを返す。つまり、オブジェクト中の参照値は全てコピーされ、その参照値が参照しているデータもコピーされる。マジックポインタ(ROM 内オブジェクトへのポインタ)が参照しているものもコピーされる。

object コピーされるオブジェクト。

全てのデータ構造が RAM 内に存在するとは保証されない。(フレームスロットの名前となるシンボルのような特定の情報はオリジナルのオブジェクトと共有されるであろう)

これとは対照的に、`Clone` は「浅い」コピーしか作らない。また、`EnsureInternal` 関数は、全てのオブジェクトが内蔵 RAM 上に存在することを保証する。

GetFunctionArgCount

`GetFunctionArgCount(`*function*`)`

関数が必要とする引数の数を返す。

function 引数の数を知りたい関数。

GetSlot

GetSlot(*frame*, *slotSymbol*)

フレーム内のスロットの値を返す。指定されたフレームだけが探索される。

frame スロットを探したいフレームへの参照。

slotSymbol 値を取得したいスロットの名前となるシンボル。

スロットが存在しなければ、関数は nil を返す。

GetVariable と違って、GetSlot は指定されたフレームだけを探す。スロット探索に継承ルールは適用されない。

そういう意味で、ドット演算子は GetSlot 関数と似ており、同様にフレームスロットの値を返す。例えば、frame.slot という式は指定されたスロットの値も返すが、指定されたフレームにスロットが見つからなかった場合、プロトタイプフレームが探索される。(親フレームは探されない)

GetVariable

GetVariable(*frame*, *slotSymbol*)

フレーム内のスロットの値を返す。スロットが見つからない場合、nil が返る。

frame スロットの探索を開始したいフレームへの参照。

slotSymbol 探したいスロットの名前を指定するシンボル。

この関数は、スロットの探索を指定された *frame* から開始し、プロトタイプ・ペアレント継承の両方を使って探索する。

HasSlot

HasSlot(*frame*, *slotSymbol*)

スロットがフレーム内にあれば、非 `nil` の値を返す。なければ `nil` を返す。スロット探索に継承は使用されない。

frame スロットを探索したいフレームの名前。

slotSymbol 見つけたいスロットの名前を指定するシンボル。

この関数はスロットの探索を指定されたフレームから開始し、プロトタイプ・ペアレント継承の両方を使って探索する。

訳注: これは間違っている。HasSlot は、継承を使わない。

HasVariable

`HasVariable(`*frame*`,` *slotSymbol*`)`

フレーム内にスロットがあれば非 `nil` の値を返す。なければ `nil` を返す。この関数はスロットが指定したフレームになければプロトタイプ・ペアレント継承両方を使う。

frame スロット検索を開始したいフレームの名前。

slotSymbol 存在をチェックしたいスロット名を示すシンボル。それはシンボルなので、シングルクォートを前に置かないといけない。

Intern

`Intern(` *string* `)`

パラメタの文字列から、シンボルを生成して返す。その名前のシンボルが既に定義済みなら、定義済みのシンボルが返される。

string シンボルの名前。

IsArray

`IsArray(`*obj*`)`

obj が配列なら、非 `nil` を返す。

obj テストしたいオブジェクト。

IsBinary

`IsBinary(`*obj*`)`

obj がバイナリなら非 `nil` を返す。

obj テストしたいオブジェクト。

IsCharacter

`IsCharacter(`*obj*`)`

obj が文字なら非 `nil` を返す。さもなくば `nil` を返す。

obj テストしたいオブジェクト。

IsFrame

`IsFrame(`*obj*`)`

obj がフレームなら非 `nil` を返す。

obj テストしたいオブジェクト。

IsFunction

`IsFunction(`*obj*`)`

obj が関数なら非 `nil` を返す。さもなくば `nil` を返す。

obj テストしたいオブジェクト。

IsImmediate

`IsImmediate(`*obj*`)`

obj がイミディエイトなら、非 `nil` を返す。

obj テストしたいオブジェクト。

IsInstance

`IsInstance(`*obj*`, `*class*`)`

class シンボルが、*obj* のクラスまたはそのサブクラスと同じなら、非 `nil` を返す。

obj テストしたいオブジェクト。

class クラスシンボル。

これは、次の書き方と同じ: `IsSubclass(ClassOf(`*obj*`), `*class*`)`

IsInteger

`IsInteger(`*obj*`)`

obj が整数なら非 `nil` を返す。さもなくば `nil` を返す。

obj テストしたいオブジェクト。

IsNumber

`IsNumber(`*obj*`)`

obj が数値(整数または実数)なら、非 `nil` を返す。さもなくば `nil` を返す。

obj テストしたいオブジェクト。

IsReadOnly

`IsReadOnly(`*obj*`)`

obj が読み込み専用なら、非 `nil` を返す。さもなくば `nil` を返す。配列、フレーム、バイナリオブジェクトが書き込み可能かどうか特定するのに `IsReadOnly` を使える。

obj テストしたい。配列、フレーム、バイナリオブジェクト。(整数の様なイミディエイトオブジェクトは、常に読み出し専用である)

例:

```
if IsReadOnly(viewBounds) then
       viewBounds := Clone(viewBounds);
```

この関数は、オブジェクトの場所を特定するのに使うことは出来ない。つまり、ヒープにあるのか、ROM にあるのか、はたまたプロテクトされたメモリにあるのか特定できないのである。NewtonScript では、ヒープ中に読み込み専用オブジェクトを置くこともできるし、別の場所にあっても書き込み出来るオブジェクトを作ることが出来るのである。

IsReal

`IsReal(`*obj*`)`

obj が実数なら、非 `nil` を返し、さもなくば `nil` を返す。

obj テストしたいオブジェクト。

IsString

`IsString(`*obj*`)`

obj が文字列なら非 `nil` を返し、さもなくば `nil` を返す。

obj テストしたいオブジェクト。

IsSubclass

`IsSubclass(`*sub*`, `*super*`)`

クラスが他のクラスのサブクラスかどうかチェックする。

sub テストしたいクラスシンボル。

super クラスシンボル。

この関数は *sub* が *super* のサブクラスであったり、*super* と同じなら非 `nil` を返す。*sub* が *super* のサブクラスでなければ `nil` を返す。Page 6-11 の関連する関数 `IsInstance` も参照。

IsSymbol

`IsSymbol(`*obj*`)`

obj がシンボルなら非 `nil` を返す。さもなくば `nil` を返す。

obj テストしたいオブジェクト。

MakeBinary

`MakeBinary(`*length*`, `*class*`)`

指定された長さとクラスを持つバイナリオブジェクトを新規作成する。

length バイナリオブジェクトのバイト数。

class クラスを指定するシンボル。

Map

`Map(`*obj*, *function*`)`

配列または、フレームの各要素のスロット名とその値それぞれに関数を適用する。

obj 配列またはフレーム。

function `nil` を返す。*obj* 内の要素またはスロットに適用したい関数。関数には `slot` と `value` という二つのパラメタが渡される。`slot` パラメタには、*obj* が配列であれば整数が含まれ、*obj* がフレームならスロット名を示すシンボルが含まれる。`value` パラメタには、`slot` パラメタで参照される配列やフレームの値が含まれる。

これは次の書き方と同じである:

```
foreach slot,value in obj do call function with (slot,value)
```

PrimClassOf

`PrimClassOf(`*obj*`)`

オブジェクトのプリミティブクラスを返す。

obj プリミティブクラスを返すオブジェクト。

この関数は、オブジェクトのプリミティブなデータ構造タイプを特定するシンボルを返す。返ってくるシンボルは次のいずれかである：`'immediate`, `'binary`, `'array`, `'frame`。

Page 6-5 の `ClassOf` も参照。

RemoveSlot

RemoveSlot(*obj*, *slot*)

フレームまたは配列からスロットを削除する。

obj スロットを削除する配列またはフレームの名前。

slot 削除したいフレームの名前を示すシンボル。あるいは削除したい配列要素の添え字。*obj* 中の *slot* を検索する際に、継承ルールは全く適用されないことに注意。

この関数は変更されたフレームまたは配列を返す。スロットが存在しない場合、何も行われず、無変更のフレームまたは配列が返る。*obj* が読み込み専用の場合、システムは例外を throw する。

ReplaceObject

ReplaceObject(*originalObject*, *targetObject*)

オブジェクトへの全ての参照を、別のオブジェクトに付け替える。

originalObject オリジナルオブジェクト。

targetObject *originalObject* に参照を向け替えたいオブジェクト。

この関数は常に nil を返す。

イミディエイトオブジェクトをこの関数のパラメタに指定することは出来ない。

以下に例を示す:

```
x:={name:"Star"};
y:={name:"Moon"};
replaceobject(x,y);
x;
#469E69   {name: "Moon"}
```

```
y;
#46A1E9    {name: "Moon"}
```

SetClass

SetClass(*obj*, *classSymbol*)

オブジェクトのクラスをセットする。

obj クラスをセットするオブジェクト。

classSymbol オブジェクトに与えるクラスの名前となるシンボル。

この関数はクラスがセットされたオブジェクトを返す。

配列、フレーム、バイナリオブジェクトに対してクラスをセットできる。イミディエイトオブジェクトにはクラスをセットできないことに注意。フレームの中に、classスロットがない場合、classスロットが作成される。例を示す:

```
x:={name: "Star"};
setclass(x, 'someClass);
#46ACC9    {name: "Star",
            class: someClass}
```

SetVariable

SetVariable(*frame*, *slotSymbol*, *value*)

フレーム内のスロットに値をセットする。*value*の値が返る。

frame スロット探索を開始するフレームへの参照。

slotSymbol 値をセットしたいスロットの名前を示すシンボル。スロットが存在しなければ、フレーム内に作成される。

value スロットの新しい値。

この関数はスロットの検索を指定されたフレームから開始し、プロトタイ

プ・ペアレント継承の両方を使う。

プロトタイプチェーンにスロットが見つかっても、そこにはセットされず、*frame* で指定したフレームまたはそのペアレントチェーンにスロットが作られる。変数の値設定においては、通常の継承ルールに従う。

SymbolCompareLex

`SymbolCompareLex(`*symbol1*`, `*symbol2*`)`

シンボルを字句的に比較する。この関数は *symbol1* が、*symbol2* より小さいときに負の値を返し、二つのシンボルが等しければ 0 を返し、*symbol1* が *symbol2* より大きければ正の値を返す。大文字小文字は区別されない。(だから、`'Hello` と `'hello` は同じ)

symbol1 シンボル

symbol2 シンボル

TotalClone

`TotalClone(`*obj*`)`

オブジェクトの「深い」コピーを作って返す。つまり、オブジェクト内で参照されるデータがコピーされる。

obj コピーされるオブジェクト。

この関数は `DeepClone` に似ているが、この関数が返すオブジェクトは全て内蔵 RAM 上に存在することが保証されるところが違う。また、`DeepClone` とは違って、`TotalClone` はマジックポインタを追跡しないので、マジックポインタ経由で参照されるオブジェクトはコピーされない。

文字列関数

これらの関数は文字列を操作する。

BeginsWith

```
BeginsWith( string, substr )
```

string が、*substr* で始まっていれば非 `nil` を返す。さもなくば `nil` を返す。この関数は大文字小文字や、発音符号は区別しない。空の *substr* は、あらゆる文字列とマッチする。

string	テストする文字列。
substr	文字列。

Capitalize

```
Capitalize( string )
```

文字列中の最初の文字を大文字にして、結果を返す。文字列 *string* は変更される。

string	変更する文字列。

CapitalizeWords

```
CapitalizeWords( string )
```

文字列中の、各単語の最初の文字を大文字にして結果を返す。文字列 *string* は変更される。

string	変更する文字列。

CharPos

`CharPos(`*str*, *char*, *startpos*`)`

指定された文字列中で、*startPos* の位置から始めて、*char* で指定された文字が見つかる場所を返す。(見つからなければ `nil` を返す)

str 指定される文字列。

char 文字列中で探したい文字。

startpos 文字検索を開始する位置。

Downcase

`Downcase( string )`

文字列中の文字を全て小文字にして返す。*string* は変更される。

string 変更される文字列。

EndsWith

`EndsWith(` *string*, *substr* `)`

string が、*substr* で終わっていたら非 `nil` を返す。さもなくば `nil` を返す。この関数は、大文字小文字と発音記号を区別しない。空の *substr* は、全ての文字列とマッチする。

string テストする文字列。

substr 文字列。

IsAlphaNumeric

`IsAlphaNumeric(`*char*`)`

char が、数字または英文字なら非 `nil` を返す。さもなくば `nil` を返す。

char テストする文字。

IsWhiteSpace

`IsWhiteSpace(`*char*`)`

char がスペース(`$\20`)、タブ(`$\09`)、改行(`$\0A`)、復帰(`$\0D`)なら非 `nil` を返す。さもなくば `nil` を返す。

char 文字。

SPrintObject

`SPrintObject(` *obj* `)`

渡されたオブジェクトを文字列にして返す。数値、文字列、文字、シンボルは通常の文字列表現にされて戻される。フレーム、配列、バイナリについては、この関数は空の文字列を返す。

論理値を文字列に変換するには、非 `nil` か `nil` かをチェックして、適当な文字列を返してやらなければならない。

注意

この関数は現在のロケーション情報に基づいて、数値のフォーマットを変更する。実数は期待しない形式にフォーマットされるかもしれない ◆

StrCompare

`StrCompare(` *a*, *b* `)`

文字列 *a* が文字列 *b* より小さい場合、負の値を返す。等しければ 0 を返す。*a* の方が大きければ正の値を返す。大文字小文字は区別されない。(だから、`"Hello"` と `"hello"` は同じである)

a	文字列
b	文字列

これは、文字列の内容比較であって、ポインタの比較ではないことに注意。

大文字小文字を区別して比較する場合は、`StrExactCompare` を使うこと。

StrConcat

`StrConcat(` *a*`,` *b* `)`

文字列 *a* に文字列 *b* を追加し、新しい文字列を結果として返す。

a	文字列。
b	文字列。

StrEqual

`StrEqual(` *a*`,` *b* `)`

二つの文字列が等しければ非 `nil` を返す。

a	文字列。
b	文字列。

大文字小文字は区別されない。この関数は内容比較であって、ポインタの比較でないことに注意。

大文字小文字を区別して比較するには、`StrExactCompare` を使うこと。

StrExactCompare

`StrExactCompare(` *a*`,` *b* `)`

文字列 *a* が *b* より小さければ負の値、等しければ 0、*a* が大きければ正の値を

返す。大文字小文字と発音符号は区別される。(よって、"Hello" と "hello" は等しくない)

a 文字列

b 文字列

これは、二つの文字列の内容比較であり、ポインタ比較ではないことに注意。

大文字小文字を区別せず比較する場合は、StrCompare あるいは StrEqual を使用すること。

StrLen

StrLen(*string*)

文字列中の文字数を返す。null 終端文字は存在していてもカウントされない。

string 文字列。

StrMunger

StrMunger(*dstString*, *dstStart*, *dstCount*, *srcString*, *srcStart*, *srcCount*)

dstString 中の文字を、*srcString* 中の文字で置き換え、切った張ったが終了した後で、相手文字列 *dstString* を返す。この関数は、*dstString* を破壊する。

dstString 相手文字列。この文字列は書き込み可能でなければならない。文字列リテラルは指定できない。それを指定すると、例外が発生する。Clone または類似の関数を使って、文字列リテラルから書き込み可能なコピーを作らなければならない。

dstStart *dstString* 内での開始位置。

dstCount *dstString* 内で置換される文字数。ここに nil を指定すると、文字列の最後まで置換が行われる。

srcString	文字列。nil を指定すると、*dstString* 内の文字を削除できる。
srcStart	*srcString* 内での、*dstString* にコピーする文字の開始位置。
srcCount	*srcString* から使用する文字数。nil を指定すると、*srcString* の最後までが使用される。

例:

```
StrMunger("abcdef", 2, 3, "ZYXWV", 0, nil)
"abZYXWVf"
StrMunger can also be used to concatenate large strings; for
example:
StrMunger(str1, StrLen(str1)+1, nil, str2, 0, nil);
```

StrPos

StrPos(*string*, *substr*, *start*)

string 中での、*substr* の開始位置を返す。*substr* が見つからなければ nil を返す。検索は、*start* 文字目から始まる。(文字列の最初の文字の位置は 0 である) この関数は大文字小文字を区別しない。

string	文字列。
substr	文字列。
start	整数。

例:

```
StrPos("abcdef", "Bcd", 0)
1
```

StrTokenize

StrTokenize(*str*, *delimiters*)

文字列を、引数 *delimiters* によって定義されるデリミタで切り出す。引数なしでクロージャを呼び出す度、次のトークンが返る。トークンがなくなると、`nil` が返る。

str トークンを切り出したい文字列。

delimiters 文字列を区切るための文字または文字列(文字のリスト)。

例えば、文章をスペースで区切られた単語に分割するには、次のようにすればよい:

```
fn := StrTokenize("the quick green fox", $ );
#441BE8D  <function, 0 arg(s) #441BE8D>
      while x := call fn with () do Print(x);
"the"
"quick"
"green"
"fox"
#2        NIL
```

StyledStrTruncate

`StyledStrTruncate(`*string*`,` *length*`,` *font*`)`

length で指定されたピクセル数に、文字列を切り捨てる。(もちろん、長さはnull 終端文字を含まない)。切り捨てられた文字列を返す。

string 文字列。

length 文字列を切り捨てる長さをピクセル数で指定する整数。

font フォント指定。これは、指定された長さに文字列の何文字が入るかを特定するために使用される。さらに詳しいフォントの指定については、*The Newton Programmer's Guide* の章「文字入力と表示」の「フォント指定」セクションを参照。

この関数は、切り捨てられた文字列の最後に、省略符号(...)を付ける。

SubStr

`SubStr(` *string*`,` *start*`,` *count* `)`

string の、*start* 文字目から *count* 個の文字を含む新しい文字列を返す。最初の文字の位置は、0 である。

string 文字列。

start 整数。

count 整数。

TrimString

`TrimString(` *string* `)`

全てのホワイトスペース(空白、タブ、改行、復帰)を文字列の頭と尻尾から取り除き、その結果を返す。*string* は変更される。

string 文字列。

Upcase

`Upcase(` *string* `)`

string 内の全ての文字を大文字にし、結果を返す。*string* は変更される。

string 文字列。

ビット演算関数

以下の関数は、ビットごとの論理演算を行う。

Band, Bor, Bxor, Bnot

`Band(`*a*`,` *b*`)`

`Bor(`*a*`,` *b*`)`

`Bxor(`*a*`,` *b*`)`

`Bnot(`*a*`)`

これらのビット演算関数は、一つまたは二つの引数を元に行った演算結果を、整数にして返す。これらは、それぞれビットごとの AND, OR, XOR, NOT を行う。

a 整数。

b 整数。

配列関数

これらの関数は、配列を操作する。

AddArraySlot

`AddArraySlot (`*array*`,` *value*`)`

配列に、新しい要素を追加する。

array 配列。

value 配列に、新しい要素として追加される値。

例:

```
myArray := [123, 456]
#1634 myArray
addArraySlot (myArray, "I want chopstix")
#12 "I want chopstix"
myArray
#1634 [123, 456, "I want chopstix"]
```

Array

`Array(` *size*, *initialValue* `)`

size 個の要素を持ち、それぞれの値が *initialValue* である新しい配列を返す。

size	整数。
initialValue	値。

ArrayInsert

`ArrayInsert(`*array*, *element*, *position*`)`

配列内に新たな要素を挿入し、変更された配列を返す。

array	変更される配列。
element	配列に挿入される要素。
position	新たな要素が挿入される位置へのインデックス。0 を指定すると、要素が配列の最初に挿入される。`Length(`*array*`)` の評価結果を指定すると、配列の一番最後に要素が挿入される。

配列の長さは一つ増加する。

ArrayMunger

`ArrayMunger(`*dstArray*`,` *dstStart*`,` *dstCount*`,` *srcArray*`,` *srcStart*`,` *srcCount*`)`

dstArray 内の要素を、*srcArray* の要素を用いて置換し、配列の切った張ったが完了した後、相手先の配列 *dstArray* を返す。この関数は *dstArray* を破壊する。

dstArray	相手先の配列。
dstStart	相手先の配列内での開始位置。
dstCount	*dstArray* 内で置換される要素の数。*dstCount* に `nil` を指定すれば、配列の最後の要素までが対象となる。
srcArray	配列。*srcArray* に `nil` を指定すると、配列要素の削除が起こる。
srcStart	相手先の配列に配置されるソース配列内の要素取り出し開始位置を指定する。
srcCount	ソース配列の中で使用する要素の数を指定する。`nil` を指定すると、ソース配列の最後の要素までが使用される。

例:

```
ArrayMunger([10,20,30,40,50], 2, 3, [55,66,77,88,99], 0, nil)
[10, 20, 55, 66, 77, 88, 99]
```

`ArrayMunger` は、二つの配列を結合する最も効果的な方法である。

`B` を `A` の前に置くには次のようにする:

```
ArrayMunger(A, 0, 0, B, 0, nil)
```

`B` を `A` の後に追加するには次のようにする:

```
ArrayMunger(A, Length(A), 0, B, 0, nil)
```

この操作は、`SetUnion`(Page6-36) でも行うことが出来るが、それは要素の重

複を阻止する余分なオプションを持っている。しかし、そうした性質を必要としないのであれば、ArrayMunger の方がずっと処理が早い。

ArrayRemoveCount

ArrayRemoveCount(*array*, *startIndex*, *count*)

配列から一つ以上の要素を削除する。

array 要素を削除したい配列。このパラメタは、この関数によって変更される。

startIndex 削除する最初の要素を指定する整数。

count 削除する要素の個数を示す整数。

削除された要素に続く要素は、左にシフトされるので、空の要素が残ることはない。

InsertionSort

InsertionSort(*array*, *test*, *key*)

等価な要素間の元の並びは維持したまま、配列をソートする。

array ソートされる配列。

test 配列がソートされる方法を示す。*test* パラメタについては、page 6-39 の説明を参照。

key 各配列要素に関するキーの値を定義する。nil, パス式、一つのパラメタを取る関数を指定する。*key* パラメタについては、page 6-39 の説明を参照。

このソートはほとんどソートされている小さい配列に対して上手く働く。このソートは $O(n^2)$ のソートである。大きな配列をソートするには、Sort または、StableSort を使用すること。

Length

Length (*array*)

配列中の要素の数、フレーム中のスロットの数、バイナリオブジェクトのバイト単位でのサイズを返す。

array 配列、フレーム、バイナリオブジェクト。

例:

```
myArray := [123, 456, "I want chopstix"];
length (myArray)
#12    3
```

配列の添え字は0から始まるが、lengthは、文字の数を返す。よって、この例での最後の要素への添え字は2である。

注意

この関数に文字列を渡すと、文字列が占領しているバイト数を返す。文字列の長さを得る場合には、StrLenを代わりに使うこと
◆

LFetch

LFetch(*array*, *item*, *start*, *test*, *key*)

配列内を線形サーチして、指定された要素を探す。要素が見つかれば、その要素を返す。見つからないときはnilを返す。また、*start*で指定した位置が、配列の長さ以上ならnilを返す。

array 検索対象の配列。

item 検索したいキーの値。

start 検索を開始する配列のインデックス。

test — *key* の値を比較する方法を示す。以下のシンボルの内の一つを指定する:

`'|=|` — 比較されるオブジェクトがイミディエイトか実数なら、値が等しいかどうかをチェックする。参照オブジェクトなら、その同一性がチェックされる。

`'|str=|` — 文字列なら、文字列の内容が等しいかどうかチェックされる。

また、非標準的なソートの状況のため、二つのキーを比較して等しいか等しくないかという比較結果を論理値または整数で返す関数オブジェクトも指定できる。この関数は等しいかどうかをテストするために呼ばれる。この関数は二つのパラメタ A, B を取る。A は LFetch に渡された item であり、B はテストされる配列の要素である。

アイテムが等しい場合、関数は非 nil（あるいは 0）を返し、さもなくば nil(あるいは非 0 の整数)を返さなければならない。

test に関数オブジェクトを渡すと、定義済みシンボルを使う場合に比べて、パフォーマンスが非常に低下する。

key — 各配列要素のキー値を定義する。`nil`, パス式、一つのパラメタを取る関数を指定する。Page 6-39 の説明を読むこと。

この関数は `LSearch` とよく似ているが、`LSearch` は、見つかった要素への添え字を返す。

使用する配列が既にソートされたものなら、関数 `BFetch` を使える。この関数はバイナリサーチアルゴリズムに基づいているので、`LFetch` や `LSearch` と比べると、大きな配列に適用したときのスピードが非常に早い。しかし、

BFetch はソートされた配列にしか使用できない。

LSearch

LSearch(*array*, *item*, *start*, *test*, *key*)

配列内を線形サーチして、指定された要素を探す。要素が見つかれば、その要素への添え字を返す。見つからないときは nil を返す。あるいは、*start* で指定した位置が、配列の長さ以上なら nil を返す。

パラメタは LFetch と同じなので省略。

この関数は、LFetch と似た働きをするが、LFetch が、添え字の代わりに見つかった要素を返すところが違う。

使用する配列が既にソートされたものなら、関数 BFind を使える。この関数はバイナリサーチアルゴリズムに基づいているので、LSearch と比べると、大きな配列に適用したときのスピードが非常に早い。しかし、BFind はソートされた配列にしか使用できない。

NewWeakArray

NewWeakArray(*length*)

length で指定された個数の要素を持つ「弱い」配列を新規作成して返す。要素の初期値は nil である。

length 作成される配列の長さを指定する整数。

「弱い」配列とは、配列要素が参照するオブジェクトがガベージコレクションの対象になることを防止しない配列である。だから、「弱い」配列からしか参照されていないオブジェクトは、次のガベージコレクションサイクルで破壊される。これが発生すると、「弱い」配列内にある参照値は、nil で置き換えられる。

「弱い」配列の目的は、ガベージコレクションの対象になることを防止しないようなオブジェクトの貯蔵庫を作る事にある。例えば、特定の型の存在するオブジェクトの全てを配列内に保持したいとして、個々のオブジェクトが生成されたときにそれを配列に追加することは出来る。もし、通常の配列を使っているなら、そうしたオブジェクトはガベージコレクションされることはない。それらは常に配列の中から参照されるからだ。そしてシステムはメモリーを使い果たしてしまうだろう。だが、「弱い」配列を使うことによって、そこからの参照がガベージコレクションには影響を与えないため、オブジェクトは普通にガベージコレクションされてしまい、メモリが解放される。

SetAdd

`SetAdd` （*array*， *value*， *uniqueOnly*）

指定された配列に要素を追加して、変更された配列を返す。要素が追加されない場合は `nil` が返る。

array	*value* で指定された要素が追加される配列。
value	*array* で指定される配列に追加される要素。
uniqueOnly	ユニークな要素だけを追加するかどうか指定する。もし、これが非 `nil` なら、`SetAdd` は *value* の値が配列内に既に存在していないときに限って要素を追加する。もし、指定した値が配列内に既に存在するようなら、`SetAdd` は `nil` を返し、要素は追加されない。*uniqueOnly* が `nil` なら、重複のチェックをせず要素は追加される。

注意

この関数が使う比較は、ポインタ比較であって、内容比較ではない。◆

SetContains

`SetContains(`*array*, *item*`)`

array 配列。

item 配列内に存在するかもしれない要素。

item が、配列要素のどれかと等しいかどうかをサーチする。どこかでマッチが起これば、この関数はマッチした要素への添え字を返す。見つからなければ `nil` を返す。

注意

この関数が使う比較は、ポインタ比較であって、内容比較ではない。◆

SetDifference

`SetDifference(`*array1*, *array2*`)`

array1 にあって、*array2* にない配列要素を全て含む配列を返す。

array1 配列。

array2 配列。

もし *array1* が `nil` なら、`nil` が返る。

注意

この関数が使う比較は、ポインタ比較であって、内容比較ではない。◆

SetLength

`SetLength (`*array*, *length*`)`

配列の長さを変更する。

array 配列。

length 整数。

この関数は配列のサイズを増やしたり減らしたりするのに便利である。配列サイズを増やした場合、新しい要素は `nil` に初期化される。

例:

```
myArray := [123, 456, "I want chopstix"]
#1634 [123, 456, "I want chopstix", 789]
setLength (myArray, 4)
#1634 [123, 456, "I want chopstix", NIL]
myArray [3] := 789
#3156 789
myArray
#1634 [123, 456, "I want chopstix", 789]
```

SetOverlaps

`SetOverlaps(`*array1*`, `*array2*`)`

array1 内の各要素と、*array2* 内の各要素を比較し、*array2* 内の要素と等しい *array1* 内の最初の要素の添え字を返す。等しい要素が見つからない場合、`nil` が返る。

array1 配列。

array2 配列。

注意

この関数が使う比較は、ポインタ比較であって、内容比較ではない。◆

SetRemove

SetRemove （*array*， *value*）

SetRemove は、*array* から指定された要素を取り除き、変更された *array* を返す。削除された要素以降のものが左に一つシフトされ、配列の長さが変わる。指定された要素が配列内に見つからない場合、nil を返す。

array	SetRemove が要素を削除する配列。
value	*array* 中の、削除する要素を指定する値。

注意

この関数が使う比較は、ポインタ比較であって、内容比較ではない。◆

SetUnion

SetUnion(*array1*， *array2*， *uniqueFlag*)

array1 と *array2* に含まれる全ての要素を含む配列を返す。

array1	配列。
array2	配列。
uniqueFlag	非 nil 値を指定すると、SetUnion が返す配列内には重複値が含まれない。このフラグを nil にすると、要素が重複していても、両配列内の全ての要素が含まれる。

両方の配列が nil なら、空の配列が戻る。

SetUnion は、重複値を取り除くことが出来る。そうした特性が必要でないなら、ArrayMunger(Page 6-28)を使用して、二つの配列を連結できる。

注意

この関数が使う比較は、ポインタ比較であって、内容比較ではない。◆

Sort

`Sort(`*array*`,` *test*`,` *key*`)`

配列をソートし、ソート後それを返す。ソートは配列を破壊する。だから、*array* パラメタで渡した配列は変更される。また、ソートの方法は安定的なものではないので、等しいキーを持つ要素の並びは、ソート前とソート後では必ずしも同じではない。

array 配列。

test ソート順の指定。A と B 二つのパラメタを取る関数オブジェクトを指定しても良い。この関数は A と B を比較して、A の方が小さければ負の整数、A, B 等しければ 0、A の方が大きければ正の整数を返すものである。

スピードを向上するには、*test* パラメタに以下のシンボルを与えると良い:

`'|<|`	数値による昇順ソート。
`'|>|`	数値による降順ソート。
`'|str<|`	文字列による昇順ソート。
`'|str>|`	文字列による降順ソート。

key 各配列要素内のソートキーを指定する。`nil` を指定すると、配列要素を直接そのまま使用する。配列要素がフレームや配列である場合、パス式を指定すれば、ソートキー取得のためにパス式が適用される。また、一つのパラメタを取り、ソートキーを返す関数を指定しても良い。

次の例では、`myArray` を要素の `timestamp` スロットの値で数値順で昇順にソートしている:

```
Sort(myArray, '|<|, 'timestamp)
```

次の例では、myArray を、名と姓を結合した文字列で、降順にソートしている:

```
Sort(myArray, '|str>|, func (e) e.first && e.last)
```

StableSort

StableSort(*array*, *test*, *key*)

等しいキーを持つ配列要素の並びを、ソート前と同じになるように保ちながら配列をソートする。

array ソートで変更される配列。

test ソート方法の指定。page 6-39 参照。

key 各配列要素のキーを定義する。nil, パス式、一つのパラメタを取る関数を指定できる。Page 6-39 参照。

このソートはワーク用メモリを必要とするので、非常に大きい配列をソートする場合や、残りメモリが少ない状況下には適していない。

ソートされた配列用の関数

このセクションでは、ソートされた配列を操作するための新しい関数について述べる。これらは、バイナリサーチアルゴリズムに基づいているので、関数名は B で始まっている。

重要

関数に渡す配列は、すでにソートされていなければならない。さもなくば、結果は不定となる。配列をソートするには、Sort, InsertionSort, StableSort 等を使う。◆

これらのソートされた配列用の関数は、それぞれ *test* と *key* パラメタを取ることで、異なるデータ構造にも対応している。一般的に、これらの関数は配列

内の要素をサーチしたり、繰り返したりする。配列内の各要素が検証されるとき、引数 *key* が配列要素からの値の抽出に使われ、それはキーと呼ばれる。そして、*key* は引数 *test* によって指定された形態で取り扱われる。

以下に、これらのパラメタを説明する:

test 配列のソート順序を示す。配列のソート方法を示するため、以下のシンボルの内一つを指定する:

`'|<|` 数値による昇順ソート。

`'|>|` 数値による降順ソート。

`'|str<|` 文字列による昇順ソート。

`'|str>|` 文字列による降順ソート。

`'|sym<|` シンボルによる昇順ソート。シンボル名の字句的な比較を用いる。

`'|sym>|` シンボルによる降順ソート。シンボル名の字句的な比較を用いる。

また、非標準的なソート状況下に対応するため、二つのキーをパラメタに取り、それらがどう並んでいるかを返す関数オブジェクトを指定することが出来る。この関数は要素間のソート関係を特定するため、ソートされた配列用関数のいくつかから呼ばれる。この関数にはA, B二つのパラメタが渡され、戻り値としてAが小さければ負の整数、A, Bが等しければ0、Aが大きければ正の整数を返さないといけない。ただ、関数オブジェクトをテストパラメタに渡すと、定義済みシンボルを渡した場合よりもパフォーマンスが格段に落ちることを注意すること。

key 各配列要素のキーを指定する。`nil` を指定すると配列要素を直接使用する。配列要素がフレームや配列である場

合、パス式を指定することで、*key* の取り出しに際して、配列の各要素にパス式が適用される。一つのパラメタ(配列要素)を引数に取り、*key* を返す関数を指定しても良い。

BDelete

`BDelete(`*array*`,` *item*`,` *test*`,` *key*`,` *count*`)`

ソートされた配列から、指定された要素を削除する。

この関数は削除された要素の数を返す。

array 変更される配列。

item 検索

されるキーの値。このキーを持つ要素が削除される。

test 配列のソート順を指定する。page 6-39 の説明を参照。

key 各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

count 削除される要素の最大数。`nil` を指定すると、マッチする要素が全て削除されることを意味する。

BDifference

`BDifference(`*array1*`,` *array2*`,` *test*`,` *key*`)`

array1 の要素のうち、*array2* に含まれていないものだけを含む、ソートされた配列を返す。

array1 最初の配列。配列は変更される。

array2 二番目の配列。配列は変更されない。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

BFetch

`BFetch(`*array*, *item*, *test*, *key*`)`

ソートされた配列内の要素を検索するのにバイナリサーチアルゴリズムを用いる。最も左(0, 1, 2, 3 と左から右に並んでいると仮定した場合)で見つかった要素が返される。要素が見つからなければ `nil` が返る。

array サーチされる配列。

item サーチされるキーの値。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

この関数は、`BFind` と似た働きをするが、`BFind` は見つかった要素の添え字を返すという点でこの関数とは異なる。

BFetchRight

`BFetchRight(`*array*, *item*, *test*, *key*`)`

ソートされた配列内の要素を検索するのにバイナリサーチアルゴリズムを用いる。最も右(0, 1, 2, 3 と左から右に並んでいると仮定した場合)で見つかった

要素が返される。要素が見つからなければ `nil` が返る。

array サーチされる配列。

item サーチされるキーの値。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

この関数は、`BFindRight` と似た働きをするが、`BFindRight` は見つかった要素の添え字を返すという点でこの関数とは異なる。

BFind

`BFind(`*array*`,` *item*`,` *test*`,` *key*`)`

ソートされた配列内の要素を検索するのにバイナリサーチアルゴリズムを用いる。最も左で見つかった要素の添え字が返される。要素が見つからなければ `nil` が返る。

array サーチされる配列。

item サーチされるキーの値。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

この関数は、`BFetch` と似た働きをするが、`BFetch` は見つかった要素を返すという点でこの関数とは異なる。

BFindRight

`BFindRight(`*array*, *item*, *test*, *key*`)`

ソートされた配列内の要素を検索するのにバイナリサーチアルゴリズムを用いる。最も右で見つかった要素の添え字が返される。要素が見つからなければ`nil`が返る。

array	サーチされる配列。
item	サーチされるキーの値。
test	配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。
key	各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

この関数は、`BFetchRight`と似た働きをするが、`BFetchRight`は見つかった要素を返すという点でこの関数とは異なる。

BInsert

`BInsert(`*array, element, test, key, uniqueOnly*`)`

ソートされた配列内の正しい位置に要素を挿入する。キーの値が等しい要素の場合、挿入は既存同一要素の左側に挿入される。

array	変更される配列。
element	挿入される新しい要素。*key* パラメタが、キーの値の抽出に使用されることに注意。
test	配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。
key	各要素のキーを定義する。`nil`, パス式、一つのパラメタ

を取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

uniqueOnly 非 `nil` の値を指定すると、既に配列内に同じ値が存在する場合は挿入されない。`'returnElt` を指定しても同じ意味であるが、この場合関数は、配列要素を返す。これは、挿入された要素を返すか、あるいはマッチした要素が配列内に見つかると、見つかった方の要素が返る。このやりかたは、メモリ節約のために、既に存在するオブジェクトを再使用するとか、ポインタの同一性を確認するとか言った用途に有用である。

`nil` を指定すると、既に同じキーを持つ要素が配列に含まれていても、要素が挿入される事を意味する。この場合、新しい要素は、同一キーを持つ要素の左側に挿入される。

この関数の戻り値は次の 3 種類になる:

- `nil` を返して、要素が挿入されなかったことを示す。
- 整数を返して、要素が挿入された位置を表す添え字を示す。
- (ユニーク(一意)なキーを持っている)挿入された配列要素を返すか、挿入したかった要素と同一のキーを持つ要素が既に配列内にあったときはその見つかった要素を返す。このタイプの戻り値は、`uniqueOnly` に `'returnElt` を指定したときだけ起こる。

以下に、この関数の使い方を示す。ここでは `uniqueOnly` に `'returnElt` をセットして、ポインタの同一性を確認している:

```
// :GetStr() returns a string input by the user
bodyColor :=
BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
interiorColor:=
BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
if bodyColor = interiorColor then Print("bad idea");
```

GetString が colorList に既にある文字列を返した場合、このコードはオリジナルの文字列が再使用できることを示す。だから、= 演算子を等値チェックに使っても上手く働くのである。これはまた、重複した文字列がガベージコレクションの対象に出来ることを意味する。なぜなら、そこへの参照がどこにも残らないからである。

BInsertRight

BInsertRight(*array, element, test, key, uniqueOnly*)

ソートされた配列内の正しい位置に要素を挿入する。キーの値が等しい要素の場合、挿入は既存同一要素の右側に挿入される。この関数は、挿入された要素への添え字を返すかまたは、挿入が起こらなかった場合は nil を返す。

array 変更される配列。

element 挿入される新しい要素。*key* パラメタが、キーの値の抽出に使用されることに注意。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。nil, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

uniqueOnly 論理値。非 nil の値を指定すると、既に配列内に同じ値が存在する場合は挿入されない。nil を指定すると、既に同じキーを持つ要素が配列に含まれていても、要素が挿入される事を意味する。後者の場合、新しい要素は、同一キーを持つ要素の右側に挿入される。

BIntersect

`BIntersect(`*array1*, *array2*, *test*, *key*, *uniqueOnly*`)`

二つの指定された配列のどちらにも含まれる要素からなる新しいソート済み配列を返す。

array1 最初の配列。これは変更されない。

array2 二番目の配列。これは変更されない。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。`nil`, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

uniqueOnly 論理値。非 `nil` を指定すると、戻り値となる配列内に重複値は許されない。 これは、*array1* と *array2* が、いずれも重複する要素を持っていない場合に上手く働く。

`nil` を指定すると、戻り値となる配列内に重複する要素が入っていても良いことを示す。結果配列は、各合致要素について最低２つの要素を持つことが保証される。合致要素は値の等しい要素を発見するからである。

等しい要素が結果配列の中に見つかると、以下のルールで並べられる: 同一ソースからの等しい値の要素は元の配列の順序で並べられ、*array1* の等しい要素と、*array2* の等しい要素では、*array1* の要素の方が前に来る。

BMerge

`BMerge(`*array1*, *array2*, *test*, *key*, *uniqueOnly*`)`

二つのソートされた配列を、一つのソートされた配列にマージし、それを返す。

array1 最初の配列。これは変更されない。

array2 二番目の配列。これは変更されない。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。nil, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

uniqueOnly 論理値。非 `nil` を指定すると、戻り値となる配列内に重複値は許されない。 これは、*array1* と *array2* それぞれが、いずれも重複する要素を持っていない場合に上手く働く。

`nil` を指定すると、戻り値となる配列内に重複する要素が入っていても良いことを示す。

等しい要素が結果配列の中に見つかると、以下のルールで並べられる: 同一ソースからの等しい値の要素は元の配列の順序で並べられ、*array1* の等しい要素と、*array2* の等しい要素では、*array1* の要素の方が前に来る。

BSearchLeft

`BSearchLeft(`*array*`,` *item*`,` *test*`,` *key*`)`

ソートされた配列から要素を見つけるのにバイナリサーチを使う。指定された *item* 以上の値を持つ要素の中で、最も小さく、左側にある値への添え字が戻る。*item* が、全要素より大きい場合は、`Length(`*array*`)` の値が返る。

array 検索される配列。

item 検索される値。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。nil, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

以下に、この関数の使い方を示す:

```
// Extract all elements between "F" and "Na"
array := ["Ag","C","F","Fe","Hg","K","N","Na","Ni","Pu","Zn"];
pos1   := Min(Length(array)-1, BSearchLeft(array,"F",'|str<|,
nil));
pos2   := Max(0,BSearchRight(array,"Na",'|str<|,nil));
ArrayMunger([],0,nil,array, pos1, pos2-pos1+1);
#4423631  ["F", "Fe", "Hg", "K", "N", "Na"]
```

BSearchRight

BSearchRight(*array*, *item*, *test*, *key*)

ソートされた配列から要素を見つけるのにバイナリサーチを使う。指定された *item* 以下の値を持つ要素の中で、最も大きく、右側にある値への添え字が戻る。*item* が、全要素より小さい場合は、-1 が返る。

array 検索される配列。

item 検索される値。

test 配列のソート順を指定する。*test* パラメタについては page 6-39 を参照。

key 各要素のキーを定義する。nil, パス式、一つのパラメタを取る関数が指定できる。*key* に関する説明は、page 6-39 の説明を参照。

この関数の使い方のサンプルは、`BSearchLeft` を参照。

整数演算関数

これらの数値演算関数は、整数を操作し、整数を返す。(いくつかの浮動小数関数も、整数を操作できる)

Abs

`Abs(`*x*`)`

整数または実数の絶対値を返す。

x	整数または実数。

Ceiling

`Ceiling(`*x*`)`

指定された実数より小さくない最小の整数を返す。(実数の整数への切り上げ)

x	実数。

Floor

`Floor(`*x*`)`

指定された実数より大きくない最大の整数を返す。(実数の整数への切り捨て)

x	実数。

Max

`Max(` *a*`,` *b* `)`

二つの整数 *a* と *b* の内、大きい方を返す。

a 整数。

b 整数。

Min

`Min(` *a*, *b* `)`

二つの整数 *a* と *b* の内、小さい方を返す。

a 整数。

b 整数。

Random

`Random（`*low*，*high*`）`

二つの整数 *low, high* 区間のランダムな整数を返す。*low, high* の値も、区間に含まれる。

low 整数。

high 整数。

例:

```
random (0, 100)
#120      72
```

Real

`Real(`*x*`)`

指定された整数を実数に変換する。

x 整数。

SetRandomSeed

`SetRandomSeed` (*seedNumber*)

指定した値で、乱数生成の種を作る。

seedNumber 整数。

同一の数値で種を作ると、種を作る度に、`Random` 関数による乱数生成が、同一パターンの乱数を返す。種に 0 を使うと、乱数の代わりに 0 を返すので、使わないこと。

注意

これは、Newton 上での唯一の乱数生成機構である。だから、他の関数がこれを呼び出すと、自分の関数が一貫した乱数列が得られないかもしれない。◆

浮動小数演算関数

浮動小数演算関数に関して説明する。

NewtonScirpt の浮動小数演算システムは、IEEE によって採用された標準 754 と 854 に準拠している。IEEE 標準の算術についてより詳しくは、*Inside Mac* の *PowerPC Numericals* の巻と、*Apple Numerics Manual Second Edition* を参照されたい。これらの本は SANE(Standart Apple Numeric Environment)について説明している。NewtonScript 環境は、SANE の多くの機能をサポートしている。

NewtonScript の浮動小数（実数とも言う）は、IEEE 標準の倍精度に対応している。数値システムは、以下の値の表現をサポートしている:

- 普通の数値 — それは $1.8x10^{308}$ から $2.2x10^{-308}$ までの範囲で、16 桁程度の精度を持つ。
- 小さい数値 — それは、$2.2x10^{-308}$ から $4.0x10^{-324}$ の範囲で 16 桁よりは 1 桁程度精度が落ちる。
- 符号付きの 0— +0 と –0 である。これらは比較すれば等しいがその動作は異なる。例えば、非 0 の値で除算した場合などである。
- 符号付きの無限 — これは `+INF` と `-INF` である。表現するには大きすぎる結果や、非 0 の分子を分母 0 で割った結果などを表現する。
- 数値でない(Not-a-Number)シンボルあるいは NaN— これは、行方不明あるいは初期化されていないデータ、あるいは実数システムでは意味が無い $\sqrt{-3}$ のような演算子の結果である。

アプリケーションのある局面では、符号付きの 0 や、無限といった概念は数学的制限の見地からは有用だと思うだろう。例えば、+0 と –0 は、比較すれば等しいはずだが、$\lim_{x \to 0-} f(x) \neq \lim_{x \to 0+} f(x)$ のようなケースもあり、これを利用することは有用であろう。同様に、g(+INF) を $\lim_{y \to \infty} g(y)$ と解釈することも有用であろう。

このセクションの関数は、IEEE 標準で明らかにされた算術操作のモデルに従っている。具体的には、それらは結果が数値システムで正確に表現できる場合は正確な結果を生成し、そうでない場合は数学的に正しい値に最も近い数値で表現できる結果を生成する。IEEE 標準は一つ以上の式において、操作の結果が数学的結果と違っていた場合や、実数では定義されていない結果を生じた場合、一つ以上の例外が発行されると明記している。起こり得る例外は次の通り:

- inexact— 結果が算術結果に対して丸めまたは他の変換を受けた
- underflow—0 でない結果が 0 または小さい数値として表現するには小さすぎ、通常の数より精度の落ちる丸めを受けた
- overflow— 結果が通常の数値として表現するには大きすぎる
- Divide by Zero— 非 0 の値を 0 で除算したら、引数の符号に応じて、+INF あるいは –INF を生じた
- invalid— 結果が数学的に定義されていない。0/0 の場合のように。

浮動小数の例外についてより詳しくは page 6-71 の「浮動小数環境の管理」を参照されたい。

IEEE 標準と SANE の一つの機能は、結果が正確に表現できない場合の丸め方向の選択である。NewtonScript システムでは、丸めは常にもっとも近い表現可能な数値に行われる(最低位ビットが 0 になるような値へと)。IEEE 標準はまた、0 方向、`+INF` 方向、`-INF` 方向への丸めも明記している。しかし、標準は、丸めの方向は、浮動小数環境の状態変数によって特定されるかのように書かれている。(「浮動小数演算環境の操作」参照)だが、NewtonScript が使用しているプロセッサである ARM ファミリー上では、丸め方向は命令毎の原則に従って特定される。

Acos

`Acos(x)`

コサインの逆関数をラジアンで返す。`Acos` は $x < -1$ あるいは $x > 1$ の時に invalid を発行する。1 以外の全ての数に、inexact を発行する。`Acos` はゼロから π までの値を返す。

x 整数または実数。

Acosh

`Acosh(x)`

x のハイパーボリックコサインの逆関数を返す。`Acosh` は $x < 1$ で invalid を発行する。これは 1 以外の全ての値で inexact を発行する。`Acosh(+INF)` は `+INF` を返すが、オーバーフローする事はない。有限の実数の最大値はだいたい 710 である。

x 整数または実数。

Asin

`Asin(x)`

x のサインの逆関数をラジアンで返す。`Asin` は、$x < -1$ または $x > 1$ で invalid を発行する。0 以外の全ての値で inexact を発行する。また、ゼロに近い有限の x 全てに対して、underflow を発行する。`Asin` は $-\pi/2$ と $\pi/2$ の間の値を返す。

x 整数または実数。

Asinh

`Asinh(x)`

x のハイパーボリックサインの逆関数を返す。`Asinh` はゼロ以外の全ての値で inexact を発行する。`Asinh(-INF)` は `-INF` を返し、`Asinh(+INF)` は、`+INF` を返す。Asinh はゼロに近い *x* で underflow を発行する。

x 整数または実数。

Atan

`Atan(x)`

x のタンジェントの逆関数をラジアンで返す。0 以外の全ての値で inexact を発行する。`Atan(-INF)` は $-\pi / 2$ を、`Atan(+INF)` は $\pi / 2$ を返す。`Atan` は、$-\pi / 2$ から $\pi / 2$ の値を返す。これは、ゼロでない *x* 全てについて inexact を発行する。

x 整数または実数。

Atan2

`Atan2(x,y)`

x / y のタンジェントの逆関数をラジアンで返す。`Atan2` は *x* と *y* の符号を、結果の 4 分円弧を特定するために使用する。$-\pi$ から π までの値を返す。`Atan` は、この関数の特殊なケースである。

x 整数または実数。

y 整数または実数。

Atanh

`Atanh(x)`

x のハイパーボリックタンジェントの逆関数を返す。`Atanh` は、$x < -1$ または $x > 1$ で、invalid を発行する。これはゼロ以外の全ての正しい値について

inexact を発行する。また、ゼロに近い有限の値で underflow を発行する。
`Atanh(-1.0)` は `-INF` を返し、`Atanh(+1.0)` は `+INF` を返す。

x 整数または実数。

CopySign

`CopySign(`*x*`,`*y*`)`

x の絶対値と、*y* の符号から値を作って返す。

x 整数または実数。

y 整数または実数。

注意

パラメタの並びは、IEEE754 標準の推奨に適合している。これは
SANE の `copysign` 関数とは逆である ◆

Cos

`Cos(`*x*`)`

x ラジアンのコサインを返す。`Cos` は、0 以外の全ての有限の値で inexact を発行する。これは、2π の周期を持つ。`Cos` は *x* が無限の場合 invalid を発行する。

x 整数または実数。

Cosh

`Cosh(`*x*`)`

x のハイパーボリックコサインを返す。`Cosh` は 0 以外の全ての有限の値で inexact を発行する。`Cosh(-INF)` と `Cosh(+INF)` は `+INF` を返す。`Cosh` は

巨大な値に対して overflow を発行する。

x 整数または実数。

Erf

`Erf(x)`

$erf(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$ を返す。これは x の誤差関数である。`Erf` は、0 以外の全ての引数に inexact を発行する。これは、0 に近い引数に対して underflow を発行する。`Erf(-INF)` は –1 を返し、`Erf(+INF)` は 1 を返す。

x 整数または実数。

数学的には、`Erf(x)` と `Erfc(x)` の和は 1 になる。しかし、roundoff または underflow が顕著に影響した場合、その関係は維持されないことがある。

Erfc

`Erfc(x)`

$erfc(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt$ を返す。これは、x の誤差関数の補数である。`Erfc` は 0 以外の全ての値で inexact を発行する。`Erfc(-INF)` は 2 を返し、`Erfc(+INF)` は +0 を返す。

x 整数または実数。

Exp

`Exp(x)`

e^x を返す。これは、x の指数である。`Exp` は、非 0 の有限の引数に対して

inexact となる。`Exp(-INF)` は +0 を返し、`Exp(+INF)` は `+INF` を返す。`Exp` は overflow を巨大な有限の *x* に対して発行する。また、巨大な負の有限の値に対して underflow を発行する。

x 整数または実数。

Expm1

`Expm1(`*x*`)`

$e^x - 1$ を返す。*x* の指数より 1 少ない値を返す。`Expm1` は *x* がゼロに近いときや、差が 0 に近いときの精度のロスを回避する。`Expm1` は全ての有限の 0 でない値に対して inexact である。`Expm1(-INF)` は –1 を、`Expm1(+INF)` は `+INF` を返す。`Expm1` は、巨大な有限の正の数には overflow を発行し、0 に近い *x* には underflow を発行する。

x 整数または実数。

Fabs

`Fabs(`*x*`)`

x の絶対値を返す。例外は発行しない。

x 整数または実数。

FDim

`FDim(`*x*`,`*y*`)`

パラメタ間の正の値の差を返す。

- $x > y$ なら、$x - y$,
- そうでなくて、$x <= y$ なら、+0
- そうでなくて、*x* が NaN なら *x*

■ そうでなくて *y* が NaN なら *y*

x 整数または実数。

y 整数または実数。

FMax

`FMax(x,y)`

二つの値のうち、大きい方を返す。NaN パラメタは紛失データとして扱われる:

■ パラメタのどちらかが NaN で、片方が数値なら、数値の方が返る。
■ どちらも NaN なら、最初のパラメタが返る。
これは FORTRAN の MAX 関数に対応している。

x 整数または実数。

y 整数または実数。

FMin

`FMin(x,y)`

二つのパラメタの内、小さい方を返す。NaN パラメタは、紛失データとして扱われる:

・ パラメタの一つが NaN で、もう一方が数値なら、数値の方が返る。
・ 両方とも NaN なら、最初のパラメタが返る。
これは FORTRAN の MIN 関数に対応している。

x 整数または実数。

y 整数または実数。

Fmod

`Fmod(`*x*`,`*y*`)`

x / *y* の整数商に対する余りを実数で返す。つまり、`Fmod` は、*x* – *y* * `Trunc(`*x* / *y*`)` を返す。

x 整数または実数。

y 整数または実数。

Gamma

`Gamma(`*x*`)`

Γ(*x*) を返す。ガンマ関数は *x* に適用される。`Gamma` は整数でない全ての x に対して正確ではない。負の整数を引数にすると、invalid を発行する。正の整数 *p* に対して、`Gamma(`*p*`)` は、(*p*–1)!を返す。0!は 1 と定義される。`Gamma(+INF)` は `+INF` を返す。`Gamma` は overflow も発行しうる。

x 整数または実数。

Hypot

`Hypot(`*x*`,`*y*`)`

x と *y* の 2 乗を合計したものの平方根を返す。引数が非常に大きかったり小さかったりしても、結果が範囲内に収まるような場合は、overflow や underflow の危険を回避する。

x 整数または実数。

y 整数または実数。

IsFinite

`IsFinite(x)`

x が有限なら `true` を、無限なら `nil` を返す。

x 整数または実数。

IsNaN

`IsNaN(x)`

x が NaN なら `true` を、*x* が数値なら `nil` を返す。

x 整数または実数。

注意

“*x* が NaN である” というのと、“*x* は数値でない” というのは同じ事ではない。NaN は、数値フォーマットを持つ非数値の値である。一方、`"foo"` は数値ではない、なぜならそれは数値オブジェクトではないからだ ◆

IsNormal

`IsNormal(x)`

x が通常の数値なら `true` を返し、0, 小さい数値、無限、NaN なら `nil` を返す。

x 整数または実数。

LessEqualOrGreater

`LessEqualOrGreater(x, y)`

x と *y* がどちらも NaN でなく、二つの引数が順序付けられていれば、`true` を

返す。さもなくば`nil`を返す。

x 整数または実数。

y 整数または実数。

LessOrGreater

`LessOrGreater(x, y)`

$x < y$ または $x > y$ なら`true`を返す。さもなくば`nil`を返す。

x 整数または実数。

y 整数または実数。

LGamma

`LGamma(x)`

$\Gamma(x)$の、自然対数を返す。ガンマ関数が x に適用される。`LGamma`は全ての正の x に対して inexact を発行する。x が 0 以下なら invalid を発行する。`LGamma(+INF)`は`+INF`を返す。

x 整数または実数。

Log

`Log(x)`

x の自然対数を返す。`Log`は 1 以外の正の有限の数値に対しては inexact を発行する。`Log(0.0)`は`-INF`を返し、divide by zero を発行する。`Log(+INF)`は`+INF`を返す。`Log`は $x < 0$ の時 invalid を発行する。

x 整数または実数。

Logb

`Logb(x)`

x が有限で非 0 の時、$1 \leq |x| * 2^{-k} < 2$ となる積分値 k を返す。`Logb(0.0)` は `-INF` を返し、divide by zero を返す。`Logb(-INF)` と `Logb(+INF)` は `+INF` を返す。

x 整数または実数。

Log1p

`Log1p(x)`

$1 + x$ の自然対数を返す。–1 以上の全ての引き数に精度を出す一方、`Log1p` は x が 0 に近い場合でも精度を維持する。`Log(1.0 + x)` の場合は、単に 1.0 と x を加算したものを使って計算されるだけである。

`Log1p` は、0 以外の –1 より大きい全ての有限の値に対して、inexact を発行する。–1 より小さい全ての x に対して、invalid を発行し、0 に近い x に対して、underflow を発行する。

`Log1p(-INF)` は `-INF` を返し、divide by zero を発行する。`Log1p(+INF)` は `+INF` を返す。

x 整数または実数。

Log10

`Log10(x)`

10 を底とした x の対数を返す。$\log_{10} = \log(x) / \log(10)$ という数学的な関係のため、`Log10` は、計算の特性を `Log` と共有している。

x 整数または実数。

NearbyInt

`NearbyInt(x)`

x もっとも近い整数に四捨五入された整数を返す。`NearbyInt` は inexact 例外を発行しないと言う点で `Rint` とは異なる。

x 整数または実数。

注意

`NearbyInt` は常にもっとも近いものに四捨五入する。◆

NextAfterD

`NextAfterD(x,y)`

x から y への方向で、次に表現できる数値を返す。

x と y が等しければ結果は x である。引数がどちらも NaN なら、どちらかの引数が返される。x が有限で、結果が無限なら、`NextAfterD` は、overflow を発行する。結果が 0 または小さい数値なら、`NextAfterD` は underflow を発行する。

x 整数または実数。

y 整数または実数。

Pow

`Pow(x,y)`

x^y を返す。$x<0$ なら、y が整数値でない場合 invalid が発行される。これは、inexact, overflow, underflow, invalid を発行しうる。

x 整数または実数。

y 整数または実数。

RandomX

`RandomX(`*x*`)`

乱数の種 *x* に基づく、2 要素の配列を返す。最初の要素は SANE `randomx` 関数が返す疑似乱数の値で、二番目の要素は `randomx` が返す新しい種である。結果は 0 から $2^{31} - 1$ の間の整数。

x 整数または実数。

Remainder

`Remainder(`*x*`,`*y*`)`

x – *n***y* の正確な差を返す。*n* は数学的な整数(NewtonScript の整数とは違い、何千ビットもの幅を持っている)で、*x* / *y* の結果をもっとも近いところに丸めたものである。結果の大きさは、*y* の半分の大きさより大きくない。結果が 0 の場合、*x* の符号を持つ。`Remainder` は、*y* がゼロまたは *x* が無限の時 invalid を発行する。overflow, underflow, inexact は発行されない。

x 整数または実数。

y 整数または実数。

RemQuo

`RemQuo(`*x*`,`*y*`)`

2 要素の配列を返す。最初の要素は `Remainder(`*x*`, `*y*`)` で、二番目は *x* / *y* のもっとも近い整数に丸められた商の下位 7 ビットに商の符号を付けたものである。

x 整数または実数。

y 整数または実数。

Rint

`Rint(x)`

`Nearbyint` と似ているが、結果が *x* と違うときに inexact 例外を発行するところが違う。

x 整数または実数。

RintToL

`RintToL(x)`

x を整数(実数型であるが)に丸め、整数に変換した値を整数で返す。`RintToL` は結果が *x* と違う場合に inexact を発行する。*x* の丸められた値が正確に整数として表現できない場合、invalid を発行し、未定義の値を返す。

x 整数または実数。

注意

`RintToL` は常にもっとも近いものに丸められる◆

Round

`Round(x)`

x に 1/2 を足して、0 方向にもっとも近い整数に丸めたものを整数化した実数を返す。(四捨五入のこと)これは、結果が *x* と違うときに inexact を返す。

x 整数または実数。

Scalb

`Scalb(x, k)`

$x*2^k$ を返す。`Scalb` は、2^k の明示的な演算を行わないので、2^k がレンジを越

えているが、結果はレンジ内に収まると言ったケースでの計算中の underflow, overflow, inexact 発行を回避する。`Scalb` と `Logb` は、有限の 0 でない *x* について、$1 \leq$ `Scalb(`*x*`, RintToL(-Logb(`*x*`))` < 2 の関係となる。

x 整数または実数。

k 整数。

SignBit

`SignBit(`*x*`)`

x の符号が負なら非 0 の整数を返し、さもなくば(*x* の符号が正なら)整数の 0 を返す。

x 整数または実数。

Signum

`Signum(`*x*`)`

$x < 0$ なら整数の –1 を、$x = 0$ なら 0 を $x > 0$ なら 1 を返す。*x* が整数なら、`Signum` は整数を返し、*x* が実数なら実数を返す。*x* が整数でも実数でもない場合は、`kFramesErrNotANumber` 例外を発行する。

x 整数または実数。

Sin

`Sin(`*x*`)`

x ラジアンのサインを返す。`Sin` は 0 以外の全ての有限の値に関して inexact を発行する。これは 2π の周期を持つ。`Sin` は *x* が無限なら invalid を、*x* が 0 に近ければ underflow を発行する。

x 整数または実数。

Sinh

`Sinh(x)`

x のハイパーボリックサインを返す。`Sinh` は 0 以外の全ての有限値の引数に inexact を発行する。`Sinh(-INF)` は `-INF` を、`Sinh(+INF)` は `+INF` を返す。`Sinh` は大きい有限値に overflow を、0 に近い値に underflow を発行する。

x	整数または実数。

Sqrt

`Sqrt(x)`

x の平方根を返す。$x < 0$ なら invalid を発行する。正の x に対しては inexact を発行することがある。

x	整数または実数。

Tan

`Tan(x)`

x ラジアンのタンジェントを返す。`Tan` は、0 以外の全ての有限値に対して inexact を発行する。これは、π の周期を持つ。`Tan` は x が無限なら invalid を、x が 0 に近ければ underflow を発行する。

x	整数または実数。

Tanh

`Tanh(x)`

x のハイパーボリックタンジェントを返す。`Tanh` は 0 以外の全ての有限値に対して inexact を発行する。`Tanh(-INF)` は –1 を返し、`Tanh(+INF)` は `+1` を返す。`Tanh` は大きい有限値に対して overflow を発行し、0 に近い値には

underflow を発行する。

x 整数または実数。

Trunc

`Trunc(`*x*`)`

x の大きさを越えない、もっとも近い整数を実数型で返す。

x 整数または実数。

Unordered

`Unordered(`*x*`, `*y*`)`

$x < y$, $x = 0$, $x > y$ のどれも満たされない(*x*, *y* のどちらかまたは両方が NaN の)場合、`true` を返す。*x*, *y* 共に NaN でなく、上記 3 条件のどれかに当てはまる場合、`nil` を返す。

x 整数または実数。

y 整数または実数。

UnorderedGreaterOrEqual

`UnorderedGreaterOrEqual(`*x*`, `*y*`)`

$x \geq 0$ あるいは、順序が付けられない(*x*, *y* のどちらかまたは両方が NaN の)場合 `true` を返す。さもなくば `nil` を返す。

x 整数または実数。

y 整数または実数。

UnorderedLessOrEqual

`UnorderedLessOrEqual(x, y)`

$x \leq y$ または x と y が順序付けられない(x, y のどちらかまたは両方が NaN の)場合、`true` を返す。さもなくば `nil` を返す。

x 整数または実数。

y 整数または実数。

UnorderedOrEqual

`UnorderedOrEqual(x, y)`

$x = y$ または x と y が順序付けられない(x, y のどちらかまたは両方が NaN の)場合、`true` を返す。さもなくば `nil` を返す。

x 整数または実数。

y 整数または実数。

UnorderedOrGreater

`UnorderedOrGreater(x, y)`

$x > y$ または x と y が順序付けられない(x, y のどちらかまたは両方が NaN の)場合、`true` を返す。さもなくば `nil` を返す。

x 整数または実数。

y 整数または実数。

UnorderedOrLess

`UnorderedOrLess(x, y)`

$x < y$ または x と y が順序付けられない(x, y のどちらかまたは両方が NaN の)場

合、true を返す。さもなくば nil を返す。

x 整数または実数。

y 整数または実数。

浮動小数環境の管理

浮動小数環境は Newton システムとその下層にあるプロセッサによって維持される状態変数の集合である。環境はどの種類の浮動小数例外が発生したかという情報を持っている。浮動小数例外は NewtonScritp の例外とは別のものである。浮動小数例外が発生した場合(例えば、巨大な数値を合計したら表現するには巨大すぎる値が生じて overflow が発行された場合など)に、システムは環境内に例外フラグを発行する。例外フラグはこのセクションの関数で、テストしたり、クリアしたり、発行したり出来る。例外がひとたび発行されると、例外フラグはこのセクションで述べる関数によってそれをクリアするまで残る。表 6-1 に示す定義済みの定数を使って、浮動小数例外をテストすることが出来る。

訳註: このセクションの引き数リストは何故か大幅に間違っている。本書をバージョンアップする際に、その細部を調査してご報告したいと思う。

表 6-1 浮動少数例外

定数	値	意味	
fe_Inexact	0x010	inexact	不正確
fe_DivByZero	0x002	divide-by-zero	0 での除算
fe_Underflow	0x008	underflow	アンダーフロー
fe_Overflow	0x004	overflow	オーバーフロー
fe_Invalid	0x001	invalid	無効
fe_All_Except	0x01F	all exceptions	全ての例外

定義済み定数を Bor(Bor(fe_Invalid, fe_DivByZero), fe_Overflow) の様にして、ビットごとの論理和を取ることによって得たフラグを使い、一度

の関数呼び出しで複数の例外を参照できる。

注意

浮動小数環境の在り方はその実装に依存する。環境を操作する関数と、そのコンポーネントは実装をあらわにせず動作する。特に、浮動小数例外は単一ビットとして実装されていてもされていなくても良い ◆

浮動小数環境を管理する関数は、ANSI C 言語の数値演算拡張の推奨に準拠している。C に関する推奨には、丸め方向の変更やテストをする関数が含まれる。丸めの方向は、多くのシステムにおいて環境によって特定されるが、ARM ファミリーのプロセッサに基づいている Newton システムは、丸め方向を命令 – 命令の原則に従って特定しているので、丸めは環境によって特定されることはない。

環境オブジェクトをパラメタに取る `FeSetEnv` や `FeUpdateEnv` 関数に、定義済み定数 `fe_Dfl_Env` を渡すことが出来る。`Fe_Dfl_Env` は、全ての例外フラグがクリアされているデフォルトの環境を意味する。

FeClearExcept

`FeClearExcept(`*excepts*`)`

excepts で示される浮動小数例外フラグをクリアする。

excepts 一つ以上の浮動小数例外をビット毎論理和した整数。

FeGetEnv

`FeGetEnv()`

現在の浮動小数環境を表すデータオブジェクトを返す。

FeGetExcept

`FeGetExcept(`*excepts*`)`

excepts で表される、現在の例外フラグの状態を表すデータオブジェクトを返す。

excepts 一つ以上の浮動小数例外をビット毎論理和した整数。

注意

例外フラグの表現は特に指定されていない ◆

FeHoldExcept

`FeHoldExcept()`

現在の浮動小数環境を表すデータオブジェクトを返し、例外フラグをクリアする。

FeRaiseExcept

`FeRaiseExcept(`*excepts*`)`

excepts で表される浮動小数例外フラグを発行する。

excepts 一つ以上の浮動小数例外をビット毎論理和した整数。

注意

浮動小数例外は、NewtonScript の一般の例外処理機構とは関係ないので、フラグの発行は、単に内部変数をセットするに過ぎない。フラグの発行は、制御フローを変更することはない ◆

FeSetEnv

`FeSetEnv(`*envObj*`)`

envObj で示される浮動小数環境をインストールする。

envObj 定義済み定数 `fe_Dfl_Env` あるいは、`FeGetEnv` や `FeHoldExcept` から返される値を指定する。

FeSetExcept

`FeSetExcept(`*flagObj*`,` *excepts*`)`

flagObj パラメタは、一つ以上の浮動小数例外フラグの、実装に依存した表現を含むオブジェクトである。*flagObj* は直前の `FeGetExcept` によってセットされているものでないといけない。`FeGetExcept` は、現在の環境を変更するので、*excepts* により示されるそれらの例外フラグそれぞれが、*flagObj* 内の対応する値とマッチする。

flagObj 一つ以上の浮動小数例外フラグの表現を含む(直前の `FeGetExcept` から返される)オブジェクト。

excepts 一つ以上の浮動小数例外をビット毎論理和した整数。

この関数は例外を発行しない。単にフラグの状態を変更するだけである。

FeTestExcept

`FeTestExcept(`*excepts*`)`

excepts によって表されるフラグの内、現在の環境に発行されている浮動小数例外のビット毎の論理和を返す。

excepts 一つ以上の浮動小数例外をビット毎論理和した整数。

FeUpdateEnv

`FeUpdateEnv(`*envObj*`)`

現在の例外フラグの状態を保存し、*envObj* で示される環境をインストールす

る。そして、保存された例外を再発行する。

envObj 定義済み定数 `fe_Dfl_Env` あるいは `FeGetEnv` または `FeHoldExcept` の両呼び出しから戻る値を指定する。

呼び出し元に対して、偽の例外を隠す関数を書くため、`FeUpdateEnv` と `FeHoldExcept` を使うことが出来る:

```
func() begin
       savedEnv := FeHoldExcept(); // clears flags
       result := ...;// ecomputation in which underflow and
                     // divide by zero are benign
       FeClearExcept(BOR(fe_Underflow, fe_DivByZero));
       FeUpdateEnv(savedEnv); // merge old flags with new
       return result
end
```

財務関数

以下の関数は財務計算を行う。

Annuity

`Annuity(r, n)`

財務計算式 $\frac{1-(1+r)^{-n}}{r}$ の値を返す。r を期間の利率とし、n を期間数とすると、p*`Annuity(r, n)` は、金額 p を n 周期に渡って投資した時の、現在価値となる。

`Annuity` は、財務的な意味の有無に関わらず r と n の範囲に対して正しく働く。

`Annuity` は、$r < -1$ で invalid を発行する。

$r = -1$ の時は、

- `Annuity(–1, n)` は、–1 を、$n < 0$ の時に返す
- `Annuity(–1, 0)` は、0 を返す
- `Annuity(–1, n)` は、`+INF` を返し、$n > 0$ の時 divide by zero を発行する

あるいは、$r > -1$ で、r が非 0 の時 `Annuity(r, 0)` は r を返す。それ以外の場合、`Annuity(0, n)` は、n を返す。`Annuity` は、全てのケースで inexact を発行し、overflow, underflow を発行しうる。

r 整数または実数。

n 整数または実数。

Compound

`Compound(r, n)`

財務計算式 $(1+r)^n$ の値を返す。*r* を期間の利率とし、*n* を期間数とすると、*p**`Compound(r, n)` は、基本金 *p* の *n* 周期後の価値である。

`Compound` は、財務的な意味の有無に関わらず *r* と *n* の範囲に対して正しく働く。

`Compound` は、$r < -1$ で invalid を発行する。

$r = -1$ の時は、

- `Compound(–1, n)` は、`+INF` を返し、$n < 0$ の時 divide by zero を発行する
- `Compound(–1, 0)` は、1 を返す
- `Compound(–1, n)` は、+0 を、$n > 0$ の時に返す

あるいは、$r > 0$ で、`Compound(r, 0)` は 1 を返す。`Compound(0, n)` は、*n* が無限の時、invalid を発行する。`Compound` は、inexact, overflow, underflow を発行しうる。

r 整数または実数。

n 整数または実数。

例外関数

これらの関数は例外を発行したり、処理したりするためのものである。例外処理に関しては、Chapter 3「制御構文」の後ろ半分の「例外処理」page 3-14 に詳しく出ている。システム例外に関しては、*The Newton Programmer's Guide* の付録「エラー」を参照のこと。

「浮動小数演算環境の管理」page 6-71 には、浮動小数例外に付いて述べている。これは NewtonScript の例外とは関連性がない。

Throw

`Throw(name, data)`

指定された名前とデータを使って例外を発行し、例外フレームを生成する。

name 発生する例外の名前を付ける例外シンボル。

data 例外用のデータ。可能な値は、名前の構造に依存し、それは表 6-2 に述べる。

表 6-2 例外フレームデータのスロット名と内容

例外シンボル	スロット名	スロットの内容
`type.ref` プレフィックスを持つパートを含む	`data`	データオブジェクト
`ext.ex.msg` プレフィックスを含むパートを含む	`message`	メッセージ文字列
その他	`error`	整数のエラーコード

`Throw` に関してより詳しくは、page 3-14 の「例外処理」を参照のこと。

Rethrow

`Rethrow()`

現在の例外を次の `try` ステートメントが処理できるように再発行する。`Rethrow` は、現在の例外を最初に `Throw` が呼ばれたときと同じパラメタで発行する。これによって、現在の例外ハンドラから、次の `Try` ステートメントの中に制御を移すことが出来る。

重要

rethrow 関数は、onexception 節の動的な範囲からだけ呼ぶことが出来る。◆

CurrentException

```
CurrentException()
```

例外を処理している間、(つまり、onexception ブロックの動的な範囲の中で)、現在の例外に結合されているフレームを返す。CurrentException によって返されるフレームを検証することで、現在どの例外を処理しているのか特定することが出来る。例えば、error という名前のスロット名が有るかどうかを、HasSlot 関数で調べることによって、適当なアクションを取ることが出来る。(フレームの形態は例外に依存しているが、例外シンボルを格納する name というスロットだけは常に存在する)。

CurrentException は、onexception 節の動的な範囲内でだけ意味のある値を返す。onexception の外では、この関数は nil を返すだけである。

メッセージ送信関数

これらの関数はメッセージを送信したり、関数を実行する。

Apply

`Apply(`*function*, *parameterArray*`)`

与えられたパラメタを使って関数を呼び出す。Apply 関数は呼びだした関数の戻り値を返す。

function 呼び出す関数。

parameterArray 関数に渡されるパラメタの配列。パラメタがない場合は nil を指定して良い。(これによって、空の配列を生成するのを抑制できる)

Apply は、渡された関数オブジェクトの環境を尊重する。Apply の使用は、call ステートメントの使用に似ている。

Apply は、関数が何個のパラメタを必要とするか実行時までわからないような場合に有用である。関数が取る引数が前もってわかっている場合は、call ステートメントを使用できる。

以下にこの関数をインスペクタで使った例を示す:

```
f:=func(x,y) x*y;
Apply(f,[10,2]);
#50      20
```

これは、次のものと同じである:

```
call f with (10,2);
```

Perform

`Perform(`*frame*, *message*, *parameterArray*`)`

フレームにメッセージを送る。つまり、*frame* 内で、*message* の名前を持つメソッドが実行される。メソッドが *frame* 内にない場合、プロトタイプ・ペアレント継承両方がメソッド探索に使われる。メソッドが見つからなければ、例外が発生する。

frame メッセージを送りたいフレーム。

message 送りたいメッセージの名前となるシンボル。

parameterArray メッセージと一緒に渡したいパラメタの配列。パラメタが無いときには、`nil` を指定してよい。(空の配列を生成する事を抑制できる)

`Perform` 関数のリターン値は、送られたメッセージのリターン値となる。

message によって指定されたメソッドは、*frame* のコンテキストの中で実行されるのであって、`Perform` を呼んだフレームのコンテキスト内で実行されるのではないことに注意。

`Perform` 関数は、メッセージを送信したいが、パラメタの数が前もってわからないような場合に有用である。既にパラメタの数がわかっている場合には、メッセージ送信構文を使うと良い。

`Perform` 関数のバリエーションに、`PerformIfDefined`, `ProtoPerform`, `ProtoPerformIfDefined` がある。

この関数をインスペクタで使った例を示す:

```
f:={multiply: func(x,y) x*y};
perform(f, 'multiply, [10,2]);
#50      20
```

注意

```
f:multiply(10,2)
```

は次のものと同じである。

```
Perform(f, 'multiply,[10,2])
```

PerformIfDefined

`PerformIfDefined(`*receiver*, *message*, *paramArray*`)`

`Perform` とほぼ同じであるが、メソッドが見つからなかった場合、例外は発行せず、`nil` を戻り値として返す点が異なっている。

ProtoPerform

`ProtoPerform(`*receiver*, *message*, *paramArray*`)`

`Perform` とほとんど同じであるが、メソッド探索にプロトタイプ継承だけを使う点が異なっている。(`Perform` は両方の継承を使う。)

ProtoPerformIfDefined

`ProtoPerformIfDefined(`*receiver*, *message*, *paramArray*`)`

`ProtoPerform` とほとんど同じであるが、メソッドが見つからなかった場合、例外は発行せず、`nil` を戻り値として返す点が異なっている。

データ抽出関数

これらの関数は様々な種類のオブジェクトから、データの固まりを抽出するために使える。

整数は 2 の補数で、ビッグ・エンディアン形式で詰め込まれたり、抽出されたりする。この形態では、バイト 0 は、MSB(最上位ビット)に置かれ、これは Newton や Mac で見られる形式である。対してリトル・エンディアンでは、バイト 0 は LSB(最低位ビット)に置かれる。Intel ベースのコンピュータはこの形式である。例として、数値 `0x12345678` の表現形式を以下に示す。

ビッグ・エンディアン	12	34	56	78
リトル・エンディアン	78	56	34	12

◆ MSB（most significant bit: 最高位ビット）

位取りを表現する際に，ビット列の中で最も重みをもつビットの位置。通常最も上位の桁を示す。

◆ LSB（least significant bit: 最低位ビット）

位取りを表現する際に，ビット列の中で最小の重みをもつビットの位置。通常最も下位の桁を示す。

全ての Unicode 変換において、文字コード 128 以上のものに関しては Mac 拡張文字セットが使われる。

ExtractByte

`ExtractByte(`*data*, *offset*`)`

offset 位置から、符号付きの１バイトを取り出す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を表す整数。

例:

```
ExtractByte("\u12345678",0);
#48        18
```

ExtractBytes

`ExtractBytes(`*data*, *offset*, *length*, *class*`)`

data の *offset* 位置から、*length* バイトのデータを取り出し、クラス *class* のバ

イナリオブジェクトとして返す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を表す整数。

length 抜き出すバイト数を表す整数。

class 戻り値のクラスを指定するシンボル

ExtractChar

`ExtractChar(`*data*`,` *offset*`)`

data の *offset* 位置から文字を取り出して返す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

指定されたオフセットから、１バイトを抜き出し、Unicode に変換して返す。

例:

```
ExtractChar("\uFFFFFFFF",0);
//$\u02C results from a ASCII to UNICODE conversion.
#2C76     $\u02C7
//Note $a is at offset 1 in a Unicode string
ExtractChar("abc",0);
#6        $\00
ExtractChar("abc",1);
#616      $a
```

ExtractLong

`ExtractLong(`*data*`,` *offset*`)`

data の与えられたオフセットから右揃えでの低位 29 ビットの符号なし整数を

抜き出し整数オブジェクトとして返す。(つまり、4 バイト値の低位 29 ビットを返す)

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

指定されたオフセットから 4 バイトを抜き出し、上位 2 ビットを無視して、30 ビットの符号付き整数を返す。

```
ExtractLong("\uFFFFFFFF",0);
#FFFFFFFC -1
ExtractLong("\uC0000007",0);
#1C       7
```

ExtractXLong

ExtractXLong(*data*, *offset*)

data の与えられたオフセットから上位 29 ビットの符号なし整数を抜き出し右揃えにして整数オブジェクトとして返す。(つまり、4 バイト値の高位 29 ビットを返す)

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

例:

```
ExtractXLong("\u0000000F",0);
#4        1
```

ExtractWord

ExtractWord(*data*, *offset*)

data の、与えられたオフセット位置から 2 バイトの符号つき整数オブジェクトを返す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

例:

```
ExtractWord("\uFFFFFFFF",0);
#FFFFFFFC -1
//if you want unsigned use:
band(ExtractWord("\uFFFFFFFF",0),0xFFFF);
#3FFFC    65535
```

ExtractCString

ExtractCString(*data*, *offset*)

null で終端された C の文字列の *offset* 位置から Unicode 文字列オブジェクトを返す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

ExtractPString

ExtractPString(*data*, *offset*)

長さを示すバイトのあとにテキストが続く Pascal スタイルの文字列の *offset* 位置から Unicode 文字列オブジェクトを返す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

ExtractUniChar

ExtractUniChar(*data*, *offset*)

data の *offset* 位置から 2 バイトを抜き出し、それらのバイトが示す Unicode 文字を返す。

data 戻り値を抜き出すデータ。

offset 戻り値が抜き出される位置を示した整数。

例:

```
ExtractUniChar("abc",0);
#616      $a
```

データ詰め込み関数

これらの関数は、様々な種類のオブジェクトにデータの固まりを詰め込むために使用される。

全ての整数は、2 の補数形式のビッグ・エンディアン形式で詰め込まれる。

▲ 警告

データ詰め込み対象となるデータオブジェクトは、十分な領域を確保していないといけない。詰め込み先が十分大きくない場合、NewtonScript のヒープ領域がおかしくなる。詰め込むオフセットには注意するように。次の式を使うと良い。

`Length(`*destObj*`)` – オフセット >= 詰め込まれるデータのサイズ

この式では、*destObj* は詰め込み先で、オフセットは *destObj* 内での詰め込み開始位置である。▲

StuffByte

`StuffByte(`*obj*, *offset*, *toInsert*`)`

toInsert の、低位バイトを *obj* の *offset* 位置から書き込む。

obj データが詰め込まれるバイナリオブジェクト。

offset *obj* 内での、詰め込み開始位置。

toInsert *obj* に詰め込むデータ。

例:

```
x := "\u00000000";
StuffByte(x,0,-1);
x[0]
#FF006    $\uFF00
x := "\u00000000";
StuffByte(x,0,0xFF);
x[0]
#FF006    $\uFF00
```

StuffChar

StuffChar(*obj*, *offset*, *toInsert*)

obj に offset 位置から一バイト詰め込む。

obj データが詰め込まれるバイナリオブジェクト。

offset *obj* 内での、詰め込み開始位置。

toInsert *obj* に詰め込まれる文字または整数。2 バイトの Unicode を *toInsert* として渡す。関数は 1 バイトの文字を作り、1 バイトを書き込む。

toInsert には、文字でも整数でも渡して良い:

- *toInsert* が整数なら、低バイトが書かれる。
- *toInsert* が文字なら、Unicode から変換されてバイトが書かれる。

例:

```
x := "\u00000000";
```

```
StuffChar(x,1,Ord($Z));
x[0]
#5A6      $Z
x := "\u00000000";
StuffChar(x,1,-1);
x[0]
#1A6      $\1A
ExtractByte(x,1)
#68       26
ExtractByte(x,0)
#0        0
```

StuffCString

`StuffCString(`*obj*`,` *offset*`,` *aString*`)`

Unicode 文字列を、C スタイルの null 終端文字列に変換し、バイナリオブジェクトの *offset* 位置から詰め込む。

obj	データが詰め込まれるバイナリオブジェクト。
offset	*obj* 内での、詰め込み開始位置。
aString	*obj* に詰め込まれる Unicode 文字列。

文字列 *aString* は Macintosh のローマ字文字列を使って ASCII に変換される。そして、*obj* の *offset* バイト目から詰め込まれる。最後に null 終端が付加される。

この関数は、*offset* から始まる *obj* に *aString* が入りきらない場合や、*offset* が負の値である場合、例外を発生する。*obj* の長さが変更されることはない。

StuffLong

`StuffLong(`*obj*`,` *offset*`,` *toInsert*`)`

第 3 パラメタとして渡した整数から、30bit の符号付きの値を 4 バイトで書き込む。

obj データが詰め込まれるバイナリオブジェクト。

offset *obj* 内での、詰め込み開始位置。

toInsert *obj* に詰め込まれるデータ。

例:

```
x := "\u00000000";
StuffLong(x,0,-1);
x[0]
#FFFF6    $\uFFFF
x[1]
#FFFF6    $\uFFFF
x := "\u00000000";
StuffLong(x,0,0x3FFFFFFA);
x[0]
#FFFF6    $\uFFFF
x[1]
#FFFA6    $\uFFFA
```

StuffPString

`StuffPString(`*obj*, *offset*, *aString*`)`

Unicode 文字列を、Pascal スタイルの文字長を表すバイトの後にテキストが続く文字列に変換し、バイナリオブジェクトの *offset* 位置から詰め込む。

obj データが詰め込まれるバイナリオブジェクト。

offset *obj* 内での、詰め込み開始位置。

aString *obj* に詰め込まれる Unicode 文字列。長さが 255 文字より長くてはいけない。

文字列 *aString* は Macintosh のローマ字文字列を使って ASCII に変換される。そして、*obj* の *offset* バイト目から文字列長を表す１バイトと文字列が、詰め込まれる。

この関数は、*offset* から始まる *obj* に *aString* が入りきらない場合や、*offset* が負の値である場合、例外を発生する。*obj* の長さが変更されることはない。

StuffUniChar

StuffUniChar(*obj*, *offset*, *toInsert*)

toInsert で表される 2 バイトの Unicode 文字を、*obj* の *offset* 位置から詰め込む。

obj データが詰め込まれるバイナリオブジェクト。

offset *obj* 内での、詰め込み開始位置。

toInsert *obj* に詰め込まれる文字または整数。

例:

```
x := "\u00000000";
StuffUniChar(x,0,"\uF00F"[0]);
x[0]
#F00F6    $\uF00F
x := "\u00000000";
StuffUniChar(x,0,0x0AA0);
x[0]
#AA06     $\u0AA0
```

StuffWord

StuffWord(*obj*, *offset*, *toInsert*)

toInsert の低位 2 バイトを、指定された *offset* から書き込む。

obj データが詰め込まれるバイナリオブジェクト。

offset *obj* 内での、詰め込み開始位置。

toInsert *obj* に詰め込まれるデータ。

例:

```
x := "\u00000000";
StuffWord(x,0,0x3FFF1234);
x[0]
#12346    $\u1234
x := "\u00000000";
StuffWord(x,0,-1);
x[0]
#FFFF6    $\uFFFF
```

グローバルな変数・関数の取得及びセット

これらの関数は、グローバルな変数または関数の取得・セット・存在確認を行うためのものである。

GetGlobalFn

GetGlobalFn(*symbol*)

グローバル関数を返す。もし、関数が見つからなければ、nil が返る。

symbol 取得したいグローバル関数の名前を表すシンボル。

GetGlobalVar

GetGlobalVar(*symbol*)

システムのグローバルフレームにあるスロットの値を返す。スロットが見つからなければ、nil が返る。

symbol 値を取得したいグローバル変数の名前を表すシンボル。

GlobalFnExists

`GlobalFnExists(`*symbol*`)`

シンボルで表されるグローバル関数が有れば非 `nil` の値を返し、さもなくば `nil` を返す。

symbol 存在を確認したいグローバル関数の名前を表すシンボル。

GlobalVarExists

`GlobalVarExists(`*symbol*`)`

シンボルで表されるグローバル変数が有れば非 `nil` の値を返し、さもなくば `nil` を返す。

symbol 存在を確認したいグローバル変数の名前を表すシンボル。

DefGlobalFn

`DefGlobalFn(`*symbol*, *function*`)`

グローバル関数を定義する。関数を特定するシンボルを返す。

symbol 定義したいグローバル関数の名前を定義するシンボル。他のグローバル関数との名前の重複を避けるためには、Newton DTS に登録した自分のシグネチャを含む、`appSymbol` を含む名前を付けると良い。

function 関数オブジェクト。

グローバル関数は、システムがリセットされたときにリセットされる事に注意。

自分のアプリケーションが作ったグローバル関数を、アプリケーションが削除されるときに削除することが重要である。これは、`UnDefGlobalFn` を、`RemoveScript` 内から呼ぶことでできる。

重要

本当に必要でない限り、グローバル関数は定義しないこと。これは、ヒープを消費する。それは、システムのグローバル関数や、他のアプリケーションのグローバル関数と衝突しうる。多くの場合、アプリケーションのベースビューのメソッドで、グローバル関数は代用できるはずである。▲

DefGlobalVar

`DefGlobalVar(`*symbol*`,` *value*`)`

グローバル変数(システムのグローバルフレーム内のスロット)を定義する。変数の値が返る。

symbol 定義したいグローバル変数の名前となるシンボル。他のグローバル変数との名前の重複を避けるためには、Newton DTS に登録した自分のシグネチャを含む、`appSymbol` を含む名前を付けると良い。

value セットしたい値。

システムは、生成されたオブジェクトが全て内蔵 RAM に格納されていることを保証する。それは、`EnsureInternal` を *symbol* で指定されるオブジェクトに適用して行う。グローバル変数はシステムリセットで消去されることに注意。

自分のアプリケーションが作ったグローバルを、アプリケーションが削除されるときに削除することが重要である。これは、`UnDefGlobalVar` を、`RemoveScript` 内から呼ぶことで出来る。

重要

本当に必要でない限り、グローバル変数は定義しないこと。これは、ヒープを消費する。それは、システムのグローバル変数や、

他のアプリケーションのグローバル変数と衝突しうる。多くの場合、アプリケーションのベースビューやスープのスロットで、グローバル変数は代用できるはずである。◆

UnDefGlobalFn

`UnDefGlobalFn(`*symbol*`)`

以前定義したグローバル関数を削除する。この関数は `nil` を返す。

symbol 削除したいグローバル関数の名前を表すシンボル。

UnDefGlobalVar

`UnDefGlobalVar(`*symbol*`)`

以前定義したグローバル変数を削除する。この関数は `nil` を返す。

symbol 削除したいグローバル変数の名前を表すシンボル。

その他の関数

その他の関数について説明する。

BinEqual

`BinEqual(`*a*`,` *b*`)`

a バイナリオブジェクト。

b バイナリオブジェクト。

二つのバイナリオブジェクトの実際のバイトを比較する。それが同じものなら非 `nil` を返す。

BinaryMunger

`BinaryMunger(`*dst*, *dstStart*, *dstCount*, *src*, *srcStart*, *srcCount*`)`

dst 内のバイトを、*src* 内のバイトで置換し、切った張ったが完了した後 *dst* を返す。*dst* の内容は破壊される。

dst	変更される値。
dstStart	*dst* 内での開始位置。
dstCount	*dst* 内で置換対象にするバイト数。*dstCount* に `nil` を指定すれば、*dst* の最後までが対象となる。
src	値。`nil` を指定すると、単に *dst* の内容を削除することが出来る。
srcStart	ソースバイナリ内での *dst* に置くデータ要素を取り出すスタート位置。
srcCount	ソースバイナリから取り出すバイト数。`nil` を指定すると、ソースバイナリの最後までが使用される。

バイト数は、0 からカウントされる。

Chr

`Chr(`*integer*`)`

10 進数の整数を、Unicode 文字に変換する。

integer	整数。

例:

```
chr(65)
$A
```

Compile

`Compile(`*string*`)`

式の並びをコンパイルし、それを評価する関数を返す。

string コンパイルしたい式。

二つの例を示す。最初のサンプルでは、x がローカル変数であることに注意。

```
compile("x:= {a:self.b, b:1234}")
#440F711  <CodeBlock, 0 args #440F711>
f:=compile("2+2");
call f with ();
#440F712 4
```

注意

NewtonScript で使用する全ての文字は、7bit アスキーでないといけない。これは、通常何の問題もないが、特定の状況で問題を起こす。次の式を試したとする:

```
Compile ("blah, blah, blah,  \u0F0F\u")
```

Unicode 文字は、7bit アスキーではなく、16bit なので、エラーが発生する。(\u が、Unicode 文字モードをスタートさせている)その代わりに、次のようにすればよい:

```
Compile ("blah, blah, blah,  \\u0F0F\\u")
```

訳註: 所が、ここで与えているのは式として成り立つはずもないものなので、いずれにせよエラーとなるのである。何を考えてるんだか。

バックスラッシュエスケープ文字が前にあるので、コンパイル時に \u が Unicode モードを開始せずに済んでいる。(\u は、単に文字列 "\u" として読まれているだけで、Unicode スイッチにはなっていない)

また、次の点も注意したい:

```
compile("func()...")
```

こうすると、関数を作る関数を返す。環境は関数コンストラクタが実行されたときに取り込まれる:

```
f := compile("func()b");
x := {a:f, b:0};
g:=x:a();
#440F713 <function, 0 args #440F711>
```

関数コンストラクタの実行により、x がメッセージのレシーバとして取り込まれるので、x がレシーバーとなる。

```
call g with ();
#440F714 0
```

よって、b が見つかってしまう。◆

Ord

Ord (*char*)

文字を、等価な Unicode の 10 進数の整数に変換する。

char 文字。

例を示す:

```
ord($A)
#104   65
```

関数とメソッドの要約

このセクションでは、この章で説明された関数とメソッドの要約を示す。

オブジェクトシステム関数

`ClassOf(`*object*`)`

`Clone(`*object*`)`

`DeepClone(`*object*`)`

`GetFunctionArgCount(`*function*`)`

`GetSlot(`*frame*, *slotSymbol*`)`

`GetVariable(`*frame*, *slotSymbol*`)`

`HasSlot(`*frame*, *slotSymbol*`)`

`HasVariable(`*frame*, *slotSymbol*`)`

`Intern(` *string* `)`

`IsArray(`*obj*`)`

`IsBinary(`*obj*`)`

`IsCharacter(`*obj*`)`

`IsFrame(`*obj*`)`

`IsFunction(`*obj*`)`

`IsImmediate(`*obj*`)`

`IsInstance(`*obj*, *class*`)`

`IsInteger(`*obj*`)`

`IsNumber(`*obj*`)`

`IsReadOnly(`*obj*`)`

`IsReal(`*obj*`)`

`IsString(`*obj*`)`

`IsSubclass(`*class1*, *class2*`)`

`IsSymbol(`*obj*`)`

`MakeBinary(`*length*, *class*`)`

`Map(`*obj*, *function*`)`

`PrimClassOf(`*obj*`)`

`RemoveSlot(`*obj*, *slot*`)`

`ReplaceObject(`*originalObject*, *targetObject*`)`

SetClass(*obj*, *classSymbol*)
SetVariable(*frame*, *slotSymbol*, *value*)
SymbolCompareLex(*symbol1*, *symbol2*)
TotalClone(*obj*)

文字列関数

BeginsWith(*string*, *substr*)
Capitalize(*string*)
CapitalizeWords(*string*)
CharPos(*str*, *char*, *startpos*)
Downcase(*string*)
EndsWith(*string*, *substr*)
IsAlphaNumeric(*char*)
IsWhiteSpace(*char*)
SPrintObject(*obj*)
StrCompare(*a*, *b*)
StrConcat(*a*, *b*)
StrEqual(*a*, *b*)
StrExactCompare(*a*, *b*)
StrLen(*string*)
StrMunger(*dstString*, *dstStart*, *dstCount*, *srcString*, *srcStart*, *srcCount*)
StrPos(*string*, *substr*, *start*)
StrReplace(*string*, *substr*, *replacement*, *count*)
StrTokenize(*str*, *delimiters*)
StyledStrTruncate(*string*, *length*, *font*)
SubStr(*string*, *substr*, *start*)
TrimString(*string*)
Upcase(*string*)

ビット演算関数

Band(*a*, *b*)

Bor(*a*, *b*)

Bxor(*a*, *b*)

Bnot(*a*)

配列関数

AddArraySlot(*array*, *value*)

Array(*size*, *initialValue*)

ArrayInsert(*array*, *element*, *position*)

ArrayMunger(*dstArray*, *dstStart*, *dstCount*, *srcArray*, *srcStart*, *srcCount*)

ArrayRemoveCount(*array*, *startIndex*, *count*)

InsertionSort(*array*, *test*, *key*)

Length(*array*)

LFetch(*array*, *item*, *start*, *test*, *key*)

LSearch(*array*, *item*, *start*, *test*, *key*)

NewWeakArray(*length*)

SetAdd(*array*, *value*, *uniqueOnly*)

SetContains(*array*, *item*)

SetDifference(*array1*, *array2*)

SetLength(*array*, *length*)

SetOverlaps(*array1*, *array2*)

SetRemove(*array*, *value*)

SetUnion(*array1*, *array2*, *uniqueFlag*)

Sort(*array*, *test*, *key*)

StableSort(*array*, *test*, *key*)

ソートされた配列の関数

BDelete(*array*, *item*, *test*, *key*, *count*)
BDifference(*array1*, *array2*, *test*, *key*)
BFetch(*array*, *item*, *test*, *key*)
BFetchRight(*array*, *item*, *test*, *key*)
BFind(*array*, *item*, *test*, *key*)
BFindRight(*array*, *item*, *test*, *key*)
BInsert(*array*, *element*, *test*, *key*, *uniqueOnly*)
BInsertRight(*array*, *element*, *test*, *key*, *uniqueOnly*)
BIntersect(*array1*, *array2*, *test*, *key*, *uniqueOnly*)
BMerge(*array1*, *array2*, *test*, *key*, *uniqueOnly*)
BSearchLeft(*array*, *item*, *test*, *key*)
BSearchRight(*array*, *item*, *test*, *key*)

整数演算関数

Abs(*x*)
Ceiling(*x*)
Floor(*x*)
Max(*a*, *b*)
Min(*a*, *b*)
Real(*x*)
Random(*low*, *high*)
SetRandomSeed (*seedNumber*)

浮動小数演算関数

Acos(*x*)
Acosh(*x*)
Asin(*x*)
Logb(*x*)
Log1p(*x*)
Log10(*x*)

Asinh(*x*)
Atan(*x*)
Atan2(*x*,*y*)
Atanh(*x*)
CopySign(*x*,*y*)
Cos(*x*)
Cosh(*x*)
Erf(*x*)
Erfc(*x*)
Exp(*x*)
Expm1(*x*)
Fabs(*x*)
FDim(*x*,*y*)
FMax(*x*,*y*)
FMin(*x*,*y*)
Fmod(*x*,*y*)
Gamma(*x*)
Hypot(*x*,*y*)
IsFinite(*x*)
IsNaN(*x*)
IsNormal(*x*)
LessEqualOrGreater(*a*, *b*)
LessOrGreater(*a*, *b*)
LGamma(*x*)
Log(*x*)
NearbyInt(*x*)
NextAfterD(*x*,*y*)
Pow(*x*,*y*)
RandomX(*x*)
Remainder(*x*,*y*)
RemQuo(*x*,*y*)
Rint(*x*)
RintToL(*x*)
Round(*x*)
Scalb(*x*,*y*)
SignBit(*x*)
Signum(*x*)
Sin(*x*)
Sinh(*x*)
Sqrt(*x*)
Tan(*x*)
Tanh(*x*)
Trunc(*x*)
Unordered(*a*, *b*)
UnorderedGreaterOrEqual(*a*, *b*)
UnorderedLessOrEqual(*a*, *b*)
UnorderedOrEqual(*a*, *b*)
UnorderedOrGreater(*a*, *b*)
UnorderedOrLess(*a*, *b*)

浮動小数演算環境の管理

FeClearExcept(*excepts*)
FeGetEnv()

`FeGetExcept(`*excepts*`)`
`FeHoldExcept()`
`FeRaiseExcept(`*excepts*`)`
`FeSetEnv(`*envObj*`)`
`FeSetExcept(`*flagObj*, *excepts*`)`
`FeTestExcept(`*excepts*`)`
`FeUpdateEnv(`*flagObj*`)`

財務関数

`Annuity(`*rate*, *periods*`)`
`Compound(`*rate*, *periods*`)`

例外関数

`Throw(`*name*, *data*`)`
`Rethrow()`
`CurrentException()`

メッセージ送信関数

`Apply(`*function*, *parameterArray*`)`
`Perform(`*frame*, *message*, *parameterArray*`)`
`PerformIfDefined(`*receiver*, *message*, *paramArray*`)`
`ProtoPerform(`*receiver*, *message*, *paramArray*`)`
`ProtoPerformIfDefined(`*receiver*, *message*, *paramArray*`)`

データ抽出関数

`ExtractByte(`*data*, *offset*`)`
`ExtractBytes(`*data*, *offset*, *length*, *class*`)`
`ExtractChar(`*data*, *offset*`)`

`ExtractLong(`*data*`,` *offset*`)`
`ExtractXLong(`*data*`,` *offset*`)`
`ExtractWord(`*data*`,` *offset*`)`
`ExtractCString(`*data*`,` *offset*`)`
`ExtractPString(`*data*`,` *offset*`)`
`ExtractUniChar(`*data*`,` *offset*`)`

データ詰め込み関数

`StuffByte(`*aString*`,` *offset*`,` *toInsert*`)`
`StuffChar(`*aString*`,` *offset*`,` *toInsert*`)`
`StuffCString(`*obj*`,` *offset*`,` *aString*`)`
`StuffLong(`*aString*`,` *offset*`,` *toInsert*`)`
`StuffPString(`*obj*`,` *offset*`,` *aString*`)`
`StuffUniChar(`*aString*`,` *offset*`,` *toInsert*`)`
`StuffWord(`*aString*`,` *offset*`,` *toInsert*`)`

グローバル変数・関数の取得とセットに関する関数

`GetGlobalFn(`*symbol*`)`
`GetGlobalVar(`*symbol*`)`
`GlobalFnExists(`*symbol*`)`
`GlobalVarExists(`*symbol*`)`
`DefGlobalFn(`*symbol*`,` *function*`)`
`DefGlobalVar(`*symbol*`,` *value*`)`
`UnDefGlobalFn(`*symbol*`)`
`UnDefGlobalVar(`*symbol*`)`

その他の関数

`BinEqual(`*a*`,` *b*`)`
`BinaryMunger(`*dst*`,` *dstStart*`,` *dstCount*`,` *src*`,` *srcStart*`,` *srcCount*`)`

`Chr(`*integer*`)`
`Compile(`*string*`)`
`Ord(`*char*`)`

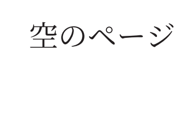
APPENDIX A

予約語

以下はNewtonScriptの予約語である。以下の語をシンボルとして使うときは、|self| のように縦棒で囲まなければならない。

and	end	local	self
begin	exists	loop	then
break	for	mod	to
by	foreach	native	try
call	func	not	until
constant	global	onexception	while
div	if	or	with
do	in	repeat	
else	inherited	return	

訳註: collect, from **は予約語ではなくなったらしい。**


空のページ

APPENDIX B

文字コード表

この付録は、Newton の文字セットにおける後半 128 文字(コード 128 から 254 まで) の Macintosh と Unicode(16bit) の文字コード対応表を収録する。後半 128 の文字を文字定数または文字列に含める場合、Unicode 文字コードを使う必要がある。Macintosh の文字コードはすでにそれを使っている人の便宜をはかるために用意した。

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
80	00C4	Ä
81	00C5	Å
82	00C7	Ç
83	00C9	É
84	00D1	Ñ
85	00D6	Ö
86	00DC	Ü
87	00E1	á
88	00E0	à
89	00E2	â
8A	00E4	ä
8B	00E3	ã
8C	00E5	å
8D	00E7	ç
8E	00E9	é
8F	00E8	è

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
90	00EA	ê
91	00EB	ë
92	00ED	í
93	00EC	ì
94	00EE	î
95	00EF	ï
96	00F1	ñ
97	00F3	ó
98	00F2	ò
99	00F4	ô
9A	00F6	ö
9B	00F5	õ
9C	00FA	ú
9D	00F9	ù
9E	00FB	û
9F	00FC	ü
A0	2020	†
A1	00B0	°
A2	00A2	¢
A3	00A3	£
A4	00A7	§
A5	2022	
A6	00B6	¶
A7	00DF	ß
A8	00AE	®
A9	00A9	©
AA	2122	™
AB	00B4	´

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
AC	00A8	¨
AD	2260	≠
AE	00C6	Æ
AF	00D8	Ø
B0	221E	∞
B1	00B1	±
B2	2264	≤
B3	2265	≥
B4	00A5	¥
B5	00B5	µ
B6	2202	∂
B7	2211	∑
B8	220F	∏
B9	03C0	π
BA	222B	∫
BB	00AA	ª
BC	00BA	º
BD	2126	Ω
BE	00E6	æ
BF	00F8	ø
C0	00BF	¿
C1	00A1	¡
C2	00AC	¬
C3	221A	√
C4	0192	ƒ
C5	2248	≈
C6	2206	∆
C7	00AB	«

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
C8	00BB	»
C9	2026	…
CA	00A0	
CB	00C0	À
CC	00C3	Ã
CD	00D5	Õ
CE	0152	Œ
CF	0153	œ
D0	2013	
D1	2014	
D2	201C	
D3	201D	
D4	2018	
D5	2019	
D6	00F7	÷
D7	25CA	◊
D8	00FF	ÿ
D9	0178	Ÿ
DA	2044	⁄
DB	00A4	¤
DC	2039	‹
DD	203A	›
DE	FB01	ﬁ
DF	FB02	ﬂ
E0	2021	‡
E1	00B7	·
E2	201A	‚
E3	201E	„

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
E4	2030	‰
E5	00C2	Â
E6	00CA	Ê
E7	00C1	Á
E8	00CB	Ë
E9	00C8	È
EA	00CD	Í
EB	00CE	Î
EC	00CF	Ï
ED	00CC	Ì
EE	00D3	Ó
EF	00D4	Ô
F0	F7FF	
F1	00D2	Ò
F2	00DA	Ú
F3	00DB	Û
F4	00D9	Ù
F5	0131	ı
F6	02C6	ˆ
F7	02DC	˜
F8	00AF	¯
F9	02D8	˘
FA	02D9	˙
FB	02DA	˚
FC	00B8	¸
FD	02DD	˝
FE	02DB	˛
FF	02C7	ˇ

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
CA	00A0	
C1	00A1	¡
A2	00A2	¢
A3	00A3	£
DB	00A4	¤
B4	00A5	¥
A4	00A7	§
AC	00A8	¨
A9	00A9	©
BB	00AA	ª
C7	00AB	«
C2	00AC	¬
A8	00AE	®
F8	00AF	¯
A1	00B0	°
B1	00B1	±
AB	00B4	´
B5	00B5	µ
A6	00B6	¶
E1	00B7	·
FC	00B8	¸
BC	00BA	º
C8	00BB	»
C0	00BF	¿
CB	00C0	À
E7	00C1	Á
E5	00C2	Â
CC	00C3	Ã

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
80	00C4	Ä
81	00C5	Å
AE	00C6	Æ
82	00C7	Ç
E9	00C8	È
83	00C9	É
E6	00CA	Ê
E8	00CB	Ë
ED	00CC	Ì
EA	00CD	Í
EB	00CE	Î
EC	00CF	Ï
84	00D1	Ñ
F1	00D2	Ò
EE	00D3	Ó
EF	00D4	Ô
CD	00D5	Õ
85	00D6	Ö
AF	00D8	Ø
F4	00D9	Ù
F2	00DA	Ú
F3	00DB	Û
86	00DC	Ü
A7	00DF	ß
88	00E0	à
87	00E1	á
89	00E2	â
8B	00E3	ã

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
8A	00E4	ä
8C	00E5	å
BE	00E6	æ
8D	00E7	ç
8F	00E8	è
8E	00E9	é
90	00EA	ê
91	00EB	ë
93	00EC	ì
92	00ED	í
94	00EE	î
95	00EF	ï
96	00F1	ñ
98	00F2	ò
97	00F3	ó
99	00F4	ô
9B	00F5	õ
9A	00F6	ö
D6	00F7	÷
BF	00F8	ø
9D	00F9	ù
9C	00FA	ú
9E	00FB	û
9F	00FC	ü
D8	00FF	ÿ
F5	0131	ı
CE	0152	Œ

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
CF	0153	œ
D9	0178	Ÿ
C4	0192	ƒ
F6	02C6	ˆ
FF	02C7	ˇ
F9	02D8	˘
FA	02D9	˙
FB	02DA	˚
FE	02DB	˛
F7	02DC	˜
FD	02DD	˝
B9	03C0	π
D0	2013	
D1	2014	
D4	2018	
D5	2019	
E2	201A	‚
D2	201C	
D3	201D	
E3	201E	„
A0	2020	†
E0	2021	‡
A5	2022	
C9	2026	…
E4	2030	‰
DC	2039	‹
DD	203A	›

continued

表 B-1 Macintosh 文字コード順の文字コード表

Mac	Unicode	Char
DA	2044	⁄
AA	2122	™
BD	2126	Ω
B6	2202	∂
C6	2206	∆
B8	220F	∏
B7	2211	∑
C3	221A	√
B0	221E	∞
BA	222B	∫
C5	2248	≈
AD	2260	≠
B2	2264	≤
B3	2265	≥
D7	25CA	◊
F0	F7FF	
DE	FB01	ﬁ
DF	FB02	ﬂ

APPENDIX C

クラスベースのプログラミング

*NewtonScript は、しばしば「オブジェクト指向の」言語と記述されている。しかし、Smalltalk や C++ のようなオブジェクト指向の経験を持つ人なら、少し混乱するだろうと思う。NewtonScript はおなじみの多くの機能を持っているが、一つ重要な違いがある。それは NewtonScript はクラスベースでなく、プロトタイプベースと言うことだ。これは、継承の為に世界をクラスとインスタンスに２分してしまうのでなく、他のオブジェクトから直接継承するという事である。

とはいえ、クラスベースのプログラミングについての知識は忘れないように。NewtonScript で、クラスをシミュレートするのは可能だし、十分望みはある。クラスをサポートする明示的な機能を言語が持っていなくても、プロトタイプの柔軟性を失うことなく、クラスのよく知られた利点を得るために、必要に応じて簡単な記述規則を使えるのだ。

クラスのどこがいいのか?

Newton プログラミングは、ビューシステムにものすごく重きを置いている。アプリケーションの構造は、ユーザーインターフェースを作るためのビューを基盤としている。Newton Toolkit は、この強力なビュー指向を反映しており、データやメソッドを追加してビューを作るのがとても簡単に出来ている。しかし、プログラムを作るのに、ビューシステムだけを使うのは必ずしも適当ではない。

複雑な多くのアプリケーションは様々な、独立した、かなり複雑なデータ構造を使う。そうした構造を、機能をカプセル化する抽象データとして実装するのが標準的なプログラミングテクニックである。オブジェクト指向プログラムではそれはクラスという形態をとる。

*Copyright © 1993, 1994 Walter R. Smith. All Rights Reserved. This article is reprinted by permission of the author.

クラスはプログラムの機能を管理しやすい断片に分割させてくれる。データ構造とそれを操作する関数を組み合わせることによって、クラスはプログラムをより理解しやすく、保守しやすいものにする。上手く作られているクラスは他のアプリケーションでも再利用可能だし、プログラマの労力を軽減してくれる。クラスというアイデアが何故いいのかについては、いろんな理由があるが、より詳しくはオブジェクト指向の本で読んでもらいたい。

ユーザーインターフェースのデザインに応じて、プログラマはアプリケーションをパートに分割するため、ビュー構造を使うだろう。内部的なデータの一部または全部をクラスとして実装するのはよいやり方である。

クラス: 簡単な復習

伝統的なクラスベースモデルのオブジェクトプログラミングの復習を始めよう。ここでは、Smalltalk のコンセプトと用語を使う。C++ 界の住人は、彼らの体系に合わせるため、議論を少しばかり翻訳する必要があるだろう。

クラスベースモデルの第一のコンセプトは驚くなかれクラスである。クラスはクラスのインスタンスと呼ばれるオブジェクトの集合の構造と動作を定義する。各インスタンスはクラスで指定されたインスタンス変数の集合を持つ。インスタンスはクラスで定義されたメソッドを実行することで、メッセージに応答する。各インスタンスは同じインスタンス変数とメソッドを持つ。さらに、クラスはクラス変数とクラスメソッドを持ち、それはそのクラスの全てのインスタンスから使用可能である。

インヘリタンス(継承)もまた、クラスにより特定される。各クラスはスーパークラスを持ち、そこから変数とメソッド定義を継承する。いくつかの言語では、クラスは複数のスーパークラスを持ちうるが、NewtonScript でそれをシミュレートするのは容易なことではない。だから、ここではそれについては考えない。

オブジェクトは通常 New とかそれに似たような呼び方をされるメッセージを

送ることによって生成される。それはまた、クラスのインスタンスに Clone メッセージを送ることによっても生成できる。メッセージがインスタンスに送られるとき、クラス(あるいはスーパークラス)内の対応するメソッドが実行される。メソッドは直接インスタンス変数やクラス変数を、個々のインスタンスから参照できる。

NewtonScript での継承

NewtonScript のオブジェクトモデルはプロトタイプベースである。フレームは他のフレームから直接継承され、そこにはクラスはない。フレームは _proto, _parent スロットを通して他のフレームにリンク出来、それら 2 つのスロットがフレームの継承パスを定義する。フレームにメッセージを送ると、実行されるメソッドは、(レシーバ)フレームのスロットを変数として使うことが出来る。変数やメソッドの参照が、レシーバー内で解決できない場合は、プロトタイプチェーンが探索される。もし必要なスロットがそこになければ、探索はペアレントチェーンを１ステップ上り、親とその親のプロトタイプチェーンを探し、これが延々と続く。

こうしたルールはビューシステム指向の Newton プログラミング環境にとても良く合っていることから起こったものである。ペアレント継承はビュー階層を通した変数とメッセージの継承を提供し、全てのサブビューから使える変数を親ビューに定義することもできる。プロトタイプ継承によって、ビューは共通のテンプレートを共有できるし、多くのデータを RAM の外に置くことが可能になる。

継承システム(そして NewtonScript の全てが)ビューシステムに密接に統合されているとは言え、実際それはいろんなやり方で適用できる一組のルールに過ぎない。ビューフレームだけでなく特定のフレームにメッセージを送ることもできるし、ビューフレームではない(ただの)フレームもまた継承ルールの利点を得ることができる。この記事が実証している様に、同じルールがクラスベースプログラミングの形態にも適用できる。

基本的なアイデア

基本的なクラスベースの NewtonScript テクニックを話そう。NewtonScript には組み込みのクラスやインスタンスのアイデアは全くないことを覚えて置いて欲しい。これから説明するのは、クラスベースの言語で作ることが出来るのと似たデータ構造を作れるようにする NewtonScript のための記述方法の集合である。だから、クラス、インスタンスその他の言葉を使うが、それらは全て特定の目的で使われるフレームの事を指している。

主なアイデアは、ペアレント継承をクラスとインスタンスの接続に使うことである。インスタンスはそのスロットがインスタンス変数となるフレームであり、その `_parent` スロットがクラス(別のフレーム)を指す。そして、クラスのスロットはメソッドを作る。

簡単なサンプルとして、クラス `Stack` を考えてみる。クラス `Stack` は、いわゆる後入れ先出しのスタックを作るものだ。それは標準の操作 `Push` と `Pop` をもち、それはアイテムを追加したり削除したりする。そして、スタック上にデータがあるかないかを判定するメソッド `IsEmpty` を持つ。その表現は非常にシンプルで、アイテムの配列と、スタックのトップにあるアイテムへのインデックスとなる整数が有るだけである。クラスのフレームは次のようなものだ:

```
Stack := {
New:
       func (maxItems)
         {_parent: self,
          topIndex: -1,
          items: Array(maxItems, NIL)},
Clone:
       func () begin
         local newObj := Clone(self);
         newObj.items := Clone(items);
         newObj
       end,
```

```
Push:
        func (item) begin
          topIndex := topIndex + 1;
          items[topIndex] := item;
          self
        end,

Pop:
        func () begin
          if :IsEmpty() then
            NIL
          else begin
            local item := items[topIndex];
            items[topIndex] := NIL;
            topIndex := topIndex - 1;
            item
          end
        end,
IsEmpty:
        func ()
          topIndex = -1
};
```

クラスフレームはNewメソッドで始まる。これはStack:New(16)のように、クラスStack自身へのメッセージとして使われることを意図したクラスメソッドである。

これは、インスタンスフレームを作るフレームコンストラクタから単純に構成される。インスタンスは常にクラスフレームを参照する_parentスロットを持つが、Newはクラスに送信されることを意図したメッセージなので、クラスポインタにはselfが使用できる。残りのスロットはインスタンス変数:topIndexとitemsを含む。topIndexはスタック用配列itemsの最上位

にあるデータの位置を示すインデックスで、これが –1 ならスタックは空であることを意味する。New メソッドはスタック上の要素の最大数を指定するための引数を一つ取る。(この記事にはそぐわないが)この最大数を動的に変更可能にすることは簡単である。

クラスのための Clone メソッドを用意するのは良い事である。これはどのくらいコピーが深く行われるか知らなくても、オブジェクトのコピーを可能にしてくれるのである。(そうした知識はクラスを取り込もうとした最初の計画の一つの理由である情報の隠蔽を妨害するものだ)この Stack のケースでは、この単純な Clone は二つのインスタンスの間で items をシェアしたままにする。このことは結果として混乱を引き起こすし、正しくない動作をするだろう。一方、DeepClone を使えば、そのインスタンスだけでなく、_parent スロット内のポインタがあるので、クラスフレーム全体をコピーするだろう。これはこのケースでは正しく動作する、しかしこの種のミスを犯すと、大量のスペースが消費されてしまう。Stackを複製する正しい方法は、インスタンスを複製し、上の Clone メソッドがやっているように、その後そこに items 配列のクローンを与えてやることだ。

一般にどのクラスにも存在する New と Clone メソッドのあとで、個々のクラスの為のメソッドが来る。Push メソッドは、topIndex の値を増やし、items 配列の終端に内容を追加する。topIndex と items のようなインスタンス変数は、その名前によって簡単にアクセスされている。なぜなら、それらはレシーバーのスロットだからだ。Pop メソッドはまず IsEmpty メソッドを呼び、スタックが空かどうかをチェックしている。もし、スタックが空なら、nil が返され、そうでなければスタック上の最上位のアイテムが返された後、topIndex の値が減算される。Pop が、直前のスタックのトップにあるアイテムの値を nil にしているのは、そうしないと、そのアイテムがガベージコレクションの対象とならないからである。

NewtonScirpt を Smalltalk におけるクラスコードに似たコードを書くために使うことができ、そして、New メッセージによってそのクラスのオブジェクトを作

ることが出来る。そして生じたインスタンスをメッセージによって使うことが出来る。もちろん、そうしたことは全てメッセージ転送にコロンを使うNewtonScriptの構文で行わないといけない。

```
s := Stack:New(16);
s:Push(10);
s:Push(20);
x := s:Pop() + s:Pop();
```

このコードの最後には、`x`の値は30になるだろう。

実践的な情報

より進んだトピックに進む前に、現状のツールでクラスベースのプログラミングを行うための実践的な情報を示す。(このセクションで話す、Newton Toolkitの実装に関する情報については、*Newton Toolkit User's Guide*を参照のこと)

現在のNewton Toolkitは、クラスベースプログラミングをサポートするための特別な機能を何も持たない。たとえばブラウザはビューの階層を表示するだけだ。しかし、アプリケーションにクラスを導入するのはそんなに難しくはない。

パッケージの中にクラスフレーム自身を構築する必要は有るし、それをNewtonScriptコードからアクセスできるようにする必要はあるが、こうしたことは、クラスを直接アプリケーションのメインビュー内のスロットに置くことで一度に出来る。上記の例で言えば、`Stack`というスロットをメインビューに追加し、その内容をブラウザにそのまま記述しておけばよい。(`Stack`への代入記号 `:=` は書かなくて良いが)

好みによっては、これを`ProjectData`に(今度は代入記号付きで)書いておけば、グローバルなコンパイル時変数としてクラスフレームを作ることが出来る。これは、実行時には使用できないので、メインビューに`Stack`スロットを作ってやる必要はあるが、その値として、ブラウザに`Stack`と打ってやるだけで

ある。スーパークラス(後で詳しく述べる)を使いたいなら、ProjectData にクラスを置かないといけない。

クラス変数

クラスフレームにスロットを追加することで、全てのクラスのインスタンスから共有可能な変数であるクラス変数を作ることもできる。これは、サブビューから共有できる変数をビューに追加するのと同じ事だ。しかし、それはビューシステムがビューのために自動的にやることを、プログラマがクラスのために手作業でやらないといけないのでややトリッキーな所がある。

クラスフレームや、コンパイル時に構築されるデータは、アプリケーションが動作しているときには読み込み専用の領域に置かれると言う事を覚えて置いて欲しい。そうしたもののスロットの値を変更するのは不可能だし、クラス変数に新たな値を代入するのは不可能だ。ビューシステムはこの問題をうまく避けるため、ヒープベースのフレームを作り、_proto スロットで、ビューを参照するようにし、そのフレームをビューとして使っている。オリジナルスロットはプロトタイプ継承でアクセスでき、スロットの作成と変更は、ヒープベースのフレームで行って、オリジナルの値をオーバーライドしている。同じトリックがクラスフレームのために使える。

たとえば、初期値が 0 のクラス変数 x が欲しいとする。クラスフレームは、ProjectData ファイルで定義され、x という名のスロットを含む:

```
TheClass := { ... x: 0 ... }
```

ベースビューは、その値が単に TheClass である、TheClass という名のスロットを持つ。アプリケーションが実行を開始した初期の段階で、自分のクラスフレームが定義されたビューの中の viewSetupFormScript に入るだろう、そこで、ヒープベースバージョンのクラスを作り、TheClass 変数にそれを代入すれば良いのである:

```
viewSetupFormScript:
    func () begin
        ...
        if not TheClass._proto exists then
            TheClass := {_proto: TheClass};
        ...
    end
```

こうして、TheClass のメソッド中で、変数 x を使ったり代入したり出来る。インスタンスはその _parent スロットを通して継承される。そして、最初に x への代入が起こると、スロット x はクラスのヒープベースの部分として作られ、初期値 0 は隠される。この操作は一度だけでよいことに注意 – あるいはフレームのチェーンを片付け、アプリケーションが開く度に行うというのもありである。上記の様に _proto スロットをチェックするのは、このこと(継続して存在しているのか、あるいは前回消されたのか)を確かめるためで、別の方法としては、メインビューの viewQuitScript 内で TheClass := TheClass._proto をセットするというのもある。

訳註: TheClass := TheClass._proto によって、TheClass の内容は変更不可能なものとなり、毎回調べる必要がなくなるという事か?

スーパークラス

スーパークラスに似たものを得るのはたやすい。単に、_proto スロットが、スーパークラスを指すようにすればいいだけである。そのためには、クラス定義を ProjectData で行い、その名前がコンパイル時に有効になるようにしないといけない。たとえば、SortedList クラスが Collection をスーパークラスに持つようにしてみる:

```
Collection := { ... };
SortedList := {_proto: Collection, ...};
```

もちろん、Collection は、SortedList への _proto セット前に定義して

おかないといけない。これの順序を何らかの理由で逆にするなら、_proto スロットは後から追加しないといけない:

```
SortedList := { ... };
Collection := { ... };
SortedList._proto := Collection;
```

サブクラスのメソッドをオーバーライドするなら、予約語 inherited を使ってスーパークラスの方のメソッドを使うと良い。もし、クラス変数があるなら、代入が、_proto チェーンの最も遠く(つまり、各クラスの初期化時に作成したラッパーで)で起こるため、各クラスは自分自身のバージョンのクラス変数を得ることに注意したい。

スープエントリをカプセル化するためにクラスを使う

クラスの利用法の一つとして、スープエントリのカプセル化が考えられる。(スープとスープエントリについては、*Newton Programmer's Guide* 参照)

通常、スープ内のエントリは、継承動作のための _parent や _proto スロットを持たないデータレコードである。理由は明白で、そんなことをするためには各エントリが継承されたフレームのコピーを持たなければならなくなるからだ。(実際、_proto スロットはスープエントリには全く入らない。これは様々な理由による)。よって、スープエントリは、通常オブジェクト指向な使い方はされない。

残念ながら、スープエントリの値は通常、要求されたスロットの集合というような複雑な要求を受けているので、それらにオブジェクトインターフェースを与えることが望ましい。クラスを個々のスープエントリのために「ラッパー」として定義することでこれが出来る。New メソッドはさらに、スープからエントリを集め、存在するエントリからオブジェクトを作るクラスメソッドや、エントリ削除、変更取消、変更保存のためのインスタンスメソッドを持つこともできる。各インスタンスは自身のスープエントリを参照するスロットを持つ。

そうしたラッパークラスがあれば、スープとその内容をオブジェクトとして扱え、データを変更したり集めたりするためのメッセージ送信をエントリに対して行える。クラスはそうして、スープ内のエントリへの要求を実装するコードの中心的存在になる。

ROM インスタンスプロトタイプ

もし、インスタンスが非常に複雑で、値を変更されたくない多くのスロットを持っているなら、ビューとクラスにおいて使ったのと同様な _proto トリックを使うことでメモリ空間を節約できる。そのトリックとは、New メソッドにおいて、クラスと、インスタンス変数プロトタイプ(それはクォートされて、アプリケーション領域に存在する)を参照する小さなフレームとしてインスタンスを作るというものである:

```
New: func ()
       {_parent: self,
        _proto: '{ ...initial instance vars... }}
```

インスタンスを背後に置く

アプリケーションには沢山のものが含まれており、インスタンスを、メモリカードやアプリケーション(これらは容易に抜いたり削除できる)より長く生存させるのは難しい。これをやる唯一の方法は、全てのクラス(とスーパークラスがあればそれも)を、ヒープ中にコピーすることであるが、これは現実には大量のスペースを消費するだろう。

最後に

このテクニックは、実際に存在するあらゆるクラスベースオブジェクトシステムを正確にシミュレートできるものではないが、クラスによく似たデータ構造の中に自分のデータ型をしまい込む為の方法を与えてくれる。これは非常に便

利だと思うし(stores, soups, cursors 等のこのモデルに従う全てのものにとって), もっとよい Newton アプリケーションを作る手助けになればと思う。楽しいプログラミングを!

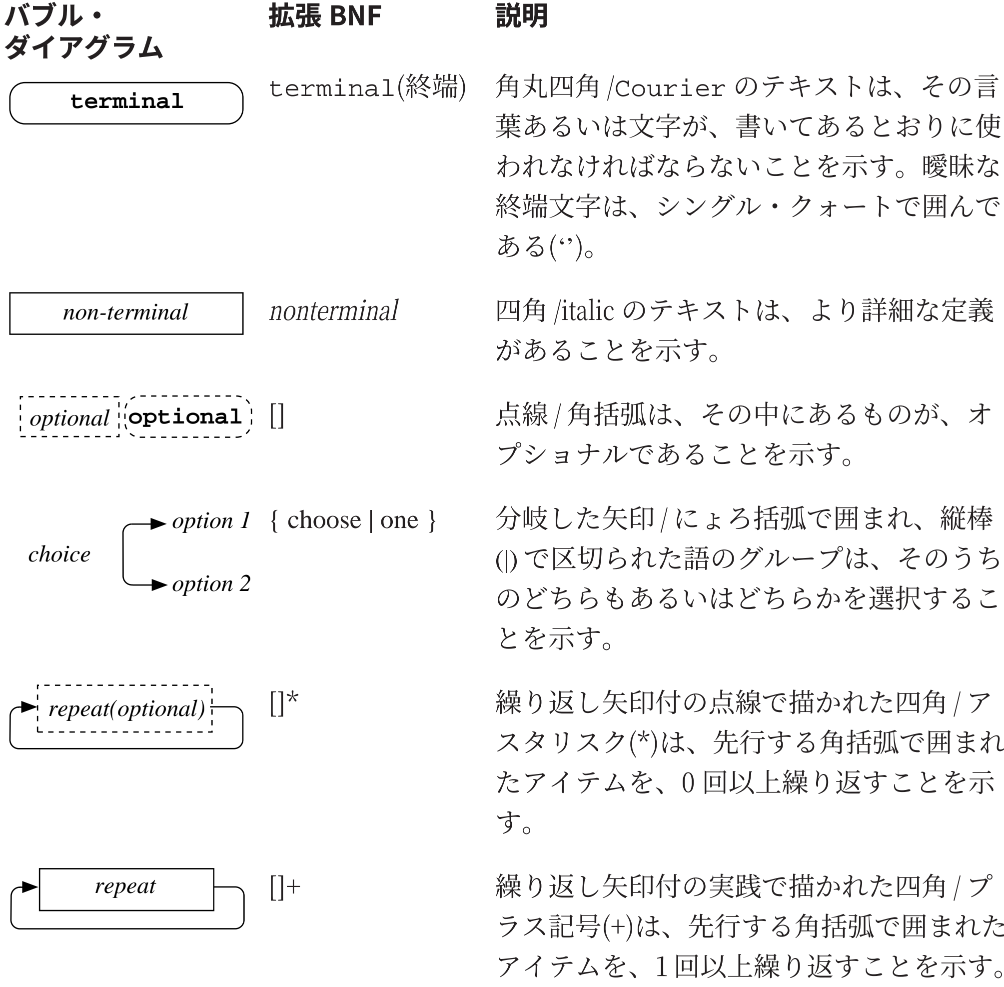
作者について

Walter Smith は Newton グループに 1988 年に参加した。彼は、NewtonScirpt と、Newton オブジェクトストアの第一級のデザイナであり、実装者である◆

APPENDIX D

NewtonScript 構文定義

このドキュメントでは定義を拡張 BNF と、Pascal 構文チャートで示す。それぞれ以下のように定義される。



バブル・ダイアグラム	拡張 BNF	説明
`terminal`	`terminal`(終端)	角丸四角 /`Courier` のテキストは、その言葉あるいは文字が、書いてあるとおりに使われなければならないことを示す。曖昧な終端文字は、シングル・クォートで囲んである(‘’)。
non-terminal	*nonterminal*	四角 /italic のテキストは、より詳細な定義があることを示す。
optional `optional`	[]	点線 / 角括弧は、その中にあるものが、オプショナルであることを示す。
choice *option 1* *option 2*	{ choose | one }	分岐した矢印 / にょろ括弧で囲まれ、縦棒(|)で区切られた語のグループは、そのうちのどちらもあるいはどちらかを選択することを示す。
repeat(optional)	[]*	繰り返し矢印付の点線で描かれた四角 / アスタリスク(*)は、先行する角括弧で囲まれたアイテムを、0 回以上繰り返すことを示す。
repeat	[]+	繰り返し矢印付の実践で描かれた四角 / プラス記号(+)は、先行する角括弧で囲まれたアイテムを、1 回以上繰り返すことを示す。

文法について

文法を、構文文法、字句文法に分けて示す。構文文法において、空白は特に意味を持たない。空白は、ただのスペース、タブ、リターン、改行を指す。コメントは空白の一部である。コメントは任意の文字を `/*` と `*/` の間に入れ、入れ子なしで構成するか、`//` と改行またはリターンの間に入れて構成する。

字句文法において、非終端はトークンでなく文字であり、空白は意味を持つ。

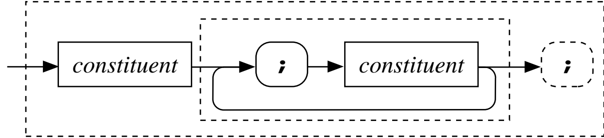
言語のほとんどの部分が式で構成されており、式で終わっている多くのものは曖昧である。曖昧な場合は、式を可能な限り拡張して解消しようとする。たとえば、`while true do 2 + 2`は`(while true do 2) + 2`でなく、`while true do (2+2)`と解釈される。このルールに影響されるのは、関数コンストラクタ・代入・繰り返し・if 式・break 式・try 式・初期化節・return 式・グローバル関数定義である。(function-constructor, assignment, iteration, if-expression, break-expression, try-expression, initialization-clause, return-expression, and global-function-decl)

訳注: 本書では文法の記述については日本語で書いていない。lex や yacc を使ってなにかをやるには、その方が楽だからだ。

構文文法

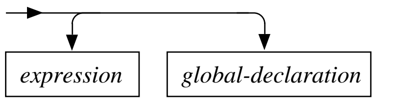
input:

[*constituent* [; *constituent*]* [;]]



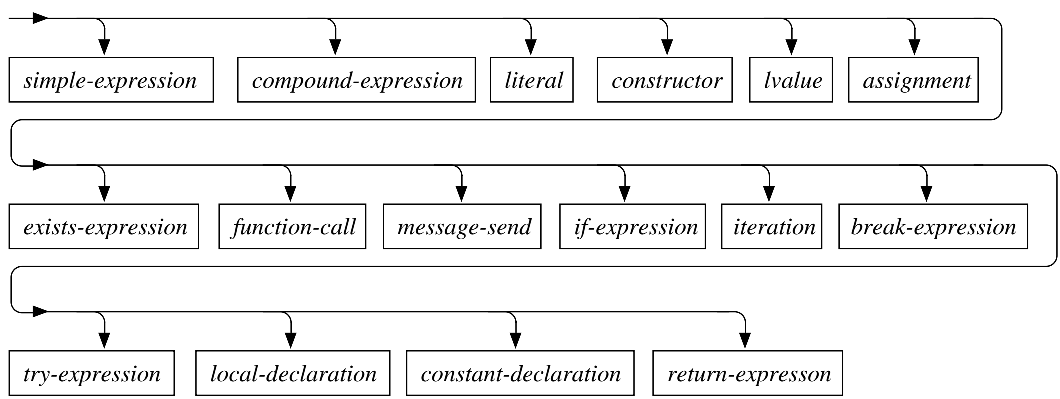
constituent:

{ *expression* | *global-declaration* }



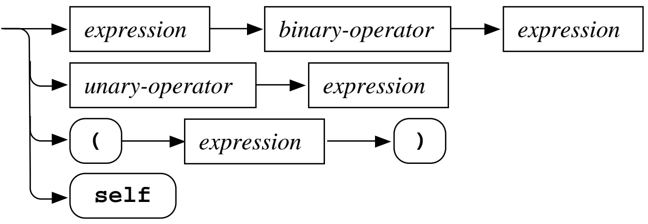
expression:

{ *simple-expression* | *compound-expression* | *literal* | *constructor* | *lvalue* | *assignment* | *exists-expression* | *function-call* | *message-send* | *if-expression* | *iteration* | *break-expression* | *try-expression* | *local-declaration* | *constant-declaration* | *return-expression* }



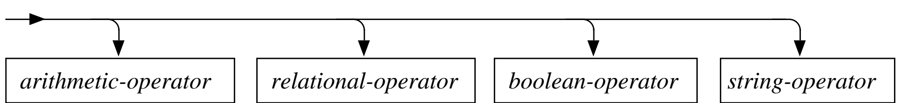
simple-expression:

{ *expression binary-operator expression* / *unary-operator expression* | (*expression*) | `self` }



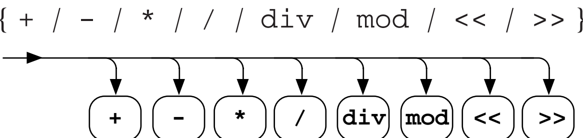
binary-operator:

{*arithmetic-operator* | *relational-operator* | *boolean-operator* | *string-operator* }



arithmetic-operator:

{ + / - / * / / / div / mod / << / >> }



relational-operator:

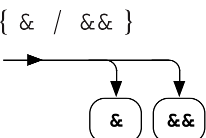
{ = / <> / < / > / <= / >= }

boolean-operator:

{ and / or }

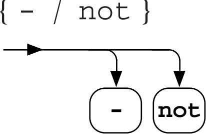
string-operator:

{ & / && }



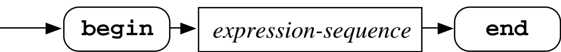
unary-operator:

{ - / not }



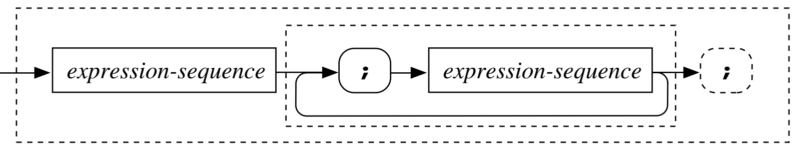
compound-expression:

`begin` *expression-sequence* `end`



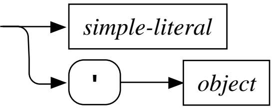
expression-sequence:

[*expression* [`;` *expression*]* [`;`]]



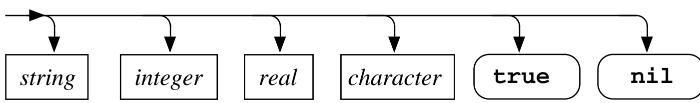
literal:

{ *simple-literal* | `'` *object* }



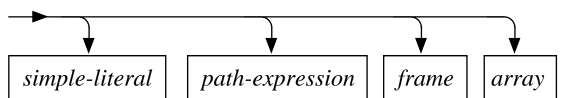
simple-literal:

{ *string* / *integer* / *real* / *character* | `true` / `nil` }



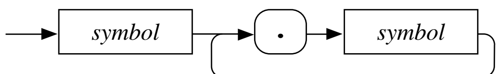
object:

{ *simple-literal* / *path-expression* / *array* / *frame* }



path-expression:

symbol [. *symbol*]+

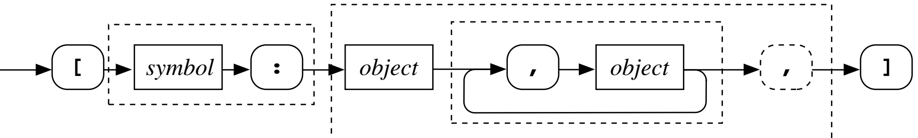


注意

' *symbol*.*symbol* ... の各ドットは曖昧である: これはパス式あるいはスロットアクセッサの継続したものと考えられる。
NewtonScript は、'x.y.z を長い一つの式と考え、('x).y.z とは考えない◆

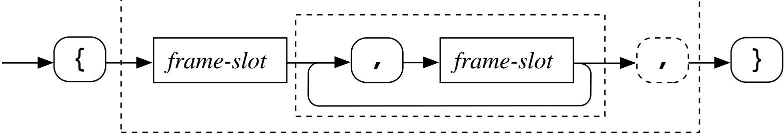
array:

'[' [*symbol* :] [*object* [, *object*]* [,]] ']'



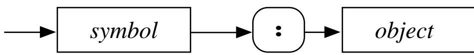
frame:

'{' [*frame-slot* [, *frame-slot*]* [,]] '}'



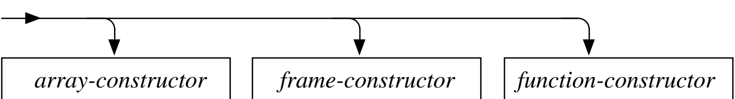
frame-slot:

symbol : *object*



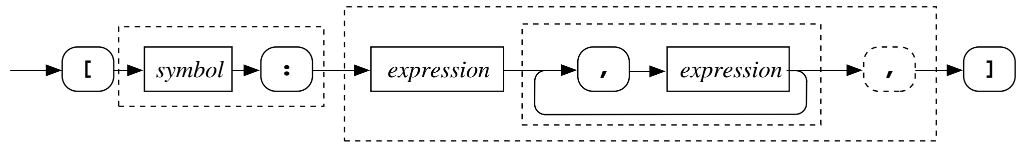
constructor:

{ *array-constructor* | *frame-constructor* |*function-constructor* }



array-constructor:

'[' [*symbol* :] [*expression* [, *expression*]* [,]] ']'

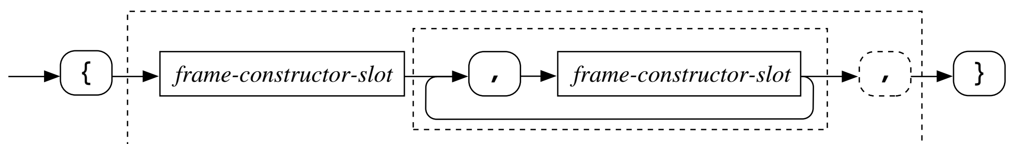


注意

‘[’ *symbol* : *symbol* (... は曖昧である: 最初のシンボルは配列のクラスまたは、メッセージのレシーバにも見えるが、NewtonScript では最初の方の解釈を取る◆

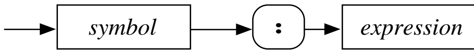
frame-constructor:

‘{’ [*frame-constructor-slot* [, *frame-constructor-slot*]* [,]] ‘}’



frame-constructor-slot:

symbol : *expression*

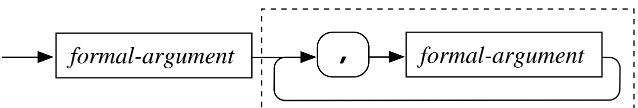


function-constructor:

`func` [`native`] ([*formal-argument-list*]) *expression*

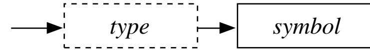
formal-argument-list:

formal-argument [, *formal-argument*]*



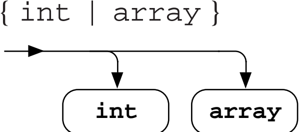
formal-argument:

[*type*] *symbol*



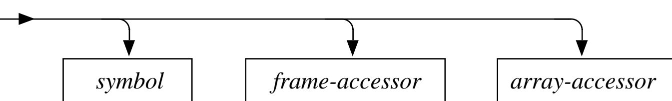
type:

{ `int` | `array` }



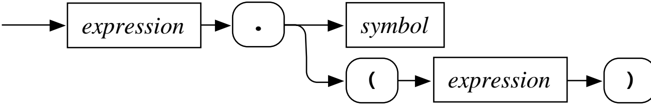
lvalue:

{ *symbol* | *frame-accessor* | *array-accessor* }



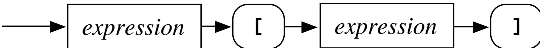
frame-accessor:

expression `.` { *symbol* | `(` *expression* `)` }



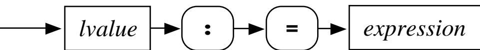
array-accessor:

expression '`[`' *expression* '`]`'



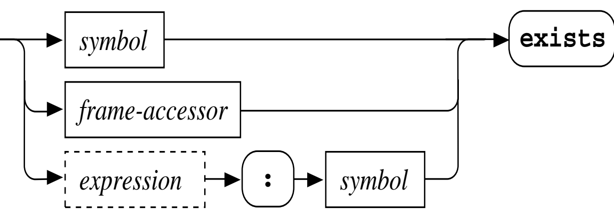
assignment:

lvalue `:=` *expression*



exists-expression:

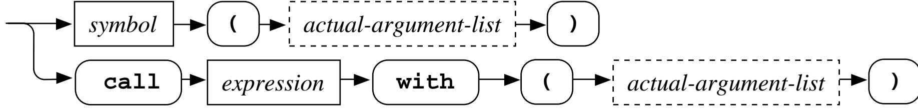
{ *symbol* | *frame-accessor* | [*expression*] `:` *symbol* } `exists`



function-call:

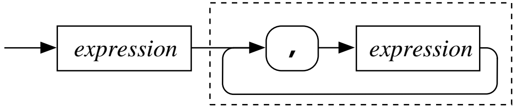
{ *symbol* ([*actual-argument-list*]) |

`call` *expression* `with` ([*actual-argument-list*]) }



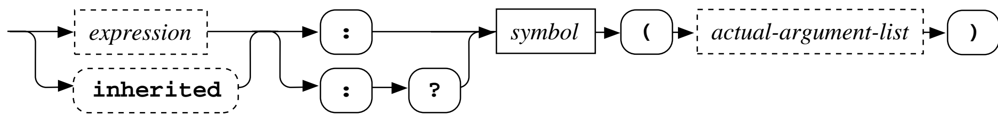
actual-argument-list:

expression [, *expression*]*



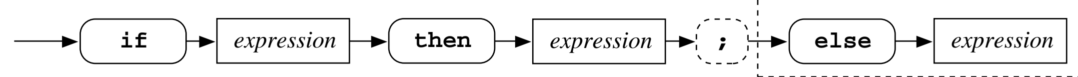
message-send:

[{ *expression* | `inherited` }] { `:` | `:?` } *symbol* ([*actual-argument-list*])



if-expression:

`if` *expression* `then` *expression* [;] [`else` *expression*]

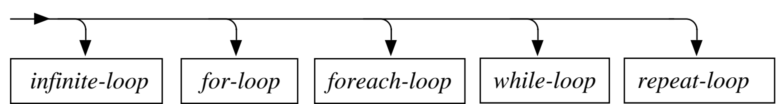


注意

`else` 節はもっとも近い、`else` 節を持たない `then` 節に接続される

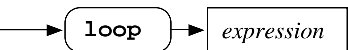
iteration:

{ *infinite-loop* | *for-loop* | *foreach-loop* | *while-loop* | *repeat-loop* }



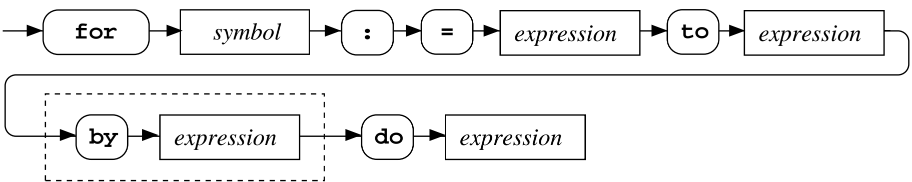
infinite-loop:

`loop` *expression*



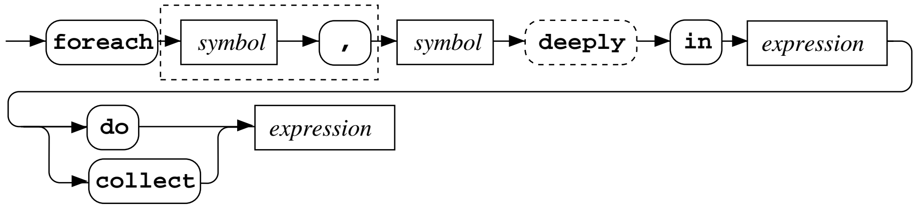
for-loop:

`for` *symbol* `:=` *expression* `to` *expression* [`by` *expression*] `do` *expression*



foreach-loop:

`foreach` *symbol* [`,` *symbol*] [`deeply`] `in` *expression* { `do` | `collect` } *expression*

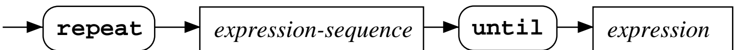


while-loop:

`while` *expression* `do` *expression*

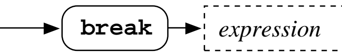
repeat-loop:

`repeat` *expression-sequence* `until` *expression*



break-expression:

`break` [*expression*]

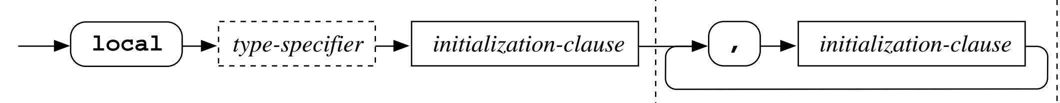


try-expression:

`try` *expression-sequence* [`onexception` *symbol* `do` *expression* [`;`]]+

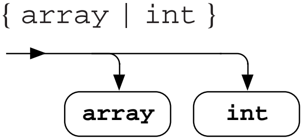
local-declaration:

`local` [*type-specifier*] *initalization-clause* [`,` *initalization-clause*]*



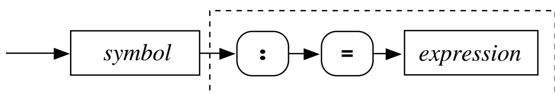
type-specifier:

{ `array` | `int` }



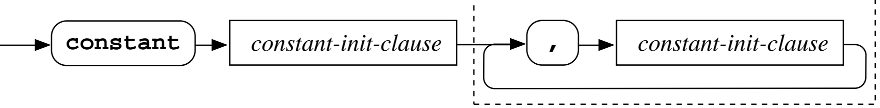
initialization-clause:

symbol [`:=` *expression*]



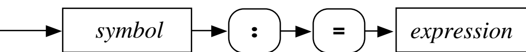
constant-declaration:

`constant` *constant-init-clause* [`,` *constant-init-clause*]*



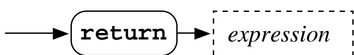
constant-init-clause:

symbol := *expression*



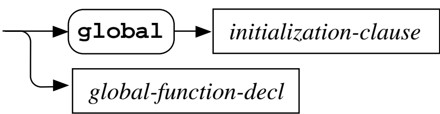
return-expression:

`return` [*expression*]



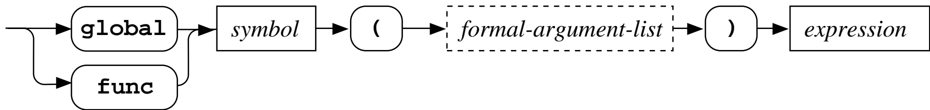
global-declaration:

{ `global` *initialization-clause* | *global-function-decl* }



global-function-decl:

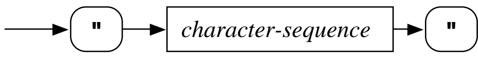
{ `global` | `func` } *symbol* ([*formal-argument-list*]) *expression*



字句文法

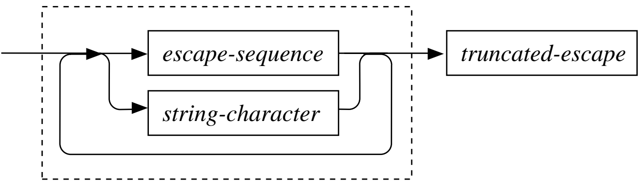
string:

" *character-sequence* "



character-sequence:

[{ *string-character* | *escape-sequence* }]* [*truncated-escape*]

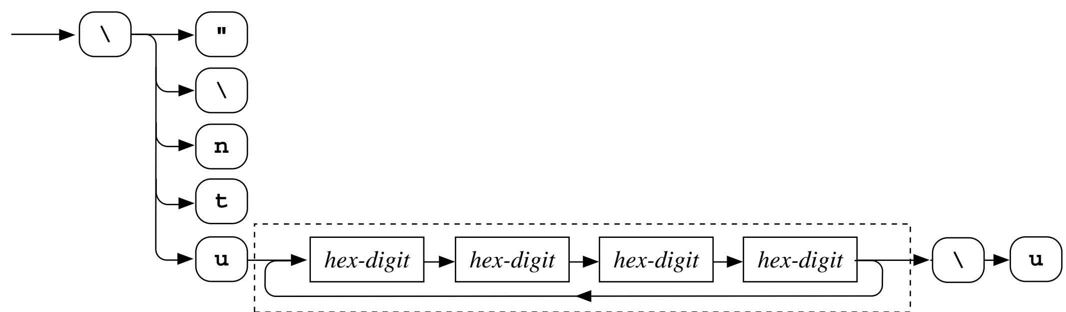


string-character:

<`tab` or any ASCII character with code 32-127 except '`"`' or '\'>

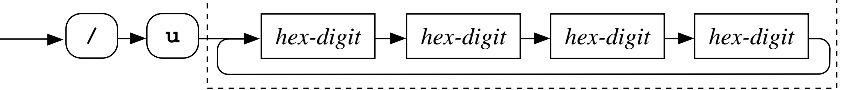
escape-sequence:

{ \ { `"` | `\` | `n` | `t` } | \ `u` [*hex-digit hex-digit hex-digit hex-digit*]* \ `u` }



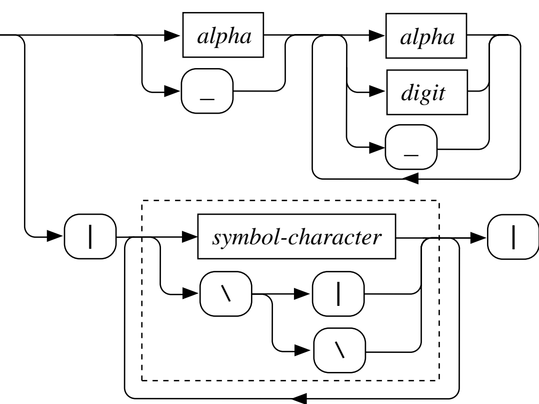
truncated-escape:

\ `u` [*hex-digit hex-digit hex-digit hex-digit*]*



symbol:

{ { *alpha* | _ } [{ *alpha* | *digit* | _ }]* |
'|' [{ *symbol-character* | \ { '|' | \ }]* '|' }



注意

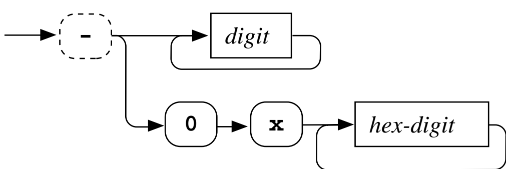
予約語は、非終端シンボルからは除外される◆

symbol-character:

<any ASCII character with code 32-127 except '|' or '\'>

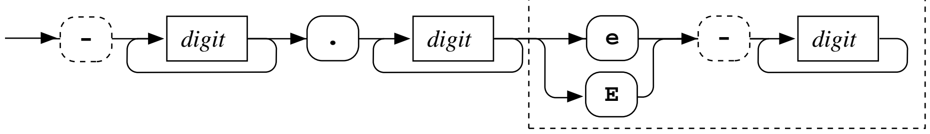
integer:

[-] { [*digit*]+ | 0x [*hex-digit*]+ }



real:

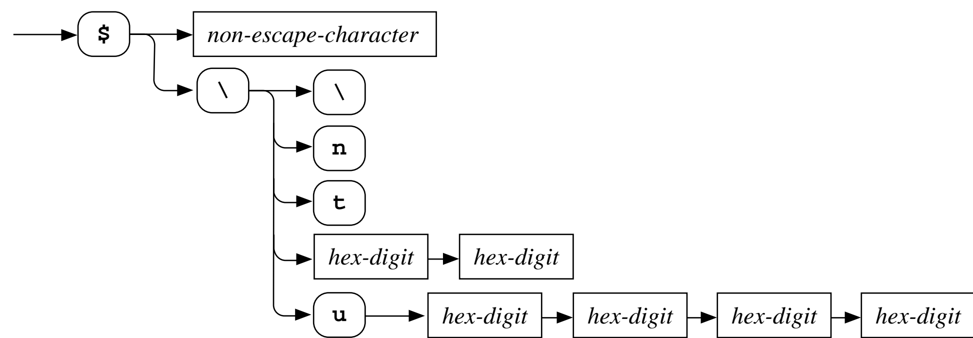
[-] [*digit*]+ . [*digit*]* [{ e | E } [-] [*digit*]+]



character:

$ { *non-escape-character* |

\ { \ | n | t | *hex-digit hex-digit* | u *hex-digit hex-digit hex-digit hex-digit* } }



non-escape-character:

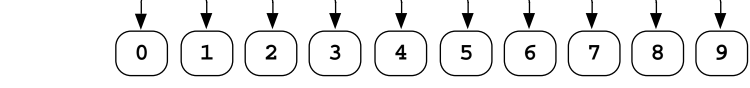
<any ASCII character with code 32-127 except '\'>

alpha :

<A-Z and a-z>

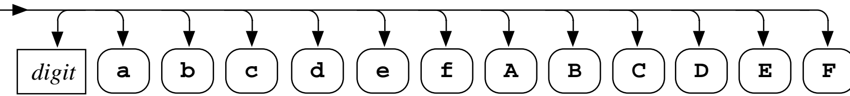
digit:

{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }



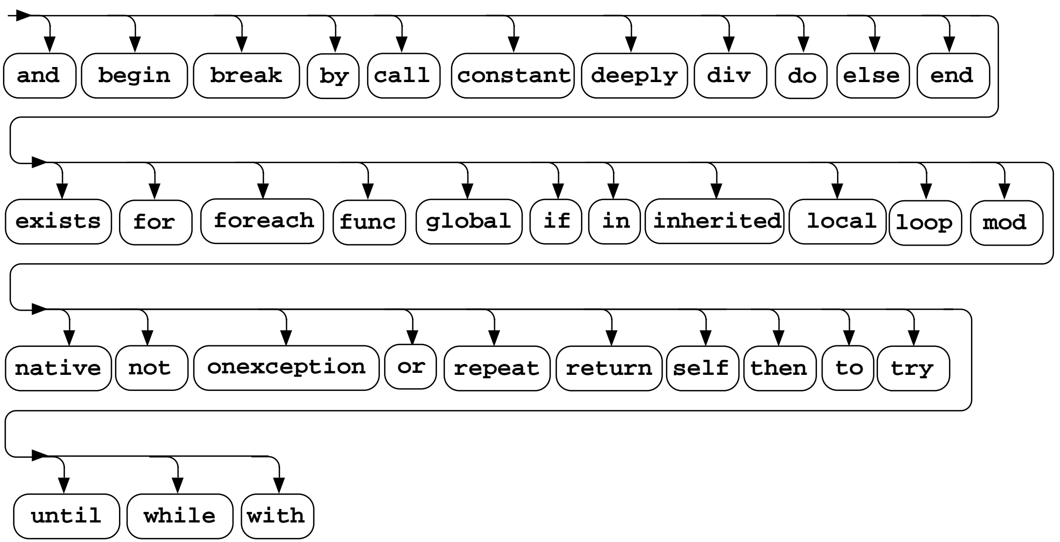
hex-digit:

{ *digit* | a | b | c | d | e | f | A | B | C | D | E | F }



reserved-word:

```
{ and|begin|break|by|call|constant|deeply|div|do|
else|end|exists|for|foreach|func|global|if|in|
inherited|local|loop|mod|native|not|onexception|
or|repeat|return|self|then|to |try|until|while|
with }
```



演算子の優先順位

演算子の優先順位については、page 2-39 の、表 2-5 参照。

APPENDIX E

クイックリファレンス

アクセス形態	プロトタイプ継承	ペアレント継承
slot	○	○
frame.*slot*	○	―
frame.（パス式）	○	―
GetVariable(*frame*, *slot*)	○	○
GetSlot(*frame*, *slot*)	―	―
frame:*message*()	○	○
frame:?*message*()	○	○
:*message*()	○	○
inherited:*message*()	○	―
inherited:?*message*()	○	―
symbol exists	○	○
frame.*slot* exists	○	―
frame.（パス式）exists	○	―
frame:*message* exists	○	○
:*message* exists	○	○
HasVariable(*frame*, *slot*)	○	○
HasSlot(*frame*, *slot*)	―	―

○: 使用　　ー: 不使用

演算子	意味	結合方向
.	スロットアクセス	→
: :?	(条件付)メッセージ送信	→
[ ]	配列要素	→
-	単項マイナス	→
<< >>	左シフト 右シフト	→
* / div mod	乗算、除算、整数除算、余り	→
+ -	加算、減算	→
& &&	文字列合成、文字列スペース入り合成	→
exists	変数・スロットの存在確認	なし
< <= > >= = <>	比較	→
not	論理否定	→
and or	論理 AND, 論理 OR	→
:=	代入	←

優先度は、上から下に向かって低くなる。

同じ行にある演算子同士の優先順位は同じ。

式	戻り値
begin *exprList* end	*exprList* 中の最後の式の値
return [*expr*]	nil または、*expr*
if *expr* then *expr*	*expr* または、nil
if *expr* then *expr1* else *expr2*	*expr1* または *expr2*
loop *expr*	break の値
for *var* := *expr1* to expr2 do *expr*	nil または、break の値
for *var* := *expr1* to expr2 by *step* do *expr*	
foreach [*slot*,] *val* in *frameOrArray* do *expr*	nil または、break の値
foreach [*slot*,] *val* deeply in *frame* do *expr*	(*deeply* は、_proto スロットも追跡する)
foreach [*slot*,] *val* in *frameOrArray* collect *expr*	*expr* の値を集めた配列、または break の値
foreach [*slot*,] *val* deeply in *frame* collect *expr*	(*deeply* は、_proto スロットも追跡する)
while *expr* do *expr*	nil または、break の値
repeat *exprList* until *expr*	nil または、break の値
break [*expr*]	nil または、*expr*
call *functionObject* with (*argList*)	*functionObject* が返す値
try *exprList* onexception *exceptionSymbol* do *expr*...	*exprList* 中で、最後に実行された *expr* または、実行された onexception の値
expr	Throw(*exSym*, *datum*), CurrentException(), Rethrow()
|evt.ex| { name: *exceptionSymbol*,	error: *integer* }
|evt.ex.msg| { name: *exceptionSymbol*,	message: *string* }
|evt.ex; type.ref| { name: *exceptionSymbol*,	data: *datum* }

<table>
<tr><th>型</th><th>例</th><th>コメント</th><th>SPriontObject</th><th>ClassOf</th><th>PrimClassOf</th></tr>
<tr><td>integer</td><td>42, 0x5BA6, -99</td><td>$-2^{29}...2^{29}-1$
(–536870911...+536870912)</td><td>10 進数</td><td>Int</td><td>Immediate</td></tr>
<tr><td>real</td><td>1000.02, -3.14,
1.0e5,1.e-12</td><td>SANE 倍精度 , 15-16 桁 ,
指数部: -308...308</td><td>1,000.02</td><td>real</td><td>Binary</td></tr>
<tr><td>boolean</td><td>nil
true</td><td>クオートしないように</td><td>null 文字列</td><td>Weird_Immediate
Boolean</td><td>Immediate</td></tr>
<tr><td>character</td><td>$a, $7, $\\,
$\F0, $\uF7FF</td><td>$\xx は、16 進 ,
$\uxxxx は、Unicode 16 進</td><td>一文字の文字列</td><td>Char</td><td>Immediate</td></tr>
<tr><td>string</td><td>"abc", "\n", "\t"
"\"", "\\"</td><td>abc, 改行 , タブ
二重引用符、バックスラッシュ</td><td>文字列</td><td>String</td><td>Binary</td></tr>
<tr><td>symbol</td><td>hiho, baz_1,

|evt.ex.msg|</td><td>254 文字まで
[a-z, A-Z, 0-9] が使用可
縦棒を使えば、どんな文字でも OK</td><td>シンボル名</td><td>Symbol</td><td>Binary</td></tr>
<tr><td>array</td><td>[arrayClass: e1, e2, e3]</td><td>クラスはオプショナル
コンマで終わってもよい</td><td>null 文字列</td><td>array class or Array</td><td>Array</td></tr>
<tr><td>frame</td><td>{ slot1: val1, slot2: val2}</td><td>コンマで終わってもよい</td><td>null 文字列</td><td>class slot or Frame</td><td>Frame</td></tr>
<tr><td>function object</td><td>func(arglist) expr</td><td>現在の環境を保持する</td><td>null 文字列</td><td>CodeBlock</td><td>Frame</td></tr>
</table>

用語集

Array **配列**

番号付きで順序付けられたオブジェクトを格納するスロット(配列要素とも言う)の並び。最初の要素の添え字は 0 である。他の非イミディエイトオブジェクト同様､配列はユーザー定義のクラスを持つことができ、その長さは動的に変更できる。

Binary object **バイナリオブジェクト**

いろいろな種類のデータを表現できるバイトの並び。それはまたユーザー定義クラスを持つことができ、サイズも動的に変更できる。バイナリオブジェクトの例は、文字列、実数、サウンド、ビットマップなどである。

Boolean **論理型**

`true` と呼ばれる種類のイミディエイト値。関数と制御構文は `nil` を `false` として使い、それ以外のものは全部 `true` である。何もなければ `true` を使う。

Child **子供**

`_parent` スロットを通して別のフレームを参照するフレーム。

Class **クラス**

オブジェクトが参照する値を示すシンボル。配列・フレーム・バイナリオブジェクトはユーザー定義型を持つことが出来る。

Constant **定数**

変更されない値。NewtonScript では、式の中でその定数が使われたときに実際の値に置換される。

Frame　フレーム

順序のない、スロットの集合。スロットは、名前と値のペアからなる。スロットの値は、どのタイプのオブジェクトでもよい。スロットは動的にフレームに足したり消したり出来る。フレームはユーザー定義クラスを持つこともできる。フレームは Pascal の record や、C の struct のようなものだが、メッセージに対応するオブジェクトとして働く。

Function object　関数オブジェクト

関数オブジェクトは、関数コンストラクタで作られる:

`func(`*args*`)` *funcBody*

実行可能な関数オブジェクトはコード同様その字句とメッセージの環境の値を持つ。この情報は関数コンストラクタが実行時に評価されたときに取り込まれる。

Global　グローバル変数

NewtonScript のコードのどこからでもアクセスできる変数。

Immediate　イミディエイト　即値

間接参照でヒープ中のオブジェクトを参照するのでなく、直接格納される値。イミディエイトは、文字・整数・論理型である。**参照型 (Reference)**も見ること。

Implementor　インプリメンタ(実装者)

メソッドが定義されているフレーム。**レシーバ**も見ること。

Inheritance　継承

スロットやデータなどの属性、そしてメソッドなどの動作をオブジェクトから利用可能にする機構。ペアレント継承は異なるタイプのビューにデータやメソッドの共有を可能にする。プロトタイプ継

承は、他のテンプレートやプロトタイプで定義されている基本定義を継承することを可能にする。

Local **ローカル変数**

それが定義された関数の中にスコープが限定される変数。関数内でローカル変数を明示的に生成する場合は、予約語 `local` を使わないといけない。

Message **メッセージ**

シンボルと引数リスト。メッセージは *frame*:*MessageName*`()` というメッセージ送信構文で送られ、レシーバーである *frame* に *MessageName* というメッセージが送られる。

Method **メソッド**

メッセージに対応して起動される、フレームスロットの中の関数。

Object **オブジェクト**

イミディエイト・配列・フレーム・バイナリオブジェクトなどの型を持つデータの小片。NewtonScript では、フレームだけがメソッドを持ち、メッセージを受け取ることが出来る。

Parent **親**

他のフレームの `_parent` スロットを通して参照できるフレーム。

Path expression **パス式**

配列やフレームへのアクセスパスを閉じこめたオブジェクト。

Proto **プロトタイプ**

別のフレームの `_proto` スロットを通して参照できるフレーム。

Receiver **レシーバ**

メッセージを送られたフレーム。関数オブジェクトの起動に関するレシーバは、疑似変数 `self` を通してわかる。

Reference **参照**
間接的に配列、フレーム、バイナリオブジェクトなどを見に行く値。**イミディエイト**参照。

Self **セルフ**
現在のレシーバを示す疑似変数

Slot **スロット**
イミディエイトや参照値を格納できるフレームや配列の要素

EPILOGUE

訳者あとがき

本書は、訳者がこっそり訳してこっそり配る、アンダーグラウンドなものとして存在する予定だった。しかし、縁有って、公式に配布可能となったものである。

訳者がこの作業を始めたのは1996年の10月頃だったから、作業開始からもう半年以上も経ったことになる。その間、Appleをめぐるいいニュースもあれば、悪いニュースもあったが、何にせよこれほどユーザーをはらはらさせる会社は珍しいだろうと思う。訳者としては、本書が無事世に出てくれたらと、切に願うものである。

話しは変わるが、訳を進めていく過程で、所どころ登場する間違った記述には閉口した。Apple程の大会社のプログラミング言語のマニュアルに間違った内容が記述され、それが堂々と数年間に渡って配布され続けているとはどういうことなのだろう？ USのNewtonチームは真摯な仕事をしているのだろうか？自らの創造物に愛情を持っているのだろうか？現在Newtonが立たされている苦境も、こういう細部をないがしろにしてきたことに起因しているのではないか？と、彼等に問いかけておきたいと思う。まあ、彼等がこれを読むことはないだろうが。

さて、ここで、本書作成に当たって、大変お世話になったかたがたにお礼を言っておきたい。

まず、厚見献志氏(URL: http://www.st.rim.or.jp/ atumi/)には大変お世話になった。訳者が本書を訳していることを知り、アップルコンピュータ株式会社に、訳者を紹介してくれた。彼がいなければ本書は存在しなかっただろうと思う。

そして、アップルコンピュータ株式会社(以降AJと書く)の方々には大変お世話になった:

AJ内田氏、姫崎氏。彼等は、初期の段階で原稿の評価、U.S.とのコンタクトを行ってくれた。そしてAJ大宮氏。彼は私の作業を高く評価してくれ、様々な形で私をバックアップしてくれた。とても感謝している。それから何といってもAJ池上氏にはとてもお世話になった。彼は私のひどい訳の全てに目を通してくれ、適切なアドバイスをしてくれるとともに、U.S.との交渉を精力的に行ってくれた。彼がいなくても、本書は存在できなかっただろうと思う。聞く所によると、相当な長髪らしい。

また、私の友人、Randolph Cook氏にも感謝したい。彼はテキサス生まれのアメリカ人である。本書の中でどうしても意味がわからないいくつかの文章は、彼の助言により理解することができた。彼はまた、素晴しい人格者であり、良き父親でもある。

一般公開に先立つNewton Japanベータテストにおいてご協力いただいた飯沼哲也氏、佐藤「二八」耕氏に感謝したい。ベータテストは諸般の事情であまり長期間行えなかったが、岡部昇氏、神田鉄平氏、堤修一氏、今里泰山氏からアドバイス、激励の言葉をいただいたことを感謝したい。(氏名登場順不同)

本書の翻訳開始から私を励まし、発表の場を提供してくれた河口ojin忠志氏には特に感謝したい。

そして最後に私の妻と息子に感謝したい。彼等は、私の最大のサポータである。

1997/6/28　齋藤匡弘

訳者紹介

齋藤匡弘(さいとうくにひろ)、1964年大阪市に生まれる。学生時代からコンピュータにのめり込み、日本橋で初代Macintoshを見て以来、熱狂的Macユーザーになる。その後、UNIX/Windows/COBOLプログラマとなる。数年前にNewtonに出会って、現在Newtonプログラマになるための修行中。NewtonJapan会員番号64番。

e-mail:brown@ga2.so-net.or.jp, URL:http://www.k-inet.com/ brown/

索引

A

Abs() 6-49
Acos() 6-54
Acosh() 6-54
ADC xv
AddArraySlot () 6-26
and 2-34
Annuity() 6-76
APDA xv
Apply() 4-6, 6-80
Array() 6-27
ArrayInsert() 6-27
ArrayMunger() 6-28
ArrayRemoveCount() 6-29
Asin() 6-54
Asinh() 6-54
Atan() 6-55
Atan2() 6-55
Atanh() 6-55

B

Band() 6-26
BDelete() 6-40
BDifference() 6-40
begin 3-1
BeginsWith() 6-18
BFetch() 6-41
BFetchRight() 6-41
BFind() 6-42
BFindRight() 6-43
BinaryMunger() 6-96
BinEqual() 6-95
BInsert() 6-43
BInsertRight() 6-45
BIntersect() 6-46
BMerge() 6-46
Bnot() 6-26
Boolean 2-2
Bor() 6-26
break 3-6, 3-14
BSearchLeft() 6-47
BSearchRight() 6-48
Bxor() 6-26

C

call by reference 4-8
call by value 4-8
Call With 構文 4-6
Capitalize() 6-18
CapitalizeWords() 6-18
Ceiling() 6-49
Char 2-2
character 2-9
CharPos() 6-19

Chr() 6-96
class 2-19
ClassOf() 2-1, 6-5
Clone() 6-6
Compile() 6-97
Compound() 6-77
constant 2-27
Constant 予約語 2-27
CopySign() 6-56
Cos() 6-56
Cosh() 6-56
CurrentException() 3-18, 6-79

D

DeepClone() 6-7
deeply 3-7
deeply オプションの例 3-10
DefGlobalFn() 6-93
DefGlobalVar() 6-94
Divide by Zero 6-53
Downcase() 6-19

E

else 節 3-3
end 3-1
EndsWith() 6-19
Erf() 6-57
Erfc() 6-57
Exists 2-36
Exp() 6-57
Expm1() 6-58
ExtractByte() 6-83
ExtractBytes() 6-83
ExtractChar() 6-84
ExtractCString() 6-86
ExtractLong() 6-84
ExtractPString() 6-86
ExtractUniChar() 6-86
ExtractWord() 6-85
ExtractXLong() 6-85

F

Fabs() 6-58
false 2-11
FDim() 6-58
fe_All_Except 6-71
fe_Dfl_Env 6-72
fe_DivByZero 6-71
fe_Inexact 6-71
fe_Invalid 6-71
fe_Overflow 6-71
fe_Underflow 6-71
FeClearExcept() 6-72
FeGetEnv() 6-72
FeGetExcept() 6-73
FeHoldExcept() 6-73
FeRaiseExcept() 6-73
FeSetEnv() 6-73
FeSetExcept() 6-74
FeTestExcept() 6-74
FeUpdateEnv() 6-74
Floor() 6-49
FMax() 6-59
FMin() 6-59
Fmod() 6-60

For 3-4
Foreach 3-7
func 4-2

G

Gamma() 6-60
GetFunctionArgCount() 6-7
GetGlobalFn() 6-92
GetGlobalVar() 6-92
GetSlot() 2-21, 6-8
GetVariable() 2-21, 6-8
global 4-6
GlobalFnExists() 6-93
GlobalVarExists() 6-93

H

HasSlot() 2-37, 3-18, 6-8
HasVariable() 6-9
Hypot() 6-60

I

If…Then…Else 3-2
inexact 6-53
inherited 4-4
InsertionSort() 6-29
Int 2-2
Intern() 6-9
invalid 6-53
IsAlphaNumeric() 6-19
IsArray() 2-2, 6-10
IsBinary() 6-10
IsCharacter() 2-2, 6-10
IsFinite() 6-61
IsFrame() 2-2, 6-10
IsFunction() 6-10
IsImmediate() 6-11
IsInstance() 2-4, 6-11
IsInteger() 2-2, 6-11
IsNaN() 6-61
IsNormal() 6-61
IsNumber() 6-11
IsReadOnly() 6-12
IsReal() 2-2, 6-12
IsString() 2-2, 6-13
IsSubclass() 2-4, 6-13
IsSymbol() 2-2, 6-13
IsWhiteSpace() 6-20

K

key パラメタ 6-39

L

latent typing 1-4
Length () 6-30
LessOrGreater() 6-62
LFetch() 6-30
LGamma() 6-62
local 2-24
Log() 6-62
Log10() 6-63
Log1p() 6-63
Logb() 6-63
Loop 3-11
LSB 6-83
LSearch() 6-32

M

MakeBinary() 6-13
Map() 6-14
Max() 6-49
Min() 6-50
MSB 6-83

N

NaN 6-52
native 4-2, 4-16
NearbyInt() 6-64
NewtonScript クラス構造 2-2
NewtonScript 構文定義 D-1
NewWeakArray() 6-32
NextAfterD() 6-64
nil 2-11
not 2-35

O

onexception 3-19
or 2-34
Ord () 6-98
overflow 6-53

P

_parent 2-19, 5-4, GL-3
:Parent() 5-11
pathExpr クラスの配列 2-22
Perform() 6-80
PerformIfDefined() 6-82
Pow() 6-64
PrimClassOf() 2-1, 6-14
_proto 2-19, 5-2, GL-3
ProtoPerform() 6-82
ProtoPerformIfDefined() 6-82

R

Random () 6-50
RandomX() 6-65
Real 2-2
Remainder() 6-65
RemoveSlot() 6-15
RemQuo() 6-65
Repeat 3-13
repeat ループ 3-13
ReplaceObject() 6-15
rethrow 3-21
Rethrow() 3-21, 6-78
Return 4-3
Rint() 6-66
RintToL() 6-66
Round() 6-66

S

Scalb() 6-66
Self 4-9, 4-12, GL-4
SetAdd () 6-33
SetClass() 2-4, 6-16
SetContains() 6-34
SetDifference() 6-34
SetLength () 6-34
SetOverlaps() 6-35
SetRandomSeed () 6-50
SetRemove () 6-36
SetUnion() 6-36

SetVariable() 6-16
SignBit() 6-67
Signum() 6-67
Sin() 6-67
Sinh() 6-68
Sort() 6-37
SPrintObject() 6-20
Sqrt() 6-68
StableSort() 6-38
StrCompare() 6-20
StrConcat() 6-21
StrEqual() 6-21
StrExactCompare() 6-21
String 2-2
StrLen() 6-22
StrMunger() 6-22
StrPos() 6-23
StrTokenize() 6-23
StuffByte() 6-87
StuffChar() 6-88
StuffCString() 6-89
StuffLong() 6-89
StuffPString() 6-90
StuffUniChar() 6-91
StuffWord() 6-91
StyledStrTruncate() 6-24
SubStr() 6-25
Symbol 2-2
SymbolCompareLex() 6-17

T

Tan() 6-68
Tanh() 6-68
test パラメタ 6-39
Throw() 3-20, 6-78
TotalClone() 6-17
TrimString() 6-25
true 2-11
Trunc() 6-69
Try ステートメント 3-19

U

UnDefGlobalFn() 6-95
UnDefGlobalVar() 6-95
underflow 6-53
unicode 2-9
Unordered() 6-69
UnorderedGreaterOrEqual() 6-69
UnorderedLessOrEqual() 6-70
UnorderedOrEqual() 6-70
UnorderedOrGreater() 6-70
UnorderedOrLess() 6-70
until 3-13
Upcase() 6-25

W

While 3-12

ア

値渡し 4-8
新しい
- 〜オブジェクトシステム関数 6-2
- 〜関数 6-2
- 〜グローバルなものを取得したり、セットする関数 6-4
- 〜サブクラス機構 1-13
- 〜ソートされた配列のための関数 6-3
- 〜その他の関数 6-5
- 〜データ詰め込み関数 6-4
- 〜配列関数 6-3
- 〜メッセージ送信関数 6-4
- 〜文字列関数 6-3

アンダースコアで始まるスロット 2-18

イ

イベントハンドラ 3-21
イミディエイト 1-2, GL-2
イミディエイト(即値) 2-2, 2-1
イミディエイト値と参照値 2-6
インプリメンタ GL-2, 4-9, 4-11
インラインオブジェクト構文 1-10

エ

演算子 2-30
- >= 2-33
- : 4-4
- :? 4-4
- / 2-32
- + 2-32
- – 2-32
- = 2-33
- < 2-33
- <= 2-33
- << 2-32
- <> 2-33
- > 2-33
- >> 2-33
- & 2-36
- && 2-36
- * 2-32
- div 2-32
- exists 2-36
- mod 2-32
- 演算子単項 - 2-35
- 優先順位 2-38

オ

オーバーライド 5-2
オブジェクト GL-3, 2-19
- 〜コンストラクタ 1-10
- 〜システム関数 6-5
- 〜とクラスシステム 2-1
- 〜の生存期間 1-7
- 〜モデル 1-2
- 〜リテラル 1-10, 2-27

親 GL-3

カ

ガベージコレクション 1-7
ガベージコレクタ 1-7
関数
- 〜オブジェクト 4-2, 4-8, GL-2
- 〜オブジェクトの例 4-12

　～起動 4-3
　～コンストラクタ 4-2
　～コンテキスト 4-8, 4-10
　～とメソッド 4-1
　～とメソッドの要約 6-98
簡単なフレーム 1-3

キ

疑似変数`self` 4-12
基本構文 1-9

ク

空白 1-9
クォーテッド式 2-28
クォーテッド定数 2-29
組み込み関数 6-1
組み込みプリミティブクラス 2-1
クラス 1-3, 2-1, GL-1
　～`Boolean` 2-2
　～`Char` 2-2
　～`Int` 2-2
　～`real` 2-12
　～構造 2-2
　～シンボル 2-4
　～スロット 2-19
クラスとサブクラス 2-4
クラスベースのプログラミング C-1
繰り返し 3-4
グローバル
　～関数 4-1
　～関数起動 4-7
　～関数宣言 4-6
　～な変数・関数の取得及びセット 6-92
　～変数 1-5, GL-2

ケ

継承 GL-2
継承チェーン 1-5
継承と探索 5-1

コ

コード 4-8
構造体 2-19
構文規約 xiv
構文文法 D-2
互換性 1-13, 6-2
コメント 1-11
　`//` タイプのコメント 1-11
　`/*...*/` タイプのコメント 1-11

サ

サブクラス 2-4
左辺値 2-30
算術演算子 2-32
参照 1-2, GL-4
　～オブジェクトカテゴリ 2-2
参照渡し 4-8

シ

式 2-24
識別子 2-13
字句環境 4-9, 4-10
字句文法 D-12
指数表記のサンプル 2-12
システム予約 2-18
実数 2-12

条件付きメッセージ送信演算子 :? 4-4
定数 2-27, GL-1
　～宣言 2-27
　～への置換 2-28
シンボル 2-13, 2-18
　|Weird%Symbol!| 2-13
シンボルパス式 2-22

ス

スコープ 1-5, 1-7
既に時代遅れな関数 6-5
スロット 2-18, GL-4
　～ class 2-19
　～式 2-18

セ

制御構文 3-1
整数 2-11
　～演算関数 6-49
　～であるパス式 2-22
　～の範囲 2-11
セミコロンセパレータ 1-9

ソ

ソートされた配列用の関数 6-38
即値 1-2, GL-2
その他の関数 6-95

タ

代入演算子 2-30
単項演算子 2-35

チ

小さい数値 6-52
抽象データ型 4-14

テ

データ隠蔽 4-14
データオブジェクトとそれらの関係 3-10
データ型 1-3
データ抽出関数 6-82
データ詰め込み関数 6-87

ト

等価、関係演算子 2-33
動的 1-8
特殊なスロット名 2-20
特殊文字と、Unicode の対応 2-10
特別な意味を持つ文字 2-10

ニ

二重継承機構 2-19, 5-1
にょろ括弧({}) 1-12

ネ

ネイティブ関数 1-13, 4-16

ハ

バイナリ 2-1
バイナリオブジェクト 2-4, GL-1
配列 2-1, 2-16, GL-1
　～アクセッサ 2-15, 2-17
　～関数 6-26
　～要素 2-15

パス式 2-18, 2-22, GL-3
バックスラッシュエスケープ文字(\) 2-9
パラメタ渡し 4-8

ヒ

非エスケープ文字 2-9
ビッグ・エンディアン 6-82
ビット演算関数 6-26
標準文字コード 2-9

フ

ブール演算子 2-34
複式 3-1
符号付きの 0 6-52
符号付きの無限 6-52
浮動小数演算関数 6-52
浮動小数環境の管理 6-71
プリミティブクラス 2-1
フレーム 1-3, 2-1, 2-18, GL-2, 4-1
〜アクセッサ 2-20
〜のクラス名 2-19
プログラミングユニット 2-19
プロトタイプ GL-3
〜継承 5-2
〜継承ルール 5-3
文法 D-2

ヘ

ペアレント GL-3
〜継承 5-4
〜継承ルール 5-5
変数 1-5, 2-24
〜のスコープ 3-2
〜の有効範囲 1-7
〜への値の代入 2-6

ホ

他のハンドラへの例外再発行 3-21

ム

無限ループ 3-11

メ

メソッド GL-3, 4-1
〜を持つフレーム 2-19
メッセージ 2-19, GL-3, 4-1
〜環境 4-9, 4-11
〜送信演算子 4-3
〜送信演算子 : 4-4
〜送信関数 6-80

モ

文字 2-9
文字セット 1-11
文字列 2-14
〜演算子 2-36
〜関数 6-18
〜内での特殊文字 2-15

ユ

ユーザー定義クラス 2-2, 2-4, 2-16
有効範囲 1-7

ヨ

予約語
`begin` 3-2

`constant` 2-27
`end` 3-2
`func` 4-2
`global` 4-6
`inherited` 4-4
`local` 2-25
`native` 4-2, 4-16
`return` 4-3

リ

リテラル配列 2-16
リトル・エンディアン 6-82
リファレンス 1-2

レ

例外
～関数 6-77
～処理 3-14
～シンボル 3-16
～シンボルの例 3-16
～シンボルパート 3-17
～の定義 3-16
～の発行 3-20
～の捕獲 3-21
～フレーム 3-17
～フレームサンプル 3-18
レコードに似たフレーム 2-19
レシーバ GL-3, 4-9, 4-11
列挙型 2-13

ロ

ローカル 2-24
～宣言 2-25
～変数 1-5, GL-3
～変数のスコープ 2-25
論理
～演算子 2-34
～型 GL-1
～値(ブール値) 2-11
～値定数 2-11

©1997 Apple Computer,Inc.1 Infinite Loop,Cupertino,CA

Apple、Apple ロゴ、AppleDesign、Apple Link、AppleShare、Apple SuperDrive、AppleTalk、HyperCard、LaserWriter、Light Bulb Logo、Mac、MacApp、Macintosh、Macintosh Quadra、MPW、Newton Toolkit、NewtonScript、Performa、QuickTime、StyleWriterおよびWorldScriptは、米国その他の国で登録されたアップルコンピュータ社の商標です。

AOCE、AppleScript、AppleSearch、ColorSync、develop、eWorld、Finder、OpenDoc、Power Macintosh、QuickDraw、SNA•ps,StarCore、および Sound Manager は米国アップルコンピュータ社の商標です。ACOTはアップルコンピュータ社のサービスマークです。

DDJ（DeveloperDepot Japan）はXplain Corporationの商標です。

Motorola および Marco は Motorola,Inc. の商標です。

NuBus は Texas Instruments の商標です。

PowerPC は International Business Machines Corporation の商標であり、所定のライセンス契約のもとで使用しているものです。

Windows は Microsoft Corporation の商標であり、SoftWindows は Microsoft Corporation の Insignia によるライセンスのもとで使用している商標です。

UNIX は UNIX System Laboratories,Inc. の商標です。

CompuServe、Pocket Quicken は Intuit、CIS Retriever は BlackLabs、PowerForms は Sestra,Inc.、ACT!は Symantec、Berlitz の各商標であり、その他の商標はそれぞれの法的所有権者に帰属します。

この出版物に記載の製品名は参照を目的としたものであり、それらの製品の使用を支持あるいは推奨するものではありません。製品の仕様および説明はすべて各メーカまたはサプライヤから提供されたものです。アップル社はこの出版物に記載した製品の選択、性能あるいは使用につき一切の責任を負いません。合意、契約、保証はすべてメーカと将来のユーザの間で直接おこなわれるものとします。

責任の制限

アップル社は、この出版物に記載した製品の内容、ならびにこの出版物の完全性および正確性に関し、一切の保証をいたしません。アップル社は、商品性および特定の目的に対する適合性を含む保証は、明示的・黙示的にかかわらず、一切いたしません。

本書の作成方法について

翻訳にあたっては、AppleDocViewer形式の原書から、内容をテキストファイルとして取りだし、Microsoft MS-Wrod 6.0Jを用いて翻訳した。翻訳に当たっては、一部 LogoVistaEtoJ2.5 を投入している。

次にこれを Adobe PageMaker6.0J でレイアウトし、Adobe Illustrator5.0J で描いた線画及び、Microsoft Equation Editor2.0 で作成した数式を加え、最終的に Adobe Acrobat フォーマットに変換した。フォントは Times, Helvetica, Courier, 中ゴシック BBB、リュウミン L-KL, Symbol, Zapf Dingbats を使用した。

さらに、本書中のプログラムリストは、Apple Message Pad 130J と接続されたNewtonToolKit 1.6.3のインスペクタウィンドウでの正常な動作確認後、誤りの見つかった部分は修正後、本書の原稿に Cut & Pasteで直接戻されたので、訳者はプログラムリストの誤植のなさに関してはある程度自信がある。誤植だらけのプログラミング入門書を出している出版社は少しは見習ってもらいたいと思っている。

上記の作業は、Apple Macintosh IIvx 及び、Apple PowerMacintosh 8500/180 上で行われた。

バージョン情報

1997/5/20	Ver. 0.9.0	アップル確認用
1997/5/30	Ver. 0.9.1	NJ 公開用ベータ版
1997/6/28	Ver. 1.0.0	一般公開版

プログラミング言語

Ｎｅｗｔｏｎ Ｓｃｒｉｐｔ

翻訳 齋藤匡弘

brown@ga2.so-net.or.jp

本書はフリーウェアです。