VectorScript Language Guide

VectorScript Language Guide

VectorScript Language Guide の文章と図は、Alexandra Duffy、Teresa Heaps、Susan Collins の各氏によって作成されました。

協力

Craig Hollinshead 氏 と Vladislav Stanev 氏に感謝の意を表します。

登録とアップデート

お手持ちの Vectorworks ソフトウェアのご登録がお済みでない場合は、以下の A&A Information ポータルサイトでご登録ください。http://www.aanda.co.jp/myinfo/index.html

Vectorworks ソフトウェアアップデートの自動通知を受け取るには、Vectorworks 環境設定のその他タブで、週または月単位で 自動的にアップデータを確認するよう選択できます。

Vectorworks 使用許諾契約

このソフトウェアの使用を拘束する使用許諾契約は、Vectorworks ReleaseNotes ディレクトリで確認できます。また、Vectorworks についてダイアログボックスの使用許諾をクリックして使用許諾契約を表示することもできます。

目次

VectorScript の概要 1

VectorScript は、Vectorworks のソフトウェアパッケージに含まれるスクリプト言語です。軽量なプログラミング言語である VectorScript の構文は Pascal と似ており、Pascal のプログラム要素を多く取り入れています。VectorScript は実際、Pascal 言語の「スーパーセット」として、Pascal の基本機能に Vectorworks CAD エンジンの機能にアクセスするための各種 API（アプリケーションプログラミングインターフェース）を加えて拡張したものです。

本ガイドでは VectorScript 言語の基本的な使い方を解説します。『VectorScript Function Reference』では、コマンドのリファレンスをまとめています。このリファレンスは VWHelp/VectorScript Reference/VSFunctionReference.html にあります。Vectorworks でのスクリプト記述に関する開発者向けドキュメントは http://developer.vectorworks.net（開発元サイト）でも入手できます。

このガイドでは VectorScript 言語の概要を紹介します。

VectorScript の背景について

VectorScript は最初、1988 年に MiniCAD+ 1.0 リリースの MiniPascal として公開されました。MiniCAD のその後のバージョンでは新しい技術の実装に合わせ、それらをサポートする API が拡張されました。1998 年の Vectorworks 発売時に、MiniPascal は VectorScript に変更されました。同時に Vectorworks にプラグインが導入され、ユーザが VectorScript 言語を使用してツール、メニューアイテム、およびオブジェクトを作成できるようになりました。

VectorScript でできること

VectorScript は比較的汎用性のあるプログラミング言語で、そのため一般的なプログラミングタスクのほとんどを実行できます。計算、値の格納、データ操作などのタスクはすべて、VectorScript 言語に含まれる標準の要素とメソッドでサポートされています。VectorScript では Vectorworks 製品に固有の拡張機能も提供しており、汎用的な言語には含まれていない新しい機能が追加されています。

オブジェクトの作成と編集

VectorScript では、Vectorworks ファイル内で直接オブジェクトを作成して編集できます。線などの基本オブジェクトだけでなく、3D の多段柱状体や複雑な 3D ソリッド図形などの複雑なオブジェクトも作成できます。VectorScript では、言語に組み込まれている多くの API を使用して、これらのオブジェクトの形状とグラフィック属性の両方を編集することもできます。

ドキュメントの制御

VectorScript には、個々の Vectorworks ファイルの各種設定を制御する API が含まれています。これらのインターフェースを使用して、デザインレイヤの縮尺や可視性といったファイルの幾何属性に加えて、塗りつぶしや線の模様の色などのグラフィック属性を取得したり設定したりできます。

拡張データ

VectorScript では、ファイルに含まれている拡張データを必要に応じて操作できます。VectorScript API を使用すると、ワークシート、データレコード、およびテクスチャにアクセスして制御し、ファイルを「詳細に編集」できます。

VectorScript でできないこと

VectorScript の機能は非常に多岐にわたりますが、これらの機能のほとんどは Vectorworks と Vectorworks ファイルを対象としたものです。VectorScript はこのような環境で使用することを前提にしているため、独立した言語に必要な機能は含まれていません。

- VectorScript には、複数のファイル間で動作する機能や、Vectorworks ファイルに関連のない状況で動作する機能は含まれていません。
- シンプルで安定した動作を確保するため、VectorScript にはメモリ割り当てを管理または制御する機能は含まれていません。
- VectorScript では、ファイル関連などの作業を行うシステムレベルの呼び出しをサポートしていません。
- VectorScript は、外部データベースなどの接続オプションを提供していません。
- VectorScript にはマルチスレッド機能がありません。

スクリプトの例

ここでは、一般的なスクリプトの基本原則を知っていただくために簡単な例を取り上げます。以下は、VectorScript メッセージバーにメッセージを表示し、5 秒後にそのメッセージを消去するという簡単なスクリプトの例です。

```
PROCEDURE FirstExample;
CONST
    kGREETING = 'Hello ';
VAR
    MyMessage : STRING;

BEGIN
    myMessage:='VectorScript';

    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

プログラムの最初にあるステートメントで手続きの名前を示し、VectorScript コンパイラに識別させます。

```
PROCEDURE FirstExample;
CONST
    kGREETING = 'Hello ';
VAR
    myMessage : STRING;

BEGIN
    myMessage:='VectorScript';

    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

スクリプトを VectorScript コンパイラに識別させます

このステートメントの後に、メインプログラムブロックと呼ばれる部分が続きます。メインプログラムブロックには、スクリプト実行時に必要となるデータ記憶域を宣言する部分と、実行される動作を指示する、スクリプトのソースコードが含まれています。

```
PROCEDURE FirstExample;
CONST
    kGREETING = 'Hello ';
VAR
    myMessage : STRING;

BEGIN
    myMessage:='VectorScript';
    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

スクリプト用のデータ記憶域を宣言します

スクリプトのソースコード

スクリプトの最後には、VectorScript コンパイラに対して上記のスクリプトコードを実行するよう指示する、特別なステートメントが置かれます。

```
PROCEDURE FirstExample;
CONST
    kGREETING = 'Hello ';
VAR
    myMessage : STRING;

BEGIN
    myMessage:='VectorScript';

    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

スクリプトを実行するよう、VectorScript コンパイラに指示します

スクリプトの各部に関する概念にはすぐに理解できないものもあるかもしれませんが、例をよく調べることで、スクリプトの構造や動き方がわかるようになります。これ以降のセクションでは、スクリプトの各部およびその基礎となる概念について詳しく説明します。

VectorScript の内容を確認する

新しいプログラミング言語を学習する最も良い方法は、実際にプログラムを書いてみることです。本ガイドとオンラインの機能リファレンス（http://developer.vectorworks.net、開発元サイト）を読み進めながら、ぜひ学んだ機能を試してみてください。手軽に VectorScript で機能を試したり、VectorScript 言語について学ぶ方法はいくつかあります。

最も簡単な方法で VectorScript の内容を確認するには、Vectorworks のファイルを開き、「ファイル」＞「取り出す」＞「VectorScript」コマンドでファイルを取り出します。取り出した後、ファイルをテキストエディタで開きます。オブジェクト、レイヤ、クラス、ファイル設定など、Vectorworks ファイル一式の VectorScript が表示されます。このスクリプトコードを元のファイルと比較すると、VectorScript を使用して特定の設定を作成する方法を確認できます。または、スクリプトのコードを一部修正して空のファイルに取り込み、変更がどのようにファイルに影響するか確認することもできます。さらに、このスクリプトコードの一部をそのままの形で、あるいはカスタムコードのベースとして自分のスクリプトで使用することもできます。

他にも、VectorScript の内容を確認する便利な方法として、Vectorworks の「ツールマクロ」および「図形選択マクロ」があります。これらのツールアイテムは VectorScript を利用して Vectorworks で動作を実行させるもので、これらを使用して VectorScript の使い方を探ることができます。「ツールマクロ」を使用すると、グラフィック属性とツールの設定を保存して後で使用できます。図形選択マクロを使用すると、ファイル内の一部のオブジェクトを選択するよう検索条件を指定できます。どちらの方法も独自のスクリプトを作成する場合に非常に便利であり、スクリプトを開いてスクリプトコードを調べることで、こうしたテクニックの使い方を確認できます。

最善の方法は、独自のスクリプトを最初から作成してみることです。Vectorworks のリソースブラウザを使用すると、空のファイルを作成して VectorScript エディタで編集できます。VectorScript エディタにはいくつかの便利な機能が含まれており、それらの機能を使用すると、API 情報など VectorScript 言語の基本要素に簡単にアクセスできます。

VectorScript を学習している間は、スクリプトを作成しても実行されなかったり、予想どおりに動作しなかったりする場合があるかもしれません。スクリプトが実行されない問題を修正するには、VectorScript のエラー情報ファイルをチェックします。スクリプトに重大なエラーがあれば、このファイルにエラーの原因が表示されます。スクリプトが予想どおりに動作しない問題を修正するには、VectorScript デバッガを使用してコードをトレース実行し、問題を特定します。また、他の多くの言語で使われる基本的な方法として、関連する変数の値を表示するステートメントをスクリプトに挿入することもできます。VectorScript には、この作業に便利な `Message()` ステートメントが用意されています。

VectorScript をうまく使って楽しんでください！

VectorScript の言語構造 2

プログラミング言語には必ず、その言語を使用してプログラムを記述する方法を指定した一連の規則があります。これらの規則は言語構造と呼ばれます。この構造は言語の最下層の構文で、変数の名前の付け方やプログラムのステートメントを区切る方法などの規則を定義します。ここでは VectorScript の基本的な言語構造について説明します。

大文字と小文字の区別

VectorScript では、大文字と小文字は区別されません。つまり、言語のキーワード、変数、関数名、その他の識別子は大文字、小文字、またはそれらを混合した形式で指定でき、どの形式も同じものと見なされます。これは、JavaScript や C などの言語とは異なる点です。

シンボル

VectorScript では、**シンボル**は言語の不可分な要素、つまり意味を持つ最小の要素です。VectorScript のソースコードはこれら一連のシンボルからなり、スクリプト内で Vectorworks に処理内容を指示します。シンボルは**トークン**とも呼ばれます。シンボルを定義するには、いくつか規則があります。

各シンボルは一連の ASCII 文字で記述され、次の規則に従う必要があります。

- 各シンボルは分離できません。シンボルを他のシンボルの内部に置くこともできません。
- シンボルは 8 ビットの ASCII 文字（技術的には ISO-8859-1 文字セット）で構成する必要があります。
- シンボルには、VectorScript 内で広範な意味と用途を与えることができます。データ記憶領域の表示、実行する数値演算の指示、スクリプト実行の制御などの用途に使用できます。
- シンボルは**区切り文字**と呼ばれる他の文字で分離されます。シンボルは区切り文字で分離され、独立した要素として識別されます。シンボルと区切り文字は交互に指定する必要があります。

区切り文字

VectorScript コンパイラでは**区切り文字**を使うことで、変数やステートメントなどの言語要素をスクリプト内で意味のあるオブジェクトとして、個別に識別できるようになります。VectorScript で使用する主な区切り文字は、スペース、タブ、および改行文字です。VectorScript ではこれらの文字を使用して言語オブジェクトを分離しますが、そうでない場合はこれらの文字を無視します。区切り文字はシンボル内には挿入できません。シンボル内に区切り文字を配置すると、2 つの分離した要素と見なされます（この場合、構文エラーが生成されます）。

VectorScript の字句要素には、スクリプトコード内で区切り文字として機能すると同時に他の機能を果たすものもあります。たとえば、VectorScript コンパイラは次のような数式を処理できます。

```
circumference:=2*3.14159*radius
```

この数式で、* 文字と用語 : = は演算を実行すると同時に区切り文字として機能します。これらの用語は**特殊シンボル**と呼ばれ、VectorScript でこのような「二重の目的」を実行する字句要素の一種です。特殊シンボルにはこの他に**コメント**や**コンパイラディレクティブ**などがあります。後続のセクションで詳しく説明します。

VectorScript コンパイラにとって、スペース、タブ、および改行に意味はないため、スクリプトコードのインデントやフォーマットに自由に使用できます。これらを使用することでスクリプトが読みやすく、理解しやすくなります。

コメント

VectorScript のコメントは、スクリプトコード内に説明テキストを配置するために使用します。多くの場合、作成者が後で参照したり、他の人がスクリプトコードを修正する場合に備えて、スクリプトを説明するために使用します。VectorScript コンパイラはコメントを無視します。

VectorScript のコメントの一般的な構文は次のとおりです。

{ **＜コメント文章＞** }

最初と最後の波括弧は、コメント文章の始めと終わりを示します。VectorScript は、C および C++ 形式のコメントはサポートしていません。

スクリプトを作成する際は、コードにコメントを付けることを強く推奨します。スクリプトにコメントがあると、作成者（あるいは他の人）が後でスクリプトを修正する必要が生じた場合に、コードの動作を簡単に把握できます。

別の構文として、次のように括弧とアスタリスクを使用することもできます。

(* **＜コメント文章＞** *)

この構文を使用すると、すでにコメントが含まれている一連のスクリプト全体をコメントアウトできます。

以下に例を示します。

```
(* ブロックコメント
{ 複数行のコメント。}
{ 他のコメント。}
*)
```

リテラル

VectorScript の**リテラル**は、スクリプトコード内に直接配置されたデータ値です。リテラルは、数値、文字列、論理値の TRUE と FALSE、または特別な値である NIL のいずれかです。以下では、それぞれのリテラルについて説明します。

整数リテラル

VectorScript の整数値は一連の数字で示され、その前に（負の値を示す）マイナス記号を付けることもできます。

3 -255 1000000

浮動小数点リテラル

浮動小数点値は、従来の小数点表記または指数表記（科学的記数法）を使用して表します。

小数点形式の浮動小数点値は次のように表します。

- プラスまたはマイナス記号（オプション）。以下の要素が続きます。
- 値の整数部分。以下の要素が続きます。
- 小数点と、値の小数部分。

指数表記の浮動小数点値は次のように表します。

- プラスまたはマイナス記号（オプション）。以下の要素が続きます。
- 値の整数部分。以下の要素が続きます。
- 小数点と、値の小数部分。以下の要素が続きます。
- e または E。以下の要素が続きます。
- プラスまたはマイナス記号（オプション）。以下の要素が続きます。
- 指数を示す 1 ～ 3 桁の整数値。前半の整数部分と小数部分に、指数部の示す値が乗算されます。

3.1415927	6.02e23	.333333333
-3.267E-04	-0.004568	1.1414e-15

VectorScript では数字のリテラルと値に寸法表記も使用でき、フィート、インチ、メートルなどの一般的な寸法表記記号を認識します。寸法表記を用いた数値リテラルの使い方に関する詳細は 17 ページの『スクリプトの単位と数値』を参照してください。

文字列リテラル

文字列リテラルは、一重引用符で囲まれた 0 個以上の任意の文字の並びです。これらは次の規則に従って表します。

- 各リテラルは一重引用符で囲む必要があります。
- 定数は複数行にわたって記述できますが、改行文字はスペースに変換されます。
- 空白、タブ、および改行はリテラル内の有効な文字としてカウントされます。
- 文字列リテラルの長さは最大 255 文字です。
- 引用符の間に文字が存在しない文字列リテラルは NULL 文字列と見なされます。
- 文字列リテラル内に一重引用符を記述するには、リテラル内に一重引用符を 2 つ続けて記述します。

```
'VectorScript'          'Nemetschek Vectorworks'
'Section A-A'           'Provide approx. 3'' clearance'
```

論理値リテラル

VectorScript の論理値リテラルは「真理値」（真または偽のいずれか）を表します。VectorScript のほとんどの比較演算は、演算の成否を示す論理値を生成します。真理値の状態は 2 つ存在する可能性があるため、VectorScript には 2 つの論理値リテラル、つまり TRUE と FALSE というキーワードがあります。

NIL リテラル

VectorScript に含まれる最後のリテラルは特殊な NIL リテラルです。VectorScript の他のリテラルは特定の種類のデータを表します。NIL リテラルはこれとは異なり、値が存在しないことを示します。ある意味で、NIL は数値以外のデータタイプでいう 0 のようなものです。NIL は通常、HANDLE データタイプに関連付けられており、NIL が使われている場合はハンドルが存在しないことを示します。

識別子

VectorScript の**識別子**は、定数、変数、データタイプ、手続き名や関数名など他の要素を参照するために使用するシンボルです。

VectorScript の識別子を記述するための規則は、多くのプログラミング言語と似ています。

- 最初の文字はアルファベット文字か下線にする必要があります。
- 以降の文字には、文字、数字、または下線が使用できます。
- 識別子にはスペース、タブ、その他の文字を含めることはできません。
- 識別子の長さに制限はありませんが、最初の 255 文字だけが意味を持ちます（VectorScript コンパイラが認識します）。

識別子が指定の規則に準拠していない場合、スクリプトはコンパイルされず、VectorScript のコンパイルエラーが生成されます。

値の識別子

- num
- SUM
- color_32bit
- _dummy
- totalLumberUsed
- A_very_fine_identifier

無効な識別子

- 52pickup
- three+two
- SUB TOTAL

予約語

予約語は VectorScript の特別なクラスのシンボルです。予約語は特殊なシンボルで、VectorScript コンパイラに対して重要な意味を持ちます。コンパイラは予約語を使用してスクリプトに関する重要な情報を判定します。また、その情報を使用してスクリプトを正しくコンパイルし、実行する方法を判定します。スクリプトでは予約語を識別子として使用しないでください。使用するとエラーや予期しない動作が発生します。

VectorScript キーワード

次の表は VectorScript の予約語（**キーワード**とも呼ばれます）の一覧です。

- ALLOCATE
- BOOLEAN
- DIV
- ELSE
- FUNCTION
- INTEGER
- NIL
- OTHERWISE
- REPEAT
- TO
- USES
- AND
- CASE
- DO
- END
- GOTO
- LABEL
- NOT
- PI
- STRING
- TRUE
- VAR
- ARRAY
- CHAR
- DOWNTO
- FALSE
- HANDLE
- LONGINT
- OF
- PROCEDURE
- STRUCTURE
- TYPE
- VECTOR
- BEGIN
- CONST
- DYNARRAY
- FOR
- IF
- MOD
- OR
- REAL
- THEN
- UNTIL
- WHILE

他のキーワード

次の表で一覧している予約語は、現時点で VectorScript コンパイラでは意味を持ちませんが、将来使用する可能性を見越して予約されているものです。これらの予約語もスクリプトには使用しないでください。使用すると、VectorScript 言語の将来のバージョンで問題が発生する可能性があります。

- FILE
- FORWARD
- IMPLEMENTATION
- INHERITED
- INTERFACE
- INTRINSIC
- OBJECT
- OVERRIDE
- PACKED
- PROGRAM
- SET
- UNIT
- USES
- WITH

VectorScript では大文字と小文字の違いが無視されるため、（begin と BEGIN など）どちらで記述しても同じ用語になります。そのため大文字をつかったものと小文字を使ったものの混用は避けるべきです。

特殊シンボル

特殊シンボルは、VectorScript に含まれるもう 1 つの特別なクラスのシンボルです。特殊シンボルは予約語と同様に、VectorScript コンパイラに対して重要な意味を持ちます。特殊シンボルはコンパイラの動作やスクリプトの制御および実行方法を示すほか、スクリプトの他のステートメントに対する区切り文字として機能します。

+	-	*
/	^	=
(	)	[
]	{	}
.	,	$
<>	<=	>=
:=	..	**

この表に示されている文字および文字ペアは、VectorScript 言語で特殊シンボルとして認識されます。個々の特殊シンボル固有の意味と用法は、以降のセクションで詳しく説明します。

変数、定数、データタイプ、数値とデータの形式 3

前の章では、VectorScript コードに直接組み込まれるデータ値であるリテラルの概念を紹介しました。このような静的データでしか動作しないスクリプトは制限が多く、柔軟性に欠けています。この制限を克服するため、VectorScript では**定数**と**変数**を使用します。定数と変数は、データ値に関連付けられている名前（技術的には**識別子**）です。変数や定数は、値を「格納する」または「含む」という言い方をします。

定数と変数を使用すると、値を名前ごとに格納したり操作したりできます。定数の場合、スクリプトの実行中に値を変更することはできません。これに対して変数の場合、名前に関連付けられた値は、その名前に新しい値を割り当てることでいつでも変更できます（このため「変数」と呼ばれます）。

VectorScript の重要な概念にはもう 1 つ、**データタイプ**があります。名前が示すように、データタイプはスクリプトで操作できる種類のデータです。データタイプはスクリプトで操作する情報に構造と意味を与えるものであり、VectorScript で効率的かつ安全に処理できるようにします。

ここではスクリプトで変数と定数を使用する方法や、VectorScript で使用できる各種データタイプや数値とデータの形式の詳細について説明します。

変数

変数は**変数宣言**によって作成されます。変数宣言は、変数名の識別子を特定のデータタイプと関連付けるものです。このデータタイプは VectorScript コンパイラに対し、データを保持するためにどれだけのメモリ領域を割り当てる必要があるかを指示します。

変数宣言の一般的な構文は次のとおりです。

< 識別子 >(,< 識別子 >, . . . ） ： < データタイプ >;

データタイプが同じ複数の識別子は、カンマで区切ったリストで指定できます。

VectorScript のタイプ宣言

```
jobName:STRING;                    i,j,k:INTEGER;
```

単純なデータタイプと配列タイプの場合、これらの宣言はスクリプトの VAR ブロックと呼ばれる特定の場所で行われます。スクリプトのこの部分はメインプログラムブロックの最初、スクリプトコード本体の前に置かれ、VAR というキーワードで示されます。変数を宣言できる場所は VAR ブロックのみです。Basic や JavaScript などの言語とは異なり、スクリプトのソースコード内で変数を宣言することはできません。

VectorScript は VAR ブロックで指定された情報を使用して、スクリプトを正しく実行するために必要なメモリを割り当てます。次の例では、スクリプトのデータ領域として 2 つの変数を宣言しています。

```
PROCEDURE Example_1;
VAR
    s:STRING;
```

```
    i:INTEGER;

BEGIN
    s:='VectorScript';
    i:=2;
    Message('Hello ',s);
    Wait(i);
    ClrMessage;
END;
Run(Example_1);
```

VAR ブロックで宣言された変数に、実際に値が割り当てられるわけではないことに注意してください。変数の格納領域に実際に値を割り当てる操作は、スクリプトの本体で行われます。VAR ブロックの目的は必要な領域を定義することであり、データを定義することではありません。

定数

定数は**定数定義**を使用して作成されます。定数定義も識別子とメモリ内の記憶領域を関連付けますが、変数の宣言とは異なり、値がその場所にただちに割り当てられます。定数の値は、定義した後にスクリプトで変更することはできません。

定数定義の一般的な構文は次のとおりです。

< 識別子 > = < 値 >;

定数は変数と異なり、明示的なデータタイプを必要としません。

定数定義はまた、スクリプト内の CONST ブロックでのみ行われます。スクリプトのこの部分はメインプログラムブロックの最初、スクリプトコード本体および VAR ブロックの前に置かれます。このブロックは CONST キーワードで示されます。VAR ブロックの場合と同様に、この種類の領域宣言（定数定義）は CONST ブロックでのみ許可されます。

次の例では、定数を使用して定義した値で、特定の市場など固有の対象向けにスクリプトをカスタマイズできるようにしています。

```
PROCEDURE Example_1;
CONST
    { 変数と区別するために大文字で表記 }
    LOCAL_GREETING_ENGLISH = 'Hello ';
    LOCAL_GREETING_FRENCH = 'Bonjour ';
VAR
    s:STRING;
    i:INTEGER;

BEGIN
    s:='VectorScript';
    i:=2;
```

```
    Message(LOCAL_GREETING_ENGLISH,s);
    Wait(i);
    ClrMessage;
END;
Run(Example_1);
```

定義した値は必要に応じてスクリプトで使用できます。定数にはデータタイプが不要であることに注意してください。VectorScript では必要に応じて暗黙のうちに、値が適切なタイプに変換されます。

定数は、任意の基本的なデータタイプ（INTEGER、LONGINT、REAL、STRING、CHAR、または BOOLEAN）を格納できます。また、VectorScript では定数の定義において三角関数や序数、その他の数値関数の使用もサポートしています。次の表は、定数定義ブロックでサポートされ、スクリプトで定数の定義に使用できる関数の一覧です。

- Abs()
- Sqr()
- Sqrt()
- Ord()
- Chr()
- Trunc()
- Round()
- Sin()
- Cos()
- Tan()
- ArcSin()
- ArcCos()
- ArcTan()
- Ln()
- Exp()

VectorScript のデータタイプ

スクリプトで使用するデータには、必ずデータタイプを関連付ける必要があります。VectorScript コンパイラはデータタイプによって、スクリプトの実行時に割り当てるメモリ領域や、そのデータに対して演算などの操作を実行する方法を判定します。

変数を宣言する時は、必ずデータタイプを指定する必要があります。また、手続きや関数を宣言する場合、それぞれのパラメータ、および関数の場合は戻り値に対して必ずデータタイプを指定する必要があります（手続きと関数の詳細は 61 ページの『ユーザ定義関数』を参照してください）。

VectorScript のデータタイプには、**基本タイプ**と**ユーザ定義タイプ**の 2 つのカテゴリがあります。基本タイプはコンパイラであらかじめ定義されていますが、ユーザ定義タイプはスクリプトコードの内部で定義されます。

基本データタイプ — 数値

VectorScript は、INTEGER、LONGINT、REAL という 3 つの数値データタイプをサポートしています。

INTEGER

INTEGER タイプの値は数値全体のサブセットです。INTEGER の値の範囲は -32768 から 32767 までで、分数や小数の部分を含めることはできません。分数や小数の部分を含む数値が INTEGER タイプの変数に割り当てられた場合、分数や小数の部分は丸められます。

VectorScript が受け付ける INTEGER タイプの変数は、INTEGER 値、または有効な INTEGER 値の範囲に含まれる LONGINT 値に限られます。

LONGINT

LONGINT タイプの値も同様に数値全体のサブセットです。LONGINT 値は INTEGER タイプより広い範囲の値を表すことができます。LONGINT 値の範囲は -2,147,483,648 から 2,147,483,647 までです。

INTEGER 値と同様、LONGINT 値に分数や小数の部分を含めることはできません。分数や小数の部分を含む数値が LONGINT タイプの変数に割り当てられた場合、分数や小数の部分は丸められます。VectorScript が受け付ける LONGINT タイプの変数は LONGINT 値または INTEGER 値です。

INTEGER タイプおよび LONGINT タイプの値が含まれる数値演算は次の規則に従います。

- INTEGER タイプの有効な値の範囲に含まれる整数定数はすべて INTEGER タイプと見なされます。LONGINT タイプの範囲に含まれるが INTEGER タイプの範囲に含まれない整数定数はすべて LONGINT タイプと見なされます。
- 演算子の両方のオペランド（単項演算子の場合は単一のオペランド）が INTEGER タイプの場合、その結果は INTEGER タイプになります（結果がそのタイプで表せない値になる場合は丸められます）。同様に両方のオペランドが LONGINT タイプの場合、その結果は LONGINT タイプになります。
- オペランドの一方が LONGINT タイプで、もう一方が INTEGER タイプの場合、INTEGER オペランドは LONGINT に変換され、その結果は LONGINT タイプになります。この値が INTEGER タイプの変数に割り当てられた場合、値は丸められます。

REAL

REAL タイプの値（浮動小数点値とも呼ばれます）は実数のサブセットで、数値の分数や小数の部分を格納できます。有効な REAL 値の範囲は 1.7 x 10e-307 から 1.7 x 10e307 までです。

VectorScript が受け付ける REAL タイプの変数は、REAL、LONGINT、または INTEGER 値です。LONGINT 値および INTEGER 値は、変数に割り当てられる前に REAL データタイプに変換されます。

基本データタイプ — テキスト

文字列リテラルを一重引用符で囲ってスクリプトに含める方法は 8 ページの『リテラル』を参照してください。また VectorScript では、スクリプトの実行中に文字列値をデータとして格納することもできます。スクリプト内でこのデータを表すために、STRING、CHAR、および CHAR 配列という 3 つのデータタイプをサポートしています。ここでは最初の 2 つのタイプを取り上げます。CHAR 配列の詳細は 27 ページの『CHAR 配列による拡張文字列サポート』を参照してください。

STRING

STRING 値は、スクリプト内で文字データを格納して操作するために使用します。STRING タイプの変数は最大 255 文字までのテキストデータを格納し、STRING データ値は任意の有効な ASCII 文字をサポートします。STRING タイプのデータ値は文字列や文字リテラルとも互換性があります。

CHAR

CHAR データ値は単一の ASCII 文字で、STRING データタイプとは種類が異なります。CHAR 値は、STRING 値から単一の文字を取得して変換するために使えるほか、スクリプト内で使用する特殊文字を定義するためにもよく使われます。

STRING 値および CHAR 値は互換性のあるタイプで、これらのタイプの値は直接割り当てたり比較したりできます。

基本データタイプ — その他

VectorScript は、**以下の**データタイプもサポートしています。

BOOLEAN

BOOLEAN データ値は、真理値（および予約語）の TRUE または FALSE のいずれかを保持できます。BOOLEAN タイプの値は、独立したデータタイプであるという点で Java や JavaScript の論理値に極めて似ており、C や C++ とは異なり TRUE や FALSE をシミュレートするのに数値を使用しません。

一般に論理値はスクリプト内で発生する比較演算の結果であり、最もよく使用するのはスクリプトの実行中に判断を行う場合です。

HANDLE

VectorScript の HANDLE 値は、メモリにある他の Vectorworks データへの参照を格納するために使用します。HANDLE タイプの値を最もよく使用するのは、オブジェクト、レイヤ、クラスなど Vectorworks の内部構造に関連するデータを参照する場合です。VectorScript では、スクリプトから直接このデータを取得または設定するための簡単な方法として VectorScript API 全般にわたり広範囲に HANDLE 値を使用します。

メモリ内のデータへの参照以外に、HANDLE 値は値 NIL に設定することもできます。9 ページの『NIL リテラル』で説明しているように、値 NIL は参照が存在しないか見つからなかったことを示します。

HANDLE 値は動的なメモリ位置への参照であるため、格納したり、ドキュメント内の特定の要素への永久的な参照のように処理したりすることはできません。HANDLE 値を格納したり再使用したりすると、スクリプト内でエラーなどの予期しない動作が引き起こされる恐れがあります。

VECTOR

VectorScript には特殊なデータタイプ VECTOR が用意されており、VectorScript 内でベクトル演算をサポートします。ベクトルは移動に関する数量を表すために使用し、方向と距離（または大きさ）で示されます。VectorScript の VECTOR は 3 つの REAL 値で構成され、これらはひとかたまりの値として処理することができます。

VECTOR 値は VectorScript 言語のベクトル API と組み合わせて使用すると、スクリプトで複雑な座標演算を行う際に大変役立ちます。この API の詳細は『VectorScript Function Reference』を参照してください。

POINT

POINT データタイプは、2D の点の座標を格納するために使用します。これは 2 つの REAL 値 x および y で構成される複合データタイプです。値は現在のドキュメントの単位と見なされ、ドキュメントの原点に対する相対値となります。

POINT3D

POINT3D データタイプは、3D 空間の点の座標を格納するために使用します。これは 3 つの REAL 値 x、y、および z で構成される複合データタイプです。値は現在のドキュメントの単位と見なされ、ドキュメントの原点に対する相対値となります。

RGBCOLOR

RGBCOLOR データタイプは、色を赤、緑、および青の 3 つのコンポーネントとして格納できます。各コンポーネントは LONGINT 値です。

数値とデータの形式

スクリプトの単位と数値

単位記号に関連付けられた数値は、以下の規則に従います。

- VectorScript は数値を解析する際、既定の適切な単位記号を調べます。数値の後に不正な文字が含まれている場合は VectorScript 警告が生じます。
- VectorScript ではユーザ定義の単位記号は調べません。
- 単位記号の付けられている VectorScript の数値は、アクティブなドキュメントで現在設定されている単位設定にかかわらず、単位記号に合わせてサイズが決定されます。以下に例を示します。

```
Rect(a,a,a + 1'2",a + 1'2");
```

この例では単位設定に関係なく、常に一辺が 1 フィート 2 インチ（14 インチ）の四角形を描画します。

```
Rect(a,a,a + 14cm,a + 14cm);
```

この例ではファイルの単位設定に関係なく、常に一辺が 14 cm の四角形を描画します。

- 単位記号の付けられていない数値は、ファイルの現在の単位設定に合わせてサイズが決定されます。以下に例を示します。

```
Rect(a,a,a + 14,a + 14);
```

この例では、ファイルの単位で一辺が 14 の四角形を描画します。現在の単位設定がフィートの場合、四角形の一辺は 14 フィートになります。単位設定がミリメートルの場合、描画される四角形の一辺は 14 mm になります。

- 数値定数は、指定した任意の単位記号に結び付けられます。以下に例を示します。

```
CONST
    kX = 5.5cm;
```

この例ではセンチメートルの単位記号に結び付けられ、それが保持されます。

絶対モードと相対モード
距離－角度モード
Write と WriteLn でデータをフォーマットする

絶対モードと相対モード

VectorScript のデフォルトの描画モードは**絶対モード**です。絶対モードでは、オブジェクトの描画または位置決め用のパラメータとして渡された値は Vectorworks 座標系の実際の座標値であると見なされます。以下に例を示します。

```
Rect(2',0',0',2');
```

この例では、左上隅が（2',0'）、右下隅が（0',2'）の四角形が描画されます。

相対モードでは、値はアクティブなドキュメントで現在の描画ペン位置からの相対オフセットであると見なされます。上と同じ例で説明します。

```
Rect(2',0',0',2');
```

呼び出し前のペンの位置が（4',2'）である場合、この呼び出しは左上隅が（4',4'）で右下隅が（6',2'）の四角形を描画します。このモードのまま続けて描画呼び出しを行うと、直前の関数呼び出しで描画されたペンの位置を起点に描画が実行されます。

VectorScript は Absolute( ) と Relative( ) の 2 つの呼び出しを使用して、ドキュメントの描画モードを明示的に設定します。これらの呼び出しを使用してドキュメントの描画モードを設定し、絶対値の代わりにオフセットを使用してオブジェクトを描画できます。以下に例を示します。

```
Relative;
MoveTo(2",2");
Poly(1",0",0",1",-1",0",0",-1");
```

この例では、一辺が 1" で、左下隅が (2",2") に配置された多角形を描画します。Relative( ) を使用せずに同じ呼び出しを行うと、絶対座標の位置を使用して別の多角形が描画されます。

相対モードを設定すると、Absolute( ) を呼び出すかスクリプトの実行が完了するまで、相対モードのアクティブな状態が続きます。スクリプトから正しい結果が得られるよう、描画モードは必ず目的の状態にリセットします。

距離－角度モード

VectorScript は、距離－角度モードと呼ばれる別の数値モードによるオブジェクトの描画もサポートしています。距離－角度モードは極座標と同様、距離と方向の角度を使用して座標が定義されます。距離と角度のペアを指定する時は、x 座標の代わりに距離を、y 座標の代わりに角度を指定します。以下に例を示します。

```
Relative;
MoveTo(2",2");
Poly(1",0",0",1",-1",0",0",-1");
```

これは次のように指定できます。

```
Relative;
MoveTo(2",2");
Poly(1",#0d,1",#90d,1",#180d,1",#270d);
```

距離－角度モードの場合、シャープ（ # ）記号を使用して、次に続く数値が角度であることを示します。

VectorScript は、距離－角度ペアの角度部分を指定するための広範な形式をサポートしています。次の表はサポートされている角度形式の一覧です。

角度形式	例
整数値	Rect(2,#90,2,#0);
10 進数値	Rect(2,#89.5,2,#359.5);
角度	Rect(2,#90d,2,#0d);
度 - 分 - 秒	Rect(2,#90d15’12",2,#25d30’45");
測量単位	Rect(20’,#N45d30’00"E,15’,#S45d15’2"W);
ラジアン	Rect(2,#1.57r,2,#0r);
グラジアン	Rect(2,#100g,2,#45g);

測量単位を使用する場合は、AngleVar( ) と NoAngleVar( ) を使用して、方位角の値が正しく解釈されるようにします。

Write と WriteLn でデータをフォーマットする

Write または WriteLn のパラメータリストに含まれる各パラメータは、次に示すように出力用にフォーマットできます。

パラメータ：[MinWidth] : [DecPlaces]

フィールド MinWidth と DecPlaces はオプションです。

MinWidth はデータ値の全体的なフィールド幅（桁数）の最小値または文字数を示します。この値は 0 以上でなければなりません。

数値とフォーマット
文字列値とフォーマット
数値と Write および WriteLn の例
文字列値と Write および WriteLn の例
スクリプトの単位と数値

数値とフォーマット

MinWidth が値の幅（桁数）より小さい場合は、Vectorworks は MinWidth を無視して値全体を表示します（後述の DecPlaces も参照してください）。MinWidth が値の幅（桁数）より大きい場合は、値の前にスペースが追加されます。

REAL データの場合、DecPlaces を使用すると値の小数点以下の表示桁数を制御できます。DecPlaces は、MinWidth フォーマット指定子とは独立して機能します。

ある値の DecPlaces を 2 に設定している場合、常に小数点以下 2 桁の精度で表示が行われます。この場合、必要に応じて MinWidth 指定子は無視されます。値の小数点以下の桁数が、指定した小数点以下の桁数より多い場合は、値が丸められます。REAL 以外の値の場合、DecPlaces はエラーを生成します。

数値と Write および WriteLn の例

文字列値とフォーマット

MinWidth 値は文字列の長さの表示指定子として機能し、MinWidth が文字列値の長さより小さい場合は文字列が切り捨てられます。MinWidth が文字列の長さより大きい場合、値の前にスペースが追加されます。

文字列値と Write および WriteLn の例

数値と Write および WriteLn の例

INTEGER 値

次の例では、フォーマットされる値（の桁数）が優先され、MinWidth に指定した値は無視されます。

```
theInt:=23456;
Write(theInt:3);
```

ファイルに '23456' が書き込まれます。

MinWidth がフォーマットされた値の幅を超えている場合、値の前にスペースが追加されます。

```
theInt:=23456;
Write(theInt:7);
```

ファイルに ' 23456' が書き込まれます。

REAL 値

次の例では、MinWidth と DecPlaces の値の組み合わせを使用して、値の文字列をフォーマットしています。表示される値は（小数点を含め）合計 6 文字で、小数点以下 2 桁が表示されます。値は、指定した表示設定に合わせて丸められます。

```
theReal:=789.128;
Write(theReal:6:2);
```

ファイルに '789.13' が書き込まれます。

DecPlaces 設定が表示される値の精度を超えている場合、DecPlaces 設定に合わせて値の最後に 0 が追加されます。MinWidth は、値と DecPlaces 設定の両方で無視されます。

```
theReal:=789.128;
Write(theReal:2:6);
```

ファイルに '789.128000' が書き込まれます。

文字列値と Write および WriteLn の例

この例では、MinWidth 指定子を変化させて、文字列全体の値から一部を表示しています。

```
theString:='This is a sample string';
Write(theString:7);
```

ファイルに 'This is' が書き込まれます。

```
theString:='This is a sample string';
Write(theString:25);
```

ファイルに ' This is a sample string' が書き込まれます。

```
Write('VectorScript':6);
```

ファイルに 'Vector’ が書き込まれます。

```
Write('VectorScript':16);
```

ファイルに '   VectorScript' が書き込まれます。

VectorScript の配列 4

VectorScript の**配列**は、単一の識別子で参照されるデータ値の集まりです。配列を使用すると、スクリプトの実行時に大量のデータを格納したり操作したりできます。

配列内のデータ値は連続した一連のメモリ領域に格納され、無作為または順番にアクセスできます。VectorScript では、**配列インデックス**を使用してこのデータにアクセスできます。配列インデックスは、配列内の特定の記憶領域に対応する INTEGER 値です。VectorScript の配列は、配列名の後にインデックス値を角括弧で囲むことでインデックス指定されます（つまり、配列から個別のデータ値が取得されます）。たとえば my_data という配列があり、i が INTEGER 変数とすると、次のように配列をインデックス指定できます。

```
my_data[i]
```

これによって配列の要素を参照できます。

VectorScript では、静的配列（ARRAY）と動的配列（DYNARRAY）の 2 種類の配列をサポートしています。ここでは、スクリプトで配列を使用するための構文と規則について説明します。

静的配列

静的配列（ARRAY）は変数に使用する場合と同じ方法で宣言しますが、変数では単一の記憶領域が割り当てられるのに対し、配列の値では一連の記憶領域が割り当てられます。静的配列の宣言は、他の変数と同様に VAR ブロックで行われます。

静的配列には 1 次元と 2 次元の形式があります。1 次元の静的配列の一般的な構文は次のとおりです。

< 識別子 > : ARRAY [m..n] OF **< データタイプ >**;

配列の宣言で、[m..n] という用語は配列の大きさ（サイズ）を示しています。[1..10] という大きさで宣言された配列は、メモリに 10 個の連続した記憶領域を割り当てます。静的配列は、任意の有効な基本データタイプとユーザ定義の STRUCTURE タイプ（詳細は 33 ページの『構造体を作成する』を参照）をサポートしています。

1 次元配列の要素から値を取得するには、前述の説明と同じ角括弧の表記を使用します。配列名は角括弧の左側に記述し、配列インデックスを示す負ではない INTEGER 値を角括弧内に記述します。

```
j := values[3];
values[23] := 15.5;
total := price[i] + tax;
```

配列インデックスは、負ではない任意の INTEGER 値、またはそのような値に解決される式です。

1 次元配列を実際に使用する例を示します。

```
PROCEDURE Example_41;
VAR
        s:STRING;
        i:INTEGER;
words:ARRAY[1..10] OF STRING;
```

```
BEGIN
    words[1]:='VectorScript ';
    words[2]:='is ';
    words[3]:='a ';
    words[4]:='fine ';
    words[5]:='language.';
    FOR i:=1 TO 5 DO s:=Concat(s,words[i]);
    Message(s);
END;
Run(Example_41);
```

この例では 10 個の要素がある STRING 配列を宣言し、スクリプトコードの最初で配列の要素に値を割り当てています。割り当て時に定数を使用して配列のインデックスを示していますが、代わりに INTEGER 値として評価される変数や他の識別子を使用することもできます。このような識別子は、後の Concat( ) 関数呼び出しの箇所で、配列要素を参照するために使用しています。

2 次元の静的配列は、1 次元配列の構文を拡張し、追加の配列インデックスを宣言に加えたものです。

< **識別子** > : ARRAY[m..n,r..s] OF < **データタイプ** >;

2 次元配列の宣言で、最初のインデックス値は配列の「行」の数、2 番目のインデックス値は「列」の数を定義します。このような 2 次元配列では、データ値を保持するために（m と r が 1 の場合は）n x s の連続した記憶領域が割り当てられます。

2 次元配列の要素にアクセスする方法は 1 次元配列の場合とほとんど同じです。

```
j := values[3,5];
values[23,1] := 15.5;
total := price[i,j] + tax;
```

2 次元配列を行と列で考えると、インデックス指定する配列要素の行と列の位置を示すために 2 つのインデックス値を使用することになります。

動的配列

VectorScript の動的配列（DYNARRAY）は静的配列と似ていますが、大きさ（サイズ）の指定方法が異なります。静的配列ではスクリプトの VAR ブロックで宣言する時に明示的にサイズを指定するのに対し、動的配列のサイズはスクリプトの実行中に宣言されます。動的配列のサイズは、スクリプトの実行中いつでも必要な領域に合わせて変更することもできます。動的配列は静的配列と同様、任意の有効な基本データタイプとユーザ定義の STRUCTURE タイプ（詳細は 33 ページの『構造体を作成する』を参照）をサポートしています。

動的配列は、静的配列と同様に 1 次元または 2 次元として指定できます。1 次元の動的配列の一般的な構文は次のとおりです。

< **識別子** > : DYNARRAY [] OF < **データタイプ** >;

動的配列は静的配列とは異なり、角括弧の中に配列のサイズ（大きさ）を含めないことに注意してください。サイズはスクリプトの実行時に定義されます。2 次元の動的配列の構文はほとんど同じです。

< 識別子 > : DYNARRAY [,] OF < データタイプ >;

1 次元の動的配列と同様、宣言時にインデックスの大きさは指定されません。ただし、配列が 2 次元であることを示すためにカンマが必要です。

動的配列のサイズを決定するため、VectorScript では ALLOCATE キーワードを（配列への参照と共に）使用し、すべてのデータ値を配列に格納するのに十分なメモリ領域を予約します。ALLOCATE を使用すると、配列を初めて使用する前に初期サイズを決定したり、必要な記憶領域が増えた（減った）場合に配列の大きさを変更したりできます。たとえば INTEGER 値を格納する配列 int_values に 5 個の記憶領域を割り当てるには、次の呼び出しを使用します。

```
ALLOCATE int_values[1..5];
```

角括弧の内側で指定された範囲は、作成して格納用に予約される要素の数を示しています。

動的配列をスクリプト内で実際に使用する例を次に示します。

```
PROCEDURE Example_42;
VAR
i,j,numtxt : INTEGER;

h : HANDLE;
textStore:DYNARRAY[] OF STRING;

BEGIN
numtxt:=Count(((T=Text) & (SEL=TRUE)));
j:=1;

ALLOCATE textStore[1..numtxt];

h:=FSActLayer;

WHILE (h <> NIL) DO BEGIN
IF (GetType(h) = 10) THEN BEGIN
textStore[j]:=GetText(h);
j:=j+1;
END;

h:=NextSObj(h);
END;

ALLOCATE textStore[1..numtxt+2];

TextOrigin(2,2);
```

```
CreateText('New text 1');
numtxt:=numtxt+1;
textStore[numtxt]:=GetText(LNewObj);

TextOrigin(2,4);
CreateText('New text 2');
numtxt:=numtxt+1;
textStore[numtxt]:=GetText(LNewObj);
FOR i:=1 TO numtxt DO BEGIN
Message('Array element ',i,' contains ', textStore[i]);
Wait(1);
END;

END;
Run(Example_42);
```

この例では、動的配列を使用して、選択セットに含まれている任意の選択文字列のテキストを格納しています。スクリプトでは最初に、他のいくつかの変数と同時に動的配列 textStore を宣言しています。VAR ブロックの宣言では動的配列が指定されていますが、この時点では記憶領域は割り当てられません。

スクリプトの本体ではまず、変数 numtxt の選択セットに含まれている選択済みテキストオブジェクトの数を格納します。この値は、次に ALLOCATE キーワードと共に使用します。

```
ALLOCATE textStore[1..numtxt];
```

このステートメントで、動的配列の記憶領域の量が初期化されます。

次に、スクリプトは選択された項目を処理し、テキストオブジェクトを見つけると、そのテキストを配列の要素に格納します。選択セット内のテキストオブジェクトの数はすでにカウントされているため、textStore のサイズは、見つかったテキスト文字列の数に応じた記憶領域に設定されています。

すべてのオブジェクトが処理された後、必要に応じて配列の大きさを変更して記憶領域を増やしたり減らしたりできます。この例では ALLOCATE を再び呼び出して、追加の記憶領域を予約しています。

```
ALLOCATE textStore[1..numtxt+2];
```

次に、新たに追加された記憶領域を使用して、スクリプトで作成されたテキストを格納します。スクリプトは最後に、textStore 配列に現在格納されている値を表示します。

配列に格納されている既存のデータの値は、配列の大きさを変更しても維持されることに注意してください。スクリプトの実行中に配列がより大きなサイズに変更された場合、VectorScript はその時点で配列にあるすべての値を維持します。配列がより小さなサイズに変更された場合も、VectorScript はできるだけ多くのデータ値を維持しようとします。ただし、より小さなサイズに変更する場合、新たに定義された配列の境界を超える場所に含まれている値は失われます。

動的配列のパフォーマンスに関する考慮事項
ベクトルと配列の表記
CHAR 配列による拡張文字列サポート
標準の STRING 関連操作を実行する

動的配列のパフォーマンスに関する考慮事項

動的配列は同等の静的配列より多くの「オーバーヘッド」を必要としますが、これはスクリプトの実行中にメモリを割り当てたり、配列の値を保持したりするためです。結果として、動的配列を使用したスクリプトの実行は静的配列を使用したスクリプトより遅くなる場合があります。

スクリプトのパフォーマンスを最大化するため、可能な限り静的配列を使用することを強く推奨します。スクリプトで動的配列が必要な場合に、ALLOCATE を頻繁に呼び出して記憶領域を予約しないでください。ALLOCATE は、スクリプトの実行中に、どうしても予約済みの記憶領域を変更しなければならない場合に限り使用します。また、ループや繰り返し実行ステートメント内では ALLOCATE を一切使用しないでください（これらのステートメントタイプの詳細は 49 ページの『繰り返しステートメント』を参照してください）。

ベクトルと配列の表記

前述のように、任意の基本データタイプについて配列を作成できます。これには VECTOR タイプも含まれています。ベクトルのフィールドにアクセスする方法として、配列形式の角括弧とドット表記という 2 つの方法がサポートされています。

配列形式の表記を使用してベクトルのフィールドにアクセスするには、配列の参照に角括弧を追加し、2 番目の角括弧内にベクトルのフィールドインデックスを指定します。次に例を示します。

```
vec_field[5][2];
```

この表記は、1 次元配列 vec_field の要素 5 にあるベクトルの、2 番目のフィールド（y 要素）にアクセスします。2 次元配列についてもこの表記を使用できます。vec_field2 が 2 次元配列の場合は次のように表記します。

```
vec_field2[4,5][2];
```

この表記は、配列の 4 番目の行および 5 番目の列にあるベクトルの、2 番目のフィールドにアクセスします。

ドット表記を使用してベクトルのフィールドにアクセスするには、インデックス指定する配列の参照に、ドット（フィールドへのアクセス）演算子とフィールド識別子を追加します。前の例の場合、次のような表記になります。

```
vec_field[5].y;
```

この表記は前の例と同様、vec_field の要素 5 にあるベクトルの、2 番目のフィールド（y 要素）にアクセスします。2 次元配列の場合も同様の表記を使用できます。

```
vec_field2[4,5].y;
```

この表記は、vec_field2 の 4 番目の行および 5 番目の列にあるベクトルの y 要素にアクセスします。

CHAR 配列による拡張文字列サポート

VectorScript では、CHAR データタイプの配列を使用する場合の特別な機能セットもサポートしています。この機能は、CHAR タイプの静的配列や動的配列について、スクリプト内で最大 32,767 文字の拡張文字列を処理できます。

CHAR タイプの配列は、VectorScript 内の特定の操作で STRING データタイプの代わりに使用できます。以下のセクションでは、VectorScript の CHAR 配列と STRING をサポートする操作について詳しく説明します。

STRING 値と CHAR 配列の間での割り当て

静的な CHAR 配列（ARRAY OF CHAR）と動的な CHAR 配列（DYNARRAY OF CHAR）はどちらも、STRING 変数への割り当てや同変数からの取得を行う時に、STRING 値の代わりに使用できます。

静的または動的な CHAR 配列を使用して STRING 変数に値を割り当てる場合、配列の長さが 255 文字を超えていれば最初の 255 文字は文字列変数にコピーされ、配列の残りの文字は無視されます。255 文字未満の値は STRING 変数に完全にコピーされます。

STRING 変数または定数から CHAR 配列に値を割り当てる場合も、同様の結果となります。CHAR 配列の長さが割り当てられる STRING 値より短い場合、配列に合わせて値が丸められます。次に例を示します。

```
PROCEDURE Example_43;
VAR
    Part_name: STRING;
    NameArray: ARRAY[1..16] OF CHAR;

BEGIN
part_name:= 'Acme Left-handed Smoke Shifter';

NameArray:=part_name;
END;
Run(Example_43);
```

この例で変数 part_name に割り当てられている STRING 値は、配列への割り当て時に次のように縮められます。

```
Acme Left-handed
```

when assigned to the array. 静的な CHAR 配列を使用して STRING 値を処理する場合、配列への格納が予測される中で最も長い STRING 値を格納できるよう、配列のサイズを宣言する必要があります。

静的な CHAR 配列とは異なり、動的な CHAR 配列のサイズは、配列に割り当てられる STRING 値の長さに自動的に変更されます。以下に例を示します。

```
PROCEDURE Example_44;
VAR
    sampleString: STRING;
    mytext: DYNARRAY[] OF CHAR;
BEGIN
sampleString:= 'VectorScript now handles lots of text';
mytext:= sampleString;
END;
Run(Example_44);
```

配列 mytext が宣言された後に使用されていない場合は、割り当てによって配列の長さが 36 に変更され、配列に次の文字列が格納されます。

```
VectorScript now handles lots of text
```

mytext にすでに値が割り当てられている場合は、割り当てによって配列の長さが 36 に変更され、STRING 値が配列に割り当てられます。それまで配列に保持されていた値は失われます。

テキストオブジェクトに文字列を取得する、または割り当てる

VectorScript では CHAR 配列を使用して、255 文字より長いテキストオブジェクトの STRING 値を設定または取得できます。VectorScript API の関数 GetText( ) と SetText( ) は、STRING 値の代わりに CHAR 配列の使用をサポートしています。

テキスト文字列を設定または取得するには、STRING のパラメータや変数の代わりに CHAR 配列の名前（角括弧なし）を使用します。以下に例を示します。

```
PROCEDURE Example_45;
VAR
    h : HANDLE;
    theText : STRING;
textArray : DYNARRAY[] OF CHAR;
BEGIN
    h:=FSActLayer;
    theText:= GetText(h);
CreateText(theText);
END;
Run(Example_45);
```

この例では、テキスト文字列の最初の 255 文字のみが返されます。動的配列を使用する場合は次のようになります。

```
PROCEDURE Example_45;
VAR
    h : HANDLE;
    theText : STRING;
    textArray : DYNARRAY[] OF CHAR;
BEGIN
    h:=FSActLayer;
    textArray:= GetText(h);
    CreateText(textArray);
END;
Run(Example_45);
```

動的配列ではテキスト文字列全体を取得して格納できます。その後でテキストの文字列全体を他の操作に使用できます。

レコードフィールドの文字列を取得する、または割り当てる

VectorScript では CHAR 配列を使用して、レコードフィールドに含まれている 255 文字よりも長い STRING 値を設定および取得することもできます。VectorScript API の関数 GetRField( ) と SetRField( ) は、STRING 値の代わりに CHAR 配列の使用をサポートしています。

レコードフィールド文字列を設定または取得するには、STRING のパラメータや変数の代わりに CHAR 配列の名前（角括弧なし）を使用します。以下に例を示します。

```
PROCEDURE Example_46;
VAR
    theText : STRING;
    longtext : ARRAY[1..512] OF CHAR;
BEGIN
    theText:= GetRField(FSActLayer,'Boring Info','Boring Notes');
    CreateText(theText);
END;
Run(Example_46);
```

この例では STRING 変数を使用しているため、フィールド内に格納されているテキスト文字列の最初の 255 文字のみを取得できます。CHAR 配列を使用する場合は次のようになります。

```
PROCEDURE Example_46;
VAR
    theText : STRING;
    longtext : ARRAY[1..512] OF CHAR;
BEGIN
longtext:= GetRField(FSActLayer,'Boring Info','Boring Notes');
    CreateText(textArray);
END;
Run(Example_46);
```

この例ではフィールドから最大 512 文字のテキストを取得し、配列に格納できます。または動的配列のサイズを変更して、レコードフィールドにあるテキストの長さを最大 32K まで対応させることも可能です。

標準の STRING 関連操作を実行する

VectorScript の文字列 API は、CHAR 配列の拡張文字列の処理もサポートしています。文字列の長さの取得、サブストリングの位置の取得、文字列の連結などの操作は、STRING 値の場合と同様に CHAR 配列に対しても実行できます。

文字列 API の関数で CHAR 配列を使用するには、関数のパラメータや戻り値として STRING 変数の代わりに CHAR 配列を使用します。以下に例を示します。

```
PROCEDURE Example_47;
VAR
    s : STRING;
    textArray : ARRAY [1..32] OF CHAR;
BEGIN
    textArray:= 'A VectorScript text string';
s:= Copy(textArray,3,12);
END;
Run(Example_47);
```

この例では、Copy( ) 関数が処理する値として STRING の代わりに CHAR 配列を使用しています。Copy( ) 操作の結果は、STRING 変数に割り当てられます。文字列 API 関数の呼び出しは、静的および動的な CHAR 配列の両方をサポートしています。

次の表は、CHAR 配列をサポートしている VectorScript API 関数の一覧です。

- Len()
- Pos()
- Concat()
- Copy()
- Delete()
- Insert()
- UprString()
- GetText()
- SetText()
- GetRField()
- SetRField()
- CreateText()

構造体 5

VectorScript の**構造体**は、変数を処理しやすいよう単一の識別子でグループ化した 1 つ以上の変数の固まりです。構造体を使用すると、複雑なデータをグループに編成するのに便利です。グループに編成されたデータは別々の実体ではなく、単一の「ユニット」として扱うことができます。

> このタイプの構成は、Pascal の標準的な用語でレコードと呼ばれます。Vectorworks や VectorScript の他の機能との競合や混乱を避けるため、VectorScript ではこの構成を構造体と呼びます。

構造体に含まれている変数は構造体のメンバーと呼ばれます。これらの変数は VectorScript のすべての基本タイプに対応します。静的配列と CHAR 配列も構造体のメンバーとしてサポートされています。また、他の構造体をメンバーとして使用する（「入れ子構造体」と呼びます）こともできます。構造体では、動的配列はサポートされません。

構造体を作成する

構造体は、スクリプトの TYPE ブロックという特別なセクションで宣言されます。この部分はオプションで、メインプログラムブロックの CONST セクションと VAR セクションの間に置かれ、この部分でのみ構造体を宣言できます。TYPE ブロックで宣言できる構造体の数に制限はありません。

構造体宣言の一般的な構文は次のとおりです。

```
<構造体名> = STRUCTURE
    <識別子>[,<識別子>,...] : <データタイプ>;
    <識別子>[,<識別子>,...] : <データタイプ>;
    ...
    ...
    <識別子>[,<識別子>,...] : <データタイプ>;
END;
```

宣言は、構造体を指す識別子で始まります。識別子の次に特殊シンボルの = とキーワード STRUCTURE が置かれ、その後に宣言されているメンバーを指定された識別子名でグループ化して扱うことが示されます。構造体のメンバーは他の変数と同様に宣言され、変数を宣言するための規則はすべてメンバーの宣言にも適用されます。構造体の宣言の終了は END キーワードで示されます。

構造体の宣言は変数や定数の宣言とは異なり、データ用の記憶領域を予約しません。その代わりに、スクリプト内で基本データタイプと同様に使用できる新しいデータタイプを定義します。このようなユーザ定義のタイプは、INTEGER、STRING などの基本タイプと同様に、変数や配列の宣言に使用できます。

たとえば、2D の点を示す構造体を定義する場合を考えてみましょう。点を示す構造体は次のように定義されます。

```
Point = STRUCTURE
    x,y : REAL;
END;
```

構造体 POINT には REAL タイプの 2 つのメンバーが含まれますが、この構造体をユーザ定義タイプとして使用する変数や配列が宣言されるまで、領域は割り当てられません。

```
PROCEDURE StructExample1;
TYPE
    POINT = STRUCTURE
x,y : REAL;
END;
VAR
    centerPt, target : POINT;
    vertex_list : ARRAY[1..20] OF POINT;
BEGIN
END;
Run(StructExample1);
```

変数 centerPt と target には、それぞれ構造体に含まれる 2 つの REAL 値の領域が含まれており、配列 vertex_list は 20 個の POINT 構造体または 40 個の REAL 値を格納するために十分なメモリ領域を予約します。POINT 構造体は、スクリプトでデータ値の領域を定義する時に使用する「テンプレート」として機能します。

構造体の値にアクセスする

構造体の中のメンバーは .（構造体メンバー）演算子を使用して直接参照できます。この演算子は参照する構造体名およびメンバー名と組み合わせて、次の形式で使用します。

< 構造体名 >.< メンバー名 >

この形式は「ドット表記」とも呼ばれ、指定のメンバーの値に直接アクセスできます。このタイプによる構造体メンバーへの参照を任意の単純変数の代わりに使用することで、値の取得や割り当てを行うことができます。

```
centerPt.x:= 0;
total:= windowData.cost + tax;
```

この表記は、値を比較する場合や VectorScript またはユーザ定義の関数に値を渡す場合にも使用できます。

```
partData.location:= GetLName(ActLayer);
GetObject(partData.name);
```

構造体の配列も、個別の構造体メンバーを参照するためにドット表記の使用をサポートしています。

```
vertices[5].x:= 2.67;
vertices[6].y:= vertices[5].y + 2.6;
```

配列要素に含まれる構造体メンバーを参照するには、メンバー演算子とメンバー名を配列要素への参照に追加します。

前述のように、構造体では静的配列をデータメンバーとして使用できます。構造体に含まれる配列は、メンバーの値を参照する場合に構文が若干複雑になりますが、それ以外の点では難しくありません。構造体内の配列要素の値を参照するには、メンバー演算子およびメンバー配列要素への参照を構造体インスタンスの識別子に追加します。

```
p.name[5]:= 'Marvin';
total:= total + winAssembly1.cost[k];
```

メンバーでない配列の場合と同様、INTEGER 値に解決される任意の式または定数を使用すると、メンバー配列要素のインデックスを指定できます。

配列をメンバーとして含む構造体の配列を作ることも可能です。ここでも、メンバー演算子と目的の配列要素への参照を組み合わせてデータ値を取得します。この場合、配列要素への参照がメンバー演算子の*両側*に使用されます。このためスクリプト内の構文がやや複雑に見える場合があります。

```
doorAssembly[3].cost[4]:= 24.55;
subtotal:=subtotal+doorAssembly[i].cost[j]+doorAssembly[i].cost[j+1];
```

これらの式は完全に有効ですが、こうした複雑な式では構文が正しく指定されているか、特に注意する必要があります。

他の構造体をメンバーとして含む構造体も、ネストされた構造体のメンバーを参照する場合はさらに複雑になります。このような場合は、メンバーの連鎖を利用して目的の値までのデータ階層をたどります。以下に例を示します。

```
PROCEDURE Example_51;
TYPE
    POINT = STRUCTURE
        x,y : REAL;
END;
CIRCLE = STRUCTURE
    ctr : POINT;
    radius : REAL;
END;
VAR
    c1,c2 : CIRCLE;
BEGIN
c1.ctr.x:= 4.5;
c2.ctr.y:= c1.ctr.y;
END;
Run(Example_51);
```

CIRCLE 構造体の宣言では POINT 構造体のインスタンスを使用して、さらに論理的にデータを編成しています。POINT インスタンスの x- または y- 要素を参照するには、入れ子構造体のメンバーの連鎖を利用します。

```
c1.ctr.x:= 4.5;
c2.ctr.y:= c1.ctr.y;
```

入れ子構造体の値を参照しアクセスするために、メンバー ctr とそのメンバーである x および y をメンバー演算子によってつなぎます。入れ子構造体のメンバーをつなぐ方法を、スクリプトで繰り返し使用することで、より深く入れ子になっている構造体のメンバーにアクセスできます。

式

6

VectorScript の値はすべて**式**を使用して割り当てられます。式は VectorScript の「語句」であり、これを評価することで値を生成できます。式には、値を表す単一の要素で構成される単純なものから、他の式やそれらへの演算の組み合わせで値を表す複雑なものまであります。

単純な式

単純な式では、単一の要素（または**オペランド**）を使用して値を表します。VectorScript の単純な式はほとんどの場合、（文字列や数値リテラルなどの）定数、変数名、または関数名です。

単純な定数式の値は、基本的に定数自体の値です。単純な変数式の値は、変数の識別子に関連付けられた値です。関数式の値は、関数の実行が完了した時に返される値です。

式	説明
`1.7`	**数値リテラル**
`'This is VectorScript'`	**文字列リテラル**
`TRUE`	**論理値リテラル**
`NIL`	**値 NIL**
`i`	**変数「i」**
`sum`	**変数「sum」**

複雑な式

複雑な式は複合式とも呼ばれ、他の式の値との間で結合や変換を行って値を算出します。次の式の値を例にとってみます。

```
i + 1.7;
```

この式は `1.7` と `i` の値から算出されます。`1.7` と `i` はどちらも単純な式でそれぞれ独自の値を含むため、これらを組み合わせて値を算出できます。

上の式では、より単純な 2 つの式の値を加算することで結果の値が得られます。式では演算子を使用して単純な式で演算を行い、より複雑な式に結合します。この場合、演算子はプラス記号で演算は加算です。

上の例では、プラス記号で結合された式もそれぞれオペランドと呼ばれます。演算子は通常、目的の演算を実行するために必要なオペランドの数でグループ分けされます。VectorScript は、単一のオペランドを必要とする単項演算子と 2 つのオペランドを必要とする二項演算子をサポートしています。

各演算子は結果の値を生成し、この値のデータタイプは演算子および値が得られるオペランドの両方を基に決定されます。演算子によっては互換性のあるオペランドのタイプが制限されている場合もあり、これらの要素はすべて結果の値のデータタイプに影響します。

演算子の優先度

VectorScript では数学と同様、**演算子の優先度**が演算を実行する順序を制御します。優先度が高い演算子は、優先度が低い演算子より先に実行されます。次の式を例にとってみます。

```
p = q + r * s;
```

乗算演算子（*）は加算演算子より優先度が高いため、乗算は加算の前に実行されます。代入演算子（=）はすべての演算子の中で最も優先度が低いため、変数 p と値の関連付け（代入）は、他の演算が完了しない限り実行されません。

演算子の優先度は、括弧を明示的に使用することで変更できます。前の例で加算を強制的に最初に実行するには、括弧を使用して次のように式を変更します。

```
p = (q + r) * s;
```

通常、優先度が不明瞭な場合は、評価の順序を明確にするために括弧を使用することを推奨します。

演算子の結合性

演算子の結合性は、優先度が同じ演算を実行する場合の順序を指定します。左から右への結合性は、演算子の優先度が等しい場合に演算が左から右の順に実行されることを意味します。次の式を例にとってみます。

```
p = q + r + s;
```

この式は次の式と同じです。

```
p = ((q + r) + s);
```

これは加算演算子に、左から右への結合性があるためです。これに対して、次の式を例にとってみます。

```
w = x = y = z;
```

この式は次の式と同じです。

```
w = ( x = (y = z));
```

これは代入演算子に、右から左への結合性があるためです。

算術演算子

算術演算子は、指定のオペランドに対して加算や乗算などの一般的な数値演算を実行します。算術演算子は、VectorScript の数値データタイプに対してのみ動作します。次の表は、VectorScript で使用可能な算術演算子の一覧です。

演算子	オペランド	優先度	結合性	操作
-	任意の数値	1	右から左	単項のマイナス（符号反転）
^	任意の数値	2	左から右	累乗
*	任意の数値	2	左から右	乗算
/	任意の数値	2	左から右	除算
DIV	INTEGER、LONGINT	2	左から右	整数除算
MOD	INTEGER、LONGINT	2	左から右	剰余（除算の余り）

演算子	オペランド	優先度	結合性	操作
+	任意の数値	3	左から右	加算
-	任意の数値	3	左から右	減算

単項のマイナス（ - ）

- を単一のオペランドの前で単項演算子として使用すると、オペランドの符号を反転します。つまり、正の値を絶対値が同じ負の値に、負の値を絶対値が同じ正の値に変換します。

加算（ + ）

+ 演算子は 2 つの数値オペランドを加算します。この演算子は加算のみに使用できます。他の多くの言語と異なり、この演算子を文字列の連結に使用することはできません。

減算（ - ）

- 演算子は、2 番目のオペランドを最初のオペランドから減算します。オペランドはどちらも数値でなければなりません。

乗算（ * ）

* 演算子は 2 つの数値オペランドを乗算します。

除算（ / ）

/ 演算子は、最初のオペランドを 2 番目のオペランドで除算します。この演算子は浮動小数点除算を実行し、両方のオペランドが INTEGER または LONGINT タイプである場合も常に REAL タイプの値が返されます。

整数除算（ DIV ）

DIV 演算子は最初のオペランドを 2 番目のオペランドで除算し、常に INTEGER タイプまたは LONGINT タイプの結果を返します。i DIV j の値は i/j の数学的な商を、最も近い INTEGER 値または LONGINT 値に切り捨てたものです。次の演算を例にとってみます。

```
j:= 36 DIV 5;
```

この演算では結果として 7 が返され、その値が変数 j に代入されます。

剰余演算（ MOD ）

MOD 演算子は最初のオペランドを 2 番目のオペランドで除算し、演算の余りを INTEGER タイプの結果として返します。次の演算を例にとってみます。

```
k:= 36 MOD 5;
```

この演算では結果として 1 が返され、その値が k に代入されます。

累乗（ ^ ）

^ 演算子は、最初のオペランドを 2 番目のオペランドが示すだけ累乗します。つまり、x^y は x の y 乗に等しくなります。

比較演算子

VectorScript では比較演算子を使用してさまざまなタイプの値を比較し、結果を論理値（TRUE または FALSE）で返します。比較演算子を使用した式の結果はほとんどの場合、スクリプトの実行フローの制御に使用されます。

次の表は、VectorScript で使用可能な比較演算子の一覧です。

演算子	オペランド	優先度	結合性	操作
<	数値、STRING、CHAR	4	左から右	より小さい
<=	数値、STRING、CHAR	4	左から右	より小さいか等しい
>	数値、STRING、CHAR	4	左から右	より大きい
>=	数値、STRING、CHAR	4	左から右	より大きいか等しい
=	任意のタイプ	5	左から右	等しい
<>	任意のタイプ	5	左から右	等しくない

より小さい（ < ）

< 演算子は、最初のオペランドが 2 番目のオペランドより小さい場合は TRUE、それ以外の場合は FALSE として評価します。オペランドは数値、文字列、または文字です。文字列は文字のエンコードに従って、アルファベット順で評価されます。

より小さいか等しい（ <= ）

<= 演算子は、最初のオペランドが 2 番目のオペランドより小さいか等しい場合は TRUE、それ以外の場合は FALSE として評価します。オペランドは数値、文字列、または文字です。文字列は文字のエンコードに従って、アルファベット順で評価されます。

より大きい（ > ）

> 演算子は、最初のオペランドが 2 番目のオペランドより大きい場合は TRUE、それ以外の場合は FALSE として評価します。オペランドは数値、文字列、または文字です。文字列は文字のエンコードに従って、アルファベット順で評価されます。

より大きいか等しい（ >= ）

>= 演算子は、最初のオペランドが 2 番目のオペランドより大きいか等しい場合は TRUE、それ以外の場合は FALSE として評価します。オペランドは数値、文字列、または文字です。文字列は文字のエンコードに従って、アルファベット順で評価されます。

等しい（ = ）

= 演算子は、2 つのオペランドが完全に等価である場合は TRUE、等価でない場合は FALSE を返します。任意のタイプのオペランドを使用できます。STRING タイプのオペランドの場合、値は文字単位で比較され、完全に同じ文字を含んでいる必要があります。

等しくない (<>)

<> 演算子は、= 演算子と正反対の評価をします。2 つの等しい値がこの演算子で比較すると、結果の値は FALSE になります。2 つの値が等しくない場合、結果は TRUE になります。

論理演算子

論理演算子は、論理値に対して比較演算にほぼ相当する操作を実行します。論理演算子は論理代数を使用してオペランドを評価し、演算の結果を返します。プログラミングでこれらの演算子が最もよく使われるのは、より小さな式を結合することで複数のオペランドを使用する複雑な比較を表す場合です。

次の表は、VectorScript で使用可能な論理演算子の一覧です。

演算子	オペランド	優先度	結合性	操作
NOT	BOOLEAN	1	右から左	論理否定
AND	BOOLEAN	7	左から右	論理積
&	BOOLEAN	7	左から右	論理積（短縮形）
OR	BOOLEAN	8	左から右	論理和
|	BOOLEAN	8	左から右	論理和（短縮形）

論理否定 (NOT)

単項の NOT 演算子は、単一の BOOLEAN タイプのオペランドの前で使用することでオペランドの値を反転します。たとえば、BOOLEAN タイプの変数 z が値 TRUE を含んでいる場合、式 NOT z は値 FALSE を返します。この演算は、より複雑な式の結果に対しても使用できます。たとえば、式 p>=q の評価結果が FALSE の場合、式 NOT(p>=q) の評価結果は TRUE になります。

Logical AND (AND)

AND 演算子は、最初のオペランドと 2 番目のオペランドの両方が TRUE の場合のみ、TRUE として評価します。どちらかのオペランドが FALSE として評価される場合、返される結果は FALSE です。AND 演算子を使用した式は、最初のオペランドの値にかかわらず、式の結果を返す前に常に両方のオペランドを評価します。

論理積 短縮形 (&)

& 演算子は、最初のオペランドと 2 番目のオペランドの両方が TRUE の場合のみ、TRUE として評価します。どちらかのオペランドが FALSE として評価される場合、返される結果は FALSE です。& を使用した式では、最初のオペランドが値 FALSE を返す場合、2 番目のオペランドは評価されません。2 番目のオペランドに副作用がある場合（値を返す関数呼び出しで生成されるオペランドなど）、それが発生しないことがあります。一般に、& 演算子に副作用を持ち込んでしまう次のような式は避けてください。

```
(a = b) & SetVectorFill(h,'Stone'){ 関数呼び出しが発生しない場合があります }
```

論理和 (OR)

OR 演算子は、最初のオペランドまたは 2 番目のオペランドが TRUE の場合、TRUE として評価します。両方のオペランドが FALSE として評価される場合のみ、FALSE の結果を返します。OR 演算子を使用した式は、最初のオペランドの値にかかわらず、式の結果を返す前に常に両方のオペランドを評価します。

論理和 短縮形（ | ）

| 演算子は、最初のオペランドまたは 2 番目のオペランドが TRUE の場合、TRUE として評価します。両方のオペランドが FALSE として評価される場合のみ、FALSE の結果を返します。| を使用した式では、最初のオペランドが値 TRUE を返す場合、2 番目のオペランドは評価されません。2 番目のオペランドに副作用がある場合（値を返す関数呼び出しで生成されるオペランドなど）、それが発生しないことがあります。一般に、| 演算子に副作用を持ち込んでしまう次のような式は避けてください。

```
(a = b) | SetVectorFill(h,'Stone') { 関数呼び出しが発生しない場合があります }
```

その他の演算子

代入演算子

13 ページの『変数』で説明したように、変数は値に関連付けられます（代入）。この値はスクリプトの実行中いつでも変更できます。これらの演算はどちらも代入演算子を使用して行われます。

: = 演算子の最初（左側）のオペランドは、変数、配列、要素、またはベクトルのフィールドか構造体のメンバーです。2 番目（右側）のオペランドには任意のタイプの値を使用できますが、この値は最初のオペランドと互換性のあるデータタイプでなければなりません。式の値は、右側のオペランドの値です。

代入演算子には、右から左への結合性があります。つまり、式では 2 番目のオペランドが最初に評価されます（VectorScript では、値と変数が互換性のあるタイプかどうかもこの順序で判定されます）。

配列アクセス演算子

23 ページの『VectorScript の配列』で説明したように、配列の要素にアクセスするには角括弧 [ ] および取得する値の位置インデックスを使用します。この角括弧一式は VectorScript では演算として扱われます。

[ ] 演算子は、最初のオペランド（角括弧の左側）として配列の名前を使用します。2 番目のオペランドは角括弧の間にあり、INTEGER 値として評価される任意の式です。

最初のオペランドとして指定された配列が 2 次元の場合、配列アクセス演算子には 3 番目のオペランドが必要で、同じく角括弧の間に置かれます。この場合、（カンマで区切られる）2 番目と 3 番目のオペランドはどちらも INTEGER 値として評価される任意の式となります。

次の式を例にとってみます。

```
price[3]
```

この式は、price 配列の 3 番目の要素に含まれている値として評価されます。2 次元配列 plant_data の場合、次の式を例にとってみます。

```
plant_data[2,i+4]
```

この式は、[2,i+4] で指定された要素に含まれている値として評価されます。式 i+4 を式のオペランドとして使用するには、INTEGER として評価される必要があります。

ベクトルと構造体メンバーへのアクセス演算子

VectorScript の . 演算子は特殊な演算子で、特定のデータタイプ、中でも特にベクトルと構造体に含まれている値に直接アクセスできます。

. 演算子の最初（左側）のオペランドには、ベクトルまたは構造体が必要です。2 番目のオペランドは、ほとんどの演算子と異なり、ベクトルのフィールド（要素）または構造体のメンバー名である必要があります。式は使用できません。ベクトルのフィールドの識別子は、3 つの有効なフィールド名 x、y、z のいずれかでなければなりません。構造体のメンバー名は、構造体タイプ宣言にある有効なメンバーに対応している必要があります。

次の式を例にとってみます。

```
distance_vector1.x
```

この式は、distance_vector1 ベクトルの x フィールドの値として評価されます。構造体を扱う場合、次の式を例にとってみます。

```
window_data.cost
```

この式は、次の構造体インスタンスのメンバー cost に含まれている値として評価されます。

```
window_data
```

ステートメント 7

VectorScript の**ステートメント**は言語の動作です。VectorScript の式は値として評価される「語句」と考えることができますが、式はどのような動作も「実行しません」。何らかの動作を行うには、完全な文章や命令に似た VectorScript ステートメントを使用する必要があります。VectorScript のステートメントは、スクリプトのタスク実行、スクリプトデータの管理、およびスクリプトの実行フロー制御を行います。

VectorScript のステートメントは常に「ブロック」内に存在し、スクリプトは単にステートメントの集合を含む 1 つの大きなブロックとして扱われます。VectorScript の各ステートメントはセミコロンで区切られ、VectorScript コンパイラはこの記号でステートメントの終わりを検出します。

ここでは、VectorScript に存在する各種のステートメントとそれらの構文について詳しく説明します。

代入ステートメント

代入ステートメントは、スクリプトの変数や類似の識別子に値を設定します。代入ステートメントは代入演算子（:=）を使用して、このシンボルの右側にある定数または識別子の値をシンボルの左側にある識別子に設定します。これは、右側の識別子の値を左側の識別子に代入すると考えることもできます。

代入ステートメントの一般的な構文は次のとおりです。

< 識別子 > := < 識別子または定数の値 >;

左側の識別子は任意の VectorScript データタイプです。配列要素、配列全体の参照、または構造体フィールドであってもかまいません。

以下に例を示します。

```
PROCEDURE Example_71;
CONST
    kInitialValue = 0;
TYPE
    POINT = STRUCTURE
        x,y:REAL;
    END;
VAR
    s : STRING;
    i : INTEGER;
    h : HANDLE;
    textdata : ARRAY[1..100] OF STRING;
    p1,p2 : POINT;
```

```
BEGIN
{ 定数の値を変数に代入 }
    i:= kInitialValue;
{ 戻り値を変数に代入 }
    h:= FSObject(ActLayer);
{ 戻り値を変数に代入 }
    s:= GetText(h);
{ 変数の値を配列要素に代入 }
    textdata[1]:= s;
{ 値を構造体メンバーに代入 }
    p1.x:= 0; p1.y:= 2;
{ メンバーの値を他のメンバーに代入 }
    p2.x:= p1.y;
{ メンバーの値を他のメンバーに代入 }
    p1.y:= p2.x;
END;
Run(Example_71);
```

この例から、代入演算子は非常に柔軟であることがわかります。この例では識別子に値を代入する時に定数、変数、構造体フィールド、および関数の戻り値を使用しています。また、複数のステートメントをセミコロンで区切って 1 行に記述できることにも注意してください。このセミコロンは各ステートメントの終わりを示します。

代入ステートメントは非常に柔軟に値の取得や代入を実行できますが、データタイプの互換性についてはいくつか規則があります。代入ステートメントを記述する時は次の規則に従う必要があります。

- REAL タイプの変数は、REAL 値、INTEGER 値、または LONGINT 値、およびこれらの結果を生成する任意の式に設定できます。
- LONGINT 変数は、LONGINT 値または INTEGER 値、あるいはこれらの値を生成する任意の式に設定できます。REAL 値にも設定できますが、値は最も近い整数値に丸められます。
- INTEGER 変数は、INTEGER 値、またはその値を生成する任意の式に設定できます。REAL 値にも設定できますが、値は最も近い整数値に丸められます。
- BOOLEAN 変数は、BOOLEAN 値、またはその値を生成する任意の式に設定できます。
- STRING 変数は、STRING 値または CHAR 値、あるいはこれらの値を生成する任意の式に設定できます。ARRAY 値または DYNARRAY OF CHAR 値にも設定できますが、配列の値は 255 文字までに丸められます。
- CHAR 変数は、CHAR 値、または CHAR 値を生成する任意の式に設定できます。STRING 値にも設定できますが、STRING が 1 文字より長い場合は丸められます。
- HANDLE 変数は、HANDLE 値、または HANDLE 値を生成する任意の式に設定できます。

代入演算子をスクリプト内で配列要素インデックスなしで使用すると、配列の値をブロックコピーできます。この方法では、要素ごとにコピーを繰り返すことなく大量のデータを簡単に転送できます。以下に例を示します。

```
PROCEDURE Example_72;
VAR
    values1,values2:ARRAY[1..5] OF INTEGER;
BEGIN
    values1[1]:= 2;
    values1[2]:= 4;
    values1[3]:= 8;
    values1[4]:= 16;
    values1[5]:= 32;
END;
Run(Example_72);
```

values1 の値を values2 に転送するには、通常、配列要素ごとに複数の代入ステートメントが必要になります。大きな配列の場合はこのタスクを実行すると時間がかかります。しかし VectorScript では代入演算子のオーバーロード（機能拡張）を使用しているため、単一のステートメントで値をコピーできます。

```
PROCEDURE Example_72;
VAR
    values1,values2:ARRAY[1..5] OF INTEGER;
BEGIN
    values1[1]:= 2;
    values1[2]:= 4;
    values1[3]:= 8;
    values1[4]:= 16;
    values1[5]:= 32;
values2:= values1;
END;
Run(Example_72);
```

代入ステートメントでは、values1 配列のデータを values2 配列の対応する要素に直接コピーします。このタイプの代入演算は動的配列に対しても実行できます。ただしどちらの場合も、代入演算子の両側に置かれた配列がまったく同じ大きさでなければ操作は完了できません。

ベクトルと構造体もこの方法でコピーできます。代入演算子の右側にあるアイテムのメンバーの値は、左側にあるアイテムの対応するメンバーフィールドにコピーされます。たとえば、ベクトル direction_vector1 の値を別のベクトルにコピーできます。

```
new_vector:= direction_vector1;
```

direction_vector1 のフィールドの値は new_vector のフィールドにコピーされます。フィールドごとに代入ステートメントを記述する必要はありません。

複合ステートメント

VectorScript では、複数のステートメントを単一のステートメントのように実行する方法として複合ステートメントをサポートしています。この機能は、いくつかのステートメントを結合して同時に実行する必要がある場合、たとえば制御ステートメントの分岐やループの中で実行する場合に特に便利です。

一連のステートメントから複合ステートメントを作成するには、最初のステートメントの前に BEGIN キーワードを付けます。列の最後には END キーワードを置き、ステートメントごとにセミコロンで区切ります。以下に例を示します。

```
BEGIN
    i:=1;
    j:= (3*2)+5;
    Message(i+j);
END;
```

BEGIN キーワードと END キーワードの間には 3 つのステートメントが含まれており、複合ステートメントが呼び出されるとこれらのステートメントはセットで実行されます。

複合ステートメントの一般的な構文は次のとおりです。

```
BEGIN
<ステートメント>; [<ステートメント>; <ステートメント>;...]
END;
```

複合ステートメントは入れ子にもできます。VectorScript コンパイラは、最後の BEGIN キーワードをスクリプト内で次に出現した END キーワードに関連付け、最後から 2 番目の BEGIN キーワードをその次の END キーワードに関連付けるなど、以下同様に各キーワードを関連付けます。BEGIN-END の対が正しく一致していないと、VectorScript エラーが発生します。

スクリプトの本体が複合ステートメントに似ている気がしたら、それは正解です。実際、任意の VectorScript スクリプト、ユーザ定義手続き、またはユーザ定義関数スクリプトの本体は 1 つの複合ステートメントです。

手続きステートメント

VectorScript の**手続きステートメント**は、既定の VectorScript API 関数およびユーザ定義の手続きや関数を呼び出して、スクリプト内で各動作を実行します。VectorScript API 関数の呼び出しを使用すると動作は Vectorworks で直接実行されます。ユーザ定義の関数は他の VectorScript ソースコードをカプセル化し、スクリプト内でこの手続きステートメントが呼び出されたときに、そのソースコードが実行されます。

手続きステートメントの一般的な構文は次のとおりです。

<手続き識別子>[(<**パラメータリスト**>)][:<**戻り値**>];

たとえば次のような関数呼び出しがあります。

```
Message('Hello VectorScript');
```

または

```
SetSelect(h);
```

これらは VectorScript の手続きステートメントの例です。ユーザ定義の手続きと関数の詳細は 59 ページの『ユーザ定義手続き』および 61 ページの『ユーザ定義関数』を参照してください。

GOTO ステートメント

GOTO ステートメントは、その GOTO ステートメントに関連付けられているラベルの直後にあるステートメントの最初に、スクリプトの実行を移します。以下に例を示します。

```
PROCEDURE Example_73;
LABEL 100;
VAR
    i,j : INTEGER;
BEGIN
    i:=10;
    j:=2;
    IF (j MOD 2 = 0) THEN GOTO 100;
    i:= i * 5;
    100: i:= i + 1;
    Message(i);
END;
Run(Example_73);
```

条件 (j MOD 2) = 0 が TRUE と評価されると、スクリプトの実行はただちにステートメント i:= i + 1 の最初に移され、式 i:= i * 5 は実行されません。

GOTO ステートメントの一般的な構文は次のとおりです。

GOTO <**ラベル**>;

GOTO ステートメントを使用する際は次の点に注意する必要があります。

- GOTO ステートメントが実行を移動できるのは、同じ手続き、関数、またはスクリプト本体の中に限られます。手続き間やスクリプト間の移動には使用できません。
- GOTO ステートメントの飛び先は常にステートメントの最初でなければなりません。
- 他のステートメントの構造に含まれているステートメントに分岐すると、結果は未定義になります。VectorScript コンパイラでは、この動作はエラーとして認識されません。

繰り返しステートメント

VectorScript は、スクリプトの一部を繰り返し実行する 3 つの方法をサポートしています。この操作をループと呼びます。VectorScript でサポートされている繰り返しステートメントは、FOR ステートメント、WHILE ステートメント、および REPEAT ステートメントです。

FOR ステートメント
WHILE ステートメント

FOR ステートメント

VectorScript の FOR ステートメントは、スクリプトの同じ部分を指定回数だけ実行します。この値は制御変数に置かれ、FOR ステートメントはその制御変数を評価して、スクリプトの実行を継続するかどうかを判断します。

FOR ステートメントの一般的な構文は次のとおりです。

```
FOR <制御変数> := <初期値> [TO | DOWNTO] <制限値>
DO <ステートメント>;
```

FOR ステートメントでは、制御変数の最初の値と最後の値（**制限値**）が設定されます。これらの値は INTEGER 値、LONGINT 値、または CHAR 値で、定数または式から計算される値を使用できます。FOR ステートメントでは、制御するスクリプト部分を実行するたびにあらかじめ制御変数の値が変更、評価されます。

FOR ステートメントには FOR-TO ステートメントと FOR-DOWNTO ステートメントの 2 種類があります。FOR-TO ステートメントでは、ステートメントで制御される部分を実行するたびに制御変数の値が 1 インクリメント（増加）します。以下に例を示します。

```
FOR i:=1 TO 10 DO Message('Pass ',i,' through FOR loop.');
```

この FOR-TO ステートメントでは、Message( ) 関数の呼び出しを実行するたびにあらかじめ制御変数 i が 1 インクリメントされて評価されます。

FOR-DOWNTO ステートメントでは、実行するたびに制御変数の値が 1 デクリメント（減少）し、これが制限値に達するまで繰り返されます。以下に例を示します。

```
j:=9;
FOR i:=10 DOWNTO 1 DO BEGIN
    Message('Pass ',i-j,'(',i,') through FOR loop.');
    j:= j - 2;
END;
```

この FOR 文では、実行するたびに i の値がデクリメントされ、これが制限値の 1 に達するまで繰り返されます。複合ステートメントを使用すると、FOR ステートメント構造に含まれる他のステートメントをいくつでも実行できることに注意してください。

FOR ステートメントを使用する際は次の点に注意する必要があります。

- FOR ステートメント内部で制御変数の制御値を変更しないでください。変更すると予測できない結果をもたらす可能性があります。
- FOR ステートメントのいずれかの制限式に制御変数を含めることはできません。
- 制御値がどちらも等しい場合、FOR ステートメントで制御されるステートメントは 1 回だけ実行されます。
- 制御値が反転している場合、FOR ステートメントはスキップされます。

WHILE ステートメント

VectorScript の WHILE ステートメントは、BOOLEAN 値を返す制御式の評価結果が TRUE である間、スクリプトの同じ部分を繰り返し実行します。WHILE ステートメントの一般的な構文は次のとおりです。

WHILE < **制御式** > DO < **ステートメント** >;

制御式は制御対象のステートメントを実行する前に評価されるため、制御対象のステートメントがまったく実行されない場合もあります。以下に例を示します。

```
PROCEDURE Example_74;
VAR
h:HANDLE;
BEGIN
h:= FActLayer;
WHILE (h <> NIL) DO BEGIN
      SetSelect(h);
    h:=NextObj(h);
END;
END;
Run(Example_74);
```

この例では FActLayer( ) 関数の呼び出しによって、アクティブレイヤにある最初のオブジェクトへのハンドルが返されます。アクティブレイヤにオブジェクトがない場合、オブジェクトを選択してレイヤの次のオブジェクトを取得する呼び出しは実行されません。

レイヤにオブジェクトが存在する場合、この例では処理するオブジェクトがなくなるまでループが繰り返され、その後で自動的にループが終了します。これは NextObj( ) 呼び出しがハンドルを返せない場合に NIL を返し、WHILE ステートメントが制御対象のステートメントを実行する前に式を評価するためで、式の評価結果が FALSE（h = NIL）になると制御対象のステートメントは実行されません。WHILE ステートメントは FOR ステートメントと異なり、制御対象のステートメント内から実行を制御できます。

REPEAT ステートメント

REPEAT ステートメントは WHILE ステートメントと同様、制御式の評価結果が FALSE になるまでスクリプトの同じ部分を繰り返し実行します。ただし REPEAT ステートメントは WHILE ステートメントと異なり、制御対象のステートメントを実行してから制御式を評価します。つまり、制御対象のステートメントは必ず最低 1 回は実行されます。

REPEAT ステートメントの一般的な構文は次のとおりです。

REPEAT < **ステートメント** > UNTIL < **制御式** >;

WHILE ステートメントのセクションにある例は、REPEAT ステートメントを使用して簡単に書き直すことができます。

```
PROCEDURE Example_75;
VAR
h:HANDLE;
```

```
BEGIN
h:= FActLayer;
REPEAT
      SetSelect(h);
    h:=NextObj(h);
UNTIL (h = NIL);
END;
Run(Example_75);
```

この形式では、h が最初 NIL だったかどうかにかかわらず REPEAT-UNTIL 構造内のステートメントが最低 1 回は実行され、それによって悪影響が生じたりエラーが発生したりする場合があります。一般に REPEAT ステートメントは、制御対象のステートメントを実行しても悪い影響が出ないような条件で使用する必要があります。WHILE ステートメントが最も適しているのは、実行を制御するための条件がすでに満たされている場合です。これに対し REPEAT ステートメントが最も適しているのは、条件がステートメントの実行によってのみ満たされる場合です。

REPEAT ステートメントは BEGIN や END を使用する必要がない点にも注意してください。これは、REPEAT キーワードと UNTIL キーワードによってその間のステートメントが複合ステートメントとして扱われるためです。

条件ステートメント

VectorScript は、スクリプト内で実行フローに影響を及ぼす判断を行う 2 つの方法をサポートしています。これは分岐と呼ばれる処理です。VectorScript でサポートしている条件ステートメントは IF ステートメントと CASE ステートメントです。

IF ステートメント

VectorScript の IF ステートメントは BOOLEAN 制御式として評価され、式の評価結果が TRUE の場合のみ制御対象のステートメントが実行されます。IF ステートメントは、制御式の評価結果が FALSE の場合に 2 番目のステートメントを実行するよう記述することもできます。

IF ステートメントの一般的な構文は次のとおりです。

IF <**制御式**> THEN <**ステートメント**> [ELSE <**ステートメント**>];

IF ステートメントの実行時に制御式が評価されて BOOLEAN の結果が生成されます。結果が TRUE の場合、THEN キーワード以降のステートメントが実行されて IF ステートメントが終了します。式の評価結果が FALSE の場合、ELSE キーワードとステートメントが存在しなければステートメント全体がスキップされます。存在する場合は ELSE キーワード以降のステートメントが実行されます。以下に例を示します。

```
IF (i mod 2) THEN Message('Even value') ELSE Message('Odd value');
```

i の値が偶数の場合、式 i MOD 2 の評価結果は TRUE となり、ステートメント Message('Even value') が実行されます。i の値が奇数の場合は Message('Odd value') が実行されます。

THEN キーワードおよび ELSE キーワードの間に含まれているステートメントの後にセミコロンは不要であることに注意してください。この場合、ELSE キーワードがステートメントの終わりを示すためです。ELSE キーワードを省略した場合はセミコロンが必要になります。

IF ステートメントは他のステートメントと同様、制御対象のステートメントとして複合ステートメントをサポートしています。IF ステートメントもネストできます。つまり、THEN キーワードの後に続くステートメントが IF ステートメントであってもかまいません。ネストによって、複数の相互排他条件の結果に基づいて動作を行うステートメントを構成できます。

IF ステートメントをネストすると、次に示すようにコードはすぐに複雑になります。

```
PROCEDURE Example_76;
VAR
i:INTEGER;
BEGIN
i:= Ord('c');
IF (i > 48) THEN IF (i > 57) THEN IF (i > 65) THEN IF (i > 90) THEN
IF (i > 97) THEN IF(i < 123) THEN Message('Lower case alpha')
ELSE Message('Out of range') ELSE Message('Some punctuation')
ELSE Message('Upper case alpha') ELSE Message('Some punctuation')
ELSE Message('Number') ELSE Message('Out of range');
END;
Run(Example_76);
```

IF と THEN の対応がわかりにくくなった場合は、複合ステートメントを使用したり字下げやコメントを書くことで、ソースコードを見やすくできます。

```
PROCEDURE Example_76;
VAR
i:INTEGER;
BEGIN
i:= Ord('c');
{Out of range}
IF (i > 48) THEN
    {number}
    IF (i > 57) THEN
        {punctuation}
        IF (i > 65) THEN
            {upper alpha}
            IF (i > 90) THEN
                {punctuation}
                IF (i > 97) THEN
                    {lower alpha}
                    IF (i < 123) THEN
```

```
                    Message('Lower case alpha')
                ELSE
                    Message('Out of range')
            ELSE
                Message('Some punctuation')
        ELSE
            Message('Upper case alpha')
    ELSE
        Message('Some punctuation')
ELSE
    Message('Number')
ELSE
    Message('Out of range');
END;
Run(Example_76);
```

CASE ステートメント

VectorScript の CASE ステートメントを使用すると、実行されるステートメントの候補リストを指定して、各ステートメントを識別する定数と関連付けることができます。CASE ステートメントの実行時に制御式が評価され、その結果が定数のいずれかと一致すると関連付けられたステートメントが実行されます。オプションとして OTHERWISE 句を追加すると、定数のリストからいずれのオプションも選択されなかった場合、この句に含まれるステートメントを実行するよう設定できます。

CASE ステートメントの一般的な構文は次のとおりです。

```
CASE <制御式> OF
    <定数>:<ステートメント>;
    <定数>:<ステートメント>;
    ...
    ...
    [OTHERWISE <ステートメント>;]
END;
```

制御式の評価結果は、INTEGER、CHAR、BOOLEAN のいずれかの値です。以下に例を示します。

```
PROCEDURE Example_77;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
```

```
CASE j OF
    49:Message('Number');
    77:Message('Upper case alpha');
    110:Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_77);
```

変数 j は INTEGER 値として評価され、この値は CASE ステートメントの定数リストと比較されます。この例では j の値がリストの定数の範囲外になるため、OTHERWISE 句が実行されます。

CASE ステートメントは、ある程度柔軟に定数を指定できます。たとえば CASE ステートメントのいくつかの条件について、同じコードの実行が必要となる場合があります。CASE ステートメントでは冗長なオプションを使用することなく、単一の CASE オプションに対して複数の定義をカンマで区切って指定できます。

```
PROCEDURE Example_78;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
    49:Message('Number');
    58,59,60,61,62,63,64:Message('Non alpha printable character');
    110:Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

制御式の評価結果がリストのいずれかの値に一致する場合、関連付けられたステートメントが実行されます。

定数値のリストが長く連続している場合、CASE ステートメントの定数を範囲で指定することもできます。範囲を指定することで、実行されるステートメントに関連付ける定数の連続した値のリストを指定することになります。

```
PROCEDURE Example_78;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
```

```
    48..57:Message('Number');
    58,59,60,61,62,63,64:Message('Non alpha printable character');
    65..90:Message('Upper case alpha');
    97..122:Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

範囲およびカンマで区切られたリストを混在させることで、実行可能なステートメントに関連付ける定数をより柔軟に指定できます。

```
PROCEDURE Example_78;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
    48..57:Message('Number');
    33..47,58..64,91..96:Message('Non alpha printable character');
    65..90:Message('Upper case alpha');
    97..122:Message('Lower case alpha');
    128,133,134,168..170:Message('Special characters');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

この例では、CASE ステートメントの定数指定に利用できる手法を生かして、分岐を行うための複雑なオプションを非常に簡潔な形式で表現しています。範囲とリストは、サポートされている他の定数タイプについても使用できます。

```
PROCEDURE Example_78;
VAR
j:CHAR;
BEGIN
j:= 'C';
CASE j OF
    '0'..'9':Message('Number');
    'A'..'Z':Message('Upper case alpha');
```

```
    'a'..'z':Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

CASE ステートメントは他のステートメントと同様、制御対象のステートメントとして複合ステートメントをサポートしています。この概念を拡張すると、ネストした CASE ステートメントを作成して、さらに複雑な分岐をスクリプトで処理できます。

```
PROCEDURE Example_78;
VAR
j:CHAR;
BEGIN
j:= 'C';
CASE j OF
    '0'..'9':Message('Number');
    'A'..'Z':Message('Upper case alpha');
    'a'..'z':Message('Lower case alpha');
    OTHERWISE BEGIN
        CASE Ord(j) OF
        33..47,58..64,91..96:Message('Non alpha printables');
        128..159:Message('Accented characters');
        168..170:Message('Special characters');
        OTHERWISE Message('Out of range');
        END;
    END;
END;
END;
Run(Example_78);
```

CASE ステートメントを使用する際は次の点に注意する必要があります。

- CASE ステートメントの定数値は制御式の値と同じタイプであることが必要です。
- 1 つの CASE ステートメントで異なる定数タイプを使用することはできません。

ユーザ定義関数 8

VectorScript では、API に組み込まれている 1,900 を超える関数呼び出しに加えて**ユーザ定義関数**を作成できます。カスタム関数を作成することで大量のスクリプトタスクを小さなタスクに分割でき、ゼロから作業を行うのではなく、以前に行った作業を利用することができます。ユーザ定義関数は**サブルーチン**とも呼ばれます。これは、名前が示すようにメインスクリプト内でタスクを実行するスクリプトコードです。

ユーザ定義関数には 2 種類あります。**手続き**は動作を実行しますが、値とは関連付けられていません。**関数**は動作を実行するだけでなく値も関連付けられており、定数や式から値を得る必要のある状況で使用できます。

ここでは、独自の手続きと関数を作成して使用する方法を詳しく解説すると共に、それらをスクリプト内で使用する場合に考慮すべき点を取り上げます。

ユーザ定義手続き

ユーザ定義手続きサブルーチンは最も一般的なタイプのサブルーチンです。このサブルーチンを使用すると、よく使われるコードを単一の識別子で「カプセル化」してスクリプト内から簡単に呼び出すことができます。

ユーザ定義手続きはスクリプトの定義ブロック（CONST、TYPE、および VAR）の後、ただしスクリプト本体の前で宣言します。スクリプト内で使用するユーザ定義手続きを作成するには、識別子とサブルーチンを関連付けてサブルーチンの使用方法を定義する手続き宣言ステートメントを作成する必要があります。ユーザ定義手続きの一般的な構文は次のとおりです。

```
PROCEDURE <手続き識別子>[(<パラメータリスト>)]
```

手続き宣言は PROCEDURE キーワードで始まり、サブルーチンブロックと関連付ける識別子が続きます。この識別子の後に手続きのパラメータリストを記述します。パラメータリストを使用するとデータをサブルーチンの内外に移動できます。また、リストの識別子はサブルーチンブロック内で変数のように使用できます。パラメータとパラメータリストについては、このセクションで再度詳しく説明します。

手続き宣言ステートメントを作成した後、サブルーチンの実際のコードを定義します。サブルーチンではスクリプトと同様に、標準 VectorScript 定義ブロック（LABEL、CONST、TYPE、または VAR）や、スクリプト内部の別の場所からサブルーチンが呼び出された時に実行するスクリプトコードを含むスクリプト本体を使用できます。次のようなスクリプトを利用する場合について説明します。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    sum:= n*(n+1)*(2*n+1)/6;
    Message('The sum of squares is:',sum);
```

```
END;
Run(SubrExample2);
```

いつでも必要な時に平方和のコードを簡単に再利用できるよう、このコードを編集します。これには演算を実行するコードをサブルーチン内に含める必要があります。サブルーチンを作成するには、まず手続き宣言ステートメントとサブルーチンの骨格を記述します。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN

END;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    sum:= n*(n+1)*(2*n+1)/6;
    Message('The sum of squares is:',sum);
END;
Run(SubrExample2);
```

宣言ステートメントは、識別子 SumOfSquares と新しいサブルーチンを関連付けます。サブルーチン識別子の次にサブルーチンのパラメータリストを記述します。このリストはオプションで、データをサブルーチンの内外に移動する方法を定義します。外側のプログラムブロックの値を直接参照することもできますが、この方法ではサブルーチンを他のコードで簡単に使用できなくなるため、サブルーチンを使用する主な利点の 1 つが失われます。

パラメータリストは、サブルーチンとの間でデータのやり取りに使われる一連の識別子（およびそれに関連付けられたデータタイプ）を宣言します。VAR キーワードは、データをサブルーチンから呼び出し元のコードに渡すための識別子を示します。パラメータリストの識別子は変数として扱われ、サブルーチンのスクリプトコード内で使用されます。

スクリプト内でサブルーチンが呼び出されると、宣言に含まれているパラメータリストは、パラメータを通してやり取りされるデータの受け渡しを行う変数識別子のリストに置き換えられます。変数識別子の順序とタイプは、宣言されたものと完全に一致している必要があります。

これでサブルーチンの骨格ができたため、和を求めるスクリプトコードをサブルーチン内に移動し、サブルーチンとして動作するよう修正します。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN
    result:= limit*(limit+1)*(2*limit+1)/6;
```

```
END;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    sum:= n*(n+1)*(2*n+1)/6;
    Message('The sum of squares is:',sum);
END;
Run(SubrExample2);
```

最後に、サブルーチンを使用するようスクリプトの本体を修正する必要があります。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN
    result:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    SumOfSquares(n,sum);
    Message('The sum of squares is:',sum);
END;
Run(SubrExample2);
```

メインプログラムブロックでスクリプトが呼び出されると、SumOfSquares のパラメータリストは変数 n と sum に置き換えられます。n に含まれている値はサブルーチンに渡され、サブルーチン内では limit という識別子で参照されます。結果の値はローカル識別子 result に格納され、サブルーチンの実行が完了するとメインプログラムブロックに戻されて sum という変数に格納されます。

サブルーチンを使用するとスクリプトを分割できるため、わかりやすくデバッグしやすい単位で管理できるようになります。SumOfSquares サブルーチンは同じスクリプトで必要なだけ再利用できるほか、コピーして他のスクリプトで使用することもできます。

ユーザ定義関数

ユーザ定義関数にはユーザ定義手続きのすべての機能が含まれていますが、スクリプトの作成時に非常に便利な追加機能として、値を関連付ける機能を備えています。ユーザ定義関数は手続きと異なり、サブルーチンから戻り値でデータを返すことができます。戻り値は値をサブルーチン識別子と関連付けます。つまり、関数は変数と同様、（式や代入ステートメントといったスクリプト内の演算のように）値が必要な場所であればどこでも使用できます。

ユーザ定義関数は手続きと同様、定義ブロックとスクリプトの本体の間で宣言します。ユーザ定義関数を作成するには、関数宣言ステートメントを使用して識別子をサブルーチンと関連付け、その使用法を定義します。ユーザ定義関数の一般的な構文は次のとおりです。

FUNCTION <**手続き識別子**>[(<**パラメータリスト**>)]:<**戻り値タイプ**>

宣言は手続きの場合と同様にキーワード（ここでは FUNCTION）で始まり、サブルーチンブロックと関連付ける識別子が続きます。次に関数のパラメータリストが置かれます。パラメータリストの機能はユーザ定義関数とユーザ定義手続きでまったく同じであるため、ユーザ定義手続きのパラメータリストに関する説明はすべてユーザ定義関数にも該当します。

ユーザ定義関数の宣言には追加条件があり、パラメータリストの後に戻り値のタイプを指定する必要があります。このデータタイプは、どういったタイプのデータが戻り値の仕組みを通して渡され、識別子と関連付けられるかを示します。

関数宣言を作成したら、ユーザ定義手続きの場合と同じ方法でサブルーチンの実際のコードを定義します。

手続きのサブルーチンと関数のサブルーチンの違いを示すため、前のセクションから平方和を計算する例を見てみます。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN
result:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    SumOfSquares(n,sum);
    Message('The sum of squares is:',sum);
END;
Run(SubrExample2);
```

SumOfSquares サブルーチンには簡単に再利用できるコードが用意されており、非常に便利です。ただし、結果はメインスクリプトに返されますが、値の取得方法をスクリプトコードから判断するのは困難です。この場合は関数サブルーチンの戻り値機構を使用すると、はるかにわかりやすくなります。関数サブルーチンを作成するには、まず宣言ステートメントにいくつかの変更を加えます。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
BEGIN
    result:= limit*(limit+1)*(2*limit+1)/6;
END;
```

```
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    SumOfSquares(n,sum);
    Message('The sum of squares is:',sum);
END;
Run(SubrExample2);
```

宣言に加える最初の変更として、サブルーチンの正しいタイプを示すためキーワードを PROCEDURE から FUNCTION に変更します。戻り値がサブルーチンの出力として使用されるため、出力パラメータの result はなくなります。次に戻り値のデータタイプを宣言に追加します。

宣言ステートメントを変更したら、戻り値をサブルーチン識別子と関連付けるためサブルーチンにもう 1 つ変更を加える必要があります。VectorScript は代入ステートメントを使用してこの関連付けを行いますが、ステートメントの左側に使用される識別子はサブルーチン識別子になります。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
BEGIN
    SumOfSquares:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    SumOfSquares(n,sum);
    Message('The sum of squares is:',sum);
END;
```

あとは関数の新しい構文に合わせてメインスクリプトを修正するだけです。

```
PROCEDURE SubrExample2;
VAR
    n,sum:INTEGER;
FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
BEGIN
    SumOfSquares:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
```

```
    n:=IntDialog('Enter the limit value','0');
    { 最初の n 個の整数の平方和 }
    sum:= SumOfSquares(n);
    Message('The sum of squares is:',sum);
END;
Run(SubrExample2);
```

見てわかるように、この例では関数サブルーチンを使用することでコードがはるかに読みやすくなり、サブルーチンのインターフェースも簡素化されています。一般に、計算などの操作の結果として値を返すサブルーチンには関数が最も適しています。手続きを使用するのは、値を返さない操作を行うサブルーチンを作成する場合です。

パラメータ

ユーザ定義のサブルーチンは VectorScript API の組み込み関数と同様、パラメータとパラメータリストを使用してデータ値をサブルーチンの内外に移動します。

仮引数と実引数

VectorScript の仮引数は、組み込み関数またはユーザ定義関数のパラメータリストで定義されているパラメータです。仮引数は関数のデータインターフェース「テンプレート」となるもので、関数呼び出しとの間でやり取りする値の順序とタイプを示しています。実引数は、スクリプトの本体で関数を使用して渡される式または値を指しています。サブルーチン SumOfSquares の宣言ステートメントの場合を例にとって説明します。

```
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
```

識別子の制限と結果はどちらもサブルーチン手続きの仮引数です。SumOfSquares がスクリプトで次のように使用されるとします。

```
SumOfSquares(n,sum);
```

サブルーチン手続きには n と sum という 2 つの実引数があります。これらの実引数には、関数呼び出しで使用されるデータと返されるデータが含まれています。スクリプトの VAR ブロックをチェックすると、2 つの識別子のデータタイプが仮引数リストにあるタイプと一致することがわかります。

値と変数パラメータ

VectorScript の値パラメータは、データの値をサブルーチンに渡すために使用されるパラメータです。これらのパラメータはサブルーチン内でローカル変数と同じように機能しますが、パラメータリストの対応する実引数から初期値を取得する点が異なります。SumOfSquares の例では次のようになります。

```
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
```

識別子 limit は値パラメータ、より正確には仮引数です。メインスクリプトで次の関数呼び出しが使用されるとします。

```
SumOfSquares(n,sum);
```

変数 n に含まれている値は値パラメータ limit に割り当てられ、サブルーチン内で使用されます。

VectorScript の変数パラメータは値パラメータとは逆に、データの値をサブルーチンから返すために使用されます。変数パラメータはパラメータリストの中で前に置かれる VAR キーワードで示され、値パラメータと同様にサブルーチン内でローカル変数として機能します。SumOfSquares の例では次のようになります。

```
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
```

識別子 result は変数仮引数であり、サブルーチンのスクリプトコード内で呼び出し元のコードに値を返すために使用されます。メインスクリプトで次の関数呼び出しが使用されるとします。

```
SumOfSquares(n,sum);
```

変数パラメータ result に含まれている値は、サブルーチンの実行完了時に変数 sum に割り当てられます。

プログラムブロックとブロックスコープ

2 ページの『スクリプトの例』で説明しているように、スクリプトはプログラムブロックとも呼ばれ、VectorScript ソースコードの基本単位となっています。プログラムブロックを構成するのは、ブロック宣言ステートメント、ブロック内でデータの宣言や定義を行う CONST、TYPE、VAR などのセクション、および実行される VectorScript のソースコードを含むブロック本体です。ユーザ定義関数ではこの概念を拡張しており、スクリプトのメインプログラムブロック内により小さなプログラムブロックをネストできます。

スクリプト内で使用される各サブルーチンは自己完結型のプログラムブロックで、独自のデータ宣言と本体を備えています。サブルーチンブロックには、他のデータ宣言およびスクリプトコードと共に独自のサブルーチンブロックをネストできます。このようにサブルーチンブロックをネストする場合はブロックスコープという重要な概念が問題となります。スクリプトでサブルーチンを作成する時は、必ずブロックスコープを考慮する必要があります。

ブロックスコープは、スクリプトの中で特定の識別子が有効と見なされて定義済みの値が関連付けられた範囲を示します。プログラムブロック内で変数、定数、または構造体が宣言されている場合、アイテムはそのプログラムブロックに対して**ローカル**であると見なされます。つまり、アイテムは宣言されたブロック内およびそのブロックに含まれる領域でのみ有効であり、値が定義されています。以下に例を示します。

```
PROCEDURE Main;
    Subroutine “A”()
        Subroutine “B” ()
        BEGIN
        END;
    BEGIN
    END;
    Subroutine “C”()
    BEGIN
    END;
BEGIN
END;
```

この例では、識別子のスコープはその位置で判定されています。

識別子の宣言位置	識別子のスコープ
Main	Main,A,B,C
Subroutine "A"	A,B
Subroutine "B"	B
Subroutine "C"	C

識別子は宣言されたプログラムブロックの外側では未定義と見なされ、ブロックの外側のスクリプトコードではアクセスや参照はできません。識別子が宣言されているブロックがサブルーチンの場合、そのサブルーチンを囲むブロックでは識別子が未定義になります。サブルーチンを囲むブロックのソースコードからそのアイテムへの参照や評価を試みると、エラーが発生したりスクリプトの実行に失敗したりします。

次の例も、ブロックスコープの概念を示したものです。

```
PROCEDURE WoodPrice;
CONST
    kTax:=0.05;
VAR
    boardFeet,price,totalCost:REAL;
PROCEDURE CalcCost(feet,ppf:REAL; VAR cost:REAL);
VAR
    baseCost:REAL;
FUNCTION AddTax(rawcost:REAL):REAL;
BEGIN
    AddTax:= rawcost+(rawcost*kTax);
END;
{ CalcCost コードの開始 }
BEGIN
    baseCost:= feet*ppf;
    cost:= AddTax(baseCost);
END;
{ CalcCost コードの終了 }
{ メインスクリプトの開始 }
BEGIN
    boardFeet:= RealDialog('Enter no. of feet','0');
    price:= RealDialog('Enter price per foot','0');
    CalcCost(boardFeet,price,totalCost);
    Message('Total cost is $',totalCost:6:2);
END;
```

```
{ メインスクリプトの終了 }
Run(WoodPrice);
```

この例では 3 つのプログラムブロック、つまりスコープの領域が存在します。最大のブロックはメインスクリプト WoodPrice です。この中にはサブルーチンブロック CalcCost があり、CalcCost の中にはサブルーチン関数とプログラムブロック AddTax があります。

WoodPrice ブロック内で定義されている変数や定数の識別子はすべて WoodPrice スクリプトコード内で参照できるほか、同ブロック内で宣言されている任意のサブルーチン内からも参照できます。これらのアイテムはスクリプトのトップレベルで定義されているためグローバルスコープがあると見なされ、スクリプト内の任意のサブルーチンからアクセスできます。

CalcCost サブルーチン内で定義されている識別子は（宣言ステートメント内の識別子を含め）、CalcCost サブルーチン、または AddTax 関数の中で参照できます。ただし CalcCost スコープの外側にある WoodPrice ブロックでは、これらの識別子は未定義となります。つまり baseCost やサブルーチン AddTax などのアイテムは、WoodPrice スクリプトの本体から直接参照することはできません。

AddTax サブルーチンで定義された識別子には、スクリプトのどのブロックよりも小さなスコープがあります。これらの識別子は、そのサブルーチン内のコードでのみ使用できます。また、これらの識別子は CalcCost および WoodPrice プログラムブロックでは未定義となり、参照できません。この例で、kTax 定数は AddTax 関数で直接参照できます。これは kTax がメインスクリプトで定義されており、グローバルスコープを含んでいるためです。ただし AddTax の結果は CalcCost サブルーチン内で宣言されており、そのサブルーチン内でのみ有効なため、メインスクリプトからは直接アクセスできません。

コンパイラディレクティブ 9

VectorScript は、スクリプトのコンパイルと実行の方法を制御するコンパイラディレクティブをサポートしています。

{$INCLUDE}

INCLUDE ディレクティブは、INCLUDE ディレクティブステートメントの位置に外部ファイルのソースコードを挿入するようコンパイラに指示します。INCLUDE ディレクティブの構文は次のとおりです。

```
{$INCLUDE <ファイルパス>}
```

VectorScript ソースコードを含むファイルのパスは、完全指定または部分的なファイルパスで指定できます。Macintosh 形式のパス区切り記号（:）と Windows 形式のパス区切り記号（¥）がサポートされています。すべてのプラットフォームでの互換性を保証するため、クロスプラットフォーム環境で使用される可能性があるスクリプトでは、Windows 形式の区切り記号を推奨します。

例：Macintosh 形式の INCLUDE ディレクティブ

```
{$INCLUDE MyHD:Vectorworks:Projects:VS:mycode:math.vss}
```

例：Windows 形式の INCLUDE ディレクティブ

```
{$INCLUDE MyHD¥Vectorworks¥Projects¥VS¥mycode¥math.vss}
```

パス情報を記述せずに指定した参照ファイルは、スクリプトを起点とする既定のデフォルトパスにあると見なされます。ドキュメントスクリプトおよびテキストファイルから実行されるスクリプトの場合、デフォルトパスは Vectorworks アプリケーションの場所です。プラグインの場合、デフォルトパスは関連付けられたプラグインが置かれているフォルダと見なされます。

別の参照ファイル内に INCLUDE ディレクティブを指定して、INCLUDE ステートメントは入れ子にすることもできます。INCLUDE ディレクティブの入れ子は、これが原因でファイルに依存関係が生まれ、特定の状況でスクリプトエラーが発生する場合があるため、注意して使用する必要があります。

また、INCLUDE ディレクティブをスクリプト内に置く時は、関数をスクリプト内で定義する前に呼び出さないよう注意する必要があります。

{$DEBUG}

DEBUG ディレクティブは、スクリプトをコンパイルして実行する時に VectorScript デバッガを起動するようコンパイラに指示します。その後、スクリプトの開発中にデバッガを使用してスクリプトの実行を観察し、制御することができます。DEBUG ディレクティブの構文は次のとおりです。

```
{$DEBUG}
```

このディレクティブは、スクリプトのメインブロックの任意の場所に置いてデバッガを起動できます。

デバッガの使用法に関する詳細は 115 ページの『VectorScript デバッガ』を参照してください。

{$NAMES}

NAMES ディレクティブは、コンパイラディレクティブで指定されたバージョンの Vectorworks で有効な識別子のみを認識するようコンパイラに指示します。このディレクティブで振り分けられる識別子には、手続き、関数、および定数の識別子があります。NAMES ディレクティブの構文は次のとおりです。

{$NAMES <**バージョン番号**>}

指定されたバージョンの製品用に定義されていない識別子は VectorScript エラーを生成します。NAMES ディレクティブは、異なるバージョンの Vectorworks との間でスクリプトの互換性をテストすることを目的としています。

例：NAMES ディレクティブ

```
{$NAMES 8}
```

この例で、VectorScript コンパイラは Vectorworks 8 で有効な識別子のみを認識します。（後のバージョンで追加された新しい関数など）コンパイラでサポートされていない識別子名はエラーが返されます。ディレクティブで指定されたバージョンとの互換性が必要なスクリプトでは使用できません。

{$STRICT}

STRICT ディレクティブは、コンパイラディレクティブで指定されたバージョンの Vectorworks で有効な構文と語義規則を認識して従うようコンパイラに指示します。STRICT ディレクティブの構文は次のとおりです。

{$STRICT <**バージョン番号**>}

指定されたバージョンに対して有効でない構文は VectorScript エラーを生成します。STRICT ディレクティブは、異なるバージョンの Vectorworks との間でスクリプトの互換性をテストすることを目的としています。

例：STRICT ディレクティブ

```
{$STRICT 7}
```

この例で、VectorScript コンパイラは MiniCAD 7 で有効な構文規則のみを認識します。（動的配列や構造体など）このバージョンで有効でない新しい構文規則はエラーが返されます。ディレクティブで指定されたバージョンとの互換性が必要なスクリプトでは使用できません。

条件付きコンパイルディレクティブ

スクリプト作成者は VectorScript の 4 つのコメントディレクティブを使用することで、VectorScript 内のコンパイルされる部分とスキップされる部分を区分できます。これにより、複雑なプロジェクトの構築作業が改善します。また、機能修正のテストが容易になります。

これらのディレクティブは、特定のコードブロックをコンパイルすべきかどうかを決定します。ディレクティブは、コンパイラがスクリプトの処理を開始するより前に処理されます。ディレクティブは、スクリプトのコメントを記述できる任意の場所に配置できます。

{$IF}

このディレクティブはコードブロックの開始位置を定義します。このディレクティブの構文は次のとおりです。

{$IF <**条件**>}

この構文には評価の対象となる論理ステートメントが含まれています。評価の結果が FALSE の場合、{$ENDIF} ディレクティブまで（またはファイルの末尾まで）のコードブロックはコンパイラにスキップされます。TRUE の場合、コードはコンパイルされます。

組み込み変数の ver が {$IF} 条件の中で使用できます。ver の値は Vectorworks ソフトウェアのバージョンに対応しています。17 は Vectorworks バージョン 2012、18 は Vectorworks バージョン 2013 を示します。

{$ENDIF}

このディレクティブは条件が記述されているコードブロックの終了位置を定義します。{$IF} ディレクティブと対になる終了ディレクティブです。両者は同じ参照ファイル内に配置する必要があります。

このディレクティブの構文は次のとおりです。

{$ENDIF}

この例で {$IF} ディレクティブと {$ENDIF} ディレクティブは、指定したバージョン番号に基づいてコードを制御しています。この例は、Vectorworks ソフトウェアのバージョンに応じて異なるダイアログボックスが表示されることを示しています。

```
PROCEDURE Test;
BEGIN
{$IF ver = 17}
    AlrtDialog('Code for Vectorworks 2012');
{$ENDIF}

{$IF ver > 17}
    AlrtDialog('Code for versions later than Vectorworks 2012');
{$ENDIF}

END;
RUN(Test);
```

{$DEFINE}

このディレクティブでは名前付きの値を定義し、{$IF} ディレクティブ条件の中で使用できます。名前を付けられた変数は定義された時点からスクリプトの最後、または未定義になるまで有効です。

このディレクティブの構文は次のとおりです。

{$DEFINE <**名前**> = <**値**>}

この例では {$DEFINE} ディレクティブを使用してコードを制御しています。

```
PROCEDURE Test;
    {$DEFINE library = 1}
```

```
    {$IF library = 1}
        {$INCLUDE Code/Library.px}
    {$ENDIF}

BEGIN
    {$IF library = 1 & ver > 17}
        LibraryFunction( 34 );
    {$ENDIF}
END;
RUN(Test);
```

{$UNDEF}

このディレクティブは以前に定義された名前付き変数を削除します。このディレクティブの構文は次のとおりです。

{$UNDEF <**名前**>}

索引

り

VectorScript Language Guide

平成２３年１１月　５日　初版発行
平成２５年　９月３０日　改訂

著作
Nemetschek Vectorworks Inc./A&A Co., Ltd.
製作・発行
エーアンドエー株式会社
〒 101-0062 東京都千代田区神田駿河台 2-3-15