

4D Compiler

リファレンス

Windows および Mac OS 版



4D Compiler
by
ACI S.A.

4D Compiler リファレンスマニュアル

Windows® 及び Mac™ OS 版バージョン 6.5

Copyright©1985-1999 ACI S.A.
All rights reserved

このマニュアルに記載されている事項は、将来予告なしに変更されることがあり、いかなる理由に関しても ACI SA は一切の責任を負いかねます。このマニュアルで説明されるソフトウェアは、本製品に同梱の License Agreement (使用許諾契約書) のもとでのみ使用することができます。

ソフトウェアおよびマニュアルの一部または全部を、ライセンス保持者がこの契約条件を許諾した上での個人使用目的以外に、いかなる目的であれ、電子的、機械的、またどのような形であっても、無断で複製、配布することはできません。

4th Dimension, 4D, 4D ロゴ, 4D Server, 4D Write, 4D Draw, 4D Calc, 4D Backup, 4D Insider, 4D Compiler, ACI, ACI ロゴ は、ACI SA の登録商標または商標です

Microsoft と Windows は、Microsoft Corporation 社の登録商標です。

Apple, Macintosh, Mac, Power Macintosh, LaserWriter, ImageWriter, ResEdit、及び QuickTime は Apple Computer 社の登録商標です。

その他、記載されている会社名、製品名は、各社の登録商標または商標です。

注意：

このソフトウェアの使用に際し、本製品に同梱の License Agreement (使用許諾契約書) に同意する必要があります。ソフトウェアを使用する前に、License Agreement を注意深くお読みください。

目次

序	章	V
	本マニュアルについて	v
	章の詳細	v
	このマニュアルの使い方	vi
	ハイパーテキストナビゲーション	vi
	規約	vii
1 章	はじめに	1
	4D Compiler の長所	1
	4D Compiler の最適な使用	2
	インタプリタ対コンパイラ	3
	インタプリタ	3
	コンパイラ	3
	4D Compiler (Macintosh および Windows)	4
	4D Compiler と 4 th Dimension	4
	対話式デバッグ	4
	なぜデータベースをコンパイルするのか	5
	実行速度	5
	コードのチェック	6
	コンパイル後のデータベースの使用	7
	データベースの保護	7
2 章	コンパイラの使用	9
	データベースの使用	9
	4D Compiler のメニュー	10
	新規	11
	開く	12
	再コンパイル	14
	閉じる	14
	保存	15
	名前をつけて保存	15
	元に戻す	15
	デフォルトプロジェクト	15
	終了	16
	コンパイルオプション	17
	オプションウィンドウ	17
	コンパイルデータベース名	17
	実行形式アプリケーションの作成	18
	エラーファイル	21
	シンボルテーブル	22
	範囲チェック	23

スクリプトマネージャ	23
警告	23
プロセッサの種類	24
追加オプション	25
最適化	26
ローカル変数初期化	26
タイプファイル	27
コンパイルパス	
（未定義変数タイプチェックの仕方）	28
デフォルトボタン変数タイプ	28
デフォルト数値型変数タイプ	28
デフォルト文字変数タイプ	29
バージョン番号自動生成	29
オプションについて	30
Windows のヘルプ	30
コンパイルの開始	30
コンパイル	31
データベースのコピー	31
変数のタイプ設定	32
コンパイル	33
4D Engine のコピー	34
コンパイル後のデータベースの使用	34
ドラッグ&ドロップ	34
4D Compiler を 4th Dimension と共に使用する	35
PowerPC、及び 80x86 用のコンパイル	35
3 章 診断ツール	37
シンボルテーブル	38
プロセスとインタープロセス変数のリスト	38
ローカル変数のリスト	40
メソッドのリスト	40
エラーファイル	41
メッセージのタイプ	41
エラーファイルの使用	44
タイプファイル	46
範囲チェック	46
範囲チェックの使用	47
異常診断	48
4 章 コンパイル用のデータベースの準備	49
変数及び配列のデータタイプ	50
変数のタイプ	50
シンボルテーブル	51
4D Compiler による変数のタイプ設定	51
コンパイラ命令	52
コンパイラディレクティブが必要な時	53
コードの最適化	56
コンパイルの時間の短縮	56

実数と文字列を使う	56
インタプリタでコンパイラ命令を使う	56
コンパイラ命令をどこに記述するか	57
C_STRING コンパイラ命令	59
まとめ	60
5 章 タイプ設定ガイド	61
インタープロセス変数とプロセス変数	61
2 種類の用途による矛盾	62
用途とコンパイラ命令の矛盾	62
暗黙のタイプ変更による矛盾	62
2 つのコンパイラ命令による矛盾	63
ローカル変数	64
配列内の対立	64
配列要素のデータタイプの変更	64
配列の次元数の変更	65
文字配列	65
暗黙のタイプ変更	65
ローカル配列	65
フォーム変数	66
数値タイプに設定されるフォーム変数	66
グラフ変数	66
プラグインオブジェクト変数	66
テキストタイプに設定されるフォーム変数	66
ポインタ	67
プラグインコマンド	68
マルチプラットフォームコンパイル	68
Macintosh あるいは Windows での実行形式のアプリケーションのコンパイルと作成	72
暗黙の引数を受け取るプラグインコマンド	73
プラグインコマンドで作成される変数	73
引数の処理	74
ポインタを使ってデータタイプの矛盾を避ける	74
引数の参照	75
予約変数	76
システム変数	76
クイックレポート変数	77
定数	77
6 章 コマンドの説明	91
配列	91
COPY ARRAY	91
SELECTION TO ARRAY, ARRAY TO SELECTION, DISTINCT VALUES, SUBSELECTION TO ARRAY	92
LIST TO ARRAY, ARRAY TO LIST	92
配列に関連したコマンドでのポインタの使用	93
ローカル配列	93
コミュニケーション	93
データエントリー	94

例外	94
文書	95
演算	96
ブレイク処理	96
文字列	96
ストラクチャへのアクセス	96
変数その他	97
Undefined コマンド	97
SAVE VARIABLE コマンドと LOAD VARIABLE コマンド	97
CLEAR VARIABLE コマンド	97
Get Pointer コマンド	98
EXECUTE コマンド	99
Trace コマンドと No Trace コマンド	99
各種のコマンドで使われるポインタ	100
7 章 最適化のためのヒント	103
コードに関するコメント	103
コンパイラ命令の使用によるコードの最適化	104
数値変数	104
文字	105
その他のヒント	106
ポインタ	107
ローカル変数	107
警告メッセージ	110
詳細警告メッセージ	111
エラーメッセージ	111
タイプチェック	112
シンタックス	114
引数	116
演算子	117
プラグインコマンド	118
総合エラー	118
範囲チェックメッセージ	119
コンパイラーメッセージ	121
Macintosh	123
データベースアイコンのカスタマイズ	124
他のデータベースファイル用アイコンのカスタマイズ	124
Windows	125
アプリケーションのアイコンをカスタマイズする	125
アプリケーションのウィンドウ名の変更	126
表記	128
索引	129

序 章

4D Compiler は、4th Dimension 用のコンパイラです。従来のコンパイラでは見られないような広範囲にわたる診断の警告やエラーメッセージを提供します。Windows 用および Macintosh 用の両方のバージョンが用意されています。

注 4D Compiler バージョン 6.5 は、4th Dimension のバージョン 6.5、および 4D Server バージョン 6.5 により開発されたデータベースをコンパイルします。4th Dimension バージョン 3 で作成されたデータベースを使用する場合には、4th Dimension バージョン 6.5 で開いてコンバートしたのち、4D Compiler バージョン 6.5 でコンパイルします。アップグレードについては ACI までお問い合わせください。

本マニュアルについて

本マニュアルは、4D Compiler のアプリケーションについて述べています。

章の詳細

本マニュアルには下記の章があります。

- **第 1 章「はじめに」**では、コンパイラの紹介、およびコンパイラとインタプリタの違いについて説明します。
- **第 2 章「コンパイラの使用」**では、4D Compiler の特長とオプションについて述べます。
- **第 3 章「診断ツール」**データベースをデバッグする際に便利な 4 つのコンパイラツール、“シンボルテーブル”、“エラーファイル”、“範囲チェック”、“タイプファイル”について説明します。
- **第 4 章「コンパイル用データベースの準備」**では、コンパイルするデータベースを作成する際の規則について説明します。

- [第5章「タイプ設定ガイド」](#)では、タイプ（型）の矛盾を引き起こす場合の例を説明します。
- [第6章「シンタックスの詳細」](#)では、コンパイル関連のコマンドについての追加情報を説明します。
- [第7章「最適化のヒント」](#)では、コードのパフォーマンスを最適化するためのヒントを紹介합니다。

付録では、コンパイラの補足説明をしています。

- [付録A「コンパイラメッセージ」](#)では、コンパイラが表示するメッセージの完全なリストをサンプルコードとあわせて紹介します。
- [付録B「アプリケーションのカスタマイズ」](#)では、アプリケーションをカスタマイズする際の情報を提供します。
- [付録C「リソースのカスタマイズ」](#)では、4D Compiler をカスタマイズする際の Customizer Plus の使用法を説明します。


このマニュアルの 使い方

まず、第1章、第2章、第3章を読んでコンパイラの機能を理解します。第2章で紹介されている機能やオプションについて読んでいくうちに、実際のデータベースをコンパイルしてみたくなるでしょう。しかし、一度目ではデータベースをコンパイルできないかもしれません。続く各章を読み進んで、コンパイラが出力するエラーメッセージやシンボルテーブルについて学習していきます。エラーメッセージの説明については、付録Aを参照してください。

コンパイルに適したコードの記述方法の詳細は、第4章、第5章、第6章をお読みください。最後の第7章には、コードの最適化に関するヒントがあります。

ハイパーテキスト ナビゲーション

本マニュアルを電子データ形式（Adobe AcrobatTM PDF）でお読みになっている場合には、ハイパーテキストリンクを利用することが可能です。ブルーで表示されている単語はハイパーテキストリンクを持っています。ただし、「目次」および「索引」には適用されません。これはすでにハイパーテキストリンクを持っているためです。

ハイパーテキストリンクをクリックすると、より多くの情報を持ったページへ即座に移動することができます。元のページに戻るには、「前の表示」ボタン  をクリックします。

また、マニュアルのページを表示しているウィンドウの左側にある「しおり」をクリックすることにより文書中を移動することもできます。

規 約

クロスプラットフォームについて 本マニュアルは、4D Compiler の Windows および Macintosh の両バージョンに関する情報をカバーしています。それぞれのバージョンのコンセプトや機能はほぼ同一ですが、必要な差異については記されています。また、本マニュアル中の画面表示は Windows 環境での 4D Compiler によるものです。Macintosh と Windows で大幅に異なる場合には、両バージョンの画面表示を掲載しています。

記載に関する規約 本マニュアルを含め、パッケージ内のすべてのマニュアルは、わかりやすいように一定の表記を使用しています。

次のような表記が使われています

注 このように強調された文章は、ソフトウェアをより効果的に使用するための注釈や近道を提供するものです。

このような注意書きは、重要な情報を表わしています。

4D Server 本マニュアル全体を通じて、4th Dimension および 4D Server/4D Client は、まとめて「4th Dimension」として参照されています。4D Server/4D Client における運用上の違いがある場合には、このような形式の注釈で説明されています。

覚えていて下さい このように強調された文章は、4D Compiler を使用する際に遭遇する可能性のある特定事項を示します。

1

はじめに

本章では、下記のような 4D Compiler についての基本的な情報を提供します。

- 4D Compiler の長所
- 4D Compiler を最適に使用するためのヒント
- インタプリタとコンパイルの根本的な違い
- 4D Compiler 製品の説明
- 4D Compiler における 4th Dimension の対話式デバッグのサポート
- なぜデータベースをコンパイルするのか
- コンパイルされたデータベースの使用

4D Compiler の長所

4D Compiler には次のような長所があります。

- データベースを系統的に分析し、広範囲にわたる診断警告やエラーメッセージを提供
- コンパイルされたデータベースの実行中に、データベースの動的チェックが可能
- 4th Dimension との対話式デバッグをサポート。
- メソッドをコンパイルし、真のマシン語を生成
- 次のようなプロセッサ用に最適化されたコードを生成：Motorola PowerPC プロセッサ（PowerPC 601/603/604/G3）および PC プロセッサ（386/486、ペンティアム最適化）
- コンパイルされたデータベースを 4D Engine とマージし、スタンドアロンの実行形式アプリケーションを作成可能。
- データベースで使用される全ての変数とそのコンパイラ命令のリストを生成。
- アクティブオブジェクト、数値、および文字列のデフォルトのデータタイプを設定。

コンパイルされるデータベースは、標準の「開く」ダイアログボックスにより開きます。これ以降は、コンパイラがすべての処理を行います。まずストラクチャファイルを複製します。次にデータベースを系統的に分析し、説明的かつ診断的なレポートを作成します。その後メソッドをマシン語に翻訳し、必要に応じてエラーファイルを作成します。新しいストラクチャファイルは、デザインモードに入れない点を除けば、元の（コンパイルされていない）ストラクチャファイルとまったく同様に使用することができます。コンパイルされたデータベースの実行速度は劇的に向上し、3 倍から 1000 倍、あるいは条件によってはそれ以上に向上する場合があります。

4D Compiler は、機械のマイクロプロセッサに適応した真の機械言語を発生させると同時に、それがインストールされていれば、マス・コプロセッサも考慮に入れております。

- 注 4D Compiler バージョン 6.5 は、4th Dimension バージョン 6.5 及び 4D Server バージョン 6.5 と共に開発された全てのデータベースをコンパイルします。4th Dimension バージョン 3 で作成されたデータベースを使用する場合は、まず 4th Dimension バージョン 6.5 で開いてコンバートしてください。その後、4D Compiler バージョン 6.5 でコンパイルします。アップグレードについては ACI までお問い合わせ下さい。

4D Compiler の最適な使用

4D Compiler の最適な使用を行なうためには、下記の点にご注意下さい。

- 効率良く記述されていないためにインタプリタでのパフォーマンスがよくないメソッドは、コンパイルしても速度が向上しない場合があります。また、そのようなコードはコンパイルできないこともあります。
エラーを検出すると、4D Compiler はコンパイルを中止します。
- 4th Dimension で正常に作動するデータベースが常にコンパイル可能であるとは限りません。4th Dimension のインタプリタは、意味や文法の矛盾に対してある程度寛容です。一方、4D Compiler は、コンパイラとしてはまれに見るほど融通がききますが、インタプリタほど柔軟ではありません。これは、インタプリタとコンパイラの本質的な違いによるものです。

4th Dimension のインタプリタとは異なり、コンパイラはそれぞれの変数が必ずひとつのデータ型に割り当てられていなければなりません。コンパイラは、それぞれの変数を正確にタイプ分けする必要があります。タイプの割り当てに関して不明確であることは許されません。たとえば、あるステートメントでは実数として使用されている変数が、他のステートメントではテキストとして使用されている場合、コンパイラはエラーメッセージを出力します。

データベース内のプロセス変数やインタープロセス変数のタイプは変更できません。また、同じメソッド内においてローカル変数のタイプは変更できません。

コンパイルするデータベースを準備するための作業の多くは、これらの条件に合致させることに集中しています。コンパイラには、データのタイプに関する問題を発見し解決するための助けとなるいくつかの優れた診断ツールが含まれています。

ある種の変数については、タイプの決定に際していくつかの選択肢があります。つまり、数値変数には実数、整数、および倍長整数の 3 つのタイプがあり、文字変数にはテキストと文字列（ストリング）の 2 つのタイプがあります。それぞれの変数について最適なタイプを選ぶことにより、コンパイルされたデータベースのパフォーマンスを向上することができます。倍長整数の変数を使用すると、実数の変数を使用するよりも、メソッドをより速く実行することができます。

インタプリタ対コンパイラ

コンピュータは、コマンドが「0」と「1」のみを使用して書かれた装置です。この2進数の言語は「マシン語」と呼ばれています。機械の心臓部であるマイクロプロセッサは、この言語しか理解することができません。

高級言語（C、Pascal、Basic、4th Dimension など）で書かれたプログラムは、コンピュータのマイクロプロセッサが理解できるように、まずマシン語に翻訳されません。

これを行うには2つの方法があります。

- ステートメントを1行ずつ翻訳しながら実行する。これがインタプリタです。
- ステートメント全体をプログラムの実行前に翻訳する。これがコンパイラです。

インタプリタ

一連のステートメントをインタプリタを使って実行する場合、その処理は下記のような段階に分けることができます。

- プログラム言語で書かれたステートメントを1つ読み込む。
- ステートメントをマシン語に翻訳する。
- ステートメントを実行する。

このサイクルは、プログラム中の各ステートメントについて実行されます。4Dを使用する際は、常にメソッド内のステートメントが翻訳されています。

コンパイラ

コンパイルされるプログラムは実行前に全体が翻訳され、この処理により一連のマシン語のステートメントを含んだ新しいファイルが生成されます。翻訳は一度だけ行なわれ、コンパイルされたプログラムは繰り返し実行することができます。

この方法での不利な点は、プログラムのデバッグや変更をオリジナル、つまりソースコードに対して行なわれなければならないということです。その後でプログラム全体を再コンパイルしなければなりません。またコンパイル後のプログラムは、修正不可能です。このため、4D Compiler はオリジナルのデータベースを上書きしないので、必要に応じて修正や再コンパイルが可能になっています。

コンパイラの利点は、「なぜデータベースをコンパイルするのか」のセクションで説明します。

4D Compiler (Macintosh および Windows)

4D Compiler は、Macintosh PPC、Windows 386/486 および Pentium を含むいくつかの異なるプラットフォーム用のマシン語を生成します。4D Compiler で作成するデータベース製品は、これらのプロセッサオプションの一部あるいはすべてを含むことが可能です。4D Compiler でコンパイルされたデータベースは、4th Dimension あるいは 4D Server とともに使用します。

また、配付自由な実行可能形式アプリケーションの作成も可能です。4D Compiler により、コンパイルされたデータベースと 4D Engine をマージし、スタンドアロンで実行可能アプリケーションを作成することができます。

4D Compiler と 4th Dimension

4D Compiler は、すべてのデータベース、プロジェクト、表 (トリガ)、フォーム、およびオブジェクトメソッドをコンパイルします。ユーザは、「ユーザ」モードと「カスタム」モードのどちらからでもデータを管理することができます。オリジナルのストラクチャファイルをそのまま残し、まったく同様に使用できる新しい (コンパイルされた) ストラクチャファイルを作成します。

4D Compiler は、エラーや不明な点を発見すると、データベースをコンパイルすることができません。エラーはテキストファイルで出力されます。この場合は、元の (コンパイルされていない) データベース内のエラーを修正して、再度コンパイルを行ないます。

対話式デバッグ

4D Compiler の特徴的な機能のひとつに、データベースのデバッグ時にエラーファイルを対話式に使用することができるという点があります。エラーファイルとコンパイルされていないデータベースの両方を、4th Dimension で同時に開くことができます。4th Dimension のメニューコマンド「次のコンパイルエラー」は、自動的に次のコンパイルエラーを見つけてハイライトさせ、エラーメッセージや警告を表示します。

なぜデータベースをコンパイルするのか

コンパイルすることの第一の利点はもちろん実行速度の向上です。他にもコンパイルに直接結びついた利点が2つあります。

- 体系的なコードチェック
- データベースの保護

実行速度

実行速度の向上は、コンパイルされたコードの二つの特徴によるものです。

- ただ一度で全体をカバーする、直接のコード翻訳
- 変数および方法のアドレスへの直接アクセス

直接的で最終的なコード翻訳

4D Compiler はソースコードを翻訳し、実行形式のファイルとして保存します。よって、インタプリタモードですべてのステートメントを翻訳するのに必要な時間が省略できます。

この点を示した簡単な例があります。50回繰り返される一連のステートメントを含んだループがあるとします。

```
For ($:;1;50)
  `一連のステートメント
```

End for

インタプリタでは、ループ内の各ステートメントはそれぞれ50回翻訳されます。コンパイラを使用すると各ステートメントの翻訳作業がなくなるため、ループ内のステートメントの50回分の翻訳が省略できることとなります。

もう一つの例を挙げましょう。データベースに On Startup データベースメソッドが含まれていると仮定します。このメソッドは、データベースを起動するたびに実行されます。しかし、一度コンパイルすれば、On Startup データベースメソッドを毎回翻訳するのに必要な時間を省略することができます。

以上、たったの2つの例ですが、この利点はすべてのメソッドのすべてのステートメントに当てはまります。

変数アドレスとメソッドアドレスへのダイレクトアクセス

インタプリタでのメソッドにおいては、変数は名前によりアクセスされます。したがって、4th Dimension は変数の値を得るためには名前にアクセスする必要があります。

コンパイルされたコードでは、コンパイラは各変数にアドレスを割り当て、変数のアドレスを直接コードに書き込むため、そのアドレスに直接アクセスすることが可能です。

注 ディスクアクセスを要する処理では、コンパイルによる速度向上の恩恵をあまり受けないことがあります。これは、コンピュータとハードディスク間の転送レートにより制限されてしまうためです。

コメントは翻訳されないので、コンパイルされたコードには現れません。よって、コメントはコンパイル後のデータベースの実行速度には影響しません。

次の表は、コンパイラとインタプリタでの実行時間を示します。これらのテストは PowerMacintosh 7100 で行なわれました。

シーケンス	インタプリタ	コンパイラ	速度比
単純な For ループを 20 万回繰り返す：			
For (\$i;1;200000) \$total:=\$i End for	4 分 50 秒	1 秒	約 300 倍
メモリーにキャッシュされている 10,000 件のレコードセレクションの文字列操作			
For (\$i;1;10000) \$x:=Substring([Table1]first;1;5) NEXT RECORD([Table1]) End for	1 分 42 秒	2.17 秒	47 倍

コードのチェック

コンパイラは、データベースをコンパイルする前に、メソッドに対して論理的かつ語彙的なチェックを行います。コードを系統的にチェックし、不明確な部分があれば警告します。一方、4th Dimension のインタプリタはメソッドが実行された場合にのみこの処理を行いません。ここで重要な違いは、データベースの通常の使用において実際に実行されるかどうかに関わらず、コンパイラは系統的にすべてのコードをチェックするという点です。

たとえば、メソッド内にさまざまな判定処理が含まれていると仮定します。判定するケースが非常に多い場合、すべてのケースに対して十分なテストを行なうことは困難です。もしテストしていないケースにエラーが含まれていたとしても、エンドユーザがそのケースに遭遇するまではわかりません。

4th Dimension のインタプリタはコンパイラよりも寛大であるため、インタプリタで問題なく動作しているデータベースの中には、最初の試みではコンパイルできないものもあります。例えばデータのタイプに関する矛盾が見つかるかもしれません。しかし、コンパイラの特徴である対話式デバッグ機能を利用して、非常に簡単に解決することができます。

データベースをコンパイルする際、コンパイラはデータベース全体を調べて、各ステートメントを分析します。コンパイラはどのような異常でも検出し、エラーメッセージや警告を出力します。これらのメッセージは 4th Dimension で開くことのできるエラーファイルに書き込まれ、対話式デバッグを可能にします。

対話式デバッグについての詳細は、[第 3 章の「エラーファイルを 4D で対話式デバッグとして使用」](#)を参照してください。

コンパイル後のデータベースの使用

コンパイル後のデータベースの使用方法は、コンパイル前のものと同様です。めざましい速度向上に気付くことでしょう。

Windows の場合、コンパイル後のデータベースを 4D、4D Server、4D Runtime、または 4D Engine で開く際、“データベース名 .CMP ” という名前のファイルがコンパイル後のストラクチャファイルと同じ階層に作成されます。これはコンパイル後のデータベースの実行に必要なファイルで、データベースが一番最初に開かれた時に作成されます。

データベースの保護

コンパイル後のデータベースは、デザインモードへのアクセスが不可能であること以外は、オリジナルのデータベースと同一のものです。これには、下記のような利点があります。

- データベースのストラクチャが、故意あるいは事故によっても変更不可能。
- メソッドがプロテクトされているので、参照または解析することができない。

2

コンパイラの使用

この章では 4D Compiler の使用方法について説明します。

4D Compiler のメニュー

- オプションウィンドウで使用できる機能とオプション
- コンパイルの手順
- 4th Dimension とコンパイラの同時使用

4D Compiler のオプションは、プロジェクトという概念に基づいて構成されています。プロジェクトは選択したオプションのグループであり、ディスクに保存することができます。つまり、コンパイラ用のスタイルシートのようなものです。

データベースをコンパイルする際に、必ずしもプロジェクトを作成する必要はありませんが、プロジェクトを利用するとコンパイル、デバッグ、再コンパイルといった一連の処理を素早く簡単に行うことができます。

プロジェクトを使用すると下記のことが可能です。

- コンパイル後のストラクチャファイルの名前を保存
- エラーファイル、シンボルテーブル、タイプファイルの作成
- 実行形式アプリケーション作成、範囲チェック、警告、スクリプトマネージャとの互換性、ターゲットマシンのマイクロプロセッサなど、さまざまなオプションの指定
- アクティブオブジェクト、数値、文字列のデフォルトタイプの指定
- デバッグに役立つローカル変数初期化オプションの指定

プロジェクトを使用してデータベースを再コンパイルするたびに、4D Compiler は指定されたファイルを作成し、要求されたオプションを使用します。プロジェクトを使用して指定するオプションについての詳細は、「[コンパイルオプション](#)」のセクションを参照してください。

データベースの使用

データベースは、4th Dimension とコンパイラの両方で同時に開くことができます。コンパイルを行うたびに、4th Dimension を終了させてからコンパイラでデータベースを開く、という必要はありません。

4th Dimension で開かれているデータベースをコンパイルするには、デザインモードからユーザモードまたはカスタムモードにした後、コンパイラに切り替えます。エラーファイルを 4th Dimension で使用している場合には、デザインモードのメ

ニューで「エラーファイル参照中止」を選択してエラーファイルを閉じてください。

コンパイル中はデータベースを使用できません。コンパイルが完了すると、4th Dimension に切り替えてデータベースに戻ることができます。

コンパイラはバックグラウンドで動作させることができます。つまり、コンパイラが作業している間に他のアプリケーションを使用することが可能です。同時に開くことのできるアプリケーションの数は、使用しているコンピューターのRAMの搭載容量によって制限されます。

注 4D Compiler のバージョン 6.5 は、4th Dimension および 4D Server バージョン 6.5 との併用を前提としています。4th Dimension バージョン 3 で作成されたデータベースに関連するプロジェクトファイルを開くことは可能ですが、コンパイルはできません。まず 4th Dimension または 4D Server バージョン 6.5 でデータベースを開き、バージョン 6.5 用のデータベースに変換する必要があります。

4D Compiler のメニュー

Macintosh では、4D Compiler のメニューバーには下記の 3 つのメニューがあります。

- アップルメニュー
- ファイルメニュー
- 編集メニュー

アップルメニューには、「4D Compiler について ...」というコマンドがあります。このメニューコマンドはコンパイラに関する情報を表示します。

編集メニューは、全ての Macintosh のアプリケーションに必要なメニューです。4D Compiler では使用しません。

Windows では、4D Compiler のメニューバーには下記の 3 つのメニューがあります。

- ファイルメニュー
- 編集メニュー
- ヘルプメニュー

編集メニューは、すべての Windows のアプリケーションに必要なメニューです。4D Compiler では使用しません。

ヘルプメニューは、4D Compiler 用の Windows ヘルプファイルへのアクセスを提供します。

コンパイラに関するメニューコマンドを含んでいるメニューは、ファイルメニューだけです。下記の 9 つのコマンドが含まれます。

- 新規 ...
- 開く ...
- 再コンパイル
- 閉じる

- 保存
- 名前をつけて保存 ...
- 元に戻す
- デフォルトプロジェクト ...
- 終了

新規

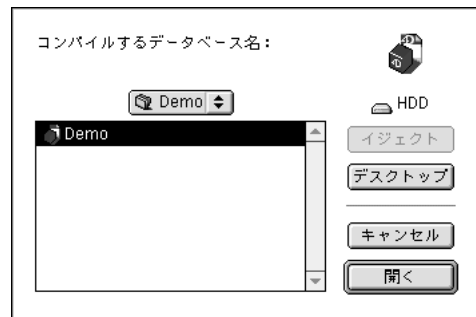
新しいプロジェクトを作成する際にこのメニューコマンドを使用します。プロジェクトにより、コンパイルするデータベースや選択するオプションを指定します。

このメニューコマンドを選択すると、コンパイラはプラットフォームに応じて標準の「開く」ダイアログボックスを表示します。このダイアログボックスを使用してコンパイルするデータベースを開きます。

Windows



Macintosh



データベースを開くと、4D Compiler はメインのオプションウィンドウを表示します。



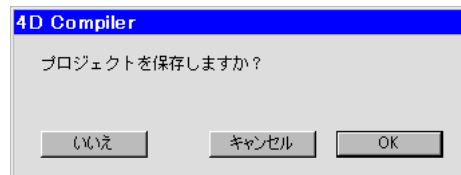
オプションウィンドウ内のそれぞれのアイコンは、コンパイルオプションを表しています。これらのオプションについては、この章の「[コンパイルオプション](#)」のセクションに記されています。

開く

既存のプロジェクトを開くには、「開く」メニューコマンドを使用します。プロジェクトを使用するとコンパイルの環境が保存されるので、同じデータベースを何度もコンパイルする際に速やかに実行することができます。

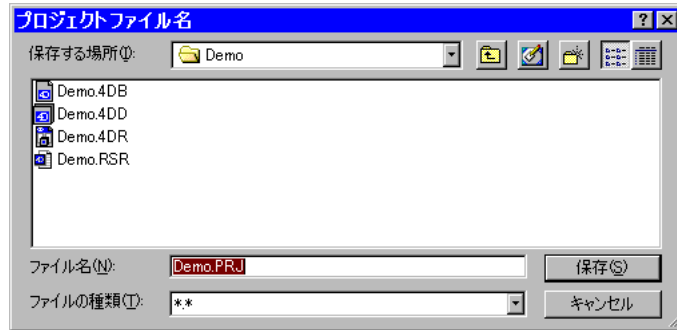
プロジェクトの概念

コンパイルのオプションを選択した後、4D Compiler は次のようなダイアログボックスを表示します。ここでプロジェクトを保存することができます。



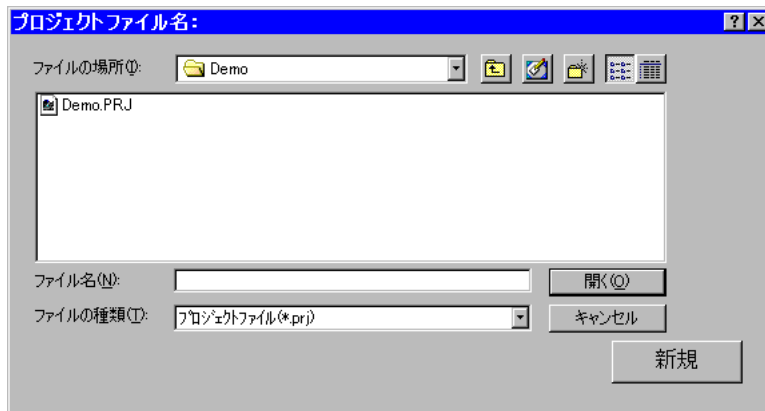
オプションの設定を保存しておく、同じデータベースを同じコンパイルオプションで素早く再コンパイルすることができるため非常に便利です。

「OK」ボタンをクリックすると、プラットフォームに応じて標準の「保存」ダイアログボックスが表示されます。プロジェクトファイルに名前を付け、「保存」ボタンをクリックします。



プロジェクトを開く

「開く」メニューコマンドを選択すると、標準の「開く」ダイアログボックスが表示されます。



既存のプロジェクトが存在しない場合にファイルメニューから「開く」を選択すると、「新規」メニューを選択した場合と同様に、「新規」ボタンによりコンパイルするデータベースを開くことができます。

プロジェクトを開いた場合には、4D Compiler オプションウィンドウが表示され、そのプロジェクトでの設定が表示されます。



プロジェクトを使用してデータベースを再コンパイルする

既存のプロジェクトを開くと、4D Compilerはそのプロジェクトで定義されているすべてのオプションを使用します。オプションの中には、コンパイル結果の情報を含んだファイルを作成するものもあります（シンボルテーブル、エラーファイルなど）。プロジェクトを再コンパイルすると、すべての関連するファイルが自動的に再作成されます。前回のコンパイルで作成されたファイルを保存しておきたい場合には、再コンパイルを行う前に必ずこれらを別のフォルダやディレクトリに移動してください。

これらのファイルについての詳細は、この章の「[コンパイルオプション](#)」のセクションを参照してください。

再コンパイル

デバッグ中のデータベースを再コンパイルするには、「再コンパイル」メニューコマンドを使用します。「再コンパイル」メニューコマンドは、4th Dimension を同時に使用して行う対話式デバッグのためのものです。4th Dimension でエラーを修正したら、4D Compilerへ切り替え、「ファイル」メニューから「再コンパイル」を選択してデータベースを再コンパイルします。対話式デバッグについての詳細は、第3章の「[エラーファイルを4Dで対話式デバッグとして使用](#)」を参照してください。

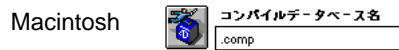
注 既存のプロジェクトを開く、あるいは新しいプロジェクトを作成する、いずれの場合でも、コンパイルするデータベースがパスワードで保護されている場合には、データベースを選択する際にデザイナのパスワードを入力するよう要求されます。

閉じる

現在のプロジェクトを閉じるには、「閉じる」メニューコマンドを使用します。現在のプロジェクトを保存せずに「閉じる」を選択すると「保存」ダイアログボックスが表示され、保存することができます。

保存	前回保存されたものの上書きして現在のプロジェクトを保存するには、「保存」メニューコマンドを使用します。
名前をつけて保存	新しいプロジェクトを保存するか、あるいは現在のプロジェクトを別の名前で保存するには、「名前をつけて保存 ...」メニューコマンドを使用します。このメニューコマンドにより、「保存」ダイアログボックスを表示し、現在のプロジェクトに新しい名前をつけて保存することができます。
元に戻す	現在のプロジェクトを前回保存したものに戻す場合には、「元に戻す」メニューコマンドを使用します。
デフォルトプロジェクト	<p>このメニューコマンドにより、新規プロジェクトにデフォルトで適用するオプションを定義することができます。</p> <p>デフォルトオプションの指定</p> <ol style="list-style-type: none">1 ファイルメニューからデフォルトプロジェクトを選択する <p>メインのオプションウインドウが表示され、デフォルトプロジェクトが定義可能になります。ここで選択したオプションは、新規プロジェクトを作成するたびに適用されます。</p> <p>デフォルトプロジェクトの設定は、4D Compiler の初期設定ファイルに保存されます。このファイルは、Macintosh 上ではシステムフォルダ：初期設定フォルダ：ACI フォルダ内に保存されます。Windows 上では Windows フォルダ¥ACI フォルダ¥4DComp.PRF ファイルに保存されます。</p> <p>デフォルトプロジェクト用のオプションは、データベース用の特定のプロジェクト用のオプションと同様に選択されます。ただ一つの違いは、ファイル作成ダイアログ・ボックスが、デフォルトプロジェクトを使用するデータベース用に使用される接尾語（サフィックス）を入力するエリアに置き換わっていることです。</p>

- 2 「コンパイルデータベース名」ボタンをクリックします
入力エリアが表示されます。



- 3 拡張子を入力します

デフォルトの拡張子（Windows では .4DC、Macintosh では .comp）が表示されます。次にコンパイルされるデータベースには、ここで指定した拡張子がデータベース名の後に付加されます。

- 4 「保存」ボタンをクリックして、デフォルトプロジェクトを保存する

デフォルトプロジェクトは必要に応じて変更できます。たとえば、あるデータベースだけをスクリプトマネージャ対応にする場合、デフォルトプロジェクトでこのオプションを設定しておく必要はありません。デフォルトプロジェクトを利用して新規にプロジェクトを作成した後で、スクリプトマネージャ対応に設定することができます。この変更は、作業しているデータベースのプロジェクトに保存されますが、デフォルトプロジェクトそのものは変更されません。

- 5 その他の拡張子を設定する

エラーファイルの拡張子は変更しないことをお勧めします。変更した場合、4D 対話式デバッグが使用できなくなります。エラーファイルについての詳細は、[第3章の「エラーファイルの使用」](#)のセクションを参照してください。

デフォルトプロジェクトの変更

- 「ファイルメニュー」から「デフォルトプロジェクト」を選択する

終了

4D Compiler を終了するには、「終了」メニューを選択します。

コンパイルオプション

最初のオプションウィンドウは、メインのコンパイルオプションを表示します。「次...」ボタンをクリックすると次のオプションウィンドウが表示され、コンパイルの微調整を行なうことができます。

オプションウィンドウ



メインのオプションウィンドウには9個のアイコンがあり、それぞれが選択または解除されるオプションを表わしています。コンパイルするデータベースに応じて、これらのオプションのいくつか、あるいはすべてを使用することができます。アイコンをクリックして、動作の一つを選択します。2回目にクリックすると、最初の設定をキャンセルします。設定に応じてアイコンの表示が変化します。

各オプションについて説明します。

コンパイルデータベース名



コンパイル後のストラクチャファイルに名前を付けるにはこのオプションを使用します。このアイコンをクリックすると、ファイル作成ダイアログボックスが表示されます。コンパイル後のデータベースのデフォルト名は、データベース名の後に “.comp” または “.4DC”、あるいはデフォルトプロジェクトで設定した拡張子が続きます。

この名前は変更可能です。「保存」または「OK」ボタンをクリックして、コンパイル後のストラクチャファイルの名前を保存します。コンパイル後のデータベースに名前を付けると、メインのオプションウィンドウ内のアイコンの右にその名前が表示されます。

コンパイルされた
データベース名



元のデータベースとコンパイル後のデータベースが同じディレクトリ内にある場合は、それぞれ違う名前を付ける必要があります。コンパイル後のデータベースに元のデータベースの名前を付けようとする、コンパイラは標準のファイル置き換えダイアログボックスを表示します。ここでファイルを置き換えることはできません。なぜなら、元のファイルはデバッグする際に必要になるからです。たとえあなたが「はい」ボタンをクリックしても、4D Compiler はエラーメッセージを表示して、コンパイル前のデータベースへの上書きを行なうことを防ぎます。

コンパイル後のデータベースは、元のデータベースの複製です。これは、デザインモードにアクセスできないこと以外は、元のデータベースと同様に動作します。デザインモードにアクセスする必要がある場合のために、4D Compiler では元のファイルへの上書きを行いません。

実行形式アプリケーションの作成

なし



選択



4D Compiler では、コンパイル後のデータベースを 4D Engine とマージさせるスタンドアロン実行形式アプリケーションの作成が可能なオプションがあります。このオプションを使用する前に、ハードディスクに 4D Engine をインストールする必要があります。

Windows プラットフォーム コンパイルを開始する前に、下記のファイルがハードディスクの同じディレクトリ内にコピーされていることを確認してください。

4DENGINE.4DE	4DENGINE.RSR
ASIFONT.FON	QTDP32.DLL
ASINTPPC.DLL	ASIPORT.RSR
ASIFONT.FON (オプション)	ASIFONT.MAP (オプション)

実行形式アプリケーションの作成

- 1 「実行形式アプリケーション作成」ボタンをクリックします。
4D Compiler は「開く」ダイアログボックスを表示します。



- 2 4D Engine を選択して、「開く」をクリックします。

メインのオプションウインドウに戻ると、選択した 4D Engine の名前が表示されます。

コンパイルを行なう前に、ストラクチャファイルと同じディレクトリ内に DISTRIB ディレクトリが作成されます。コンパイル実行中、コンパイラは 4D Engine とコンパイル後のストラクチャを使用し、実行形式アプリケーションを作成します。

作成されるファイルは、“データベース名 .EXE”、“データベース名 .4DC”、“データベース名 .RSR” の 3 つです。また、DISTRIB ディレクトリ内に “ASIFONT.FON”、“ASINTPPC.DLL”、“ASIPORT.RSR”、および “QTDP32.DLL” がコピーされます。

“データベース名 .EXE” は、実行形式のアプリケーションです。“データベース名 .4DC” と “データベース名 .RSR” は、コンパイル後のストラクチャです。これらのファイルを 4th Dimension、4D Server、4D First で開くことはできません。

- 注 デフォルトでは、実行形式アプリケーションには一般的なアイコンが割り当てられています。付録 B「アプリケーションのカスタマイズ」に従って、このアイコンをカスタマイズすることができます。

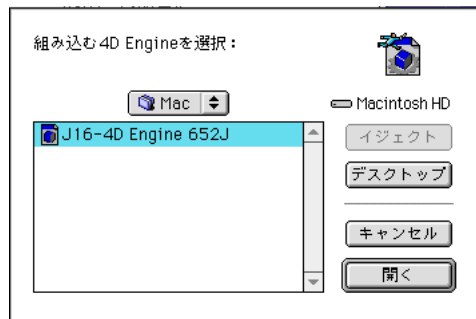
- 実行形式アプリケーションのインストールを完了するには
- DISTRIB ディレクトリ内に次のコンポーネントを置きます。
 - データファイルである “ データベース名 .4DD ”
 - WIN4DX および MAC4DX ディレクトリ (プラグインを使用している場合)
 - 必要であれば、“ CTL3D32.DLL ” および “ CTL3D.DLL ”
- データベースを配付するには
- Stirling Technologies 社の「InstallShield」等の市販のインストーラを使用することができます。このタイプのインストーラには下記のような利点があります。
 - ユーザが標準あるいはカスタマイズのインストールを選択できる
 - アイコンをカスタマイズできる
 - インストール時に複数枚のディスクを使用できる

Macintosh プラットフォーム

実行形式アプリケーションを作成するには。

- 1 「実行形式アプリケーション作成」ボタンをクリックします。

4D Compiler により「開く」ダイアログボックスが表示され、マージする 4D Engine を選択することができます。



- 2 4D Engine を選択して、「開く」をクリックします。

メインのオプションウィンドウに戻ると、選択した 4D Engine の名前が表示されます。コンパイルの実行中、コンパイラは 4D Engine とコンパイル後のストラクチャを使用し、ダブルクリック可能な単体の実行形式アプリケーションを作成します。Macintosh では、4D Engine とコンパイル後のストラクチャファイルが一つのファイルにマージされます。

注 実行形式アプリケーションのカスタマイズについての詳細は、[付録 B 「アプリケーションのカスタマイズ」](#)を参照してください。

エラーファイル

なし

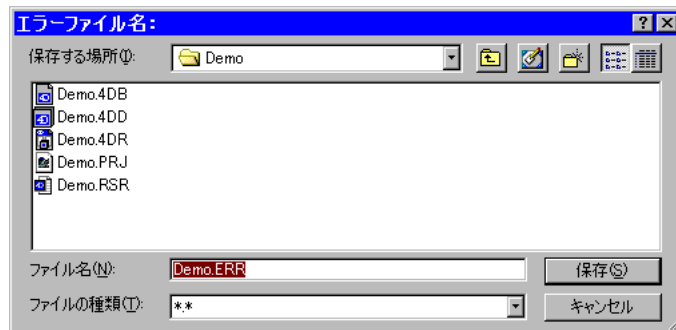


選択

このオプションを使用すると、タイプの割り当てとコンパイルの途中で検出されたエラーおよび警告のリストを含んだテキストファイルを出力します。このファイルは 4th Dimension で開いて対話式デバッグに使用することができます。

また、エラーファイルにはコンパイル時の警告とさらに詳細な警告も含まれています。詳しくは、この章の「警告」のセクションを参照してください。

このアイコンをクリックすると、4D Compiler は標準のファイル作成ダイアログボックスを表示します。デフォルトのファイル名は、データベース名の後に拡張子 “.err ” あるいはデフォルトプロジェクトウィンドウで設定した拡張子が続きます。



対話式デバッグを行う場合には、デフォルト名を使用しなければなりません。「保存」または「OK」ボタンをクリックすると、メインのオプションウィンドウにエラーファイルの名前が表示されます。

このファイルは、次の 2 通りに使用できます。

- テキストファイルとして、コンパイラで出力されたメッセージの記録を保存できます。
- 4th Dimension でデータベースをリアルタイムにデバッグできます。

エラーファイルについての詳細は、第 3 章と付録 A のエラーメッセージのリストを参照してください。

シンボルテーブル



データベースのシンボルテーブルをテキストファイルで出力するには、このオプションを使用します。このファイルには、コンパイルされるデータベース内のすべてのオブジェクトに関する情報が含まれています。これには、プロセス変数、インタープロセス変数、ローカル変数、これらのデータタイプ、およびデータタイプが決定されたメソッドの名前が含まれています。

また、シンボルテーブルには、メソッドと関数、これらのパラメータのデータタイプ、関数の戻り値のデータタイプが含まれています。

このアイコンをクリックすると、4D Compiler は標準の「保存」ダイアログボックスを表示します。デフォルトのファイル名はデータベース名の後に、“ .sym ” (Windows) または “ .symb ” (Macintosh)、あるいはデフォルトプロジェクトウィンドウで設定した拡張子が続きます。



この名前は変更可能です。「保存」または「OK」ボタンをクリックすると、シンボルテーブルの名前がメインのオプションウィンドウに表示されます。

シンボルテーブルについての詳細は、[第3章](#)を参照して下さい。

範囲チェック



範囲チェックは、非常に強力な診断機能を提供します。コンパイル後のデータベースの実行中にメソッドの状態をモニターします。

範囲チェックは、コンパイル中にメッセージを表示するものではありません。コンパイル後のデータベースの使用中にのみメッセージが表示されます。これは、データベースの実行中にのみ表面化するような問題に対して、非常に有用な機能です。範囲チェックについての詳細は、[第3章](#)を参照してください。

スクリプトマネージャ



Macintosh ではスクリプトマネージャ、Windows では2バイトのシステムに対応している4th Dimension でデータベースを実行する場合に、このオプションを使用します。

スクリプトマネージャとは、日本語、中国語、アラビア語、ヘブライ語などの、アルファベットやローマン以外の文字でデータベースの使用を可能にするものです。スクリプトマネージャ (Macintosh) や2バイトのシステム (Windows) 上で稼動する4th Dimension でデータベースを使用する場合には、このオプションを選択してください。

それ以外の場合には、スクリプトマネージャを使用しないでください。データベースの実行が遅くなることがあります。

注 日本語環境では、「対応する」を選択してください。

警告



エラーファイルに広範囲な診断メッセージを出力するには、このオプションを使用します。

次のような場合、警告メッセージが非常に有効です。

- すでにコードがコンパイル可能な状態であるが、コードの質をさらに高めたい場合
- エラーではなくても、コンパイラが完全に評価できないようなステートメントが含まれている場合、コンパイラはコードをもとに類推を行うが、その推定が正しいかどうか確認したい場合。

このオプションが選択されていないと、コンパイラは何の警告メッセージも表示しません。「標準」オプションが選択されると、コンパイラは警告メッセージをエラーファイルに出力します。「詳細」オプションが選択されると、コンパイラはより詳しい警告を出力します。このオプションについての詳細は、[第 3 章](#)を参照してください。

プロセッサの種類

4D Compiler では、コンパイル対象のマイクロプロセッサを選択できます。ここでの選択は、コンパイルを行うマシンではなく、コンパイル後のデータベースを実行する Macintosh や Windows マシンに搭載されているプロセッサの種類です。適切なオプションを選択すると、データベースを実行するマシンの能力を引き出すことができます。

- Motorola PowerPC (PowerMacintosh)
- 386/486 及び Pentium (Windows)

Motorola PowerPC (PowerMacintosh)

601/603/604



Off

このバージョンで生成される PPC のコードは、Power Macintosh に搭載されている Power PC 601、603、604 と互換性があります。

このオプションでコンパイルを行うと、データベースは Power Macintosh に対し完全に最適化されます。4D Compiler により生成されたネイティブの PPC コードは、PPC 対応 (PPC ネイティブまたは FAT バイナリ) の 4th Dimension、4D Client、4D Engine で実行可能です。4D 環境全体が Power Macintosh の速度で動作します。

PC (Windows)

386/486



Pentium
最適化



Off

386 または 486 プロセッサを搭載する PC 用にコンパイルを行なう場合には、386/486 オプションを選択してください。目的のマシンが Pentium を搭載していれば、「Pentium 最適化」オプションを選択してください。

注 「Pentium 最適化」オプションでコンパイルされたデータベースは、386/486 ベースのマシンでも稼働します。

コンパイル後のデータベースを 4D Server とさまざまな PC クライアントで使用するには、最も高性能なコンパイルオプションを選択します。つまり、Pentium と 486 の両方のクライアントが存在する場合、「Pentium 最適化」を選択してください。

追加オプション

2 番目のオプションウィンドウにアクセスするには、メインのオプションウィンドウの下にある「次 ...」ボタンをクリックしてください。2 番目のオプションウィンドウには、8 個のアイコンがあります。



各オプションの詳細を紹介します。

最適化



4D Compiler では、2種類のコード生成方法を選択できます。

- 「標準」は、コンパイルを速やかに行い、通常の（最適化されていない）コードを生成します。これは、コード内のエラーを発見する際に便利です。
- 「最適化」は、最適化されたコンパクトなコードを生成します。この方法は、コードの生成処理に要する時間が長くなりますが、コンパイル後のデータベースの実行速度は向上します。このオプションは、エラーを修正した後の最終バージョンのデータベースをコンパイルする際に使用することをおすすめします。

ローカル変数初期化



このオプションには3つの選択肢があります。デフォルトの設定は「0にする」で、これはすべてのローカル変数を初期化したことを確認していない段階に選択するオプションです。他の2つの設定は、データベースのパフォーマンスを最適化したい場合に使用するものです。

メソッドを実行する際、ローカル変数用のスペースがメモリに確保されます。これらのローカル変数は、メソッドの終了時に消滅します。メソッドの実行中、コンパイル後のコードの動作は、これら3つの設定のいずれかによって変わります。下記にローカル変数初期化オプションを設定した際の説明をします。

- 「0にする」：ローカル変数を作成し、値をすべて空（文字列には空のストリング、数値には0）にします。
- 「なし」：ローカル変数を作成し、初期化は行いません。コード内で変数を正しく初期化している場合、このオプションを選択するとデータベースの実行が最適化されます。これは、初期化に要する時間が節約できるためです。
- ランダム値にする：ローカル変数を作成し、ランダム値に初期化します。4D Compiler は、常に一定のランダム値で変数を初期化します。この意図的な「バグ」により、データベースの実行時に意図しない動作が誘発され、初期化し忘れたローカル変数を認識することができます。このオプションは、データベースの開発段階で使用します。

データベースの最終コンパイル時には、「0にする」または「なし」を使用してください。

- ▼ 例：
 - プロジェクトメソッド
 - ...
 - C_LONGINT(\$vIVar)

```

、 ...
、 (ここでは、$vIVar は常に明示的に初期化されるものとする)
、 ...
Case of
  ¥ ($vIVar=0)
  、 ステートメント A を実行 ...
  ¥ ($vIVar=1)
  、 ステートメント B を実行 ...
  ¥ ($vIVar=2)
  、 ステートメント C を実行 ...
else
  、 ステートメント D を実行 ...
End case

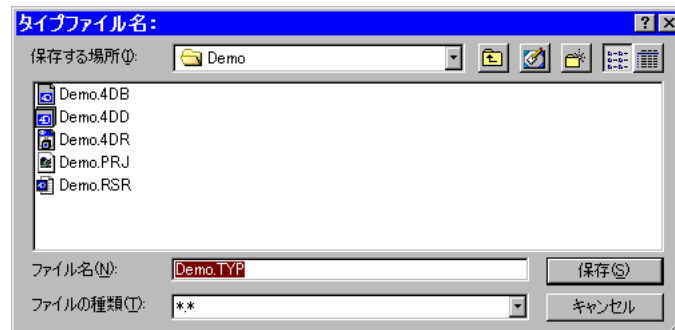
```

ローカル変数の \$vIVar を初期化しなかった場合、インタプリタではステートメント A が実行されます。ローカル変数初期化オプションを「なし」に設定してコンパイルされたデータベースでは、ステートメント D が実行されます。

タイプファイル



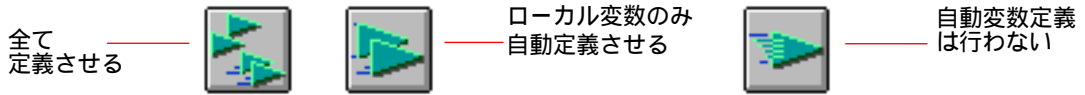
このオプションを選択すると、コンパイラはすべての変数を定義するコンパイラ命令を含むファイルを作成します。この定義をデータベースで使用する場合には、コンパイラ命令をひとつあるいは複数のメソッドにコピーします。アイコンをクリックすると、「名前をつけて保存」ダイアログボックスが表示されます。



デフォルトのファイル名は、データベース名の後に、“ .type ” (Macintosh) または “.typ ” (Windows)、あるいはデフォルトプロジェクトウインドウで設定した拡張子が続きます。

コンパイラ命令の使用に関する詳細は、[第 4 章](#)を参照して下さい。

コンパイルパス (未定義変数タイプ チェックの仕方)



- すべて定義させる：4D Compiler はコンパイルに必要なすべての段階を実行し、明示的に宣言されていないすべての変数のタイプを推測します。
- ローカル変数のみ自動定義させる：このオプションは、すべてのプロセス変数とインタープロセス変数のタイプが宣言されている場合に適切なものです。この宣言は、開発者が自分で行なうか、あるいはタイプファイルの内容をメソッドにペーストして行ないます。このオプションが選択されていると、コンパイラはタイプの設定をスキップすることができるため、コンパイル時間が速くなります。コンパイルは、データベースの複製、コンパイル、そしてファイル生成の順に行なわれます。
- 自動変数定義は行わない：すべてのプロセス変数、インタープロセス変数、およびローカル変数のタイプが宣言されている場合には、このオプションを使用します。この設定によりコンパイル時間は短縮されますが、作成されるコードには影響を及ぼしません。

注 コンパイルパスを変更すると、コンパイル時間を節約することができます。しかし、同時にコンパイラが出力する警告の数も減ることになるため、注意が必要です。

デフォルトボタン変数タイプ



コンパイラは、タイプが設定されていない変数を、最もカバーする範囲の広いタイプに設定しようとします。このオプションは、アクティブオブジェクトのタイプを強制的に「実数」または「倍長整数」にします。ただし、データベース内のコンパイラ命令に優先するものではありません。このオプションは次のアクティブオブジェクトに影響します。

- チェックボックス、3D チェックボックス
- ボタン、ハイライトボタン、透明ボタン、3D ボタン、ピクチャボタン、ボタングリッド、ラジオボタン、ラジオピクチャ、3D ラジオボタン
- ピクチャメニュー、階層ポップアップメニュー、階層リスト

デフォルトボタン変数タイプを「倍長整数」にすると、データベースの実行を最適化することが可能です。

デフォルト数値型変数タイプ



コンパイラは、タイプが設定されていない数値変数を、最もカバーする範囲の広いタイプに設定しようとします。このオプションは、数値のタイプを強制的に「実数」または「倍長整数」にします。ただし、データベース内のコンパイラ命令に優先するものではありません。「倍長整数」を選択すると、データベースの実行を最適化することが可能です。

デフォルト文字変数 タイプ



コンパイラは、タイプが設定されていない文字変数を、最もカバーする範囲の広いタイプに設定しようとします。このオプションは、文字列のタイプを強制的に「テキスト」または「固定長文字列」にします。ただし、データベース内のコンパイラ命令に優先するものではありません。

デフォルトの文字列タイプを「固定長文字列」に設定すると、入力エリアが現れ、文字列のデフォルトの長さを設定することができます。このオプションを選択すると、データベースの実行を最適化することが可能です。

コンパイラは、3つの基準によりタイプ設定を行いません。優先度の高い順に並べると次のようになります。

- コンパイラ命令（すべてに優先）
- デフォルトタイプオプション
- コンパイラ命令で宣言されず、デフォルトタイプを持たない変数には、コンパイラは適切なタイプを割り当てようとします。

バージョン番号自動 生成



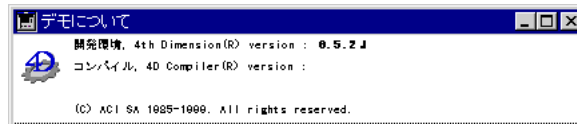
このオプションは、コンパイル後のデータベースにバージョン番号（1.0dx）を付加します。この番号はプロジェクト内に保存され、“x”はコンパイルするたびに増加していきます。

このオプションを選択すると入力エリアが現れて、バージョン番号の末尾に付く“x”の値の入力や変更が可能です。

Windows では、SET ABOUT コマンド使用時に表示される“4th Dimension について”ダイアログボックスにバージョン番号が表示されます。

Macintosh では、コンパイラにより “ vers ” リソースが作成されます。このリソースは、Finder の「情報を見る」メニューコマンドで使用されます。このリソースがすでにストラクチャに存在する場合は更新されます。

Windows



Macintosh



オプションについて

いずれのコンパイルオプションも、必ず選択する必要があるわけではありません。何もオプションを選択しなくても、コンパイルを開始することは可能です。この場合、「OK」ボタンをクリックすると、「保存」ダイアログボックスが表示され、コンパイル後のデータベース名を入力するよう要求されます。

コンパイル後のデータベース名を指定せずにプロジェクトを作成することも可能です。この場合は、実際にコンパイルを開始する際に、コンパイル後のデータベース名を入力するよう要求されます。

Windows のヘルプ

Windows では、広範囲なオンラインヘルプが用意されており、4D Compiler のヘルプメニューからアクセスすることができます。

コンパイルの開始

- コンパイルのオプションの選択後、コンパイルを開始するにはメインのオプションウインドウの「OK」ボタンをクリックします。
これ以降は、コンパイラにより自動的に作業が進められます。
- 注 データベースがパスワードによって保護されている場合、「OK」ボタンをクリックするとパスワードの入力が要求されます。コンパイルするには、“デザイナー”または“管理者”を選択し、正しいパスワードを入力して「OK」ボタンをクリックすると、コンパイルが開始されます。

プラグインが見つからない場合、4D Compiler はコンパイルを中断し、場所の入力を要求します。この点に関してはこの章の「コンパイル」のセクションを参照してください。

オプションウインドウの「Cancel」ボタンをクリックすると、4D Compiler は現在のプロジェクトを閉じます。

コンパイル

コンパイルする前に、データベースがインタプリタで正常に動作することを確認してください。本マニュアルの第4章のガイドラインに従ってデータベースを準備します。

データベースのコンパイル中には、次のような「4D Compiler ウィンドウ」が表示されます。ウィンドウの上部には、コンパイル中のメソッド名が表示されています。

ウィンドウの下部には、エラーを含むメッセージが表示されます。中央部分にはエラーカウンタと警告カウンタがあり、コンパイルが進むに従って数字が増えています。

「一時停止」ボタンと「中止」ボタンは、一時的に、あるいは完全に中止するためのものです。詳細については「[コンパイル処理の中断](#)」のセクションを参照して下さい。

コンパイル処理の主な段階は次のとおりです。

- データベースのコピー
- 変数のタイプ設定
- コンパイル
- 4D Engine のコピー

この流れは、オプションウィンドウで選択されたオプションに基づいています。コンパイルの状況は、ウィンドウ上部の小さなアイコンで表わされます。

データベースのコピー

データベースのストラクチャファイル（拡張子なしのファイル）のみが複製され、コンパイルされます。このファイルにはデータベース内のすべてのメソッドが含まれています。データファイルは、コンパイル後のストラクチャファイル、あるいはコンパイルされていないストラクチャファイルのどちらからでも開くことができます。

変数のタイプ設定

この段階では、コンパイル後のデータベース用のシンボルテーブルを作成します。次の 3 つのパスがあります。

- コンパイラ命令によるタイプ設定パス：コンパイラ命令や配列定義ステートメントを見つけ、変数や配列にデータタイプを割り当てます。
- プロセス変数とインタープロセス変数のタイプ設定パス：コンパイラ命令でタイプ設定されなかったプロセス変数とインタープロセス変数にタイプを割り当てます。
- ローカル変数のタイプ設定パス：ローカル変数にタイプを割り当てます。

それぞれのパスはコンパイル中に表示されるアイコンで表わされます。メソッドのサイズが小さい場合やお使いのマシンが高速な場合、速すぎて見えないこともあります。

コンパイラ命令によるタイプ設定パス

このパスは次のアイコンが表示されます。1

このパスでは、コンパイラ命令で宣言された変数のデータタイプを探して保存します。同様にデータベース内の配列を探して、そのデータタイプを保存します。

プロセス及びインタープロセス変数のタイプ設定パス

このパスは次のアイコンを表示します。2

このパスでは、コンパイラ命令で宣言されなかったプロセス変数とインタープロセスの変数のデータタイプを探して保存します。コンパイラは、変数の使用目的に応じてタイプを設定します。

ローカル変数のタイプ設定パス

このパスは次のアイコンが表示されます。3

データベース内のローカル変数のデータタイプを探して保存します。ローカル変数にリンクされたコンパイラ命令が存在する場合には、それを利用してタイプを設定します。コンパイラ命令が存在しない場合には、変数の使用目的に応じてタイプを設定します。

一般的な考慮

変数の設定において、第 5 章「タイプ設定ガイド」に記されたタイプ設定の矛盾が見つかることがあります。

矛盾が起きている場合、4D Compiler はそれぞれのパスで最初に見つかった段階でのタイプ設定を使用します。データベースメソッドとプロジェクトメソッドは、4th Dimension で作成された順に解析されていきます。

次に続くのはテーブルメソッド（トリガ）、フォームメソッド、フォーム上に作られたオブジェクトメソッドで、4th Dimension で作成された順番に解析されます。

タイプ設定処理のまとめ

タイプ設定が正常に完了するか、あるいはコンパイル不可能なほどのエラーが発生していなければ、次の「コンパイル」のセクションで説明されるコンパイルが開始されます。

タイプ設定でエラーが発生すると、コンパイル処理には進まず、さらに次のタイプ設定パスを実行します。


このパスは次のアイコンが表示されます。5

このエラーには次の 2 つの原因があります。

- 同じ名前を持つ2つのオブジェクトがある場合。例えば、メソッドと変数、変数とプラグインコマンド、2つのプロジェクトメソッドなどです。同じ名前を持つ2つの変数のタイプが異なる場合、このデータタイプの矛盾は記録されますが、コンパイルは中止されません。
- タイプを判別できないプロセス変数やインタープロセス変数が見つかった場合。
コンパイラが出力するメッセージをもとにコードを修正するかどうかは自由です。

コンパイル

メソッドをマシン語に翻訳して保存するパスです。引き続きエラーがサーチされますが、ここではタイプ設定に関するものではなく、文法やその他の矛盾に関するエラーが対象です。

このパスは、次のアイコンが表示されます。 

エラーが見つからなければ、コンパイルされたデータベースが作成されます。エラーが見つかるとデータベースは作成されないの、エラー箇所を修正する必要があります。

「実行形式アプリケーション作成」オプションが選択されている場合、コンパイルが成功すると4D Engine とのマージが行なわれます。

コンパイル処理の中断

タイプ設定やコンパイルの処理中は、「一時停止」あるいは「中止」ボタンをクリックすることにより中断することができます

これは、コンパイラがエラーや警告メッセージを出力した場合に有効です。メッセージに該当するメソッドを参照することができます。

メソッドの参照には次の2つの方法があります。

- マウスでリストをスクロールし、目的のメソッドをクリックする。
- エラーや警告が発生しているメソッドから次のメソッドに「Tab」キーで移動する。反対方向に移動するには「Shift+Tab」を使用する。

コンパイラはエラーや警告を2つの方法で表示し、メッセージの内容に応じて使い分けます。

- メソッド内にエラーを発見すると、コンパイラはそのメソッド名を太字で表示します。
- メソッド内のステートメントの警告を示す場合には、コンパイラはメソッド名をイタリック体で表示します。
- メソッド内にエラーと警告の両方が含まれている場合には、メソッド名は太字のイタリック体で表示されます。

クリックすることにより、メソッドを選択します。



エラー及び警告は、ウインドウ下部に表示されます。

下記の内容を 3 行で表わします。

- メソッド内の行番号
- エラーや警告の見つかった行
- エラーや警告の説明

4D Engine のコピー

「実行形式アプリケーション作成」オプションを選択すると、コンパイラは 4D Engine のコピーをマージします。

コンパイル後のデータベースの使用

コンパイル後のデータベースは、コンパイル前のデータベースと同様に開くことができます。ただし、デザインモードに入ることはできません。また、データベースを開くには少なくとも一つ以上のカスタムメニューを作成しておかなければなりません。



実行形式アプリケーションを作成した場合には、アプリケーションアイコンをダブルクリックします。デフォルトのアプリケーションアイコンはこのような形ですが、カスタマイズすることもできます。アイコンのカスタマイズについての詳細は付録 B を参照してください。

コンパイル後のデータベースは、コンパイル前のデータベースと同じように動作します。

ドラッグ&ドロップ

ストラクチャファイルやプロジェクトファイルを 4D Compiler のアイコンの上にドラッグ&ドロップすることによりコンパイルを開始することができます (Macintosh の場合、ドラッグ&ドロップには System 7 以上の OS が必要です)。これは、ファイルのアイコンをクリックして、マウスボタンを押したままファイルを 4D Compiler のアイコン上に重ねて、マウスを放す、という操作です。これにより 4D Compiler が起動し、コンパイルが可能になります。

- ストラクチャファイルを 4D Compiler アイコン上にドラッグ&ドロップすると、メインのオプションウィンドウが表示されます。また、Option キー (Macintosh) または Alt キー (Windows) を押しながらファイルをドラッグ&ドロップすると、プロジェクトを保存するかどうか確認のダイアログボックスが表示され、ただちにコンパイルが開始されます。この場合はデフォルトプロジェクトで設定されたパラメータが使用されます。
- プロジェクトファイルを 4D Compiler アイコン上にドラッグ&ドロップすると、メインのオプションウィンドウが表示されます。また、Option キー (Macintosh) または Alt キー (Windows) を押しながらファイルをドラッグ&ドロップすると、ただちにコンパイルが開始されます。この場合はドラッグ&ドロップしたプロジェクトで設定されたパラメータが使用されます。

4D Compiler を 4th Dimension と共に使用する

PowerPC、及び 80x86 用のコンパイ ル

データベースをコンパイルする際には、そのデータベースが PowerPC、80x86 マシンのいずれの 4th Dimension (または 4D Runtime) でも使用できるかどうかという点に留意してください。また、4D Server を使用する場合は、PowerPC、80x86 マシンのいずれのクライアントもデータベースに接続できるかどうかという点にも考慮します。つまり、PowerPC、80x86 のすべてのマシンで動作するようにデータベースをコンパイルすると、どのプラットフォームでデータベースを実行しても、ハードウェアやソフトウェアの機能を生かし、すべての環境で最適に動作することを確実にします。これは特に 4D Server にも当てはまり、メソッド内のプラットフォーム特定のコード部分だけが各クライアントマシンの “.rex” ファイルにダウンロードされます。

もし PowerPC、あるいは 80x86 マシン専用でデータベースをコンパイルしたとすれば、同じプロセッサファミリを搭載したマシンでしかデータベースを実行することができません。

特定のプロセッサファミリを搭載したマシンでのみデータベースを使用することが明らかな場合だけ、プロセッサを特定してコンパイルします。そうでなければ、使用する可能性のあるプロセッサをすべて選択してコンパイルします。

3

診断ツール

デバッグをより簡単にするための4つの診断結果がコンパイラにより出力されま
す。

- シンボルテーブル:

データベースの解析に有用です。データベース内で使用されている変数を速やかに
チェックすることができます。また、4D Compilerにより出力されたエラーメッ
セージの解析にも使用できます。

- エラーファイル:

テキストファイルとして、または4th Dimension内で使用し、データベースのデ
バッグを容易にします。

- 範囲チェック:

コンパイル後のデータベースにおいて、メソッドの実行をモニター、または管理
するための高度なツールです。

- タイプファイル:

データベース内で使用されているすべての変数をコンパイル命令でタイプ設定し
たファイルです。

シンボルテーブル、エラーファイル、タイプファイルは、テキスト形式のファイ
ルを開くことのできる任意のアプリケーションで開くことができます。

この章では、これらのファイルとその使用方法について説明します。

シンボルテーブル

シンボルテーブルはテキスト形式のファイルで、テキストエディタやワープロで開くことができます。データベース内のすべてのオブジェクト情報が含まれており、各カラムがタブで区切られています。このファイルは次の4つの部分に分けられています。

- インタープロセス変数のリスト
- プロセス変数のリスト
- メソッド内およびオブジェクトメソッド内のローカル変数のリスト
- トリガ（テーブルメソッド）のリスト、プロジェクトメソッドや関数、およびそのパラメータのリスト

プロセスとインタープロセス変数のリスト

これらのリストは4つのカラムに分割されます。

最初のカラムは、データベース内のプロセス変数、インタープロセス変数、および配列の名前です。変数は50音順（シフトJISコード順）に並んでいます。

2番目のカラムは、各オブジェクトのデータタイプです。オブジェクトのタイプは、コンパイラ命令により決定されるか、またはオブジェクトの使用法から推測されます。データタイプが決定できない場合には、空欄になります。

3番目のカラムは、オブジェクトが配列の場合、何次元であるかを表わします。

4番目のカラムは、コンパイラがオブジェクトのデータタイプを決定したコンテキストへの参照が表示されます。複数のコンテキストで変数を使用されている場合には、変数のデータタイプを決定するためにコンパイラが参照したコンテキストが表示されます。

変数のシンボル

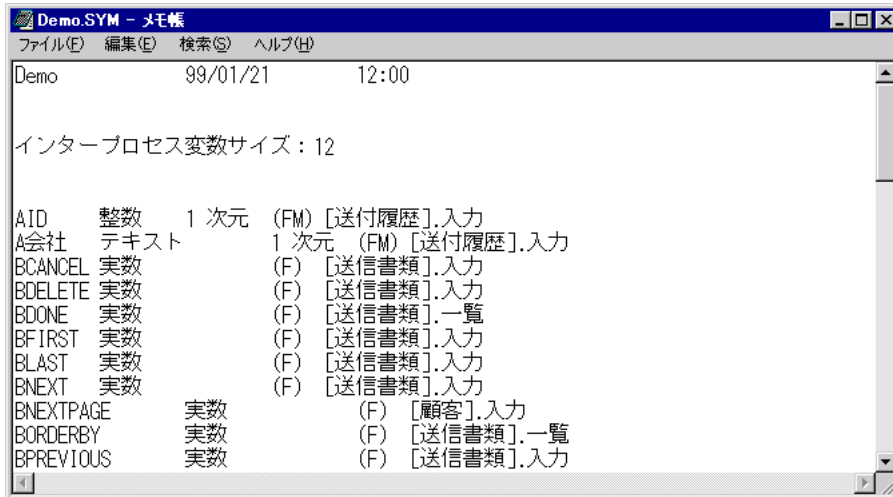
次のシンボルは変数の名前がどこで発見されたかを表示します。

- 変数がデータベースメソッド内で発見された場合、(M)*の後にデータベースメソッド名が表示されます。
- 変数がプロジェクトメソッド内で発見された場合、(M)の後にメソッド名が表示されます。
- 変数がトリガ（テーブルメソッド）内で発見された場合、(TM)の後にテーブル名が表示されます。
- 変数がフォームメソッド内で発見された場合、(FM)の後にフォーム名が表示されます。
- 変数がオブジェクトメソッド内で発見された場合、フォーム名、テーブル名、(OM)の後にオブジェクトメソッド名が表示されます。
- 変数がフォーム内のオブジェクトであり、メソッド、フォーム、オブジェクトメソッドのいずれでも使用されていない場合は、(F)の後にフォーム名が表示されます。

各リストの最後には、プロセス変数およびインタープロセス変数のサイズが表示されます。作成したプロセス変数の数には注意する必要があります。コンパイルの際、4D Compiler はどのプロセスでプロセス変数を使用されているかを確認することができません。プロセス変数は、各プロセス内で異なる値を持つことができます。つまり、すべてのプロセス変数は、新規プロセスが開始されるたびに複製されます。たとえば、50K バイトのプロセス変数がある場合、プロセスごとに50K バイトのメモリが要求されることとなります。

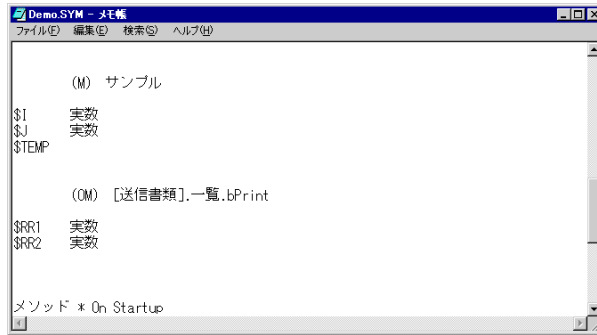
また、プロセス変数に必要な容量は、プロセスのスタックサイズには関連していません。

プロセス変数のリストの例を示します。



ローカル変数のリスト ローカル変数のリストは、データベースメソッド、プロジェクトメソッド、トリガ（テーブルメソッド）、フォームメソッド、オブジェクトメソッドの順でソートされています。ここではローカル変数を使用しているメソッドだけが表示されます。

ローカル変数のリストの例を示します。

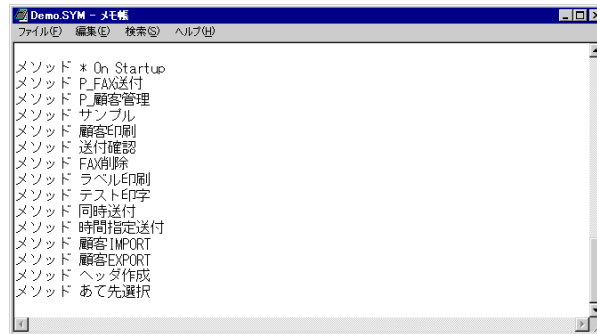


メソッドのリスト

ファイルの最後には、データベースメソッドとプロジェクトメソッドの全リストが含まれています。パラメータのデータタイプ、および関数の戻り値データタイプも合わせて表示されます。この情報は、次のようにフォーマット表示されます。

メソッド名 (パラメータのデータタイプ): 戻り値のデータタイプ

メソッドのリストの例を示します。



エラーファイル

エラーファイルには、コンパイル中に出力されたメッセージが含まれています。
次の2通りの使用方法があります。

- テキストファイルとして、テキストエディタやワープロで開く
- 4th Dimension で対話型デバッグに使用する

メッセージのタイプ 4D Compiler は次の3つのタイプのメッセージを出力します。

- 全般的なエラー
- 特定の行に関連するエラー
- 警告

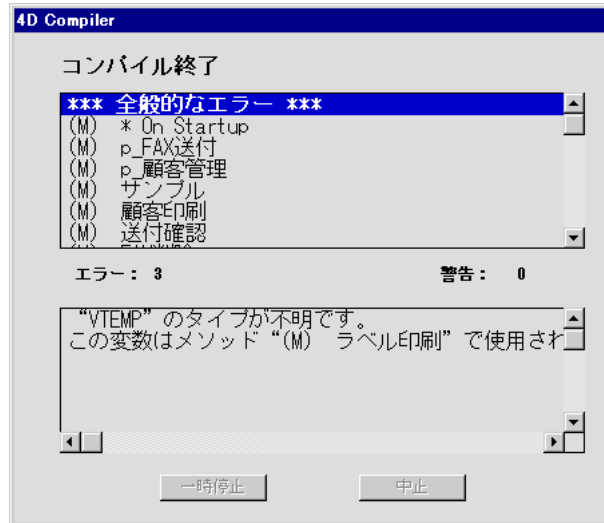
全般的なエラー

データベースのコンパイルを不可能にしてしまうエラーです。「全般的なエラー」は、特定のメソッドやオブジェクトメソッドに関連するエラーではないため、このように呼ばれています。

このエラーは、エラーファイルの先頭、および4D Compiler ウィンドウ内のメソッドリストの上に表示されます。



「全般的なエラー」をクリックすると、ウインドウ下部に関連するメッセージが表示されます。



- コンパイラから全般的なエラーが出力されるのは、次の2つの場合があります。
- プロセス変数のデータタイプが決定出来ない場合。
 - 複数のオブジェクトが同じ名前を持つ場合。

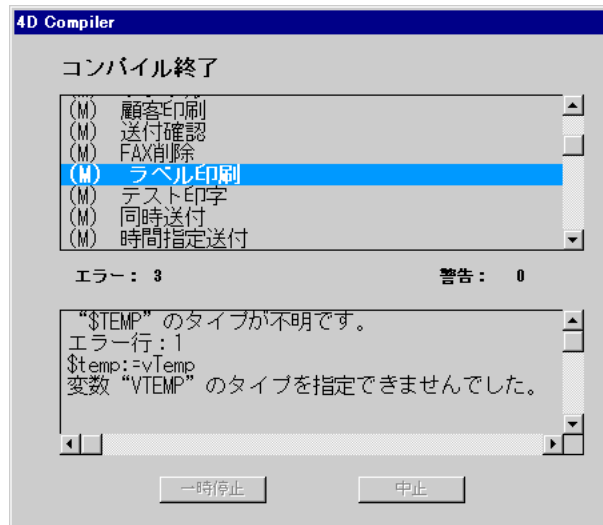
これは、どのオブジェクトに名前を関連付けさせるのが判断できないためです。
一般的なエラーのリストは、[付録](#)を参照して下さい。

特定の行に関連するエラー このエラーは、コンテキスト、つまりエラーが見つかった行が説明とともに表示されます。これはデータタイプや文法に関する矛盾が見つかった場合に出力されるエラーです。

このエラーを含んだメソッド名は、ウインドウ上部に太字で表示されます。メソッドの1つを選択すると、ウインドウ下部に次のような情報が表示されます。

- 行番号
- エラーが発生したステートメント
- エラーの詳細

エラーの例を次に示します。

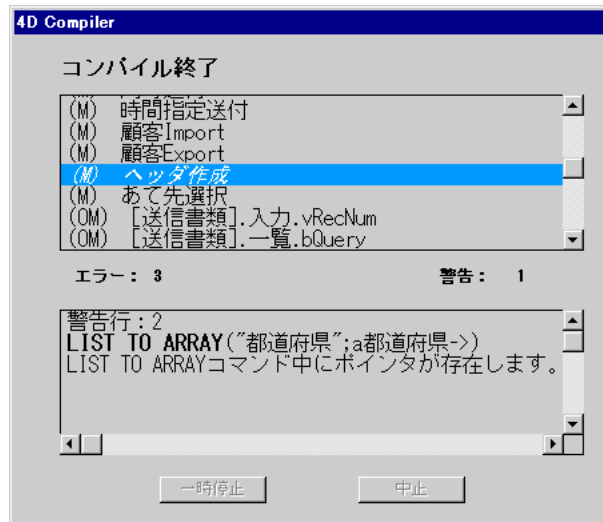


このエラーリストについては、[付録](#)を参照して下さい。

警告

警告はエラーではありません。警告はデータベースのコンパイルを妨げるものではなく、エラーになる可能性のあるコードを指摘するものです。

警告が検出されたメソッド名は、ウインドウ上部にイタリック体で現れます。メソッドの1つを選択すると、警告の詳細がウインドウ下部に表示されます。



警告は次の順序で表示されます。

- 行番号
- 警告に関連するステートメント
- 警告の内容

「警告」オプションで「詳細」を選択すると、他の警告も表示されます。

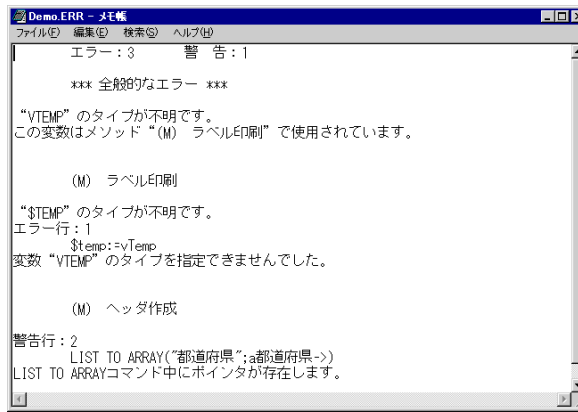
警告リストについては[付録](#)を参照して下さい。

エラーファイルの使用

エラーファイルは2通りの使用方法があります。テキストファイルとして扱う方法、もうひとつは4th Dimension 内で使用する方法です。

エラーファイルをテキストとして使用

テキストエディタを使用してエラーファイルを開いた場合の例を示します。



エラーファイルの構成は次のようになります。

- エラーと警告の数がファイルの先頭に表示されます。
- 一般的なエラーのリストが表示されます。
- 最後にその他のエラーと警告が4th Dimension で作成されたメソッドの順に表示されます。

エラーや警告は、次の形式のフォーマットで表示されます。

- メソッド中の行番号
- エラーや警告が発生した行
- エラーの説明

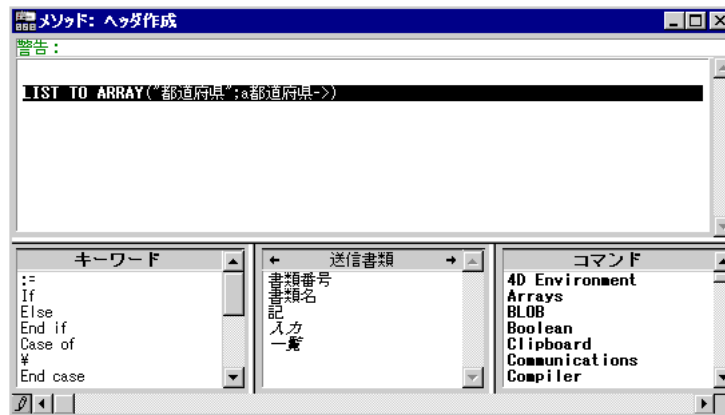
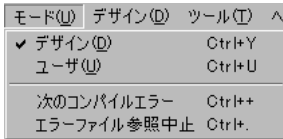
エラーファイルを4Dで対話式デバッグとして使用

エラーファイルを使用して、対話式でエラーを修正するには

- 1 コンパイルを実行してエラーファイルを作成する。

作成されたエラーファイルを4th Dimension で開くには、データベースと同じディレクトリ内にファイルが存在しなければなりません。また、ファイル名はデフォルトで付けられる“データベース名.err”にしておく必要があります。

- 注 Windows では、データベースと同じディレクトリ内に “データベース名.ERC” ファイルも必要です。
- 4th Dimension を起動し、コンパイル前のデータベースを開きデザインモードにする。
 「モード」メニューに、対話型デバッグ用の 2 つのメニューコマンド、「次のコンパイルエラー」および「エラーファイル参照中止」が追加されます。
 - 「モード」メニューから「次のコンパイルエラー」を選択する。
 4D Compiler がエラーや警告を検出した最初のメソッドが自動的に開きます。
 ウィンドウの上部にはエラーや警告の内容が表示され、関連する行がハイライトされます。
 - 指摘されたエラーを修正する。
 警告の場合は、コードを修正する必要がない場合もあります。
 - 「モード」メニューから「次のコンパイルエラー」を選択して、次のエラーや警告へ移動する。
 次のエラーあるいは警告が表示されます。



このように、エラー箇所を検索するのに時間を費やすことなく、エラーを速やかに修正していくことが可能です。ただし、メソッドに関連しない一般的なエラーは、ここでは表示されません。

デバッグを終了するには、「エラーファイル参照中止」メニューコマンドを選択します。デザインモードを抜けると、エラーファイルは自動的に閉じられます。

タイプファイル

このファイルは4つの部分に分かれています。

- インタープロセス変数用のコンパイラ命令
- プロセス変数用のコンパイラ命令
- ローカル変数用のコンパイラ命令
- パラメータ用のコンパイラ命令

タイプファイルは、コンパイル時間を節約するための重要な助けとなります。このファイルの内容を、次に示す変数やパラメータのメソッドにペーストするだけです。

- 名前が“COMPILER”で始まるタイプ設定メソッド内のプロセス変数、インタープロセス変数、パラメータ。
- ローカル変数が使用されているメソッド。詳細は第5章を参照して下さい。

範囲チェック

他のオプションはすべてコンパイル時に影響するものですが、この範囲チェックはコンパイル後のデータベースの実行時に開始されるものです。つまり、データベースの実行中のみ範囲チェックのメッセージが出力されます。範囲チェックは、いわば現場のコントローラのようなもので、データベース内のオブジェクトのステータスを評価するものです。

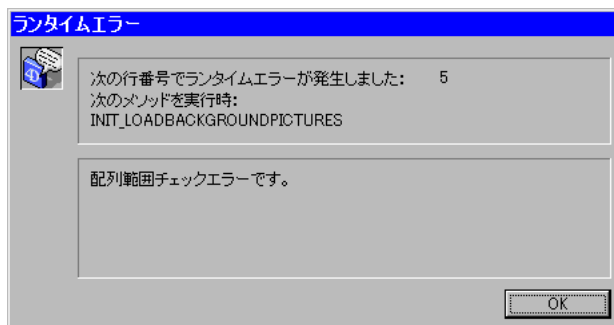
範囲チェックの動作例を示します。テキスト配列“MyArray”を定義し、“MyArray”の要素数はメソッドにより変化するものとします。5番目の要素に“Hello”という文字を代入する場合、次のように記述します。

```
MyArray{5}:="Hello"
```

このとき、“MyArray”の要素数が5以上であれば、問題はありません。代入は正常に行なわれます。しかし、“MyArray”の要素数が5未満であれば、代入は無効になります。

このような状態は、コンパイル時に検出することができません。コンパイラは、メソッドがどのような状況で実行されるかどうか事前に予測することができないからです。範囲チェックを使用することにより、データベースの実行中に何が起きているのかを監視することができます。

前の例では、4th Dimension で次のようなメッセージが表示されます。



範囲チェックは、配列、ポインタ、文字列の処理を行なう際に非常に有用です。

範囲チェックのメッセージのリストについては[付録](#)を参照して下さい。

範囲チェックの使用

範囲チェックを有効にすると、コンパイル後のデータベースの動作が遅くなる場合があります。したがって、最終的に完成したデータベースをコンパイルする際は、このオプションを指定しないでください。

範囲チェックは、データベースの開発およびデバッグの段階で付加的に使用するものであり、最終的なコンパイルは範囲チェックなしで行ないます。

コンパイルの最大の目的は、データベースの動作速度を向上させることであり、同時に信頼性を高めることでもあります。したがって、開発時には範囲チェックを使用することをお勧めします。

異常診断

範囲チェックは、データベースの不備な点を診断するのに特に有効です。例えば、データベースの実行中に異常が発見されたとします。この原因について推測する前に、コンパイラに用意されている手がかりを思い出してください。

考えられる異常としては次のようなものがあります。

- 4th Dimension からエラーメッセージが表示される場合。
可能であれば、4th Dimension の指示に従ってエラーを修正します。指示が一般的で修正の手がかりにならない場合、範囲チェックを指定して再コンパイルします。データベースを再度テストすると、先程エラーが表示された箇所、より詳しい情報が表示されます。
- コンパイル後のデータベースとコンパイル前のデータベースの動作が一致しない場合。
警告メッセージを参照してください。範囲チェックを指定して再コンパイルする必要があるかもしれません。
- インタプリタでは正常に動作するが、コンパイル後はクラッシュまたは異常終了する場合。
デバッガに精通していれば、どのメソッドでクラッシュしたのかを調べることができます。デバッガにあまり詳しくない場合の簡単な解決方法は、範囲チェックを指定して再コンパイルすることです。
- インタプリタでは動作するが、コンパイル後のデータベースでシステムがクラッシュする場合。
コンパイル後のデータベースで使用しているプラグインが、コンパイル前に使用していたものと同じかどうか調べて下さい。
- 数値変数や文字列変数が期待するような値を返さない場合。
コンパイル時のプロジェクトファイルの「ローカル変数初期化」オプションの設定を確認したり、シンボルテーブルを作成してすべての変数が正しくタイプ設定されているかどうかを確認してください。

4

コンパイル用のデータベースの準備

4th Dimension インタプリタでは、変数やメソッドの名前を自由に付けることができます。インタプリタは、コンパイラに比べてタイプの矛盾や不一致に関する許容範囲が寛大です。

コンパイルするデータベースを作成する際には、次の2つの原則を守ってください。

- 変数やプロジェクトメソッド、プラグインコマンドにはユニークな（重複しない）名前を付ける。
- 変数には、1つのデータタイプしか割り当てない。

この原則は、通常の正しいプログラミング方法です。コンパイル後の4th Dimension データベースで処理される操作に限ったことではありません。こうしたことがコンパイルするデータベースに要求されるのは、コンパイラにとって、データベース内のオブジェクトをすべて明確に認識することが必要不可欠だからです。

変数は、最もまぎらわしいオブジェクトです。そこで、ここでは変数について集中的に説明します。

- 異なる2つの変数に同じ名前を使う。
- 1つの変数に異なるデータタイプを割り当てる。

コンパイラは、コンパイル後のデータベースを生成する前にさまざまな角度からチェックを行い、曖昧な箇所を見つけると診断メッセージを出力します。この原則を意識せずに作成されたデータベースをデバッグする際に、このメッセージは有効です。

変数及び配列のデータタイプ

4th Dimension の変数には、次の 3 種類があります。

- ローカル変数
- プロセス変数
- インタープロセス変数

注 各変数タイプの性質については、『4th Dimension ランゲージリファレンス』を参照してください。

プロセス変数とインタープロセス変数は、4D Compiler にとっては構造的に同じものです。第 5 章「[タイプ設定ガイド](#)」では、両方の種類の変数に同じ説明が行われています。

プロセス変数の使用には、インタープロセス変数より多少の注意が必要です。4D Compiler では、変数がどのプロセスで使われるかを知ることができないので、新規プロセスの開始時には、プロセス変数がすべて複製されます。プロセス変数はプロセスごとに異なる値を持つことができますが、タイプはデータベース全体を通して同じです。

変数のタイプ

このセクションでは、4th Dimension の変数と配列の各種データタイプについて振り返ってみましょう。

変数には 12 のタイプがあります。

ブール	グラフ	ポインタ
BLOB	整数	実数
日付	倍長整数	テキスト
文字列	ピクチャ	時間

配列には、9 つのタイプがあります。

ブール	整数	ポインタ
日付	倍長整数	実数
文字列	ピクチャ	テキスト

シンボルテーブル

インタプリタでは、変数は2つ以上のデータタイプを持つことも可能です。これはコードをコンパイルするのではなく、解釈するからです。4th Dimension は各ステートメントを個別に解釈し、コンテキストを理解します。

しかし、コンパイラでは状況が異なります。インタプリタが1行ずつ処理するのに対して、コンパイルする際には、データベース全体を一度に処理しなければいけません。コンパイラの処理手順は、次の通りです。

- データベース内のオブジェクトを体系的に解析します。ここでいうオブジェクトとは、データベース、プロジェクト、フォーム、トリガー、オブジェクトメソッドです。
- オブジェクトを調べ、データベースで使われている変数のデータタイプを決定して、変数とプロシージャのテーブル（シンボルテーブル）を生成します。
- 変数すべてのデータタイプが確定すると、データベースの翻訳（コンパイル）を行います。ただし、各変数のデータタイプがすべて確定できなければ、データベースをコンパイルすることはできません。

同じ変数名で異なる2つのデータタイプが見つかったら、どちらを採用すればよいのかコンパイラには判断できません。オブジェクトのタイプを決めてメモリアドレスを割り当てるために、コンパイラはそのオブジェクトの厳密な定義、つまり、名前とデータタイプを必要とします。コンパイラはデータタイプからオブジェクトのサイズを決定します。

コンパイル後のデータベースごとにマップが作られ、各変数の名前（または識別子）位置（メモリアドレス）、変数が占める空間（データタイプで決まる）が記録されます。このマップを「シンボルテーブル」と呼びます。

4D Compiler による変数のタイプ設定

変数のタイプを直接的にかつ明確に指定する方法は、メソッド内でコンパイラ命令を使うことですが、必ずしも、変数すべてにコンパイラ命令を使わねばならないという訳ではありません。従来のコンパイラで必要だった変数ごとにタイプを定義するという、煩わしい作業は4D Compiler では不要です。

変数の使われ方を調べ、曖昧でない場合は可能な限りタイプを決定します。

- ▼ 例えば、

V1:=12.5

と記述した場合、変数 V1 のデータタイプ は実数になります。

- ▼ 同様に、

V2:=" これは、例題 "

と記述すると、V2 はテキストタイプの変数になります。

また、上の例ほど明確でない場合でも変数のタイプを決定することができます。

- ▼ 以下に例を示します。

V1:=12.5

```
V2:="これは、例題"  
V3:=V1
```

ここでは、V3 は V1 と同じタイプになります。

- ▼ 似たような例で、

```
V4:=2*V2
```

この場合も、V4 は V2 と同じタイプでになります。

同様に、4th Dimension コマンドや、メソッドに対するコールからも変数のデータタイプを決定します。たとえば、ブールタイプの引数と日付タイプの引数をプロシージャに渡すと、4D Compiler は、呼ばれたプロシージャのローカル変数 \$1 と \$2 にそれぞれ、ブールタイプと日付タイプを割り当てます。

4D Compiler で推測によりデータタイプを決定する場合、整数、倍長整数、文字のような制限のあるデータタイプは割り当てません。4D Compiler がデフォルトで割り当ててるのは、一番広い範囲をカバーできるタイプです。

- ▼ 例えば、次のように書くと、

```
Number:=4
```

4 は整数ですが、他の状況では値が 4.5 になる可能性があるので、コンパイラは “Number” に “実数タイプ” を割り当てます。

変数のタイプを整数や倍長整数、文字にする場合は、コンパイラ命令で定義します。これらのデータタイプはメモリ占有量が少なく、他のタイプに比べて処理速度が速いので、コンパイラ命令を使った方がよいでしょう。

すでに自分で変数のタイプを定義していて、タイプ付けが首尾一貫しており、完全であると確信できる場合は、この作業を繰り返さないように 4D Compiler に指示できます。定義が完全でなかった場合、4D Compiler からはコンパイル時にエラーが表示され、必要な変更を行なうよう求められます。

コンパイラ命令

コンパイラ命令に関する説明は、『4th Dimension ランゲージリファレンス』マニュアルを参照してください。

```
C_STRING ({method}; length; variable1 {...;variableN})
```

```
C_BOOLEAN ({method}; variable1 {...;variableN})
```

```
C_DATE ({method}; variable1 {...;variableN})
```

```
C_GRAPH ({method}; variable1 {...;variableN})
```

```
C_INTEGER ({method}; variable1 {...;variableN})
```

```
C_TIME ({method}; variable1 {...;variableN})
```

```
C_PICTURE ({method}; variable1 {...;variableN})
```

```
C_LONGINT ({method}; variable1 {...;variableN})
```

```
C_REAL ({method}; variable1 {...;variableN})
```

```
C_POINTER ({method}; variable1 {...;variableN})
```

```
C_TEXT ({method}; variable1 {...;variableN})
```

```
C_BLOB ({method};variable1 {...;variableN})
```

コンパイラ命令を使うと、変数を明示的に定義することができます。使用方法は次の通りです。

C_BOOLEAN (Var)

このように、変数 Var という変数を作り、そのタイプはブールであることをコンパイラに指示します。

アプリケーションでコンパイラ命令が使われていれば、4D Compiler は必ずそれを使用するのでタイプを推測するという作業をしなくて済みます。

コンパイラ命令は、代入や用途から得られた結果より優先されます。

コンパイラ命令 **C_INTEGER** で定義した変数は、実際には **C_LONGINT** で定義したのと同じです。これらは実際には、-2147483648 から +2147483647 までの倍長整数です。

コンパイラディレクティブが必要な時

コンパイラ命令が有効なのは、以下のような場合です。

- コンパイラで前後関係から変数のデータタイプを決定できないとき。
- コンパイラが決定するタイプを使用したくないとき。
- コンパイラ命令を使うとコンパイル時間が短縮できるとき。

曖昧になる場合

コンパイラで変数のタイプを決定できないこともあります。このような場合、4D Compiler からは必ずエラーメッセージが出力されます。4D Compiler でデータタイプを決定できない場合は、主に 3 つの原因があります。

- 複数データタイプ
- コンパイラが決定したタイプが曖昧な場合
- タイプを判断する決め手がない

複数データタイプ

データベース内で、変数が異なるタイプで再度定義された場合、コンパイラはエラーと判断します。このエラーは簡単に訂正できます。

コンパイラは最初に見つけたタイプを採用し、2 回目以降は 1 回目に割り当てたデータタイプを使用します。

- ▼ 簡単な例を示します。

```
Variable:=True           `メソッド A
Variable:="月は緑色"    `メソッド B
```

“メソッド A” の後に “メソッド B” がコンパイルされると、ステートメント “Variable:=" 月は緑色 ” は、“Variable” に対するデータタイプの変更とみなされます。コンパイラは、データタイプが再定義されていて、修正が必要であることをユーザに示します。ほとんどの場合、2 番目の変数の名前を変更すればエラーは修正できます。

コンパイラが決定したタイプが曖昧な場合

時々、4D Compiler が決定したオブジェクトのタイプを 4D Compiler 自身がそのオブジェクトにとって適切なタイプはないと判断する事があります。この場合、コンパイラ命令で、変数タイプを明確に指定しなければいけません。

▼ アクティブオブジェクト用のデフォルト値を使った例

フォーム内で、コンボボックス、ポップアップメニュー、タブコントロール、ドロップダウンリスト、メニュー/ドロップダウンリスト、及びスクロールエリアのデフォルト値を割り当てることができます。これはオブジェクト用のオブジェクトプロパティウィンドウのデータ制御ページにあるデフォルト値ダイアログで設定することができます。データエントリーを管理するオブジェクトについての詳細は、『*4th Dimension デザインリファレンス*』マニュアルを参照してください。

デフォルト値は、オブジェクトの名前である配列内に自動的にロードされます。

オブジェクトをメソッド内で使わないと、4D Compiler はテキスト配列として定義しますが、動作上、表示の初期化を行わなければならない場合には、次のようにコーディングするでしょう。

If (Before)

MyPopup:=2

End if

このとき、4D Compiler は MyPopup を実数として定義しようとしませんが、すでにテキスト配列と定義しており、この2つのタイプの不一致によりデータタイプを決定できなくなります。この場合、フォームメソッドあるいは **COMPILER** メソッド(「[開発者がタイプ定義する変数](#)」を参照して下さい)に明確に配列を定義する必要があります。

If (Before)

ARRAY TEXT(MyPopup;2)

MyPopup:=2

End if

タイプを判断する決め手がない

定義せずに変数が使われていて、前後関係からデータタイプを決定できないような状況です。こうなると、コンパイラにとっての決め手はコンパイラ命令しかありません。

こうした状況は、主として次の2種のコンテキストにおいて起こります。ポインタが使われている場合、または複数のシンタックスを持つコマンドに変数が使われている場合です。

■ ポインタ:

ポインタは、自分自身のデータタイプを返すことはできませんが、指している変数のデータタイプを返すことができません。次のような場合、

Var1:=5.2` (1)

Pointer:=>Var1` (2)

Var2:=Pointer->` (3)

ポインタ “Pointer” が指す変数のタイプは (2) で定義されていますが、“Var2” のタイプは定義されていません。4D Compiler はコンパイルの過程でポインタを認識できますが、それがどのタイプの変数を指しているのか知る手段がないのです。そのため、“Var2” のデータタイプを判定できません。この場合、次のようなコンパイラ命令が必要になります。

C_REAL (Var2)

■ 複合シンタックスコマンド :

複数のシンタックスを持つコマンドに関しては、どのシンタックスと引数が使われているのかコンパイラには、判定できません。

GET FIELD PROPERTIES コマンドは2つのシンタックスを受け入れます。

GET FIELD PROPERTIES (table number; field number; type; length ; index)

GET FIELD PROPERTIES (field pointer ; type ; length; index)

コマンドの引数がシンタックスに対応するタイプに設定されていない場合は、データベース内の任意の場所で、コンパイラ命令を使用してタイプを設定する必要があります。

■ オプション引数付のコマンド :

各種データタイプのいくつかのオプション引数付のコマンドを使用する場合には、4D Compiler はどのオプション引数が使用されたのかを決定することができません。

GET LIST ITEM コマンドには倍長整数とブールの2つのオプション引数があります。

GET LIST ITEM (*List;itemPos;itemRef;itemText;sublist;expanded*)

GET LIST ITEM (*List;itemPos;itemRef;itemText;expanded*)

データベースの中で変数が定義されておらず、またその使用方法においてもタイプが明確ではない場合には、コマンドに渡された変数のタイプを決定するのにコンパイラ命令を使用しなければいけません。

■ URL 経由で呼び出されたメソッド :

URL 経由で呼び出される 4D メソッドを書く場合、メソッド中で \$1 を使用しなくても、テキスト変数 \$1 を定義しなければいけません。

C_TEXT(\$1)

4D Compiler は、メソッドが URL 経由で呼び出されることを知ることはできません。

コードの最適化

コンパイラ命令を使用して、数値変数を整数や倍長整数に設定したり、文字変数を文字列タイプに設定すると、メソッドの処理速度が速くなります。

カウンタにローカル変数を使用した場合、変数のタイプを設定しないと、4D Compiler はその変数を実数とみなします。倍長整数に設定すると、コンパイルしたデータベースの効率が良くなります。なぜなら、実数データがメモリを 10 バイト使うのに対し、倍長整数にすると、4 バイトしか使わないからです。10 バイトのカウンタをインクリメントすると、4 バイトの場合より時間がかかります。また、実数の計算は整数の計算よりも処理速度が遅いからです。実数の計算は Apple の SANE 環境で行われますが、これは、整数で計算するより時間がかかります。

注 必要以上にコンパイラ命令を使っても間違いにはなりません。コンパイラ命令はコンパイル時に使われますが、コンパイルした結果には含まれません。

コンパイルの時間の短縮

データベースで使用する変数がすべて明示的に定義されていれば、4D Compiler でタイプ定義を調べる必要はありません。この場合、オプションを設定して翻訳フェーズだけを行なうように指定できます。この操作により、コンパイル時間を半分以下に短縮することができます。詳細に関しては、第 2 章「コンパイル」を参照してください。

実数と文字列を使う

整数と定義した変数に実数値を代入したり、10 文字の文字列として定義した変数に 30 文字の文字列を代入したりすると、4D Compiler はコンパイラ命令に応じた値を代入します。整数の変数に実数を代入すると、整数部だけが代入されます。10 文字の文字列変数に 30 文字の文字列を代入すると、最初の 10 文字だけが使われます。コンパイラはどちらのケースもタイプ矛盾とはみなしません。

- ▼ 例を示します。下記のように記述した場合、

```
C_INTEGER (vInteger)  
vInteger:=2.55
```

4D Compiler は、数値の整数部分を切り上げます (2.55 ではなくて 3 になります)

- ▼ 次は、文字列処理の例です。以下のように記述すると、

```
C_STRING (10;MyString)  
MyString:=" 本日は晴天なり "
```

4D Compiler は文字列定数の最初の 10 文字、“本日は晴天”だけを代入します。文字の場合、このようなケースは範囲チェックオプションで調べることができます。

インタプリタでコンパイラ命令を使う

コンパイルしないデータベースには、コンパイラ命令は不要です。各ステートメントで、変数がどのように使用されているかを判断して、インタプリタが自動的に変数のタイプを設定するからです。また、データベース内で変数のタイプを自由に変えることができます。

インタプリタがこのように柔軟に対応するので、インタプリタとコンパイル後では、データベースの動作が異なることがあります。

- ▼ 例えば、次のように記述すると、

C_LONGINT (MyInt)

また、データベースの他の場所で、下記のように記述すると、

MyInt:=3.1416

この例では、コンパイラ命令が代入ステートメントより前に解釈されれば、インタプリタでもコンパイル後でも“Myint”には同じ値(3)が代入されます。

4th Dimension インタプリタは、コンパイラ命令を使って変数のタイプを定義します。コンパイラ命令を検出すると、インタプリタはその命令に従って変数のタイプを定義します。それ以降のステートメントで間違った値を割り当てようとすると(たとえば、数値変数に文字を割り当てると)割り当ては行われずエラーが表示されます。

この2つのステートメントのどちらが先に現れようと、コンパイラにとって問題ではありません。はじめにデータベース全体を調べてコンパイラ命令を探すからです。しかし、インタプリタはシステムだてて処理するわけではなく、実行される順にステートメントを解釈します。もちろんユーザが何を行うかにより、この順序は毎回異なります。つまり、変数を定義する場合、その変数を使用する前にコンパイラ命令を実行するようデータベースを作成することが必要です。

コンパイラ命令をどこに記述するか

コンパイラ命令には、コンパイラに変数のタイプ付けをまかせるかどうかによって以下の2通りの考え方があります。

4D Compiler でタイプ定義される変数

- ローカル変数、プロセス変数、インタープロセス変数に応じて、変数が初めて使われるメソッドやオブジェクトメソッドでコンパイラ命令を使います。コンパイラ命令は、必ず変数が初めて使われる場所、つまり一番初めに実行されるはずのメソッドで使うようにしてください。

注 コンパイル時、4D Compiler は 4th Dimension での作成の順序でメソッドを処理します。エクスペローラで表示される順番ではありません。

- コンパイラ命令で定義するプロセス変数とインタープロセス変数は、**On Startup** データベースメソッドまたは **OnStartup** データベースメソッドから呼び出されるメソッドにまとめてください。
- それらが現れるメソッド内のローカル変数を宣言します。

注 『4th Dimension ランゲージリファレンス』には、コンパイラ命令があるメソッドは、必ずしも実行される必要はないと記述されています。そのメソッドが実行されなくても、コンパイラは(通常通り)コンパイラ命令によって変数のタイプを判断しますが、インタプリタでデータベースを使用する場合とコンパイル後のデータベースとでは結果が違ってくるかも知れません。

開発者がタイプ定義する変数

4D Compiler にタイプ定義をチェックさせたくなければ、コンパイラ命令を識別できるようなコードを 4D Compiler に与える必要があります。

このための規約は、プロセス変数、インタープロセス変数に関するコンパイラ命令と引数を、名前の先頭が "COMPILER" で始まる 1 個あるいは複数個のメソッドに入れる、というものです。たとえば、"COMPILER"、"COMPILER1"、"COMPILERtype" というメソッド名にします。

さまざまなコンパイラ命令をすべて同じプロシージャに入れることができますが、コードの理解性や保守の観点から、コンパイラへの定義情報が必要な変数のためのコンパイラ命令を 4 つに区分しておくことをおすすめします。

- インタープロセス変数
- プロセス変数
- ローカル変数
- メソッドへ渡す引数

コンパイラでは、メソッドへの呼びだしをコンパイルするために、メソッドが受け取る引数のタイプを必要とします。

引数定義のためのシンタックスは次のとおりです。

Directive (*MethodName*; parameter)

- ▼ 例えば、次の行はメソッド *MyMethod* が受け取る引数 "\$1" のタイプをブールと定義する場合は、次のように書きます。

C_BOOLEAN (*MyMethod*; \$1)

注 このシンタックスは、インタプリタでは実行されません。

各区分内のコンパイラ命令の一覧には、4D Compiler 自身の出力を使用することもできます。

コンパイル後のデータベースとインタプリタのデータベースとの互換性を保つために、4th Dimension でこれらのメソッドを実行することも可能です。

特定の引数

- データベースメソッドによって受け取られた引数

これらの引数が明確に定義されていないと、4D Compiler によってタイプが決定されます。定義する場合は、そのデータベースメソッド内で行なわなければいけません。

この引数定義は、**COMPILER** メソッド内には書き込みができません。

- ▼ 例:

On Web Connection が 2 つのテキスト・引数、\$1 及び \$2 を受け取った場合。データベースメソッドの最初では、C_TEXT(\$1;\$2) を記述しなければいけません。

- トリガ

トリガの結果である \$0 引数 (倍長整数) は、引数が明確に定義されていなければ、4D Compiler によってタイプ決定されます。定義する場合は、トリガー自身の中で行なわなければいけません。

この引数定義は、**COMPILER** メソッド内には書き込みができません

- "On Drag Over" フォームのイベントを受け入れるオブジェクト

"On Drag Over" フォーム・イベントの結果である \$0 引数（倍長整数）は、引数が明確に定義されていない場合は、4D Compiler によってタイプが決定されます。定義する場合は、オブジェクトメソッドの中で行なわなければいけません。

この引数定義は、**COMPILER** メソッド内には書き込みができません

注 4D Compiler は \$0 引数を初期化しません。従って、"**On Drag Over**" フォームイベントを使用したら直ちに \$0 を初期化しなければいけません。

- ▼ 例

```
C_LONGINT($0)
If (Form event=On Drag Over)
  $0:=0
  .....
  If ($DataType#Is Picture)
    $0:=-1
  End if
  .....
End if
```

C_STRINGコンパイラ命令

C_STRING のシンタックスは、他の命令とは異なります。文字の最大長を表す引数 "サイズ" があります。

```
C_STRING (length; variable1 {;...;variableN})
```

C_STRING は、固定長の文字列を扱うので、文字列の長さを指定する必要があります。コンパイルするデータベースでは、変数ではなく定数を使って文字列の長さを指定しなくてはなりません。以下に例を示します。

- ▼ 例

インタプリタでは、

```
TheLength:=15
C_STRING (TheLength;TheString)
```

4th Dimension は TheLength を解析し、**C_STRING** コンパイラ命令で、TheLength を値に置き換えます。

しかし、コンパイラでこのコマンドを使用して変数のタイプを設定する際には、特定の代入ステートメントを考慮に入れませんかから、TheLength の値が 15 であることはわかりません。文字列の長さがわからなくては、シンボルテーブルにその文字列用のスペースを確保できません。そこで、文字列の長さを定義する際はコンパイルを念頭において定数を使ってください。

- ▼ 例えば、ステートメントをこのように使用します。

```
C_STRING (15;TheString)
```

次のようなコマンドで定義する固定長文字配列についても同じです。

ARRAY STRING (*length;array name;size*)

配列の文字列の長さを示す引数は、定数にしてください。

注 文字フィールドの長さ（最大 80 文字まで）と、固定長文字変数を混同しないでください。 **C_STRING** コマンド、または **ARRAY STRING** コマンドで定義できる文字列のサイズは 1 から 255 までです。

このコマンドのシンタックスでは、1 行で同じ長さの変数を複数定義することができます。異なる長さの文字列を複数定義する場合は、別の行で行います

文字列の長さは、4D 定数あるいは 16 進法定数で指定することができます。

▼ 例：

C_STRING (4D_Constant;TheString)

ARRAY STRING(4D_Constant;TheArray;size)

C_STRING (0x000A;TheString)

ARRAY STRING(0x000A;TheArray;size)

まとめ

ここでは、コンパイラで変数のデータタイプを解析する際の作業全般について説明しました。さらに詳しく理解していただくためには、次の 2 つの章をお読み下さい。本章の内容を例をあげて解説し、さらに、以下の内容について説明します。

- 起こりやすいデータタイプの矛盾についてのガイドと、タイプ矛盾を避ける方法
- 4th Dimension コマンドをコンパイルする際の特別な注意事項

5

タイプ設定ガイド

この章では、データベースのコンパイルの障害になるタイプ矛盾について、さらに詳しく説明します。矛盾を引き起こすオブジェクトのタイプを1つずつ取り上げていきます。オブジェクトの種類は、以下の通りです。

- インタープロセスとプロセス変数
- ローカル変数
- 配列
- フォーム変数
- ポインタ
- プラグインコマンド
- 予約変数

インタープロセス変数とプロセス変数

プロセス変数やインタープロセス変数のタイプの矛盾は、次のように分類できません。

- 2種類の用途による矛盾
- 用途とコンパイラ命令の矛盾
- 暗黙のタイプ変更による矛盾
- 2つのコンパイラ命令による矛盾

2 種類の用途による 矛盾

最も単純なタイプの矛盾は、1つの変数名で2つの異なるオブジェクトを指している場合です。

たとえば、以下のように書き、

```
Variable:=5
```

また、同じデータベースに、次のように書いたとします。

```
Variable:=True
```

この2つのステートメントは、タイプの矛盾を引き起こします。ほとんどの場合、どちらか一方の変数名を変更するだけで解決できます。

用途とコンパイラ命令の矛盾

たとえば、以下のように書き、

```
Variable:=5
```

また、同じデータベースに、次のように書いたとします。

```
C_BOOLEAN (Variable)
```

コンパイラ命令が最初に処理されるので、Variable はブールタイプに設定されますが、“Variable:=5”というステートメントがあるので、タイプの矛盾とみなされます。変数名を変えるか、コンパイラ命令を変更すれば解決できます。

1つの式に使われている変数のデータタイプが異なる場合も、タイプの矛盾が起こるので、4D Compiler は、タイプの不一致とみなします。

- ▼ 簡単な例があります。

```
Bool:=True
```

`Bool のデータタイプはブールです。

```
C_INTEGER (< >Integer)
```

```
< >Integer:=3
```

`代入値はコンパイラの指定と同じタイプ

```
Var:=< >Integer+Bool
```

`データタイプが一致しない変数
`を使用した演算

暗黙のタイプ変更による矛盾

関数の中には、返す値のデータタイプが明確に決まっているものがあります。このような関数が返す値を、異なったタイプの変数に代入すると、データタイプの矛盾が起こります。

- ▼ 例えば、インタプリタでは以下のように書いてもエラーになりません。

```
IdentNo:=Request ("Identification Number")
```

`Ident No のデータタイプはテキストです。

```
If (OK=1)
```

```
IdentNo:=Num (IdentNo)
```

`Ident No のデータタイプは実数です。

```
QUERY ([Contacts]ID=IdentNo)
```

```
End if
```

この例では、3行目にタイプの矛盾があります。タイプの矛盾を解決する方法としては、別の名前の中間変数を作ればよい場合や、この例のようにメソッドの構造の変更で修正できる場合もあります。

```
IdentNo:=Num (Request ("Identification Number"))
```


\Ident No のデータタイプは実数です。

```
If (OK=1)
  QUERY ([Contacts]ID=IdentNo)
End if
```

2つのコンパイラ命令による矛盾

同じ変数に対して、2つの異なるコンパイラ命令を使用すると、タイプの再定義になります。たとえば、1つのデータベース内で、次のように書いたとします。

```
C_BOOLEAN (Variable)
```

そして

```
C_TEXT (Variable)
```

コンパイラによってタイプの矛盾が検出され、エラーファイルに出力されます。一般に、どちらかの変数名を変えれば問題は解決します。

C_STRING コマンドで文字長を変更すると、データタイプの矛盾が起こることがあります。たとえば、次のように書くと、コンパイラはタイプの矛盾とみなします。

```
C_STRING (5;MyString)
```

```
MyString:="Flour"
```

```
C_STRING (7;MyString)
```

```
MyString:="Flowers"
```

文字タイプの変数が定義されると、コンパイラは与えられたサイズでエリアを割り当てなければならないからです。

コンパイラは短い方のサイズを採用します。このような矛盾を解決するには、サイズ指定が必要なコンパイラ命令は1回だけしか使わないようにすることです。以下のように書くことができます。

```
C_STRING (7;MyString)
```

```
MyString:="Flour"
```

```
MyString:="Flowers"
```

ローカル変数

ローカル変数のデータタイプの矛盾は、プロセス変数やインタープロセス変数とほとんど同じです。唯一の違いは、特定のメソッドの中でのみタイプが一貫していればよい、という点です。

プロセス変数やインタープロセス変数の場合、矛盾が起こるのはデータベース全体のレベルでしたが、ローカル変数で問題になるのは、メソッドのレベルです。たとえば、1つのメソッドの中で、

```
$Temp:="Flowers"
```

と、記述し、そして次に

```
$Temp:=5
```

と記述するとエラーになりますが、メソッド M1 には、次のように記述できます。

```
$Temp:="Flowers"
```

そして、メソッド M2 には、

```
$Temp:=5
```

と記述できます。これは、ローカル変数の範囲がデータベース全体ではなくメソッドだからです。

配列内の対立

配列の要素数は、タイプの矛盾とは無関係です。コンパイル前のデータベースと同じく、配列は動的に管理されます。配列の要素数はどのメソッドでも変更できるし、最大要素数を定義する必要もありません。そのため、配列の要素数を 0 にしたり、要素を追加し、消去し、内容を削除することができます。

コンパイルを前提にしてデータベースを作る場合は、以下のような原則に従ってください。

- 配列要素のデータタイプを変えない
- 配列の次元数を変えない
- 文字配列では、文字の長さを変えない

配列要素のデータタイプの変更

配列を ARRAY INTEGER などのコマンドで定義したら、その配列はデータベース全体を通して整数配列でなければいけません。

次のように書くと、

```
ARRAY INTEGER (MyArray;5)  
ARRAY BOOLEAN (MyArray;5)
```

コンパイラでは MyArray のタイプを決定できません。どちらか一方の配列名を変えてください。

配列の次元数の変更

コンパイル前のデータベースでは、配列の次元数を変更できません。コンパイラでシンボルテーブルを作成する場合、1次元配列と2次元配列では処理方法が異なるので、1次元配列を2次元配列に定義し直すことはできません。逆も同じです。従って、同じデータベースでは、次のように定義することはできません。

```
ARRAY INTEGER (Array1;10)
ARRAY INTEGER (Array1;10;10)
```

しかし、同じアプリケーション内で以下のように記述することはできます。

```
ARRAY INTEGER (Array1;10)
ARRAY INTEGER (Array2;10;10) 配列名が違うことに注目
```

同じデータベース内で配列の次元数を変更することはできませんが、配列の要素数は変更できます。2次元配列の1番目の配列の要素数を変更するには、次のように記述します。

```
ARRAY BOOLEAN (Array;5)
ARRAY BOOLEAN (Array;10) 配列の要素数を変更する
```

注 2次元配列は、実際には複数の1次元配列から構成されています。詳細は、『*4th Dimension ランゲージリファレンス*』を参照してください。

文字配列

文字配列には、固定長文字列と同じ規則が適用されます。

次のように書くと、

```
ARRAY STRING (5;Array;10)
ARRAY STRING (10;Array;10)
```

コンパイラはサイズの矛盾とみなします。解決するには一番大きいサイズを定義してください。それよりサイズの小さい文字列は、4D Compiler が自動的に処理します。

暗黙のタイプ変更

COPY ARRAY, LIST TO ARRAY, ARRAY TO LIST, SELECTION TO ARRAY, ARRAY TO SELECTION, SUBSELECTION TO ARRAY, DISTINCT VALUES コマンドを使用する際、意識的に、または誤って要素のデータタイプや次元数、文字配列の文字サイズを変更してしまうことがあります。前で説明した3つの状況のうち、必ずいずれかに該当します。

コンパイラはエラーメッセージを出力しますので、修正すべき点が明確になります。暗黙に配列のタイプが変更される例は、[第6章「配列」](#)を参照して下さい。

ローカル配列

データベースでローカル配列を使う場合は、あらかじめ4th Dimension でそれらを明示的に定義しなければいけません。たとえば、メソッドで10要素のローカル整数配列を作る場合、そのメソッドに次の行を追加します。

```
ARRAY INTEGER ($MyArray; 10)
```

フォーム変数

フォーム内で作られる変数（ボタン、ドロップダウンリストボックスなど）はすべてプロセス変数とインタープロセス変数です。インタプリタでは、これらの変数のタイプを変更しても問題にはなりません、コンパイルする場合は注意が必要です。しかし、考え方は単純で、

- コンパイラ命令を使って、フォーム変数のタイプを定義する
- コンパイラがデフォルトで適切なデータタイプを割り当てる

数値タイプに設定されるフォーム変数

次のフォーム変数は、プロジェクト内で実数以外の指定がされていないければ、実数タイプに設定されます。

チェックボックス、3Dチェックボックス、ボタン、ハイライトボタン、透明ボタン、3Dボタン、ピクチャボタン、ボタングリッド、ラジオボタン、3Dラジオボタン、ラジオピクチャ、ピクチャメニュー、階層式ポップアップメニュー、及び階層式リスト。

次のフォーム変数は、プロジェクト内で倍長整数を選択しても実数タイプに設定されます。

ルーラー、ダイアル、及びサーモメーター

フォーム変数について、データタイプの矛盾が起こるとすれば、それは、データベースの他の場所で使われている別の変数に、同じ名前が付けられている場合です。このような場合は、変数の名前を変更してください。

グラフ変数

グラフエリアのデータタイプは自動的にグラフタイプになるので、タイプの矛盾が起こることはまずありません。グラフタイプの変数について、データタイプの矛盾が起こるとすれば、それは、データベースの他の場所で使われている別の変数に、同じ名前が付けられている場合です。このような場合は、変数の名前を変更してください。

プラグインオブジェクト変数

プラグインオブジェクトは常に倍長整数ですので、データタイプの矛盾はまずありません。プラグインエリアタイプの変数についてデータタイプの矛盾が起こるとすれば、それは、データベースの他の場所で使われている別の変数に、同じ名前が付けられている場合です。このような場合は、変数の名前を変更してください。

テキストタイプに設定されるフォーム変数

テキストタイプに設定されるフォーム変数は8個あります。それは、入力不可変数、入力可変数、ドロップダウンリスト、メニュー/ドロップダウンリスト、スクロールエリア、ポップアップメニュー、コンボボックス及びタブコントロールの8種類になります。

これらの変数は2つのカテゴリーに分類できます。

- 単純変数：入力不可の変数、入力可の変数
- 表示変数：ドロップダウンリスト、メニュー/ドロップダウンリスト、スクロールエリア、ポップアップメニュー、コンボボックス及びタブコントロール。

単純変数

デフォルトのデータタイプはテキストです。メソッドやオブジェクトメソッドで使われるときは、ユーザが選択したデータタイプが割り当てられます。同じ名前

で異なるタイプの変数が存在する場合以外は、データタイプの矛盾が起こることはありません。

表示変数

変数はフォーム内で配列を表示する為に使用されます。オブジェクトプロパティウィンドウのデフォルト値ダイアログボックス内にデフォルト値を入力すると、配列定義コマンド **ARRAY STRING** または **ARRAY TEXT** を使用して、関連する変数を明確に定義しなければいけません。

ポインタ

ポインタを使うと、4th Dimension の強力で多彩な機能を活用することができます。コンパイル後もポインタの利点をそのまま活用できます。1つのポインタでデータタイプの異なる変数を指定することができます。ポインタが示す変数にタイプの異なるデータを代入して矛盾を引き起こさないようにしてください。ポインタで参照する変数のデータタイプを変更しないよう注意してください。

- ▼ この問題の例を示します。

```
Variable:=5.3
Pointer:=>Variable
Pointer->:=6.4
Pointer->:=False
```

この例では、ポインタで参照する変数のタイプは実数です。これにブールの値を代入しているため、データタイプの矛盾が起こります。1つのメソッド内で、異なる目的のためにポインタを使う場合には、参照先の変数のタイプを確認してください。

- ▼ ポインタの正しい使い方の例を示します。

```
Variable:=5.3
Pointer:=>Variable
Pointer->:=6.4
Bool:=True
Pointer:=>Bool
Pointer->:=False
```

ポインタには、参照するオブジェクトとの関係だけが定義されています。ポインタによって起こるデータタイプの矛盾をコンパイラが検知できないのはこのためです。矛盾があっても、“タイプチェック処理”フェーズや“コンパイル処理”フェーズでエラーメッセージは出力されません。ただし、ポインタに関する矛盾を見つける方法がまったくないわけではありません。“範囲チェック”オプションによってポインタの使用状況をいくらかは解析できます。“範囲チェック”オプションについては、[第3章「範囲チェック」](#)を参照してください。

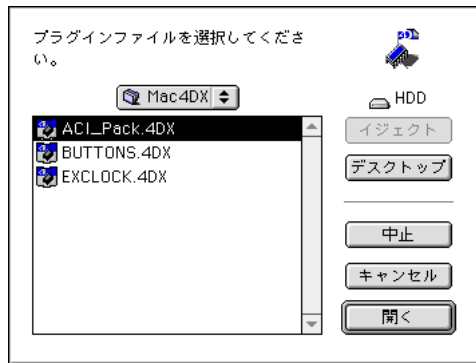
プラグインコマンド

4th Dimension のプラグインを使用する場合、4D Compiler は、こうしたコマンドへの呼び出しを含んでいるメソッドもコンパイルします

プラグインコマンドを含むアプリケーションをコンパイルするには、

- Windows のプラグインを "WIN4DX" フォルダに、Macintosh のプラグインを "MAC4DX" フォルダに入れます。これらフォルダは両方とも、プロシージャファイルと同じ階層か、あるいは ACI フォルダの中に配置します。コンパイラは、これらのファイルを複製しませんが、これらを分析して、これらのルーチンの適切な定義を決定します。

プラグインがプロシージャと階層にない場合には、4D Compiler はプラグインを含んだファイルを指定するダイアログを表示します。



プラグインコマンドファイルを選択し、「開く」ボタンをクリックして下さい。コンパイラはメソッドの定義を分析し、呼び出しがメソッドの定義と一致したものであれば、タイプ定義の矛盾が起こる危険性はありません。プラグインコマンドに定義されている引数より少ない個数の引数を渡すこともできます。この場合、省略する引数はその並びの後ろにあるということが前提です。

マルチプラットフォームコンパイル

Windows 版 4D Compiler でのプラットフォーム独立型コンパイル

4D Server(Win) に 4D Client(Mac) から接続する場合、プラグインは Macintosh 版を使用します。

4D Client(Mac) から 4D Server(Win) にあるプラグインを使用するには、

- 1 まず Macintosh 版プラグインを、ご使用のマシンにインストールします。
- 2 4DTransporter を使って Macintosh 版プラグインをトランスポートします。

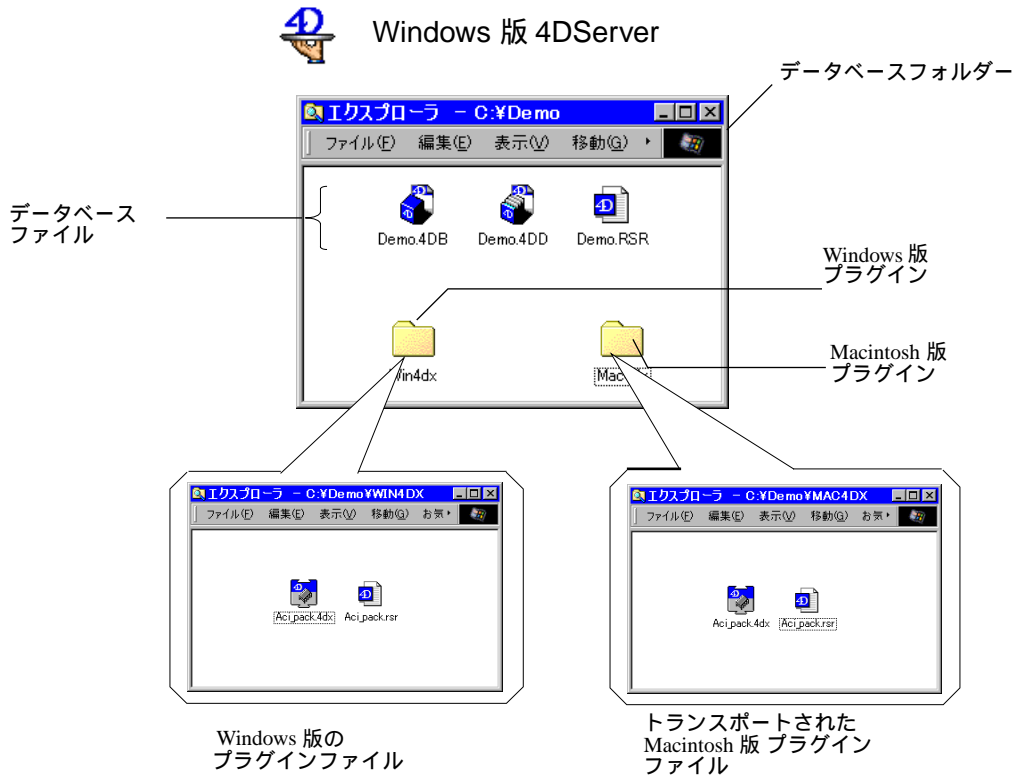
注 別プラットフォームへの変換については、4DTransporter のマニュアルを参照して下さい。

「データベース名.PC」フォルダが作成され、その中にトランスポートされた Macintosh 版プラグインが、「Plug-inName.4DX」ファイルと「Plug-inName.RSR」ファイルとして作成されます。

- 3 Windows 上に「Mac4DX」という名前の、新しいフォルダを作成します。
- 4 トランスポートされた2つのファイルを Macintosh から Windows 上の「Mac4DX」フォルダにコピーして下さい。
- 5 「Mac4DX」フォルダを「Win4DX」フォルダとともに、データベースのストラクチャファイルと同じ階層に置いて下さい。
- 6 4D Server を実行し、データベースを開いて下さい。

これで、Macintosh 版、Windows 版の両クライアントでプラグインを使うことができるようになります。

下記の図はこのプロセスを示したものです。



- 注 Proc.Ext ファイル内の 680xx コマンドを使用しているデータベースをコンパイルするには、最初に、4DTransporter で "Proc.Ext" ファイルをトランスポートさせて、"Proc.Esr" ファイルを、ストラクチャファイルと同一レベルに置かなければいけません。
- 4D Server を使ったプラットフォーム独立型についての詳細は、各プラグインのマニュアルを参照して下さい。
-

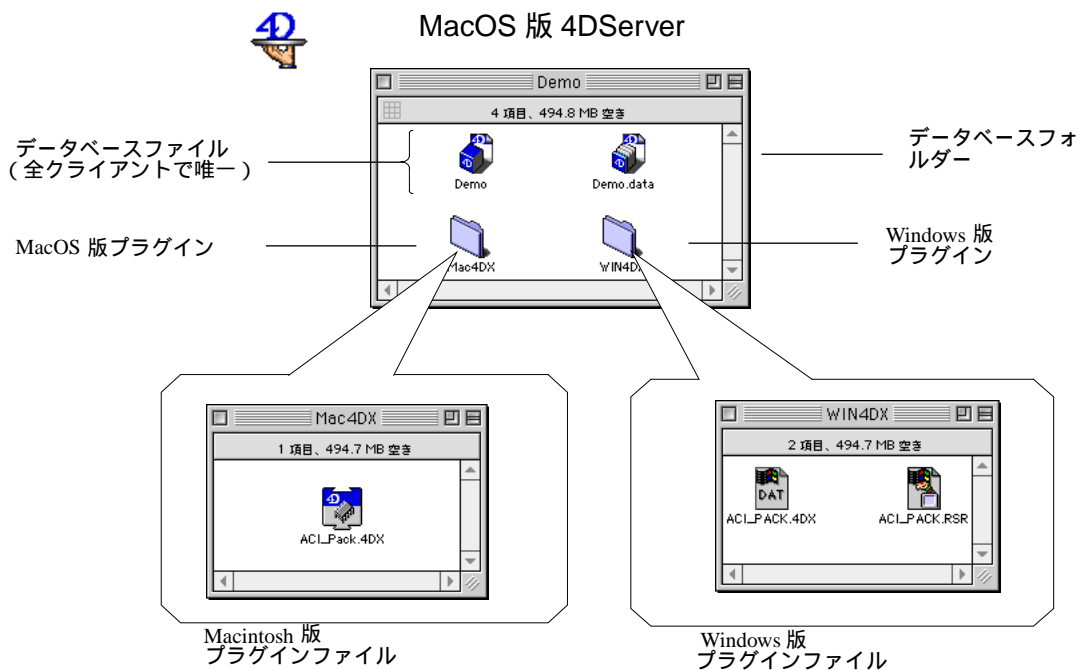
Macintosh 版
4D Compiler でのプラットフォーム独立型コンパイル

4D Server(Mac) に 4D Client(Win) から接続する場合、プラグインは Windows 版を使用します。

4D Client(Win) から 4D Server(Mac) にあるプラグインを使用するには、Windows 版プラグインを、Windows マシンにインストールします。

- 7 インストールした Windows プラグインを、ご使用の Macintosh にコピーします。
Windows 版プラグインは、「Plug-inName.4DX」ファイル及び、「Plug-inName.RSR」ファイル、及び「Plug-inName.HLP」から構成されます。
- 8 Macintosh 上に、新しいフォルダを作成し、それに「Win4DX」と名前を付けます。
Windows 版プラグインのファイルを、このフォルダにコピーします。
- 9 「Win4DX」フォルダを「Mac4DX」とともに、データベースストラクチャファイルと同じ階層に配置します。

下記の図はこのプロセスを示したものです。



10 4D Server を実行しデータベースを開いて下さい。

これで、Macintosh 版、Windows 版の両クライアントでプラグインを使うことができるようになります。

注 Proc.Ext ファイル内の 680xx コマンドを使用しているデータベースをコンパイルするには、"Proc.Esr" ファイルを、ストラクチャファイルと同一レベルに置かなければいけません。4D Server を使ったプラットフォーム独立型についての詳細は、各プラグインのマニュアルを参照して下さい。

マルチプラットフォーム コンパイルについての 重要事項

プラグインを使用しているデータベースをマルチプラットフォーム用にコンパイルするためには Macintosh 版および Windows 版プラグインのバージョンは同じでなければいけません。

- 1つのデータベースが Macintosh および Windows 上で同じコマンドを使用している場合、それぞれのコマンドの各 ID 番号と引数の数は、Macintosh 版および Windows 版とも同一でなければいけません。
- コンパイル中は、Macintosh 版および Windows 版のプラグインは同一名称でなければいけません。

- ▼ 両プラットフォームから接続可能な 4D Server を構築するためのコンパイル時のファイル構成例を 4D Draw プラグインで示します。
 - Macintosh 版 4D Compiler での構成例。
 - DatabaseName**
 - DatabaseName.data**
 - Mac4DX**
 - 4DDraw.4DX**
 - Win4DX**
 - 4DDraw.4DX**
 - 4DDraw.RSR**
 - Windows 版 4D Compiler での構成例。
 - DBName.4DB**
 - DBName.RSR**
 - DBName.4DD**
 - Mac4DX**
 - 4DDraw.4DX**
 - 4DDraw.RSR**
 - Win4DX**
 - 4DDraw.4DX**
 - 4DDraw.RSR**

MacintoshあるいはWindowsでの実行形式のアプリケーションのコンパイルと作成

Macintosh 版 4D Compiler で実行形式のアプリケーションを作成する場合、データベースのストラクチャファイル、「Mac4DX」フォルダ、Proc.Ext ファイルが同じ階層にあることを確認して下さい。

Windows 版 4D Compiler で実行形式のアプリケーションを作成する場合、データベースのストラクチャファイル、「Win4DX」フォルダが同じ階層にあることを確認して下さい。同じ階層にないと 4D Compiler は、コンパイル中にエラーメッセージを表示します。

OLE ライブラリー (Windows) でのコンパイル

OLE ライブラリーを使用しているデータベースをコンパイルする前に次の事を行して下さい。

- 1 「Win4DX」フォルダがないときは作成して下さい。
- 2 OLETOOLS.DLL ファイルの名前を "OLETOOLS.4DX" に変更します。
- 3 OLETOOLS.4DX と OLETOOLS.RSR ファイルを「Win4DX」フォルダ内に配置します。

コンパイルしたデータベースを 4th Dimension 上で実行する際、次のいずれかの方法を取る事ができます。

- 「Win4DX」フォルダに OLETOOLS.4DX と OLETOOLS.RSR ファイルを配置する。
OLETOOLS.DLL と OLETOOLS.RSR ファイルを 4D.EXE ファイルとして同じ階層に配置する。

実行形式のアプリケーションを実行するときは、「Win4DX」フォルダに OLETOOLS.4DX と OLETOOLS.RSR ファイルを入れておく必要があります。

暗黙の引数を受け取る プラグインコマンド

特定のプラグイン（4D Calc, 4D Write, 4D SQL Server など）は、暗黙に 4th Dimension ルーチン呼び出すルーチンを実行します。例えば 4D Draw の場合。ルーチン **WR ON EVENT** の構文は次のようになります。

WR ON EVENT (EventMethod)

最後の引数は、あなたが 4th Dimension で作成したメソッドの名前です。このメソッドはイベントを受け取られるたびに 4D Write によって呼び出されます。

メソッドは自動的にこれらの引数を受け取ります。

引数	タイプ	内容
\$1	倍長整数	4D Write エリア
\$2	倍長整数	イベントコード
\$3	倍長整数	エリアが属するフォームのテーブル番号
\$4	倍長整数	エリアが自動保存されるフィールド番号

コンパイラがこれらの引数の存在を認識し、これらを考慮するためには、引数をコンパイル命令、またはメソッド内での使用方法によってタイプ設定しなければなりません。メソッドで使用する場合には、タイプを明確に推測できるように使用する必要があります。

プラグインコマンド で作成される変数

プラグインコマンドで作成される変数は、その変数がデータベースストラクチャのコード内で定義された場合にのみ、4D Compiler によって認識されます。

Macintosh

4D Compiler にこれらの変数を確実に認識させる為には、ResEdit のようなリソースエディタを使って、データベースストラクチャのファイル内にリソースを作成する必要があります。プロセス変数の場合は VAR# リソースに、インタープロセス変数の場合は VAR<> リソースに、それぞれの変数を定義することができます。

Windows

VAR# 及び VAR<> のリソースが存在する場合は、Macintosh プラットフォーム上と同様に、Windows のプラットフォーム上でも認識され使用することができます。これらのリソースが必要な場合は、Macintosh 上で ResEdit のようなリソースエディタを使って作成しなければいけません。

引数の処理

ローカル変数、 $\$0...\n 、の処理はこれまでに述べた規則に従います。他のローカル変数と同様に、プロシージャの実行中にデータタイプを変更することはできません。ここでは、タイプの矛盾を起こす可能性のある2つのケースについて検討します。

- タイプ変更が必要な場合：
ポインタの使用によりデータタイプの矛盾を避けることができます。
- 引数を間接参照する場合。

ポインタを使って データタイプの矛盾 を避ける

変数のタイプを変更することはできませんが、ポインタを使って異なるタイプのデータを参照することはできます。

- ▼ 1次元配列のメモリサイズを返す関数を作成します。この例ではテキスト配列とピクチャ配列を除き、実数の結果を返します。さらにテキスト配列でピクチャ配列のデータサイズは計算式で求められないため、文字列の結果を返します。この関数の引数は、メモリサイズを調べようとする配列へのポインタです。

この操作を行うには2つの方法があります。

- ローカル変数のデータタイプを気にせずに使用する方法。このメソッドはインタプリタでしか動作しません。
- ポインタを使い、インタプリタでもコンパイル後でも処理できるようにする方法。

、インタプリタモードのみでの *MemSize* 機能

\$Size:=Size of array (\$1->) + 1

\$Type:=Type (\$1->)

Case of

(\$Type=14)

\$0:=8+(\$Size*10)

¥ (\$Type=15)

\$0:=8+(\$Size*2)

¥ (\$Type=16)

\$0:=8+(\$Size*4)

¥ (\$Type=18)

\$0:=String (8+(\$Size*4)) + "+Sum of text sizes"

¥(\$Type=19)

\$0:=String (8+(\$Size*4)) + "+Sum of picture sizes"

End case

このメソッドでは、\$0 のデータタイプが \$1 の配列によって違うので、コンパイルできません。次にポインタを使う方法を示します。

```

`インタプリタ及びコンパイルモードでの MemSize 関数
$Size:=Size of array ($1->) +1
$Type:=Type ($1->)
VarNum:=0
Case of
  ¥($Type=14)           `実数配列
    VarNum:=8+($Size*10) `VarNum は実数
  ¥($Type=15)           `配列整数
    VarNum:=8+($Size*2)  ``VarNum は実数
  ¥($Type=16)           `配列長整数
    VarNum:=8+($Size*4)  `VarNum は実数
  ($Type=18)            `配列はテキスト (Text)
    VarText:=String (8+($Size*4)) +"+Sum of text sizes"
  ¥($Type=19)           `配列はピクチャー
    VarText:=String (8+($Size*4)) +"+Sum of picture sizes"
End case
If (VarNum#0)
  $0:=->VarNum
Else
  $0:=->VarText
End if

```

2つの関数は、以下の点が違います。

- 1番目の関数では、結果が変数であること。
- 2番目の関数では、結果が変数へのポインタであること。

結果はポインタで簡単に参照できます。

引数の参照

4D Compiler は、引数の間接参照の機能をサポートしています。インタプリタでは、引数の数とデータタイプを自由に設定できます。この機能は、タイプの矛盾や引数の矛盾（呼ばれた側で、セットされていない引数を使う）さえなければ、コンパイル後も保証されています。

タイプの矛盾の元になるので、間接参照する引数は、すべて同じデータタイプにしてください。間接参照を上手に使うためには、間接参照する引数は他の引数の後に配置するようにします。

メソッド内で、間接参照は "\${\$i}" のように表示します。"\$i" は数値変数です。"\${\$i}" を "包括引数" と呼びます。

- ▼ 以下は間接参照の例です。数値を合計し、引数として与えられたフォーマットで編集して返すような関数を考えてください。数値の個数は、メソッドが呼ばれるたびに変わります。このメソッドでは数値と編集フォーマットを引数としてメソッドに渡さなければなりません。

この関数は、以下のようにして呼びます。

```
Result := MySum ("##0.00";125.2;33.5;24)
```

この場合、数値を合計し、指定されたフォーマットに編集して "182.70" が返されます。引数は正しい順序で渡してください。最初にフォーマット、次に値です。

MySum 関数です。

```
$Sum:=0
For ($i;2;Count parameters)
  $Sum:=$Sum+${i}
End for
$0:=String ($Sum;$1)
```

この関数は、次のように引数の個数を替えて使う事ができます。

```
MySum ("##0.00";125.2;2;33.5;24)
```

あるいは

```
MySum ("000";1;18;4;23;17)
```

他のローカル変数と同様に、包括引数をコンパイラ命令で定義する必要はありませんが、曖昧になりそうな時や最適化のために必要な場合は、以下のように記述します。

```
C_INTEGER (${2})
```

これは、2番目以降の引数のデータタイプが実数であるという意味です。データタイプは実数です。*\$1* のデータタイプはこのコマンドには関係ありません。

注 コンパイラは、タイプ設定フェーズでこのコマンドを使います。定義内の数字は変数ではなく定数でなければなりません。

予約変数

4th Dimension の変数にはコンパイラによって、データタイプと名前が割り当てられているものがあります。このため、ユーザはこれらの変数名を、新しい変数や、メソッド、関数、プラグインコマンドに使用することはできません。インタプリタと同様、条件式等で使用することはできます。

システム変数

4th Dimension のシステム変数及びそれらのデータタイプの完全なリストを紹介します。

システム変数	タイプ
OK	倍長整数
Document	文字列 (255)
FldDelimit	倍長整数
RecDelimit	倍長整数
Error	倍長整数
MouseDown	倍長整数
KeyCode	倍長整数
Modifiers	倍長整数
MouseX	倍長整数
MouseY	倍長整数
MouseProc	倍長整数

クイックレポート変数 クイックレポートで計算用のカラムを作る際、4th Dimension は、第 1 カラム用に
変数 C1、第 2 カラムに変数 C2、第 3 カラムに変数 C3、のように自動的に変数を作
作成します。この処理はユーザには見えません。

これらの変数をフォーミュラで使用する場合は、他の変数と同様に、C1, C2,
.....Cn はタイプ変更することができません。

定数

次の表は、4th Dimension で定義されている定数のリストです。

定数	タイプ	値
<i>4D Client</i>	倍長整数	4
<i>4D Engine</i>	倍長整数	1
<i>4D First</i>	倍長整数	6
<i>4D Runtime</i>	倍長整数	2
<i>4D Runtime Classic</i>	倍長整数	3
<i>4D Server</i>	倍長整数	5
<i>4th Dimension</i>	倍長整数	0
<i>Abbr Month Date</i>	倍長整数	6
<i>Abbreviated</i>	倍長整数	2
<i>Aborted</i>	倍長整数	-1
<i>ACK ASCII code</i>	倍長整数	6
<i>Activate event</i>	倍長整数	8
<i>Activate window bit</i>	倍長整数	0
<i>Activate window mask</i>	倍長整数	1
<i>Ala Macintosh</i>	倍長整数	1
<i>Ala Windows</i>	倍長整数	2
<i>Alternate dialog box</i>	倍長整数	3
<i>April</i>	倍長整数	4
<i>Array 2D</i>	倍長整数	13
<i>At sign</i>	倍長整数	64
<i>August</i>	倍長整数	8
<i>Auto key event</i>	倍長整数	5
<i>Automatic interface</i>	倍長整数	-1
<i>Backspace</i>	倍長整数	8
<i>Backspace Key</i>	倍長整数	8
<i>BEL ASCII code</i>	倍長整数	7
<i>Black</i>	倍長整数	15
<i>Black and white</i>	倍長整数	0
<i>Blue</i>	倍長整数	6
<i>Bold</i>	倍長整数	1

定数	タイプ	値
<i>Boolean array</i>	倍長整数	22
<i>Brown</i>	倍長整数	13
<i>BS ASCII code</i>	倍長整数	8
<i>C string</i>	倍長整数	0
<i>CAN ASCII code</i>	倍長整数	24
<i>Caps Lock key bit</i>	倍長整数	10
<i>Caps Lock key mask</i>	倍長整数	1024
<i>Carriage return</i>	倍長整数	13
<i>Changed resource bit</i>	倍長整数	1
<i>Changed resource mask</i>	倍長整数	2
<i>Command key bit</i>	倍長整数	8
<i>Command key mask</i>	倍長整数	256
<i>Compact compression mode</i>	倍長整数	1
<i>Condensed</i>	倍長整数	32
<i>Control key bit</i>	倍長整数	12
<i>Control key mask</i>	倍長整数	4096
<i>Copland interface</i>	倍長整数	3
<i>CR ASCII code</i>	倍長整数	13
<i>Dark Blue</i>	倍長整数	5
<i>Dark Brown</i>	倍長整数	10
<i>Dark Green</i>	倍長整数	9
<i>Dark Grey</i>	倍長整数	11
<i>Data bits 5</i>	倍長整数	0
<i>Data bits 6</i>	倍長整数	2048
<i>Data bits 7</i>	倍長整数	1024
<i>Data bits 8</i>	倍長整数	3072
<i>Date array</i>	倍長整数	17
<i>DC1 ASCII code</i>	倍長整数	17
<i>DC2 ASCII code</i>	倍長整数	18
<i>DC3 ASCII code</i>	倍長整数	19
<i>DC4 ASCII code</i>	倍長整数	20
<i>December</i>	倍長整数	12
<i>Default background color</i>	倍長整数	-2
<i>Default dark shadow color</i>	倍長整数	-3
<i>Default foreground color</i>	倍長整数	-1
<i>Default light shadow color</i>	倍長整数	-4
<i>Degree</i>	実数	0.0174532925199432958

定数	タイプ	値
<i>DEL ASCII code</i>	倍長整数	127
<i>Delayed</i>	倍長整数	1
<i>Delete Record Event</i>	倍長整数	3
<i>Demo Version</i>	倍長整数	1
<i>Disk event</i>	倍長整数	7
<i>DLE ASCII code</i>	倍長整数	16
<i>Does not exist</i>	倍長整数	-100
<i>Double quote</i>	倍長整数	34
<i>Down Arrow Key</i>	倍長整数	31
<i>e number</i>	実数	2.71828182845904524
<i>EM ASCII code</i>	倍長整数	25
<i>End Key</i>	倍長整数	4
<i>ENQ ASCII code</i>	倍長整数	5
<i>Enter</i>	倍長整数	3
<i>Enter Key</i>	倍長整数	3
<i>EOT ASCII code</i>	倍長整数	4
<i>ESC ASCII code</i>	倍長整数	27
<i>Escape</i>	倍長整数	27
<i>Escape Key</i>	倍長整数	27
<i>ETB ASCII code</i>	倍長整数	23
<i>ETX ASCII code</i>	倍長整数	3
<i>Executing</i>	倍長整数	0
<i>Extended</i>	倍長整数	64
<i>Extended real format</i>	倍長整数	1
<i>F1 Key</i>	倍長整数	-122
<i>F10 Key</i>	倍長整数	-109
<i>F11 Key</i>	倍長整数	-103
<i>F12 Key</i>	倍長整数	-111
<i>F13 Key</i>	倍長整数	-105
<i>F14 Key</i>	倍長整数	-107
<i>F15 Key</i>	倍長整数	-113
<i>F2 Key</i>	倍長整数	-120
<i>F3 Key</i>	倍長整数	-99
<i>F4 Key</i>	倍長整数	-118
<i>F5 Key</i>	倍長整数	-96
<i>F6 Key</i>	倍長整数	-97
<i>F7 Key</i>	倍長整数	-98

定数	タイプ	値
<i>F8 Key</i>	倍長整数	-100
<i>F9 Key</i>	倍長整数	-101
<i>Fast compression mode</i>	倍長整数	2
<i>February</i>	倍長整数	2
<i>FF ASCII code</i>	倍長整数	12
<i>Floating window</i>	倍長整数	14
<i>Four colors</i>	倍長整数	2
<i>Friday</i>	倍長整数	6
<i>FS ASCII code</i>	倍長整数	28
<i>Full Version</i>	倍長整数	0
<i>Green</i>	倍長整数	8
<i>Grey</i>	倍長整数	14
<i>GS ASCII code</i>	倍長整数	29
<i>Has grow box</i>	倍長整数	4
<i>Has highlight</i>	倍長整数	1
<i>Has window title</i>	倍長整数	2
<i>Has zoom box</i>	倍長整数	8
<i>Help Key</i>	倍長整数	5
<i>HH MM</i>	倍長整数	2
<i>HH MM AM PM</i>	倍長整数	5
<i>HH MM SS</i>	倍長整数	1
<i>Hidden modal dialog</i>	倍長整数	6
<i>Home Key</i>	倍長整数	1
<i>Hour Min</i>	倍長整数	4
<i>Hour Min Sec</i>	倍長整数	3
<i>HT ASCII code</i>	倍長整数	9
<i>In contents</i>	倍長整数	3
<i>In drag</i>	倍長整数	4
<i>In go away</i>	倍長整数	6
<i>In grow</i>	倍長整数	5
<i>In menu bar</i>	倍長整数	1
<i>In system window</i>	倍長整数	2
<i>In zoom box</i>	倍長整数	8
<i>Integer array</i>	倍長整数	15
<i>INTEL 386</i>	倍長整数	386
<i>INTEL 486</i>	倍長整数	486
<i>Into current selection</i>	倍長整数	0

定数	タイプ	値
<i>Into named selection</i>	倍長整数	2
<i>Into set</i>	倍長整数	1
<i>Into variable</i>	倍長整数	3
<i>Is a directory</i>	倍長整数	0
<i>Is a document</i>	倍長整数	1
<i>Is Alpha Field</i>	倍長整数	0
<i>Is BLOB</i>	倍長整数	30
<i>Is Boolean</i>	倍長整数	6
<i>Is color</i>	倍長整数	1
<i>Is Date</i>	倍長整数	4
<i>Is gray scale</i>	倍長整数	0
<i>Is Integer</i>	倍長整数	8
<i>Is LongInt</i>	倍長整数	9
<i>Is new record</i>	倍長整数	-3
<i>Is not compressed</i>	倍長整数	0
<i>Is Picture</i>	倍長整数	3
<i>Is Pointer</i>	倍長整数	23
<i>Is Real</i>	倍長整数	1
<i>Is String Var</i>	倍長整数	24
<i>Is Subtable</i>	倍長整数	7
<i>Is Text</i>	倍長整数	2
<i>Is Time</i>	倍長整数	11
<i>Is Undefined</i>	倍長整数	5
<i>ISO L1 a acute</i>	文字列	á
<i>ISO L1 a circumflex</i>	文字列	â
<i>ISO L1 a grave</i>	文字列	à
<i>ISO L1 a ring</i>	文字列	å
<i>ISO L1 a tilde</i>	文字列	ã
<i>ISO L1 a umlaut</i>	文字列	ä
<i>ISO L1 ae ligature</i>	文字列	æ
<i>ISO L1 Ampersand</i>	文字列	&
<i>ISO L1 c cedilla</i>	文字列	ç
<i>ISO L1 Cap A acute</i>	文字列	Á
<i>ISO L1 Cap A circumflex</i>	文字列	Â
<i>ISO L1 Cap A grave</i>	文字列	À
<i>ISO L1 Cap A ring</i>	文字列	Å
<i>ISO L1 Cap A tilde</i>	文字列	Ã

定数	タイプ	値
<i>ISO L1 Cap A umlaut</i>	文字列	Ä
<i>ISO L1 Cap AE ligature</i>	文字列	&AELig;
<i>ISO L1 Cap C cedilla</i>	文字列	Ç
<i>ISO L1 Cap E acute</i>	文字列	É
<i>ISO L1 Cap E circumflex</i>	文字列	Ê
<i>ISO L1 Cap E grave</i>	文字列	È
<i>ISO L1 Cap E umlaut</i>	文字列	Ë
<i>ISO L1 Cap Eth Icelandic</i>	文字列	Ð
<i>ISO L1 Cap I acute</i>	文字列	Í
<i>ISO L1 Cap I circumflex</i>	文字列	Î
<i>ISO L1 Cap I grave</i>	文字列	Ì
<i>ISO L1 Cap I umlaut</i>	文字列	Ï
<i>ISO L1 Cap N tilde</i>	文字列	Ñ
<i>ISO L1 Cap O acute</i>	文字列	Ó
<i>ISO L1 Cap O circumflex</i>	文字列	Ô
<i>ISO L1 Cap O grave</i>	文字列	Ò
<i>ISO L1 Cap O slash</i>	文字列	Ø
<i>ISO L1 Cap O tilde</i>	文字列	Õ
<i>ISO L1 Cap O umlaut</i>	文字列	Ö
<i>ISO L1 Cap THORN Icelandic</i>	文字列	Þ
<i>ISO L1 Cap U acute</i>	文字列	Ú
<i>ISO L1 Cap U circumflex</i>	文字列	Û
<i>ISO L1 Cap U grave</i>	文字列	Ù
<i>ISO L1 Cap U umlaut</i>	文字列	Ü
<i>ISO L1 Cap Y acute</i>	文字列	Ý
<i>ISO L1 Copyright</i>	文字列	©
<i>ISO L1 e acute</i>	文字列	é
<i>ISO L1 e circumflex</i>	文字列	ê
<i>ISO L1 e grave</i>	文字列	è
<i>ISO L1 e umlaut</i>	文字列	ë
<i>ISO L1 eth Icelandic</i>	文字列	ð
<i>ISO L1 Greater than</i>	文字列	>
<i>ISO L1 i acute</i>	文字列	í
<i>ISO L1 i circumflex</i>	文字列	î
<i>ISO L1 i grave</i>	文字列	ì
<i>ISO L1 i umlaut</i>	文字列	ï

定数	タイプ	値
<i>ISO L1 Less than</i>	文字列	<
<i>ISO L1 n tilde</i>	文字列	ñ
<i>ISO L1 o acute</i>	文字列	ó
<i>ISO L1 o circumflex</i>	文字列	ô
<i>ISO L1 o grave</i>	文字列	ò
<i>ISO L1 o slash</i>	文字列	ø
<i>ISO L1 o tilde</i>	文字列	õ
<i>ISO L1 o umlaut</i>	文字列	ö
<i>ISO L1 Quotation mark</i>	文字列	"
<i>ISO L1 Registered</i>	文字列	®
<i>ISO L1 sharp s German</i>	文字列	ß
<i>ISO L1 thorn Icelandic</i>	文字列	þ
<i>ISO L1 u acute</i>	文字列	ú
<i>ISO L1 u circumflex</i>	文字列	û
<i>ISO L1 u grave</i>	文字列	ù
<i>ISO L1 u umlaut</i>	文字列	ü
<i>ISO L1 y acute</i>	文字列	ý
<i>ISO L1 y umlaut</i>	文字列	ÿ
<i>Italic</i>	倍長整数	2
<i>January</i>	倍長整数	1
<i>July</i>	倍長整数	7
<i>June</i>	倍長整数	6
<i>Key down event</i>	倍長整数	3
<i>Key up event</i>	倍長整数	4
<i>Left Arrow Key</i>	倍長整数	28
<i>LF ASCII code</i>	倍長整数	10
<i>Light Blue</i>	倍長整数	7
<i>Light Grey</i>	倍長整数	12
<i>Line feed</i>	倍長整数	10
<i>Load Record Event</i>	倍長整数	4
<i>Locked resource bit</i>	倍長整数	4
<i>Locked resource mask</i>	倍長整数	16
<i>Long</i>	倍長整数	3
<i>LongInt array</i>	倍長整数	16
<i>Macintosh 68K</i>	倍長整数	1
<i>Macintosh byte ordering</i>	倍長整数	1
<i>Macintosh double real format</i>	倍長整数	2

定数	タイプ	値
<i>Macintosh interface</i>	倍長整数	0
<i>Macintosh node</i>	倍長整数	860
<i>Mac OS Printer Port</i>	倍長整数	0
<i>Mac OS Serial Port</i>	倍長整数	1
<i>March</i>	倍長整数	3
<i>MAXINT</i>	倍長整数	32767
<i>MAXLONG</i>	倍長整数	2147483647
<i>MAXTEXTLEN</i>	倍長整数	32000
<i>May</i>	倍長整数	5
<i>Millions of colors 24 bit</i>	倍長整数	24
<i>Millions of colors 32 bit</i>	倍長整数	32
<i>MM DD YYYY</i>	倍長整数	4
<i>MM DD YYYY Forced</i>	倍長整数	7
<i>Modal dialog</i>	倍長整数	9
<i>Modal dialog box</i>	倍長整数	1
<i>Monday</i>	倍長整数	2
<i>Month Date Year</i>	倍長整数	5
<i>Mouse button bit</i>	倍長整数	7
<i>Mouse button mask</i>	倍長整数	128
<i>Mouse down event</i>	倍長整数	1
<i>Mouse up event</i>	倍長整数	2
<i>Movable dialog box</i>	倍長整数	5
<i>NAK ASCII code</i>	倍長整数	21
<i>Native byte ordering</i>	倍長整数	0
<i>Native real format</i>	倍長整数	0
<i>NBSP</i>	倍長整数	202
<i>No current record</i>	倍長整数	-1
<i>No such data in clipboard</i>	倍長整数	-102
<i>November</i>	倍長整数	11
<i>NUL ASCII code</i>	倍長整数	0
<i>Null event</i>	倍長整数	0
<i>October</i>	倍長整数	10
<i>On Activate</i>	倍長整数	11
<i>On Background</i>	倍長整数	3
<i>On Clicked</i>	倍長整数	4
<i>On Close Box</i>	倍長整数	22
<i>On Close Detail</i>	倍長整数	26

定数	タイプ	値
<i>On Data Change</i>	倍長整数	20
<i>On Deactivate</i>	倍長整数	12
<i>On Display Detail</i>	倍長整数	8
<i>On Double Clicked</i>	倍長整数	13
<i>On Drag Over</i>	倍長整数	21
<i>On Drop</i>	倍長整数	16
<i>On External Area</i>	倍長整数	19
<i>On Getting Focus</i>	倍長整数	15
<i>On Keystroke</i>	倍長整数	17
<i>On Load</i>	倍長整数	1
<i>On Losing Focus</i>	倍長整数	14
<i>On Menu Selected</i>	倍長整数	18
<i>On Open Detail</i>	倍長整数	25
<i>On Outside Call</i>	倍長整数	10
<i>On Printing Break</i>	倍長整数	6
<i>On Printing Detail</i>	倍長整数	23
<i>On Printing Footer</i>	倍長整数	7
<i>On Printing Header</i>	倍長整数	5
<i>On Unload</i>	倍長整数	24
<i>On Validate</i>	倍長整数	3
<i>Operating system event</i>	倍長整数	15
<i>Option key bit</i>	倍長整数	11
<i>Option key mask</i>	倍長整数	2048
<i>Orange</i>	倍長整数	2
<i>Outline</i>	倍長整数	8
<i>Page Down Key</i>	倍長整数	12
<i>Page Up Key</i>	倍長整数	11
<i>Palette window</i>	倍長整数	720
<i>Parity Even</i>	倍長整数	12288
<i>Parity None</i>	倍長整数	0
<i>Parity Odd</i>	倍長整数	4096
<i>Pascal string</i>	倍長整数	1
<i>Paused</i>	倍長整数	5
<i>PC byte ordering</i>	倍長整数	2
<i>PC double real format</i>	倍長整数	3
<i>Pentium</i>	倍長整数	586
<i>Period</i>	倍長整数	46

定数	タイプ	値
<i>Pi</i>	実数	3.141592653589793239
<i>Picture array</i>	倍長整数	19
<i>Picture data</i>	文字列	PICT
<i>Picture Document</i>	文字列	PICT
<i>Plain</i>	倍長整数	0
<i>Plain dialog box</i>	倍長整数	2
<i>Plain fixed size window</i>	倍長整数	4
<i>Plain no zoom box window</i>	倍長整数	0
<i>Plain window</i>	倍長整数	8
<i>Plug-in window</i>	倍長整数	5
<i>Pointer array</i>	倍長整数	20
<i>Power Macintosh</i>	倍長整数	2
<i>PowerPC 601</i>	倍長整数	601
<i>PowerPC 603</i>	倍長整数	603
<i>PowerPC 604</i>	倍長整数	604
<i>Preloaded resource bit</i>	倍長整数	2
<i>Preloaded resource mask</i>	倍長整数	4
<i>Protected resource bit</i>	倍長整数	3
<i>Protected resource mask</i>	倍長整数	8
<i>Protocol DTR</i>	倍長整数	30
<i>Protocol None</i>	倍長整数	0
<i>Protocol XONXOFF</i>	倍長整数	20
<i>Purgeable resource bit</i>	倍長整数	5
<i>Purgeable resource mask</i>	倍長整数	32
<i>Purple</i>	倍長整数	4
<i>QT Animation compressor</i>	文字列	rle
<i>QT Compact video compressor</i>	文字列	cdvc
<i>QT Graphics compressor</i>	文字列	smc
<i>QT Photo compressor</i>	文字列	jpeg
<i>QT Raw compressor</i>	文字列	raw
<i>QT Video compressor</i>	文字列	rpza
<i>Quote</i>	倍長整数	39
<i>Radian</i>	実数	57.29577951308232088
<i>Real array</i>	倍長整数	14
<i>Red</i>	倍長整数	3
<i>Regular window</i>	倍長整数	8

定数	タイプ	値
<i>Return Key</i>	倍長整数	13
<i>Right Arrow Key</i>	倍長整数	29
<i>Right control key bit</i>	倍長整数	15
<i>Right control key mask</i>	倍長整数	32768
<i>Right option key bit</i>	倍長整数	14
<i>Right option key mask</i>	倍長整数	16384
<i>Right shift key bit</i>	倍長整数	13
<i>Right shift key mask</i>	倍長整数	8192
<i>Round corner window</i>	倍長整数	16
<i>RS ASCII code</i>	倍長整数	30
<i>Saturday</i>	倍長整数	7
<i>Save Existing Record Event</i>	倍長整数	2
<i>Save New Record Event</i>	倍長整数	1
<i>Scaled to Fit</i>	倍長整数	2
<i>Scaled to fit prop centered</i>	倍長整数	6
<i>Scaled to fit proportional</i>	倍長整数	5
<i>September</i>	倍長整数	9
<i>Shadow</i>	倍長整数	16
<i>Shift key bit</i>	倍長整数	9
<i>Shift key mask</i>	倍長整数	512
<i>Short</i>	倍長整数	1
<i>SI ASCII code</i>	倍長整数	15
<i>Sixteen colors</i>	倍長整数	4
<i>SO ASCII code</i>	倍長整数	14
<i>SOH ASCII code</i>	倍長整数	1
<i>SP ASCII code</i>	倍長整数	32
<i>Space</i>	倍長整数	32
<i>Speed 115200</i>	倍長整数	1022
<i>Speed 1200</i>	倍長整数	94
<i>Speed 1800</i>	倍長整数	62
<i>Speed 19200</i>	倍長整数	4
<i>Speed 230400</i>	倍長整数	1021
<i>Speed 2400</i>	倍長整数	46
<i>Speed 300</i>	倍長整数	380
<i>Speed 3600</i>	倍長整数	30
<i>Speed 4800</i>	倍長整数	22
<i>Speed 57600</i>	倍長整数	0

定数	タイプ	値
<i>Speed 600</i>	倍長整数	189
<i>Speed 7200</i>	倍長整数	14
<i>Speed 9600</i>	倍長整数	10
<i>Stop bits One</i>	倍長整数	16384
<i>Stop bits One and a half</i>	倍長整数	-32768
<i>Stop bits Two</i>	倍長整数	-16384
<i>String array</i>	倍長整数	21
<i>STX ASCII code</i>	倍長整数	2
<i>SUB ASCII code</i>	倍長整数	26
<i>Sunday</i>	倍長整数	1
<i>SYN ASCII code</i>	倍長整数	22
<i>System heap resource bit</i>	倍長整数	6
<i>System heap resource mask</i>	倍長整数	64
<i>Tab</i>	倍長整数	9
<i>Tab Key</i>	倍長整数	9
<i>TCP Authentication</i>	倍長整数	113
<i>TCP DNS</i>	倍長整数	53
<i>TCP Finger</i>	倍長整数	79
<i>TCP FTP Control</i>	倍長整数	21
<i>TCP FTP Data</i>	倍長整数	20
<i>TCP Gopher</i>	倍長整数	70
<i>TCP HTTP WWW</i>	倍長整数	80
<i>TCP IMAP3</i>	倍長整数	220
<i>TCP Kerberos</i>	倍長整数	88
<i>TCP KLogin</i>	倍長整数	543
<i>TCP Nickname</i>	倍長整数	43
<i>TCP NNTP</i>	倍長整数	119
<i>TCP NTalk</i>	倍長整数	518
<i>TCP NTP</i>	倍長整数	123
<i>TCP PMCP</i>	倍長整数	1643
<i>TCP PMD</i>	倍長整数	1642
<i>TCP POP3</i>	倍長整数	110
<i>TCP Printer</i>	倍長整数	515
<i>TCP RADACCT</i>	倍長整数	1646
<i>TCP RADIUS</i>	倍長整数	1645
<i>TCP Remote Cmd</i>	倍長整数	514
<i>TCP Remote Exec</i>	倍長整数	512

定数	タイプ	値
<i>TCP Remote Login</i>	倍長整数	513
<i>TCP Router</i>	倍長整数	520
<i>TCP SMTP</i>	倍長整数	25
<i>TCP SNMP</i>	倍長整数	161
<i>TCP SNMPTRAP</i>	倍長整数	162
<i>TCP SUN RPC</i>	倍長整数	111
<i>TCP Talk</i>	倍長整数	517
<i>TCP Telnet</i>	倍長整数	23
<i>TCP TFTP</i>	倍長整数	69
<i>TCP UUCP</i>	倍長整数	540
<i>TCP UUCP RLOGIN</i>	倍長整数	541
<i>Text array</i>	倍長整数	18
<i>Text data</i>	文字列	TEXT
<i>Text Document</i>	文字列	TEXT
<i>Text with length</i>	倍長整数	2
<i>Text without length</i>	倍長整数	3
<i>Thousands of colors</i>	倍長整数	16
<i>Thursday</i>	倍長整数	5
<i>Truncated Centered</i>	倍長整数	1
<i>Truncated non Centered</i>	倍長整数	4
<i>Tuesday</i>	倍長整数	3
<i>Two fifty six colors</i>	倍長整数	8
<i>Underline</i>	倍長整数	4
<i>Up Arrow Key</i>	倍長整数	30
<i>Update event</i>	倍長整数	6
<i>US ASCII code</i>	倍長整数	31
<i>Use PicRef</i>	倍長整数	131072
<i>Use PICT resource</i>	倍長整数	65536
<i>VT ASCII code</i>	倍長整数	11
<i>Waiting for input output</i>	倍長整数	3
<i>Waiting for internal flag</i>	倍長整数	4
<i>Waiting for user event</i>	倍長整数	2
<i>Wednesday</i>	倍長整数	4
<i>White</i>	倍長整数	0
<i>Windows</i>	倍長整数	3
<i>Windows 3.1 interface</i>	倍長整数	1
<i>Windows 95 interface</i>	倍長整数	2

定数	タイプ	値
<i>Windows MIDI Document</i>	文字列	MID
<i>Windows node</i>	倍長整数	138
<i>Windows Sound Document</i>	文字列	WAV
<i>Windows Video Document</i>	文字列	AVI
<i>Yellow</i>	倍長整数	1 4.

6

コマンドの説明

4D Compiler は、4th Dimension の通常のコマンドシンタックスに基づいてコマンド処理を行います。この点に関して、コンパイルのためにデータベースを特に変更する必要はありません。

変数のデータタイプを決定付けるようなコマンドは、データタイプ矛盾の原因になることがあります。また、コマンドの中には複数のシンタックスを持つものがあり、どのシンタックスが最適なのか知っておくと役に立ちます。この章では、こうしたコマンドについて項目別に説明します。

配列

4D Compiler が配列のデータタイプを決める際に使う 4th Dimension コマンドは、以下の 7 種類です。

COPY ARRAY (*from; to*)
SELECTION TO ARRAY (*field; array*)
LIST TO ARRAY (*list; array; {linked array}*)
ARRAY TO LIST (*array; list; {linked array}*)
DISTINCT VALUES (*field; array*)
ARRAY TO SELECTION (*array; field*)
SUBSELECTION TO ARRAY (*start; end; field; array*)

COPY ARRAY

COPY ARRAY コマンドは 2 個の配列タイプの引数を使います。引数の一方がどこにも定義されていないと、4D Compiler は定義されている方のデータタイプから未定義の配列のデータタイプを決定します。

- 1 番目の引数が別のところで定義されている場合。2 番目の配列には、最初の配列のデータタイプが適用されます。
- 2 番目の引数が定義されている場合。1 番目の配列には、2 番目の配列のデータタイプが適用されます。

4D Compiler はデータタイプを厳密にチェックするので、**COPY ARRAY** コマンドは同じタイプの配列間で行わなければなりません。

そのため、整数と倍長整数と実数、あるいは、テキスト配列と文字配列で文字列の長さがまちまちな場合、などのようにタイプの似ている配列間のコピーをする際には、要素を 1 つずつコピーする必要があります。

- ▼ たとえば、整数配列から実数配列に要素をコピーする場合は、次のように処理します。

```

$Size:=Size of array (ArrInt)
ARRAY REAL (ArrReal;$size)      `実数配列と整数配列のサイズを同じにする。
For ($i;1;$Size)
  ArrReal{$i}:=ArrInt{$i}        `要素を1つ1つコピーする。
End for

```

注 処理中に配列の次元数を変更できないことに注意してください。1次元の配列を2次元の配列にコピーすると、4D Compiler はエラーメッセージを出力します。

SELECTION TO ARRAY, ARRAY TO SELECTION, DISTINCT VALUES, SUBSELECTION TO ARRAY

タイプが定義されていない配列のデフォルトタイプは、SELECTION TO ARRAY コマンドで指定したフィールドのデータタイプになります。

以下のように記述すると、

```
SELECTION TO ARRAY ([MyTable]IntField;MyArray)
```

“ MyArrey ” は整数配列になります (“ IntField ” が整数フィールドのとき)

配列を定義しておく場合は、フィールドと同じデータタイプにするよう注意してください。整数、倍長整数、実数は似ていますが、同じタイプではありません。

文字タイプのフィールドを使用するコマンドで、配列をあらかじめ定義せずに引数として使うと、配列のデータタイプはテキストに定義されます。

あらかじめ配列を文字またはテキストタイプとして宣言してある場合は、その指定が適用されず。テキストタイプのフィールドについても同様で、宣言されたタイプが優先されます。

SELECTION TO ARRAY コマンドは、1次元の配列でしか使用できません。SELECTION TO ARRAY コマンドにはもう1つのシンタックスがあります。

SELECTION TO ARRAY (Table; Array)

このケースでは、変数配列は倍長整数の配列になります。SUBSELECTION TO ARRAY コマンドも同様な働きをします。

LIST TO ARRAY, ARRAY TO LIST

LIST TO ARRAY コマンドで引数として使用できるのは、1次元の文字配列と1次元のテキスト配列の2種類だけです。

このコマンドの場合、引数に使う配列をあらかじめ宣言する必要はありません。デフォルトでは、宣言されていない配列は、テキスト配列になります。

配列に関連したコマンドでのポインタの使用

ポインタのセクションで説明したように、配列を定義するコマンドの引数にポインタ参照が使われていると、4D Compiler にはタイプの矛盾を発見できません。

Pointer-> が配列を表示するところで、下記のように記述した場合、

SELECTION TO ARRAY ([Table]Field;Pointer->)

「Pointer->」が配列だとすると、フィールドと配列のタイプが同じかどうかチェックできません。フィールドと配列のタイプの矛盾が起こらないようプログラマが気をつけるべきです。ポインタで参照する配列は、必ずタイプを定義してください。

4D Compiler は、引数にポインタを使用している配列定義ステートメントを見つけると、警告メッセージを出力します。警告メッセージはこの種の矛盾を見つける際に役立ちます。

ローカル配列

データベースで、ローカル配列（定義されたメソッド内のみで有効な配列）を使用している場合は、使用前に明確に宣言しておく必要があります。ローカル配列を定義するには、**ARRAY REAL**、**ARRAY INTEGER** など、配列を定義するコマンドを使います。

- ▼ たとえば、プロシージャで 10 個の要素を持つローカルな整数配列を作る場合、次のようなコマンドを使って、使用前に定義しておきます。

ARRAY INTEGER (\$myarray;10)

注 配列の定義に関する詳細は『4th Dimension ランゲージリファレンス』を参照してください。

コミュニケーション

SEND VARIABLE (変数)

RECEIVE VARIABLE (変数)

上の 2 つのコマンドは変数をディスクに、保存または読み込む場合に使われます。引数は変数です。

コマンドから受け取る変数のタイプは、常に引き渡したときと同じタイプでなくてはなりません。

変数のリストをファイルに送るときには、誤ってデータタイプを変えてしまう恐れがあるので、リストの先頭に送る変数のデータタイプを指定することをおすすめします。

変数を受け取る際には、常にタイプが返されることとなります。**RECEIVE VARIABLE** コマンドをコールした後、**Case of** 文を使って、次から受け取るデータを処理できます。

▼ 例

```

`Send variable example
SET CHANNEL (12;"Document1")
$Type:=Type ([Client]Total)
SEND VARIABLE ($Type)
For (i;1;Records in selection([Client]))
    $Total:=[Client]Total
    SEND VARIABLE ($Total)
End for
SET CHANNEL (11)
`変数を受け取る例
SET CHANNEL (12;"Document1")
RECEIVE VARIABLE ($Type)
Case of
:($Type=0)
    RECEIVE VARIABLE ($String) `` プロセス中の変数を受領した。
:($Type=1)
    RECEIVE VARIABLE ($Real) `` プロセス中の変数を受領した。
:($Type=2)
    RECEIVE VARIABLE ($Text) `` プロセス中の変数を受領した。
End case
SET CHANNEL (11)

```

注 Type 関数から返される値に関しては、『4th Dimension ランゲージリファレンス』を参照してください。

データエンタリー

Type (引数)

コンパイルしたデータベースの変数はデータタイプが1つに決まっているので、この関数は無意味なようですが、ポインタを使っている場合には便利です。たとえば、ポインタで参照している変数のデータタイプを調べるようなこともあります。ポインタは参照先を変更できるので、どのオブジェクトを示しているのかユーザが常に把握しているとは限らないからです。

例外

```

ON EVENT CALL( イベントメソッド {; プロセス名} )
ON SERIAL PORT CALL ( シリアルメソッド )
ABORT
IDLE

```

例外処理を扱うために **IDLE** というコマンドがあります。ON EVENT CALL コマンドや ON SERIAL PORT CALL コマンドを使う場合は、必ず **IDLE** コマンドを使ってください。このコマンドはイベント管理命令を行います。

Macintosh のイベント（マウスクリックやキー操作など）は、4th Dimension のカーネルにしか検知できません。ほとんどの場合、カーネルコールはコンパイル後のコードそのものにより、ユーザに対してトランスペアレント（透過的）な形で起動されます。

- ▼ 一例を示します。

```

`MouseClicked Method
If (MouseDown=1)
  ◊vTest:=True
  MESSAGE (" マウスがクリックされました ")
End if

`Wait メソッド
◊vTest:=False
ON EVENT CALL ("MouseClicked")
While (Not(◊vTest))          ` イベントの待ちループ
  ...
  `カーネルコールのない式
End while
ON EVENT CALL ("")

```

次のように IDLE コマンドを追加してください。

```

`Wait メソッド
◊vTest:=False
ON EVENT CALL (MouseClicked)
While (Not(◊vTest))
  IDLE ` イベントを検知するカーネルコール
End while
ON EVENT CALL ("")

```

- 注 **ON SERIAL PORT CALL** コマンドはバージョン 6 の 4th Dimension に存在していますが、これは旧バージョンで作成されたデータベースとの互換性を保持するためです。バージョン 6 からは、別プロセスを設けることにより、同じ機能を実現できます。

ABORT

このコマンドは、エラー処理プロジェクトメソッド内でのみ使用して下さい。これは 4th Dimension で使用した場合とまったく同じように動作しますが、**EXECUTE**、**APPLY TO SELECTION**、**APPLY TO SUBSELECTION** コマンドから呼び出されたメソッド内の場合は例外です。このような状況は避けた方がよいでしょう。

文書

Open Document
Create Document
Append Document

これらの関数が返す文書ファイル参照番号のデータタイプは時間タイプです。

演算

Mod (*value;divider*)

4th Dimension では、25 を 3 で割った余りを求める場合、次の 2 通りの方法があります。

Variable:=Mod (25;3)

あるいは

Variable:=25%3

4D Compiler はこの 2 つの式を区別します。Mod 関数はすべての数値に使用できますが、% 演算子は整数と倍長整数にしか使用できません。演算子 % の演算数が、倍長整数データタイプの範囲を越えた場合には、返される結果が誤りである場合が多くなります。

ブ레이크処理

Subtotal (*data*)

コンパイルしたデータベースでは、Subtotal 関数はブ레이크処理を起こしません。ブ레이크処理を起こすためには **BREAK LEVEL** コマンドを使い、集計対象の指定には **ACCUMULATE** コマンドを使ってください。

文字列

Ascii (文字)

インタプリタでは、Ascii 関数に渡す文字列が空でもデータが入っていても構いませんが、コンパイル後は空の文字列を渡すことができません。Ascii 関数の引数に変数の場合に空の文字列を渡していても、コンパイル中にエラーを見つけることはできません。

ストラクチャへのアクセス

Field (フィールドポインタ) または Field (テーブル番号; フィールド番号)
Table (フィールドポインタ) または Table (テーブル番号) または Table (フィールドポインタ)

2 つのコマンドは、与えられた引数によって、返す値のデータタイプが異なります。

- ポインタを与えると、Table 関数は数値を返します。
- 数値を与えると、Table 関数はポインタを返します。

コンパイラでは、これらの関数から結果のデータタイプを決定できません。このような場合は、コンパイラ命令を使用して明確に定義してください。

変数その他

Undefined (variable)
SAVE VARIABLE (document; variable1; {...; variableN})
LOAD VARIABLE (document; variable1; {...; variableN})
CLEAR VARIABLE (variable)
Get Pointer (variable)
EXECUTE (formula)
TRACE
NO TRACE

Undefined コマンド

4D Compiler では変数が未定義ということはありません。コンパイルしたデータベースでは、On Startup データベースメソッドの実行前に、変数はすべてヌルに初期化されます。そのため、Undefined 関数は常に False (偽) を返します。4D Compiler は Undefined 関数を見つけると警告メッセージを出力します。

注 アプリケーションがコンパイルモードで実行されているかどうかを知るには、Compiled application コマンドを呼び出して下さい。

SAVE VARIABLE コマンドと LOAD VARIABLE コマンド

インタプリタでは、LOAD VARIABLE コマンドの実行後に Undefined 関数を使って変数が未定義かどうか調べることにより、文書ファイルの存在の有無をチェックできます。コンパイル後はこの方法を使えません。Undefined 関数が常に False (偽) を返すからです。

このテストはインタプリタでもコンパイル後でも次のようにして実行できます。

- どの変数にとっても無効な値で、ロードする変数を初期化する。
- LOAD VARIABLE コマンドを実行した後、ロードした変数のうちの 1 つを初期値と比較する。

メソッドは以下ようになります。

```

Var1:="xxxxxx"           "xxxxxx" は LOAD VARIABLE によって
Var2:="xxxxxx"           `返されない値
Var3:="xxxxxx"
Var4:="xxxxxx"
LOAD VARIABLE ("Document";Var1;Var2;Var3;Var4)
If (Var1="xxxxxx")       `ドキュメントが見つからなかった
`...
Else `ドキュメントが見つかった
End if

```

CLEAR VARIABLE コマンド

インタプリタでは、CLEAR VARIABLE コマンドには次の 2 つのシンタックスがあります。

CLEAR VARIABLE (変数)
CLEAR VARIABLE ("a")

コンパイル後のデータベースでは、1 番目のシンタックス CLEAR VARIABLE (変数) は変数を再度初期化します。(数値は 0 に、文字列やテキストは空にする等) なぜならば、コンパイル後は未定義の変数は存在しないからです。従って、コン

パイル後は、テキスト、ピクチャ、BLOB、配列タイプの変数を除き、CLEAR VARIABLE コマンドで変数のメモリを解放することはできません。

配列の場合、CLEAR VARIABLE コマンドは、要素数が 0 の新しい配列の定義を意味します。

整数配列なら、CLEAR VARIABLE (配列) は以下の 2 つのいずれかの式と同じ結果になります。

ARRAY INTEGER (Array;0) `1 次元配列である場合
ARRAY INTEGER (Array;0;0) `2 次元配列である場合

2 番目のシンタックス CLEAR VARIABLE ("a") は、コンパイラでは使えません。なぜならコンパイラは名前ではなくアドレスで変数にアクセスするからです。

Get Pointer コマンド

Get Pointer 関数は、与えられた引数のポインタを返す関数です。

ポインタの配列を初期化するとき、配列の各要素はそれぞれ与えられた変数を示します。そのような変数が V1、V2、...V12 だとすると、以下のように書くことができます。

```

ARRAY POINTER (Arr;12)
Arr{1}:=>V1
Arr{2}:=>V2
.
.
Arr{12}:=>V12

```

次のように書くこともできます。:

```

ARRAY POINTER (Arr;12)
For ($i;1;12)
    Arr{$i}:=Get Pointer ("V"+String ($i))
End for

```

この処理が終了すると、各要素が変数 Vi を指すポインタの配列ができます。

この 2 つの書き方は、両方ともコンパイルできますが、他の場所で変数 V1...V12 のタイプが明らかにされていないと、コンパイラはデータタイプを決定できません。そのため、このような変数は別の場所で明示的に使用するか、または定義する必要があります。

明示的に変数を定義する方法は、2 通りあります。

- コンパイラ命令を使って V1...V12 を定義する。

```

C_LONGINT (V1;V2;V3...V12)

```

- メソッドで V1...V12 に値を代入する。

```

V1:=0
V2:=0
.
.
V12:=0

```

EXECUTE コマンド

EXECUTE コマンドは、インタプリタでは有効ですが、コンパイル後にはその利点を活かすことができません。

コンパイル後は、引数として EXECUTE コマンドに渡された時点でメソッド名が解釈されるため、4D Compiler の利点を活用できない上に、引数のシンタックスチェックもできません。また、引数にローカル変数を使用することもできません。

EXECUTE は、複数のステートメントに置き換えることができます。

▼ 例を示します。

```
$Num:= 印刷番号  
EXECUTE ("Print"+String($Num))
```

これは次のように書き換えることができます。

```
Case of  
  ¥($Num=1)  
    Print1  
  ¥($Num=2)  
    Print2  
  ¥($Num=3)  
    Print3  
  ,...  
End case
```

EXECUTE コマンドは必ず置き換え可能です。実行するメソッドはデータベースのプロジェクトメソッドのリストから選ばれたもので、その数は有限です。ですから、EXECUTE コマンドは必ず Case of 文や他のコマンドで置き換えられます。さらに、コードの実行速度は EXECUTE コマンドよりも速くなります。

Trace コマンドと No Trace コマンド

この2つのコマンドはデバッグの段階で使用します。コンパイル後のデータベースでは機能しません。4D Compiler はこれらのコマンドを無視するので、メソッド中に残しておいても問題ありません。

各種のコマンドで使われるポインタ

下記のコマンドは注意して使用して下さい。

PRINT FORM	CREATE EMPTY SET
PRINT LABEL	QUERY
DIALOG	QUERY BY FORMULA
INPUT FORM	QUERY SELECTION BY FORMULA
OUTPUT FORM	QUERY SELECTION
APPLY TO SELECTION	COPY NAMED SELECTION
EXPORT DIF	CUT NAMED SELECTION
EXPORT SYLK	REDUCE SELECTION
EXPORT TEXT	ORDER BY
IMPORT DIF	ORDER BY FORMULA
IMPORT SYLK	LOCKED ATTRIBUTES
IMPORT TEXT	GOTO RECORD
CREATE SET	GOTO SELECTED RECORD
ADD TO SET	PAGE SETUP
REMOVE FROM SET	REPORT
LOAD SET	GRAPH TABLE

これらのコマンドには、共通の特徴が1つあります。これらは最初のテーブルパラメータを省略し、2番目のパラメータをポインタにすることができます。

コンパイルモードでは、テーブルパラメータを省略することは簡単ですが、これらコマンドの1つに渡された最初のパラメータがポインタである場合には、コンパイラはポインタが何を参照しているのかを知ることができません。その結果、コンパイラはそれをテーブルポインタとして扱ってしまいます。

- ▼ 次のメソッドを考えてみましょう。メソッドで使われているさまざまなオブジェクトが存在していれば、このメソッドはインタプリタのもとではエラーを起こさずに実行できます。しかし、4D Compilerはこのメソッドをコンパイルできません。

DEFAULT TABLE([aTable])

　`セットに関するコマンド

V:="Set name"

P:=->V

ADD TO SET(P->)

CREATE EMPTY SET(P->)

CREATE SET(P->)

LOAD SET(P->;"MyDocument")

　`フォームに関するコマンド

V:="Form name"

P:=->V

INPUT FORM(P->)

OUTPUT FORM(P->)

DIALOG(P->)

　`命名セレクションに関するコマンド

V:="Selection name"
P:=->V
COPY NAMED SELECTION(P->)
CUT NAMED SELECTION(P->)
` 書出しと読みに関するコマンド
V:="Document name"
P:=->V
EXPORT DIF(P->)
EXPORT SYLK(P->)
EXPORT TEXT(P->)
IMPORT DIF(P->)
IMPORT SYLK(P->)
IMPORT TEXT(P->)
` 検索とソートに関するコマンド
P:=->[aTable]aField
QUERY(P->#"")
QUERY BY FORMULA(P->#"")
QUERY SELECTION(P->#"")
QUERY SELECTION BY FORMULA(P->#"")
ORDER BY(P->;>)
` 印刷に関するコマンド
V:="Document name"
P:=->V
PRINT LABEL(P->)
REPORT(P->)
V:="Form name"
P:=->V
PAGE SETUP(P->)
PRINT FORM(P->)
` セレクションに関するコマンド
P:=->[aTable]aField
APPLY TO SELECTION(P->="")
N:=1
P:=->N
GRAPH TABLE(P->;[aTable]aField;[aTable]AnotherField)
` レコードに関するコマンド
N:=1
P:=->N
GOTO RECORD(P->)
GOTO SELECTED RECORD(P->)

7

最適化のためのヒント

“ 良いプログラムをつくる ” ための決定的な方法を説明するのは難しいことですが、良い構造を持つプログラムの利点を再度ここで強調します。4th Dimension は構造化プログラミングが可能であり、この能力はプログラミングの大きな助けになります。

構造化されたデータベースとそうでないデータベースとでは、コンパイルに費やす労力は同じでも結果は大きく違ってきます。たとえば、n 個のオブジェクトに対する共通メソッドは、同じステートメントで書かれた n 個のオブジェクトメソッドより、インタプリタであれ、コンパイル後であれ、はるかに良い結果をもたらすことでしょう。つまり、プログラミングの質がコンパイル後のコードの品質にも影響を与えるのです。

4th Dimension で実行することにより、コードを段階的に改善します。4D Compiler を頻繁に使うことにより、間違いを訂正するフィードバックを得て、最も効果的な解決方法に到達できます。

この章では、単純な繰り返しの作業にかかる時間を短縮するためのアドバイスや秘訣を紹介します。

コードに関するコメント

プログラミングテクニックによっては、コードが他の人にとって理解しづらいものもあります。また、自分で修正する場合でも時間が経つと分からなくなることもあります。従って、メソッドには、ふんだんにコメントを入れてください。コメントが多すぎるとインタプリタデータベースでは実行が遅くなりますが、コンパイル後のデータベースにはまったく影響しません。

コンパイラ命令の使用によるコードの最適化

コンパイラ命令を使うと、コードの実行速度がかなり速くなります。用途から変数のタイプを決定する場合、一番広い範囲をカバーできるタイプを設定します。たとえば、変数のタイプを “Var:=5” というステートメントで定義する場合、整数しか使用していないくても、コンパイラはタイプを実数に設定します。

変数のタイプを、整数や倍長整数、文字に限定できるときは必ずコンパイラ命令で定義してください。

注 こうした最適化は、デフォルトのタイプを選択してプロジェクトの面から設定することもできます。しかし、各データベースのコンパイルで同じコンパイルオプションを使うようにするべきです。また、コンパイル命令はデータベース内で常に使用できます。

数値変数

コンパイルプロジェクトが他のものに設定されていないければ、コンパイラ命令でタイプ設定していない数値変数にコンパイラが割り当てるデータタイプは実数です。しかし、実数の計算は倍長整数の計算より遅いので、数値変数の値として整数しか使わない場合には、コンパイラ命令 C_INTEGER か C_LONGINT で変数を定義すると効果的です。

ループのカウンタは常に整数として定義しておくといでしょう。次の 2 つの空のループの例の実行時間を比べてみてください。

```
For ($i;1;50000)  
End for
```

\$i がコンパイラ・ディレクティブ (例えば、\$i is real) で宣言されると、時間は 19 秒です。\$i がコンパイラ・ディレクティブ (C_INTEGER) で宣言されると、これは瞬時ループです。

注 実行速度は、使用マシンによって異なります。

4th Dimension の関数の中には整数を返すものがあります (Ascii 関数など)。4D Compiler は、このような関数の結果を未定義の変数に代入する場合も、変数のタイプを整数ではなく実数にします。変数が別のタイプの値に使われないことが明らかな場合は、必ずコンパイラ命令で変数を宣言してください。

▼ ここで簡単な例を示します。指定された範囲内のランダムな数を返す関数です。

```
$0:= Random % ($2-$1+1)+$1
```

このように記述されていると、4D Compiler は \$0 のタイプを整数や倍長整数ではなく実数に設定してしまうので、プロシージャにコンパイラ命令を使用してください

```
C_LONGINT ($0)  
$0:= Random % ($2-$1+1)+$1
```

メソッドの戻り値に使用するメモリスペースも少なく、プロシージャの実行速度も速くなります。

- ▼ もう一つの例を示します。2つの変数を倍長整数として定義します。

```
C_LONGINT($var1;$var2)
```

そして、計算の結果が3つ目のタイプの定義されていない変数に返されます。

```
$var3:=$var1+$var2
```

4D Compiler は、3つ目の変数を整数とします。結果を倍長整数としたいのであれば、倍長整数として明確に定義しなければなりません。

注 コンパイルモードでの計算結果は、計算結果を受け取る変数のタイプではなく、計算時のデータタイプである事に注意して下さい。

- ▼ 下記の例では、計算は倍長整数として計算しています。

```
C_REAL($var3)
```

```
C_LONGINT($var1;$var2)
```

```
$var1:=2147483647
```

```
$var2:=1
```

```
$var3:=$var1+$var2
```

コンパイルモード及びインタプリタモードの両方で、\$var3 は -2147483647 となります。

- ▼ しかし、次の例では、

```
C_REAL($var3)
```

```
C_LONGINT($var1)
```

```
$var1:=2147483647
```

```
$var3:=$var1+1
```

4D Compiler は、\$var1 を倍長整数とみなします。

コンパイルモードでは、計算は倍長整数としている為、\$var3 は -2147483647 となります。インタプリタモードでは、計算が実数としている為、\$var3 は 2147483648 となります。

ボタンは実数ですが、倍長整数に定義できる具体的なケースでもあります。

文字

コンパイルプロジェクトで特別に指定しない場合、文字変数のデフォルトのタイプとしてテキストが割り当てられます。次のように書くと、

```
MyString:=" こんにちは "
```

コンパイラは "MyString" のタイプをテキストとみなします。

この変数をたびたび使用するなら、**C_STRING** コマンドで定義した方がよいでしょう。テキスト変数を処理するより、長さが決まっている文字タイプの変数を処理する方がはるかに速いからです。コンパイラ命令の動きを決めるこのルールを覚えておいてください。

注 文字の値を比較したい場合、文字自身についてというよりも、その ASCII 値としての比較をして下さい。通常文字比較では、分音符などの全ての文字を考慮します。

その他のヒント

このセクションは、2次元配列、フィールド、およびポインタについてのヒントです。

2次元配列

2次元配列は、2番目の次元が1番目の次元より大きい方がうまく処理できます。例えば、次のように定義された配列は、

```
ARRAY INTEGER (Array;5;1000)
```

以下のような配列よりもうまく処理されます。

```
ARRAY INTEGER (Array;1000;5).
```

フィールド

フィールドを使って演算する場合は、フィールドは使わずに値を変数に代入して計算する方が実行効率がよくなります。

▼ 次のようなメソッドがある場合

```
Case of
```

```
  ¥([Contacts]City="Saratoga")
```

```
    Ship:="Blue"
```

```
  ¥([Contacts]City="Reno")
```

```
    Ship:="Red"
```

```
  ¥([Contacts]City="Boston")
```

```
    Ship:="FedEx"
```

```
End case
```

このメソッドは、下記のように記述すると実行速度が速くなります。

```
$Dest:=[Contacts]City
```

```
Case of
```

```
  ¥($Dest="Saratoga")
```

```
    Ship:="Blue"
```

```
  ¥(Dest="Reno")
```

```
    Ship:="Red"
```

```
  ¥($Dest="Boston")
```

```
    Ship:="FedEx"
```

```
End case
```

ループの中でこのようなコードが頻繁に実行されると、パフォーマンスは著しく違ってきます。

ポインタ

フィールドの場合と同じように、ポインタ参照よりも変数を使った方が速くなります。ポインタ参照される変数で何回も計算する場合、値を変数に格納すると時間を節約できます。

- ▼ たとえば、ポインタ “MyPtr” がフィールドや変数を指しており、その値を使って一連のテストをする場合

```

Case of
  ¥(MyPtr->=1)
    `Sequence1
  ¥(MyPtr->=2)
    `Sequence2
.
.
End case

```

この Case of ステートメントは、以下のように記述すればさらに速くなります。

```

$Temp:=MyPtr->
Case of
  ¥($Temp=1)
    `処理 1
  ¥($Temp=2)
    `処理 2
.
.
End case

```

ローカル変数

コードを作成する場合は、できるだけローカル変数を使ってください。ローカル変数には、次の利点があります。

- ローカル変数は、データベースのスペースを多く必要としません。ローカル変数は、それらが使われるメソッドに入るときに作られ、メソッドの実行が終わると破棄されます。
- 生成されるコードは、ローカル変数（特に倍長整数）に関して最適化されます。これはループカウンタに便利です。

A

コンパイラメッセージ

ここでは、4D Compiler が出力するメッセージについて説明します。メッセージは、以下の 5 種類があります。

- 警告メッセージ
- 詳細警告メッセージ
- エラーメッセージ
- 範囲チェックメッセージ
- コンパイラメッセージ

警告、詳細警告、エラーの各メッセージはエラーファイルに出力され、対話型デバッグ時に 4th Dimension の画面に表示されます。

範囲チェックメッセージは、コンパイル後のデータベースの実行時に「アラート」ダイアログボックスに表示されます。

コンパイラメッセージは、コンパイル処理実行中のメッセージです。コンパイル処理の実行中に「アラート」ダイアログボックスに表示されます。

注 この付録に掲載されているメッセ - ジは、4D Compiler の開発中に作成されているため完全なものではありません。

警告メッセージ

警告メッセージは、4D Compiler の “ タイプチェック処理 ” フェーズで出力されます。ここでは、各メッセージを問題のあるコード例とともに示します。

「 **COPY ARRAY** コマンド中にポインタが存在します。」

COPY ARRAY (Pointer-> ; Array)

「 **SELECTION TO ARRAY** コマンド中にポインタが存在します。」

SELECTION TO ARRAY (Pointer-> ; MyArray)

SELECTION TO ARRAY ([MyTable] MyField; Pointer->)

「 **ARRAY TO SELECTION** コマンド中にポインタが存在します。」

ARRAY TO SELECTION (Pointer-> ; [MyTable] MyField)

「 **LIST TO ARRAY** コマンド中にポインタが存在します。」

LIST TO ARRAY ("List" ; Pointer->)

「 **ARRAY TO LIST** コマンド中にポインタが存在します。」

ARRAY TO LIST (Pointer-> ; "List")

「 配列定義コマンド中にポインタが存在します。」

ARRAY REAL (Pointer-> ; 5)

ARRAY REAL (Array ; Pointer->) と書いた場合、このメッセージは出力されません。配列の次元数はデータタイプに影響を与えないからです。ポインタで参照する配列は、事前に定義する必要があります。

「 **Undefined** コマンドは使用しないでください。」

If (Undefined (Variable))

コンパイルしたデータベース中の **Undefined** コマンドは、常に **False** (偽) を返します。

「このメソッドは、パスワードによって保護されています。」

「フォーム 1 ページ目の自動動作ボタンの名前がありません。」

すべてのボタンは名前を持っていないければなりません。

詳細警告メッセージ

詳細警告メッセージは、ユーザが「メイン」ウインドウで“詳細警告”を指定した場合だけ出力されます。メッセージは、エラーファイルに格納されます。

「ポインタの参照先を文字として処理します。」

```
Pointer->[[2]]:="a"
```

「文字列のインデックスを数値タイプとして処理します。」

```
MyString[[Pointer->]]:="a"
```

「配列のインデックスを実数タイプとして処理します。」

```
ALERT (MyArray{Pointer->})
```

ポインタでテキスト配列または文字配列の要素を参照している場合。

プラグインコマンドの呼び出しでパラメータが不足しています。

```
SP SELECT CELL(Area)
```

エラーメッセージ

エラーメッセージは、“タイプチェック処理”フェーズで生成されエラーファイルに書き込まれます。

各メッセージには例を示してあります。エラーメッセージを以下の6つに分けて説明します。

- タイプチェック
- シンタックス
- 引数
- 演算子
- プラグイン
- 総合エラー

タイプチェック

「変数のタイプはブールから実数に変更できません。」

「変数のタイプが異なるため代入できません。」

```
MyReal:=12.3
MyBoolean:=TRUE
MyReal:=MyBoolean
```

```
`MyReal は実数。
`MyBoolean はブール。
`ブールは実数に代入できません。
```

「文字列の長さは変更できません。」

```
C_STRING (3;MyString)
C_STRING (5;MyString)
```

「配列の次元数は変更できません。」

```
ARRAY TEXT (MyArray;5;5)
ARRAY TEXT (MyArray;5)
```

「配列定義コマンドに要素数がありません。」

```
ARRAY INTEGER (MyArray)
```

「変数が必要です。」

```
COPY ARRAY (MyArray;")
```

「定数が必要です。」

```
C_STRING (Variable;MyString)
```

「変数のタイプが不明です。」

「この変数はメソッドで使用されています。」:

変数のタイプを決められません。コンパイラ命令を補ってください。

「定数のタイプが無効です：文字。」

```
OK:= " 本日は晴天なり "
```

「メソッド "nn" が不明です。」

この行は存在しないメソッドを呼んでいます。

誤ったフィールドの使用。

```
MyDate:= Add to date(BooleanField; 1;1;1)
```

「文字列の長さは 255 文字 (バイト) までです。」

```
C_STRING (325;MyString)
```

「変数 Variable は、メソッドではありません。」

```
Variable (1)
```

「変数 Variable は、配列ではありません。」

Variable{5} := 12

「結果が式と一致しません。」

Text := "Number" + Num (\$i)

「タイプが一致しません。」

Integer := MyDate*Text `日付とテキストの乗算はできません。

「引数 \$ のインデックスが数値ではありません。」

\$i := "3" ` \$i のタイプがテキスト。
\${\$i}:= 5

「定数のタイプが無効です：文字。」

IntArray {"3"} := 4 ` インデックスのタイプがテキスト。

「変数 " Variable " のタイプをテキストから配列に変更できません。」

C_TEXT (Variable)
COPY ARRAY (TextArray; Variable)

「変数 " Variable " のタイプをテキストから数値に変更できません。」

Variable := Num (Variable)

「配列 " IntArray " は、整数タイプの配列からテキストタイプの変数に変更できません。」

ARRAY TEXT (IntArray;12)

IntArray が他の所で整数配列として定義されている場合。

「ポインタタイプ以外の変数をポインタとして参照しています。」

Variable-> := 5

Variable のタイプがポインタではない場合。

「テキスト変数を数値変数として使用できません。」

Variable := 3.5

「フィールドの使い方に誤りがあります」

Variable := [MyTable]MyDate

[MyTable]MyDate は日付フィールドで Variable は数値です。

シンタックス

「ポインタタイプ以外の変数をポインタとして参照しています。」

Variable := Num (" 本日は晴天なり ")>

この関数は使用できません。

「シンタックスエラー」

If (Boolean)

End For

End If のはず。

「}」がありません。」

文中の左カッコ{ の数が右カッコ} より多い場合。

「{」がありません。」

文中の右カッコ} の数が左カッコ{ より多い場合。

「) 」がありません。」

文中の左カッコ(の数が右カッコ) より多い場合。

「(" 形のカッコがありません。」

文中の右カッコ) の数が左カッコ(より多い場合。

「フィールドが必要です。」

If (Modified (Variable))

Modified 関数にはフィールドを渡してください。

「{」が必要です。」

C_INTEGER (Array 2)

Array(2):=1

「変数が必要です。」

C_INTEGER ([MyTable]MyField)

「数値が必要です。」

C_INTEGER (\${ "3" })

「;"」が必要です。」

COPY ARRAY (Array1 Array2)

「文字参照記号が多すぎます。」

「定数タイプが無効です。」

MyString[[3:= "a" - on Windows

「文字参照記号が多すぎます。」

MyString3]]:= "a" - on Windows

「ここではサブテーブルを使えません。」

ARRAY TO SELECTION (array; subtable)

「If ステートメントの判定結果はブールタイプでなければなりません。」

If (MyReal)

MyReal が数値変数です。

「式が複雑すぎます。」

ステートメントを分割して短くしてください。

「メソッドが複雑すぎます。」

メソッド中に 601 以上の異なる **Case** または 101 以上の **If...End if** 構造がありません。

「フィールドが不明です。」

使用メソッドがおそらく他のデータベースからコピーされたもので、存在しないフィールドへの参照が式に含まれています。

「テーブルが不明です。」

使用メソッドがおそらく他のデータベースからコピーされたもので、存在しないテーブルへの参照が式に含まれています。

「演算とタイプが一致しません。」

Pointer := ->Variable + 3

変数タイプが異なるため代入できません。」

「変数 "Variable" のタイプはテキストから実数に変更できません。」

「文字列インデックスの使い方に誤りがあります。」

MyReal[[3]] あるいは、**MyString[[Variable]]**

MyReal が数値変数で、Variable が数値変数以外の場合。

引数

「結果が式と一致しません。」

MyMethod (Num(MyString))

MyMethod の引数にブール式が必要な場合。

「タイプが一致しません。」

「このメソッドに渡す引数が多すぎます。」

DEFAULT TABLE (table; form)

「定数タイプが無効です：整数」

MyMethod (3+2)

MyMethod の引数にブール式が必要な場合。

「変数のタイプが異なるため代入できません。」

C_INTEGER (\$0)

＼\$0 は整数

\$0 := False

＼ここで \$0 にブールを代入

「引数 \$ のインデックスが数値ではありません。」

C_INTEGER (\${3})

For (\$i;3;5)

\${i} := String (\$i)

＼ここで \${i} はテキスト

End For

「このコマンドに引数は不要です。」

SHOW TOOL BAR (MyVar)

DEFA ULT TAB LE

「変数 " MyString " のタイプはテキストからブールに変更できません。」

MyMethod (MyString)

MyMethod 引数にはブール式が必要な場合。

「定数タイプが無効です：文字。」

Calculate ("3+2")

＼引数のタイプはテキスト

メソッド "Calculate " の中でコンパイラ命令 **C_INTEGER(\$1)** が記述されているとき。

「COPY ARRAY コマンドの引数が、変数です。」

COPY ARRAY (variable; array)

「引数 "\$1" のタイプが呼ぶ側のメソッドと呼ばれる側のメソッドで一致していません。」

Print ("LaserWriter")

メソッド "Print" で、\$1 が数値の場合。

「タイプが一致しません。」

\$1 := String (\$1)

「変数 " MyArray " のタイプは配列からブールに変更できません。」

MyMethod (MyArray)

配列をメソッドに渡すときには、その配列のポインタを渡してください。

「引数としてそのコマンドに渡せません。」

RECEIVE VARIABLE (\$1)

「引数 "\$1" のタイプはブールから整数に変更できません。」

GET FIELD PROPERTIES (tablename; fieldnumber; type; \$1)

演算子

「演算とタイプが一致しません」

Boolean2 := Boolean1 + True

「演算子 > は不要です。」

QUERY ([MyTable];[MyTable]MyField=0;>)

「変数 " Picture2" のタイプはピクチャから実数に変更できません。」

If (Number = Picture2)

Number が倍長整数で Picture2 がピクチャ。

「この変数タイプに -(マイナス) 符号を付けることはできません。」

Boolean := -False

プラグインコマンド 「プラグインコマンド "nn" の定義が正確ではありません。」
プラグインコマンドの定義に誤りがある場合。

「プラグインコマンドに対する引数が足りません。」
「プラグインコマンドに対する引数が多すぎます。」

総合エラー

「同じ名称のメソッドが複数あります : nn。」

データベースをコンパイルするには、プロジェクトメソッドすべてに異なる名前をつける必要があります。

内部エラー No. xx

このメッセージが出力されたら、ACI Technical Support に電話して、エラー番号を教えてください。

Variable のタイプが不明です。

「この変数は M1 メソッドで使用されています。」

変数のタイプが判断できません。コンパイラ命令を使ってください。

「ローカル変数の合計サイズが 32KB を超えました。サイズ : xx バイト」

ローカル変数の数を減らす必要があります。このメッセージには、固定長文字列 (C_STRING コマンドで定義された文字列) 以外のローカル変数が使っているメモリサイズが示されます。これは、Macintosh、Windows いずれのプラットフォームにも適用されます。

「オリジナルのメソッドが壊れています。」

オリジナルのストラクチャでメソッドが壊れています。該当するメソッドを削除するか置き換えてください。

「4D のコマンドではありません。」

メソッドが壊れているか、または、そのコマンドが 4th Dimension に追加される前にリリースされたバージョンのコンパイラを使用しています。

「フォーム "Format" の変数 "Variable" のタイプは変更できません。」

フォーム内のグラフィックタイプの変数に OK といった名前をつけると、このメッセージが出力されます。

「関数と変数が同じ名前です : Name。」

関数か変数いずれかの名前を変えてください。

「関数 "String" の名前がフォーム "入力 1" の変数と同じです。」

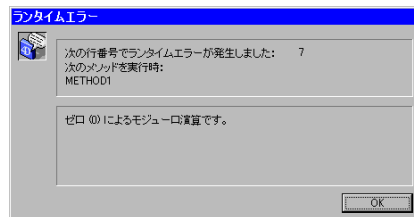
関数または変数いずれかの名前を変えてください。

「メソッドと変数が同じ名前です：Name。」
メソッドか変数の名前を変えてください。

「プラグインコマンドと変数が同じ名前です：Name。」
プラグインコマンドか変数の名前を変えて下さい。

範囲チェックメッセージ

範囲チェックメッセージは、コンパイル後のデータベースの実行中に 4th Dimension で、次の図のウインドウに表示されます。



φ ゼロによる割算が発生しました £

```
Var1:=0
Var2:=2
Var3:=Var2%Var1
```

「EXECUTE 内での無効な引数」
変数がデータベース内で明確に現れない場合。

「変数へのポインタがコンパイラには未知のものです。」

```
Pointer := Get pointer ("Variable")
```

変数がデータベース内で明確に宣言されていない場合。

「ポインタを使用して再タイプ設定を試行しました。」

```
BoolVar:= Pointer->
```

ポインタが整数タイプのフィールドを指す場合。

「ポインタの誤った使用。」

Character := StringVar [[Pointer->]] - on Windows

Character := StringVar ≤Pointer->≥ - on Macintosh

ポインタが数値を指さない場合。

「結果が変数の範囲を越えました。」

MyArray{17} := 2.3

上記のステートメントが実行されたとき、MyArray は要素数が 5 個の配列です。このメッセージは MyArray{17} にアクセスしようとする则表示されます。

⊘ ゼロによる割算が発生しました £

Var1 := 0

Var2 := 2

Var3 := Var2/ Var1

「引数がありません。」

カレントプロシージャには引数が 3 つしか渡されていないのに、ローカル変数 \$4 を使っているような場合にこのメッセージが表示されます。

「ポインタが初期化されていません

MyPointer-> := 5

MyPointer がまだ初期化されていない場合。

「代入先が小さすぎます。」

C_STRING (5;MyString1)

C_STRING (10;MyString2)

MyString2 := "TheString"

MyString1 := MyString2

`MyString2 は 9 文字ですが、

`MyString1 は 5 文字しか格納できません。

⊘ 文字参照エラー £

i:=30

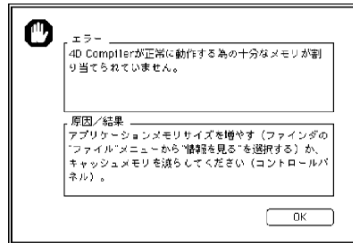
MyString≤i≥ := MyString2

「引数に渡された文字列が空、もしくは、初期化されていません。」

MyString[[1]]:= ""

コンパイラーメッセージ

作業環境があまり良くないとき、4D Compiler はメッセージを表示して、ユーザに作業環境を改善するように促します。このメッセージは下図のようにアラートボックスに表示されます。



図中の“エラー”の領域には、問題点が示されます。“原因・結果”の領域には、その問題を解決する手段が示されます。

B

アプリケーションの カスタマイズ

4D Compiler を使用している場合、データベースと 4D Engine をマージして、ダブルクリックで起動できるスタンドアロンのアプリケーションを作成することができます。これは、「オプション」ウインドウの「4D Engine を組み込む」オプションを選択するだけで実現できます。このオプションに関する詳細は、[第2章「実行形式アプリケーションの作成」](#)を参照してください。

この付録に記載されているどの手順に従ってもアプリケーションのカスタマイズを行なうことができます。

Macintosh

カスタムアイコンを作成する前には ResEdit などのリソースエディタとコンパイル後の実行形式のアプリケーションが必要です。

注 ResEdit は、Apple Computer, Inc. によるリソース編集用のユーティリティです。以下の説明は、ResEdit の使い方を良くご存じの方を対象にしています。

データベースアイコンのカスタマイズ

データベースのアイコンをカスタマイズするには、以下の手順に従ってください。

- 1 コンパイルしてマージしたストラクチャファイルを ResEdit で開きます。
以下の手順でアプリケーションのクリエイータを指定します。

- 2 BNDL リソースを開き、自分のアプリケーションの Creator 4 文字 (バイト) で OwnerName を置き換えます。

Creator は、ハードディスク上のいかなる Macintosh アプリケーションの Creator とも異なるものにする必要があります。不用意に既存のアプリケーションの Creator を使ってしまうと、指定したアプリケーションのアイコンだけではなく、既存のアプリケーションのアイコンも変更されてしまいます。

ご自分のアプリケーションを販売する場合は、指定した Creator がユニークかどうか Apple Computer, Inc. に確認して、発売された他の Macintosh アプリケーションで使われていないかを確認してください。

- 3 BNDL リソース内のアイコンを編集します。

変更するアイコンをダブルクリックします。カラーモニタで使用する 3 種類のアイコンを変更することができます。

- 4 SIG# リソースを開いて、自分のアプリケーション用の 4 文字の Creator を入力してください

- 5 各リソースを変更し終わったら、「File」メニューから「Save」を選んで保存します。アプリケーションを閉じて、ResEdit を終了します。

これで、ユーザのアプリケーションアイコンのカスタマイズは完了です。アプリケーションの実行中に、そのアプリケーションで作成されるデータファイルやクイックレポート、ラベル、検索条件などのファイルアイコンもカスタマイズできます。

他のデータベースファイル用アイコンのカスタマイズ

4th Dimension データベースで作成された他の全てのファイル (例えば、データファイル、クイックレポート、ラベル、保存された検索、等) のアイコンもカスタマイズすることができます。

これらのファイルのアイコンをカスタマイズするには、

- 1 ICN#, icl4 リソースを開き、ビットマップエディットツールを使ってアイコンをそれぞれ変更します。
- 2 アイコンの編集が終了したら、「File」メニューから「Save」を選びます。

3 アプリケーションを閉じて、終了します。

コンパイラでカスタムアプリケーションを作成した後、Desktop を作り直してください。この操作で、Finder によってユーザのカスタムアイコンが Desktop ファイルに追加されます。Desktop の再作成が終了すると、作成したカスタムアイコンが表示されます。



注 Desktop を再構築するには、Finder の「特別」メニューから「再起動」を選び、option キーとコマンドキーを押し続けます。すると、ダイアログボックスが表示され、Desktop を作り直すかどうかを尋ねてきます。

Windows

アプリケーションのアイコンをカスタマイズする

Borland International 社の Resource Workshop™ のような Windows リソースエディタを使用する必要があります。

実行形式のアプリケーションのアイコンをカスタマイズするには、

- 1 実行形式のアプリケーションのファイル (MyDatabase.EXE) をリソースエディタで開きます。
- 2 ICON リソースを選択します。

標準の 4th Dimension アイコンに関連した幾つかのアイコンを見つけることができます。

実行形式のアプリケーションのアイコンは、APPICON ID ナンバーに位置しています。

- 3 アイコンを変更するには、リソースをダブルクリックして下さい。
- 4 操作を完了するには、実行形式のアプリケーションのファイルを保存して下さい。
- 5 4D Compiler で、 実行形式アプリケーション作成 オプションをオンにしてデータベースをコンパイルします。

コンパイルが終了すると、実行形式のファイル、DataBaseName.EXE が作成されます。

アイコンをアプリケーションと関連させるには、

- 1 プログラムマネージャ内に Windows Installer program を持っていない場合には、「File」メニューのプロパティを選択します。
- 2 Change Icon を選択し、DataBaseName.exe を選択します。

これにより、カスタマイズされたアイコンを、プログラムマネージャのレベルで実行形式のアプリケーションに関連させます。

注 Windows95 及び WindowsNT では、アイコンを正しくインストールさせるためには、インストーションプログラムを使用しなければなりません。

アプリケーションの ウィンドウ名の変更

実行形式のアプリケーションを Windows 用 4D Compiler で作成する際、アプリケーションのウィンドウのデフォルト名は実行形式のファイルに対応します。例えば、DISTRIB のフォルダには、次のファイルがあります：MyDataBase.4DC、MyDataBase.RSR、MyDataBase.EXE、ASINTPPC.DLL、など。アプリケーションのウィンドウ名は、"MyDataBase" です。アプリケーションのウィンドウは、Mac OS あるいは Windows でカスタマイズできます。

Mac OS

- 1 4D Transpoter を使用し、コンパイルされていないストラクチャファイル Windows 用から Macintosh バージョン用に変換します。
- 2 リソースエディタを使って、データベースのストラクチャファイルを開きます。
- 3 タイプ "STR " のリソースを、13010 の ID で作成します。
- 4 アプリケーションのウィンドウ名をこのリソースに入力します。
- 5 データベースを Windows 用に変換して、Windows 側に戻します。
スタートアップ時のメインウィンドウは、入力した名前になります。

Windows

- 1 4th Dimension を使って、ストラクチャファイルを開きます。
- 2 新しいプロジェクトメソッドを作成します。
- 3 SET STRING RESOURCE コマンドを使って、ストラクチャファイル内に 13010 の ID で、"STR " タイプのリソースを作成します。
以下のようなコードのラインを書きます。

SET STRING RESOURCE (13010; "My Application")

このメソッドを実行した後、データベースをコンパイルして実行形式のアプリケーションを作成します。スタートアップ時のメインウィンドウは "My Application" というタイトルになります。

C

リソースのカスタマイズ

4th Dimension に含まれたユーティリティである Customizer Plus を使用して 4D Compiler のリソースをカスタマイズすることができます。

注 Customizer Plus に関する詳細は、4th Dimension に含まれている Customizer Plus Reference マニュアルを参照して下さい。

Customizer Plus 内から 4D コンパイラーのアプリケーションを選択すると、次のようなウィンドウが現れます。

Windows



Macintosh



1つのリソース・アイコン、¢表記£が表示されます。

表記

☿ 表記 ☿ アイコンをダブルクリックすると、一覧が表示されます。4D Compiler で使用する 4th Dimension コマンドをこのダイアログボックスによって指定します。



ポップアップのリストから言語を選択します。「エラー」ファイル、「タイプ」ファイル、「シンボルテーブル」ファイルには指定された言語を使って出力されません。

索引

記号

% (モジュール演算子) 96
() (丸っこ) 114
.CMP ファイル (Windows) 7
[[]] (文字参照記号) 114, 115
{ } (中かっこ) 114

数字

386/486 プロセッサ 24
4D Compiler 4
 オプションウィンドウ 17 30
 コンパイルオプション 17 30
 タイプ設定変数 51 52
 タイプ定義される変数 57
 パスワードで保護されたデータベース 30
 メニュー 10 16
 最適な使用 2
4D Compiler Pro 18, 123
4D Compiler ウィンドウ 31
 一時停止ボタン 31
 中止ボタン 31, 33
4D Engine 18
4D Engine を組み込む 123
4D Runtime 1, 4
4D Compiler (Macintosh および Windows) 4
68020/30/40 プロセッサ 1

A

ABORT コマンド
 構文 94 95
ACCUMULATE コマンド 96
ADD TO SET コマンド 100
ALERT コマンド 111
Append Document 関数
 構文 95

APPLY TO SELECTION コマンド 100
ARRAY INTEGER コマンド 112
ARRAY POINTER コマンド 98
ARRAY REAL コマンド 110
ARRAY STRING コマンド 60, 67
ARRAY TEXT コマンド 67, 112, 113
ARRAY TO LIST コマンド

 構文 92

ARRAY TO LIST コマンド 110
ARRAY TO SELECTION コマンド

 構文 92

ARRAY TO SELECTION コマンド 110

Ascii 関数

 構文 96

B

BREAK LEVEL コマンド 96

C

Case of 構文 93, 99, 106, 115

CLEAR VARIABLE コマンド

 構文 97

COPY ARRAY コマンド 110, 112, 114

COPY NAMED SELECTION コマンド 100

Create Document 関数

 構文 95

CREATE EMPTY SET コマンド 100

CREATE SET コマンド 100

C_STRING コマンド 59

Customizer Plus 127

 トランスレーション (表記) リソース 128

CUT NAMED SELECTION コマンド 100

D

DEFAULT FILE コマンド 116

DIALOG コマンド 100

E

EXECUTE コマンド
...の使用をさける方法 99
構文 97, 99

EXPORT DIF コマンド 100
EXPORT SYLK コマンド 100
EXPORT TEXT コマンド 100

F

Field 関数
構文 96
For...End for 構文 5

G

GET FIELD PROPERTIES コマンド 55
GET LIST ITEM コマンド 55
Get Pointer 関数
構文 97, 98
GOTO RECORD コマンド 100
GOTO SELECTED RECORD コマンド 100
GRAPH FILE コマンド 100

I

IDLE コマンド
構文 94 95
If...End if 構文 115
If...End if 構文 117
IMPORT DIF コマンド 100
IMPORT SYLK コマンド 100
IMPORT TEXT コマンド 100
INPUT FORM コマンド 100

L

LIST TO ARRAY コマンド
構文 92
LIST TO ARRAY コマンド 110
LOAD SET コマンド 100
LOAD VARIABLE コマンド
構文 97

M

Mod 関数
構文 96
Modified 関数 114

Motorola PowerPC 24

N

NO TRACE コマンド
構文 97, 99
Num 関数 114, 116

O

ON EVENT CALL コマンド
構文 94 95
ON SERIAL PORT CALL コマンド
構文 94 95
Open Document 関数
構文 95
ORDER BY FORMULA コマンド 100
ORDER BY コマンド 100
OUTPUT FORM コマンド 100

P

PAGE SETUP コマンド 100
Pentium 24
Power Macintosh 24
PowerPC 24
PRINT FORM コマンド 100
PRINT LABEL コマンド 100

Q

QUERY BY FORMULA コマンド 100
QUERY SELECTION BY FORMULA コマンド 100
QUERY SELECTION コマンド 100
QUERY コマンド 100

R

RECEIVE VARIABLE コマンド
構文 93
RECEIVE VARIABLE コマンド 117
REDUCE SELECTION コマンド 100
REMOVE FROM SET コマンド 100
REPORT コマンド 100
ResEdit 123

S

SAVE VARIABLE コマンド
構文 97
SELECTION TO ARRAY コマンド
構文 92

SELECTION TO ARRAY コマンド 110

SEND VARIABLE コマンド

構文 93

SHOW TOOL BAR 関数 116

String 関数 116, 117

SUBSELECTION TO ARRAY コマンド

構文 92

Subtotal 関数

構文 96

T

Table 関数

構文 96

TRACE コマンド

コマンド 99

構文 97

Type 関数

構文 94

U

Undefined 関数

構文 97

Undefined 関数 110

URL

...経由で呼び出されたメソッド 55

あ

アドレス 5

い

一時停止ボタン

4D Compiler ウィンドウ 31

4D Compiler ウィンドウ

一時停止ボタン 33

インタープロセス変数 38, 50

フォーム変数 66 67

...のタイプの矛盾 61 63

インタプリタ 3, 5, 6, 51, 75, 96, 97

...で動作するデータベース 31

実行時間 6

インタプリタモード 5

え

エラー

表示 33

エラーファイル 4, 7, 21, 24, 41 46, 111

テキストとして 44

...の構成 44

...内の警告 93

警告 43 44

使用 44 46

一般的なエラー 41 42

対話式デバッグに使用 44

特定行に関連するエラー 42 ???

エラーファイル参照中止メニューコマンド 10

エラーメッセージ 111 119

シンタックス 114 115

タイプ設定 112 113

プラグインコマンド 118

引数 116 117

演算子 117

総合エラー 118 119

演算

構文 96

演算子

エラーメッセージ 117

お

オプションウィンドウ 12, 17 30

Macintosh スクリプトマネージャ 23

エラーファイル 21

コンパイルデータベース名 17

コンパイルパス 28

シンボルテーブル 22

タイプファイル 27

デフォルトボタンタイプ 28

デフォルト数値タイプ 28

デフォルト文字タイプ 29

バージョン番号自動生成 29

プロセッサの種類 24 25

ローカル変数初期化 26

「次...」ボタン 25

警告 23

最適化 26

実行形式アプリケーションの作成 18

範囲チェック 23

オプションウィンドウ 14

オプション引数付きのコマンド 55

か

カーネルコール 95

カウンタ 56, 104

- く**
 クイックレポート 77
 クイックレポート変数 77
- け**
 警告 9, 43 44, 110 111
 表示 33
 警告メッセージ 48, 110
- こ**
 構造化プログラミング 103
 COPY ARRAY コマンド
 構文 91 92
 構文の詳細
 ABORT コマンド 94 95
 Append Document 関数 95
 ARRAY TO LIST コマンド 92
 ARRAY TO SELECTION コマンド 92
 COPY ARRAY コマンド 91 92
 Create Document 関数 95
 IDLE コマンド 94 95
 LIST TO ARRAY コマンド 92
 ON EVENT CALL コマンド 94 95
 ON SERIAL PORT CALL コマンド 94 95
 Open Document 関数 95
 RECEIVE VARIABLE コマンド 93
 SELECTION TO ARRAY コマンド 92
 SEND VARIABLE コマンド 93
 SUBSELECTION TO ARRAY コマンド 92
 Type 関数 94
 コミュニケーション 93 94
 データエントリー 94
 ローカル配列 93
 配列コマンド 91 93
 配列関連コマンドでのポインタ 93
 文書 95
 例外 94 95
 SAVE VARIABLE コマンド 97
 Ascii 関数 96
 CLEAR VARIABLE コマンド 97
 EXECUTE コマンド 97, 99
 Field 関数 96
 Get Pointer 関数 97, 98
 LOAD VARIABLE コマンド 97
 Mod 関数 96
 NO TRACE コマンド 97, 99
 SAVE VARIABLE コマンド 97
 Subtotal 関数 96
 Table 関数 96
 TRACE コマンド 97, 99
 Undefined 関数 97
 ストラクチャへのアクセス 96
 その他 97 99
 ブレイク処理 96
 演算 96
 各種のコマンドで使用されるポインタ 100
 使用に注意するコマンド 100
 文字列 96
 変数 97 98
 効率良く記述されていないメソッド 2
 コードの最適化 26
 コードの翻訳 5
 コマンド 100
 コマンドの説明 91 101
 コミュニケーション
 構文 93 94
 コンパイラ 3
 実行時間 6
 実行速度 6
 コンパイルメッセージ 121
 コンパイラ命令 51 60
 4D Compiler でタイプ定義する変数 57
 C_BLOB 52
 C_BOOLEAN 52
 C_DATE 52
 C_GRAPH 52
 C_INTEGER 52, 56, 104, 114, 116
 C_LONGINT 52, 57, 104
 C_PICTURE 52
 C_POINTER 52
 C_REAL 52
 C_STRING 52, 56, 59, 63, 105, 112, 120
 C_TEXT 52
 C_TIME 52
 インタプリタで使う 56 57
 タイプの矛盾 62 63
 タイプ設定変数 51 60
 データ設定の割り当て 32, 52 60
 ...の使用によるコードの最適化 56, 104 107
 ...を手動で識別する 58
 引数 58
 開発者がタイプ定義する変数 58
 記述する場所 57 59
 必要な時 53 55
 コンパイルオプション 17 30
 Macintosh スクリプトマネージャ 23

- エラーファイル 21
- オプションウィンドウ 17 30
- オプションについて 30
- コンパイルデータベース名 17
- コンパイルパス 28
- シンボルテーブル 22
- タイプファイル 27
- デフォルトボタンタイプ 28
- デフォルト数値タイプ 28
- デフォルト文字タイプ 29
- バージョン番号自動生成 29
- プロセッサの種類 24 25
- ローカル変数のタイプ設定パス 32
- ローカル変数初期化 26
- 警告 23
- 実行形式アプリケーションの作成 18
- 範囲チェック 23
- コンパイルオプション最適化 26
- コンパイルの利点
 - 実行速度 5 6, 7
 - データベースの保護 6 7
- コンパイルパス 28
- コンパイルプロセス 31 35
 - コンパイラ命令によるタイプ設定パス 32
 - コンパイルパス 33
 - コンパイル時間の短縮 56
 - タイプ設定フェーズ 32 33
 - データベースのコピー 31
 - データベース用 2
 - ...用のデータベースの準備 49 60
 - 開始 30
- コンボボックス 66

- さ
- 再コンパイル
 - プロジェクトを使用 14

- し
- システム変数 76
- 実行形式アプリケーションの作成 9, 18
- 実行形式アプリケーション作成 9, 33
- 実行速度 5 6
- シンタックスエラー 114 115
- シンボルテーブル 22, 37, 38 40, 51
 - データタイプを確定 51
 - プロセス変数のリスト 38
 - メソッドのリスト 40
 - ローカル変数のリスト 40

- す
- 数値変数 2
- スクリプトマネージャ 9, 23
- スクロールエリア 66
- Startup** メソッド 57
- ストラクチャへのアクセス
 - 構文 96

- そ
- 総合エラー 118 119
- その他
 - 構文 97 99

- た
- タイプファイル 27, 37, 46
- タイプ設定 61 90
- C_STRING** コマンド 59
 - URL 経由で呼び出されたメソッドを使う場合 55
 - オプション引数付きのコマンドを使う場合 55
 - フォーム変数 66 67
 - プラグインコマンド 68 73
 - プロセスとインタープロセス変数 61 63
 - ポインタの使用 67, 74 75
 - ポインタを使う場合 54
 - ローカル変数 64
 - 引数処理 74 76
 - 次数と文字列 56
 - 数値変数 52, 104 105
 - 定数 77 90
 - 配列 64 65
 - 複数のシンタックスコマンドを使う場合 55
 - 文字変数 56
 - 文字列 105
 - 変数 51
 - 予約変数 76
- タイプ設定フェーズ 32 33
 - ...でのエラー 32
 - ...のまとめ 32
- タイプ設定変数
 - コンパイラ命令 51 60
- タイプ変更
 - 配列 65
 - 変数 62
- 対話式デバッグ 1, 4, 7, 14, 21, 44, 46
 - ...のためにエラーファイルに名前をつける 21
- タブコントロール 66
- ダブルクリックで起動できるアプリケーション

...のカスタムアイコン 123
単純変数 66

ち

中止ボタン
4D Compiler ウィンドウ 31

て

定数
前もって定義された... 77 90
データエントリー
構文 94
データタイプ設定, タイプ設定も参照 61
データベース
コンパイル後の...の使用 34
コンパイル中 9
コンパイル用の準備 49 60
開く 1
再コンパイル 14
保護 6 7

データベースメソッド
シンボルテーブル内の... 40

テキスト vi
デバッグング 4
デフォルトプロジェクト 15
デフォルトボタンタイプ 28
デフォルト数値タイプ 28
デフォルト文字タイプ 29

と

ドラッグ&ドロップ 34
トランスレーション(表記)リソース
Customizer Plus 128
ドロップダウンリスト 66

に

入力可の変数 66
入力不可の変数 66

は

バージョン番号生成
自動 29
配列 120
2次元 106
シンボルテーブル内の... 38
ローカル 65

...のタイプの矛盾 64 65
...の暗黙のタイプ変更 65
...の次元変更 92
次元数の変更 65
探す 32
文字 65
要素のデータタイプの変更 64

配列コマンド

構文 91 93
配列に関連したコマンド
ポインタの使用 93
配列定義ステートメント 32
配列定義コマンド 67
パスワードによる保護 30
範囲チェック 23
ポインタに関係する矛盾 67
範囲チェックメッセージ 119 120
範囲チェック 9, 37, 46 48
異常診断 48
使用 47

ひ

引数
On Drag Over フォームイベント 59
エラーメッセージ 116 117
コンパイラ命令 58
データベースメソッドが受取る 58
間接参照 75
処理 74 76
包括 75
表示変数 66

ふ

ファイルメニュー 10 16
デフォルトプロジェクト 15
開く 12
元に戻す 15
再コンパイル 14
新規 11
閉じる 14
保存 15
名前をつけて保存 15
フィールド 106
フォーム変数
グラフ 66
コンボボックス 66
サーモメーター 66
スクロールエリア 66

- ダイアル 66
- タブコントロール 66
- チェックボタン 66
- ドロップダウンリスト 66
- ピクチャ 66
- プラグインオブジェクト 66
- ボタン 66
- ポップアップメニュー 66
- メニュー 66
- メニュー / ドロップダウンリスト 66
- リスト 66
- ルーラー 66
- ...のタイプの矛盾 66 67
- 単純変数 66
- 入力可 66
- 入力不可 66
- 表示変数 66
- 複数のシンタックスコマンド 55
- プラグイン
 - マルチプラットフォームコンパイル 68 72
- プラグインコマンド
 - エラーメッセージ 118
 - ...に引数を渡す 68
 - ...のタイプの矛盾 68 73
 - 暗黙の引数 73
- ブレイク処理
 - コンパイルされたデータベース内での... 96
 - 構文 96
- プロジェクト
 - データベースの再コンパイル 14
 - デフォルト 15
 - ...の概念 12 14
 - ...の定義 9
 - 既存の...を開く 12 14
- プロジェクトメソッド
 - シンボルテーブル内の... 40
- プロセス変数 38, 50
 - シンボルテーブル内の... 38 39
 - フォーム変数 66 67
 - ...のタイプの矛盾 61 63
- プロセッサ 24 25
 - 386/486 24
 - Motorola PowerPC 24
 - Pentium 24
 - PowerPC 24
- 文書
 - 構文 95
- へ
- ヘルプ
 - Windows 30
- 単純変数 66
- 変数
 - 4D Compiler でタイプ定義する 57
 - インタープロセス 61 63
 - クイックレポート 77
 - コンパイラ命令でタイプ設定する 51 60
 - コンボボックス 66
 - システム 76
 - シンボルテーブル 51
 - スクロールエリア 66
 - タイプ設定 32 33, 49 60
 - タブコントロール 66
 - デフォルトボタンタイプ 28
 - デフォルト数値タイプ 28
 - デフォルト文字タイプ 29
 - ドロップダウンリスト 66
 - フォーム 66 67
 - プラグインコマンドで作成される 73
 - プロセス 61 63
 - ポップアップメニュー 66
 - メニュー / ドロップダウンリスト 66
 - ローカル 64, 107
 - ...のタイプ 50
 - ...のタイプの矛盾 ?? 67
 - 開発者がタイプ定義する 58
 - 構文 97 98
 - 数値 52
 - 単純 66
 - 入力可 66
 - 入力不可 66
 - 表示 66
- ほ
- ポインタ
 - エラーメッセージ 113, 115
 - データタイプの判断 54
 - ...でフィールドを参照する 107
 - ...のタイプの矛盾 67
 - ...を使ってデータタイプの矛盾を避ける 74 75
 - ...を与える 96
 - 各種のコマンドで使用される... 100
 - 警告 110
 - 詳細警告 111
 - 配列の... 98
 - 配列関連コマンドでの 93

範囲チェックメッセージ 120
変数のデータタイプ 94

包括引数 75

ボタン 105

ポップアップメニュー 66

ま

マシン語 3

マス・コープロセッサ 2

マルチプラットフォームコンパイル

OLE ライブラリー 72

プラグイン用 68 72

実行形式アプリケーション 72

め

メインのオプションウィンドウ 12

メインのオプションウィンドウ 18

メニュー 10 16

エラーファイル参照中止 45 46

次のコンパイルエラー 4, 45

メニュー / ドロップダウンリスト 66

も

文字タイプの変数 63

文字配列 65

文字列 2

構文 96

よ

変数

予約 76

予約変数

クイックレポート 77

システム 76

る

ループカウンタ 56, 104

れ

例外

構文 94 95

ろ

ローカル配列

構文 93

ローカル変数 38, 65, 74, 106, 107, 116

カウンタをインクリメントする 56

シンボルテーブル内の... 40

タイプ設定 32

...のタイプの矛盾 64

定義 57

ローカル変数初期化

初期化 26