4D Compiler®

リファレンスマニュアル

Windows® and Mac™OS版

4D Compiler

by

David Hemmo

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。

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、いかなる変更に関してもACI SAおよびACI US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。

© ACI SA / ACI US 1985 - 1996; All rights reserved
© 4D Compiler 1985 - 1996 ACI SA. All rights reserved.
Author: David Hemmo

ACI®、4D®、4th Dimension®、4D Runtime®、4D Server™、4D Calc® 4D Compiler®、4Dロゴ、4th Dimensionロゴ、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®、Power Macintosh™、LaserWriter®、Image Writer®、QuickTime® はApple Computer Inc. の登録商標または商標です。

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。

序章

このマニュアルについて ・・・・・・・・・・ix
　マニュアルの構成 ・・・・・・・・・・ix
　このマニュアルの使い方 ・・・・・・・・・・x
　マニュアル記述について ・・・・・・・・・・x

第 1 章 はじめに

4D Compiler の使用 ・・・・・・・・・・1-2

インタプリタとコンパイラ ・・・・・・・・・・1-3
　インタプリタ ・・・・・・・・・・1-3
　コンパイラ ・・・・・・・・・・1-3

4D Compiler 製品の種類 ・・・・・・・・・・1-4
　4D Cimpiler（Macintosh 用、Windows 用）・・・・・・・・・・1-4
　4D Cimpiler Pro（Macintosh 用、Windows 用）・・・・・・・・・・1-4
　プラットフォーム エクステンション ・・・・・・・・・・1-4

4D Compiler と 4th Dimension ・・・・・・・・・・1-5
　対話型デバッグ ・・・・・・・・・・1-5

なぜデータベースをコンパイルするか？ ・・・・・・・・・・1-5
　実行速度 ・・・・・・・・・・1-5
　コードのチェック ・・・・・・・・・・1-7
　アプリケーションの保護 ・・・・・・・・・・1-8

4D Compiler の使用例 ・・・・・・・・・・1-8
　デモを試してみる ・・・・・・・・・・1-8
　データベースのコンパイル ・・・・・・・・・・1-9
　コンパイルしたデータベースを使う ・・・・・・・・・・1-12

自分のデータベースをコンパイルする ・・・・・・・・・・1-13

第 2 章 ユーザガイド

4D Compiler のメニュー ········ 2-2
新規 ········ 2-3
開く ········ 2-4
再コンパイル ········ 2-6
閉じる ········ 2-7
保存 ········ 2-7
新規保存 ········ 2-7
復帰 ········ 2-7
デフォルトプロジェクト ········ 2-7
終了 ········ 2-8

コンパイルオプション ········ 2-8
「メイン」ウインドウ ········ 2-9
コンパイルデータベース名 ········ 2-9
実行形式アプリケーションの作成 ········ 2-10
エラーファイル ········ 2-11
シンボルテーブル ········ 2-12
範囲チェック ········ 2-13
スクリプトマネージャ ········ 2-13
警告 ········ 2-14
プロセッサの種類 ········ 2-14
最適化 ········ 2-17
ローカル変数の初期化 ········ 2-18
タイプファイル ········ 2-18
コンパイルパス ········ 2-19
デフォルトボタンタイプ ········ 2-19
デフォルト数値タイプ ········ 2-20
デフォルト文字タイプ ········ 2-20
バージョン番号の自動生成 ········ 2-21
オプションについて ········ 2-21
バルーンヘルプ ········ 2-21
Windows のヘルプ ········ 2-22

コンパイルの開始 ········ 2-22

コンパイル ········ 2-22
データベース取り込み処理 ········ 2-23
変数タイプチェック処理 ········ 2-24
コンパイル処理 ········ 2-25
コンパイル後のデータベースを使用する ········ 2-27
ドラッグ＆ドロップ ········ 2-27

4th Dimension とともに 4D Compiler を使用 ········ 2-28
Power Macintosh 用に最適化された 4D Compiler ········ 2-28

第 3 章 診断ツール

シンボルテーブル 3-2
インタープロセス変数一覧とプロセス変数一覧 3-2
ローカル変数一覧 3-3
プロシージャおよび関数一覧 3-4
エラーファイル 3-4
メッセージのタイプ 3-4
エラーファイルの使い方 3-7
タイプファイル 3-10
範囲チェック 3-10
「範囲チェック」の使用法 3-11
異常診断 3-11

第 4 章 コンパイルの準備

変数と配列のデータタイプ 4-2
変数のタイプ 4-2
シンボルテーブル 4-3
4D Compiler による変数のタイプ設定 4-4
コンパイラ命令 4-5
コンパイラ命令が必要な場合 4-6
コードの最適化 4-7
コンパイル時間の短縮 4-7
実数と文字列を使う 4-8
インタプリタでコンパイラ命令を使う 4-8
コンパイラ命令をどこに記述するか 4-9
C_STRING コンパイラ命令 4-10
まとめ 4-11

第 5 章 タイプ設定ガイド

インタープロセス変数とプロセス変数 5-2
2 種類の用途による矛盾 5-2
用途とコンパイラ命令の矛盾 5-2
暗黙のタイプ変更による矛盾 5-3
2 つのコンパイラ命令の矛盾 5-3
ローカル変数 5-4

配列のタイプ矛盾 …… 5-5
配列要素のデータタイプの変更 …… 5-5
配列の次元数の変更 …… 5-5
文字配列 …… 5-6
暗黙のタイプ変更 …… 5-6
ローカル配列 …… 5-6
レイアウト変数 …… 5-7
数値タイプに設定されるレイアウト変数 …… 5-7
グラフ変数 …… 5-7
外部オブジェクト変数 …… 5-7
テキストタイプに設定されるレイアウト変数 …… 5-8
ポインタ …… 5-8
外部プロシージャ（エクステンション）…… 5-9
マルチプラットフォームのコンパイル …… 5-10
暗黙引数を受け取る外部ルーチン …… 5-10
引数の処理 …… 5-11
ポインタを使ってデータタイプの矛盾を避ける …… 5-11
引数の間接参照 …… 5-13
予約変数 …… 5-14
システム変数 …… 5-14
クイックレポート変数 …… 5-15

第 6 章 コマンドの説明

配列 …… 6-2
COPY ARRAY コマンド …… 6-2
SELECTION TO ARRAY コマンド …… 6-3
LIST TO ARRAY コマンド …… 6-3
DISTINCT VALUESコマンド …… 6-3
配列関連コマンドでのポインタの使用 …… 6-4
ローカル配列 …… 6-4
通信コマンド …… 6-4
データ入力 …… 6-5
例外処理コマンド …… 6-6
文書 …… 6-7
演算 …… 6-7
ブレーク処理 …… 6-7
文字列 …… 6-8

ストラクチャへのアクセス ・・・・・・・・・・6-8
変数、その他 ・・・・・・・・・・6-8
Undefined コマンド ・・・・・・・・・・6-9
SAVE VARIABLE コマンドと LOAD VARIABLE コマンド ・・・・・・・・・・6-9
CLEAR VARIABLE コマンド ・・・・・・・・・・6-10
Get Pointer コマンド ・・・・・・・・・・6-10
EXECUTE コマンド ・・・・・・・・・・6-12
TRACE コマンドと NO TRACE コマンド ・・・・・・・・・・6-12
各種のコマンドで使われるポインタ ・・・・・・・・・・6-13

第 7 章 最適化のためのヒント

コードに関するコメント ・・・・・・・・・・7-2
コンパイラ命令の使用によるコードの最適化 ・・・・・・・・・・7-2
数値変数 ・・・・・・・・・・7-2
文字 ・・・・・・・・・・7-3
その他のヒント ・・・・・・・・・・7-4
ポインタ ・・・・・・・・・・7-4
ローカル変数 ・・・・・・・・・・7-5

付録 A コンパイラのメッセージ

警告メッセージ ・・・・・・・・・・A-2
詳細警告メッセージ ・・・・・・・・・・A-3
エラーメッセージ ・・・・・・・・・・A-4
タイプチェック ・・・・・・・・・・A-4
シンタックス ・・・・・・・・・・A-6
引数 ・・・・・・・・・・A-8
演算子 ・・・・・・・・・・A-10
外部プロシージャ ・・・・・・・・・・A-10
総合エラー ・・・・・・・・・・A-10
範囲チェックメッセージ ・・・・・・・・・・A-13
コンパイラメッセージ ・・・・・・・・・・A-14

付録 B アプリケーションアイコンのカスタマイズ

Macintosh ・・・・・・・・・・B-1
Windows ・・・・・・・・・・B-3

付録 C　4D Compiler リソースのカスタマイズ

表記 ··C-2
ファイル種類（Macintosh のみ）··C-2

索引

4D Compiler および4D Compiler Proは、4th Dimension 用のコンパイラです。従来のコンパイラにはない、広範で詳細な警告とエラー診断メッセージが用意されています。

4D Compiler バージョン 2.5 は、Windows とMacintosh のいずれでも動作するプラットフォームに依存しないアプリケーションです。このマニュアルには、Windows と Macintosh 両方のバージョンの情報が収められています。

4D Compilerまたは4D Compiler Pro：

- データベースを体系的に解析し、詳細な警告メッセージやエラーメッセージを出力します。
- コンパイル後のデータベースの実行中にデータベースを動的にチェックします。
- 4th Dimensionによる対話形式のデバッグ機能をサポートします。
- プロシージャやスクリプトをコンパイルし、アセンブルコードを生成します。
- Motorola の 68000 プロセッサ（68000、68020/30/40、68LC040、680xx+6888x/68040）、Motorola PowerPC プロセッサ（PowerPC601/603/604）、PC プロセッサ（386/486、Pentium）用に最適化されたコードを生成することができます。
- コンパイルしたデータベースに 4D Engine のコピーをマージし、スタンドアロンの実行形式アプリケーションを作成します。
- データベースで使用した変数とコンパイラ命令すべてののリストを作成します。
- 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が設定できます。

注：作成する 4D Compiler それぞれの特性に応じて、コンパイラのプロセッサオプションをいくつか選択することもできるし、すべて選択することもできます。また、実行形式アプリケーションの生成は、4D Compiler Pro によって製作する場合にのみ有効な機能です。

コンパイル対象のデータベースは「ファイルオープン」ダイアログボックスによって開きます。それから後の処理はすべてコンパイラが行います。ストラクチャファイルを複製し、データベースを体系的に解析し、記述レポートと診断レポートを作成し、プロシージャのコピーを機械語に翻訳します。また、必要に応じて、エラーファイルも作成します。コンパイルしてできあがった新しいストラクチャファイルは、オリジナル（未コンパイル）のストラクチャファイルとまったく同じように使えます。ただし、「デザイン」モードは使用できません。コンパイル後のデータベースは処理速度が 3 倍から 1000 倍とめざましく向上し、処理によってはそれ以上に速くなる場合もあります。

4D Compiler は、数値演算コプロセッサがインストールされていればそれを考慮した上、マシンのマイクロプロセッサが受け付ける機械語コードを生成します。

注：4D Compiler バージョン 2 は、4th Dimension バージョン 3 と 4D Server バージョン 1で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 2 のデータベースを使う場合は、データベースを 4th Dimension バージョン 3 でオープンしてデータベースを変換し、4D Compiler バージョン 2 でコンパイルします。新しいバージョンが必要な場合は ACI までお問い合わせください。

このマニュアルについて

このマニュアルは、 4D Compiler または4D Compiler ProのWindows版およびMacintosh版のクロスマニュアルです。

マニュアルの構成

このマニュアルの章は、以下のような構成になっています。

- **第 1 章「はじめに」：**コンパイラの概要を紹介し、コンパイラとインタプリタの違いについて説明します。また、データベースをコンパイルする簡単な例題を示します。
- **第 2 章「ユーザガイド」：**4D Compiler の機能とオプションを説明します。
- **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コンパイラの 4 つのツール、“ 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」：**コンパイルを前提にデータベースを作成するときの規則について説明します。
- **第 5 章「タイプ設定ガイド」：**タイプの矛盾を引き起こすソースコードの代表的な例を紹介します。
- **第 6 章「コマンドの説明」：**ランゲージのコマンドの中でコンパイルだけに関連するものについて説明します。
- **第 7 章「最適化のためのヒント」：**コードの処理効率の最適化に役立つヒントをいくつか紹介します。

さらに、「付録 A」、「付録 B、「付録 C」には 4D Compiler の補足説明があります。「付録 A」は、コンパイラによって表示されるメッセージのリストです。各メッセージには、該当するコード例をあげています。「付録 B」では、ユーザアプリケーションのアイコンをカスタマイズする手順と、それに必要な情報を説明します。「付録 C」には Customizer Plus を使用した 4D Compiler の設定変更に関する情報があります。

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、4D Compiler の機能を理解してください。第 2 章で述べる機能とオプションを読んでいくうちに、実際のデータベースをコンパイルしてみたくなるかもしれません。しかし、1 回でコンパイルできることはまずないでしょう。第 4 章以降の章を読んでエラーメッセージとシンボルテーブルの使い方を学習してください。4D Compiler から出力されるエラーメッセージの意味は、付録 A を参照してください。

次に、第 4 章、第 5 章、第 6 章を読んで、コンパイルできるコードを記述するためにはどうすればよいのかを理解してください。最後に、第 7 章と付録 C のコンパイルコードを最適化するためのヒントを読んでください。

マニュアル記述について

4D Compiler および4D Compiler Proの Macintosh と Windows 両バージョンの内容はほとんど同じですが、必要な場合は、その違いについて説明されています。また、Macintosh と Windows 両環境の図が使われていますが、両方の内容に違いがある場合は、両バージョンの図が示されます。

このマニュアルも含めて、パッケージ内のマニュアルはすべて、理解の手助けになるよう一定の表記法を使用しています。次のような説明法が使われています。

注：このように強調された文は、4D Compiler をより効果的に使用するための注釈や近道を提供します。

このような注意書きは、重要な情報に対する注意を促しています。

このような警告は、データが失われる恐れがあることを示します。

この章では 4D Compiler の基本について説明します。以下のような内容になっています。

- 4D Compiler の使用法
- インタプリタ上とコンパイル後のプログラムの相違
- 4D Compiler 製品の種類
- 4D Compiler と 4th Dimension の対話型デバッガとの関係
- データベースにコンパイルが必要な理由
- チュートリアル

4D Compiler の使用

4D Compiler を活用するためには、以下の点に注意してください。

■ コーディングに不備がある場合は、インタプリタで正常に動作しているプロシージャも、コンパイルするとうまく動作しない場合があります。また、コーディングに不備がある場合はコンパイルもできません。エラーが検出されるとコンパイルが完了しないからです。

■ 4th Dimension のもとで動作するデータベースがすべて正常にコンパイルできるとは限りません。4th Dimension のインタプリタは、ある程度セマンティック（意味）やシンタックス（文法）に矛盾があっても処理できるからです。4D Compiler は、コンパイラとしては非常に寛容ですが、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基本的な違いによるものです。

4th Dimension とは違い、 4D Compiler では、各変数に割り当てるデータタイプを必ず 1 種類にする必要があります。つまり、変数ごとにデータタイプを 1 つ明確に割り当てなければなりません。たとえば、あるステートメントで実数として使われている変数が、他のステートメントでテキストとして使われると、コンパイラからエラーメッセージが出力されます。

プロセス変数やインタープロセス変数のタイプは、タイプ変更できません。ローカル変数の変数タイプは、その変数が使われるプロシージャやスクリプトの中ではタイプの変更できません。

データベースを作成する際の労力の多くは、これらの条件を満たすために費やされます。しかし、コンパイラには、データタイプの問題を見つけて、修正するための便利な診断ツールが用意されています。

ある種の変数についてはタイプの決定について際して選択肢があります。たとえば、数値変数のタイプには、実数、整数、倍長整数の 3 種類、文字列変数のタイプには、テキストと文字の 2 種類があります。これらの中から各変数に適したタイプを選択することにより、コンパイル後のデータベースのパフォーマンスを上げることができます。たとえば、倍長整数の変数を使えば、実数を使った時よりも速い速度でプロシージャを実行できます。同様に、テキスト変数よりも固定長の文字列を使った方が実行速度を速くできます。

インタプリタとコンパイラ

この節では、インタプリタとコンパイラの基本的な違いについて説明します。

コンピュータの内部で、コマンドは“0”と“1”だけを使って書かれています。マシンの心臓部であるマイクロプロセッサは、他の言語を理解できません。この2進言語を「機械語」と呼びます。

C、Pascal、BASIC、4th Dimensionなどの高級言語で書かれたプログラムは、まず、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機械語に翻訳されます。これを行うには、以下の2つの方法があります。

■ ステートメントを1つずつ翻訳しながら実行する方法。この方式を「インタプリタ」と呼びます。

■ プログラムを実行する前に、ステートメントを一括して翻訳する方法。この方式を「コンパイラ」と呼びます。

インタプリタ

インタプリタを使ってステートメントのまとまりを実行する場合、その処理は次のような段階に分けることができます。

■ インタプリタ用の言語で書かれたステートメントを1つ読み込む。

■ 読み込んだステートメントを機械語に翻訳する。

■ ステートメントを実行する。

インタプリタは、1つ1つのステートメントについてこのサイクルを実行します。

コンパイラ

コンパイラを使用したアプリケーションの動きは、インタプリタとは違います。実行に先だって、プログラム全体を翻訳します。その結果、機械語のステートメントを格納した新しいファイルを作成します。プログラムは1回翻訳するだけで何度も繰り返し実行できます。

この方法で不便なことは、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」コードに対して行う点です。その後、プログラム全体を再コンパイルする必要があります。プログラム開発者はコンパイルしたプログラムを直接、修正できません。このように、4D Compilerはオリジナルのデータベースには手を加えないので、変更後に再度コンパイルします。

コンパイル後のデータベースの利点は数々ありますが、この章の後の“データベースをコンパイルする意義”に説明があります。

4D Compiler 製品の種類

4D Compiler 製品には複数の種類があります。ここでは 4D Compiler と 4D Compiler Pro の相違について説明します。

4D Compiler（Macintosh 用、Windows 用）

4D Compiler からは、Macintosh 68000、Macintosh PPC、Windows 386/486、Windows Pentium を含めた、複数の異なるプラットフォーム用の機械語が生成されます。
4D Compiler による製品は、これらのプロセッサのいずれか、あるいはプロセッサすべてに対応することが可能です。4D Compiler は、4th Dimension または 4D Server とともに使用するコンパイル後のデータベースを作成します。

4D Compiler Pro（Macintosh 用、Windows 用）

4D Compiler Pro は 4D Compiler と同じ機能を持ち、さらに、実行形式のアプリケーションを作成することができ、このアプリケーションは無制限配付が可能です。4D Compiler Pro により、コンパイル後のデータベースを 4D Engine のコピーとマージし、スタンドアロンの実行形式アプリケーションを作成することができます。この4D Compiler Pro は、
4D SDK™製品の一部でもあります。

4D SDK は、4th Dimension アプリケーションの設計、コンパイル、開発のための完全な開発環境です。4D SDK は、4th Dimension、4D Insider、4D Engine、4D Compiler Pro といった複数のソフトウエア要素で構成されています。

プラットフォーム エクステンション

両方のプラットフォームで動作するデータベースをコンパイルするためには、プラットフォームエクステンションは必要ありません。4D Compiler、4D Compiler Proにはいずれも Macintosh環境、Windows環境の両用にコンパイルするためのオプションが含まれています。

4D Compiler と 4th Dimension

4D Compiler はグローバルプロシージャ、ファイル／レイアウトプロシージャ、スクリプトをすべてコンパイルし、オリジナルのストラクチャファイルとまったく同じように使用できる新しいストラクチャファイルを作成します。ユーザは、「ユーザ」モードと「ランタイム」モードを使用することができます。4D Compiler は、オリジナルのストラクチャファイルをそのまま残し、新しい（コンパイル後の）ストラクチャファイルを作成します。

4D Compiler は、データベースにエラーや曖昧な箇所を見つけると、多くの場合コンパイルできなくなります。エラーはテキストファイルとして出力されます。オリジナルの（未コンパイルの）データベースでエラーを修正して、再コンパイルする必要があります。

対話型デバッグ

4D Compiler のユニークな特徴は、エラーファイルを使用して対話形式でデバッグが行える点です。未コンパイルのデータベースとエラーファイルは4^{th} Dimension によって同時にオープンできます。4th Dimension のメニューコマンド「コンパイラエラー」を選ぶと、コンパイルエラー行が自動的に見つけ出され、反転表示され、エラーメッセージや警告メッセージが表示されます。

なぜデータベースをコンパイルするか？

コンパイルの最大の利点は、いうまでもなく実行速度が速くなることですが、その他にもコンパイルに直接関連のある次のような利点があります。

- 体系的なコードチェック
- データベースの保護

実行速度

コンパイル後のコードは、次のようなコンパイラの特性により実行速度が速くなます。

- 一度に全部を翻訳する、直接コード翻訳
- 変数とプロシージャ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 はコードを翻訳し、結果を実行形式のファイルに保存します。従って、インタプリタのようにステートメントを翻訳する必要がないので、その分、高速に動作します。

これを示す簡単な例を次に示します。50 回繰り返すループを考えてみましょう。

```
For (i ; 1 ; 50)
  `一連のステートメント
End for
```

インタプリタは、ループ中のステートメントをそれぞれ 50回翻訳します。コンパイルを行うと、ステートメント各々に対する翻訳のフェーズがなくなるため、ループ中の各ステートメントの 50回分の翻訳が省略できます。

もう 1 つ、例をあげましょう。データベースに Startup プロシージャがあるとします。このプロシージャはデータベースを起動すると一番最初に実行されます。ところが、コンパイルしてしまえば、この Startup プロシージャを翻訳するのに必要な時間を節約することができます。

以上たった 2つの例ですが、これらの利点は実際にすべてのプロシージャやスクリプトのステートメントにも当てはまります。

変数アドレスとプロシージャアドレスへの直接アクセス

インタプリタは、変数を名前でアクセスします。従って、4th Dimension は変数の値にアクセスするためには、変数の名前にアクセスする必要があります。

コンパイラは、各変数にアドレスを割り当て、コードに変数のアドレスを書き込みます。従って、そのアドレスに直接アクセスできます。

注：ディスクアクセスを必要とする処理は、ほとんど影響を受けません。ディスクのアクセス処理の実行速度は、コンピュータとハードディスクの間の転送速度に制限されるからです。

コメントは翻訳されないのでコンパイル後のデータベースには存在しません。ですから、コメントはコンパイル後のデータベースの実行時間に影響を与えません。

次の表は、「コンパイル」モードと「インタプリタ」モードの実行速度を示したものです。これらのテストは Power Macintosh 6100/66 で行いました。

シーケンス	インタプリタ	コンパイラ	高速化比
単純な For ループを 100万回繰り返す：			
For($i; 1; 1000000) End for	9分50秒	1秒	約 600 倍の速さ
メモリ上にキャッシュされた 10,000 件のレコードセレクションに対する文字列操作：			

For($i ; 1; 10000)
$x:=Substring ([File1]first ; 1 ; 5)
NEXT RECORD ([File1)
End for
	36秒	3秒	12 倍の速さ

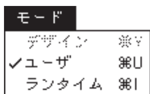
コードのチェック

コンパイラは、データベースをコンパイルする前にプロシージャとスクリプトのチェックを行います。ユーザの書いたコードを系統的にチェックし、曖昧箇所を指摘します。一方、4th Dimension はこれらの処理をプロシージャの実行時に行います。これらの間には、重要な違いがあります。コンパイラは、実際に実行されるか否かに関係なくコードすべてを系統的にチェックします。

たとえば、プロシージャの中に実行されるべきステートメント以外に、さまざまな判定処理があるとします。その判定の数が非常に多い場合、すべてのケースをテストすることは不可能です。テストしていないケースにシンタックスエラーがあっても、インタプリタの場合、エンドユーザがそのケースを実行するまでわかりません。

データベースをコンパイルする場合、データベース全体について 1ステートメントずつ解析されます。何か異常があればそれを検知し、エラーメッセージや警告メッセージを出力します。メッセージはエラーファイルに書き込まれます。このファイルを 4th Dimension から開いて、対話型デバッグに使用できます。対話型デバッグについては第 3 章の「4th Dimensionでエラーファイルを対話的に使う方法」で詳しく説明します。

アプリケーションの保護



コンパイルしたデータベースは、オリジナルのデータベースのコピーです。唯一の違いは、コンパイルしたデータベースから「デザイン」モードへアクセスできない点です。これには、次のような利点があります。

- 意図的にであれ、間違いであれ、データベース構造を変更できない。
- 内容を見たり調べることができないので、プロシージャが保護される。

4D Compiler の使用例

ここで、コンパイルによる処理速度の向上を示す簡単な例を紹介します。この例題を行うためには、4D Compiler が必要なので、インストールを済ませていない場合はインストールしてください。

デモを試してみる

ディスケットの“デモ”フォルダ／ディレクトリには 4th Dimension の小さなデータベースが入っています。4D Compiler を簡単に紹介するためのデータベースです。

1. 4th Dimensionを起動して、「デモ」データベースを開きます。

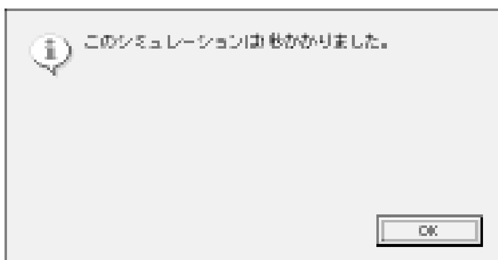
データベースは、「ランタイム」モードで開きます。

2. 「ファイル」メニューから「デモンストレーション」を選択します。

レコードのリストが表示されます。このリストの要素を処理するプロシージャを実行させることができます。

3. 「シミュレーション」ボタンをクリックします。

アプリケーションは、データベース内の値をそれぞれ処理するプロシージャを実行します。処理が終わると、所要時間がアラートボックスに表示されます。



注：処理時間は 使用するマシンによって異なります。

4. 「OK」ボタンをクリックして「ファイル」メニューから「終了」を選択します。

4th Dimension は終了し、オペレーティングシステムに戻ります。

データベースのコンパイル

ここでは、データベースをコンパイルし、再び実行速度を測定してみましょう。

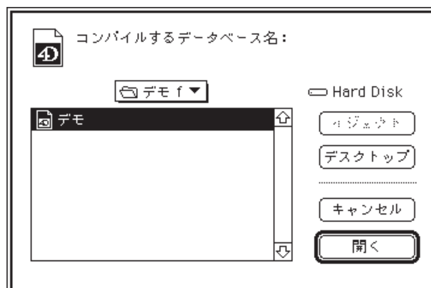
1. プログラムのアイコンをダブルクリックして 4D Compiler 起動します。

2. 「ファイル」メニューから「新規」を選択します。

「ファイルオープン」ダイアログボックスが表示されます。

3. “ デモ ” ディレクトリまたは “ デモ f ” フォルダ内のデータベースを開きます。

Windows版



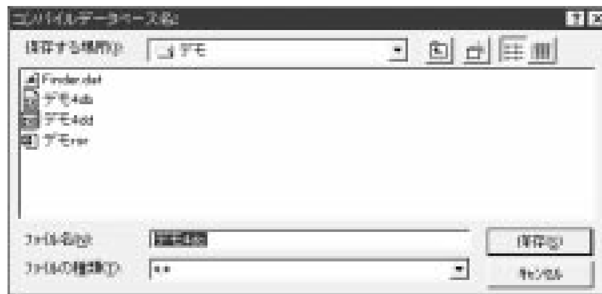
MacOS版

「オプション」ウインドウが表示されます。オプションについては、第 2 章の「オプションウインドウ」の節を参照してください。

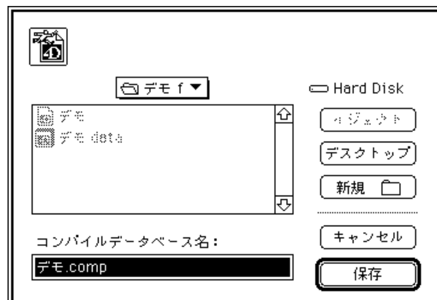
この例題では、オプションを使う必要はありません。

4.「OK」ボタンをクリックします。

「ファイル保存」ダイアログボックスが表示されます。



Windows版



MacOS版

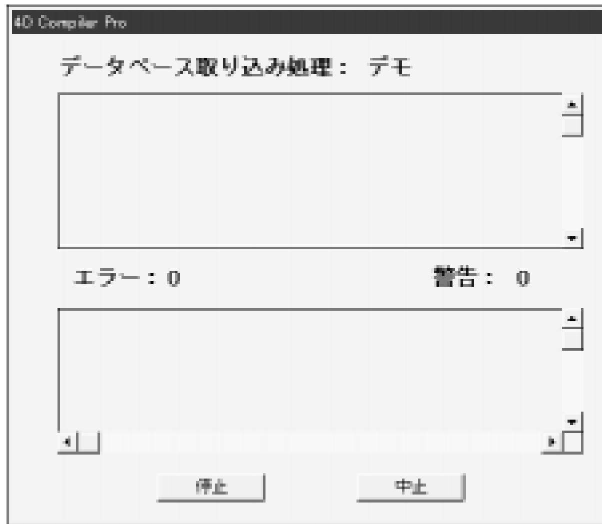
コンパイル後のデータベース名を指定しないと、コンパイルは行われません。デフォルトの名前は、Macintosh では “ デモ.comp ”、Windows では “ デモ.4CD ” です。

5. 「保存」ボタンをクリックします。

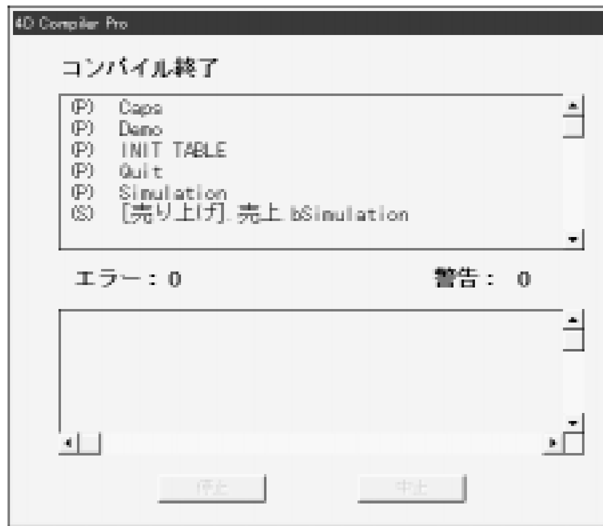
プロジェクトを保存するかどうか聞かれます。プロジェクトとは「オプション」ウインドウで設定された内容を再利用のために保存しておくためのファイルです。プロジェクトは実際に作業する上で非常に便利です。この例では、一時的なコンパイルを行うだけですからプロジェクトを保存する必要はありません。

6. 「いいえ」ボタンをクリックします。

コンパイル処理が始まります。コンパイル実行中は 4D Compiler ウィンドウが表示されます。コンパイルの進行状況が示され、必要に応じてエラーや警告メッセージが表示されます。



コンパイルが終了すると、“ コンパイル終了 ” というメッセージがウィンドウの上端に表示されます。



7.「ファイル」メニューから「終了」を選択します。

コンパイルしたデータベースを使う

4D Compiler 2.5 によって生成されたコンパイル後のデータベースを使うには、バージョン 3.5（またはそれ以降）の 4th Dimension が必要です。

1. 4th Dimension を起動し、ストラクチャファイル “ デモ.4CD ” または “ デモ.comp ” を開きます。

コンパイル後のデータベースのデータファイル名を求められます。

2. “ デモ.data ” ファイルを選択します。

コンパイル済みのデータベースは、未コンパイルのデータベースと同じように使用できます。

3. 「デモを試してみる」で説明した手順でパフォーマンスを比較してみてください。

実行速度が驚くほど速くなったことが実感できるでしょう。

自分のデータベースをコンパイルする

以上、この短い練習を終えてみて、自分のデータベースをコンパイルしてみたいと思われたことでしょう。

残念ながら、手持ちのデータベースを 1 回でコンパイルできることはほとんどありません。4th Dimension のインタプリタはコンパイラよりも制限が少ないので、インタプリタで問題なく動作するデータベースでも、1 回ではコンパイルできないことがほとんどです。

4D Compiler で数十ものタイプの矛盾が検知されることがありますが、コンパイラの対話型デバッグ機能を使えば簡単に修正すことができます。

次の第 2 章は、コンパイラのユーザーガイドです。第 2 章以降で、デバッグの手順とコンパイラ用に最適化したコードの書き方について説明します。

この章では、4D Compiler の使い方について次の 4 つの項目に分けて説明します。

■ 4D Compiler のメニュー

■ 「オプション」ウインドウで使用できる機能とオプション

■ 4th Dimension とコンパイラの同時使用

■ コンパイル処理

4D Compiler のオプションは、“ プロジェクト ” の概念に基づいています。プロジェクトはユーザが設定したオプションの集合で、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と考えてください。

コンパイル時には、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、再コンパイルを素早く簡単に行うことができます。

また、プロジェクトによって、以下のことが可能になります。

■ コンパイル後のストラクチャファイルの名前の保存

■ 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、保存

■ 「実行形式アプリケーションの作成」、「範囲チェック」、「詳細警告」、「スクリプトマネージャ対応」、「プロセッサの種類」等、各種オプションの指定

■ 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

プロジェクトを使ってデータベースを再コンパイルすると、プロジェクトに設定したファイルが自動的に生成され、指定したオプションが使われます。プロジェクトで指定するオプションの詳細については、この章の「コンパイルオプション」を参照してください。

コンパイル対象のデータベースは、4th Dimension とコンパイラの両方から同時に開くことができます。コンパイルのたびに、いちいち 4th Dimension を終了せずに作業が続けられます。

4th Dimension で開いているデータベースをコンパイルする場合は、「デザイン」モードを抜け、「ユーザ」モードまたは「ランタイム」モードにした後、コンパイラに切り替えます。4th Dimension でエラーファイルを使っていた場合は、「デザイン」モードの「モード」メニューから「エラーファイル参照終了」を選択してファイルを閉じます。

コンパイル中はデータベースを使用できません。コンパイルが完了した後は、
4th Dimension に切り替えて、データベースに戻れます。

4D Compiler をバックグラウンドで走らせることもできます。つまり、4D Compiler で作業している間に他のアプリケーションを使うことができます。「メイン」ウインドウにいるときもアプリケーションに切り替えられます。同時に開けるアプリケーションの数は使用マシンの RAM サイズに依存します。

4D Compiler のメニュー

Macintosh版の4D Compiler のメニューバーには、以下の 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」というメニューアイテムがあります。このアイテムは、コンパイラに関する情報を表示します。

「編集」メニューは、Macintosh のアプリケーションに必須のメニューです。4D Compiler では使用しませんが、デスクアクセサリ用に提供されます。

Windows 上でも、4D Compiler のメニューバーには 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、Windows のアプリケーションに必須のメニューです。4D Compiler では使用しません。

「ヘルプ」メニューにより、4D Compiler 用の Windows ヘルプファイルにアクセスできます。

4D Compiler で使用するのは、「ファイル」メニューだけです。「ファイル」メニューには、下図のように 9 つのメニューコマンドがあります。

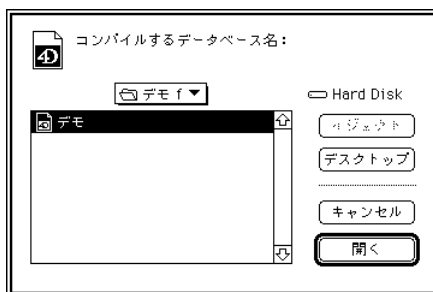
ファイル
新規... ⌘N
開く... ⌘O
再コンパイル ⌘R
閉じる
保存 ⌘S
名前を変えて保存...
元に戻す
デフォルトプロジェクト...
終了 ⌘Q

新規

「新規」メニューコマンドを使用して、新しいプロジェクトを作成します。プロジェクトにより、コンパイル対象のデータベースと、コンパイル時のオプションを指定します。

このアイテムを選択すると、プラットフォームに応じて標準の「ファイルオープン」ダイアログボックスが表示されます。このダイアログボックスを使ってコンパイルするデータベースを開きます。

Windows版



Macintosh版

データベースを開くと、「メイン」ウインドウが表示されます。

「オプション」ウインドウの 10 個のアイコンはそれぞれ 4D Compiler の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」で詳しく説明します。

開く

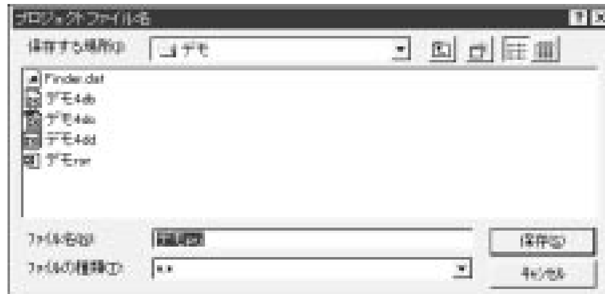
「開く」メニューコマンドは、既存のプロジェクトを開く場合に使用します。プロジェクトを使うと、コンパイル情報がプロジェクトに保存されるため、迅速に同じデータベースのコンパイルを繰り返し行うことができます。

プロジェクトの概念

例題で見たように、さまざまなコンパイルオプションがあります。オプションを選択し終えると、次のようなダイアログボックスが表示されます。

プロジェクトを使用すると、オプションの選択結果を保存できます。オプションを保存しておくと、同じコンパイルオプション設定でデータベースを簡単に繰り返しコンパイルできます。

「はい」ボタンをクリックすると、使用しているプラットオフォームに応じた標準の「ファイル保存」ダイアログが表示されます。



プロジェクトファイルに名前を付けたら、「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「ファイルオープン」ダイアログボックスが表示されます。

プロジェクトを選択すると、そのプロジェクトの「オプション」ウインドウが表示されます。プロジェクトに関する設定情報がすべて表示されます。

既存のプロジェクトがないときに「ファイル」メニューから「開く」を選択した場合は、「新規」ボタンを使って、コンパイル対象のデータベースを開くことができます。

プロジェクトを使ったデータベースの再コンパイル

既存のプロジェクトを開くと、そのプロジェクトに設定されたオプションがすべて使われます。これらのオプションの中には、シンボルテーブル、エラーファイルなど、コンパイル結果の情報ファイルの作成を指示するものもあります。再コンパイルすると、これらのファイルも自動的に再作成されます。前回のコンパイルで作成したファイルを残す場合は、再コンパイルの前に、他のフォルダまたはディレクトリに保存してください。

これらのファイルに関する詳細は、この章の「コンパイルオプション」を参照してください。

再コンパイル

「再コンパイル」メニューコマンドにより、デバッグ中のデータベースを再コンパイルします。「再コンパイル」メニューコマンドは、4th Dimension の同時使用による対話型デバッグのためのコマンドです。4th Dimension でエラー修正を終了した後 4D Compiler に切り替え、「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。対話型デバッグに関する詳細は、第 3 章の「4th Dimension でエラーファイルを対話型デバッグに使う方法」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、いずれの場合も、コンパイル対象のデータベースがパスワードで保護されていると、データベースを選択した際にデザイナのパスワード入力を求められます。

閉じる

「閉じる」メニューコマンドを使用して、プロジェクトを閉じます。プロジェクトを保存せずに「閉じる」を選択すると、「ファイル保存」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

「保存」メニューコマンドにより、プロジェクトを保存します。前回保存したバージョンが現在のプロジェクトで置き換えられます。

新規保存

「新規保存」メニューコマンドは、新しいプロジェクトを保存する場合や、現在のプロジェクトを別の名前で保存する場合に使用します。このメニューコマンドにより、「ファイル保存」ダイアログボックスが表示されるので、現在のプロジェクトの新しい名前を入力します。

復帰

「復帰」メニューコマンドは、現在のプロジェクトを前回保存したバージョンに戻す場合に使用します。

デフォルトプロジェクト

このメニューコマンドにより、新規プロジェクトに適用するデフォルトのオプションを定義することができます。

「ファイル」メニューから「デフォルトプロジェクト」を選択します。「メイン」ウインドウが表示されます。ここで定義したプロジェクトは、Macintosh 上ではシステムフォルダ下の「初期設定」／「ACI」フォルダ内の 4D Compiler Prefs ファイルに保存され、Windows 上では「Windows」ディレクトリ下の「Aci」ディレクトリ内の 4D Comp.PRF ファイルに保存されます。新規のプロジェクトでは必ず、デフォルトプロジェクトとして選択したオプションが表示されます。

デフォルトプロジェクトのオプションは、データベースの通常のプロジェクトのオプションと同じようにして選択します。唯一の違いは、コンパイル結果情報のファイル（エラーファイル、シンボルテーブル、コンパイル後ストラクチャなど）の保存場所を指定するための「ファイル作成」ダイアログボックスではなく、データベース名の後に続けるデフォルトの拡張子用の入力エリアが表示される点です。

1.「コンパイルデータベース名」ボタンをクリックします。

入力する場所が表示されます。

MacOS版

Windows版

2. 拡張子を入力します。

次に作成されるコンパイル後のデータベースには、デフォルトで、指定した拡張子がデータベース名の後に付加されます。

このデフォルトの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選択しておかねばならないわけではありません。デフォルトの設定で新しいプロジェクトを開いた後で、スクリプトマネージャ対応にできます。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自体には保存されません。

デフォルトプロジェクトを変更する場合は、「ファイル」メニューから「デフォルトプロジェクト」を選択します。プロジェクトの設定を行い、「OK」ボタンをクリックします。

終了

「終了」メニューコマンドにより、4D Compiler を終了します。

コンパイルオプション

最初のウインドウにはメインのオプションが表示されます。「次...」ボタンをクリックして表示する次のウインドウは、「オプション」ウインドウです。ここでは、データベースのコンパイルに関する微調整を設定することができます。

「メイン」ウインドウ

このウインドウには 10 個のアイコンがあります。各アイコンは、1 種類のオプションを表し、選択したり選択を解除することができます。4D Compiler による製品のために、これらのオプションのいずれか、あるいはすべてを利用することができます。アイコンをクリックして、アイコンで表現されている動作を選択します。再度クリックすると、前の動作はキャンセルされます。アイコンの表示には現在の設定が反映されます。

以下で、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

このオプションを使用して、コンパイル後のストラクチャファイルの名前を設定します。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 ” の後に続けて “ .comp ” ／ “ .4DC ” またはデフォルトプロジェクトで設定した拡張子を付加したものになっています。

この名前は変更できます。「保存」／「OK」ボタンをクリックすると、コンパイル後のストラクチャファイル名が保存されます。コンパイル後のデータベースに名前を付けると、「メイン」ウインドウでアイコンの右側にその名前が表示されます。

コンパイル後のデータベース名

オリジナルのデータベースとコンパイルしたデータベースを同じディレクトリ（フォルダ）に作る場合、異なる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オリジナルと同じ名前を指定しようとすると、標準の「ファイル置き換え」ダイアログボックスが表示されます。オリジナルのデータベースはデバッグのために必要なので、上書きしないでください。ダイアログボックスの「はい」ボタンをクリックしても、4D Compiler からエラーメッセージが表示され、オリジナルデータベースへの上書きはできません。

コンパイルしたデータベースは、オリジナルデータベースのコピーです。オリジナルとまったく同じように動作しますが、「デザイン」モードは使用できません。「デザイン」モードにアクセスする必要があるので、オリジナルのファイルへの上書きはできないようになっています。

実行形式アプリケーションの作成

4D Compiler Pro をご使用の場合はこのオプションにより、コンパイル後のデータベースに 4D Engine をマージし、スタンドアロンの実行形式アプリケーションを作成することができます。このオプションを使う前に、4D Engine をハードディスクにコピーしてください。

4D Engine は 4D Compiler Pro とともにインストールされます。

このオプションをクリックすると、「ファイルオープン」ダイアログボックスが表示されます。

4D Engine を選択し、「開く」ボタンをクリックします。「メイン」ウインドウに戻ると、選択した 4D Engine の名前が表示され、このオプションが選択されたことを示します。

コンパイル時に、4D Engine とコンパイル後のストラクチャから実行形式のアプリケーションが生成されます。Macintosh 上では 4D Engine とコンパイル後のストラクチャをマージします。Windows 上では、「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 種類のファイルが生成されます。

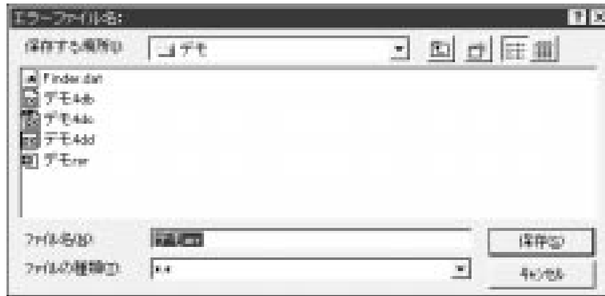
実行形式のアプリケーションには、デフォルトで共通なアプリケーションアイコンが割り当てられます。このアイコンはカスタマイズ可能で、その方法については「付録 B」に説明があります。

エラーファイル

このオプションを使用すると、タイプチェックやコンパイル中に見つかったエラーと警告のリストがテキストファイルの形で出力されます。このファイルは 4th Dimension から開いて対話型デバッグに利用できます。

また、エラーファイルにはコンパイル時の警告と詳細警告も出力されます。詳細については、この章の「警告」の節を参照してください。

「エラーファイル」アイコンをクリックすると、標準の「ファイル作成」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 ” の後に続けて “ .err ”” または “ 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付加されたものになっています。



対話型デバッグを行う場合は、デフォルトの名前を使用します。「保存」または「OK」ボタンをクリックすると、“ エラーファイル ” の名前が「メイン」ウインドウに表示されます。

エラーファイルには 2 通りの使い方があります。

■ テキストファイルとして、コンパイラによって生成されるメッセージの記録を残すことができます。

■ 4th Dimension では、データベースのデバッグに使用することができます。

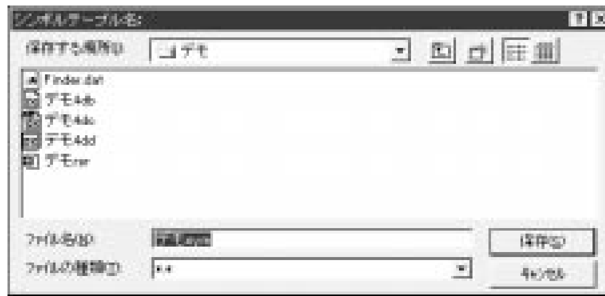
エラーファイルに関する詳細は、第 3 章と「付録 A」のエラーメッセージリストを参照してください。

シンボルテーブル

このオプションを指定すると、データベースのシンボルテーブルがテキストファイルで出力されます。このファイルには、コンパイル対象のデータベース内のオブジェクトに関する情報がすべての入っています。プロセス変数や、インタープロセス変数、ローカル変数のデータタイプ、およびデータタイプを定義したプロシージャの名前が含まれます。

さらに、シンボルテーブルには、すべてのプロシージャと関数、引数、また、関数の場合には返り値のデータタイプも入っています。

「シンボルテーブル」アイコンをクリックすると、標準の「ファイル保存」ダイアログボックスが表示されます。“ シンボルテーブル ” のデフォルト名は、Windows版では “ データベース名 ” の後に続けて “ .SYM ”、Macintosh版では “ .symb ”、または「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付加されたものになっています。



“ シンボルテーブル ” のファイル名は任意の名前に変更できます。「保存」／「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」ウインドウに表示されます。

シンボルテーブルに関する詳細は、第 3 章を参照してください。

範囲チェック

「範囲チェック」を指定すると、強力な診断機能を利用することができます。この機能はコンパイル後のデータベースの実行中にプロシージャの状態を監視します。

「範囲チェック」オプションは、コンパイル中にはメッセージを表示しません。この “ 範囲チェック ” の効果が現れるのは、コンパイルしたデータベースの実行中です。このオプションは、実際にデータベースの実行時に表面化する問題を見つけるために提供されています。範囲チェックに関する詳細は、第 3 章を参照してください。

スクリプトマネージャ

スクリプトマネージャ対応バージョンの 4th Dimension とともにデータベースを使う場合は、必ずこのオプションを指定してください。

スクリプトマネージャによって、日本語、中国語、アラビア語、ヘブライ語など、アルファベットと Roman 以外の文字が混在する環境でプログラムが機能できるようになります。スクリプトマネージャの管理下で動作する 4th Dimension でデータベースを使用する場合は、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」を選択しないでください。データベースの動きが多少遅くなります。（日本語バージョンの場合は、必ず選択してください。）

警告

このオプションによって、エラーファイルに詳細な診断メッセージが出力されます。この機能は、以下のような場合に特に役に立ちます。

- コードはコンパイルできているが、コードの質をさらに高めたい場合。
- コンパイラからは完全に評価できないようなステートメントがある場合。これには、コンパイルエラーが起きていない場合も含みます。コンパイラコードをもとに類推を行いますが、これらの推定が正しいかどうかを確かめようとする場合です。

このオプションが選択されていないと、警告メッセージは表示されません。「標準」オプションが選択されている場合、コンパイラはエラーメッセージの代わりに簡単なメッセージを表示します。こうしたメッセージは自動的にエラーファイルに書き出されます。「詳細」オプションが選択されていると、コンパイラは簡単かつ詳細な警告を生成します。

このオプションに関する詳細は、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の種類を選択できます。データベースの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動かす Macintosh または PC のモデルを選択してください。実際に使用するプロセッサを特定して、コンパイルオプションを適切に選択すれば、プロセッサの特性を活かすことができます。選択肢は次のようになっています。

- Motorola 68xxx（Macintosh）
- Motorola PowerPC（Power Macintosh）
- 386/486 および Pentium（Windows）

注：4D Compiler 製品の中には、コンパイルオプションすべてを使用できないものもあります。

Motorola 68xxx（Macintosh）

68000 用にコンパイルされたデータベースは、すべての Macintosh 上で動作します。68020 と 68030 マイクロプロセッサは、基本的に 68000 より高性能です。3 番目のオプション（「68020/68030 + 68LC040」または「680x0 + 6888x / 68040」）を選択すると、これらの機能を活用することができます。

注：68000 用にコンパイルしたデータベースは、68000 を搭載した Macintosh と 68020 や 68030、68040 を搭載した Macintosh でも動作します。しかし、68020/30/40 用にコンパイルしたデータベースは 68000 を搭載した Macintosh では動作しません。この場合、コンパイル後のデータベースを起動するとエラーメッセージが表示されます。

68020 または 68030 プロセッサを搭載した Macintosh がコンパイルの対象機種である場合数値演算用プロセッサ（6888x）はあってもなくても構いません。68040 はボード上に数値演算プロセッサを備えていますが、68LC040 は持っていません。オプションの中から適切なものを選択してください。

対象機種		
CPU	**コプロセッサ？**	**コンパイルオプション**
68000	**使用せず**	**68000**
68020 または 68030	**なし**	**68020/68030 + 68LC040**
68020 または 68030	**あり（6888x）**	**680x0 + 6888x / 68040**
68LC040	**使用せず**	**68020/68030 + 68LC040**
68040	**あり（ボード上）**	**680x0 + 6888x / 68040**

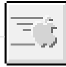
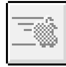
コンパイル後のデータベースを、複数モデルの Macintosh 上で動かす場合（シングルユーザ）や、クライアントワークステーションに複数モデルの Macintosh を使用する場合は、すべてのMacintosh 上でデータベースが動作できるように、コンパイルオプションとしては最下位のものを選択する必要があります。たとえば、数値演算用のコプロセッサを持たない 68020/030 の Macintosh クライアントが 1 台でも存在していれば、コプロセッサ有りのオプションは指定できません。「68020/030 + 68LC040」オプションを使用します。

68000 以外のオプションを選択した場合は、32K 以上のプロセス変数と 32K 以上のインタープロセス変数を使用できます。ただし、プロセス変数の設定には注意が必要です。プロセス変数がどのプロシージャで使われているか、コンパイル時に 4D Compiler からは判断できません。プロセス変数には、プロセスごとに異なる値を持たせることができます。そこで、プロセスが新しく開始されるごとに全プロセス変数の複製が作られます。

たとえば、プロセス変数が 50K あるとすると、プロセスごとに 50K の追加メモリが必要になります。

また、プロセス変数に使われるメモリスペースは、プロセスのスタックサイズにはまったく関連がないことを覚えておいてください。

Motorola PowerPC（Power Macintosh）

601/603/604   なし

このバージョンで生成された PPC コードは、Power Macintosh™ コンピュータに使用されている PowerPC プロセッサ 601、603、604 に対応しています。

このオプションを選択してコンパイルすると、動作環境は完全に Power PC 対応に最適化されます。4D Compiler によって生成されたネイティブコードは、PPC 対応（PPC ネイティブまたは FAT）の 4th Dimension、4D Client、4D Engine で実行させることができます。4D 環境全体が、Power Macintosh の処理速度で動作します。

技術上の理由から、Power Macintosh でプロシージャあるいはスクリプトから、New Process 関数を呼ぶ場合、プロセスに割り当てるスタックサイズを指示する数値引数には偶数を使用する必要があります

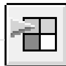
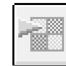
たとえば、次のようにしてください。

<>CustProc:=**New process** ("mDoCust" ; 32000 ; "Customers")

次のような指定はできません。

<>CustProc:=**New process** ("mDoCust" ; 32767 ; "Customers")

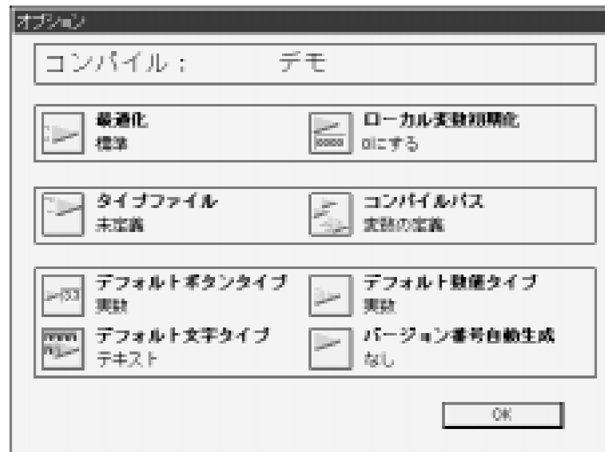
PC（Windows)

386/486  Pentium  なし

コンパイル後のデータベースを4D ServerとさまざまなPCクライアントとともに使用する場合は、最も高性能なコンパイルオプションを選択することができます。Pentiumと486のクライアントがある場合は、Pentiumを選択します。

「オプション」ウインドウ

「オプション」ウインドウは、「メイン」ウインドウの下にある「次...」ボタンをクリックすると開きます。「オプション」ウインドウには8個のアイコンがあります。



以下に、各オプションの詳細を示します。

最適化

4D Compiler では、2種類のコード生成方法のいずれかを選びます。

■ 「標準」の生成方法は、コンパイル時間が速く、通常のコードを生成します。

■ 「最適化」の生成方法は、最適化されたコンパクトなコードを生成します。この方法は生成処理に要する時間が長くなりますが、データベースの実行は速くなります。このオプションは、データベースの最終版に適用することをおすすめします。

ローカル変数の初期化

このオプションには、選択肢が 3 種類あります。プロシージャの実行時には、ローカル変数用の場所が確保されます。これらのローカル変数はプロシージャの終了時に消去されます。

プロシージャを開始する際に、コンパイル後のコードは次のいずれかの動作をとることができます。

■ ローカル変数を作成し、各変数をデフォルトでヌル値（文字列には空の文字列、数値には 0）に初期化する。変数を初期化していない場合は、このオプションを使用します。

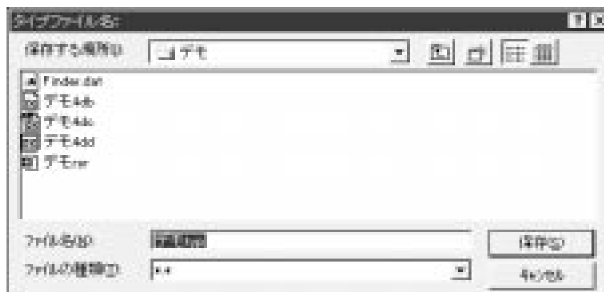
■ 変数を作成し、ランダムな値で初期化する。このオプションで、忘れたままになっているローカル変数を特定することができます。ローカル変数は常に同じ値のランダム値で初期化されす。このオプションはデバッグ用にのみ使用してください。

■ 変数を作成し、デフォルトでは初期化しない。コンパイラによって変数が初期化されないので、プロシージャの実行速度が速くなります。変数を正しく初期化している場合にこのオプションを使用すると、実行時間を節約することができます。

製品の最終版には、1 番目か 3 番目のオプションを使用します。

タイプファイル

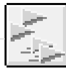
2 番目のオプションを選ぶと、変数をすべて定義するコンパイラ命令の入ったファイルが生成されます。データベースにこれらの定義を利用する場合は、コンパイラ命令をひとつあるいは複数のプロシージャにコピーします。アイコンをクリックすると、「新規保存」ダイアログボックスが表示されます。

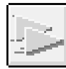


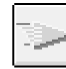
このファイルのデフォルト名は、Windows版では、“ データベース名 ” の後に続けて “ .TYP ”、Macintosh版では “ .type ”、またはデフォルトプロジェクトで設定した拡張子を追加したものになります。

コンパイラ命令の配置に関する詳細は、第 4 章を参照してください。

コンパイルパス







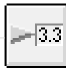
■ **すべての変数の定義：**この場合、4D Compiler はコンパイルに必要な段階をすべて実行します。

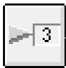
■ **プロセス変数とインタープロセス変数の定義：**このオプションは、プロセス変数とインタープロセス変数すべてがタイプ設定されている場合にのみ指定できます。このタイプ設定は、開発者が行うか、またはタイプファイルの内容をデータベースにペーストして行います。このオプションが設定されていると、タイプ付けのフェーズが省略されるので、コンパイル速度が速くなります。コンパイルは、データベースのコピー、コンパイルパス、およびファイルの生成の順に行われます。

■ **変数の定義：**このオプションは、プロセス変数、インタープロセス変数、ローカル変数すべてのタイプが設定されている場合に設定します。コンパイル速度はさらに向上します。このオプションによって、データベース作成時間は短縮されますが、生成されるコードには影響しません。

注：コンパイルパスを変更すると、コンパイルに要する時間を短縮できますが、パスの変更には注意が必要です。タイプ付けを省略した場合、コンパイラで検出される警告の数が減るからです。

デフォルトボタンタイプ





コンパイラは、タイプが設定されていない変数をデフォルトで最も範囲が広いタイプに設定します。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。データベース内のコンパイラ命令は、このオプションに優先します。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

■ ボタン

■ ハイライトボタン

■ 透明ボタン

■ ラジオボタン

■ ラジオピクチャ

このオプションは、「倍長整数」オプションを選ぶとデータベースの実行を最適化します。

デフォルト数値タイプ

コンパイラは、タイプなし変数のタイプをデフォルトで最も範囲が広いタイプに設定します。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」にすることができます。データベース内のコンパイラ命令は、このオプションより優先します。「倍長整数」オプションを選ぶと、このオプションにより、データベースの実行は最適化されます。

デフォルト文字タイプ

コンパイラは、タイプなし変数のタイプをデフォルトで最も範囲が広いタイプに設定します。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」にすることができます。データベース内のコンパイラ命令は、このオプションより優先します。

デフォルトの文字タイプを「固定長文字列」にすると、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。このオプションにより、データベースの実行は最適化されます。

タイプに設定には 3 種類の基準があり、次に示す優先順位で変数のタイプを決定します。

■ コンパイラ命令はすべてに優先

■ 「デフォルトタイプ」オプション

■ 未定義で、かつデフォルトのタイプも持たない変数には適切なタイプの割り当てを試みる

バージョン番号の自動生成

このオプションを選択すると、コンパイル後のデータベースには自動的にバージョン番号が振られます。この番号はプロジェクトに保存され、コンパイルするたびに更新されます。

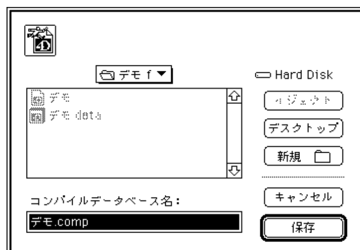
Macintosh では、“ vers ” タイプリソースが存在しない場合、コンパイラによって新規に作成されます。このリソースは、ファインダ上の「情報を見る」メニューコマンドで使われます。このリソースがすでに存在している場合は、4D Compiler によって修正されます。

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

このオプションを選択すると、入力エリアが現れ、コンパイラがデータベースに付ける番号を入力したり修正することができます。

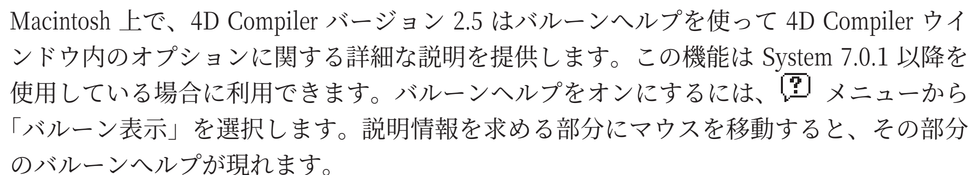
オプションについて

必須のコンパイルオプションはありません。オプションを設定せずにコンパイルを始めることができます。この場合は、「OK」ボタンをクリックすると、「ファイル保存」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の名前を入力するよう求められます。



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できます。コンパイルを開始した時に、コンパイル後のデータベースの名前の入力するよう求められます。

バルーンヘルプ



Macintosh 上で、4D Compiler バージョン 2.5 はバルーンヘルプを使って 4D Compiler ウインドウ内のオプションに関する詳細な説明を提供します。この機能は System 7.0.1 以降を使用している場合に利用できます。バルーンヘルプをオンにするには、? メニューから「バルーン表示」を選択します。説明情報を求める部分にマウスを移動すると、その部分のバルーンヘルプが現れます。

Windows のヘルプ

Windows 上で、4D Compiler には外部ヘルプが含まれており、4D Compilerの「ヘルプ」メニューによってアクセスすることができます。

コンパイルの開始

コンパイルオプションの選択が完了したら、「メイン」ウインドウの「OK」ボタンをクリックしてコンパイルを開始します。以降、コンパイラによって自動的に処理が行われます。他の作業は必要ありません。

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、デザイナのパスワードの入力を求められます。パスワードで保護されたデータベースをコパイルするには、“ デザイナ ” または “ 管理者 ” を選択し、正しいパスワードを選択して「OK」ボタンをクリックすると、データベースのコンパイルが開始されます。

外部プロシージャを発見できない場合、4D Compiler はコンパイル処理を中断し、格納場所の入力を求めます。この点ついては、次節の「コンパイル」を参照してください。

「オプション」ウインドウの「キャンセル」ボタンをクリックすると、現在のプロジェクトが閉じられ、プロジェクトが保存されていない場合は、保存するかどうか聞かれます。

コンパイル

データベースを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。このマニュアルの第 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。必要に応じて、このマニュアルの後ろの部分も参照してください。

データベースのコンパイル処理が行われている間、次のような「4D Compiler」ウインドウが表示されます。

このウインドウの上段には、処理が進むに従い、プロシージャとスクリプトの名前が次々と表示されます。

ウインドウの下段には、エラーも含めたメッセージのリストが表示されます。この上部には、エラーカウンタと警告カウンタが表示されます。カウンタは、コンパイル処理中自動的に更新されます。

「停止」ボタンでコンパイル処理を一時的に停止することができます。また「中止」ボタンで完全に中止できます。詳細に関しては、この章の「コンパイル処理への割り込み」を参照してください。

コンパイル処理の主なフェーズは、以下の 3 つです。

■ データベース取り込み処理

■ 変数のタイプチェック処理

■ コンパイル処理

上段のウインドウの左側を移動する小さなアイコンが表示され、現在処理中のフェーズを示します。この過程は、コンパイラウインドウで選択したオプションによって管理されます。

データベース取り込み処理

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）だけが複製され、コンパイルされます。このファイルにはデータベースのプロシージャとスクリプトが入って

います。データファイルは、コンパイルしたストラクチャファイルからも、オリジナルのストラクチャファイルからも開くことができます。

「実行形式アプリケーションの作成」オプションを選択すると、このフェーズで4D Engineも複製されます。

変数タイプチェック処理

このフェーズでは、コンパイル後のデータベース用にシンボルテーブルが作られます。このフェーズには、以下の 3 つの処理があります。

- **コンパイラ命令タイプチェック処理**。コンパイラ命令や配列定義ステートメントを見つけ、それらを使って変数や配列にデータタイプを割り当てます。
- **プロセスおよびインタープロセス変数タイプチェック処理**。コンパイラ命令タイプチェック処理でタイプが決まらなかったプロセスとインタープロセス変数にタイプを割り当てます。
- **ローカル変数タイプチェック処理**。ローカル変数にタイプを割り当てます。

各処理の経過はアイコンで識別できますが、プロシージャが小さかったり、コンピュータが速い場合、アイコンはほとんど見えません。

コンパイラ命令タイプチェック処理

この処理では、次のアイコンが表示されます：

コンパイラ命令によって定義されている変数を見つけ、そのデータタイプを納めます。同様にデータベース中の配列を見つけ、そのデータタイプを納めます。

プロセスおよびインタープロセス変数タイプチェック処理

この処理では、次のアイコンが表示されます：

コンパイラ命令で定義されていないプロセス変数とインタープロセス変数を見つけ、そのデータタイプを納めます。コンパイラは、変数の使われ方によってタイプを決定します。

ローカル変数タイプチェック処理

この処理では、次のアイコンが表示されます：

データベース内のローカル変数を見つけ、そのデータタイプを納めます。ローカル変数に対応するコンパイラ命令があれば、それを使ってタイプを決定します。コンパイラ命令がないローカル変数の場合は、変数の使われ方によってタイプを決定します。

全体的なチェック

第 5 章の「タイプ設定ガイド」で説明するタイプ定義の矛盾が見つかるフェーズです。

矛盾が起きている場合、各処理の中で、その変数が最初に認識された時のタイプが使われます。グローバルプロシージャは 4th Dimension の「プロシージャ」ダイアログボックスの一覧に表示される順に解析されます。

次に、ファイルプロシージャ、レイアウトプロシージャ、レイアウトに作られたスクリプトが、ファイルとレイアウトの順に解析されます。

ソートしたりコピーしたりして順序を変更すると、データベースの実行は同じでも、タイプの矛盾を示すメッセージは変わります。

タイプチェック処理のまとめ

タイプチェック処理が正常に終了するか、エラーが見つかったとしてもデータベースをコンパイルできないほどでなければ、次に述べる“ コンパイル処理 ”フェーズが自動的に始められます。

全体タイプ定義エラーが見つかると、“ コンパイル処理 ”フェーズには進まずに、さらにタイプチェックのフェーズが行われます。

この処理では、次のアイコンが表示されます：

このようなエラーには、以下の2つの原因があります。

■ 同じ名前のオブジェクトが 2 つある場合。つまり、プロシージャと変数、変数と外部プロシージャ、2 つのグローバルプロシージャなどです。同じ名前でタイプの異なる変数が 2 個見つかった場合、データタイプの矛盾は記録されますが、この種の矛盾でコンパイルが中止されることはありません。

■ タイプを判定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。

メッセージ内容をもとにエラーを修正するのは皆さんです。

コンパイル処理

プロシージャを機械語に翻訳して保存するフェーズです。ここでも、4D Compilerはエラーを探します。タイプ定義に関するエラーではなく、コンパイルエラー（シンタックスエラーやその他の矛盾、不一致など）を調べます。

この処理では、次のアイコンが表示されます：

コンパイル処理でエラーがなければ、コンパイルしたデータベースが作成されます。

エラーが見つかると、コンパイルしたデータベースは作成されません。まず、エラーを修正する必要があります。

コンパイル処理への割り込み

タイプチェックやコンパイル処理が行われている時に、「停止」または「中止」ボタンをクリックして割り込むことができます。

エラーや警告メッセージが出力されたときはいつでも、割り込みが役に立ちます。メッセージが表示されたプロシージャの間を自由に移動できるからです。移動には、以下の 2 種類の方法があります。

■ マウスでプロシージャのリストをスクロールし、プロシージャをクリックする。

■ “ Tab ” キーを使い、エラーや警告を含むプロシージャから次のプロシージャへ移動する。反対方向に移動する場合は、“ Shift-Tab ” を使います。

エラーや警告は 2 通りの方法で示され、これらの方法はメッセージの性質に応じて使い分けられます。

■ プロシージャやスクリプトのエラーが見つかると、該当するプロシージャまたはスクリプトの名前はボールドで表示されます。

■ プロシージャやスクリプトの警告を示す場合は、該当するプロシージャまたはスクリプトの名前がイタリック体で表示されます。

■ エラーと警告の両方があるプロシージャは、名前がボールドイタリック体で表示されます。

プロシージャはクリックして選択します。

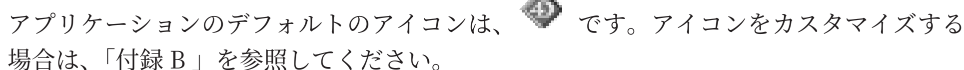
エラーや警告は下のパネルに記述されます。以下の内容を 3 行で示します。

- プロシージャ中の行番号
- エラーや警告が検知された行
- エラーや警告の説明

コンパイル後のデータベースを使用する

コンパイル後のデータベースは、4th Dimension のデータベースと同じよう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。

実行形式のアプリケーションを作成した場合は、アプリケーションアイコンをダブルクリックしてください。

アプリケーションのデフォルトのアイコンは、 です。アイコンをカスタマイズする場合は、「付録 B 」を参照してください。

コンパイルしたデータベースは、オリジナル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またはプロジェクトファイルを 4D Compiler アイコンの上にドラッグ＆ドロップしてコンパイラを起動することができます（Macintosh版では System 7 以降のバージョン、Windows版では、Windows 95が必要です）。これは、ファイル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 アイコンの上に移動させてマウスボタンを離すという操作です。

4th Dimension とともに 4D Compiler を使用

4D Compiler 2.5バージョンは、4th Dimension バージョン 3.5、4D Server™ バージョン 1.5 に対応しており、「Mac4DX」フォルダまたは「WIN4DX」ディレクトリ内に配置された 4D エクステンションを検知します。

Power Macintosh 用に最適化された 4D Compiler

バージョン 2.5 の 4D Compiler は、Power Macintosh 用に完全に最適化されています。Power Macintosh 上のコンパイル速度がかなり向上しました。

680xx と PowerPC 用のコンパイル

データベースのコンパイル時には、そのデータベースが 680xx あるいは PowerPC 搭載の Macintosh 上で、シングルユーザ向けの 4th Dimension や 4D Runtime とともに使われることを心に留めておいてください。さらに 4D Server の場合、680xx または PowerPC 搭載の Macintosh 上で 動作する 4D Client がデータベースに接続する可能性があります。つまり、データベースのコンパイル時に 680xx と PowerPC 両方に対応するようにコンパイルしておけば、アプリケーションはすべての構成に関して最適に動作し、使用プラットフォームのハードウエアとソフトウエアを活用できます。これは特に 4D Server に関して言えることで、使用するプラットフォームに対応するコードインスタンス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。

680xx または PowerPC だけに限定してコンパイルすると、そのデータベースは同種のプロセッサ用バージョンの 4th Dimension でしか使用できなくなります。680xx のみに対応してコンパイルしたデータベースは、Power Macintosh バージョンの 4D では動作しません。また逆に、PowerPC のみに対応してコンパイルしたデータベースは 680xx 用バージョンの 4D では動作しません。いずれの場合も、アラートが表示され、データベースを開くことができません。

プロセッサを限定してコンパイルするのは、データベースを動作させる Macintosh が一種類のプロセッサタイプに限られている場合だけにしてください。限定できない場合は、両方のマイクロプロセッサに対応できるようにコンパイルします。

コンパイラから、デバッグの手助けとなる 4 種類の診断結果が出力されます。

- **シンボルテーブル：**データベースの解析に役立ちます。作成したデータベースの変数をシンボルテーブルを使ってチェックしてください。シンボルテーブルは、4D Compiler から出力されたエラーメッセージの理解に役立ちます。
- **エラーファイル：**エラーファイルはテキストファイルとして見たり、4th Dimension から使用して、データベースのデバッグをより迅速に行うことができます。
- **範囲チェック：**コンパイル後のアプリケーションで、プロシージャの実行を監視し、制御するための高度なツールです。
- **タイプファイル：**データベースで使用している変数すべてをコンパイラ命令でタイプ付けしたファイルです。

テキストドキュメントを開いて表示するためのアプリケーションを使い、シンボルテーブルファイル、エラーファイル、タイプファイルを開くことができます。4D Compiler から出力される 3 種類のファイルをダブルクリックした時に立ち上がるアプリケーションの種類を “ Customizer Plus ” で設定できます。このユーティリティは、4th Dimension に含まれています。詳細については、「付録 C」を参照してください。

この章では、上記 3 種類のファイルと、その使い方について詳しく説明します。

シンボルテーブル

シンボルテーブルは、テキストファイルです。テキストエディタやワードプロセッサ等で開くことができます。情報はタブで区切られたカラムになっています。シンボルテーブルには、データベース内のオブジェクトに関する情報がすべて含まれています。オブジェクト情報は、以下の 4 つ部分に分かれています。

- インタープロセス変数一覧
- プロセス変数一覧
- プロシージャ、スクリプト中のローカル変数一覧
- グローバルプロシージャと関数の一覧、引数がある場合は引数も含みます。

インタープロセス変数一覧とプロセス変数一覧

これらの一覧は、4 個のカラムに分かれています。

- 1 番目のカラムは、データベースで使われているインタープロセス変数、プロセス変数および配列の名前です。これらの変数は、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って決められるか、あるいはオブジェクトの使用法から推測されます。データタイプを決定できない場合は、空欄になります。

- 3 番目のカラムには、オブジェクトが配列の場合、その次元数が出力されます。

- 4 番目のカラムには、コンパイラがオブジェクトのデータタイプを確定した文脈が表示されます。変数が複数の場所で使用されている場合は、コンパイラがデータタイプを決定するのに使用した場所を示します。

シンボルテーブルで使われている記号を以下に示します。変数の名前が見つかった場所を示します。

- 変数がグローバルプロシージャで使われていた場合、プロシージャに付けられている名前が示されます。名前の前に（P）が付いています。

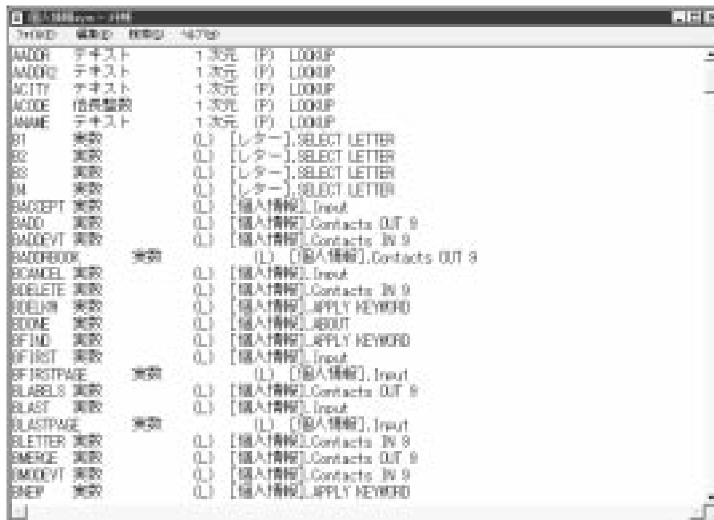
- 変数がファイルプロシージャで使われていた場合、ファイル名が示されます。名前の前に（FP）が付いています。

- 変数がレイアウトプロシージャで使われていた場合、ファイル名、レイアウト名が示されます。名前の前に（LP）が付いています。

- 変数がスクリプトで使用されていた場合、ファイル名、レイアウト名、オブジェクト名が出力されます。その前に（S）が付いています。

■　変数がレイアウト上のオブジェクトで、プロシージャ、フォーミュラ、スクリプトのいずれにも使われていない場合、その所属しているレイアウトの名前が示されます。レイアウト名に（L）が付きます。

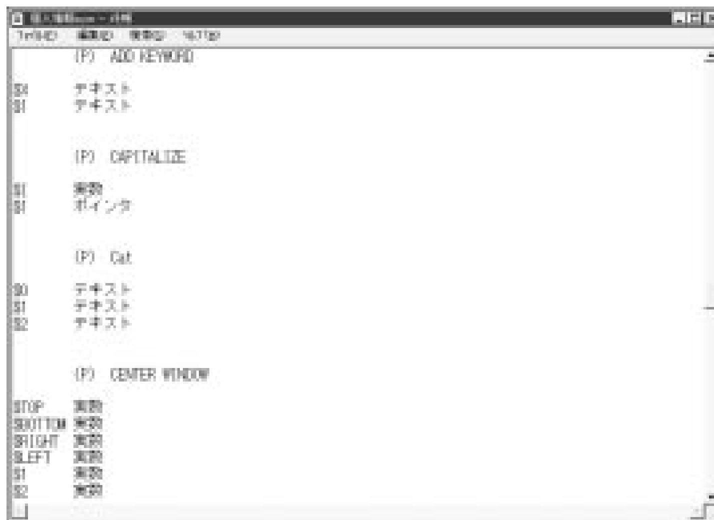
プロセス変数一覧の例を以下に示します。



ローカル変数一覧

ローカル変数一覧は、グローバルプロシージャ、ファイルプロシージャ、レイアウトプロシージャ、スクリプトの順に、4th Dimension 内の順番でソートされています。ローカル変数を使用しているプロシージャだけが表示されます。

次の図は、ローカル変数一覧の例です。

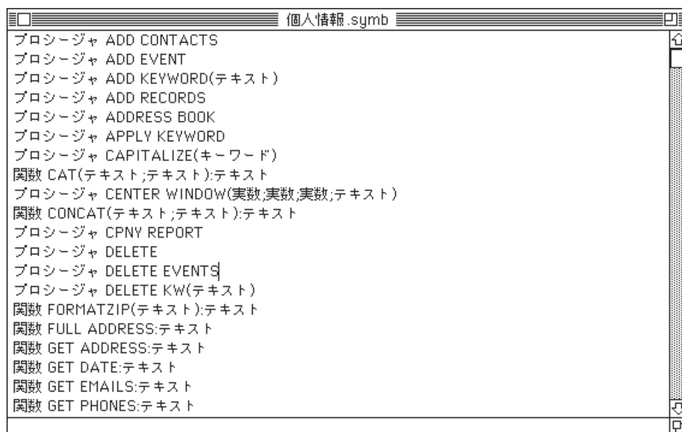


プロシージャおよび関数一覧

ファイルの最後は、グローバルプロシージャおよび関数の一覧です。各プロシージャの引数タイプ、関数の場合は返す値のタイプも以下のようなフォーマットで表示されます。

プロシージャ名 (引数のデータタイプ) : 返す値のデータタイプ

プロシージャおよび関数一覧の例を以下に示します。



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入っています。このファイルには、以下の 2 通りの使い方があります。

■ テキストファイルとして、テキストエディタやワードプロセッサで開いて使う。

■ 4th Dimension で対話型デバッグに使う。

メッセージのタイプ

4D Compiler から出力されるメッセージには、次の 3 種類のタイプがあります。

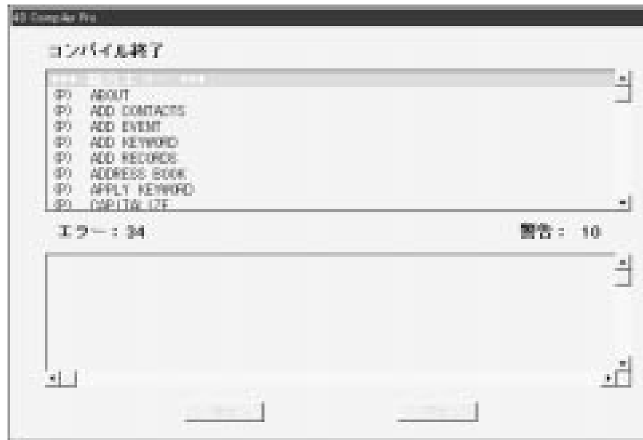
■ 総合エラーメッセージ

■ 特定行のエラーメッセージ

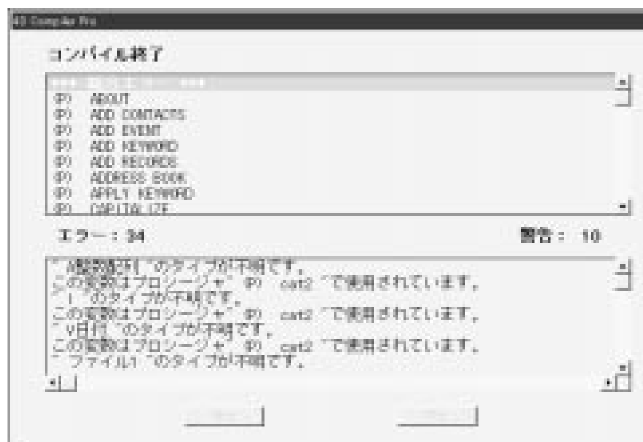
■ 警告メッセージ

総合エラー

データベースをコンパイルできなくなるエラーです。エラーファイルの先頭に出力されます。「4D Compiler」ウインドウでは、このメッセージの次にプロシージャのリストが表示されます。



総合エラーをクリックすると、エラーを説明するメッセージが次のように下側のパネルに表示されます。



コンパイラから総合エラーメッセージが出力されるのは、以下の 2 つの場合です。

- プロセス変数のデータタイプが決定できなかった場合
- 複数タイプのオブジェクトが同じ名前を持つ場合

1 番目は 4D Compiler で変数のタイプを特定できない場合、2 番目は指定された名前をどのオブジェクトに割り当てるか決定できない場合です。

これらのエラーは、特定のプロシージャやスクリプトに結び付いていないので、「総合エラー」と呼びます。「付録 A」に総合エラーのリストがあります。

特定行に結びついたエラー

このエラーは、エラーが見つかった行とエラーメッセージとともに表示されます。

データタイプや記述に矛盾のある式が見つかると、このエラーが出力されます。

この種のエラーが存在するプロシージャやスクリプトは、上側のウインドウにボールド体で表示されます。こうしたプロシージャを選択すると、そのプロシージャのエラー情報すべてが下側のパネルに、次のような形で表示されます。

- 行番号
- エラーのあった文の、プロシージャ内の表記
- エラーの説明

以下にコーディング上のエラー例を示します。

「付録 A」にエラーのリストがあります。

警告

警告は、エラーではありません。警告があっても、データベースはコンパイルできます。警告は、エラーになる可能性がある箇所を示します。

警告が検出されたプロシージャやスクリプトは、上側のウインドウにイタリック体で表示されます。プロシージャを1つ選択すると、警告情報が下側のパネルに表示されます。

警告情報は、以下の順に表示されます。

■ 行番号

■ 警告があるステートメント（プロシージャ中のステートメントをそのまま表示してある）

■ 警告の説明

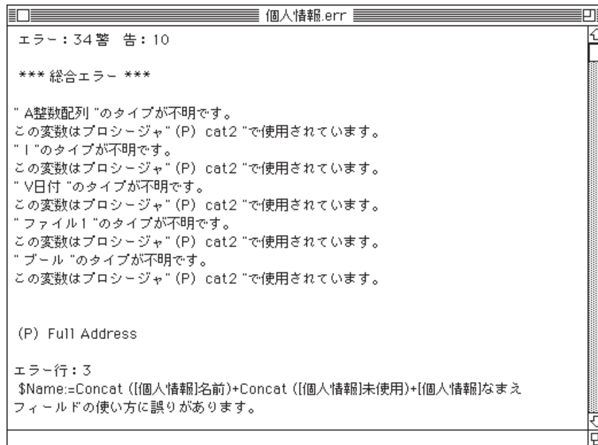
警告の「詳細」オプションを選択している場合、他の警告も表示されます。警告のリストは「付録 A」にあります。

エラーファイルの使い方

エラーファイルには2通りの使い方があります。1つはテキストファイルとして使う方法、もう1つは、4th Dimension で使う方法です。

エラーファイルのテキストとしての使い方

テキストエディタを使ってエラーファイルを開くと、以下のようになります。



```
個人情報.err

エラー：34警　告：10

*** 総合エラー ***

" A整数配列 "のタイプが不明です。
この変数はプロシージャ" (P) cat2 "で使用されています。
" I "のタイプが不明です。
この変数はプロシージャ" (P) cat2 "で使用されています。
" V日付 "のタイプが不明です。
この変数はプロシージャ" (P) cat2 "で使用されています。
" ファイル1 "のタイプが不明です。
この変数はプロシージャ" (P) cat2 "で使用されています。
" ブール "のタイプが不明です。
この変数はプロシージャ" (P) cat2 "で使用されています。


(P) Full Address

エラー行：3
$Name:=Concat ([個人情報]名前)+Concat ([個人情報]未使用)+[個人情報]なまえ
フィールドの使い方に誤りがあります。
```

エラーファイルの構成は、次のようになっています。

- ファイルの先頭にエラーと警告の個数が出力されます。
- 次に総合エラーのリストが出力されます。
- 最後に 4th Dimension の中で表示される順序で、プロシージャやスクリプトごとにソートされたその他のエラーと警告が出力されます。

エラーと警告は、以下の形式でリストされます。

- プロシージャ内の行番号
- エラーや警告のある行
- エラーの説明

4th Dimension でエラーファイルを対話型デバッグに使う方法

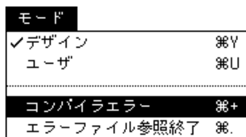
エラーファイルを使って、対話形式でエラーを修正できます。ここでは、その方法を説明します。

1.　コンパイラを実行させ、エラーファイルを作成する。

4th Dimension からエラーファイルを開くためには、データベースのあるフォルダに
4D Compiler のデフォルト名でエラーファイルを作成してください。

2.　4th Dimensionを起動し、オリジナルのデータベースを開き「デザイン」モードにします。

「モード」メニューに対話型デバッグ用の「コンパイラエラー」と「エラーファイル参照終了」の2つのメニューコマンドが自動的に追加されています。



3.「モード」メニューから「コンパイラエラー」コマンドを選択します。

4D Compiler がエラーまたは警告を検出した最初のプロシージャまたはスクリプトが自動的に開けられます。

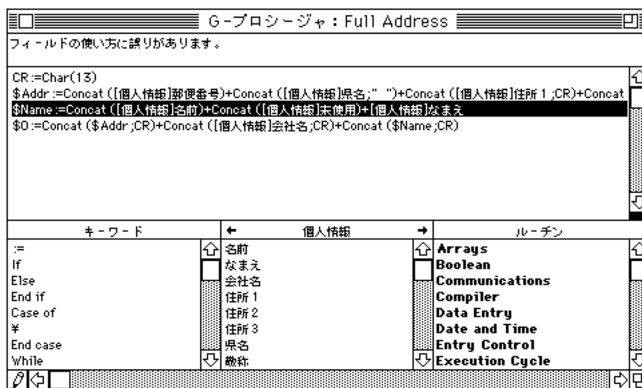
ウインドウの上端には、エラーまたは警告の種類が表示されます。

4. コンパイラから指摘されエラーを修正します。

メッセージが警告の場合、必ずしもコードを変更する必要はありません。

5.「モード」メニューから「コンパイラエラー」コマンドを選択して、次のエラーまたは警告に移ります。

次のエラーまたは警告が表示されます。



このようにすれば、エラー箇所を捜す手間がはぶけるので短時間でエラーを修正できます。特定のプロシージャ行に対応しない総合エラーはこの方法では表示されません。「エラーファイル参照終了」メニューコマンドを使って、対話型デバッグを終了します。「デザイン」モードから抜けると、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、次の 4 つの部分に分かれています。

■ インタープロセス変数のコンパイラ命令

■ プロセス変数のコンパイラ命令

■ ローカル変数のコンパイラ命令

■ 引数のコンパイラ命令

このファイルは、コンパイル時間を節約する場合に非常に有効です。このファイルの内容は、以下のタイプ付け用のプロシージャにペーストするだけで利用できます。

■ プロセス変数やインタープロセス変数、引数については、名前が「COMPILER」で始まるタイプ定義プロシージャに入れます。

■ ローカル変数については、それを使用しているプロシージャ内に入れます。詳細は第 5 章を参照してください。

範囲チェック

他のオプションはすべてコンパイル時の操作に影響を与えますが、「範囲チェック」指定だけはコンパイル後のデータベースを実行した時に効果が現れます。つまり、「範囲チェック」のメッセージはコンパイル後のデータベースの実行中にだけ出力されます。

「範囲チェック」は " 状況 " コントローラです。つまり、その時々のデータベースのオブジェクトの状態を評価します。「範囲チェック」の動作例を示します。

テキストタイプの配列 " a配列 " を定義し、" a配列 " の配列要素数は、プロシージャの実行に応じて変化するとします。

" a配列 " の 5 番目の要素に " こんにちは " という文字列を代入する場合、次のように記述します。

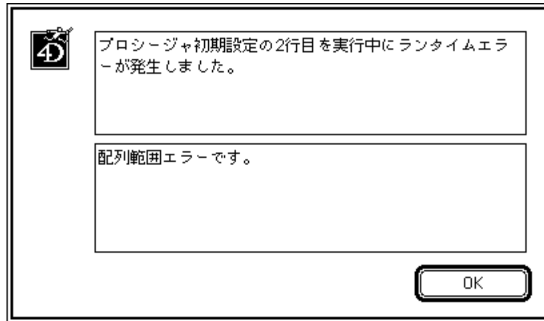
a配列{5}:="こんにちは"

その時に " a配列 " の要素数が 5 個以上なら、問題はありません。代入は正常に行われます。しかし、" a配列 " の要素数が 5 個以下なら、代入は無効です。

このような場合は、プロシージャの実行が必要ですから、コンパイル時には見つけられません。プロシージャがどのように実行されるかは、コンパイラにはわかりません。

データベースの使用中に起きていることを監視できるのは、「範囲チェック」だけです。

前の例では、コンパイラによって 4th Dimension から次のようなメッセージが表示されます。



配列、ポインタ、文字列を使って処理を行う場合に、「範囲チェック」がどれほど便利かおわかりでしょう。

「範囲チェック」を指定した場合に、コンパイラから表示されるメッセージについては「付録 A」を参照してください。

「範囲チェック」の使用法

「範囲チェック」を指定すると、指定しなかった場合に比べてコンパイル後のデータベースの実質的な処理速度が遅くなります。従って、完成したデータベースを最後にコンパイルする時には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、デバッグの過程で付加的に使用するべきです。最終的なコンパイルは、「範囲チェック」なしで行ってください。

コンパイラの最大の目的は、データベースの処理速度を向上させることであり、同時にアプリケーションの信頼性を高めることです。そこで、データベースの開発時には、「範囲チェック」を使うことをおすすめします。

異常診断

ここでは「範囲チェック」の重要性について、その背景とともに説明します。

データベースの実行時に異常が見つかったとします。原因になりそうな場所を推測する前に、コンパイラに用意されている手掛かりを思い出してください。

異常の可能性を以下に示します。

■ 4th Dimension からエラーメッセージが表示される。可能な場合は、4th Dimensionの指示に従ってデータベースのエラーを修正します。指示が一般的で修正の手掛かりにならない場合は、「範囲チェック」を指定してデータベースを再コンパイルし、データベースを再度テストしてください。4th Dimensionのメッセージが表示された箇所について、コンパイラからさらに詳しい情報が得られるでしょう。

■ 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きが違う。「警告」メッセージを詳しく検討してください。「範囲チェック」オプションも指定した方がよいでしょう。

■ データベースがインタプリタでは正常に機能するが、コンパイル後はエラーが発生して、システムの異常停止を引き起こしたり、「プログラムマネージャ」あるいは「ファインダ」に戻ったりする。これは、アセンブラや Pascal でプログラムを作成しているときによく起こる現象です。デバッガの使い方を知っている場合は、デバッガを使えば異常停止したプロシージャの名前がわかります。デバッガに詳しくない場合、最も簡単な解決方法は「範囲チェック」オプションを付けて再コンパイルすることです。ほとんどの場合、これで問題点が見つかります。

■ データベースがインタプリタでは動作するが、コンパイル後はシステムの異常停止が発生する。コンパイル後のデータベースで使用している外部ルーチンが、コンパイルした時点のものと同じかどうか調べてください。

■ 文字列変数や数値変数の値が予定している値と違う。コンパイルプロジェクトのデフォルトタイプオプションを確認してください。

4th Dimension インタプリタでは、変数やプロシージャ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書く際には、次の 2 つの原則を守ってください。

■ 変数やグローバルプロシージャ、外部プロシージャにはユニークな（重複しない）名前を付ける。

■ 変数には、明示的にせよ暗黙にせよ、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。

曖昧さは、主に次の 2 つの原因によって生じます。

■ 異なる2 つの変数に同じ名前を使う。

■ 1 つの変数に（明示的にせよ暗黙にせよ）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書かれたデータベースをデバッグする際に、このメッセージは役に立ちます。

変数と配列のデータタイプ

バージョン 3.5 の 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。「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

ここで、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、ブール、文字、日付、整数、倍長整数、グラフ、時間、ピクチャ、実数、ポインタ、テキストの 11 種類のタイプがあります。

配列には、ブール配列、文字配列、日付配列、整数配列、倍長整数配列、ピクチャ配列、実数配列、ポインタ配列、テキスト配列の 9 種類のタイプがあります。

シンボルテーブル

インタプリタの下では、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なりません。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、グローバルプロシージャ、レイアウトプロシージャ、ファイルプロシージャ、スクリプトです。

■ 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ると、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、プロシージャ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。

たとえば、

```
vワーク1:=12.5
```

と記述した場合、変数“vワーク1”のデータタイプは実数になります。

同様に、

```
vワーク2:="これは、例題"
```

と記述すると、“vワーク2”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。

```
vワーク1:=12.5
vワーク2:="これは、例題"
vワーク3:=vワーク１
```

この場合、“vワーク3”は“vワーク1”と同じタイプになります。

似たような例で、

```
vワーク4:=2 * vワーク2
```

この場合も、“vワーク4”は“vワーク2”と同じタイプになります。

同様に、4th Dimension コマンドや、プロシージャ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るのは、一番広い範囲をカバーできるタイプです。たとえば、以下のような場合、

```
v数字:= 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“ v数字 ”に“ 実数 ”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』マニュアルにあります。

C_STRING (**サイズ** ; **変数**1 {;...;**変数**N})
C_BOOLEAN (**変数**1 {;...;**変数**N})
C_DATE (**変数**1 {;...;**変数**N})
C_GRAPH (**変数**1 {;...;**変数**N})
C_INTEGER (**変数**1 {;...;**変数**N})
C_TIME (**変数**1 {;...;**変数**N})
C_PICTURE (**変数**1 {;...;**変数**N})
C_LONGINT (**変数**1 {;...;**変数**N})
C_REAL (**変数**1 {;...;**変数**N})
C_POINTER (**変数**1 {;...;**変数**N})
C_TEXT (**変数** 1{;...;**変数**N})

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。使い方は次の通りです。

C_BOOLEAN (v**変数**)

こうして、“ v変数 ”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、代入や用途から導か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。

コンパイラ命令が必要な場合

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 2 つの原因があります。

- 複数データタイプ。
- 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、変数が異なる文で再度タイプ定義された場合、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。

以下に例を示します。

```
v変数:=True           `プロシージャ１
v変数:="月は緑色"     `プロシージャ２
```

“ プロシージャ１ ” の後に “ プロシージャ２ ” がコンパイルされると、ステートメント “ v変数:="月は緑色" ” は、“ v変数 ” 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。

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すが、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、

```
v変数１:=5.2                              `(1)
vポインタ:=>>v変数１                       `(2)
v変数２:=vポインタ>>                       `(3)
```

ポインタ“ vポインタ ”が指す変数のタイプは (2) で定義されていますが、“ v変数２ ”のタイプは決定されません。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、“ v変数２ ”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

C_REAL (v**変数２**)

が必要です。

■ **複数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。たとえば、

FIELD ATTRIBUTES コマンドには、次の2つのシンタックスがあります。

FIELD ATTRIBUTES (ファイル番号 ; **フィールド番号** ; **タイプ** ; **長さ** ; **インデックス)**
FIELD ATTRIBUTES (フィールドポインタ ; **タイプ** ; **長さ** ; **インデックス)**

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。

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、プロシージャの処理速度が速くなります。以下に一般的な例を示します。

カウンタにローカル変数を使うとします。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

行うように指定できます。こうすることよって、コンパイル時間を半分以下に短縮できます。詳細に関しては、第 2 章の 4〜10 ページ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。

たとえば、次のように書いた場合、

```
C_INTEGER (v整数)
v整数:=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。以下のように書くと、

```
C_STRING (10 ; v文字)
v文字:="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、変数がどういう使われ方をしているかを見て、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。たとえば、次のように記述して、

```
C_LONGINT (v整数)
```

データベースの他の場所で、次のように記述したとします。

```
v整数:=3.1416
```

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、上の例では、インタプリタでもコンパイル後でも “ v変数 ” 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など）、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、インタプリタは系統だてて処理するわけではありません。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、定義を必要とする変数を使用する文に先立って、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重要です。

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。

コンパイラによって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プロシージャやスクリプトでコンパイラ命令を使う。コンパイラ命令は、必ず変数が初めて使われる場所、つまり一番最初に実行されるはずのプロシージャで使うようにしてください。

注：コンパイル時、4D Compiler は 4th Dimension が与える順序、つまり「プロシージャ」ダイアログボックスに現れる順序でプロシージャを処理します。

■ さまざまな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、“ Startup ” プロシージャまたは “ Startup ” プロシージャから呼び出されるプロシージャにまとめてください。

■ ローカル変数はそれが使用されるプロシージャで定義します。

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プロシージャは、必ずしも実行される必要はないと記述されています。そのプロシージャ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プロシージャに入れる、というものです。たとえば、“ COMPILER ”、“ COMPILER1 ”、“ COMPILERtype ” というプロシージャ名にします。

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。

以下の 4 種類の区分です。

- インタープロセス変数
- プロセス変数
- ローカル変数
- プロシージャに渡す引数

コンパイラでは、プロシージャへの呼びだしをコンパイルするために、プロシージャ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。

命令 (プロシージャ名 ; 引数)

たとえば、次の行はプロシージャ " MyProcedure " 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 (MyProcedure ; $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 でこれらのプロシージャを実行することも可能です。

C_STRING コンパイラ命令

C_STRING コマンドのシンタックスは、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 (サイズ ; 変数1{;...; v**変数**N})

C_STRING コマンド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。

インタプリタでは、

v**長さ**:=15
C_STRING (v**長さ** ; The**文字**)

のような設定も、問題なく使用できます。

4th Dimensionは“v長さ”を解析し、**C_STRING** コンパイラ命令で、“v長さ”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ら、“v長さ”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。たとえば、

C_STRING (15 ; The**文字**)

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 (長さ ; **配列名** ; **サイズ)**

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。

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、実地に即して理解していただくために、次の 2 つの章を読むことをおすすめします。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

タイプ設定ガイド　5

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。

- インタープロセス変数とプロセス変数
- ローカル変数
- 配列
- レイアウト変数
- ポインタ
- 外部プロシージャ
- 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す。

- 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、

```
v変数:=5
```

と書き、同じデータベース内で次のように書いたとします。

```
v変数:=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え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、

```
v変数:=5
```

と書き、同じアプリケーション内で次のように書いたとします。

```
C_BOOLEAN (v変数)
```

コンパイラ命令が最初に処理されるので、v変数はブールタイプに設定されますが、“ v変数:=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。

簡単な例を示します。

```
vブール:=True                     `“ vブール ” のデータタイプはブールである
C_INTEGER (<>v整数)
<>v整数:=3                        `代入値はコンパイラの指定と同じタイプ
v結果:=<>v整数 + vブール          `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それとは異なる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。

インタプリタでは、次のように書いてもエラーになりません。

```
v番号:=Request ("識別 コード")          `“ v番号 ” のデータタイプはテキスト
If (OK=1)
  v番号:=Num (v番号)                    `“ v番号 ” のデータタイプはここでは実数
  SEARCH ([得意先]コード=v番号)
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、この例のようにプロシージャの構造の変更で修正できるきる場合もあります。

```
v番号:=Num (Request ("識別 コード"))    `“ v番号 ” のデータタイプは実数
If (OK=1)
  SEARCH ([得意先]コード=v番号)
End if
```

2 つのコンパイラ命令の矛盾

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 (v変数)
```

そして、

```
C_TEXT (v変数)
```

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 (4 ; v**文字列**)
v**文字列**:="**かめ**"
C_STRING (6 ; v**文字列**)
v**文字列**:="**うさぎ**"

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 (6 ; v**文字列**)
v**文字列**:="**かめ**"
v**文字列**:="**うさぎ**"

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、1 つのプロシージャやスクリプト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、プロシージャやスクリプトのレベルです。たとえば、1 つのプロシージャの中で、

$**ワーク**:="**うさぎ**"

と記述し、さらに、

$**ワーク**:=5

と記述するとエラーになりますが、プロシージャ「P1」で

$**ワーク**:="**うさぎ**"

と記述し、

$**ワーク**:=5

とプロシージャ「P2」で記述することができます。ローカル変数の範囲はデータベース全体ではなくプロシージャ内だけだからです。

配列のタイプ矛盾

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プロシージャでも変更できるし、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、文字長を変えない

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なりません。

次のように書くと、

ARRAY INTEGER (a**配列** ; 5)
ARRAY BOOLEAN (a**配列** ; 5)

コンパイラでは " a配列 " 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す。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。

そのため、同じデータベースで、

ARRAY INTEGER (**配列**1 ; 10)
ARRAY INTEGER (**配列**1 ; 10 ; 10)

上のように記述できませんが、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 (**配列**1 ; 10)
ARRAY INTEGER (**配列**2 ; 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えることはできませんが、配列の要素数は変えられます。2 次元配列の 1 番目の配列の要素数は変えられます。次のように記述することができます。

```
ARRAY INTEGER (配列3 ; 5)
ARRAY INTEGER (配列3 ; 10)                    `配列の要素数を変更する
```

参考：2 次元配列は、実際には複数の 1 次元配列から成っています。詳細は、『4th Dimensionランゲージリファレンス』の「配列とポインタ」の章を参照してください。

文字配列

文字配列には、固定長文字列と同様の理由から、同じ規則が適用されます。

次のように記述すると、

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

コンパイラはサイズの矛盾とみなします。解決方法は単純です。一番大きいサイズを使ってください。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**、**DISTINCT VALUES** などの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。必ず、前で説明した 3 つの状況のうちの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。通常、修正すべき点はかなり明らかになるはずです。暗黙に配列のタイプが変更される例は、次章の配列コマンドの説明にあります。

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なりません。たとえば、プロシージャで 10 要素のローカル整数配列を作る場合、そのプロシージャに次の行を追加します。

```
ARRAY INTEGER ($配列 ; 10)
```

レイアウト変数

レイアウト内で作られる変数（ボタン、ポップアップメニューなど）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が、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、レイアウト変数のタイプを定義する
- コンパイラがデフォルトで適当なデータタイプを割り当てる

というものです。コンパイラがレイアウト変数のタイプを設定する法則については、以下で説明します。

数値タイプに設定されるレイアウト変数

レイアウト変数の中で、チェックボックス、ボタン、ハイライトボタン、透明ボタン、ラジオボタン、ラジオピクチャ、ルーラ、ダイアル、サーモメータは実数タイプに設定されます。ただし、プロジェクト内でその他の指定がある場合を除きます。

レイアウト変数について、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。

外部オブジェクト変数

外部ルーチンエリアは常に倍長整数です。データタイプの矛盾はまずありえません。外部ルーチ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レイアウト変数

テキストタイプに設定される変数は、入力不可の変数、入力可の変数、ドロップダウンリスト ボックス（ポップアップメニュー）、スクロールエリアの 4 種類です。

これらの変数は、以下の 2 つに分けられます。

- **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ボックス（ポップアップメニュー）、スクロールエリア

単純変数

デフォルトのデータタイプはテキストです。スクリプトやプロシージャで使われるときは、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

ドロップダウンリストボックス（ポップアップメニュー）とスクロールエリアは配列の内容を表示するだけです。これらを配列として定義しないとテキスト変数にタイプ設定されます。前述の配列に関する原則を守っていれば問題はありません。

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すことができます。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。たとえば、以下のような場合です。

```
v変数:=5.3
pポインタ:=>>v変数
pポインタ>>:=6.4
pポインタ>>:=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ので、データタイプの矛盾が起こります。1つのプロシージャ内で、異なった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。以下にポインタの正しい使用例を示します。

```
v変数:=5.3
pポインタ:=>>v変数
```

```
pポインタ>>:=6.4
vブール:=True
pポインタ:=>>vブール
pポインタ>>:=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”フェーズではエラーメッセージは出力されません。

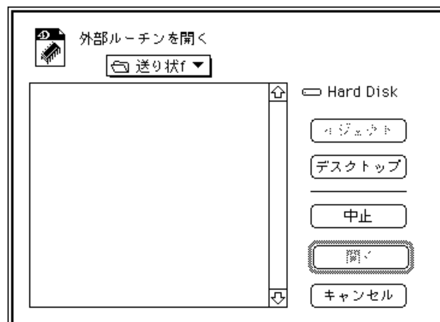
ただし、ポインタに関係する矛盾を見つける方法がまったくないわけではありません。“範囲チェック”オプションによってポインタの使われ方をいくらかは解析できます。“範囲チェック”オプションについては、第3章で詳しく説明しています。

外部プロシージャ（エクステンション）

4th Dimension の外部ルーチンやエクステンションを使用する場合、4D Compiler は、こうしたルーチンへの呼び出しを含んでいるプロシージャをコンパイルすることになります。

エクステンションを持つアプリケーションをコンパイルする際には、Windows のエクステンションは「WIN4DX」フォルダ／ディレクトリに入れ、Macintosh のエクステンションは「Mac4DX」フォルダ／ディレクトリに入れておきます。いずれの場合も、こうしたディレクトリやフォルダは、ストラクチャファイルと同じ階層に配置しておく必要があります。コンパイラはこれらのファイルの複製は作りませんが、さまざまなルーチンの宣言と矛盾しないかどうか解析します。

エクステンションがどこか他の場所にある場合は、必要なエクステンションが入ったファイルの置き場所を聞かれます。対象ドキュメントをクリックして選択し、「開く」ボタンをクリックします。



この時、プロシージャの定義が解析されます。また、エクステンションに対する呼び出しに誤りがなければ、タイプ定義矛盾によるエラーの危険はありません。

外部プロシージャ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。

外部ルーチンによって作られるプロセス変数とインタープロセス変数（VAR# と VAR<>）は、それが外部プロシージャに含まれている場合は、コンパイル時に 4D Compiler で認識されます。プラットフォーム独立のエクステンションには、プロセス変数とインタープロセス変数のエントリ ポイントを持つためこれが可能になります。

マルチプラットフォームのコンパイル

コンパイルを行っているプラットフォーム以外のプラットフォームで動かすためのデータベースをコンパイルするためには、対応する 4D Compiler プラットフォームエクステンションを入手する必要があります。Windows 上でコンパイルを行い、コンパイル後のデータベースを Macintosh や Power Macintosh で動作させるためには、コンパイル前に、対応する Macintosh／Power Macintosh のエクステンションを「Mac4DX」ディレクトリまたは「Proc.ESR」ファイルに入れておく必要があります。同様に、Macintosh 上でコンパイルし、コンパイル後のデータベースを Windows 上で動作させる場合は、Windows 用のエクステンションを「WIN4DX」フォルダまたは「Proc.Ext」ファイルに入れておく必要があります。

2 種類の異なるプラットフォーム上で動作するように設計されたデータベースをコンパイルするためには、ストラクチャと同じ階層に「Mac4DX」と「WIN4DX」フォルダ／ディレクトリを置く必要があります。

注：4D Compiler バージョン 2.5 は従来のエクスターナル方式にも対応しています。データベースストラクチャに納められたり、「Proc.Ext」ファイルに入っている外部ルーチンも正しく認識します。

暗黙引数を受け取る外部ルーチン

ある種の外部パッケージ（4D Calc、4D Draw、4D SQL Serverなど）は、暗黙に
4th Dimension ルーチンを呼び出すルーチンを使います。

たとえば、4D Draw の場合、**DR ON EVENT** コマンドの書式は次のようになります。

DR ON EVENT (イベントプロシージャ)

このコマンドの引数は、4th Dimensionで作ったプロシージャの名前です。このプロシージャは、イベントを受け取るたびに 4D Draw から呼ばれ、自動的に次のような引数を受け取ります。

引数	タイプ	説明
$1	倍長整数	4D Draw エリア
$2	倍長整数	イベントコード
$3	倍長整数	エリアが属するレイアウトのファイル番号
$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、引数をコンパイラ命令、またはプロシージャ内での使用法によってタイプ設定しなければなりません。プロシージャ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。

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。これを説明する例を示します。

1 次元配列のメモリサイズを返す関数を作るとします。テキスト配列とピクチャ配列を除き、結果はすべて実数になります。テキスト配列とピクチャ配列のデータサイズは計算式では求められないからです。

テキスト配列とピクチャ配列に関しては、文字列を返します。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、以下の 2 通りの方法があります。

- ローカル変数のデータタイプを気にせずに使用する。このプロシージャ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。

```
`関数“メモリサイズ”インタプリタ用
$サイズ:=Size of array ($1>>) + 1
$タイプ:=Type ($1>>)
Case of
  \ ($タイプ=14)      `実数配列
   $0:=8+($サイズ*10)                      `$0は実数(数値)
  \ ($タイプ=15)      `配列は整数
   $0:=8+($サイズ*2)                       `$0は実数(数値)
  \ ( $タイプ=16)     `倍長整数配列
   $0:=8+($サイズ*4)                       `$0は実数(数値)
   `以下の場合$0はテキスト
  \ ($タイプ=18)                           `テキスト配列
   $0:=String (8+($サイズ*4))+"テキスト長の合計"'
  \ ($タイプ=19)                           `ピクチャ配列
   $0:=String (8+($サイズ*4))+"ピクチャサイズの合計"
End case
```

上のプロシージャでは、$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。

```
`関数“メモリサイズ”インタプリタ用、コンパイラ用
$サイズ:=Size of array ($1>>) + 1
$タイプ:=Type ($1>>)
v数値:=0
Case of
  \ ($タイプ=14)      `実数配列
   v数値:=8+($サイズ*10)                   `$0は実数(数値)
  \ ($タイプ=15)      `整数配列
   v数値:=8+($サイズ*2)                    `$0は実数(数値)
  \ ($タイプ=16)      `倍長整数配列
   v数値:=8+($サイズ*4)                    `$0は実数(数値)
  \ ($タイプ=18)      `テキスト配列
   vテキスト:=String (8+($サイズ*4))+"テキスト長の合計"
  \ ($タイプ=19)                           `ピクチャ配列
   vテキスト:=String (8+($サイズ*4))+"ピクチャサイズの合計"
End case
if (v数値 # 0)
  $0:=>>v数値
Else
  $0:=>>vテキスト
End if
```

2 つの関数は、以下の点が違っています。

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

結果はポインタで簡単に参照できます。

引数の間接参照

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てください。

プロシージャ内で、間接参照は " ${$i} " のように表します。" $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、プロシージャが呼ばれるたびに変わります。このプロシージャには数値と編集フォーマットを引数としてプロシージャ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。

```
結果:=合計編集 ("##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。

以下は、関数 " 合計編集 " です。

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

この関数は、次のように引数の個数を変えて使うことができます。

合計編集 ("##0.00" ; 125.2 ; 2 ; 33.5 ; 24)

または

合計編集 ("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、以下のように書きます。

C_INTEGER (${4})

これは、4 番目以降の引数のデータタイプが整数であるという意味です。データタイプは整数です。$1、$2、$3 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、プロシージャ、関数、外部プロシージャ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

以下に、4th Dimensionのシステム変数とそのデータタイプを示します。

システム変数	データタイプ
OK	倍長整数
Document	文字(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ときには、他の変数と同じように C1、C2、...Cn のタイプは変更できないことに注意してください。

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 4 種類です。

COPY ARRAY (**コピー元** ; **コピー先**)
SELECTION TO ARRAY (**フィールド** ; **配列**)
LIST TO ARRAY (**配列** ; **リスト** ; {**リンク配列**})
DISTINCT VALUES (**フィールド** ; **配列**)

COPY ARRAY コマンド

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。

次のいずれの場合もこのような推測が行われます。

■ 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、固定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。

```
$サイズ:=Size of array (整数配列)
ARRAY REAL (実数配列 ; $サイズ)          `実数配列と整数配列のサイズを同じにする。
For ($i ; 1 ; $サイズ)
  実数配列{$i}:=整数配列{$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 コマンド

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 ([私のファイル]整数フィールド ; 私の配列)

“ 私の配列 ” は整数配列になります（“ 整数フィールド ” 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、同じで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デフォルトではテキストになります。

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ます。テキストタイプのフィールドについても同様です。宣言が優先されます。

SELECTION TO ARRAYコマンドは、1 次元の配列でしか使用できないことを覚えておいてください。**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 (ファイル ; 配列)

この場合、変数 “ 配列 ” は倍長整数の配列です。

LIST TO ARRAY コマンド

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 種類の配列だけです。

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。

DISTINCT VALUESコマンド

DISTINCT VALUES コマンドの場合、作成される配列のタイプは、文字配列として定義されていなければテキストになります。このコマンドは、1 次元配列でのみ使用できます。「コンパイル」モードでは、**DISTINCT VALUES** コマンドはインデックス付き文字フィールドにのみ使用できます。

配列関連コマンドでのポインタの使用

ポインタの節で説明したように、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。

以下のように書いた場合、

SELECTION TO ARRAY ([**マスタ**]**コード** ; P**ポインタ**>>)

“ Pポインタ>> ” 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にするのはユーザです。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（定義されたプロシージャ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 STRING ($**私の配列** ; 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。

通信コマンド

SEND VARIABLE (**変数**)
RECEIVE VARIABLE (**変数**)

上の 2 つのコマンドは、変数をディスクに書き込んだり、受け取ったりする場合に使われます。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します。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になります。**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。

以下に例をあげます。

```
`変数送信の例
SET CHANNEL (12 ; "文書1")
$タイプ:=Type ([顧客]合計)
SEND VARIABLE ($タイプ)
For  ($i ; 1 ; Records in Selection)
  $合計:=[顧客]合計
  SEND VARIABLE ($合計)
End for
SET CHANNEL (11)
```

```
`変数受信の例
SET CHANNEL (12 ; "文書1")
RECEIVE VARIABLE ($タイプ)
case of
  \ ($タイプ=0)
    RECEIVE VARIABLE ($文字)          `受信した変数の処理
  \ ($タイプ=1)
    RECEIVE VARIABLE ($実数)          `受信した変数の処理
  \ ($タイプ=2)
    RECEIVE VARIABLE ($テキスト)      `受信した変数の処理
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。

データ入力

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えられるので、どのオブジェクトを指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。

例外処理コマンド

ON EVENT CALL (エラープロシージャ)
ON SERIAL PORT CALL (シリアルプロシージャ)
ABORT
IDLE

例外処理を扱うために **IDLE** というコマンドが、4th Dimension言語に加えられています。**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ということができます。

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。

一方、イベント待ちのループのように、4th Dimension が受身でイベントを待っているときは、当然カーネルコールはありません。

以下にその例を示します。

```
`MouseClickプロシージャ
<>vテスト:=True
MESSAGE ("マウスがクリックされました。")
```

```
`Waitプロシージャ
<>vテスト:=False
ON EVENT CALL ("マウスクリック")
While (Not (<>vテスト))                  `イベント待ちのループ
  ..         `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プロシージャ
<>vテスト:=False
ON EVENT CALL ("マウスクリック")
While (Not (<>vテスト))
  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 3.x 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 3.x からは、RECEIVE PACKET コマンドを利用した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コマンド

このコマンドは、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、時間です。

演算

Mod (数値1 ; **数値**2)

4th Dimension では、25 を 3 で割った余りを求めるる場合、次の 2 通りの書き方があります。

v**変数**:=**Mod** (25 ; 3)

または

v**変数**:=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えますが、% 演算子は整数と倍長整数にしか使えません（% 演算子を使った場合、被演算子が倍長整数の制限を越えてしまうと、演算結果は保証されません）。値が倍長整数の範囲内にあるとわかっている場合は、必ず % 演算子を使うようにしてください。

ブレーク処理

Subtotal (データ)

コンパイルしたデータベースでは、**Subtotal** 関数はブレーク処理を起こしません。ブレーク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** 関数の引数が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。

空の文字列がこの関数に渡されていると、“ 範囲チェック ” オプションによりメッセージが表示されます。

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 (ファイル番号;フィールド番号)
File (ファイルポインタ) または File (ファイル番号)

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違っています。

- ポインタを与えると、**File** 関数は数値を返します。
- 数値を与えると、**File** 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 (変数)
SAVE VARIABLE (文書 ; 変数1 ; {...;**変数**N})
LOAD VARIABLE (文書 ; 変数1 ; {...;**変数**N})
CLEAR VARIABLE (変数)
Get Pointer (名前)
EXECUTE (ステートメント)
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えません。コンパイルしたデータベースでは、*Startup* プロシージャの実行前に、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。

この法則は配列には、当てはまりません。コンパイル後のデータベースで配列に対して **Undefined** 関数を適用すると、True（真）が返ります。

注：Undefined 関数を使うと、実行中のデータベースがコンパイルされたものかどうか判断できます。データベースをオープンした直後に、任意の変数について、定義されているかどうかテストしてみてください。Undefined 関数の結果が True（真）**なら、そのデータベースは未コンパイルです。False（偽）であればコンパイルされています。**

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。

プロシージャは以下のようになります。

```
v変数1:="xxxxxx"'                        `"xxxxxx"はLOAD VARIABLEで
v変数2:="xxxxxx"                         `ロードされない値
v変数3:="xxxxxx"
v変数4:="xxxxxx"
LOAD VARIABLE ("文書ファイル" ; v変数1 ; v変数2 ; v変数3 ; v変数4)
If (v変数1="xxxxxx")                     `ドキュメントがみつからなかった
  `...
Else                                     `ドキュメントがみつかった。
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）。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、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 (配列 ; 0)          `1次元配列の場合
ARRAY INTEGER (配列 ; 0 ; 0)      `2次元配列の場合
```

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。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します。配列の各要素はそれぞれ与えられた変数を指します。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 (p配列 ; 12)
p配列{1}:=>>V1
p配列{2}:=>>V2
・
・
p配列{12}:=>>V12
```

また、次のように書くこともできます。

```
ARRAY POINTER (p配列 ; 12)
For (i ; 1 ; 12)
  p配列{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、または定義するかしてください。

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)

■ プロシージャで 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、インタプリタでは役に立ちますが、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、**EXECUTE** コマンドに引数として渡されるプロシージャ名はインタープリトされます。そのため、4D Compiler 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ができません。

EXECUTE コマンドは、複数のステートメントで置き換えることができます。例を紹介します。

```
$数値:=カウンタ
EXECUTE ("印刷"+String ($数値))
```

これは次のように書き換えることができます。

```
Case of
 \ ($数値=1)
  印刷1
 \ ($数値=2)
  印刷2
 \ ($数値=3)
  印刷3
 `
  ...
End case
```

EXECUTE コマンドは必ず書き換え可能です。実行するプロシージャはデータベースのグローバルプロシージャ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、プロシージャ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

次のプロシージャを考えてみましょう。プロシージャ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、このプロシージャ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 はこのプロシージャをコンパイルできません。

```
DEFAULT FILE ([aFile])
 `セットに関するコマンド
V:="セット名"
P:=>>V
ADD TO SET (P>>)
CREATE EMPTY SET (P>>)
CREATE SET (P>>)
LOAD SET (P>> ; "私の文書")
 `レイアウトに対するコマンド
V:="レイアウト名"
P:=>>V
INPUT LAYOUT (P>>)
OUTPUT LAYOUT (P>>)
DIALOG (P>>)
 `命名セレクションに関するコマンド
V:="セレクション名"
P:=>>V
COPY NAMED SELECTION (P>>)
CUT NAMED SELECTION (P>>)
 `書き出しと読み込みに対するコマンド
V:="文書名"
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:=>>[aFile]aFiled
SEARCH (P>>#"")
SEARCH BY FORMULA (P>>#"")
SEARCH SELECTION (P>>#"")
SEARCH SELECTION BY FORMULA (P>>#"")
SORT SELECTION (P>> ; >)
 `印刷に関するコマンド
V:="文書名"
P:=>>V
PRINT LABEL (P>>)
REPORT (P>>)
V:="レイアウト名"
P:=>>V
```

PAGE SETUP (P>>)
PRINT LAYOUT (P>>)
`セレクションに関するコマンド
P:=>>[aFile]aFiled
APPLY TO SELECTION (P>>="")
N:=1
P:=>>N
GRAPH FILE (P>> ; [aFile]aField ; [aFile]AnotherField)
`レコードに関するコマンド
N:=1
P:=>>1
GOTO RECORD (P>>)
GOTO SELECTED RECORD (P>>)

以下のコマンドの使用には、注意が必要です。

ADD TO SET
GOTO RECORD
GOTO SELECTED RECORD
APPLY TO SELECTION
LOAD SET
SEARCH
SEARCH SELECTION
SEARCH SELECTION BY FORMULA
DIALOG
EXPORT TEXT
EXPORT DIF
EXPORT SYLK
CREATE EMPTY SET
OUTPUT LAYOUT
INPUT LAYOUT
GRAPH FILE
PRINT LAYOUT
PRINT LABEL
IMPORT TEXT
IMPORT DIF
IMPORT SYLK
CREATE SET
SORT SELECTION
PAGE SETUP
COPY NAMED SELECTION
MOVE NAMED SELECTION
REDUCE SELECTION

“ 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ておきたいと思います。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るはずです。

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 個のスクリプトに対する共通プロシージャは、同じステートメントで書かれた n 個のスクリプトより、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が経つと分からなくなります。従って、プロシージャには、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“v変数:=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っ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デフォルトで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。

たとえば、ループのカウンタは常に整数として定義しておくとよいでしょう。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
```

$i がコンパイラ命令で定義されていない場合（$i は実数になります)、実行時間は 19 秒です。$i がコンパイラ命令（**C_INTEGER**）で定義されていると、このループは一瞬のうちに終了します。

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

この関数は整数を返します。

このように書かれていると、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

プロシージャ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。

ボタンは実数ですが、倍長整数に定義できるの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、

私の文字列:="**こんにちは**"

コンパイラは“私の文字列”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：文字を比較する場合は、文字そのものではなく Ascii 関数による値を使用してください。通常の文字比較では、同じ発音の文字など、文字として考えられるものすべてを考慮するからです。

その他のヒント

この節は 2 次元配列、フィールド、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。

たとえば、次のように定義された配列は、

ARRAY INTEGER (配列 ; 5 ; 1000)

以下のような配列よりもうまく処理されます。

ARRAY INTEGER (配列 ; 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。

次のようなプロシージャがあるとします。

```
Case of
  \ ([顧客]地方="関東")
    住所ラベル:="青"
  \ ([顧客]地方="関西")
    住所ラベル:="赤"
  \ ([顧客]地方="九州")
    住所ラベル:="黄"
End case
```

このプロシージャは、以下のように書くと実行速度が速くなります。

```
$地方:=[顧客]地方
Case of
  \ ($地方="関東")
    住所ラベル:="青"
  \ ($地方="関西")
    住所ラベル:="赤"
  \ ($地方="九州")
    住所ラベル:="黄"
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“ Myポインタ ”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とします。次のように書くことができます。

```
Case of
 \ (Myポインタ>>=1)
   '処理1
 \ (Myポインタ>>=2)
   '処理2
 .
 .
End case
```

この **Case of** ステートメントは、以下のように書けばさらに速くできます。

```
$ワーク:=Myポインタ>>
Case of
 \ ($ワーク=1)
   '処理1
 \ ($ワーク=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たくさん必要としません。ローカル変数は、それらが使われるプロシージャに入るときに作られ、プロシージャ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。

コンパイラのメッセージ

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ージは、4D Compiler の開発中に作成されたものです。補足情報については、4D Compiler のプログラムディスクの「追加／修正情報」をご覧ください。

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 (p**ポインタ**>>;a**配列**)

「SELECTION TO ARRAYコマンド中にポインタが存在します。」

SELECTION TO ARRAY (p**ポインタ**>>;a**配列**)
SELECTION TO ARRAY (**[私のファイル]私のフィールド**;p**ポインタ**>>)

「ARRAY TO SELECTIONコマンド中にポインタが存在します。」

ARRAY TO SELECTION (p**ポインタ**>> ; **[私のファイル]私のフィールド**)

「LIST TO ARRAYコマンド中にポインタが存在します。」

LIST TO ARRAY ("**リスト**" ; p**ポインタ**>>)

「ARRAY TO LISTコマンド中にポインタが存在します。」

ARRAY TO LIST (p**ポインタ**>> ; "**リスト**")

「配列定義コマンド中にポインタが存在します。」

ARRAY REAL (p**ポインタ**>> ; 5)

ARRAY REAL (a配列;pポインタ>>)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** (v**変数**))

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。

「このプロシージャはパスワードで保護されています。」

「レイアウト１ページ目の自動動作ボタンの名前がありません。」

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。」

P**ポインタ**>>[[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。」

v**文字列**[[p**ポインタ**>>]]:="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 (a**配列**{p**ポインタ**>>})

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。

「外部ルーチンの呼び出しでパラメータが不足しています。」

SP SELECT CELL (**領域**)

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
- 外部プロシージャ
- 総合エラー

タイプチェック

「変数"vブール"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。」

```
v実数 := 12.3              `v実数は実数
vブール := True            `vブールはブール
v実数 := vブール           `ブールは実数に代入できない
```

「文字列の長さは変更できません。」

```
C_STRING (3 ; v文字列)
C_STRING (5 ; v文字列)
```

「配列の次元数は変更できません。」

```
ARRAY TEXT (a配列 ; 5 ; 5)
ARRAY TEXT (a配列 ; 5)
```

「配列定義コマンドに要素数がありません。」

```
ARRAY INTEGER (a配列)
```

「変数が必要です。」

```
COPY ARRAY (a配列 ; "")
```

「定数が必要です。」

```
C_STRING (v変数 ; v文字列)
```

「"v変数"のタイプが不明です。」
「この変数はプロシージャ"(P)Pプロシージャ"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。」

v**変数** := "**本日は晴天なり**"

「プロシージャ"ｎｎ"が不明です。」

この行は存在しないプロシージャを呼んでいます。

「DISTINCT VALUESコマンドの配列への使用は、文字またはテキストタイプにのみ使用できます。」

DISTINCT VALUES (**ブールフィールド** ; a**整数配列**)

フィールドがインデックス付きで定義されていない場合。

「DISTINCT VALUESコマンドは、インデックス付きの文字・数値フィールドに対してのみ使用できます。」

「文字列の長さは、225文字（バイト）までです。」

C_STRING (325 ; v**文字列**)

「変数"v変数"はプロシージャではありません。」

v**変数**(1)

「変数"v変数"は配列ではありません。」

v**変数**{5} := 12

「結果が式と一致しません。」

v**テキスト** := "**番号**" + **Num** (i)

「タイプが一致しません。」

v**整数** := v**日付** * v**テキスト** `$i**のタイプはテキスト**

「引数${}のインデックスが数値ではありません。」

$i := "3" `$i**のタイプはテキスト**
${$i} := 5

「定数のタイプが無効です：文字。」
a整数配列{"3"} := 4 **`インデックスのタイプがテキスト**

「変数"v変数"のタイプをテキストから配列に変更できません。」

```
C_TEXT (v変数)
COPY ARRAY (vテキスト配列;v変数)
```

「変数"vブール変数"のタイプをブールタイプから実数タイプに変更できません。」

```
v実数:=vブール変数
```

「変数"v変数"のタイプをテキストから数値に変更できません。」

```
v変数 := Num (v変数)
```

「配列"a整数配列"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 (a整数配列;12)
```

“ a整数配列 ” 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。」

```
v変数>> := 5
```

“ v変数 ” 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。」

```
v変数 := 3.5
```

「フィールドの使い方に誤りがあります。」

```
v変数 := [私のファイル]私の日付
```

“ [私のファイル]私の日付 ” は日付フィールドでv変数は数値です。

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。」

```
v変数 := Num ("良い天気です")>>
```

この関数は使用できません。

「シンタックスエラー」

```
If (vブール)
End for                                   `End ifのはず
```

「"{"が多すぎます。」

文中の左カッコ { の数が右カッコ } より多い場合。

「"}"が多すぎます。」

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。

「"("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** (v**変数**))

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 (a**配列**)
a**配列** (2) := 1

「変数が必要です。」

C_INTEGER ([**私のファイル**]**私のフィールド**)

「数値が必要です。」

C_INTEGER (${"3"})

「";"が必要です。」

COPY ARRAY (a**配列**1 a**配列**2)

「文字参照記号が多すぎます。」
「定数タイプが無効です。」

v**文字列**[[3 := "a"

「文字参照記号が多すぎます。」

v**文字列**3]] := "a"

「ここではサブファイルを使えません。」

ARRAY TO SELECTION (a**配列** ; **サブファイル**)

「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 (v**実数**)

“ v実数 ” は数値変数です。

「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。

「プロシージャが複雑すぎます。」

プロシージャ中に 601 以上の異なる Case または 101 以上の If...End if 構造があります。

「フィールドが不明です。」

使用プロシージャが恐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。

「ファイルが不明です。」

使用プロシージャが恐らく他のデータベースからコピーされたもので、存在しないファイ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。」

p**ポインタ** := >>v**変数** + 3

「変数タイプが異なるため代入できません。」
「変数"v変数"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。」

v**実数**[[3]] または v**文字列**[[v**変数**]]

“ v実数 ” が数値変数で “ v変数 ” が数値変数以外の場合。

引数

「結果が式と一致しません。」

Pプロシージャ (**Num** (v**文字列**))

“ Pプロシージャ ” の引数にはブール式が必要な場合。

「タイプが一致しません。」
「このプロシージャに渡す引数が多すぎます。」

DEFAULT FILE (**ファイル** ; **レイアウト**)

「定数タイプが無効です：整数」

Pプロシージャ (3+2)

“ Pプロシージャ ” の引数には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はブール
```

「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 ($i)                `${$i}はテキスト
End for
```

「このコマンドに引数は不要です。」

Current date (v**日付**)

「このコマンドには最低1つの引数が必要です。」

DEFAULT FILE

「変数"v文字列"のタイプはテキストからブールに変更できません。」

Pプロシージャ (v**文字列**)

“ Pプロシージャ ” の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。」

```
P計算 ("3+2")                          `引数のタイプはテキスト
```

プロシージャ “ P計算 ” 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コマンドのパラメータは、変数です。」

COPY ARRAY (v**変数** ; a**配列**)

「引数"$1"のタイプが呼ぶ側のプロシージャと呼ばれる側のプロシージャで一致していません。」

Pプリント ("**レーザーライタ**")

プロシージャ “ Pプリント ” で、$1 が数値の場合。

「タイプが一致しません。」

$1 := String ($1)

「変数"a配列"のタイプは配列からブールに変更できません。」

Pプロシージャ (a**配列**)

配列をプロシージャに渡すときには、その配列のポインタを渡してください。

「引数としてそのプロシージャに渡せません。」

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。」

FIELD ATTRIBUTES (ファイル番号 ; **フィールド番号** ; **タイプ** ; $1)

演算子

「演算とタイプが一致しません」

v**ブール**2 := v**ブール**1 **+ True**

「演算子>は不要です。」

SEARCH ([My**ファイル**] ; [My**ファイル**]My**フィールド** ; >)

「変数"vピクチャ2"のタイプはピクチャから実数に変更できません。」

If (v**数値** = v**ピクチャ**2)

“ v数値 ” が倍長整数で “ vピクチャ2 ” 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。」

v**ブール** := **-False**

外部プロシージャ

「外部ルーチン"ｎｎ"の定義が正確ではありません。」

外部ルーチンの定義に誤りが正しくない場合。

「外部ルーチンに対する引数が足りません。」
「外部ルーチンに対する引数が多すぎます。」

総合エラー

「同じ名称のプロシージャが複数あります：ｎｎ。」

データベースをコンパイルするには、グローバルプロシージャ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xx」
このメッセージが出力されたら、ACI Technical Support に連絡し、エラー番号を教えてください。

「"v変数"のタイプが不明です。」
「この変数はプロシージャ"(P)P処理"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。

「インタープロセス変数の合計サイズが32KBを超えました。サイズ：xxバイト」
インタープロセス変数の数を減らす必要があります。ローカル変数を使うことを検討してください。このメッセージには、配列と固定長文字列（**C_STRING** コマンドで定義された文字列）以外のインタープロセス変数が使用しているメモリサイズが示されます。68000 以外のモードでコンパイルするとこのメッセージは現れません。

「プロセス変数の合計サイズが32KBを超えました。サイズ：xxバイト」
プロセス変数の数を減らす必要があります。ローカル変数の使用を検討してください。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）以外のプロセス変数が使用しているメモリサイズが示されます。68000 以外のモードでコンパイルするとこのメッセージは現れません。

「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。

「プロシージャのサイズが32 KBを超えました。」
プロシージャをもっと小さいプロシージャに分割してください。複雑なコードを書かないでください。これは、Macintoshのみに適用されます。

「オリジナルのプロシージャが壊れています。」
オリジナルのストラクチャでプロシージャが壊れています。該当するプロシージャ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。」
プロシージャが壊れているか、または、使っているコンパイラはそのコマンドが
4th Dimension に追加される前にリリースされたバージョンです。

「レイアウト"Lレイアウト"の変数"v変数"のタイプは変更できません。」
レイアウト内のグラフタイプの変数に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。」
プロシージャか変数いずれかの名前を変えてください。

「関数"String"の名前がレイアウト"入力1"の変数と同じです。」

アクティブオブジェクトの名前を変えてください。

「プロシージャと変数が同じ名前です：Name。」
プロシージャか変数の名前を変えてください。

「外部プロシージャ"Ext1"が"Proc.Ext"ファイルにもストラクチャファイルにもありません」

外部プロシージャがインストールされていません。

「ゼロによる割算(モジューロ)が発生しました。」

v**変数**1 := 0
v**変数**2 := 2
v**変数**3 := v**変数**2 % v**変数**1

「EXECUTEコマンドの引数が誤っています。」

v変数がデータベースに明示的に現れない場合。

「未定義の変数を指しています。」

p**ポインタ** := **Get pointe**r ("v**変数**")

v変数がデータベース内で明示的に定義されていない場合。

「ポインタを使用してタイプを変更しようとしました。」

v**ブール変数** := p**ポインタ>>**

pポインタが整数タイプのフィールドを指す場合。

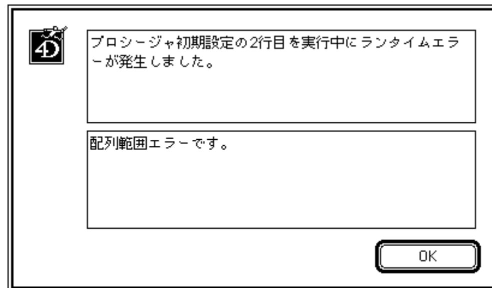
「誤ったポインタが式に使われています。」

v**文字列** := v**文字変数**[[p**ポインタ**>>]]

pポインタが数値タイプの変数を指していない場合。

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。



「結果が変数の範囲を越えました。」

a**整数配列**{1} := 32767
a**整数配列**{1} := a**整数配列**{1} + 12

「配列チェックエラー」

a**配列**{17} := 2.3

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

「ゼロによる割算が発生しました。」

v**変数**1 := 0
v**変数**2 := 2
v**変数**3 := v**変数**2 % v**変数**1

「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。」

v**ポインタ**>> := 5

“ vポインタ ” が初期化されていない場合。

「代入先が小さすぎます。」

C_STRING (5 ; v**文字列**1)
C_STRING (10 ; v**文字列**2)

v**文字列**2 := "ｱｲｳｴｵｶｷｸｹ"
v**文字列**1 := v**文字列**2　`v**文字列**2**は**9**文字ですが、**v**文字列**1**には**5**文字しか格納できない。**

「文字参照エラー。」

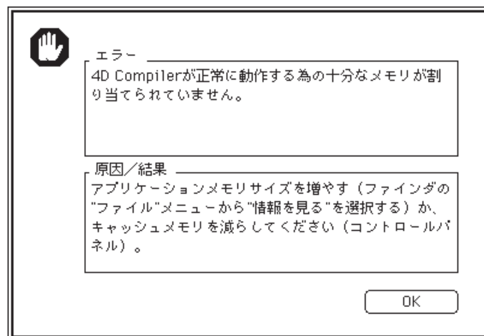
i := 30
v**文字列**[[i]] := v**文字列**2

「パラメータが空の文字列です。」

My**文字列** := ""
v**数値** := **Ascii** (My**文字列**)

コンパイラメッセージ

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 ”のエリアには、その問題を解決する手段が表示されます。

4D Compiler Pro 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 章を参照してください。

このオプションを使用して作成されたアプリケーションは、4D Compiler のデフォルトのアプリケーションアイコンを持っています。このアイコンは、以下に説明する手順でカスタマイズできます。また、4th Dimension のデータベースを使って作成するドキュメント（データファイル、クィックレポート、ラベル、保存した検索条件...）のアイコンもカスタマイズできます。

Macintosh

カスタムアイコンは 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 とも違っていなくてはなりません。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、他の発売された Macintosh アプリケーションで使われていないかを確かめてください。

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ィックレポート、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。

これらの使用ファイルのアイコンをカスタマイズするには、次のようにしてください。

1. ICN#、icl8、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、デスクトップを作り直してください。この操作で、ファインダによってユーザのカスタムアイコンがデスクトップファイルに追加されます。デスクトップの再作成が終了すると、作成したカスタムアイコンが現れます。

注：デスクトップを再構築するには、ファインダの「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、デスクトップを作り直すかどうかを尋ねてきます。

Windows

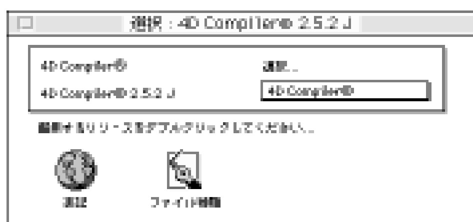
実行形式のアプリケーションをカスタマイズするには、ImageEditor などのグラフィックソフトウエアを利用します。ImageEditor は Microsoft Corp による、Win32™SDK に含まれています。カスタマイズ用のアイコンを作成したら、たとえば「MyDB.ico」のように“ ico ”という拡張子の付いた名前でファイルを保存します。

4D Compiler Pro では、「実行形式アプリケーション」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「MyDB.EXE」ができあがります。「プログラムマネージャ」で、「ファイル」メニューの「プロパティ」アイテムを選択します。「Select Change Icon（アイコン変更）」を選択し、「MyDB.ico」を選択します。これで、実行形式のアプリケーションにカスタムアイコンを割り当てることができます。

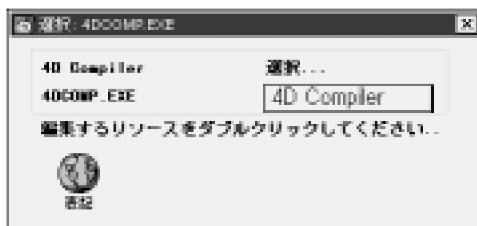
4th Dimension に含まれている Customizer Plus を使用して、4D Compiler のリソースをカスタマイズすることができます。

注：Customizer Plus に関する詳細は、Windows版では、4th Dimension に添付の「Custo.hlp」ヘルプファイルを、Macintosh版では、「Customizer Plus」オンラインドキュメントを参照してください

Customizer Plus で 4D Compiler アプリケーションを選択すると、次のようなウインドウが表示されます。



Macintosh 版



Windows 版

Macintosh 上では、アイコンが 2 個表示されます。「表記」と「ファイル種類」です。Windows 上では「表記」アイコン 1 個が表示されます。

表記

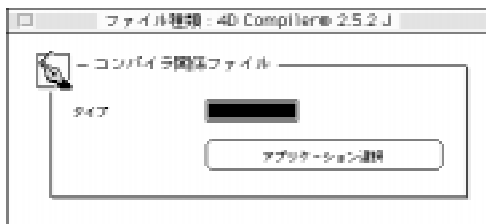
「表記」アイコンをダブルクリックすると、一覧が表示されます。

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す。

ファイル種類（Macintosh のみ）

「ファイル種類」リソースにより、4D Compiler で作成されたテキストファイルを編集するアプリケーションを指定します。

「ファイル種類」リソースを編集するには、「ファイル種類」アイコンをダブルクリックします。次のようなファイルが表示されます。



Creator タイプを知っている場合は直接入力することもできるし、アプリケーションの位置を指定するために「アプリケーション選択」ボタンをクリックすることもできます。4D Compiler は選択されたアプリケーションの Creator タイプを使用します。

「コンパイラ関係ファイル」ウインドウを閉じると、選択結果が保存されます。

記号

%（モジューロ演算子）　6-7
()（丸カッコ）　A-7
[[]]（文字参照記号）　A-7,A-8
{}（中カッコ）　A-6

数値

386/486　2-16
4D Compiler　1-5
　「オプション」ウインドウ　2-17
　コンパイルオプション　2-8 2-21
　…で使用される変数　4-9
　…内のバルーンヘルプ　2-21
　…のインストール　1-8
　パスワードが設定されたデータベース　2-22
　変数のタイプ設定　4-4 4-5
　メニュー　2-2 2-8
4D Compiler Pro　1-4,2-10,2-11,B-1
「4D Compiler」ウインドウ　2-23
　コンパイル処理の進捗状況表示　1-11
　「中止」ボタン　2-23,2-26
　「停止」ボタン　2-23,2-26
4D Compiler製品　1-4
4D Engine　2-10,2-11,B-1
4D Insider　1-4
4D Runtime　vii,1-4
4D SDK　1-4,2-11
68xxx　2-28
68020/30/40プロセッサ　vii

コマンド＆関数

ABORTコマンド　6-6
ACCUMULATEコマンド　6-7
ADD TO SETコマンド　6-14
ALERTコマンド　A-3
Append document関数　6-7
APPLY TO SELECTIONコマンド　6-14
ARRAY INTEGERコマンド　A-4
ARRAY POINTERコマンド　6-10
ARRAY REALコマンド　A-2
ARRAY STRINGコマンド　4-11
ARRAY TEXTコマンド　A-4,A-6
ARRAY TO LISTコマンド　A-2
ARRAY TO SELECTIONコマンド　A-2
Ascii関数　6-8,7-3,A-14
BREAK LEVELコマンド　6-7
Case of構文　6-5,6-12,7-4,A-8
CLEAR VARIABLEコマンド　6-8,6-10
COPY ARRAYコマンド　6-2,A-2,A-4,A-7
Create document関数　6-7
Current Date関数　A-9
C_STRINGコマンド　4-10
DEFAULT FILEコマンド　A-9
DIALOGコマンド　6-14
DISTINCT VALUESコマンド　6-3
EXECUTEコマンド　6-8,6-12
　…の使用を避ける方法　6-12
EXPORT TEXTコマンド　6-14
Field関数　6-8
FIELD ATTRIBUTESコマンド　4-7
File関数　6-8
For...End for構文　1-6
Get Pointer関数　6-8,6-10
GOTO RECORDコマンド　6-14

GOTO SELECTED RECORDコマンド 6-14
IDLEコマンド　6-6
If...End if構文　A-7,A-10
LIST TO ARRAYコマンド　6-2,6-3,A-2
LOAD SETコマンド　6-14
LOAD VARIABLEコマンド　6-8,6-9
Mod関数　6-7
Modified関数　A-7
NO TRACEコマンド　6-8,6-12
Num関数　A-6,A-8
ON EVENT CALLコマンド　6-6
ON SERIAL PORT CALLコマンド　6-6
Open document関数　6-7
RECEIVE VARIABLEコマンド　6-4,A-9
SAVE VARIABLEコマンド　6-8,6-9
SEARCHコマンド　6-14,A-10
SEARCH BY FORMULAコマンド　6-14
SEARCH SELECTIONコマンド　6-14
SELECTION TO ARRAYコマンド　6-2,6-3,A-2
SEND VARIABLEコマンド　6-4
String関数　A-9
Subtotal関数　6-7
TRACE コマンド　6-8,6-12
type関数　6-5
Undefined関数　6-8,6-9,A-2

ア,あ

暗黙のタイプ変更　5-3

イ,い

インタープロセス変数　3-2,4-2
インタプリタモード　1-3,1-5,1-6,1-13,2-22,4-3,5-13,6-8,6-10
　実行速度　1-7

ウ,う

WIN4DXディレクトリ（フォルダ）　2-28

エ,え

エラー
　…の表示　2-26
エラーファイル　1-5,1-7,2-11,2-14,3-4 3-9,A-4
　警告　3-7
　総合エラー　3-5 3-6
　テキストとしての…　3-8
　特定行についてのエラー　3-6
　…内の警告　6-4
　…の構成　3-8
　…の使用　3-7 3-9
　…の対話的な使用　3-8
「エラーファイル参照終了」メニューコマンド 2-2,3-8 3-9
エラーメッセージ A-4 A-12
演算　6-7
演算子　A-10

オ,お

「オプション」ウインドウ　1-10,2-5,2-17,B-1
　「コンパイルパス」アイコン　2-19
　「最適化」アイコン　2-17
　「自動バージョン番号付け」アイコン　2-21
　「タイプファイル」アイコン　2-18
　「デフォルト数値タイプ」アイコン　2-20
　「デフォルトボタンタイプ」アイコン　2-19
　「デフォルト文字タイプ」アイコン　2-20
　「ローカル変数初期化」アイコン 2-18

カ,か

カーネルコール　6-6
外部プロシージャ　2-22,A-10,A-12
カウンタ　4-7,7-2

Customizer Plus　3-1,C-1
　「表記」リソース　C-2
　「ファイル種類」リソース　C-2
空の文字列　6-8

キ,き

機械語　viii,1-3

ク,く

クイックレポート変数　5-15
グローバルプロシージャ
　シンボルテーブル内の…　3-4

ケ,け

警告　2-1,3-7,A-2
　…の表示　2-26
警告メッセージ　3-12

コ,こ

コーディングに不備のあるプロシージャ　1-2
構造化プログラミング　7-1
コンパイラ命令　2-24,4-4,4-5 4-7, 4-7 4-11,5-2,5-3,7-2
　C_BOOLEAN　4-5
　C_DATE　4-5
　C_GRAPH　4-5
　C_INTEGER　4-5,4-8,7-2,A-7,A-8
　C_LONGINT　4-5,4-8,7-2
　C_PICTURE　4-5
　C_POINTER　4-5
　C_REAL　4-5
　C_STRING　4-5,4-8,4-10,5-3,7-3,A-4, A-5,A-11,A-13
　C_TEXT　4-5
　C_TIME　4-5
　インタプリタでの使用　4-8,4-8 4-9
　…が必要な場合　4-6 4-7
　…でのコードの最適化　4-7,7-2 7-5
　…の配置　4-9
　…を手動で識別する　4-9
　…をどこで使用するか　4-9
コンパイラメッセージ　A-14
コンパイルオプション　2-8 2-21
　エラーファイル　2-11
　警告　2-14
　コンパイル後のデータベース名　2-9
　シンボルテーブル　2-12
　実行形式アプリケーションの作成 2-10
　スクリプトマネージャ　2-13
　範囲チェック　2-13
　プロセッサの種類　2-14 2-17
　コンパイル処理　2-22 2-23
　コンパイラ命令タイプチェック処理　2-24
　コンパイル時間の短縮　4-7
　コンパイルパス　2-25
　データベースのコピー　2-23
　データベースの…　1-2
　…の開始　2-22
　変数タイプチェック処理　2-24
　ローカル変数タイプチェック処理 2-24
　割り込み　2-26
コンパイルパス　2-25
「コンパイルパス」アイコン　2-19
コンパイルモード　1-3 1-4
　実行時間　1-7

サ,さ

再コンパイル
　プロジェクトを使用した…　2-6
最適化アイコン　2-17

シ,し

システム変数　5-14
シンタックスエラー　A-6 A-8
シンボルテーブル　ix,3-1,3-2 3-4,4-3
　…とは　4-3
　プロセス変数一覧　3-2
　ローカル変数一覧　3-3
「実行形式アプリケーション作成」オプション　2-1,2-10,2-24,B-1

バージョン番号の自動生成　2-21

ス,す

数値演算コプロセッサ　viii
数値変数　1-2
スクリプトマネージャ　2-1,2-13
Startupプロシージャ　1-6,4-9
ストラクチャへのアクセス　6-8

ソ,そ

総合エラー　A-10 A-12

タ,た

タイプチェック処理
　数値変数　4-4,7-2 7-3
　テキスト変数　4-7
　文字列　7-3
タイプチェック処理フェーズ　2-24 2-25
　…のまとめ　2-25
タイプファイル　3-1,3-10
「タイプファイル」アイコン　2-18
対話型デバッグ　vii,1-5,1-7,2-6,2-11,3-8,3-9
　エラーファイルの名前　2-12
ダブルクリックで起動するアプリケーション
　…のカスタムアイコン　B-1
　…の標準アイコン　2-27

チ,ち

「中止」ボタン
　「4D Compile」rウインドウ内の… 2-23

ツ,つ

通信（コミュニケーション）　6-4 6-5
「次...」ボタン　2-17

テ,て

「停止」ボタン
　「4D Compiler」ウインドウ内の… 2-23
データ入力　6-5
データベース
　コンパイルした…の使用　1-12
デフォルト数値タイプ　2-20
デフォルトプロジェクト　2-7
デフォルトボタンタイプ　2-19
デフォルト文字タイプ　2-20

ト,と

ドラッグ＆ドロップ　2-27

ハ,は

配列　2-24,6-2 6-4,A-13
　暗黙のタイプ変更　5-6
　1次元のコマンド　6-3
　シンボルテーブル内の…　3-2
　次元数の変更　5-5
　2次元…　7-4
　…の次元変更　6-2
　文字…　5-6
配列関連コマンド
　ポインタの使用　6-4
配列定義ステートメント　2-24
配列変数
　…のタイプ矛盾　5-5
範囲チェック　ix,2-1,2-13,3-1,3-10 3-12,5-9,6-8
バルーンヘルプ
　Macintoshの…　2-21
パスワード　2-22
Power Macintosh　2-16,2-28
PowerPC　2-16,2-28
　…用のコンパイル　2-28

ヒ,ひ

引数　A-8 A-9
引数の受け渡し　5-11
「表記」リソース
　Customizer Plus　C-2

フ,ふ

「ファイル種類」リソース　C-2
「ファイル」メニュー　2-3 2-8
　「再コンパイル」コマンド　2-6
　「新規」コマンド　2-3
　「新規保存」コマンド　2-7
　「デフォルトプロジェクト」コマンド　2-7
　「閉じる」コマンド　2-7
　「開く」コマンド　2-4
　「復帰」コマンド　2-7
　「保存」コマンド　2-7
ファインダ
　「情報を見る」メニューアイテム　2-21
フィールド　7-4
複数シンタックスコマンド　4-7
ブレーク処理　6-7
　コンパイルされたデータベース内での…　6-7
プロジェクト　1-11,2-1,2-4
　デフォルト…　2-7
　…の概念　2-4 2-6
プロセス変数　3-2,4-2
　シンボルテーブル内の…　3-2 3-3
プロセス変数とインタープロセス変数
　タイプ矛盾　5-2 5-4
プロセッサ
　386/486　2-16
　680xx　2-28
　Motorola 68xxx　2-15
　Motorola PowerPC　2-16
　Pentium　2-16
　PowerPC　2-16
文書　6-7

ヘ,へ

ヘルプ
　Macintoshのバルーン…　2-21
　Windowsの…　2-22
変数
　4D Compilerによる使用　4-9
　…タイプ　4-2
　ローカル…　7-5
変数とその他　6-8
変数のタイプ定義　4-4 4-5
　実数と文字列　4-8
Pentium　2-16

ホ,ほ

包括引数　5-13,A-9
ボタン　7-3
ポインタ　4-6 4-7,5-8,6-4,6-5,6-8,6-10,6-13 6-14,7-4,A-2 A-3,A-6,A-8,A-13
　タイプ変更を避けて使用する 5-11 5-13

マ,ま

Mac4DXフォルダ　2-28

メ,め

「メイン」ウインドウ　2-4,2-9

モ,も

「モード」メニュー
　「エラーファイル参照終了」メニューコマンド　3-8 3-9
　「コンパイラエラー」メニューコマンド　1-5,3-8 3-9
文字変数　5-4
文字列　1-2,6-8
Motorola 68xxx　2-15
Motorola PowerPC　2-16

ヨ,よ

予約変数　5-14

ル,る

ループカウンタ　4-7,7-2

レ,れ

レイアウト変数　5-7 5-8
　外部オブジェクト　5-7
　グラフ　5-7
　サーモメータ　5-7
　スクロールエリア　5-8
　単純変数　5-8
　ダイアル　5-7
　チェックボックス　5-7
　ボタン　5-7
　ポップアップメニュー　5-8
　ルーラ　5-7
例外処理コマンド　6-6
ResEdit™　B-1

ロ,ろ

ローカル配列　5-6
ローカル変数　3-2,4-7,4-9,5-12,7-4,7-5,
A-9,A-13
　シンボルテーブル内の…　3-3
　…のタイプ設定　2-24
　…のタイプ矛盾　5-4
「ローカル変数初期化」アイコン　2-18