4D Compiler 6.7

リファレンス

Windows® and Mac™ OS

4D

4D Compiler 6.7 リファレンス
Windows® and Mac™ OS

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4D SAは一切の責任を負いかねます。このマニュアルで説明されるソフトウェアは、本製品に同梱のLicense Agreement（使用許諾契約書）のもとでのみ使用することができます。

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

4th Dimension、4D Server、4D、4Dロゴ、およびその他の4D製品の名称は、4D SAの商標または登録商標です。

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

Apple, Macintosh, Mac, Power Macintosh, Laser Writer, Image Writer, ResEdit, 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。

注意

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

目次

目次

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

本マニュアルについて

本マニュアルは、4D Compilerのアプリケーションについて説明しています。

各章について

本マニュアルには下記の章があります。

- **第1章「はじめに」**：コンパイラの紹介、およびコンパイラとインタプリタの違いについて説明します。
- **第2章「コンパイラの使用」**：4D Compilerの特長とオプションについて説明します。
- **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4つのコンパイラツール、“シンボルテーブル”、“エラーファイル”、“範囲チェック”、“タイプファイル”について説明します。
- **第4章「コンパイル用データベースの準備」**：コンパイルするデータベースを作成する際の規則について説明します。
- **第5章「タイプ設定ガイド」**：タイプ（型）の矛盾を引き起こす場合の例を説明します。
- **第6章「シンタックスの詳細」**：コンパイル関連のコマンドについての追加情報を説明します。
- **第7章「最適化のヒント」**：コードのパフォーマンスを最適化するためのヒントを紹介します。

付録では、コンパイラの補足説明をしています。

■ **付録A「コンパイラメッセージ」**：コンパイラが表示するメッセージの完全なリストをサンプルコードと共に紹介します。

■ **付録B「アプリケーションのカスタマイズ」**：アプリケーションをカスタマイズする際の情報を提供します。

■ **付録C「リソースのカスタマイズ」**：4D Compilerをカスタマイズする際のCustomizer Plusの使用法を説明します。

このマニュアルの使用方法

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

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

ハイパーテキストナビゲーション

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

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

また、マニュアルのページを表示しているウインドウの左側にある「しおり」をクリックすることによりドキュメント中を移動することもできます。

規約

クロスプラットフォームについて

本マニュアルは、4D Compilerの Windows版およびMacintosh版の両バージョンに関する情報をカバーしています。それぞれのバージョンの概念や機能はほぼ同一ですが、必要な差異については記されています。また、本マニュアル中の画面表示はWindows環境での4D Compilerによるものです。Macintosh版と Windows版で大幅に異なる場合には、両バージョンの画面表示を掲載しています。

記載に関する規約

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

次のような表記が使用されています。

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

このような注意書きは、重要な情報を表わしています。

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

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

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

■ 4D Compilerの長所

■ 4D Compilerを最適に使用するためのヒント

■ インタプリタとコンパイルの根本的な違い

■ 4D Compiler製品の説明

■ 4D Compilerにおける4th Dimensionの対話式デバッグのサポート

■ なぜデータベースをコンパイルするのか

■ コンパイルされたデータベースの使用

4D Compilerの長所

4D Compilerには次のような長所があります。

■ データベースを系統的に分析し、広範囲に渡る診断警告やエラーメッセージを提供します。

■ コンパイルされたデータベースの実行中に、データベースの動的チェックが可能です。

■ 4th Dimensionとの対話式デバッグをサポートします。

■ メソッドをコンパイルし、真のマシン語を生成します。

■ 次のようなプロセッサ用に最適化されたコードを生成します：
Motorola PowerPCプロセッサ（PowerPC 601/603/604/G3）およびPCプロセッサ（386/486、ペンティアム最適化）

■ コンパイルされたデータベースを4D Engineとマージし、スタンドアロンの実行形式アプリケーションを作成することが可能です。

■ データベースで使用されるすべての変数とそのコンパイラ命令のリストを生成します。

■ 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を設定します。

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

4D Compilerは、機械のマイクロプロセッサに適応した真の機械言語を発生させます。

4D Compilerの最適な使用

4D Compilerの最適な使用を行うためには、下記の点にご注意ください。

■ 効率良く記述されていないためにインタプリタでのパフォーマンスがよくないメソッドは、コンパイルしても速度が向上しない場合があります。また、そのようなコードはコンパイルできないこともあります。エラーを検出すると、4D Compilerはコンパイルを中止します。

■ 4th Dimensionで正常に作動するデータベースが常にコンパイル可能であるとは限りません。4th Dimensionのインタプリタは、意味や文法の矛盾に対してある程度寛容です。一方、4D Compilerは、コンパイラとしてはかなり融通がききますが、インタプリタほど柔軟ではありません。これは、インタプリタとコンパイラの本質的な違いによるものです。

コンパイラは、4th Dimensionのインタプリタとは異なり、それぞれの変数が必ず1つのデータ型に割り当てられていなければなりません。コンパイラは、それぞれの変数を正確にタイプ分けする必要があります。タイプの割り当てに関して不明確であることは許されません。例えば、あるステートメントでは実数として使用されている変数が、他のステートメントではテキストとして使用されている場合、コンパイラはエラーメッセージを出力します。

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

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

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

インタプリタ対コンパイラ

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

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

これを行うには2つの方法があります。

■ **ステートメントを1行ずつ翻訳しながら実行する**：これがインタプリタです。

■ **ステートメント全体をプログラムの実行前に翻訳する**：これがコンパイラです。

インタプリタ

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

■ プログラム言語で書かれたステートメントを1つ読み込みます。

■ ステートメントをマシン語に翻訳します。

■ ステートメントを実行します。

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

コンパイラ

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

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

コンパイラの利点は、「なぜデータベースをコンパイルするのか」の節で説明します。

4D Compiler（MacintoshおよびWindows）

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

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

4D Compilerと4th Dimension

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

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

対話式デバッグ

4D Compilerの特徴的な機能の1つには、データベースのデバッグ時にエラーファイルを対話式に使用することができるという点があります。エラーファイル、およびコンパイルされていないデータベースの両方を、4th Dimensionで同時に開くことができます。4th Dimensionのメニュー「次のコンパイルエラー」は、自動的に次のコンパイルエラーを見つけてハイライトさせ、警告やエラーメッセージを表示します。

なぜデータベースをコンパイルするのか

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

■ 体系的なコードチェック

■ データベースの保護

実行速度

実行速度の向上は、コンパイルされたコードの2つの特徴によるものです。

■ 一度だけで全体をカバーする、直接のコード翻訳

■ 変数および方法のアドレスへの直接アクセス

直接的で最終的なコード翻訳

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

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

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

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

もう1つの例を挙げましょう。データベースにOn Startupデータベースメソッドが含まれていると仮定します。このメソッドは、データベースを起動するたびに実行されます。しかし、一度コンパイルすれば、On Startupデータベースメソッドを毎回翻訳するのに必要な時間を省略することができます。

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

変数アドレスとメソッドアドレスへのダイレクトアクセス

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

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

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

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

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

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

For ($i;1;10000)
  $x:=Substring([Table1]first;1;5)
  NEXT RECORD([Table1])
End for
	1分42秒	2.17秒	47倍

コードのチェック

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

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

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

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

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

コンパイル後のデータベースの使用

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

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

データベースの保護

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

■ データベースのストラクチャが、故意あるいは事故によって変更されません。

■ メソッドがプロテクトされているので、参照または解析することができません。

コンパイラの使用　2

この章では4D Compilerの使用方法について説明します。

■ 4D Compilerのメニュー

■ オプションウインドウで使用できる機能とオプション

■ コンパイルの手順

■ 4th Dimensionとコンパイラの同時使用

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

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

プロジェクトを使用すると下記のことが可能です。

■ コンパイル後のストラクチャファイルの名前の保存

■ 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

■ 実行形式アプリケーション作成、範囲チェック、警告、スクリプトマネージャとの互換性、ターゲットマシンのマイクロプロセッサなど、さまざまなオプションの指定

■ アクティブオブジェクト、数値、文字列のデフォルトタイプの指定

■ デバッグに役立つローカル変数初期化オプションの指定

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

データベースの使用

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

4th Dimensionで開かれているデータベースをコンパイルするには、デザインモードからユーザモードまたはカスタムモードにした後、コンパイラに切り替えます。エラーファイルを4th Dimensionで使用している場合には、デザインモードのメニューで「エラーファイル参照中止」を選択してエラーファイルを閉じてください。

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

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

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

4D Compilerのメニュー

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

■「アップル」メニュー

■「ファイル」メニュー

■「編集」メニュー

「アップル」メニューには、「4D Compilerについて...」というコマンドがあります。このメニュー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、すべてのMacintoshのアプリケーションに必要なメニューです。4D Compilerでは使用しません。

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

■「ファイル」メニュー

■「編集」メニュー

■「ヘルプ」メニュー

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

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

コンパイラに関するメニューを含んでいるのは「ファイル」メニューだけです。下記の9つの項目が含まれます。

■ 新規...

■ 開く...

■ 再コンパイル

■ 閉じる

■ 保存

■ 名前をつけて保存...

■ 元に戻す

■ デフォルトプロジェクト...

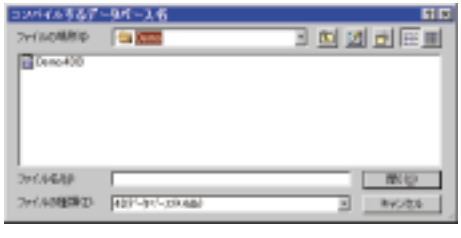
■ 終了

新規

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

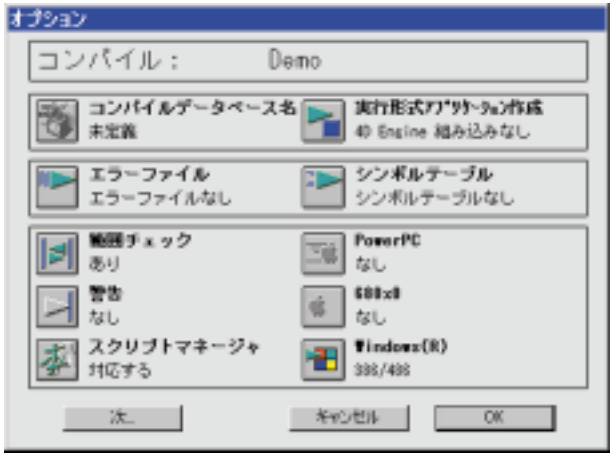
このメニューを選択すると、コンパイラはプラットフォームに応じて標準の「開く」ダイアログボックスを表示します。このダイアログボックスを使用してコンパイルするデータベースを開きます。

Windows



Macintosh

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



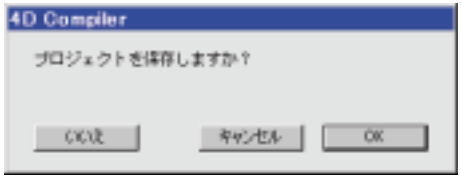
オプションウインドウ内のそれぞれのアイコンは、コンパイルオプションを表しています。これらのオプションついては、この章の「コンパイルオプション」の節を参照してください。

開く

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

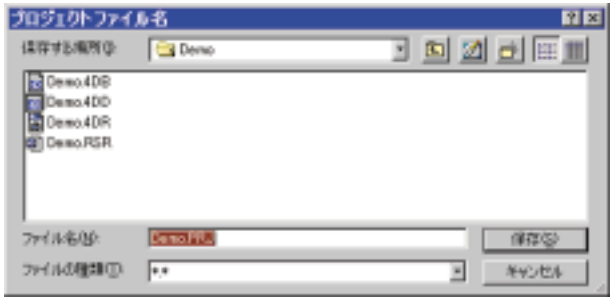
プロジェクトの概念

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



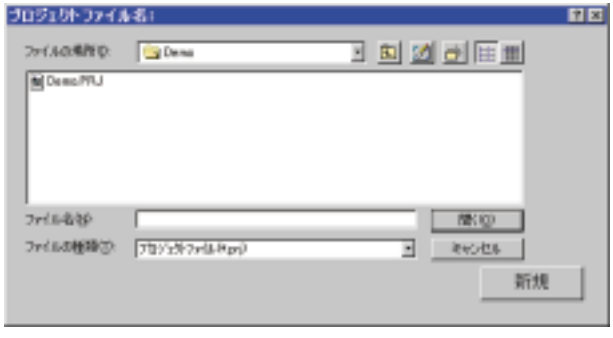
オプションの設定を保存しておくと、同じデータベースを同じコンパイルオプションで素早く再コンパイルすることができるため、非常に便利です。

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



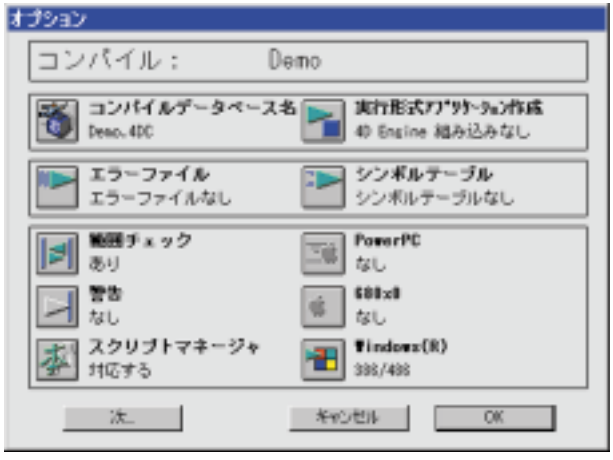
プロジェクトを開く

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



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

プロジェクトを開いた場合、4D Compilerは「オプション」ウインドウを表示し、そのプロジェクトでのすべて設定が表示されます。



プロジェクトを使用してデータベースを再コンパイルする

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

これらのファイルについての詳細は、この章の「コンパイルオプション」の節を参照してください。

再コンパイル

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

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

閉じる

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

保存

前回保存されたものに上書きして現在のプロジェクトを保存するには、「保存」メニュー項目を使用します。

名前をつけて保存

新しいプロジェクトを保存するか、あるいは現在のプロジェクトを別の名前で保存するには、「名前をつけて保存...」メニューを使用します。このメニューにより、「保存」ダイアログボックスを表示し、現在のプロジェクトに新しい名前をつけて保存することができます。

元に戻す

現在のプロジェクトを前回保存したものに戻す場合には、「元に戻す」メニューを使用します。

デフォルトプロジェクト

このメニューにより、新規プロジェクトにデフォルトで適用するオプションを定義することができます。

▼ デフォルトオプションを指定するには：

1.「ファイル」メニューからデフォルトプロジェクトを選択する。

メインの「オプション」ウインドウが表示され、デフォルトプロジェクトが定義可能になります。ここで選択したオプションは、新規プロジェクトを作成するたびに適用されます。

デフォルトプロジェクトの設定は、4D Compilerの初期設定ファイルに保存されます。このファイルは、Macintosh上では“システムフォルダ：初期設定フォルダ：4Dフォルダ”内に保存されます。Windows上では“Windowsフォルダ\4Dフォルダ\4DComp.PRFファイル”に保存されます。

デフォルトプロジェクト用のオプションは、データベース用の特定のプロジェクト用のオプションと同様に選択されます。唯一の違いは、「ファイル作成」ダイアログボックスが、デフォルトプロジェクトを使用するデータベース用に使用される接尾語（サフィックス）を入力するエリアに置き換わっていることです。

2.「コンパイルデータベース名」ボタンをクリックする。

入力エリアが表示されます。

Windows

Macintosh

3. 拡張子を入力する。

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

4.「保存」ボタンをクリックして、デフォルトプロジェクトを保存する。

デフォルトプロジェクトは必要に応じて変更できます。例えば、あるデータベースだけをスクリプトマネージャ対応にする場合、デフォルトプロジェクトでこのオプションを設定しておく必要はありません。デフォルトプロジェクトを利用して新規にプロジェクトを作成した後で、スクリプトマネージャ対応に設定することができます。

この変更は、作業しているデータベースのプロジェクトに保存されますが、デフォルトプロジェクトそのものは変更されません。

5. その他の拡張子を設定する。

エラーファイルの拡張子は変更しないことをおすすめします。変更した場合、4Dで対話式デバッグが使用できなくなります。エラーファイルについての詳細は、第3章の「エラーファイルの使用」の節を参照してください。

▼ デフォルトプロジェクトを変更するには：

■「ファイルメニュー」から「デフォルトプロジェクト」を選択する。

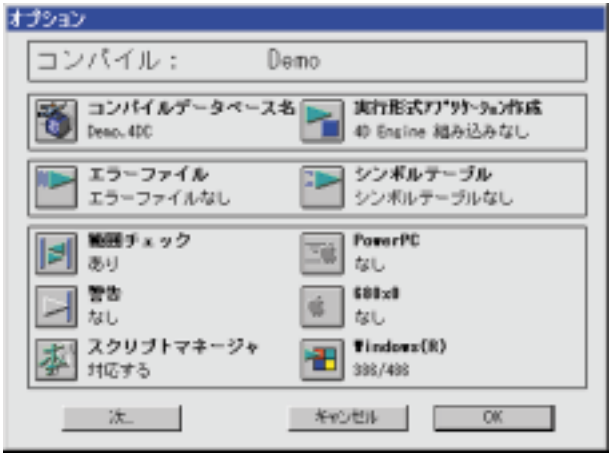
終了

4D Compilerを終了するには、「終了」メニューを選択します。

コンパイルオプション

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

「オプション」ウインドウ



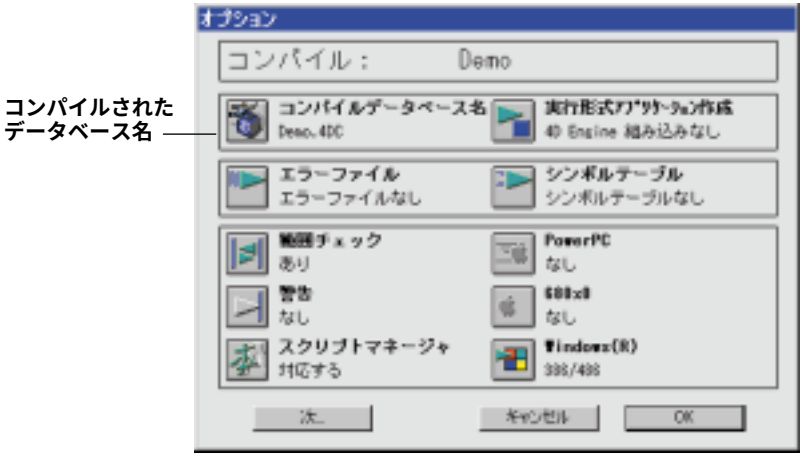
メインの「オプション」ウインドウには9個のアイコンがあり、それぞれが選択または解除されるオプションを表わしています。コンパイルするデータベースに応じて、これらのオプションのいくつか、あるいはすべてを使用することができます。アイコンをクリックして、動作の1つを選択します。2回目にクリックすると、最初の設定をキャンセルします。設定に応じてアイコンの表示が変化します。

各オプションについて説明します。

コンパイルデータベース名

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

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



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

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

実行形式アプリケーションの作成

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

Windows プラットフォーム

コンパイルを開始する前に、下記のファイルがハードディスクの同じフォルダ内にコピーされていることを確認してください。

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

▼ 実行形式アプリケーションを作成するには：

1.「実行形式アプリケーション作成」ボタンをクリックする。

4D Compilerは「開く」ダイアログボックスを表示します。

2. 4D Engineを選択して、「開く」ボタンをクリックする。

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

コンパイルを行なう前に、ストラクチャファイルと同じフォルダ内に“ DISTRIB ”フォルダが作成されます。コンパイル実行中、コンパイラは4D Engineとコンパイル後のストラクチャを使用し、実行形式アプリケーションを作成します。

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

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

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

▼ 実行形式アプリケーションのインストールを完了するには：

■“ DISTRIB ” フォルダ内に次のコンポーネントを置きます。

■ データファイルである “ データベース名.4DD ”

■ WIN4DXおよびMAC4DXフォルダ（プラグインを使用している場合）

▼ データベースを配付するには：

■ 4D社の「4D Install Maker」等の市販のインストーラを使用することができます。このタイプのインストーラには下記のような利点があります。

■ ユーザが標準あるいはカスタマイズのインストールを選択できます。

■ アイコンをカスタマイズできます。

■ インストール時に複数枚のディスクを使用できます。

Macintosh プラットフォーム

▼ 実行形式アプリケーションを作成するには：

1 「実行形式アプリケーション作成」ボタンをクリックする。

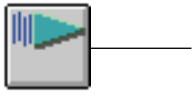
4D Compilerは「開く」ダイアログボックスが表示します。ここで、マージする4D Engineを選択することができます。

2 4D Engineを選択して、「開く」をクリックする。

メインの「オプション」ウインドウに戻ると、選択した4D Engineの名前が表示されます。コンパイルの実行中、コンパイラは4D Engineとコンパイル後のストラクチャを使用し、ダブルクリック可能な単体の実行形式アプリケーションを作成します。Macintoshでは、4D Engineとコンパイル後のストラクチャファイルが1つのファイルにマージされます。

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

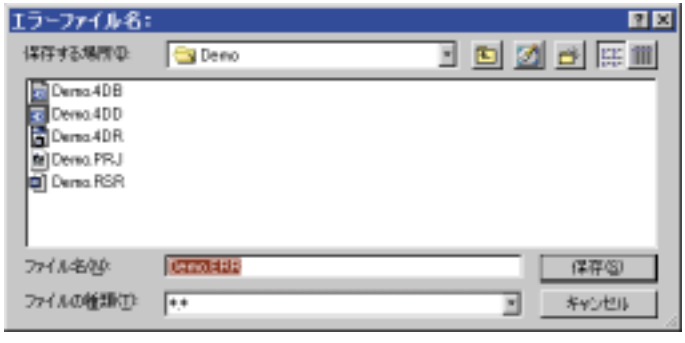
エラーファイル



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

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

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



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

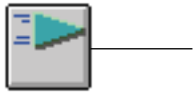
このファイルは、次の2通りに使用できます。

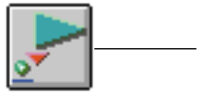
■ テキストファイルとして、コンパイラで出力されたメッセージの記録を保存できます。

■ 4th Dimensionでデータベースをリアルタイムにデバッグできます。

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

シンボルテーブル

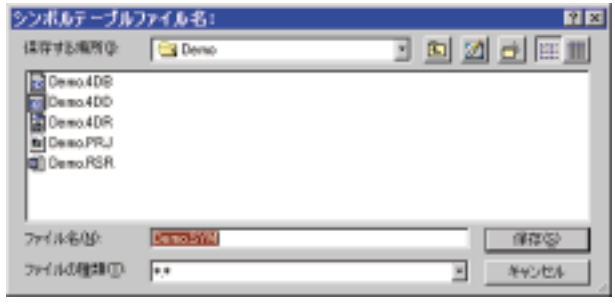




データベースのシンボルテーブルをテキストファイルで出力するには、このオプションを使用します。このファイルには、コンパイルされるデータベース内のすべてのオブジェクトに関する情報が含まれています。これには、プロセス変数、インタープロセス変数、ローカル変数、これらのデータタイプ、およびデータタイプが決定されたメソッド名が含まれています。

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

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



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

シンボルテーブルについての詳細は、第3章を参照してください。

範囲チェック

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

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

スクリプトマネージャ

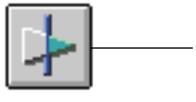
Macintoshではスクリプトマネージャ、Windowsでは2バイトのシステムに対応している4th Dimensionでデータベースを実行する場合に、このオプションを使用します。

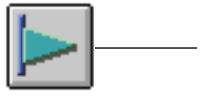
スクリプトマネージャとは、日本語、中国語、アラビア語、ヘブライ語などの、アルファベットやローマン以外の文字でデータベースの使用を可能にするものです。スクリプトマネージャ（Macintosh）や2バイトのシステム（Windows）上で稼動する4th Dimensionでデータベースを使用する場合には、このオプションを選択してください。

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

注：日本語環境では、「対応する」を選択してください。

警告





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

次のような場合、警告メッセージが非常に有効です。

■ すでにコードがコンパイル可能な状態であるが、コードの質をさらに高めたい場合。

■ エラーではなくても、コンパイラが完全に評価できないようなステートメントが含まれている場合、コンパイラはコードをもとに類推を行うが、その推定が正しいかどうか確認したい場合。

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

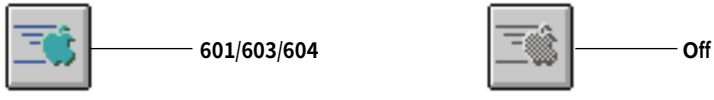
プロセッサの種類

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

■ Motorola PowerPC (PowerMacintosh)

■ 386/486及びPentium（Windows）

Motorola PowerPC（Power Macintosh）



このバージョンで生成されるPPCのコードは、Power Macintoshに搭載されているPower PC 601、603、604、G3と互換性があります。

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

PC(Windows)

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

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

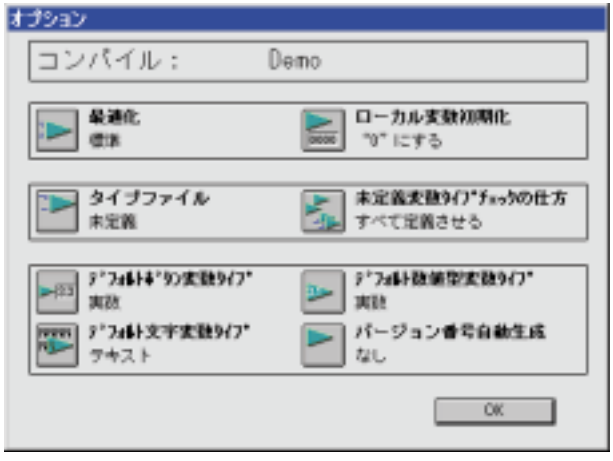
コンパイル後のデータベースを4D Serverとさまざまなクライアントで使用する場合には、最も高性能なコンパイルオプションを選択します。つまり、Pentiumと486の両方のクライアントが存在する場合、「Pentium最適化」を選択してください。

次のオプションから選択できます。

実行するマシンのオプション	PPCのオプション	Windowsのオプション
PPC	On	Off
386/486 PC	Off	386/486
Pentium	Off	Pentium optimized
PPC and 386/486 PC	On	386/486
PPC and Pentium	On	Pentium optimized
Pentium and 386/486 PC	Off	Pentium optimized
PPC, Pentium and 386/486 PC	On	Pentium optimized

追加オプション

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



各オプションの詳細を紹介します。

最適化

4D Compilerでは、2種類のコード生成方法を選択できます。

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

ローカル変数初期化

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

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

■“0”**にする**：ローカル変数を作成し、値をすべて空（文字列は空の文字列、数値は0）にします。

■ **なし**：ローカル変数を作成し、初期化は行いません。コード内で変数を正しく初期化している場合、このオプションを選択するとデータベースの実行が最適化されます。これは、初期化に要する時間が節約できるためです。

■ **ランダム値にする**：ローカル変数を作成し、ランダム値に初期化します。4D Compilerは、常に一定のランダム値で変数を初期化します。この意図的な「バグ」により、データベースの実行時に意図しない動作が誘発され、初期化し忘れたローカル変数を認識することができます。このオプションは、データベースの開発段階で使用します。

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

▼ 使用例：

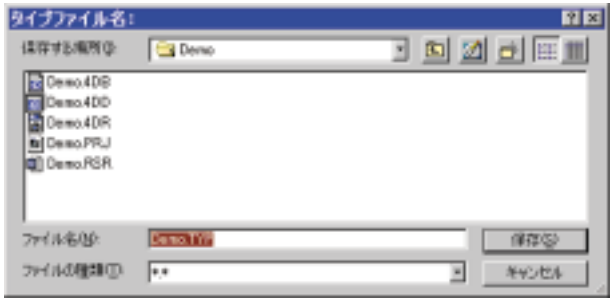
```
   `プロジェクトメソッド
   `...
C_LONGINT($vlVar)
   `...
   `（ここでは、$vlVarは常に明示的に初期化されるものとする）
   `...
Case of
   \($vlVar=0)
      `ステートメントAを実行...
   \($vlVar=1)
      `ステートメントBを実行...
   \($vlVar=2)
```

```
        `ステートメントCを実行...
else
        `ステートメントDを実行...
End case
```

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

タイプファイル

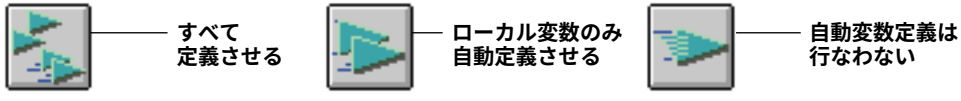
このオプションを選択すると、コンパイラはすべての変数を定義するコンパイラ命令を含むファイルを作成します。この定義をデータベースで使用する場合には、コンパイラ命令を1つあるいは複数のメソッドにコピーします。アイコンをクリックすると、「名前をつけて保存」ダイアログボックスが表示されます。



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

コンパイラ命令の使用に関する詳細は、第4章を参照してください。

コンパイルパス（未定義変数タイプチェックの方法）



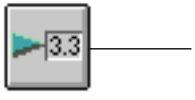
■ **すべて定義させる**：4D Compilerはコンパイルに必要なすべての段階を実行し、明示的に宣言されていないすべての変数のタイプを推測します。

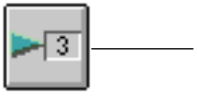
■ **ローカル変数のみ自動定義させる**：このオプションは、すべてのプロセス変数とインタープロセス変数のタイプが宣言されている場合に適切なものです。この宣言は、開発者が自分で行うか、あるいはタイプファイルの内容をメソッドに貼り付けして行います。このオプションが選択されていると、コンパイラはタイプの設定をスキップすることができるため、コンパイル時間が速くなります。コンパイルは、データベースの複製、コンパイル、そしてファイル生成の順に行なわれます。

■ **自動変数定義は行なわない**：すべてのプロセス変数、インタープロセス変数、およびローカル変数のタイプが宣言されている場合には、このオプションを使用します。この設定によりコンパイル時間は短縮されますが、作成されるコードには影響を及ぼしません。

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

デフォルトボタン変数タイプ





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

■ チェックボックス、3Dチェックボックス

■ ボタン、ハイライトボタン、透明ボタン、3Dボタン、ピクチャボタン、ボタングリッド、ラジオボタン、ラジオピクチャ、3Dラジオボタン

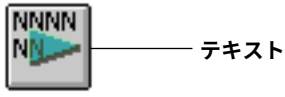
■ ピクチャメニュー、階層ポップアップメニュー、階層リスト

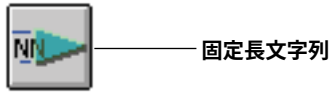
デフォルトボタン変数タイプを「倍長整数」にすると、データベースの実行を最適化することが可能です。

デフォルト数値型変数タイプ

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

デフォルト文字変数タイプ





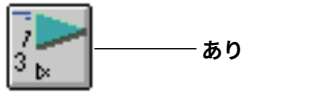
コンパイラは、タイプが設定されていない文字変数を、カバーする範囲が最も広いタイプに設定しようとします。このオプションは、文字列のタイプを強制的に「テキスト」または「固定長文字列」にします。ただし、データベース内のコンパイラ命令に優先するものではありません。

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

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

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

バージョン番号自動生成



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

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

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

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

Windows

Macintosh

オプションについて

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

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

Windows のヘルプ

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

コンパイルの開始

▼ コンパイルのオプションの選択後、コンパイルを開始するには：

■ メインの「オプション」ウインドウの「OK」ボタンをクリックします。

これ以降は、コンパイラにより自動的に作業が進められます。

注：データベースがパスワードによって保護されている場合、「OK」ボタンをクリックするとパスワードの入力が要求されます。コンパイルするには、「デザイナ」または「管理者」を選択し、正しいパスワードを入力して「OK」ボタンをクリックすると、コンパイルが開始されます。

プラグインが見つからない場合、4D Compilerはコンパイルを中断し、場所の入力を要求します。詳細は、この章の「コンパイル」の節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、4D Compilerは現在のプロジェクトを閉じます。

コンパイル

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

データベースのコンパイル中には、「4D Compiler」ウインドウが表示されます。ウインドウの上部には、コンパイル中のメソッド名がスクロールされていきます。

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

「一時停止」ボタンと「中止」ボタンは、一時的に、あるいは完全に中止するためのものです。詳細については「コンパイル処理の中断」の節を参照してください。

コンパイル処理の主な段階は次の通りです。

■ データベースのコピー

■ 変数のタイプ設定

■ コンパイル

■ 4D Engineのコピー

この流れは、「オプション」ウインドウで選択されたオプションにもとづいています。コンパイルの状況は、ウインドウ上部の小さなアイコンで表わされます。

データベースのコピー

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

変数のタイプ設定

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

■ **コンパイラ命令によるタイプ設定パス**：コンパイラ命令や配列定義ステートメントを見つけ、変数や配列にデータタイプを割り当てます。

■ **プロセス変数とインタープロセス変数のタイプ設定パス**：コンパイラ命令でタイプ設定されなかったプロセス変数とインタープロセス変数にタイプを割り当てます。

■ **ローカル変数のタイプ設定パス**：ローカル変数にタイプを割り当てます。

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

コンパイラ命令によるタイプ設定パス

このパスは のアイコンが表示されます。

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

プロセスおよびインタープロセス変数のタイプ設定パス

このパスは  のアイコンを表示します。

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

ローカル変数のタイプ設定パス

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

一般的な考慮

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

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

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

タイプ設定処理のまとめ

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

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

このパスは  のアイコンが表示されます。

このエラーには次の2つの原因があります。

■ 同じ名前を持つ2つのオブジェクトがある場合：

例えば、メソッドと変数、変数とプラグインコマンド、2つのプロジェクトメソッド等です。同じ名前を持つ2つの変数のタイプが異なる場合、このデータタイプの矛盾は記録されますが、コンパイルは中止されません。

■ タイプを判別できないプロセス変数やインタープロセス変数が見つかった場合。

コンパイラが出力するメッセージをもとにコードを修正するかどうかは自由です。

コンパイル

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

このパスは、  のアイコンが表示されます。

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

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

コンパイル処理の中断

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

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

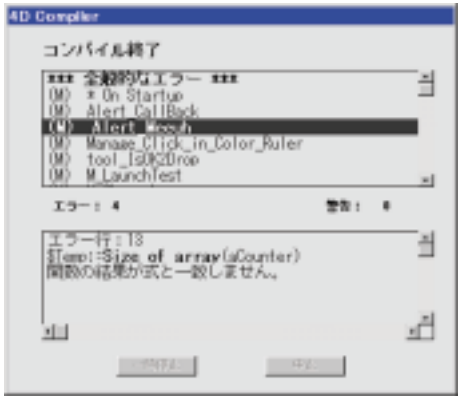
メソッドの参照には次の2つの方法があります。

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

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

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

クリックすることにより、メソッドを選択します。



エラーおよび警告は、ウインドウ下部に表示されます。

下記の内容を3行で表わします。

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

4D Engine のコピー

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

コンパイル後のデータベースの使用

コンパイル後のデータベースは、コンパイル前のデータベースと同様に開くことができます。ただし、デザインモードに入ることはできません。また、データベースを開くには少なくとも1つ以上のカスタムメニューを作成しておかなければなりません。

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

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

ドラッグ&ドロップ

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

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

4D Compilerを4th Dimensionと共に使用する

PowerPC、および80x86用のコンパイル

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

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

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

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

■ シンボルテーブル：

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

■ エラーファイル：

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

■ 範囲チェック：

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

■ タイプファイル：

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

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

この章では、これらのファイルとその使用方法について説明します。

シンボルテーブル

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

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

プロセスとインタープロセス変数のリスト

これらのリストは4つのカラムに分割されます。

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

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

3番目のカラムは、オブジェクトが配列の場合、何次元であるかを表します。

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

変数のシンボル

次のシンボルは変数の名前がどこで発見されたかを表示します。

- 変数がデータベースメソッド内で発見された場合、(M)*の後にデータベースメソッド名が表示されます。
- 変数がプロジェクトメソッド内で発見された場合、(M)の後にメソッド名が表示されます。
- 変数がトリガ（テーブルメソッド）内で発見された場合、(TM)の後にテーブル名が表示されます。
- 変数がフォームメソッド内で発見された場合、(FM)の後にフォーム名が表示されます。
- 変数がオブジェクトメソッド内で発見された場合、フォーム名、テーブル名、(OM)の後にオブジェクトメソッド名が表示されます。

■ 変数がフォーム内のオブジェクトであり、メソッド、フォーム、オブジェクトメソッドのいずれでも使用されていない場合は、(F)の後にフォーム名が表示されます。

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

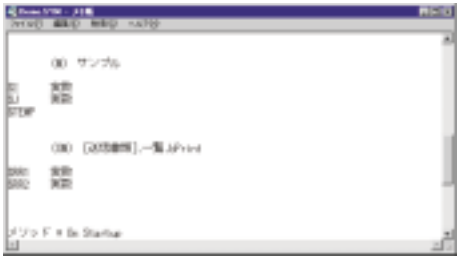
また、プロセス変数に必要な容量は、プロセスのスタックサイズには関連していません。

プロセス変数のリストの例を示します。

ローカル変数のリスト

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

ローカル変数のリストの例を示します。

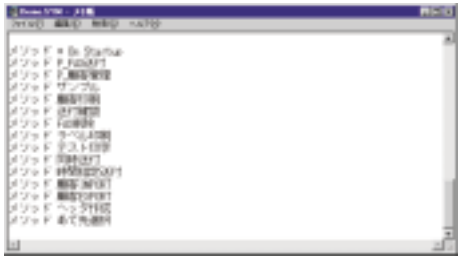


メソッドのリスト

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

メソッド名（パラメータのデータタイプ）：戻り値のデータタイプ

メソッドのリストの例を示します。



エラーファイル

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

■ テキストファイルとして、テキストエディタやワープロで開くことができます。

■ 4th Dimensionで対話型デバッグに使用できます。

メッセージのタイプ

4D Compilerは次の3つのタイプのメッセージを出力します。

■ 全般的なエラー

■ 特定の行に関連するエラー

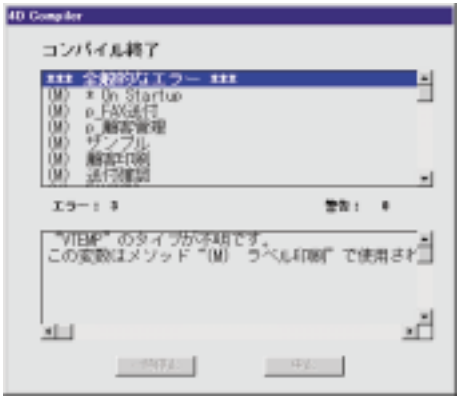
■ 警告

全般的なエラー

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

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

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



コンパイラから全般的なエラーが出力されるのは、次の2つの場合があります。

■ プロセス変数のデータタイプが決定できない場合

■ 複数のオブジェクトが同じ名前を持つ場合

これは、どのオブジェクトに名前を関連付けさせるかを判断できないためです。

一般的なエラーのリストは、付録Aを参照してください。

特定の行に関連するエラー

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

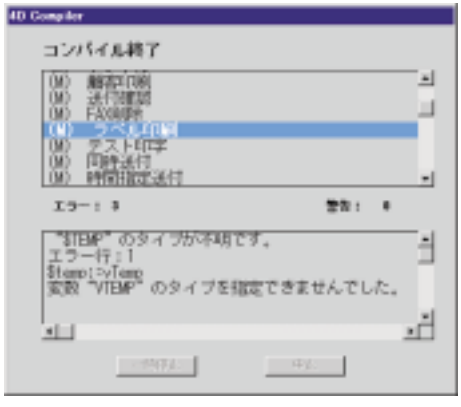
このエラーを含んだメソッド名は、ウインドウ上部に太字で表示されます。メソッドの1つを選択すると、ウインドウ下部に次のような情報が表示されます。

■ 行番号

■ エラーが発生したステートメント

■ エラーの詳細

エラーの例を次に示します。



このエラーリストについては、付録Aを参照してください。

警告

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

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

警告は次の順序で表示されます。

■ 行番号

■ 警告に関連するステートメント

■ 警告の内容

「警告」オプションで「詳細」を選択すると、他の警告も表示されます。

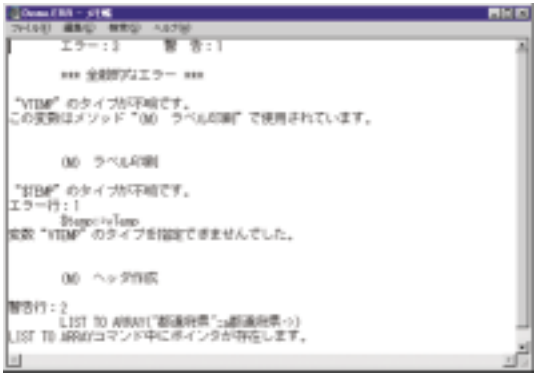
警告リストについては付録Aを参照してください。

エラーファイルの使用

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

エラーファイルをテキストとして使用

テキストエディタを使用してエラーファイルを開いた場合の例を示します。



エラーファイルの構成は次のようになります。

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

エラーや警告は、次の形式のフォーマットで表示されます。

- メソッド中の行番号
- エラーや警告が発生した行
- エラーの説明

エラーファイルを 4D で対話式デバッグとして使用

▼ エラーファイルを使用して、対話式でエラーを修正するには：

1. コンパイルを実行してエラーファイルを作成する。

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

注：Windowsでは、データベースと同じフォルダ内に“データベース名.ERC”ファイルも必要です。

2. **4th Dimensionを起動し、コンパイル前のデータベースを開き、デザインモードにする。**

「モード」メニューに、対話型デバッグ用の2つのメニュー、「次のコンパイルエラー」および「エラーファイル参照中止」が追加されます。

3. **「モード」メニューから「次のコンパイルエラー」を選択する。**

4D Compilerがエラーや警告を検出した最初のメソッドが自動的に開きます。

ウインドウの上部にはエラーや警告の内容が表示され、関連する行がハイライトされます。

4. **指摘されたエラーを修正する。**

警告の場合は、コードを修正する必要がない場合もあります。

5. **「モード」メニューから「次のコンパイルエラー」を選択して、次のエラーや警告へ移動する。**

次のエラーあるいは警告が表示されます。

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

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

タイプファイル

このファイルは4つの部分に分かれています。

■ インタープロセス変数用のコンパイラ命令

■ プロセス変数用のコンパイラ命令

■ ローカル変数用のコンパイラ命令

■ パラメータ用のコンパイラ命令

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

■ 名前が“ COMPILER ”で始まるタイプ設定メソッド内のプロセス変数、インタープロセス変数、パラメータ。

■ ローカル変数が使用されているメソッド（詳細は第5章を参照してください）。

範囲チェック

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

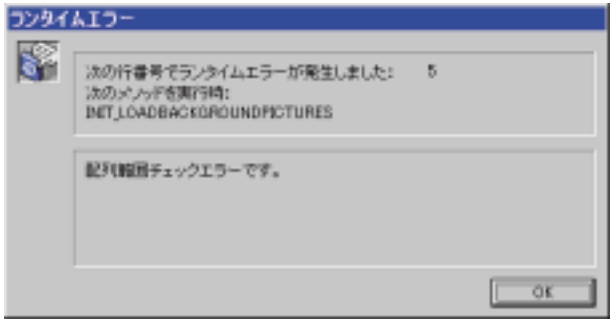
範囲チェックの動作例を示します。テキスト配列“ MyArray ”を定義し、“ MyArray ”の要素数はメソッドにより変化するものとします。5番目の要素に“ Hello ”という文字を代入する場合、次のように記述します。

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

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

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

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



範囲チェックは、配列、ポインタ、文字列の処理を行う際に非常に有用です。

範囲チェックのメッセージのリストについては付録Aを参照してください。

範囲チェックの使用

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

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

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

異常診断

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

考えられる異常としては次のようなものがあります。

■ 4th Dimensionからエラーメッセージが表示される場合：

可能であれば、4th Dimensionの指示に従ってエラーを修正します。指示が一般的で修正の手がかりにならない場合、範囲チェックを指定して再コンパイルします。データベースを再度テストすると、エラーが表示された箇所で、より詳しい情報が表示されます。

■ コンパイル後のデータベースとコンパイル前のデータベースの動作が一致しない場合：

警告メッセージを参照してください。範囲チェックを指定して再コンパイルする必要があり得ます。

■ インタプリタでは正常に動作するが、コンパイル後はクラッシュまたは異常終了する場合：

デバッガに精通していれば、どのメソッドでクラッシュしたのかを調べることができます。デバッガにあまり詳しくない場合の簡単な解決方法は、範囲チェックを指定して再コンパイルすることです。

■ インタプリタでは動作するが、コンパイル後のデータベースでシステムがクラッシュする場合：

コンパイル後のデータベースで使用しているプラグインが、コンパイル前に使用していたものと同じかどうか調べてください。

■ 数値変数や文字列変数が期待するような値を返さない場合：

コンパイル時のプロジェクトファイルの「ローカル変数初期化」オプションの設定を確認する、あるいはシンボルテーブルを作成してすべての変数が正しくタイプ設定されているかどうかを確認してください。

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

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

■ 変数やプロジェクトメソッド、プラグインコマンドにはユニークな（重複しない）名前を付けること

■ 変数には、1つのデータタイプしか割り当てること

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

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

■ 異なる2つの変数に同じ名前を使います。

■ 1つの変数に異なるデータタイプを割り当てます。

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

変数及び配列のデータタイプ

4th Dimensionの変数には、次の3種類があります。

■ ローカル変数

■ プロセス変数

■ インタープロセス変数

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

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

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

変数のタイプ

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

変数には12のタイプがあります。

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

配列には、9つのタイプがあります。

ブール	**整数**	**ポインタ**
日付	**倍長整数**	**実数**
文字列	**ピクチャ**	**テキスト**

シンボルテーブル

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

しかし、コンパイラでは状況が異なります。インタプリタが1行ずつ処理するのに対して、コンパイルする際には、データベース全体を一度に処理しなければなりません。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、データベース、プロジェクト、フォーム、トリガ、オブジェクトメソッドです。
- オブジェクトを調べ、データベースで使われている変数のデータタイプを決定して、変数とプロシージャのテーブル（シンボルテーブル）を生成します。
- 変数すべてのデータタイプが確定すると、データベースの翻訳（コンパイル）を行います。ただし、各変数のデータタイプがすべて確定できなければ、データベースをコンパイルすることはできません。

同じ変数名で異なる2つのデータタイプが見つかった場合、コンパイラはどちらを採用すればよいのかを判断できません。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、すなわち、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かつ明確に指定する方法は、メソッド内でコンパイラ命令を使用することです。しかし、必ずしも変数すべてにコンパイラ命令を使用しなければならないわけではありません。変数ごとにタイプを定義するという従来のコンパイラで必要だった煩わしい作業は、4D Compilerでは不要です。変数の使われ方を調べ、曖昧でない場合は可能な限りタイプを決定します。

▼ 例えば、

```
V1:=12.5
```

と記述した場合、変数“V1”のデータタイプは実数になります。

▼ 同様に、

```
V2:="これは、例題"
```

と記述すると、“ V2 ” はテキストタイプの変数になります。

また、上の例ほど明確でない場合でも変数のタイプを決定することができます。

▼ 以下に例を示します。

```
V1:=12.5
V2:="これは、例題"
V3:=V1
```

ここでは、“ V3 ” は “ V1 ” と同じタイプになります。

▼ 似たような例で、

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

この場合も、“ V4 ” は “ V2 ” と同じタイプになります。

同様に、4th Dimensionのコマンドや、メソッドに対するコールからも変数のデータタイプを決定します。例えば、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。

▼ 例えば、次のように書くと、

```
Number:=4
```

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

変数のタイプを整数や倍長整数、文字にする場合は、コンパイラ命令で定義します。これらのデータタイプはメモリ占有量が少なく、他のタイプに比べて処理速度が速いので、コンパイラ命令を使用することをおすすめします。

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』マニュアルを参照してください。

C_STRING ({method}; length; variable1 {;...;variableN})
C_BOOLEAN ({method}; variable1 {;...;variableN})
C_DATE ({method}; variable1 {;...;variableN})
C_GRAPH ({method}; variable1 {;...;variableN})
C_INTEGER ({method}; variable1 {;...;variableN})
C_TIME ({method}; variable1 {;...;variableN})
C_PICTURE ({method}; variable1 {;...;variableN})
C_LONGINT ({method}; variable1 {;...;variableN})
C_REAL ({method}; variable1 {;...;variableN})
C_POINTER ({method}; variable1 {;...;variableN})
C_TEXT ({method}; variable1 {;...;variableN})
C_BLOB ({method};variable1 {;...;variableN})

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

C_BOOLEAN (Var)

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

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

コンパイラ命令は、代入や用途から得られた結果より優先されます。

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

コンパイラディレクティブが必要な場合

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場合。

■ コンパイラが決定するタイプを使用したくない場合。

■ コンパイラ命令を使用するとコンパイル時間が短縮できる場合。

曖昧になる場合

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

■ 複数データタイプ

■ コンパイラが決定したタイプが曖昧な場合

■ タイプを判断する決め手がない

複数データタイプ

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

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

▼ 簡単な例を示します。

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

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

コンパイラが決定したタイプが曖昧な場合

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

▼ アクティブオブジェクト用のデフォルト値を使用した例：

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

デフォルト値は、オブジェクトの名前である配列内に自動的にロードされます。

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

```
If (Before)
   MyPopup:=2
End if
```

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

```
If (Before)
   ARRAY TEXT(MyPopup;2)
   MyPopup:=2
End if
```

タイプを判断する決め手がない

定義せずに変数が使用されていて、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、

```
Var1:=5.2                `(1)
Pointer:=->Var1          `(2)
Var2:=Pointer->          `(3)
```

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

```
C_REAL (Var2)
```

■ 複合シンタックスコマンド：

複数のシンタックスを持つコマンドに関して、コンパイラは、どのシンタックスと引数が使用されているのかを判定できません。

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

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

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

■ オプション引数付のコマンド：

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

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

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

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

■ URL経由で呼び出されたメソッド：

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

C_TEXT($1)

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

コードの最適化

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

カウンタにローカル変数を使用した場合、変数のタイプを設定しないと、4D Compilerはその変数を実数と見なします。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バイト使用するのに対し、倍長整数にすると、4バイトしか使用しないからです。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用し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用されますが、コンパイルした結果には含まれません。

コンパイルの時間の短縮

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

実数と文字列を使用する

整数と定義した変数に実数値を代入する、あるいは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見なしません。

▼ 例を示します。下記のように記述した場合、

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

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

▼ 次は、文字列処理の例です。以下のように記述した場合、

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

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

インタプリタでコンパイラ命令を使用する

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

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

▼ 例えば、次のように記述した場合、

```
C_LONGINT (MyInt)
```

また、データベースの他の場所で、下記のように記述した場合、

```
MyInt:=3.1416
```

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

4th Dimensionインタプリタは、コンパイラ命令を使用し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（例えば、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表示されても、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立てて処理するわけではなく、実行される順にステートメントを解釈します。もちろんユーザが何を行うかにより、この順序は毎回異なります。つまり、変数を定義する場合、その変数を使用する前にコンパイラ命令を実行するようデータベースを作成することが必要です。

コンパイラ命令をどこに記述するか

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

4D Compilerでタイプ定義される変数

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

注：コンパイル時、4D Compilerは4th Dimensionで作成した順序でメソッドを処理します。エクスプローラで表示される順番ではありません。

■ コンパイラ命令で定義するプロセス変数とインタープロセス変数は、On Startupデータベースメソッドまたは OnStartupデータベースメソッドから呼び出されるメソッドにまとめてください。

■ それらが表示されるメソッド内のローカル変数を宣言します。

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

開発者がタイプ定義する変数

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

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

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

■ インタープロセス変数

■ プロセス変数

■ ローカル変数

■ メソッドへ渡す引数

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

引数定義のためのシンタックスは次の通りです。

Directive (MethodName; parameter)

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

C_BOOLEAN (MyMethod; $1)

注：このシンタックスは、インタプリタでは実行されません。

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

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

特定の引数

■によって受け取られた引数：

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

この引数定義は、COMPILERメソッド内には書き込みができません。

▼例：

On Web Connectionが2つのテキスト引数、“ $1 ”及び“ $2 ”を受け取った場合、データベースメソッドの最初では、**C_TEXT**($1;$2)を記述する必要があります。

■ トリガ

トリガの結果である“ $0 ”引数（倍長整数）は、引数が明確に定義されていなければ、4D Compilerによってタイプ決定されます。定義する場合は、トリガ自身の中で行う必要があります。

この引数定義は、COMPILERメソッド内には書き込みができません。

■“ On Drag OVer ”フォームイベントを受け入れるオブジェクト

“ On Drag OVer ”フォームイベントの結果である“ $0 ”引数（倍長整数）は、引数が明確に定義されていなければ、4D Compilerによってタイプが決定されます。定義する場合は、オブジェクトメソッドの中で行う必要があります。

この引数定義は、COMPILERメソッド内には書き込みができません。

注：4D Compilerは“ $0 ”引数を初期化しません。したがって、“ On Drag Over ”フォームイベントを使用したら直ちに“ $0 ”を初期化しなければなりません。

▼例：

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

C_STRING コンパイラ命令

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

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

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

▼例：

インタプリタでは、

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

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

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

▼例えば、ステートメントをこのように使用します。

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

次のようなコマンドで定義する固定長文字配列についても同じです。

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

配列の文字列の長さを示す引数は、定数にしてください。

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

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

文字列の長さは、4D定数あるいは16進法定数で指定することができます。

▼例：

```
C_STRING (4D_Constant;TheString)
ARRAY STRING(4D_Constant;TheArray;size)
C_STRING (0x000A;TheString)
ARRAY STRING(0x000A;TheArray;size)
```

まとめ

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

■ 起こりやすいデータタイプの矛盾についてのガイドと、タイプ矛盾を避ける方法

■ 4th Dimensionコマンドをコンパイルする際の特別な注意事項

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

■ インタープロセスとプロセス変数

■ ローカル変数

■ 配列

■ フォーム変数

■ ポインタ

■ プラグインコマンド

■ 予約変数

インタープロセス変数とプロセス変数

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

■ 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。

例えば、以下のように書き、

```
Variable:=5
```

また、同じデータベースに、次のように書いたとします。

```
Variable:=True
```

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

用途とコンパイラ命令の矛盾

例えば、以下のように書き、

```
Variable:=5
```

また、同じデータベースに、次のように書いたとします。

```
C_BOOLEAN(Variable)
```

コンパイラ命令が最初に処理されるので、“ Variable ” はブールタイプに設定されますが、“ Variable:=5 ” というステートメントがあるので、タイプの矛盾と見なされます。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見なします。

▼ 簡単な例があります。

```
Bool:=True                  `Boolのデータタイプはブールです。
C_INTEGER (<>Integer)
<>Integer:=3                `代入値はコンパイラの指定と同じタイプ
Var:=<>Integer+Bool         `データタイプが一致しない変数を使用した演算
```

暗黙のタイプ変更による矛盾

関数の中には、戻り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。このような関数の戻り値を異なったタイプの変数に代入すると、データタイプの矛盾が起こります。

▼ 例えば、インタプリタでは以下のように書いてもエラーになりません。

```
IdentNo:=Request ("Identification Number")
   `Ident Noのデータタイプはテキストです。
If (OK=1)
   IdentNo:=Num (IdentNo)
   `Ident Noのデータタイプは実数です。
   QUERY ([Contacts]ID=IdentNo)
End if
```

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

```
IdentNo:=Num (Request ("Identification Number"))
   `Ident No のデータタイプは実数です。
If (OK=1)
   QUERY ([Contacts]ID=IdentNo)
End if
```

2 つのコンパイラ命令による矛盾

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

```
C_BOOLEAN (Variable)
```

そして

```
C_TEXT (Variable)
```

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

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

```
C_STRING (5;MyString)
MyString:="Flour"
C_STRING (7;MyString)
MyString:="Flowers"
```

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

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

```
C_STRING (7;MyString)
MyString:="Flour"
MyString:="Flowers"
```

ローカル変数

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

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

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

と記述し、次に、

```
$Temp:=5
```

と記述するとエラーになりますが、メソッドM1には、次のように記述できます。

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

そして、メソッドM2には、

```
$Temp:=5
```

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

配列内の対立

配列の要素数は、タイプの矛盾とは無関係です。コンパイル前のデータベースと同じく、配列は動的に管理されます。配列の要素数はどのメソッドでも変更でき、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にすること、要素の追加や消去、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こと

■ 配列の次元数を変えないこと

■ 文字配列では、文字の長さを変えないこと

配列要素のデータタイプの変更

配列を**ARRAY INTEGER**等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。

次のように書いた場合、

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

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

配列の次元数の変更

コンパイル前のデータベースでは、配列の次元数を変更できます。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。

したがって、同じデータベースでは、次のように定義することはできません。

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

しかし、同じアプリケーション内で以下のように記述することはできます。

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

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

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

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

文字配列

文字配列には、固定長文字列と同じ規則が適用されます。

次のように書いた場合、

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

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

暗黙のタイプ変更

COPY ARRAY、**LIST TO ARRAY**、**ARRAY TO LIST**、**SELECTION TO ARRAY**、**ARRAY TO SELECTION**、**SUBSELECTION TO ARRAY**、**DISTINCT VALUES**のコマンドを使用する際、意識的に、あるい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。これらは、前で説明した3つの状況のうち、必ずいずれかに該当します。

コンパイラはエラーメッセージを出力するため、修正すべき点が明確になります。暗黙に配列のタイプが変更される例は、第6章の「配列」の節を参照してください。

ローカル配列

データベースでローカル配列を使用する場合は、あらかじめ4th Dimensionでそれらを明示的に定義する必要があります。例えば、メソッドで10要素のローカル整数配列を作る場合、そのメソッドに次の行を追加します。

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

フォーム変数

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

■ コンパイラ命令を使用して、フォーム変数のタイプを定義します。

■ 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ます。

数値タイプに設定されるフォーム変数

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

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

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

ルーラー、ダイアル、及びサーモメータ

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

プラグインオブジェクト変数

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

テキストタイプに設定されるフォーム変数

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

これらの変数は2つのカテゴリーに分類できます。

■ **単純変数**：入力不可の変数、入力可の変数

■ **表示変数**：ドロップダウンリスト、メニュー/ドロップダウンリスト、スクロールエリア、ポップアップメニュー、コンボボックス及びタブコントロール

単純変数

デフォルトのデータタイプはテキストです。メソッドやオブジェクトメソッドで使用される場合は、ユーザが選択したデータタイプが割り当てられます。同じ名前で異なるタイプの変数が存在する場合以外は、データタイプの矛盾が起こることはありません。

表示変数

変数はフォーム内で配列を表示するために使用されます。「オブジェクトプロパティ」ウインドウの「デフォルト値」ダイアログボックス内にデフォルト値を入力すると、配列定義コマンド**ARRAY STRING**または**ARRAY TEXT**を使用して、関連する変数を明確に定義する必要があります。

ポインタ

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

▼ この問題の例を示します。

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

この例では、ポインタで参照する変数のタイプは実数です。これにブールの値を代入しているので、データタイプの矛盾が起こります。1つのメソッド内で、異な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。

▼ ポインタの正しい使い方の例を示します。

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

ポインタには、参照するオブジェクトとの関係だけが定義されています。ポインタによって起こるデータタイプの矛盾をコンパイラが検知できないのはこのためです。矛盾があっても、“タイプチェック処理”フェーズや“コンパイル処理”フェーズでエラーメッセージは出力されません。ただし、ポインタに関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用状況をいくらかは解析できます。“範囲チェック”オプションについては、第3章「診断ツール」の「範囲チェック」の節を参照してください。

プラグインコマンド

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

▼ プラグインコマンドを含むアプリケーションをコンパイルするには：

■ Windowsのプラグインを“ WIN4DX ”フォルダに、Macintoshのプラグインを“ MAC4DX ”フォルダに入れます。これらフォルダは両方とも、ストラクチャファイルと同じ階層に配置します。コンパイラは、これらのファイルを複製しませんが、これらを分析して、これらのルーチンの適切な定義を決定します。

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

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

マルチプラットフォームコンパイル

Windows版4D Compilerでのプラットフォーム独立型コンパイル

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

▼ 4D Client(Macintosh)から4D Server(Windows)にある4Dプラグインを使用するには：

1 はじめにMacintosh版プラグインをWindowsマシンにインストールする。

2 4D Transporterを使用してMacintosh版プラグインをトランスポートする。

注：別プラットフォームへの変換については、『4D Transporter リファレンス』マニュアルを参照してください。

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

3 **Windows上に「Mac4DX」という名前の新しいフォルダを作成する。**

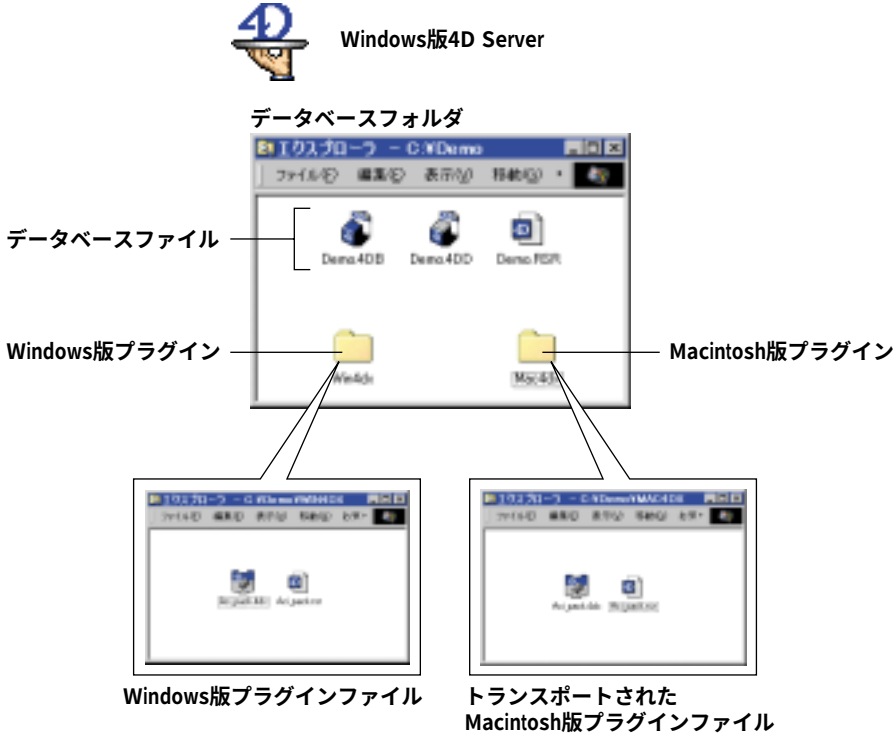
4 **トランスポートされた2つのファイルをMacintosh上からWindows上の「Mac4DX」フォルダにコピーする。**

5 **「Mac4DX」フォルダを「Win4DX」フォルダと共にデータベースのストラクチャファイルと同じ階層に置く。**

6 **4D Serverを実行し、データベースを開く。**

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

下記の図はこのプロセスを示したものです。



注：Proc.Extファイル内の680xxコマンドを使用しているデータベースをコンパイルするには、最初に、4D Transporterで“Proc.Ext”ファイルをトランスポートさせ、“Proc.Esr”ファイルをストラクチャファイルと同一レベルに置く必要があります。

4D Serverを使用したプラットフォーム独立型についての詳細は、各プラグインのマニュアルを参照してください。

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

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

▼4D Client(Windows)から4D Server(Macintosh)にある4Dプラグインを使用するには：

1 **はじめにWindows版プラグインを、Windowsマシンにインストールする。**

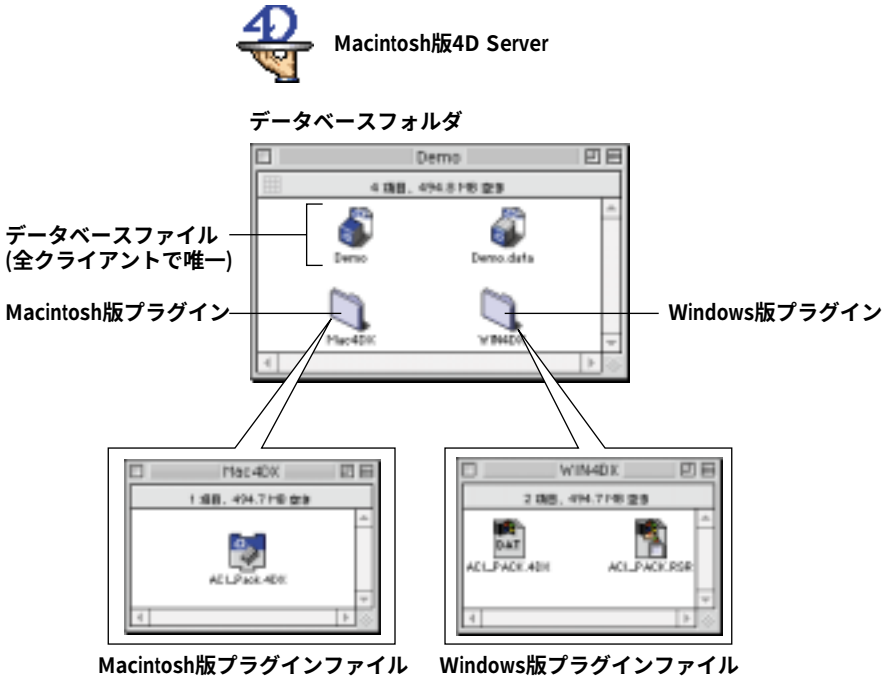
2 **インストールしたWindowsプラグインをMacintoshにコピーする。**
Windows版プラグインは、「Plug-inName.4DX」ファイル、「Plug-inName.RSR」ファイル、「Plug-inName.HLP」の3つから構成されます。

3 **Macintosh上に、新しいフォルダを作成し、それに「Win4DX」と名前を付ける。**

4 **Windows版プラグインのファイルを、このフォルダにコピーする。**

5 **「Win4DX」フォルダを「Mac4DX」とともに、データベースストラクチャファイルと同じ階層に配置する。**

下記の図はこのプロセスを示したものです。



6 4D Serverを実行し、データベースを開く。

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

注：Proc.Extファイル内の680xxコマンドを使用しているデータベースをコンパイルするには、“Proc.Esr”ファイルをストラクチャファイルと同一レベルに置く必要があります。4D Serverを使用したプラットフォーム独立型についての詳細は、各プラグインのマニュアルを参照してください。

マルチプラットフォームコンパイルについての重要事項

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

■ 1つのデータベースがMacintoshおよびWindows上で同じコマンドを使用している場合、それぞれのコマンドの各ID番号と引数の数は、Macintosh版およびWindows版とも同一でなければなりません。

■ コンパイル中は、Macintosh版およびWindows版のプラグインは同一名称でなければなりません。

▼ 両プラットフォームから接続可能な4D Serverを構築するためのコンパイル時のファイル構成例を4D Drawプラグインで示します。

■ Macintosh版4D Compilerでの構成例：

```
DatabaseName
DatabaseName.data
Mac4DX
    4DDraw.4DX
Win4DX
    4DDraw.4DX
    4DDraw.RSR
```

■ Windows版4D Compilerでの構成例：

```
DBName.4DB
DBName.RSR
DBName.4DD
Mac4DX
    4DDraw.4DX
    4DDraw.RSR
Win4DX
    4DDraw.4DX
    4DDraw.RSR
```

Macintosh あるいは Windows での実行形式の作成とコンパイル

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

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

OLE ライブラリ（Windows）でのコンパイル

OLEライブラリを使用しているデータベースをコンパイルする前に次のことを実行してください。

1 「Win4DX」フォルダがないときは作成する。

2 OLETOOLS.DLL ファイル名を “ OLETOOLS.4DX ” に変更する。

3 OLETOOLS.4DX と OLETOOLS.RSR ファイルを「Win4DX」フォルダ内に配置する。

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

■「Win4DX」フォルダに OLETOOLS.4DX と OLETOOLS.RSR ファイルを配置する。

■ OLETOOLS.DLL と OLETOOLS.RSR ファイルを 4D.EXE ファイルとして同じ階層に配置する。

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

暗黙の引数を受け取るプラグインコマンド

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

WR ON EVENT(エリア;イベント;メソッド)

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

メソッドは自動的にこれらの引数を受け取ります。

引数	タイプ	内容
$0	**倍長整数**	
$1	倍長整数	4D Writeエリア
$2	倍長整数	Shiftキー
$3	倍長整数	Altキー
$4	倍長整数	Ctrlキー
$5	倍長整数	イベントのタイプ
$6	倍長整数	値は引数＜イベント＞に依存

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、またはメソッド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。メソッドで使用する場合には、タイプを明確に推測できるように使用する必要があります。

プラグインコマンドで作成される変数

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

Macintosh

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

Windows

VAR#及びVAR<>のリソースが存在する場合は、Macintoshプラットフォーム上と同様に、Windowsのプラットフォーム上でも認識され、使用することができます。これらのリソースが必要な場合は、Macintosh上でResEditのようなリソースエディタを使用して作成する必要があります。

4D コンポーネント

バージョン6.7では、4th Dimensionと4D Insiderを使用して、4Dコンポーネントを作成、管理することができるようになりました。4Dコンポーネントは4Dオブジェクトのライブラリのようなもので、それぞれのオブジェクトには可視性や更新の可否を示す属性（プライベート、プロテクト、パブリック）が割り当てられます。4Dコンポーネントの管理に関する詳細については4D Insiderのドキュメントを参照してください。

原則として、4Dコンポーネントの開発者はコンポーネントがコンパイルでき、コンフリクトを起こさないようにする必要があります。しかし、この可能性を完全に取り去ることはできません。コンポーネントに属するオブジェクトが原因でコンパイルエラーが発生した場合、4D Compilerはオブジェクトの属性に従って、以下のような情報を表示します：

- **プライベート：** 4D Compilerは4Dコンポーネントの名前を表示しますが、関連するオブジェクトの名前は表示しません。
- **プロテクトおよびパブリック：** 4D Compilerは他のデータベースオブジェクトと同様に、オブジェクト名を表示します（標準の動作）。

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

ポインタを使用し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、ポインタを使用して異なるタイプのデータを参照することはできます。

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

この操作を行うには2つの方法があります。

■ ローカル変数のデータタイプを気にせずに使用する方法：

このメソッドはインタプリタでしか動作しません。

■ ポインタを使い、インタプリタでもコンパイル後でも処理できるようにする方法

```
   `インタプリタモードのみでのMemSize機能
$Size:=Size of array ($1->)+1
$Type:=Type ($1->)
Case of
   \($Type=14)                                   `実数配列
      $0:=8+($Size*10)                  `$0は実数（数値）
   \($Type=15)                                   `配列は整数
      $0:=8+($Size*2)                   `$0は実数（数値）
   \($Type=16)                                   `配列は倍長整数
      $0:=8+($Size*4)                   `$0は実数（数値）
   `下記の例では$0はテキスト(Text)です。
   \($Type=18)                                   `配列はテキスト
      $0:=String (8+($Size*4))+"+Sum of text sizes"
   \($Type=19)                                `配列はピクチャ
      $0:=String (8+($Size*4))+"+Sum of picture sizes"
End case
```

このメソッドでは、“ $0 ”のデータタイプが“ $1 ”の配列によって違うため、コンパイルできません。次にポインタを使用する方法を示します。

```
`インタプリタ及びコンパイルモードでのMemSize関数
$Size:=Size of array ($1->)+1
$Type:=Type ($1->)
VarNum:=0
Case of
   \($Type=14)                                   `実数配列
      VarNum:=8+($Size*10)              `VarNumは実数
```

```
    \($Type=15)                                          `配列整数
        VarNum:=8+($Size*2)                        `VarNumは実数
    \($Type=16)                                          `配列長整数
        VarNum:=8+($Size*4)                        `VarNumは実数
    \($Type=18)                                          `配列はテキスト(Text)
        VarText:=String (8+($Size*4))+"+Sum of text sizes"
    \($Type=19)                                          `配列はピクチャ
        VarText:=String (8+($Size*4))+"+Sum of picture sizes"
End case
If (VarNum#0)
    $0:=->VarNum
Else
    $0:=->VarText
End if
```

2つの関数は、以下の点が違っています。

■ 1番目の関数では、結果が変数であること

■ 2番目の関数では、結果が変数へのポインタであること

結果はポインタで簡単に参照できます。

引数の参照

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

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

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

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

この関数は、以下のようにして呼びます。

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

この場合、数値を合計し、指定されたフォーマットに編集して“ 182.70 ”が返されます。

引数は正しい順序で渡してください。最初にフォーマット、次に値です。

以下は、MySum関数です。

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

この関数は、次のように引数の個数を替えて使うことができます。

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

あるいは

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

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

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

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

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

予約変数

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

システム変数

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

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

クイックレポート変数

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

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

定数

4th Dimensionで定義されている定数の全リストは、『4th Dimensionランゲージリファレンス』マニュアルを参照してください。4Dの定数は、デザインモードのエクスプローラでも表示されます。

4D Compilerは、4th Dimensionの通常のコマンドシンタックスにもと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用する4th Dimensionコマンドは、以下の7種類です。

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

COPY ARRAY

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

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

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

そのため、整数と倍長整数と実数、あるいは、テキスト配列と文字配列で文字列の長さが一定でない場合等のように、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。

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

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

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

SELECTION TO ARRAY、ARRAY TO SELECTION、DISTINCT VALUES、SUBSELECTION TO ARRAY

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

以下のように記述すると、

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

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

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に注意してください。整数、倍長整数、実数は似ていますが、同じタイプではありません。

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

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

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

```
SELECTION TO ARRAY (Table; Array)
```

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

LIST TO ARRAY、ARRAY TO LIST

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

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

配列に関連したコマンドでのポインタの使用

ポインタについて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。

Pointer->が配列を表示するところで、下記のように記述した場合、

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

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

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

ローカル配列

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

▼例えば、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用前に定義しておきます。

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

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

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

上記2つのコマンドは変数をディスクに保存または読み込む場合に使用されます。引数は変数です。

コマンドから受け取る変数のタイプは、常に渡した時と同じタイプでなければなりません。

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

変数を受け取る際には、常にタイプが返されることになります。**RECEIVE VARIABLE** コマンドを呼び出したら、Case of 文を使用して、次から受け取るデータを処理できます。

▼ 例

```
   `変数を送る例
SET CHANNEL (12;"Document1")
$Type:=Type ([Client]Total)
SEND VARIABLE ($Type)
For (i;1;Records in selection([Client]))
   $Total:=[Client]Total
   SEND VARIABLE ($Total)
End for
SET CHANNEL (11)

`変数を受け取る例
SET CHANNEL (12;"Document1")
RECEIVE VARIABLE ($Type)
Case of
   \($Type=0)
      RECEIVE VARIABLE ($String)          `プロセス中の変数を受領した。
   \($Type=1)
      RECEIVE VARIABLE ($Real)            `プロセス中の変数を受領した。
   \($Type=2)
      RECEIVE VARIABLE ($Text)            `プロセス中の変数を受領した。
End case
SET CHANNEL(11)
```

注：Type 関数からの戻り値に関しては、『4th Dimension ランゲージリファレンス』を参照してください。

データエントリー

Type (引数)

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

例外

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

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

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

▼ 一例を示します。

```
   `MouseClick メソッド
If (MouseDown=1)
   <>vTest:=True
   MESSAGE ("マウスがクリックされました")
End if

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

次のように**IDLE**コマンドを追加します。

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

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

ABORT

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

ドキュメント

Open Document
Create Document
Append Document

これらの関数が返すドキュメントファイル参照番号のデータタイプは時間タイプです。

演算

Mod (value;divider)

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

Variable:=**Mod** (25;3)

あるいは

Variable:=25%3

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

ブレーク処理

Subtotal (data)

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**コマンドを使用し、集計対象の指定には**ACCUMULATE**コマンドを使用してください。

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。

ストラクチャへのアクセス

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

2つのコマンドは、与えられた引数によって、戻り値のデータタイプが違います。

■ ポインタを与えると、**Table**関数は数値を返します。

■ 数値を与えると、**Table**関数はポインタを返します。

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

変数その他

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

Undefined コマンド

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

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

SAVE VARIABLE コマンドと LOAD VARIABLE コマンド

インタプリタでは、**LOAD VARIABLE**コマンドの実行後に**Undefined**関数を使用して変数が未定義かどうか調べることにより、ドキュメント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用できません。**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します。

■ **LOAD VARIABLE**コマンドを実行した後、ロードした変数のうちの1つを初期値と比較します。

メソッドは以下のようになります。

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

CLEAR VARIABLE コマンド

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

```
CLEAR VARIABLE (変数)
CLEAR VARIABLE (“a”)
```

コンパイル後のデータベースでは、1番目のシンタックス**CLEAR VARIABLE**（変数）は変数を再度初期化します（数値は0に、文字列やテキストは空にする等）。理由は、コンパイル後は未定義の変数が存在しないからです。したがって、コンパイル後はテキスト、ピクチャ、BLOB、配列タイプの変数を除き、**CLEAR VARIABLE**コマンドで変数のメモリを解放することはできません。

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

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

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

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

Get Pointer コマンド

Get Pointer関数は、与えられた引数のポインタを返す関数です。

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

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

次のように書くこともできます。:

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

この処理が終了すると、各要素が変数“Vi”を指すポインタの配列ができます。

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

明示的に変数を定義する方法は、2通りあります。

■ コンパイラ命令を使用してV1...V12を定義するには：

```
C_LONGINT(V1;V2;V3/V12)
```

■ メソッドでV1...V12に値を代入するには：

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

EXECUTEコマンド

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

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

EXECUTEコマンドは、複数のステートメントに置き換えることができます。

▼ 例

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

これは次のように書き換えることができます。

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

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

Traceコマンドと No Traceコマンド

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

各種のコマンドで使用されるポインタ

下記のコマンドは注意して使用してください。

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

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

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

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

```
DEFAULT TABLE([aTable])
   `セットに関するコマンド
V:="Set name"
P:=->V
ADD TO SET(P->)
CREATE EMPTY SET(P->)
CREATE SET(P->)
LOAD SET(P->;"MyDocument")
   `フォームに関するコマンド
V:="Form name"
P:=->V
INPUT FORM(P->)
OUTPUT FORM(P->)
DIALOG(P->)
   `命名セレクションに関するコマンド
V:="Selection name"
P:=->V
COPY NAMED SELECTION(P->)
CUT NAMED SELECTION(P->)
   `書き出しと読み込みに関するコマンド
V:="Document name"
P:=->V
EXPORT DIF(P->)
EXPORT SYLK(P->)
EXPORT TEXT(P->)
IMPORT DIF(P->)
IMPORT SYLK(P->)
IMPORT TEXT(P->)
   `検索と並び替えに関するコマンド
P:=->[aTable]aField
QUERY(P->#"")
QUERY BY FORMULA(P->#"")
QUERY SELECTION(P->#"")
QUERY SELECTION BY FORMULA(P->#"")
ORDER BY(P->;>)
   `印刷に関するコマンド
V:="Document name"
P:=->V
PRINT LABEL(P->)
REPORT(P->)
V:="Form name"
```

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

定数

独自の4DK#リソース（定数）を作成する場合は、宣言された数値のタイプが倍長整数（L）または実数（R）であること、文字列のタイプが文字列（S）であることを確認してください。それ以外のタイプは警告メッセージが表示されます。

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

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

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

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

コードに関するコメント

プログラミングテクニックによっては、コードが他の人にとって理解しづらいものもあります。また、自分で修正する場合でも時が経つと分からなくることもあります。したがって、メソッドには、なるべくコメントを入れるようにしてください。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用すると、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。例えば、変数のタイプを “ Var:=5 ” というステートメントで定義する場合、整数しか使用していなくても、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は同じコンパイルオプションを使用すべきです。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用しない場合には、コンパイラ命令**C_INTEGER**か**C_LONGINT**で変数を定義すると効果的です。

ループのカウンタは常に整数として定義しておくとよいでしょう。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
```

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

注：実行速度は、使用マシンによって異なります。

4th Dimensionの関数の中には整数を返すものがあります（**Ascii**関数等）。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用されないことが明らかな場合は、必ずコンパイラ命令で変数を宣言してください。

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

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

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

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

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

▼ もう1つの例を示します。2つの変数を倍長整数として定義します。

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

そして、計算の結果が3つ目のタイプの定義されていない変数に返されます。

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

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

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

▼ 下記の例では、計算は倍長整数として計算しています。

```
C_REAL($var3)
C_LONGINT($var1;$var2)
$var1:=2147483647
$var2:=1
$var3:=$var1+$var2
```

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

▼ しかし、次の例では、

```
C_REAL($var3)
C_LONGINT($var1)
$var1:=2147483647
$var3:=$var1+1
```

4D Compilerは、“ $var1 ”を倍長整数と見なします。

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

ボタンは実数ですが、倍長整数に定義できる具体的なケースでもあります。

文字

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

```
MyString:="こんにちは"
```

上記の場合、コンパイラは“ MyString ”のタイプをテキストと見なします。

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

注：文字の値を比較したい場合、文字自身についてというよりも、そのASCII値としての比較をしてください。通常の文字比較では、読み分け記号等のすべての文字を考慮します。

その他のヒント

この節は、2次元配列、フィールド、およびポインタについてのヒントを記述します。

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実行効率が上がります。例えば、次のように定義された配列は、

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

以下のような配列よりも実行効率が上がります。

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

フィールド

フィールドを使用して演算する場合、変数にフィールドの値を代入することにより、フィールドよりも変数上で演算する方が実行効率が上がります。

▼次のようなメソッドで考慮します。

```
Case of
   \([Contacts]City="Saratoga")
      Ship:="Blue"
   \([Contacts]City="Reno")
      Ship:="Red"
   \([Contacts]City="Boston")
      Ship:="FedEx"
End case
```

このメソッドは、下記のように記述すると実行速度が速くなります。

```
$Dest:=[Contacts]City
Case of
   \
```

```
($Dest="Saratoga")
        Ship:="Blue"
    \(Dest="Reno")
        Ship:="Red"
    \($Dest="Boston")
        Ship:="FedEx"
End case
```

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

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用する方が速くなります。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。

▼例えば、ポインタ“MyPtr”がフィールドや変数を指しており、その値を使用して一連のテストをする場合は以下の通りです。

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

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

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

ローカル変数

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

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

付録 A ：コンパイラメッセージ

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

■ 警告メッセージ

■ 詳細警告メッセージ

■ エラーメッセージ

■ 範囲チェックメッセージ

■ コンパイラメッセージ

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

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

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

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

警告メッセージ

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

■「COPY ARRAY コマンド中にポインタが存在します。」

```
COPY ARRAY (Pointer->; Array)
```

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

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

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

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

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

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

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

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

■「配列定義コマンド中にポインタが存在します。」

ARRAY REAL (Pointer->;5)

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

■「**Undefined** コマンドは使用しないでください。」

If (**Undefined** (Variable))

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

■「このメソッドは、パスワードによって保護されています。」

■「フォーム1ページ目の自動動作ボタンの名前がありません。」

すべてのボタンは名前を持っていなければなりません。

詳細警告メッセージ

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

■「ポインタの参照先を文字として処理します。」

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

■「文字列のインデックスを数値タイプとして処理します。」

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

■「配列のインデックスを実数タイプとして処理します。」

ALERT (MyArray{Pointer->})

ポインタでテキスト配列または文字配列の要素を参照している場合。

■「プラグインコマンドの呼び出しでパラメータが不足しています。」

SP SELECT CELL(Area)

エラーメッセージ

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

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

■ タイプチェック

■ シンタックス

■ 引数

■ 演算子

■ プラグイン

■ 総合エラー

タイプチェック

■「変数のタイプはブールから実数に変更できません。」

「変数のタイプが異なるため代入できません。」

MyReal:=12.3	`MyReal **は実数。**
MyBoolean:=TRUE	`MyBoolean **はブール。**
MyReal:=MyBoolean	`**ブールは実数に代入できません。**

■「文字列の長さは変更できません。」

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

■「配列の次元数は変更できません。」

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

■「配列定義コマンドに要素数がありません。」

ARRAY INTEGER (MyArray)

■「変数が必要です。」

COPY ARRAY (MyArray;"")

■「定数が必要です。」

C_STRING (Variable;MyString)

■「変数のタイプが不明です。」

「この変数はメソッドで使用されています。」：

変数のタイプを決められません。コンパイラ命令を補ってください。

■「定数のタイプが無効です：文字。」

OK:="**本日は晴天なり**"

■「メソッド“nn”が不明です。」

この行は存在しないメソッドを呼んでいます。

■「誤ったフィールドの使用。」

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

■「文字列の長さは255文字（バイト）までです。」

C_STRING (325;MyString)

■「変数Variableは、メソッドではありません。」

Variable(1)

■「変数Variableは、配列ではありません。」

Variable{5}:=12

■「結果が式と一致しません。」

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

■「タイプが一致しません。」

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

■「引数$のインデックスが数値ではありません。」

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

■「定数のタイプが無効です：文字。」

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

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

```
C_TEXT (Variable)
COPY ARRAY (TextArray; Variable)
```

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

```
Variable:=Num (Variable)
```

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

```
ARRAY TEXT (IntArray;12)
```

“ IntArray ” が他の所で整数配列として定義されている場合。

■「ポインタタイプ以外の変数をポインタとして参照しています。」

```
Variable->:=5
```

“ Variable ” のタイプがポインタではない場合。

■「テキスト変数を数値変数として使用できません。」

```
Variable:=3.5
```

■「フィールドの使い方に誤りがあります」

```
Variable:=[MyTable]MyDate
```

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

シンタックス

■「ポインタタイプ以外の変数をポインタとして参照しています。」

```
Variable:=Num ("本日は晴天なり")->
```

この関数は使用できません。

■「シンタックスエラー」

```
If (Boolean)
End For                          `End If のはず。
```

■「" } " がありません。」

文中の右カッコ **"** } " の数が左カッコ **"** { " の数より少ない場合。

「" { " がありません。」

文中の左カッコ **"** { " の数が右カッコ **"** } " の数より少ない場合。

■「") " がありません。」

文中の右カッコ **"**) " の数が左カッコ **"** (" の数より少ない場合。

■「" (" 形のカッコがありません。」

文中の左カッコ **"** (" の数が右カッコ **"**) " の数より少ない場合。

■「フィールドが必要です。」

```
If (Modified (Variable))
```

Modified 関数にはフィールドを渡してください。

■「" { " が必要です。」

```
C_INTEGER (Array 2)
Array(2):=1
```

■「変数が必要です。」

```
C_INTEGER ([MyTable]MyField)
```

■「数値が必要です。」

```
C_INTEGER (${"3"})
```

■「" ; " が必要です。」

```
COPY ARRAY (Array1 Array2)
```

■「文字参照記号 " ≥ " または "]] " が足りません。」

```
MyString[[3:="a" - Windows 上
MyString≤3:="a" - Macintosh 上
```

■「文字参照記号 “ ≤ ” または “ [[” が足りません。」

MyString≥3:="a" - Macintosh **上**

MyString3]]:="a" - Windows **上**

■「ここではサブテーブルを使えません。」

ARRAY TO SELECTION (array; subtable)

■「If ステートメントの判定結果はブールタイプでなければなりません。」

If (MyReal)

“ MyReal ” が数値変数です。

■「式が複雑すぎます。」

ステートメントを分割して短くしてください。

■「メソッドが複雑すぎます。」

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

■「フィールドが不明です。」

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

■「テーブルが不明です。」

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

■「演算とタイプが一致しません。」

Pointer:= ->Variable+3

■「変数タイプが異なるため代入できません。」

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

「文字列インデックスの使い方に誤りがあります。」

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

“ MyReal ” が数値変数で、“ Variable ” が数値変数以外の場合。

引数

■「結果が式と一致しません。」

```
MyMethod (Num(MyString))
```

MyMethodの引数にブール式が必要な場合。

■「タイプが一致しません。」

「このメソッドに渡す引数が多すぎます。」

```
DEFAULT TABLE (table; form)
```

■「定数タイプが無効です：整数」

```
MyMethod (3+2)
```

MyMethodの引数にブール式が必要な場合。

■「変数のタイプが異なるため代入できません。」

```
C_INTEGER ($0)          `$0は整数
$0:= False              `ここで$0にブールを代入
```

■「引数$のインデックスが数値ではありません。」

```
C_INTEGER (${3})
For ($i;3;5)
   ${$i}:=String($i)          `ここで${$i}はテキスト
End For
```

■「このコマンドに引数は不要です。」

```
SHOW TOOL BAR(MyVar)
```

■「このコマンドには1つ以上の引数が必要です。」

```
DEFAULT TABLE
```

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

```
MyMethod (MyString)
```

MyMethod引数にはブール式が必要な場合。

■「定数タイプが無効です：文字。」

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

メソッド“Calculate”の中でコンパイラ命令**C_INTEGER**($1)が記述されている場合。

■「COPY ARRAY コマンドの引数が、変数です。」

COPY ARRAY (variable; array)

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

Print ("LaserWriter")

メソッド “ Print ” で、＄1が数値の場合。

■「タイプが一致しません。」

$1:= **String** ($1)

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

MyMethod (MyArray)

配列をメソッドに渡すときには、その配列のポインタを渡してください。

■「引数としてそのコマンドに渡せません。」

RECEIVE VARIABLE ($1)

■「引数 “ $1 ” のタイプはブールから整数に変更できません。」

GET FIELD PROPERTIES (tablenumber; fieldnumber; type; $1)

演算子

■「演算とタイプが一致しません。」

Boolean2 := Boolean1+True

■「演算子>は不要です。」

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

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

If (Number = Picture2)

Numberが倍長整数でPincure2がピクチャです。

■「この変数タイプに-(マイナス)符号を付けることはできません。」

Boolean:=-False

プラグインコマンド

■「プラグインコマンド“ nn ”の定義が正確ではありません。」

プラグインコマンドのの定義に誤りがある場合。

■「プラグインコマンドに対する引数が足りません。」

「プラグインコマンドに対する引数が多すぎます。」

総合エラー

■「同じ名称のメソッドが複数あります： nn。」

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

■「内部エラー No.xx」

このメッセージが出力されたら4D社テクニカルサポートに問い合わせ、エラー番号を教えてください。

■「Variableのタイプが不明です。」

「この変数はM1メソッドで使用されています。」

変数のタイプが判断できません。コンパイラ命令を使用してください。

■「ローカル変数の合計サイズが32KBを超えました。サイズ： xxバイト」

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

■「オリジナルのメソッドが壊れています。」

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

■「4Dのコマンドではありません。」

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

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

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

■「関数と変数が同じ名前です： Name。」

関数か変数いずれかの名前を変えてください。

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

関数または変数いずれかの名前を変えてください。

■「メソッドと変数が同じ名前です： Name。」

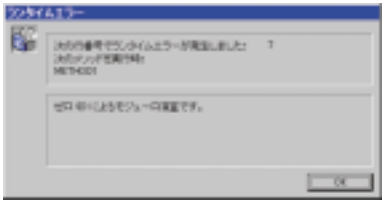
メソッドか変数の名前を変えてください。

■「プラグインコマンドと変数が同じ名前です： Name。」

プラグインコマンドか変数の名前を変えてください。

範囲チェックメッセージ

範囲チェックメッセージは、コンパイル後のデータベースの実行中に、4th Dimensionで以下に示したウインドウに表示されます。



■「ゼロによる割算が発生しました。」

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

■「EXECUTE 内での無効な引数」

変数がデータベース内で明確に表示されない場合。

■「変数へのポインタがコンパイラには未知のものです。」

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

変数がデータベース内で明確に宣言されていない場合。

■「ポインタを使用して再タイプ設定を試行しました。」

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

ポインタが整数タイプのフィールドを指す場合。

■「ポインタの誤った使用。」

```
Character:= StringVar [[Pointer->]] - Windows上
Character:= StringVar ≤Pointer->≥ - Macintosh上
```

ポインタが数値を指さない場合。

■「結果が変数の範囲を越えました。」

```
MyArray{17} := 2.3
```

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

■「ゼロによる割算が発生しました。」

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

■「引数がありません。」

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

■「ポインタが初期化されていません。」

```
MyPointer->:= 5
```

MyPointerがまだ初期化されていない場合。

■「代入先が小さすぎます。」

```
C_STRING (5;MyString1)
C_STRING (10;MyString2)
MyString2:= "TheString"
MyString1:= MyString2
   `MyString2は9文字ですが、MyString1は5文字しか格納できません。
```

■「文字参照エラー」

```
i:=30
MyString≤i≥ := MyString2
```

■「引数に渡された文字列が空、もしくは、初期化されていません。」

```
MyString≤1≥:="" Macintosh上
MyString[[1]]:="" Windows上
```

コンパイラメッセージ

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

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

付録 B ：アプリケーションのカスタマイズ

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

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

Macintosh

カスタムアイコンを作成する前にはResEdit等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、アップルコンピュータ社によるリソース編集用のユーティリティです。以下の説明は、ResEditの使い方を良くご存じの方を対象にしています。

データベースアイコンのカスタマイズ

▼ データベースのアイコンをカスタマイズするには、以下の手順に従ってください。

1 コンパイルしてマージしたストラクチャファイルをResEditで開く。

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クリエータ4文字（バイト）でOwnerNameを置き換える。

クリエータは、ハードディスク上のいかなるMacintoshアプリケーションのクリエータとも異なるものにする必要があります。既存のアプリケーションのクリエータを不用意に使用してしまうと、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、指定したクリエータがユニークかどうかをアップルコンピュータ社に確認し、発売された他のMacintoshアプリケーションで使われていないかどうかを確認してください。

3 BNDL リソース内のアイコンを編集する。

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文字のクリエータを入力する。

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、ResEditを終了する。

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。

他のデータベースファイル用アイコンのカスタマイズ

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

▼ これらのファイルのアイコンをカスタマイズするには：

1 ICN#、icl8、icl4 リソースを開き、ビットマップ編集ツールでアイコンをそれぞれ変更する。

2 アイコンの編集が終了したら、「File」メニューから「Save」を選択する。

3 アプリケーションを終了する。

コンパイラでカスタムアプリケーションを作成した後はデスクトップを作り直してください。この操作で、Finderによってユーザのカスタムアイコンがデスクトップファイルに追加されます。デスクトップの再構築後は、作成したカスタムアイコンが表示されます。

注：デスクトップを再構築するには、Finderの「特別」メニューから「再起動」を選び、「option+command」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。

付録 B ：アプリケーションのカスタマイズ

Windows

アプリケーションのアイコンをカスタマイズする

実行形式のアプリケーションのアイコンをカスタマイズするには、アイコンファイル（データベース名.ico）をストラクチャファイル（データベース名.4DB）と同じ階層に置きます。

1 4D Compilerで、「実行形式アプリケーション作成」オプションを選択する。

標準のオープンファイルダイアログが開きます。

2 4D Engineのファイルを選択する。

3 「OK」ボタンをクリックする。

コンパイル中、4D Compilerは4D Engine.exeとアイコンにマージします。コンパイルが終了すると、“ データベース名.EXE ” という実行ファイルと、コンパイルされたストラクチャファイルが作成されます。

アプリケーションのウインドウ名の変更

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

Macintosh

1 4D Transpoterを使用し、コンパイルされていないストラクチャファイルをWindows用からMacintosh用に変換する。

2 リソースエディタを使用して、データベースのストラクチャファイルを開く。

3 タイプ “ STR ” のリソースを、13010のIDで作成する。

4 アプリケーションのウインドウ名をこのリソースに入力する。

5 データベースをWindows用に変換して、Windows側に戻す。

スタートアップ時のメインウインドウは、入力した名前になります。

Windows

1 4th Dimensionを使用して、ストラクチャファイルを開く。

2 新しいプロジェクトメソッドを作成する。

3 SET STRING RESOURCEコマンドを使用して、ストラクチャファイル内に13010のIDで、“STR”タイプのリソースを作成する。

以下のようなコードのラインを書きます。

```
SET STRING RESOURCE (13010;"My Application")
```

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

付録 C ：リソースのカスタマイズ

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

注：Customizer Plusに関する詳細は、4th Dimensionに含まれている『Customizer Plusリファレンス』マニュアルを参照してください。

Customizer Plus内から4D Compilerの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。

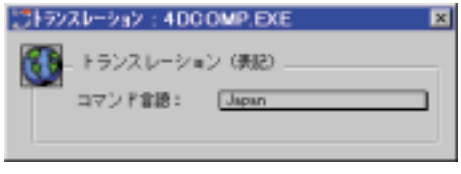
Windows

Macintosh

リソースアイコン「トランスレーション」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のコマンドを、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用して出力されます。

索引

索引