BASIC interpretor "Plain_Basic"
ソースコード解説書

はじめに

筆者は、1980年代の後半に、大型コンピュータの環境で、マイコンBasic並のインタフェースを持つインタプリタを、Fortran言語を使って開発し、NUCE_BASICと名付けて利用していました。これは、新規のプログラミング言語を開発することに目的があったのではなく、専門的色彩の濃い幾何モデリングのサブルーチンライブラリを、使い易くするユーザインタフェースのツールを工夫することにありました。普通、ライブラリを使うとなると、毎回、Fortran 言語を使ってプログラミングをしなければばなりません。この面倒さを救うため、サブルーチンを、マイコンの BASIC を利用するような感覚で、追加のコマンドとして使うことができるような、独立した一つのプログラムを、Fortran 言語で作成することを考えました。その骨格となる BASIC インタプリタ部分がNUCE_BASICでした。

1980 年代、多くの人は、８ビットマイコン BASIC を使ってプログラミングを覚えましたので、NUCE_BASICのユーザインタフェースの仕様もそれに倣いました。そのため、BASIC言語本体部分の説明を簡単に済ませても、ユーザが不便に思うことはありませんでした。BASICの言語体系も、当初は初心者の利用言語(Beginner's All-purpose Symbolic Instruction Code)で出発したにも拘わらず、パソコンで利用するVisual Basicなどは高級言語に変貌してしまいました。したがって、Plain_Basic自体は、当初の精神が生かせるような、プログラミング入門の教材としての使い方を意識して作成しました。

現在のWindowsパソコンの環境は、Fortran言語の利用が不便になりました。それに合わせるため、Fortranで書いた資産の大部分は、パソコンでも汎用性のあるVisual BasicやC/C++言語に書き直す作業をしなければばなりません。Plain_Basicは、Fortran版のNUCE_BASICを2006年にC/C++言語で書き直したバージョンです。開発ツールにはBorland C++Builderを使いました。プログラム書き直す作業は、当初、Fortran のソースコードを読んで C 言語に変換するユーティリティを試しましたが、部分的に手を加える改良には不便でしたので、Fortran のソースコードを参考にしながら作業しました。したがって、C 言語の関数単位は元の Fortran サブルーチン単位を引継ぐスタイルが大部分です。Plain_Basic に特殊なコマンドを追加して使いたい場合は、それらを C/C++で書いて組み込まなければなりません。そのためには、Plain_Basic 本体のソースコードの公開と同時に中身の解説が必要です。これが、この解説書の趣旨です。

島田　静雄　2006年6月版

章　目　次

図版目次

表目次

1.　Plain_Basic の設計概説

1.1　機能の設計思想

少し高級な関数電卓としての使い方を意図したこと

そもそも、コンピュータは数値計算を目的として開発された装置です。数値計算は、専門ごとに多様な目的と要求がありますので、それらに合わせるように自前でプログラミングをすることが常識でした。そのプログラミングツールは､Fortran が主流であって､次善として 16 ビットパソコンで利用する BASIC でした。パソコン利用が大衆化し、その機能も高機能化してきましたので、数値計算にコンピュータを利用するユーザ数が相対的に減り、結果的に数値計算が手軽にできるプログラミングツールも不便になりました。込み入った数値計算のプログラムを使う場合であっても、事前に一寸した予備計算をしたいこともありますので、このときに電卓（電子式卓上計算器）に手が出ます。Windows パソコンのアクセサリには電卓があります。電卓は、数値を直接扱い、また一過性の計算ツールですので、計算手順をプログラミングした再計算ができません。Plain_Basic の設計思想は、この穴埋め的な意義があり、BASIC 流のプログラミングのできる**関数電卓**として使うことを意図しました。

基本的なユーザインタフェースは CUI としたこと

コンピュータプログラムとは、何かの処理を目的とした処理単位の総称です。首尾一貫した流れ作業をしたいとき、幾つかの処理単位を使い分け、データを加工して次の処理単位に渡すような手順を考えます。このときにユーザが関わる方法のことを**ユーザインタフェース**(user interface)と言い、大別して **CUI**(character UI)と **GUI**(graphical UI)とに分けます。DOS の環境は、ユーザの要求をすべてキーボードからの文字並びの入力で行いますので、CUI と **DOS** の環境とは、ほぼ同義です。この文字並びを通称で**コマンド**と言います。コマンドを知らないか、忘れると、コンピュータを前にして何もできなくなります。また、キーボードの使い方に慣れていないと作業が捗りません。これを改良する一つの方法が **Windows** で採用されている GUI であって、アイコンの図柄や、文字で説明してあるメニューなどをマウスで選択してクリックする方式です｡CUI の方式を時代遅れと思う人がいますが､そうではありません。テキストエディタやワードプロセッサは、マウスだけでは全く作業ができません。Plain_Basic は、Windows の環境で機能しますが、BASIC 流の内部プログラム文を編集して使いますので、ユーザインタフェースに保守的な CUI を採用し、マウスを補助的に使います。

バッチファイルの利用ができる設計であること

DOS の環境では、コンピュータがプログラムの実行を開始するファイルは、拡張子(.exe)を持つものの他に、(.bat)があり、バッチファイルと言います。前者の中身は機械語です。バッチファイルはテキストファイルですので、中身を見ることができますし、編集もできます。ただし、これも一種のプログラミングですので、決められたキーワードがありますし、文法もあります。Plain_Basic が実行状態に入ると、擬似的な DOS の環境になって、ユーザからのキーボードの入力を受け付け、それを判断して何かの処理をします。この書式は Plain_Basic の文法として決めてあり、これが同時に Plain_Basic 本体のコーディング仕様です。このキーボード入力文字並びを、そっくりテキストファイルに作成して、それを Plain_Basic に読み込ませると、キーボード入力と全く同じことができます。つまり、このファイルは、Plain_Basic 用のバッチファイルです。バッチファイルを使うことの利点は、ユーザがキーボードやマウス処理のためにコンピュータに貼り付いている必要が無くて、一種の自動実行ができます。これを、教育やデモンストレーションに応用することができます。

ライブラリをコマンドとして組み込むこと

Fortran を利用する科学技術計算では、種々のサブルーチンライブラリをユーザのプログラムに取り込んで使います。ユーザのプログラミングは、使いたいサブルーチンにデータを渡す準備をすること、また、処理された結果をデータとして取り出すこと、が殆どです。このインタフェースでは、サブルーチンをインタラクティブな BASIC 言語の環境で、追加コマンドのように使いたい希望がありました。それを実現させるため、Fortran 言語で BASIC インタプリタを作り、そこに、サブルーチンをコマンドのように組み込むツールを開発しました。追加コマンドのない骨格だけの BASIC インタプリタが NUCE_BASIC です。Plain_Basic は、これを C/C++で書き換えたものです。ただし、初心者向けのプログラミング教育に利用することを考えて、基本的なグラフィックス用サブルーチンを追加した、機能を絞ったバージョンです。

言語モデルはマイコンの BASIC にあること

Plain_Basic の言語仕様は、今は無い「JIS 基本 BASIC(JIS C6207-1982)」に一応準拠していますが、1980 年代の８ビットマイコンのベストセラーであった NEC PC8001 の BASIC 言語仕様を真似たものです。現在のパソコンの環境では、簡単な数値計算は、事務計算に出発点のある EXCEL を利用する人が増えました。これは、変数名などに文字を使う代数式を扱うことができません。少しクセのある VBA(Visual Basic for Applications)を利用することもできますが、やや不便なところがあります。Plain_Basic の設計思想は、BASIC 流のプログラミングのできる関数電卓として使うことです。８ビットのマイコンが広く利用されていた時代、その BASIC 言語が単純でしたので、多くの人は、取り立てて初心者教育を受けなくても、プログラミング技法を簡単に覚えることができました。現在の Visual Basic などは、BASIC を標榜してはいますが、相当に高級化しましたので、初心者が利用するにはハードルが高くなってしまいました。

プログラミング教育のツールに使うこと

そもそも'BASIC'言語の開発の発端は、初心者が簡単にプログラミングを理解してもらう教育目的を意図しましたので、プログラミング言語としての特徴的な基本機能を一通り揃えています。BASIC 言語を実行させる環境は、「コンピュータ本体・キーボード・モニタ・ファイル装置・OS」で構成します。プログラミングは、この全体を<u>システム</u>として捉え、一通りの知識を踏まえる必要があります。1980 年代の８ビットのマイコンは、この全体を必要最小限で構成した、ハード・ソフト一体型のシステムでした。Windows のパソコンの環境で、擬似的に８ビットマイコンと BASIC の言語環境を再現すれば、プログラミング教育に必要な最小限のシステム構成が揃います。これも Plain_Basic の開発方針の一つです。

教育目的には興味を引く使い方が欲しいこと

目的を持たないお稽古事として、コンピュータ、さらにはプログラミングを勉強するのは味気無いものです。教育に使う場合は、学習者が興味を持つような題材を選ぶと教育効果が上がります。その題材として、数値計算法だけでは現実味が少なく、コンピュータグラフィックスは人気があります。８ビットのマイコンが 1980 年代に、アマチュアを中心に爆発的に普及した最大の理由が、コンピュータゲームの開発に使うことでした。そして、それを支えたのがグラフィックスの機能でした。コンピュータグラフィックスは、勉強の題材としてかなり奥が深いものです。プログラミングの入門ツールとしての Plain_Basic は、最小限のグラフィックスコマンドを準備し、簡単な作画の例題を準備しました。グラフィックスを自分で工夫するためには数値計算が必要ですので、それが数値計算法の勉強に役立ちます。このグラフィックス用コマンドは、教育利用を意識して基本機能だけに絞ってあります。

文字処理機能は不充分であること

'BASIC'をキーワードに持つプログラミングツールの特徴は、文字処理用関数が他のプログラミング言語に比べて豊富にある便利さです。これは、８ビットのマイコンを英文ワードプロセッサの開発に利用した歴史的な背景があったためです。ただし、Plain_Basic は数値計算を主な利用対象としましたので、文字処理関数を含めてありません。文字処理は、プログラム文の編集に利用するラインエディタと、結果の出力に使う PRINT コマンドで必要ですが、単純なデフォルトの書式変換を採用しています。Plain_Basic は、テキストエディタの機能を持つ <u>RichTextEdit</u> を載せてありますので、この上で文書の編集ができます。Plain_Basic では、変数定義に文字列型を持っていますが、一つの変数は文字長さ 15 バイトまでを保存できる固定長にしてあります。

ファイルの扱いを単純化したこと

ディスクを使いこなすことは、中身のファイル構造の知識を始め、初心者には多くの勉強が必要です。電卓はファイルを使わないコンピュータです。８ビットのマイコン時代、ディスクは特別な装置でしたので、プログラムの保存と、データの保存とを一つのファイルで扱いました。その方法は、プログラム文とデータ文と、どちらも行番号（ラベル）を付けて、全体を一つの文書ファイルとすることです。データ文であることを示すには、キーワードの DATA を行頭に付けます。このようなデータファイルの考え方を<u>内部ファイル</u>と言います。Plain_Basic のプログラムは、内部ファイルの読み出しに READ 文を使います。データファイルの編集は、BASIC プログラム文の編集に含ませます。ファイルの扱いに関するコマンドは、LOAD/SAVE/MERGE の三つと、バッチファイルを呼ぶ DECK で済ませました。

1.2　Plain_Basic のユーザインタフェース

Plain_Basic の作業画面

Plain_Basic を立ち上げたときの画面を図 1.1 に例示します。これは、Windows の機能を使って擬似的に DOS の環境を構成しています。最上段から、タイトルバー、メニューバー、キーボードの文字入力をモニタする一行分のテキスト入力枠を持ちます。その下にテキストウインドウとグラフィックスウインドウの二つの子ウインドウが並ぶ MDI(Multi Document Interface)構成です。DOS の環境を再現するだけであれば、コンソール形式のテキストウインドウ一つで済みますが、グラフィックスウインドウを加えてあるのが特徴です。これらの画面単位を通称でウインドウと言います。ただし、グラフィックス関係のウインドウの用語と紛らわしいので、Windows プログラミングの専門用語ではフォーム(form)の名称を使います。この用語の使い分けは多少曖昧です。この画面の構成要素個々には定義名があり、総称ではオブジェクト(物の意)と言います。Plain_Basic へのユーザの指示は、原則として、キーボードからテキスト入力枠への文字列入力だけ、つまり CUI で行います。メニューバーの項目はマウスで選択しますが、こちらは「Windows 画面の位置と寸法などを変える・画面データをファイルに保存する・Help を参照する」など、BASIC 言語の使い方とは独立したオプション的な処理に当てます。二つの子ウインドウは、Plain_Basic からの出力専用です。ただし、テキストウインドウは、RichTextEdit が載せてありますので、そのまま、独立したテキストエディタとして使うことができます。

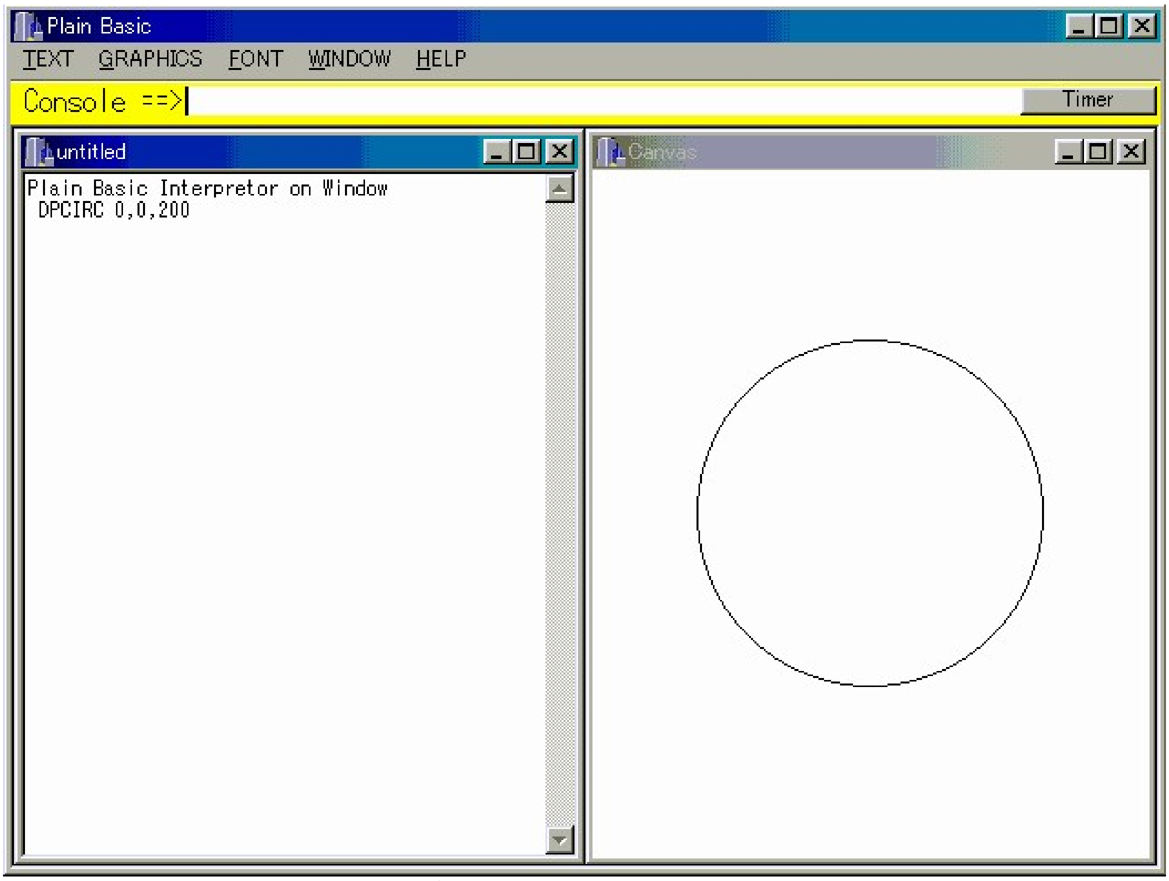

図 1.1　Plain_Basic の立ち上げ時の画面（コマンド DPCIRC で円を一つ描かせました）

テキスト編集の機能も CUI で行うこと

Windows のメニュー項目は、マウスで選択する目的で準備します。しかし、Plain_Basic ではなるべくマウスを使う処理が少なくなるように設計しました。例えば、EDIT メニューの定番は、テキスト編集を助ける「Cut/Copy/Paste」ですが意図的に省きました。これは Ctrl キーを押しながら x/c/v キー打つことで代行できるからです。Plain_Basic の終了は、タイトルバー右端の閉じるボック $\boxed{\text{X}}$ をマウスでクリックしますが、これもキーボードから QUIT コマンドで行えるようにしてあります。子ウインドウの消去は、コマンドの CLS が定番です。ただし、子ウインドウが二つありますので、グラフィックス画面の消去は DPERAS で行います。どちらも、メニューの項目には含ませてありません。

コマンド駆動型の処理の集合

ユーザとの対話(interactive)で処理を進めるプログラムは、ユーザが何かの操作（アクション）をすることをコンピュータが待ち続けます。このアクションを、Windows の環境ではイベントと言い、待機状態をイベント待ちと言います。CUI の環境でのアクションは、キーボードからのコマンドまたはステートメントの文字並びの入力です。文字並びはテキスト入力枠でモニタされ、Enter キーを押すことでその文字並びが取り込まれます。その文字並びを解読し、その意味によって、決められた処理に分岐します。一通りの処理が済めば、再びイベント待ちに戻ります。この処理原理を模式的に表したものが図 1.2 であって、path selector と命名しました。左の矩形枠の部分は文字列の入力と解読部分であり、そこから A,B,C,…と名前を付けた処理が、論理的には並列に準備されています。ユーザの文字列を解読すると、どれか一つの処理が選択され、処理が済めば、再び元に戻ります。このような制御の方式をコマンド駆動型(command driven)と言います。

イベント駆動型の処理はコマンド処理の前処理

Windows の制御方式も図 1.2 と原理的に同じですが、こちらはイベント駆動(event driven)と当て、分岐先はシステム側で準備した処理です。図 1.2 の処理 A,B,C…に当たるルート名は、Windows システム側が定義した Callback 関数名が割り当てられます。ユーザが Windows の画面上で何かの操作をすると、システムはそれをイベントとして受け取り、「何の御用でしょうか」と問い合わせをしてくる窓口が決められています。この方式は電話の Callback 処理と似ていることからそう呼ばれます。プログラミングの用語としては、イベント関数、イベントハンドラー、イベントプロシージャ、などと言います。CallBack を受け取るこれらの指定関数は、プログラミングツール側（ここでは Borland C++Builder）で定義され、Windows パソコンのシステムが呼びます。そこに、ユーザがしたい処理のコードを記入します。イベントの種類は非常に多いので、何も記入がなければイベントそのものが無視されます。ユーザが追加した処理の綜合が、結果としてコマンド駆動を構成します。

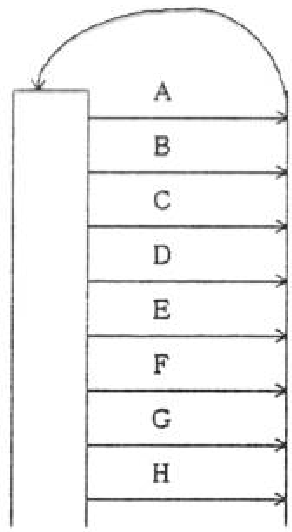

図 1.2　Path Selector （選択グラフ）の原理図

文字列解読処理が基本処理であること

Plain_Basic のコマンド処理の受け付け場所は、ただ１箇所、テキスト入力枠です。そして、イベントを解読するイベント関数（CallBack 関数）は、ここでは OnKeyPress です（実装名は少し違います)。キーボードから入力される文字並びは、バッファーに保存されます。CR キー（Enter キー）を受け取ると、その文字並びを持って一旦システムに制御が帰ります。そこから間接的にタイマ制御で別のイベント関数に処理を渡します。そこでの処理が一段落すると再びイベント待ちに戻ります。少し遠回りの処理を行いますが、これはバッチ処理を行わせるためです。この処理部分の論理を次節で説明します。

1.3　バッチ処理を組み込む設計

最初はCUIで設計すること

Plain_Basicは、Windowsの環境で、擬似的にDOSのユーザインタフェースを構成するようにCUIで設計し、マウスを利用するGUI処理を最小限に抑えました。メニューを選択するGUI処理も、キーボードショートカットを組み込めばキー操作もできますが、少し操作が面倒になりますのでコーディングに組み込みませんでした。コンピュータを利用するユーザは、コンピュータと対話するような操作が必要ですので、コンピュータの前から離れられない不便さがあります。コンピュータに作業の手順を覚えさせておいて、自動的に実行させるようにすれば、離れることが可能です。その前段階として、コンピュータの制御をすべて、キーボードからの文字並びの入力で行うようにします。これがCUIのインタフェースです。この同じ文字並びをテキストファイルに作成しておいて、そのファイルをキーボードに代えて読み出して実行させることをバッチ処理と言います。マウスを使ってアイコンやメニューを選択するGUIの方式では、その操作手順を再現するようなテキストファイルを構築できないのです。

バッチファイルとプログラムファイルとの違い

Plain_Basicの実行は、テキスト１文を単位として意味を解読し、図1.2で示したように個別の処理に分岐して再びシステムに戻ります。テキスト１行は１文が基本ですが、複数の文をコロン（：）で区切って一行に含ませることもできます。コマンド文とは原則として１文で処理が完結するものを言います。ステートメント（BASIC流で書いたプログラム文）は、複数文のまとまりで機能するものを指します。コマンド文は、バッチファイルに構成しておいて順に読み出して実行させることができます。ステートメントの場合には複数の文が関係し、処理が行き来します。そのため、ステートメントは、内部メモリにラベルを付けて保存しておいて、図1.2のループをどの順で実行させるかの補助にラベルを使います。これがPlain_Basicのプログラムです。このプログラムもテキストファイルとして保存しておくことができ、プログラムファイルと言います。こちらはバッチファイルと似ていますが、読み出し順に実行できないことも起こります。プログラムファイルを内部メモリに読み込むのはLOADコマンド、それを実行させるのがRUNコマンドです。これらのキーワードは、バッチファイルのテキストに使うことはできますが、プログラムファイルのテキストに使うことはできません。

バッチ処理の実行にはPAUSE文が必要

バッチ処理を実行させるために準備するテキストファイルがバッチファイルです。DOSのモードでは、ファイルの拡張子を .bat と定義しています。Plain_Basic ではテキストファイルであれば拡張子は何でも構いません。ユーザがキーボードから文字列を直接入力する操作に較べると、バッチファイルからのテキスト入力は圧倒的に高速です。しかし、処理の途中経過を確かめる余裕時間ができませんので、要所要所で実行を止めるPAUSEコマンドを挿むようにして確認時間を取ります。PAUSEコマンドは、秒単位で待ち時間を指定して処理を停止し、待ち時間が終了したら処理を再開させます。秒数が残っていても、待ちを中断して再開させる手段も必要です。これには、ユーザがキーボードから何かの文字入力、例えばEnterキーを打つようにします。PAUSE文は、一時的に処理を止めるように事前に指示しておくコマンドです。バッチ処理を中止したいときは、PAUSE 文で待ちになっているとき、ESC キーを押します。ただし、プログラムが暴走して無限ループを回っていると、システムに働きかけて処理を中断させる方法がありません。普通は(ctrl+alt+del)キーを使う以外止めようがありません。

割り込みの制御と判断にフラグを利用すること

コンピュータシステムは、何も実行していない状態でも、内部ではイベント発生を監視するループを回っています。イベントを検知すると、オブジェクトの種類で決められたイベント関数に分岐します。これを、割り込み(interrupt)とも言います。ただし、割り込みは、何かの連続処理を特権的に中断して別の処理をこなし、それが済むと、普通はもとの流れに戻ります。割り込みはシステムに直接働きかけますので、非常に危険な側面があります。Windows のシステムは、この危険を避けるようにイベント関数に処理が分岐するようになっています。イベント関数に何も指示がなければ、そのイベントは無視されます。ユーザは、イベント関数内に処理したいコードをプログラミングします。そこで幾つかの条件判断をするときに、定義を決めた論理変数を使います。これをフラグ（flag：旗の意）と言います。Plain_Basic では幾つかの動作モードを決めてあります。そのモードを表す論理変数が、フラグとしてイベント処理で利用されます。PAUSE 文による制御を含め、バッチ処理を実現するにはフラグで判断を使い分けますので、フラグの種類を次節の表1.1に示します。

1.4　Plain_Basic の動作の概要

プログラムの入口

どのプログラミング言語においても、コンピュータシステムが最初に制御を渡す入口が決められています。そのモジュール名は、main をキーワードに持つのが標準です。明示的に入口プログラムを指定する場合と、暗黙の約束を持つ場合とがあります。プログラマは、プログラムの始まりが、どこにあって、どのように制御が移動するかを意識してプログラミングをします。そして、プログラムの終了も正しく行わせます。システム側から制御が入る入口は、C 言語では明示的に main を使います。Windows のアプリケーションプログラミングでは、WinMain が最初の入口です。ただし、ベンダーのプログラミングツールの中には、WinMain 自体を自動発行してくれる製品がありますが、それをブラックボックス化している製品もあります。WinMain からの移動先はシステムが管理していて、一般のプログラマにはブラックボックスです。このブラックボックスの中にイベントを監視するループがあり、イベントに対応したイベント関数にシステムから制御が入ります。どのような条件のときに、どのイベント関数に制御が入るかをユーザがプログラミングします。これが、Windows の環境になって必要になったオブジェクト指向プログラミングです。そして、そのイベント関数からの戻り先もシステムですのでブラックボックスです。Plain_Basic では、キーボードからの文字入力を受け付ける OnKeyPress イベント関数が実質的なプログラムの入口です。ここから、どのように処理が流れるかを、この節で解説します。Plain_Basic 本体のソースコードは、作業性の関係で複数の.cpp と.h ファイルに分けました。個別のコンパイルと全体に綜合するリンクは、C++Builder の IDE（統合開発環境：integrated development environment）で行います。

表 1.1　プログラムの実行状態（モード）を表すフラグ

フラグ	標準状態	解　　　説
bErrorFlag	off	エラーが発生したときに on となり、当該処理を中止します。
bAutokey	off	プログラム編集モードのときに on になります。
bRunMode	off	プログラム文が実行状態に入ると on になります。 標準状態は off であって直接モードです。
bDeckMode	off	バッチ処理に入ると on にします。
bFileMode	off	LOAD/SAVE/MERGE でファイルを開くと on にします。
bTraceFlag	off	プログラム文が実行(bRunMode=on)されるとき、 実行されるラベルをコンソールにリストします。
bEchoFlag	on	キーボードから、またはバッチファイルからの入力テキストがコンソールにエコー出力されます。
bTimerFlag	off	PAUSE コマンドが実行されているときに on です。
備考：フラグは論理型の変数を当ててあります。on は true、off は false の値です。		

四つの実行モードがあること

Plain_Basic の動作は、ユーザ側から見るとき、四つの実行モード（状態）があります。<u>直接モード</u>、<u>プログラム編集モード</u>、<u>プログラム実行モード</u>、そして<u>バッチ処理モード</u>です。Plain_Basic がどのような状態にあるかを表すパラメータ(フラグと言います)が決めてあって、それを判定して種々の処理に分岐します。Plain_Basic が立ち上がったとき、フラグは、表 1.1 に示す標準状態になっています。テキスト入力枠のウインドウにフォーカスがあって、文字入力のカーソルが点滅します。マウスを使ってメニューを選択することも、またテキストウインドウを選択してエディタとしての利用もできます。Plain_Basic の動作は、テキスト入力枠からの文字入力で始めますので、テキスト入力枠をアクティブにしておくのが基本です。この状態が直接モードです。Windows の環境で、システムが管理している状態ですので、これを「システムの制御状態にある」と言うことにします。Plain_Basic の動作の概要は、以下に箇条書きします。

- テキスト入力枠の文字データを順に一文字づつ判定するのは OnKeyPress イベント関数です（実装する関数名は少し違います)。また、どのように処理を組み立てているかは、この箇条書き項目の後で、リストを添えて解説します。このイベント関数は一文字読むごとにシステムに制御が帰りますので、文字入力の途中でメニュー処理などの道草をすることもできます。一行分の文字並びを入力し、Enter キー（改行キー）を押すと、始めて全体の文字並びが textBuff[]に取り込まれます。テキスト入力枠に表示されている文字並びは消去され、エコーフラグが on であれば、テキストウインドウにエコーを表示します。この textBuff[]の文字列をデータとして、図 1.2 に概念的に示したコマンド解読と実行処理に入ります。ここが実質的な Plain_Basic のメインプログラム部分になりますので、その入口を pbMain()として一つ独立関数に編集しました。
- 上記のメインプログラム pbMain()は、OnKeyPress イベント関数の中から呼ぶのが単純ですが、バッチ処理に対応させるため、少し遠回りですが、OnTimer イベント関数の中から間接的に呼びます。それを実現するため、OnKeyPress イベント関数から制御がシステムに帰るときも、タイマを起動させます。このタイマは、決められた時間間隔で OnTimer イベント関数を呼びます。タイマ割り込みの時間間隔を充分小さく(1ms)に設定しておけば、キー入力に戻ることはありません。OnTimer イベント関数に制御が入れば、タイマを停止させます。OnTimer イベント関数から出て制御がシステムに帰るときは、タイマは停止していますので、改めてキーボード入力を受け付けます。
- pbMain()は、文字列データを行単位で切り出しながら連続して解読するように、無限 for ループを持っています。文字列データの解釈と実行は、for ループの中で pbMain2()を呼んで行わせます。ここからは、C/C++言語でコーディングした非常に多くの関数が繋がります。Plain_Basic は、図 1.2 のように並列に多くの処理を組み込むことができるからです。そこでは、コマンドまたはステートメントの実行が bRunMode の on/off 条件と合わなければエラーフラグ（bErrorFlag）を立てて帰ってきますので、エラーの場合には for ループを出てシステムに制御が帰ります。
- Plain_Basic のフラグは、最初は標準状態ですが、文字列を解読する段階でモードが変ります。RUN コマンドであると、プログラム実行モード（bRunMode=on）に変り、プログラム文の保存してある内部のテキスト領域のアドレスにポインタに付替えて for ループの最初に戻り、以降 for ループを繰り返して回ります。プログラム文の終端にくるか、END 文または STOP 文にくると、（bRunMode=off）に戻し、for ループから抜け、システムに制御が帰ります。
- PAUSE 文をプログラム文の中で使っているときは、bRunMode=on のモードであっても、for ループから抜けて、仮にシステムに制御が帰ります。これは、待ち秒数が残っていても、キーボードの任意のキー入力を受け付けて、<u>待ち</u>を取り消す処理をさせるためです。待ち秒数が無くなった場合も含め、再度タイマを起動して OnTimer イベント関数に制御を戻します。
- LOAD コマンド、MERGE コマンドは、テキスト形式のプログラムファイルから一行単位でプログラム文を読み出して、それを内部のプログラム文の領域に格納します。この場合、(bFileMode=on) のフラグを立て、テキストファイルの終りまで読む間、for ループを回ります。
- 直接モードのときは、FOR-NEXT, GOSUB-RETURN、IF-THEN-ELSE などを除き、一行のテキスト文で処理が完結するコマンドとステートメントは、pbMain()の for ループを一度通っただけで制御がシステムに帰り、次のテキストを読むため、キーボード入力を受け付けます。
- DECK コマンドは、キーボードに代えて、バッチファイルからテキストを入力するように切り換えます。バッチファイルとプログラムファイルとの違いは、後者のテキストには行頭すべてに整数ラベルが付きます。DECK コマンドは LOAD/MERGE コマンドと殆ど同じ動作ですが、ファイルの読み込み前後で Plain_Basic の環境設定に違いがあります。DECK コマンドでは、読み込んだテキストのエコーリストがすぐ得られますので、LOAD/MERGE に続けて LIST コマンドで確認リストを取る処理を省くことができます。
- DECK コマンドは、LOAD/MERGE の処理と同じように、pbMain()の中の for ループ内でテキストの解読と処理とが行われ、ファイルの終りまで読めば制御がシステムに帰ります。処理の切れ目でバッチ処理を打ち切りたいときは、あらかじめ PAUSE コマンドを挿入しておきます。この待ち状態のときはキーボードの入力を受け付けるようになっています。任意のキーは待ちを取り消し、Esc キーはバッチ処理を打ち切ります。

OnKeyPress イベント関数の解説

- Plain_Basic の作業画面では、テキスト入力枠を常に表示させておくため、テキスト入力枠に使う一行分の Edit コントロールは、親ウインドウのツールバーに載せてあります。
- Edit コントロールにフォーカスがあれば、キーボードからの文字入力がそこに表示され、打った文字の種類を持って OnKeyPress イベント関数に制御が入ります。普通は何もしないで制御をシステムに帰し、次の文字入力を待ちます。Enter キー(改行)キーが打たれると、それまでの入力文字を TextBuff[]に取り込み、Edit コントロールの文字を消去します。エコーフラグが(bEchoFlag=on)であれば、入力文字をテキストウインドウに表示し、タイマを起動して制御をシステムに帰します。これが基本的な処理手順ですが、フラグの状態で幾つかの処理が加わります。
- 制御がシステムに帰ると、システムは、直ぐにタイマイベントを検知しますので、制御は OnTimer イベント関数に入ります。システムタイマの Interval プロパティを 1ms に設定してありますので、実際上、再度 OnKeyPress イベント関数には入りません。
- ユーザにテキスト入力を要請するプロンプトは、Edit コントロールの左側に、「Console ==>」と表示します。プログラム編集モード(bAutokey=on)のときは、ラベル番号を自動的に増やしながら発行しますので、例えば 「Auto 10=>」のようにプロンプトが変り、テキスト部分だけキー入力します。ただし、テキストウインドウには番号付きでエコー表示します。
- PAUSE コマンドが実行されると、(bTimerFlag=on)になり、制御はシステムに帰っています。タイマは１秒間隔でタイマイベントを発行するようにしますので、1 秒ごとに OnTimer イベント関数に入ります。ここでは秒数のカウントダウンをして、すぐに制御をシステムに返します。この間にキー入力が可能ですので、何かのキー入力を受け付ければタイマを停止させ、PAUSE 状態は解除されます。

表 1.2　OnKeyPress イベント関数のリスト

```
void __fastcall TFormMain::Edit2KeyPress(TObject *Sender, char &Key)
{                                       // 2006-09-10
    String LabelNo, TextInput;
//
    if (bTimerFlag) {                       // cancel Pause waiting
        pbTimerDeactivate();
        if ((bDeckMode) && (Key==VK_ESCAPE)) {
            fclose(fdeck);
            bDeckMode=false;};              // cancel Deck Mode
        FormMain->Timer1->Enabled = true;
        return;};
//
    if(Key!=VK_RETURN) return;              // repeat key input
//
    TextInput=Edit2->Text;
    iTextLength = TextInput.Length();
    if (iTextLength > (BUFFER_SIZE-10)) {
        Write6(TextInput);
        pbErrList(ErrString23);
        TextInput ="";};
    StrPCopy(textBuff, TextInput);
    if(bEchoFlag){
        LabelNo="";
        if (bAutokey) LabelNo =AnsiString((Word)iCurrentLabel);
        Write6(LabelNo + " " + TextInput);};
    Edit2->Text="";
    if (bAutokey) {
        iCurrentLabel += iIncreNo;
        LabelDisplay();}
    FormMain->Timer1->Enabled = true;
}
```

OnTimer イベント関数の解説

- タイマがイベント信号を発行する時間間隔は、二種類（1 秒と 1ms）を切り換えて使い分けます。キーボードからの文字有力を受け付けている状態は、タイマは停止しているか、PAUSE 状態で時間間隔が１秒になっていますので、キーイベントに入ることができます。
- PAUSE 状態でないとき、タイマは停止しています。文字入力が済んで、文字列の解読が済んで、その文字列の解読処理をさせるには、OnKeyPress イベント関数から出るとき、タイマを 1ms で起動させます。そうすると、ユーザのキー入力に応答する暇なしに、直ぐに OnTimer イベント関数に制御が入ります。
- OnTimer イベント関数に制御が入ると、タイマを停止させ、単純に制御を pbMain()に渡します。ここが、Plain_Basic の実質的な処理の入口です。ここから帰ると、基本的には制御をシステムに戻しますので、キーボード入力を受け付けます。
- PAUSE は、pbMain()の中でコマンド解読によって起動されます。このときは、時間間隔を１秒に設定し、(bTimerFlag=on)で pbMain()から帰ってきますので、そのまま制御をシステムに帰します。システムは１秒間隔で OnTimer イベント関数を呼びますが、(bTimerFlag=on)の場合には待ち秒数をカウントダウンして、制御をシステムに帰します。OnTimer イベント関数の時間間隔が１秒ですので、その間にキーボード入力があれば、OnKeyPress イベント関数に制御が入ります。この場合には、PAUSE が取り消されるようにします。
- OnTimer イベント関数の中で、待ち秒数がなくなったことを検知すると、タイマを停止し、通常のコマンド解読処理に戻り、pbMain()に制御が入り直します。
- バッチ処理は、pbMain()の中でバッチファイルが open され、(bDeckMode=on)のフラグが立ちます。通常は pbMain()から処理が出ません。しかし、バッチファイルの中で PAUSE コマンドが呼ばれると(bTimerFlag=on)となって、pbMain()から制御が戻り、さらに制御がシステムに帰ります。したがって、キー入力を受け付けますので、PAUSE を終了させるか、バッチ処理も同時に中止するかの選択が OnKeyPress イベント関数の中で選択できます。
- PAUSE 時間が終了したとき、まだバッチ処理の残りがあるなら、1ms の時間間隔でタイマを起動させて一旦システムに制御を戻し、OnTimer イベント関数に入り直し pbMain()を呼びます。少し回りくどい方法ですが、goto 文を使わないコーディングをしました。
- タイマが起動していない場合には、pbMain()から制御が帰ったとき、システムに制御を帰します。このとき、テキスト入力枠にフォーカスをセットします。

表 1.3　OnTimer イベント関数のリスト

```
void __fastcall TFormMain::Timer1Timer(TObject *Sender)
{                                           // 2006-09-10
     String LabelNo;
     if (bTimerFlag){
          iTime--;
          LabelNo = AnsiString((int)iTime);
          FormMain->Label1->Caption = LabelNo;
          if (iTime>0) return;
          pbTimerDeactivate();
          if (!bDeckMode) return;
     };
     FormMain->Timer1->Enabled = false;
     pbMain();
     if (bTimerFlag) return;
     if (bDeckMode) {
          FormMain->Timer1->Enabled = true;
          return;};
     Edit2->SetFocus();
}
```

1.5　ソースコードの全体構成

Plain_Basic は C/C++でコーディングしたこと

Plain_Basic の原形は、Fortran でコーディングした NUCE_BASIC です。元のサブルーチンは、その数が約 60 ありました。これらは、管理の便のため、複数のファイルに分けてあります。サブルーチンは、他のライブラリプログラムと組み合わせて利用しますので、区別が付き易いように、名前の英字名は B の頭文字を付けました。Fortran の命名規則では、名前は英数字 6 文字以内の制限がありましたので、名前はやや記号的です。Fortran の利用環境が専門に偏り、一般の利用に不便になりましたので、改めて 2006 年に、パソコンレベルでも汎用性のある C/C++言語に直してコーディングしました。元の Fortran 言語のソースコードを読んで、C 言語に変換するユーティリティの利用も試みました。しかし、その C 言語のソースコードでは部分的な修正作業に向きませんので、最初から C 言語の特徴を生かすように書き直しました。その際、元になった Fortran のソースコードのアルゴリズムを生かしましたので、元のサブルーチン単位で C/C++言語の関数に書き直したものが大部分です。この種の関数は、英字名の頭を pb で始めてあります。グラフィックス関係の関数は、dp の頭文字を持つ名前ですが、これは元の Fortran のサブルーチン名そのままです。

オブジェクト管理用のモジュールを追加すること

Plain_Basic は、擬似的な DOS 画面を複数個使い、これを介して入出力処理をします。この画面単位がフォーム、通称でウインドウです。画面の構成要素個々について、幾何学的デザインと機能のプログラミングが Plain_Basic 本体と密接に絡みます。この部分のソースコードは、Windows の機能を使うために余分に必要になったモジュールです。フォームごとに三つのファイルの組で構成され、その拡張子は.cpp　.h　.dfm です。この骨格は Borland C++Builder が自動作成しますので、ユーザレベルではその(.cpp)ファイルにコードを追加します。そこには、比較的分かり易い名前、例えば CaptionDisplay、などの関数名を使いました（表 1.4 参照)。

表 1.4　Plain_Basic の関数名一覧

1	CaptionDisplay	31	pbData2	61	pbQuit
2	ClipboardCopy	32	pbDefType	62	pbRead0
3	ClipboardPaste	33	pbDelete	63	pbRegister
4	consoleClear	34	pbDim	64	pbRenum
5	ConsoleFont	35	pbDim1	65	pbSaveName
6	ConsoleInitialize	36	pbDim3	66	pbSet0
7	ConsolePrint	37	pbEditLine	67	pbSet1
8	dpCirc	38	pbEncd0	68	pbSet2
9	dpDraw	39	pbEncd1	69	pbSet4
10	dPenSz	40	pbErase	70	pbSkip
11	dPenTx	41	pbErrList	71	pbSkipB
12	dpEras	42	pbErrList2	72	pbSkipL
13	dpMark	43	pbFileName	73	pbSkipP
14	dpMove	44	pbFor	74	pbSkipQ
15	dpText	45	pbFunc	75	pbSortAddress
16	dpWind	46	pbIfGo	76	pbSortLineNr
17	GraphicsFont	47	pbLet1	77	pbTake
18	LabelDisplay	48	pbList	78	pbText0
19	LoadBmpFile	49	pbListF	79	pbText1
20	pbAcc	50	pbListLines	80	pbTimerActivate
21	pbAcc2	51	pbLoadName	81	pbTimerDeactivate
22	pbAuto	52	pbMain	82	PrintBmpFile
23	pbCellBack	53	pbMain2	83	ReadFileToConsole
24	pbCellTake	54	pbName	84	SaveBmpFile
25	pbCharNext	55	pbNew	85	SaveFileFromConsole
26	pbCharRead	56	pbNext	86	VRNDM
27	pbCharTake	57	pbNextFor	87	Write6
28	pbCommand1	58	pbOption		
29	pbCommand2	59	pbPause		
30	pbData1	60	pbPrint		

表 1.5　pbMain()のリスト：動作モードの判定をして文字並びのバッファーを選択する

```
void pbMain(void)               /* set address of text string */
{                                   // 2006-09-10
     Char *chptr, *r;
     String Text;
     Word nLabel, len, iKwCod;
     bErrorFlag = false;
//                                  /* enter here first with console input */
     for (;;){                  // >>>>>>>>>>>>> for Loop-A >>>>>>>>>>>>>
          if(!bRunMode) {       /* direct mode */
               if (!bFileMode) execPointer=textBuff;
               if (bFileMode) {     /* file input, Load/Merge mode */
                    r = fgets( textBuff, BUFFER_SIZE, fin );
                    if( r == NULL ) {
                         fclose(fin);
                         bFileMode = false;
                         pbSkipL();
                    }else{
                         iTextLength = strlen(textBuff)-1;
                         if (textBuff[iTextLength] == '¥x0a')
                              textBuff[iTextLength] = '¥0';
                              execPointer=textBuff;
                    };              // endif r
               };                   // endif bFileMode
          };                        // endif !bRunMode
//
          if (bRunMode){        /* program mode */
               chptr = nextProgPtr;
               nLabel = *((Word *)chptr);
               execPointer = chptr+4;
               if(nLabel == 0 ) {
                    bRunMode = false;
                    pbSkipL();
               }else{
                    iCurrentLabel= nLabel;
                    len = *((Word*)(chptr+2)) ;
                    nextProgPtr += len;
                    if(bTraceFlag ) {
                         Text=AnsiString((Word)iCurrentLabel);
                         Write6(Text);};
               };                   // endif (nLabel)
          };                        // endif (bRunMode) */
//
          pbMain2();                /* string analyze and execute */
//
          if (bErrorFlag){
               if (bDeckMode) {fclose(fdeck); bDeckMode = false;};
               if (bFileMode) {fclose(fin);   bFileMode = false;};
               return;};
          if (bTimerFlag) return;
          if (bRunMode) continue;
          if (bFileMode ) continue;
          if (!bDeckMode) return;
       /* Deck file */
          r = fgets( textBuff, BUFFER_SIZE, fdeck);
          if( r == NULL ) {
               fclose(fdeck);
               pbSkipL();
               bDeckMode=false;
               return;};
          iTextLength = strlen(textBuff)-1;
          if (textBuff[iTextLength] == '¥x0a')
               textBuff[iTextLength]='¥0';
          if(bEchoFlag) Write6((String)textBuff);
          return;
     };                         // >>>>>>>>>>>>> end of for Loop-A >>>>>>>>>>>>>
}
```

表 1.6　pbMain2()のリスト：文字列解読とコマンド起動処理

```
void pbMain2(void)          /* string analyzer */
{                               // 2006-09-07
    Char ch;
    short nVal;
    Word iKwCod;
//
    for (;;) {
        ch = pbCharRead();
        switch (cbtype){
            case DIGIT:   /* save the statement into program area */
                if (bRunMode){pbErrList(ErrString02); return;};
                if (ch =='0'){pbErrList(ErrString08); return;};
                nVal = pbEncd1(); if (bErrorFlag) return;
                ch =pbCharRead();
                if ((cbtype != ALPHA) && (ch != '¥0')) {
                    pbErrList(ErrString21); return;};
                nByte = iTextLength + 1 - (execPointer - textBuff);
                pbEditLine(nVal);
                break;
            case EOS:
                execPointer++;continue;
            case EOL:
                return;
            case ALPHA:                 /* command analyser */
                pbName(); if (bErrorFlag) return;
                iKwCod = iKwrdID;
                if ((strKwName != "exit") && (bAutokey)) {
                    /* when program editing mode is continued */
                    execPointer=textBuff;
                    ch = pbCharRead();
                    nVal = iCurrentLabel - iIncreNo;
                    nByte = iTextLength;
                    pbEditLine(nVal); if (bErrorFlag) return;
                    break;
                };
                switch(iKwrdCode){ /* switch selector by Name */
                    case 1:             /* COMMAND */
                        if (iKwrdSubCode < 100) {pbCommand1(iKwCod);break;};
                        pbCommand2(iKwCod);break;
                    case 2:             /* STATEMENT not implemented */
                    case 3:         /* FUNCTION  not implemented */
                    case 4:             /* OPERATOR  not implemented */
                        pbErrList(ErrString02); return;
                    case 5:         /* VARIABLE // fall through */
                    case 6:             /* ARRAY*/
                        pbLet1(); break;
                    default:
                        pbErrList(ErrString02); return;
                };  /* end of switch selector (iKwrdCode) */
                break;                  /* exit from case ALPHA */
        };                              /* end switch selector (cbtype) */
//
        if (cbtype == EOS) {        /* multi statement */
            execPointer++;continue;};
        break;
    };                                  /* end of for */
}
```

文字並びの最初の文字種で判定の大枠を決める

文字並びの解読と処理の分岐は pbMain2()から始まります。処理の対象である文字並びデータのアドレスは、共通変数である文字型のポインタ exexPointer で渡されますので、具体的な文字データのバッファー位置は抽象化されています。文字並びを、意味のある語単位（C 言語で言うトークン）に分解する手始めは、先行するスペースとタブを読み飛ばして最初の 1 文字を pbCharRead()で取りだします。この文字種は、英字であるか、数字であるか、文末または行末かを判定します。数字の場合には、これを BASIC プログラム文のラベルと見なし、内部プログラム領域に文字並びを pbEditLine()で転送します。英字の場合は、pbName()に処理を渡し、そこで英数字名に切り出します。さらに、その英数字名が、キーワードであるか、変数名であるか、それとも、まだ宣言されていない変数名かを調べます。キーワード名の場合には固有のコードを持ちますので、コマンド別の処理に分岐させます。つまり、図 1.2 の path selector が実行されます。コマンドやステートメントの種類が多いので、pbCommand1(), pbCommand2(), pbCommand3()のように分けてあります。変数名である場合、代数式の代入文であると見なして、pbLet1()に制御を渡します。表 1.5 は、pbCommand1()のリストを示します。コマンドのキーワード別に switch 文で分岐させていますので、多くの処理が並びます。数行のリストで済むものはここに含ませますが、処理が入り組むものは個別の関数に分岐させるようにしてあります。pbCommand2(),と pbCommand3()はライブラリ用の関数を繋ぐルーチンです。引き数の形式がすべて同じになります。表 1.6 に、グラフィックス処理を繋いだリストを示します。代入文は、文頭に LET のキーワードを付けませんので、他のコマンド処理と独立して pbLet1()にまとめます。

表 1.5　pbCommand1()の概念を表すリスト：選択グラフを構成する

```
//========================================================================
void pbCommand1(Word iKwCod)
{
     String Text;
     Word nLabel, iRet, iRet0;
     Char fName[64], fDName[64], * chptr;
     switch (iKwCod){
          case KWD_AUTO:
               ------------------------------
          case KWD_EXIT:
               ------------------------------
          case KWD_DELETE:
               ------------------------------
          case KWD_LIST:
               ------------------------------
          ------------------------------
          ------------------------------
          default: pbErrList(ErrString02); break;
     };   // end switch (iKwCod)
}
```

表 1.6　pbCommand2()のリスト：ライブラリを接続する部分

```
//========================================================================
void pbCommand2(Word iKwCod)
{
    Word nOpr;
    int k1, k2, k3, k4, k5, k6, k7;
    int k8, k9, k10,k11,k12,k13,k14;
//
    pbText0(nOpr); if (bErrorFlag) return;
    k1 = (int) vBuff[1];
    k2 = (int) vBuff[15];
    k3 = (int) vBuff[29];
    k4 = (int) vBuff[43];k4=k4; // dummy statements
    k5 = (int) vBuff[57];k5=k5;
    k6 = (int) vBuff[71];k6=k6;
    k7 = (int) vBuff[85];k7=k7;
    k8 = (int) vBuff[99];k8=k8;
    k9 = (int) vBuff[113];k9=k9;
    k10 = (int) vBuff[127];k10=k10;
    k11 = (int) vBuff[141];k11=k11;
    k12 = (int) vBuff[155];k12=k12;
    k13 = (int) vBuff[169];k13=k13;
    k14 = (int) vBuff[183];k14=k14;
//
    switch (iKwCod){
        case KWD_DPWIND: dpWind(vBuff[k1],vBuff[k2],vBuff[k3]); break;
        case KWD_DPMOVE: dpMove(vBuff[k1],vBuff[k2]); break;
        case KWD_DPDRAW: dpDraw(vBuff[k1],vBuff[k2]); break;
        case KWD_DPERAS: dpEras(); break;
        case KWD_DPCIRC: dpCirc(vBuff[k1],vBuff[k2],vBuff[k3]); break;
        case KWD_DPTEXT: dpText(vBuff[k1],vBuff[k2],vBuff[k3]); break;
        case KWD_DPENSZ: dPenSz(vBuff[k1]); break;
        case KWD_DPENTX: dPenTx(vBuff[k1]); break;
        case KWD_DPMARK: dpMark(vBuff[k1]); break;
        default: pbErrList(ErrString02); break;
    };
}
```

表 1.7　pbLet1()のリスト：代数式を解読する部分

```
/*=========================== 840702 === FORTRAN 77S ======= BLET1
      SUBROUTINE BLET1(TEXT,ICTRL)
  EQUATION ANALYZER.
*/
void pbLet1(void)
{
    double DIS,AX,AY;
    Word ist0, ist, ist1, icType, idType,nCnt,mCnt,ntnv,ntnd;
    Word iMod;
    Char ch;
    int iPoint;
/* set l_value of equation */
    if (bErrorFlag) return;
    icType = iVarType;
    ist0 = pbCellTake();
    ist = ist0;
    iMod = 0;                   // array evaluation Mode
    pbSet0(ist, iMod); if (bErrorFlag) return;
```

```
/* shall be squal sign '=' */
     ch = pbCharRead();
     if (ch != '=') {pbErrList(ErrString02); return;};
/* set r_value of equation */
     execPointer++;
     ist1 = pbCellTake();
     iKwrdCode = 0;
     iMod = 1;                          // arithmetical calculation Mode
     pbSet0(ist1,iMod); if (bErrorFlag) return;
     icType = ListCell[ist0].icType;
     idType = ListCell[ist1].icType;
     iPoint = ListCell[ist0].elementAddress;
/* assignment */
     switch (icType){
          /* 1: ---- ASSIGNMENT EXPRESSION SCALOR=SOMETHING */
          case INTEGER: // fall through
          case DOUBLE:
               switch (idType){
                    case INTEGER: // fall through
                    case DOUBLE:  // fall through
                         vBuff[iPoint] = ListCell[ist1].dValue[0];
                         break;
                    default:
                         pbErrList(ErrString13); return;
               };
               break;
          case STRING:
               switch (idType){
                    case STRING:
                         vBuff[iPoint] = ListCell[ist1].dValue[0];
                         vBuff[iPoint+1] = ListCell[ist1].dValue[1];
                         break;
                    default:
                         pbErrList(ErrString13); return;
               };
               break;
     };
/* return ListCells */
     pbCellBack(ist0);
     pbCellBack(ist1);
}
```

共通変数名でデータの効率的な利用を図ること

Fortran のサブルーチンは、処理に使うデータや変数を、呼び出し側から得る場合も、結果を返す方法にもアドレス参照の引き数を主に使います。したがって、C/C++言語に直した関数も、アドレス参照を使う引き数が多くなりました。Fortran では、引き数の個数を減らすには、共通領域(COMMON)を経由する方法があります。C/C++に変更する場合も共通変数で扱うように変更して、引き数の数や種類を減らしました。関数は複数のファイルに分けて作成しますので、他のファイルからの参照には extern 宣言で関連を付けます。ただし、意図しない箇所で変数の値が書き換えられるとエラーが起こりますので、その点の注意が必要でした。複数のファイルから参照する定数は、名前を#define 文で定義したものを多く使います。これらの定義は、ソースコードを判読するときに必要ですので、下にリストを示します。

表 1.8　定数の定義名

```
#define EOL       0    /* physical end of text string '¥0' */
#define EOS       58   /* logical End of Statement ':' */
#define BUFFER_SIZE         128
#define SIZE_FOR_PROGAREA   32000
#define STACK_MAX           30
#define SIZE_FOR_ARRAYS     10000
#define TOP_OF_VARIABLE     200
#define DIGIT      1
#define SYMBOL     2
#define ALPHA      3

#define COMMAND    1
#define STATEMENT  2
#define FUNCTION   3
#define OPERATOR   4
#define VARIABLE   5
#define ARRAY      6
#define INTEGER    101
#define DOUBLE     201
#define STRING     302

#define PLUS                0    // +
#define MINUS               1    // -
#define MULTIPLY            2    // *
#define DIVIDE              3    // /
#define POWER               4    // ^
#define LOGICAL_AND         5    // &
#define EQUAL               14   // =
#define LESS_THAN           15   // <
#define GREATER_THAN        16   // >
#define LESS_OR_EQUAL       17   // <=
#define GREATER_OR_EQUAL    18   // >=
#define NOT_EQUAL           19   // <>
#define LOGICAL_OR          20   // @, |
#define  KWD_AUTO      1001
#define  KWD_BREAK     1002
#define  KWD_EXIT      1003
#define  KWD_DELETE    1004
#define  KWD_LIST      1005
#define  KWD_RENUM     1006
#define  KWD_LOAD      1007
#define  KWD_SAVE      1008
#define  KWD_MERGE     1009
#define  KWD_END       1010
#define  KWD_NEW       1011
#define  KWD_DEFINT    1012
```

```
#define  KWD_DEFDBL   1014
#define  KWD_DEFSTR   1015
#define  KWD_DIM      1016
#define  KWD_ERASE    1017
#define  KWD_DECK     1018
#define  KWD_REM      1019
#define  KWD_CONT     1020
#define  KWD_GOTO     1021
#define  KWD_RUN      1022
#define  KWD_GOSUB    1023
#define  KWD_RESTORE  1024
#define  KWD_STOP     1025
#define  KWD_DATA     1026
#define  KWD_READ     1027

#define  KWD_PRINT    1029
#define  KWD_FOR      1030
#define  KWD_TO       1031
#define  KWD_STEP     1032
#define  KWD_NEXT     1033
#define  KWD_IF       1034
#define  KWD_THEN     1035
#define  KWD_ELSE     1036
#define  KWD_ON       1037
#define  KWD_RETURN   1038
#define  KWD_TROFF    1039
#define  KWD_TRON     1040
#define  KWD_ECHOFF   1041
#define  KWD_ECHON    1042
#define  KWD_OPEN     1043
#define  KWD_CLOSE    1044
#define  KWD_LET      1058
#define  KWD_CLS      1059
#define  KWD_PAUSE    1060
#define KWD_QUIT      1099

#define  KWD_DPWIND   1161
#define  KWD_DPMOVE   1164
#define  KWD_DPDRAW   1165
#define  KWD_DPENSZ   1166
#define  KWD_DPENTX   1167
#define  KWD_DPMARK   1168
#define KWD_DPERAS    1169
#define KWD_DPCIRC    1170
#define KWD_DPTEXT    1171

#define  KWD_ABS      3021
#define  KWD_ATN      3022
#define  KWD_COS      3023
#define  KWD_EXP      3024
#define  KWD_LOG      3025
#define  KWD_RND      3026
#define  KWD_SGN      3027
#define  KWD_SIN      3028
#define  KWD_SQR      3029
#define  KWD_TAN      3030
#define  KWD_ATN2     3031
#define  KWD_CBRT     3032
#define  KWD_DIS      3033
#define  KWD_FLAG     3034
#define  KWD_MOD      3035
```

```
const String ErrString01 ="Err01:NEXT without FOR";
const String ErrString02 ="Err02:Syntax Error";
const String ErrString03 ="Err03:RETURN without GOSUB";
const String ErrString04 ="Err04:out of data";
const String ErrString05 ="Err05:illegal function call";
const String ErrString06 ="Err06:over flow";
const String ErrString07 ="Err07:out of memory";
const String ErrString08 ="Err08:undefined line number";
const String ErrString09 ="Err09:subscript out of range";
const String ErrString10 ="Err10:redimensioned array";
const String ErrString11 ="Err11:division by zero";
const String ErrString12 ="Err12:program area overflow";
const String ErrString13 ="Err13:type mismatch";
const String ErrString14 ="Err14:out of string space";
const String ErrString15 ="Err15:string too long";
const String ErrString16 ="Err16:string formura too complex";
const String ErrString17 ="Err17:can not CONTinue";
const String ErrString18 ="Err18:undefined function";
const String ErrString19 ="Err19:no resume";
const String ErrString20 ="Err20:resume without error";
const String ErrString21 ="Err21:unprintable error";
const String ErrString22 ="Err22:missing operand";
const String ErrString23 ="Err23:line buffer overflow";
const String ErrString24 ="Err24:file protected";
const String ErrString25 ="Err25:bad file data";
const String ErrString26 ="Err26:disk open error";
const String ErrString27 ="Err27:disk area overflow";
const String ErrString28 ="Err28:file not open";
const String ErrString29 ="Err29:disk read error";
const String ErrString30 ="Err30:name exceeds 15 bytes";
```

表 1.9　共通変数名

```
/* Program area cotrols */
Char progBuff[SIZE_FOR_PROGAREA];
Word codeTop;              /* top address of unused progBuff */
Word iTextLength;
Word nByte;                /* effective Text Length */
Word cbtype;

/* AUTO numbering controls */
Word iAutokey;
Word iStartNo;
Word iIncreNo;

/* I/O controls */
Char textBuff[BUFFER_SIZE]; /* 128 bytes for console input */
Char printBuff[BUFFER_SIZE];
String textInput ;
FILE *fin, *fout, *fdeck;   /* input and output files */
String InputFileName;
String OutputFileName;
Word ReadRegister;         /* READ/Data control */
Word DataRegister;         /* data file open flag */
Char *recoverPointer;
Word DataRecordID;

/* execution controls */
bool bErrorFlag;           /* return flag when error */
bool bDeckMode = false;    /* batch file on active */
bool bRunMode = false;     /* run mode on active */
bool bFileMode = false;    /* =true on LOAD/MERGE */
bool bTraceFlag = false;   /* tron/troff */
bool bEchoFlag = true;     /* echon/echoff */
Char * execPointer;        /* progBuff or printBuff or NULL */
Char * resumePointer;      /* restart address by "cont" */
Char * currentBuffer;      /* textBuff or program address */
Char * currentProgPtr;
Char * nextProgPtr;
Word iCurrentLabel;        /* 0 under direct mode */
Word nextLineNumber;       /* next Label number */
Word nextRegister;         /* for-next controls */
Word returnRegister;       /* gosub-return controls */

/* READ-DATA controls */
Char * currentDataPtr;
Char * nextDataPtr;
Char * dataPointer;
Word iDataLabel;

int iFLAG = 0;             /* system constant */

/* Alphabetic keyword names control */
Char kwName[16];           /* alphabetic name */
String strKwName;          /* as above */
Word iKwrdCode;            /* 1 to 6 */
Word iKwrdSubCode;         /* function code etc. */
Word iKwrdID;              /* 1001 to 6000 */
Word iKwrdNarg;            /* number of dimensional components 0-4 */
int iKwrdPoint;            /* address to the controls */

/* Management on variables and arrays
```

```
double vBuff[SIZE_FOR_ARRAYS];   /* current size 10K */
Word typeOfVariable[26];    /* alphabetically specified */
Word maxVarSize = SIZE_FOR_ARRAYS;
Word iVarTop = TOP_OF_VARIABLE;
Word iVarEnd;               /* end address of data, initially = iVarTop */
Word iVarDim;               /* number of dimension: 0-4 */
Word iVarSize;              /* number of components in element:1-4,6,12 */
Word iVarElementSize[4];    /* A[d1][d2] ... */
Word iElementCode;          /* type code:1-3,11-16,21-27 */
Word iVarType;              /* 101 --- 1612 */

union {
     double dValue;         // first control data
     struct {
          unsigned int ElementLength: 32;  // including 4 control data
          Word ElementID: 16;              // 101 - 1612
          unsigned char ElementDim: 8;     // 0 to 4
          unsigned char ElementSize: 8;    // 1-4,6,12
          } SV;
     } UE;
union {                     // second control data
     double dValue;
     Word dimSize[4];
     } UD;
union {
     double dValue[2];      // third and forth control data
     char ElementName[16];
     } UC;                  // from here component data

/* statement list encoding */
int NrOfOperand;            /* counter of operand number */
int NumData[10];            /* list of integer data */

/* Management of stacks */
Word stackMaxSize;          // NMAX
Word stackNextCell;         // IPOINT
Word stackRemainder;        // NFREE
Word stackMaxUsed;          // IPMAX
struct {
     Word iMother;          // 1 parent cell no.
     Word iDaughter;        // 2
     Word iPar;             // 3 pointer to parenthesis data
     Word iNextPar;         // 4
     Word iCode;            // 5 function code etc.
     Word iDim;             // 6 number of arguments; 0=single Element
     Word iSenior;          // 7
     Word iJunior;          // 8  iSibling
     Word iOpeCode;         // 9
     Word icType;           // 10    INTEGER, DOUBLE, STRING etc.
     Char elementName[16];
     int elementAddress;
     Char * returnPtr;      // for 'FOR or GOSUB'
     Char * nextProgPtr;    // save nextProgPtr
     double dValue[12];     // all variables are saved as double
} ListCell[STACK_MAX];
```

2.　文字並びの解読処理

2.1　文字並びのデータ構造

文字並びを保存する領域が幾つかあること

C 言語の環境では、文字データは char 型の配列で扱います。Plain_Basic のソースコードでは、キーボードから一行分の文字並びを受け取るバッファー領域 textBuff[]、リスト書き出し用 printBuff[]を使い、128 バイト分を予約しています。BASIC 流のプログラム文を内部に保存するため progBuff[]を準備しますが、これは 30KB 長さとしました。変数名、キーワード名の長さは 15 バイト以内に制限しましたので、作業用に 16 バイト長のローカルな配列が幾つかあります。一方、Visual Basic では文字列型(String)があって、不定長さの文字並びのデータを一つの変数のように扱うことができます。二つの文字並びを比較する場合には文字列型の変数が便利です。C++言語では、class 定義で文字列型を使うことができますので、文字型配列と共用し、作業次第で相互にデータを変換しています。なお、RichTextEdit に表示する文字データの保存領域は、Windows のシステム機能を利用していますので、Plain_Basic のソースコードには宣言がありません。

文字並び解読の基本的な約束

Plain_Basic は、文字並びの読み込みの仕掛けと、その文字並びを解読する処理とがコーディングの主要部を構成しています。直接モードの場合、文字並びの読み込み部分は、「Console ==>」のラベルが付いたテキスト入力枠を使います。キーボードからのテキスト文字列を受け付け、それを文字型配列のバッファーtextBuff[]に転送します。バッチ処理モードでは、ファイルからの文字並びが同じ textBuff[]に入ります。取り込まれた文字列の解読は、Plain_Basic の構文規則に則って行ないます。この文字並びは、最初の文字を判断することから始めます。それを要約すると下のようになります。

(1)　先行するスペース、タブは無視して読み飛ばします。
(2)　途中にあるスペース、タブの連続は、一つのスペース扱いです。
(3)　スペースは、意味のある文字並び（トークン）に分解するときの補助に使います。
(4)　文字並びは、英字か数字で始まります。記号が最初にくるとエラーとします。
(5)　英字で始まる文字並びは名前として切り出します。名前は英数字並びの語です。
(6)　一つの名前の区切りは、スペースまたは記号で判断します。
(7)　名前は、キーワードか、そうでなければ変数名と解釈します。
(8)　名前にドル記号($)の接尾辞が付く場合は、これを含めて文字型変数名と解釈します。
(9)　キーワードで始まる文字並びは、コマンドまたはステートメントです。
(10) 変数名で始まる文字並びは、数式（代入文）と見なします。
(11) 数学関数名は英字ですが、最初にくる場合(左辺値)はエラーとします。
(12) 数字が最初にくるときは、その数字は BASIC プログラム文のラベルと解釈します。
(13) このとき、最初の数字を 0 で始めません。

文字処理の関数の種類

入力された文字並びは、一文字ずつ順に読み出して、意味のある単語単位、または数字並びに切り出します。一文字単位で文字を読み出すのは、pbCharRead(), pbCharNext(), pbCharTake()の関数を使い分けます。英字で始まる単語は、名前の解読処理 pbName()が行なわれます。数字の場合には、数字並びを内部表現である２バイトの整数型か、８バイトの実数型の数に変換します。これをエンコードと言います。エンコードに使われる処理には二つの関数 pbEncd0()と pbEncd1()とがあります。ただし、Plain_Basic では、８進数、16 進数を扱いません。プログラム文のラベルは整数を当てますので、整数専用の pbEncd1 を主に使います。pbEncd0 は、一般的なエンコード処理ですので、数式処理のところで使われます。コマンドやステートメントは、その後に引き数並びを持ちます。引き数の文字並びの解読は、pbText0()、pbText1()で行います。コマンドの引き数は、数字または単純な英字だけを使うので、単純な文字処理の応用でコーディングします。ステートメントの場合、引き数に変数も使いますので、その変数のアドレスを参照しなければなりません。この解読は、入り組んだ処理になります。変数に関する処理の説明は５章以降で扱います。

2.2　一文字を取り出す関数 pbChar Read/Next/Take

共通変数のポインタを使う

　文字型のバッファー配列から１文字を読み出す処理は、その配列の先頭アドレスを元に、文字位置の絶対アドレスを文字型ポインタの共通変数 execPointer にセットして参照します。対象とする配列のアドレスは作業に応じて変えますが、ここで解説する関数の外で行います。したがって、作業中の配列のアドレスを仮に保存して別の配列を読み、再び元のアドレスに復元する処理も行われます。ポインタが指す文字位置は、最初、バッファー配列の先頭にあります。文字読み出し手順の便宜のため、ポインタの扱いが異なる３つの文字型関数(表 2.1 参照)を使いますが、どれも基本的には同じです。関数のリストは表 2.2 に示します。これらの３つの関数は、文字の種類が何か（英字・数字・記号）を通知する共通変数 cbtype に値をセットします（表 2.3 参照)。文字並び終端の判別は、C 言語の仕様により、NULL キャラクタ’¥0’で識別します。英字は内部作業でキーワードと比較する目的がありますので、すべて小文字に直して取り出します。なお、ここで扱う文字解読は、漢字を記号と解釈します。

表 2.1　文字読み出しの関数（ポインタは読み出した文字位置を与えます）

関数名	処　　理
Char pbCharRead(void)	スペース、タブを読み飛ばして１文字を読取る
Char pbCharNext(void)	ポインタを一つ進めてから１文字を読取る
Char pbCharTake(void)	現在位置の文字を読取る

表 2.2　機能の違う三つの文字読み出し関数のリスト

```
//=================2006-02-17==========================================
Char pbCharRead(void){      /* find character skipping spaces */
     Char ch;
     for(;;){
          ch = *execPointer++;
          if (ch != ' ' && ch != '¥t') break;
     }
     execPointer--;
     return pbCharTake();
}
//=================2006-02-17==========================================
Char pbCharNext(void){      /* find next character  */
     execPointer++;
     return pbCharTake();
}

//=================2006-02-17==========================================
Char pbCharTake(void){      /* take next character incrementally */
     Char ch;
     ch = *execPointer;
     if (ch >= 'A' && ch <= 'Z'){
          ch = ch - 'A' + 'a';
          cbtype = ALPHA; return ch;};
     if (ch >= 'a' && ch <= 'z'){
          cbtype = ALPHA; return ch;};
     if (ch >='0' && ch <= '9'){
          cbtype = DIGIT; return ch;};
     if (ch == '¥0'){
          cbtype = EOL; return ch;};
     if (ch == ':'){
          cbtype = EOS; return ch;};
     cbtype = SYMBOL;
     return ch;
}
```

表 2.3　内部作業に使う共通変数名

共通変数名	値	英字記号	解　説
Char * execPointer			配列内の文字位置へのポインタ
Word cbtype	0	EOL	end of line, 文字列の終端文字 '¥0'
	1	DIGIT	数字 (0 - 9)
	2	SYMBOL	EOL,EOS 以外のすべての記号コード
	3	ALPHA	英字 (A - Z),(a - z)
	59	EOS	end of statement, ステートメント区切り':'コロン

2.3　名前を取り出す関数 pbName()

処理のあらまし

pbName()は、英字名の取り出しとその英字名の性質（プロパティ）を解読する処理です。キーボードから入力された文字列などを読み出して解読する処理の中で、英字が現われた所で、この pbName()に処理が入ります。文字ポインタは、その先頭英字位置を指しています。そこから始まる英数字並びを名前として切り出し、共通変数領域の文字配列に納め、その名前のプロパティも共通変数領域に返します。名前の区切りは、記号文字が現われたところで判定し、ポインタはその記号文字を指すように変ってpbName から帰ります。名前のプロパティとは、その名前がキーワードであるか、単純変数名か配列変数名であるか、変数名であるときはその型・保存位置などの情報を言います。キーワードとの比較は、キーワードのデータを納めたデータ構造の定義を利用します。また、変数の場合には、そのデータの内部記憶方式の定義を必要とします。ここでは、これらのデータ記憶方式の説明をして、名前検索の手順を解説します。

英字名のプロパティを引き出して準備する

英字名のプロパティは、下のような項目であって、コモンのデータ領域に作成します。変数の場合は、これに続けてやや複雑な情報を使いますので、プロパティの種類を下に続けて示します。

表 2.4　名前のプロパティ

プロパティ	解　説
Char kwName[16]	読み出した英字名の保存場所、名前の長さは最大 15 バイトに制限しました。
String strKwName	上の英字名を文字列型に直して、名前の比較や印刷データに使います
Word iKwrdCode	1 = 基本のキーワード名：Plain_Basic の予約語です。 2 = 追加分のコマンド名：(Plain_Basic では組み込んでいない。) 3 = 関数名：例えば ABS,SIN,COS などです。 4 = Plain_Basic では組み込んでいない（演算子名：例えば AND,OR,XOR などです） 5 = 単純変数名：上の 1～4 に当たらないとき。 6 = 配列変数名：左カギ括弧 [が続くことで配列名と判断します。
Word iKwrdNarg	引き数の数です。コマンド、関数、配列の場合に使います。
Word iKwrdSubCode	複数の基本キーワードには遠し番号が割り当てられています。
Word iKwrdID	1000*iKwrdCode + iKwrdCode の数値です。#define 文で名前が定義してあります。

表 2.5　変数・配列のプロパティ

プロパティ	解　説
Word iElementCode	1 = 整数型：内部的には倍精度実数で処理しています。 2 = 実数型：倍精度実数（8 バイト長）だけを扱います。 3 = 文字型：15 バイトまでの固定長が 1 単位です。終端記号に'¥0'が付きます。 (4 以上は幾何言語 G-Basic で定義する型番号です。ここでは使いません。)
Word iVarSize	8 バイト単位で計算した変数の寸法です。整数型・実数型は 1、文字型は 2 です。
Word iVarType	100*iElementCode + iVarSize の値です。#define 文で名前が定義してあります。 整数は 101、実数は 201、文字型は 302 です。
=(Word iKwrdNarg)	0 = 単純変数を表します。(引き数なしの配列名を参照しているときが 0 です。) 1,2,3,4 = 配列の次元数です。例えば、A[3],B[2,2],C[2,3,3],D[4,3,2,3]の形です。
Word iElementSize[4]	配列個々の次元の要素寸法を格納します。上の例 D では 4,3,2,3 と入ります。

キーワードはユーザ型で定義した構造体を配列に作る

キーワードのデータ型は、構造体 **KWORDITEM** で定義し、幾つかのキーワード構造体を配列で宣言してコモン領域に作成します。一つのキーワード構造体は、下のようにユーザ型を定義しました。キーワードの個数は、Plain_Basic 単独では 60 個程度ですが、外部サブルーチンを追加すると個数が増えますので、キーワード名の頭文字のアルファベット別に構造体の配列を作成して参照の能率が良くなるようにしました。

表 2.6　キーワードを保存する形式に使う構造体

```
typedef struct {
    Word id;           // 16 ビットの符号なし整数です。iKwrdID が入ります。
    Word nArg;         // 括弧付きで引用する関数名、サブルーチンなどの、引き数の数です。
    Char name[12];     // 英小文字・数字で表すキーワードの定義名で、最大 11 バイトです。
}KWORDITEM;
```

変数と配列は連続した大きな作業用配列に詰める

ユーザが宣言する変数・配列は一続きの作業用倍精度実数の配列 vBuff[] に、ユーザの宣言順に格納します。この作業用配列の寸法は SIZE_FOR_ARRAYS で宣言しますが、差し当たりは 10K を予約してあります。単純変数は、内部的には次元数 0 の配列として扱います。配列のデータは、頭に倍精度実数(8 バイト)で勘定した 4 語の管理用データを付け、その後に実際のデータを詰めます。そのため、単純変数であっても、作業領域には 5 語を使います。vBuff[]の配列管理用に幾つかのプロパティが用意されていますので、まず、それを（表 2.7）に説明します。

表 2.7　vBuff[]の管理に使うプロパティ

double vBuff[SIZE_FOR_ARRAYS]	#define SIZE_FOR_ARRAYS 10000 としてあります。
Word typeOfVariable[26]	英字名別の型情報(INTEGER,DOUBLE,STRING)を納めます。
int maxVarSize	= SIZE_FOR_ARRAYS
int iVarTop	#define TOP_OF_VARIABLE　200 としてあります。
int iVarEnd	初期値は=iVarTop です。

変数・配列の実際の値は vBuff[]の相対アドレスで参照する

vBuff[]は、大きな連続した配列ですが、その先頭の 200 語分は、関数やサブルーチンの引き数の情報を一時的に保存する作業領域に予約してあります。Plain_Basic 単独では 200 語も必要ではありませんが、幾何データを扱うバージョンで必要になるために予約したものです。この説明は、コマンド参照処理の章で詳述します。ユーザが宣言する変数または配列は、相対アドレス位置 **iVarTop** から入ります。これらのデータは、作業用配列内 vBuff[]の相対アドレス（先頭を 0 番地としたアドレス）で参照しますが、3 種類を使い分けます(表 2.8)。

表 2.8　vBuff[]の中で変数・配列の位置を表す相対アドレス（8 バイト単位で勘定する）

int iKwrdPoint	配列の管理用データの位置を示します。
iKwrdPoint + 4	配列の先頭アドレス。管理用データの寸法 4 を上に加えた値です。
int iVarPoint	配列の引き数から計算した実際のアドレス。単純変数では上と同じです。

管理用データの内部構造は共用体で定義する

配列管理用の４語は、倍精度実数１語８バイト（64 ビット長）を分割して目的別のデータを格納します。それらは構造体(struct)および共用体(union)に構成してあります。その詳細は、表 2.9 のように決めてあります。

表 2.9　管理用データの構成

共用体 UE	構造体 SV	unsigned int ElementLength:32	当該の変数または配列寸法+4。
		unsigned short ElmentID:16	型情報 iVarType が入ります。
		unsigned char ElementDim:8	配列次元数 iKwrdNarg が入ります。
		unsigned char ElementSize:8	iVarSize が入ります。
	double dValue		倍精度実数 1 個分 8 バイトです。
共用体 UD	unsigned short dimSize[4]		iElementSize[4]のコピーが入ります。
	double dValue		倍精度実数 1 個分 8 バイトです。
共用体 UC	char ElementName[16]		変数名または配列名の保存場所です。
	doubel dValue[2]		倍精度実数 2 個分 16 バイトです。

英字名の切り出し

キーワード名・変数名・配列名は頭文字が英字で始まり、英字と数字とを含む英数字名です。15 文字より長い名前を使いません。文字列の終を示すコード'¥0'のために１バイト分使います。英字は、大文字・小文字を区別しませんが、内部的には英小文字に直します。プログラム文はどちらを書いても構いません。プログラム文は、ユーザの入力したままで保存され、ファイルにもそのまま保存して利用できます。以前に作成した BASIC のプログラム文は英大文字だけを使いましたが、それらを再利用することができます。ユーザが宣言する変数名・配列名は、二つ以上の単語を繋いで一つの名前にしたいことがあります。ハイフンや点を使うことができませんので、その代わりにアンダースコア（__）を含めます。変数の型は、英字名の先頭文字(topChar)で決めます。変数の型は、型定義文 DEFINT,DEFDBL,DEFSTR で指定できます。デフォルトでは、Fortran 言語の習慣を踏襲して、I-N の英字の場合には整数型、それ以外は倍精度実数型に約束してあります。８ビットマイコン時代から、型定義文字(%, !, #, $)を接尾辞に使うと、整数型・単精度実数型・倍精度実数型・文字型を表す約束があって、便利に使えました。その仕様のうち、文字型の接尾辞($)だけを採用しました。記号#は、ファイル装置の番号指定用ですが、Plain_Basic では使いません。整数型のデータであっても、内部的にはすべて倍精度実数型で処理していて、型変換のときに小数以下の切り捨て処理をしています。

英数字名の区切りが括弧であるとき

意味のある英数字名の終りは、１文字づつ読み込んだ文字がスペースなどの記号文字に変ったときです。ただし、型宣言文字$は名前文字に繰り込みます。切り出した名前は、文字型配列 kwName[]に転送され、後ろに EOL('¥0')が追加されます。キーワード名や変数名・配列名との照合では、処理を簡単にするため、文字列型(String)に型変換したデータ strKwName の方を使います。この時点で、英数字名は、キーワードか単純変数名の可能性があります。そこで、区切り判定に使った文字が左括弧 '(' であるときは、関数名、カギ括弧'[' では配列名の仕様です。左括弧があれば、右括弧があるはずですので、pbDim1() を呼んで内部のコンマの数を勘定します。例えば、コンマが(，，，)と３つ見つかれば、４つの引き数がありますので、その数を iKwrdNarg に記憶させます。このとき、元の文字列解読に使うポインタ(execPointer)は、pbDim1()を呼ぶ前の値に復元します。なお、外部で作成したサブルーチンは引き数並びを括弧で括ってありますが、この処理名を Plain_Basic の追加コマンドに組み込むときは、括弧無しの書式で使い、関数扱いにならないようにします。

キーワード名との照合
　名前引き出しの次の処理は、キーワード名との照合です。英字名の頭文字は topChar に残してありますので、英字の頭文字別に配列に作成したキーワードの構造体を参照して、一致するキーワード名を探索します。一致するキーワードが見つかれば、表 2.10 の例のようにプロパティを設定して pbName()の処理から抜けます。

表 2.10　キーワードの照合結果を転送する値

IKwrdID = pKwrd->id	例えば、"LIST"は KWD_LIST で参照しますが、 #define KWD_LIST=1005 と定義されています この例では、iKwrdCode = 1, iKwrdSubCode= 5 です。 このキーワードでは、iKwrdNarg = 0 です。 変数ではありませんので、ポインタは 0 です。
IKwrdCode = iKwrdID / 1000	
iKwrdSubCode = iKwrdID - 1000*iKwrdCode	
iKwrdNarg = pKwrd->nArg	
IkwrdPoint = 0	

変数・配列名との照合
　キーワード名との一致が得られなかった場合には、配列名との照合に入ります。単純変数名は、引き数無しの配列として扱いますので、配列と同じ領域にデータを作成します。英字名(kwName)と一致する配列名を vBuff[]内で検索した場合、４通りの結果を得ます。名前の探索でも見つからないときは、変数または配列を作成します。これは pbName()で行うのではなく、pbName()から制御が帰ってから別の処理 pbSet0()などで行わせます。

表 2.11　変数名・配列名の検索結果を転送する場合の値

変数名で見つかった	iKwrdPoint： 変数管理位置の相対アドレスが入る iKwrdCode：　=5 になっている。 IkwrdSubCode：　=0 は単純変数である。 iKwrdnArg：　= 0 になっている。 iKwrdID： データ型が入る。例えば INTEGER=101。
既に宣言されている配列名が 引き数無しで見つかった	iKwrdPoint：上と同じ iKwrdCode：　=6 になっている。 iKwrdNarg：　=0 としておく（配列次元数） iKwrdID：管理データからデータ型をコピーする。 IElementSize[4]：管理データから次元要素数をコピーする
引き数付きの配列名が見つかった	iKwrdPoint：上と同じ iKwrdCode： = 6 になっているので、そのまま。 iKwrdNarg： 0 ではない。(配列次元数) 実の配列次元数と同じでなければエラーとする。 iKwrdID：管理データからデータ型をコピーする。 IElementSize[4]：管理データから次元要素数をコピーする
変数名・配列名ともになかった	iKwrdPoint： =0 で帰る

pbName()は、幾つかのプロシージャ内で呼び出され、その都度コモン領域にあるプロパティの値は上書きされますので、コモンデータ領域のプロパティ変数を生で引用すると異なった値を使う危険があります。したがって、呼び出し側では、必要なプロパティはその作業環境にコピーして使うようにしてあります。

表 2.12　pbName()のソースコードリスト

```
void pbName(void)                        // 2006-03-01
{          /* extract alpha-numeric name and set its characteristics */
     Char * dst;
     Char ch, topChar;
     int iArg, iSubCod, iDim, ibytecount, iPoint, iTypeA;
     bool bFound = true;
     bool bEnd;
/* initialize */
     iKwrdID = 0; iKwrdCode = 0; iKwrdNarg = 0; iKwrdPoint = 0;
     iKwrdSubCode = 0; iTypeA=0; ibytecount = 0; iSubCod = 0; iArg = 0;
     bFound = true; bEnd = false;
/* extract alpha-numeric name that may have underscores */
     execPointer--;
     dst = kwName;
     for(;;) {
          ch = pbCharNext();
          ibytecount++;
          if (ibytecount == 1) topChar = ch;
          if (ibytecount>14) {pbErrList(ErrString30); return;};
          if( cbtype == ALPHA || cbtype == DIGIT || ch =='_'){
               *dst++ = ch; continue;};
          break;
     };
/* set type of variable testing whether suffix is added */
     if (ch == '$'){iTypeA = STRING;};
     if (iTypeA != 0){
          *dst++ = ch;
          ibytecount++ ;
          ch =pbCharNext();};
     *dst++ = '¥0';
     iVarType = typeOfVariable[topChar-'a'];
     if (iTypeA != 0) iVarType = iTypeA;
     iKwrdCode = 5;          // possibly an element or a keyword
/* if bracket is used such as A[,,], find number of arguments */
     if (ch == '[') {
          iKwrdCode = 6;     // possibly an array
          iArg = pbDim1(); if(bErrorFlag)return;
          iKwrdNarg = iArg;};
/* if parenthesis is used such as B(,,), find number of arguments */
     if (ch == '(') {
          iKwrdCode = 3;     // possibly a function
          iArg = pbDim1(); if(bErrorFlag)return;
          iKwrdNarg = iArg;};
/* search on Keyword Table */
     strKwName = (String) kwName;
     KWORDITEM * pKwrd = KWordsIndex[topChar - 'a'];
     if( pKwrd != NULL ) {
          for(;;) {
               if( pKwrd->id == 0 ) break;          // not found
               if(strKwName ==((String) pKwrd->name)) {
                    iKwrdID = pKwrd->id;
                    iKwrdCode = iKwrdID / 1000;
                    iKwrdSubCode = iKwrdID - 1000*iKwrdCode;
                    iKwrdNarg = pKwrd->nArg;
                    iVarType=0;
                    bEnd = true; break;                       // found
               };
               pKwrd++;
```

```
        };
        if (bEnd) return;                                   // jump out
    };

/* seach on variables name */
    iPoint = iVarTop;
    for (;;){
        if (iPoint >= iVarEnd) {                            // not found
            iPoint = 0;
            if (iArg != 0) iKwrdCode = 6;
            bFound = false; break;};
        UE.dValue = vBuff[iPoint];
        UC.dValue[0] = vBuff[iPoint+2];
        UC.dValue[1] = vBuff[iPoint+3];
        if ((String)UC.ElementName == strKwName) break;     // found
        iPoint += UE.SV.ElementLength;
    };

    if (bFound) {
        UE.dValue = vBuff[iPoint];
        UD.dValue = vBuff[iPoint+1];
        iKwrdID = UE.SV.ElementID;      // 101 - 2712
        iSubCod = 1;
        iDim = UE.SV.ElementDim;        // 0 - 4
        if ((iArg != 0) && (iDim != iArg)) {
            pbErrList(ErrString16); return;};
        if (iDim != 0) {
            for (int i=0; i<iDim; i++) {iVarElementSize[i] = UD.dimSize[i];};
            iKwrdCode = 6;
            iVarDim=iDim;};
    };
/* resume */
    iKwrdPoint = iPoint;
    iKwrdSubCode = iSubCod;
    iKwrdNarg = iArg;
}
```

2.4　数字のエンコード処理の関数 pbEncd0/pbEncd1

整数だけのエンコードに pbEncd1 を使う

BASIC プログラム文のラベルに使われる整数は、2 バイト長の正の整数(最大 32767)の範囲の値であって、0 と負の数は使いせん。また、+－の符号付きでも使いません。コマンドの中でラベルを使うとき、ラベルの数の範囲を表す、例えば「LIST 10-500」のように－記号を区切り文字(delimiter)として使うことがあります。したがって、ラベルの数を解読するエンコード処理は、簡単に組み立てます。この目的に使う関数は pbEncd1 としました。文字列の解読では、数字が現われたときに、このルーチンに入り、数字以外の文字が現われたところを区切りとして整数値を返します。

表 2.13　整数であることが分かっている場合のエンコード、pbEncd1()のリスト

```
//==================2005-12-15=========================================/*
   encode to a short integer string in the text. such as in
         AUTO 10,10; RENUM 10,100,100; LIST 10-900
   plus sign(+) causes error.
*/
short pbEncd1(void)         /* encode to an integer with sign*/
{                                 // 2006-02-07
     int nVal;
     short nSgn, nShort, ibcount;
     Char ch;
     nSgn = 1;
     ch = pbCharRead();      // precedent spaces are skipped
/* if minus sign (-) exist */
     execPointer--;
     ch = pbCharNext();
     switch (cbtype){
          case SYMBOL:
               switch (ch){
                    case '+':
                         ch = pbCharNext();
                         if (cbtype != DIGIT){ pbErrList(ErrString16); return 0;};
                         execPointer--; break;
                    case '-':
                         nSgn = -1;
                         ch = pbCharNext();
                         if (cbtype != DIGIT){ pbErrList(ErrString16); return 0;};
                         break;
                    default:
                         pbErrList(ErrString17); return 0;
               };break;
          case DIGIT:execPointer--; break;
          default:
               pbErrList(ErrString17);
               return 0;
     };
/* encode procedure from string of digit */
     nVal = 0;
     for(ibcount=0; ibcount<6; ibcount++){
          ch = pbCharNext();
          if (cbtype != DIGIT) {
               if (nVal > 32767){pbErrList(ErrString18); return 0;};
               return (nVal * nSgn);};
          nVal *= 10;
          nVal += (ch - '0');
     };
     return (nVal * nSgn);
}
```

一般の実数のエンコードは pbEncd0()を使う

数学式で実数値を扱う場合には、整数だけでなく、-0.523E5 のような実数表現も使います。整数であっても符号付きで、絶対値が 65535 より大きい場合もありますので、pbEncd0 の方を使います。数式の中で実数値が現われたとき、＋－の符号文字と数字とが繋がっている場合と、その間にスペースが入ることもあります。＋－の符号文字は、演算子としての使い方がありますので、これらの文字が現われたら直ぐに pbEncd0 に処理を引き渡すことはしません。また、小数を表す場合、必ず０で始めます。例えば、0.125 を表すときに、小数点で始める　.125 のような表現を許していません。数字並びの切れ目は演算子記号やスペースなどです。指数表現が続く場合には、英字の E に続けて符号付きで２桁程度の整数が並びますので、この部分のエンコードには、上の pbEncd1 を利用します。ただし、E 以外の英字があらわれるとエラーとします。pbEncd0 によるエンコードは、数字並びの解析結果として整数型か実数型の区別が得られます。しかし、Plain_Basic の内部での扱いは、総て倍精度実数として扱います。

表 2.14　実数表現の数字のエンコード、pbEncd0()のリスト

```
double pbEncd0(void)        // 2006-02-07
{            /* a character string such as '1234.5678E-3' is analysed.*/
#define INT_MODE   0
#define REAL_MODE  1
     double fvalue = 0.0;   // real part value such as 1234
     double fdec =0.0;      // decimal part, such as 5678
     double fsign=1.;
     double dec =1.0;       // power of tenth, such as 10000
     const double d10=10.;
     int mode = INT_MODE;
     int ibyte, nval;
     Char ch;
     if (bErrorFlag) return 0.0;
/* initialize */
     ch = pbCharRead();
     if (cbtype == SYMBOL) {         // test + or -
          switch (ch){
               case '+': break;
               case '-': fsign=-1. ; break;
               default: pbErrList(ErrString02); return 0.0;
          };
          ch = pbCharNext();
          if (cbtype != DIGIT ){ pbErrList(ErrString02); return 0.0;};
     };
/* main loop of encoding */
     execPointer--;
     for (;;){
          ch = pbCharNext();
          switch (cbtype){
               case DIGIT:
                    ibyte = ch - '0';
                    switch (mode){
                         case INT_MODE:
                              fvalue *= d10;
                              fvalue += (double) ibyte;
                              continue;
                         case REAL_MODE:
                              fdec *= d10;
                              fdec += (double) ibyte;
                              dec *= d10;
                              continue;
                    };
               case SYMBOL:
                    switch (ch){
```

```
                    case '.':
                         if (mode==INT_MODE) {mode=REAL_MODE;continue;}
                         else {pbErrList(ErrString02); return 0.0;};
                    default:
                         fvalue += (fdec/dec);
                         fvalue *= fsign;
                         return fvalue;              // end of encoding
               };
          case ALPHA:
               if ((ch == 'd') || (ch == 'e')){
                    ch = *execPointer++;
                    nval = pbEncd1();
                    if (bErrorFlag) return 0.0;
                    if (abs(nval)>35) {pbErrList(ErrString06); return 0.0;};
                    fvalue += (fdec/dec);
                    fdec = pow(d10, nval);
                    fvalue *= fdec;
                    fvalue *= fsign;
                    return fvalue;
               }else{pbErrList(ErrString02); return 0.0;};
          default:                              // EOL or EOS
               fvalue += (fdec/dec);
               fvalue *= fsign;
               return fvalue;     // end of encoding
     };             // end switch(cbtype)
  };                // next for loop
}
```

2.5　引き数リストの解読処理：pbText0/pbText1

整数だけのリスト解読はpbText1()を使う

整数だけのリストを引き数に持つコマンドは、下のような種類があります。

表2.15　整数リストを使うコマンド

コマンド	引き数の数	例
AUTO	2	AUTO 100, 10 AUTO , 20 AUTO
DELETE	2	DELETE 100 DELETE 100-800 DELETE -1000 DELETE
RENUM	3	RENUM 10, 100, 100 RENUM
LIST	2	LIST 10-900 LIST 30 LIST -1000 LIST

整数リストの書き方は、コンマ’,’で区切る場合と、ハイフン’-’を挿む場合の二種類です。ここでのハイフンは、区切り記号(delimiter)として使います。マイナスの整数ではなく、数の範囲を表しますので、見易さのためにスペースを挿むのは自由です。リストの数字を省く書式も許しています。デフォルト値は0です。pbText1()は、delimiter文字を引き数指定にしました。整数のリストは、pbEncd1()を使ってエンコードし、共通領域の配列NumData[]に納めます。

表2.16　整数引き数のリスト解読、pbText1()

```
void pbText1(Char delimiter)
{                                // 2006-03-21
     int iCma, nVal;
     Char ch;
     iCma = 0;              // in case when a number is missed
     NrOfOperand = 0;
     for (;;){
          ch = pbCharRead();
          nVal = 0;
          switch (cbtype){
               case ALPHA:
                    pbErrList(ErrString02); return;
               case SYMBOL:
                    if (ch !=delimiter) {
                         pbErrList(ErrString02); return;};
                    if (iCma == 1){iCma = 0; execPointer++; break;};
                    NumData[NrOfOperand] = 0; NrOfOperand++;
                    execPointer++;
                    break;
               case DIGIT:
                    nVal=pbEncd1();
                    iCma=1;
                    NumData[NrOfOperand] = nVal;
                    NrOfOperand++;
                    break;
               case EOL:// fall through
               case EOS: return;
          };
     };
}
```

外部サブプログラムの引き数リスト作成は pbText0()を使う

Plain_Basic の本体は、別に作成した任意のサブプログラムを、追加コマンドまたは関数として組み込むことを考えてあります。その書式は、下のようなリスト形式です。

- コマンドの場合　　*procedureName*　A1, A2, A3, …
- 関数の場合　　*functionName*(A1, A2, A3, …)

ここに、*procedureName* 、*functionName* は、キーワードとして登録する英数字名です。コマンドと関数との区別は、この文字列を解読するとき、括弧の有る無しで判断します。Plain_Basic では、外部関数を使いませんが、グラフィックス用のサブルーチンをコマンドとして組み込みます。A1, A2, ...が引き数のリストの意味です。コンマだけの場合、および、要求された数の引き数に足りないときには、省略値が入るように決めます。また、余分な引き数があればエラーとします。引き数並びは、アドレス参照の変数名の他、具体的な数字データ（例えば 3.14 など）や引用符で囲んだ文字（例えば "abc123"）、式文（例えば a*x+b）を使うこともあります。引き数はすべて倍精度実数のアドレスで渡しますので、それを利用する側で変数型の調整を行わせます。

変数・配列ともに一続きの作業用配列を使う

Plain_Basic では、変数も配列も、共に一続きの倍精度数の配列 vBuff[]を部分的に区切って定義しますので、内部的には下のような形になります。

- *procedureName*　vBuff[K1], vBuff[K2], vBuff[K3], vBuff[K4], …
- *functionName*（vBuff[K1], vBuff[K2], vBuff[K3], vBuff[K4], …）

ここに、K1,K2,…は、配列 vBuff[]内の相対アドレスです。そのため、作業用に、変数領域 vBuff[]の先頭に 14 個の引き数用仮変数領域を予約しておきます。引き数 1 個当たり 14 語を使います。第 2.3 節の表 2.7 で、変数領域の先頭の相対アドレス iVarTop を 200 から始めたのは、この作業領域分を確保するためです。

仮引き数領域のデータ構造

倍精度実数を保存する領域 vBuff[]の先頭から、14 語ずつに区切った引き数の保存領域は、「型情報・実体位置を示す相対アドレス・引き数実体用 12 語分」と使います。最後の項目は、Plain_Basic では、倍精度型で１語、文字型で２語あれば良いのですが、幾何言語の変数では 12 語が必要ですので、その仕様を流用したためです。引き数の実体位置を示す相対アドレスは、変数名の場合は変数実体のアドレスが入りますが、数値や式文の場合にはその値を計算し、引き数実体用領域に保存しますので、相対アドレスは直ぐ次隣の位置を指します。

引き数解読は pbSet0()に処理を渡す

pbText0()のソースコードを表 2.17 に示すます。引き数の文字並びを解読し、その値・アドレスなどを求める作業用はリストセルを使い、そのリストセルを持って pbSet0()に処理を渡して計算します。この処理はかなり入り組んでいますので、第４章で解説します。

表 2.17　一般的な引き数のリスト解読、pbText0()

```
void pbText0(Word& nOpr)              // 2006-02-09
{
    Word iAddress, iMod, ist;
    Char ch;
    int iPoint;
  /* set default NULL arguments */
    nOpr = 0;
    for (int i=0;i<196; i++) {vBuff[i]=0.;};
    for (int i=0;i<196; i+= 14) {
        vBuff[i]=(double) DOUBLE;
        vBuff[i+1] = (double) (i+2); };
  /* test whether 'Keyword(A,B,C,...)' */
    ch = pbCharRead();
    execPointer--;
    if ((cbtype == EOL) || (cbtype == EOS)) return;
  /* main loop */
    for (;;){
        ch=pbCharNext();
        if ((cbtype == EOL) || (cbtype == EOS)) return;
        if (nOpr >14 ) {pbErrList(ErrString16);return;};
        if (ch == ',') {nOpr++; continue;};
      /* expression analyse */
        iMod = 1;
        iKwrdCode = 0;
        if (cbtype==ALPHA) pbName(); if (bErrorFlag) return;
      /* iMod =3 when name of array without parenthesis */
        if ((iKwrdCode ==6) && (iKwrdNarg==0)) iMod=3;
        ist = pbCellTake(); if (bErrorFlag) return;
        pbSet0(ist,iMod);   if (bErrorFlag) return;
        nOpr++;
        iAddress = 14*(nOpr-1);
        vBuff[iAddress]=ListCell[ist].iCode;
        if (ListCell[ist].elementAddress !=0) {
            vBuff[iAddress+1] = ListCell[ist].elementAddress; };
        for (int i=0;i<12;i++) {vBuff[iAddress+2+i]=ListCell[ist].dValue[i];};
        if (iMod != 1) {       // name of array
            iPoint=ListCell[ist].elementAddress + 4;
            vBuff[iAddress+2] = vBuff[iPoint];};
        pbCellBack(ist);
        ch = pbCharRead();
        if (ch == ',')  continue;
        execPointer--;
    };
}
```

3.　プログラム文の編集と管理

3.1　概説

プログラム文の保存領域を内部に持つこと

Plain_Basic は、原則的にコマンド駆動型のアプリケーションの集合です。処理の対象は主として数値データです。変数間の代数計算や、数値を代入する処理の文は、コマンド文とは言わず、ステートメント文に分類します。PRINT 文は、代表的なコマンド文です。用語の使い分けはやや曖昧ですが、コマンド文は、単独で処理が完結する１単位の命令の意義を持ち、ステートメント文の方は複数の文が相互に関連する場合を言います。FOR-NEXT 文が代表的なステートメント文です。何かの処理は、コマンド文またはステートメント文をテキスト入力枠から一行単位で入力させて直ぐに実行させることができます。これが直接モードです。一つの文の文字数が少ないときは、複数の文をコロン（：）で区切って一行にまとめることができます。この入力文の集合をまとめて保存しておいて、それを順に読み出して実行させるのが能率的です。この保存と実行の方法が二つあります。一つは、外部のテキストファイルに入力文を作成しておいて、テキスト入力枠からの入力に代える方法であって、バッチ処理モードです。二つ目は、これが本命の方法ですが、Plain_Basic 自体にテキストエディタの機能を持たせ、そこで文を編集して内部的に保存します。そして、その文を順に読み出して実行(RUN)させる方法です。これがプログラム実行モードです。この場合の文字並びは、行頭に整数のラベルを付けます。ラベル付きの文の集合が Plain_Basic のプログラムです。

プログラム文一行はラベル込みで 120 バイト以下とする

コンソールタイプライタを文字入力に使っていた時代、一行に納まる文字は、プラテン(platen)の物理的な幅から決まる 80 バイトで字数制限を受けました。現在のモニタは解像度が上がりましたので、プログラム文一行の物理的長さを、約 120 バイトにしました。テキスト入力枠は一行分を行末の自動折り返し無しで表示しますので、長い文字並びは頭の方が見えなくなります。スクロールバーはありませんが、カーソルを移動させれば見ることが出来ます。テキスト入力枠に入力された文字並びは、改行によって Plain_Basic に取り込まれ、そこの文字並びはクリアされ、エコーがテキストウインドウに表示されます。このテキストウインドウの横幅に入りきれない文字並びは、折り返して表示されます。

入力文字並びは実行されるか保存される

入力文字並びの先頭が英字であれば、直ぐに Plain_Basic が実行します（直接モード）。数字で始まる場合はプログラム文と解釈され、先頭の数字をラベルとして、内部のプログラム保存領域に文字並びを保存します。プログラム文１行の文字数は長短様々です。保存領域を有効に利用するため、プログラム文を連続して詰めます。一行のプログラム文は、管理用のデータを頭につけた図 3.1 のような論理構造です。管理データは、行のラベルとその行全部の文字数を、それぞれ２バイトの符号無し整数に作成し、それに続けて文字が並びます。文字並びは、その終端を表すコード'¥0'を持たせます。図 3.1 は、説明のために文字を左から右に並べて示しますが、連続したビット並びで考えると左右が逆に並びます。差し当たって特に問題にはなりませんが、２バイトを繋いで整数扱いをするときには上位バイト・下位バイトの区別を考慮しなければならないことがあります。図 3.1 では、相対的に右が上位バイト、左が下位バイトです。

…	ラベル保存	文字数保存	M	O	J	I	D	A	T	A	¥0	次のラベル	…

図 3.1　プログラム文 1 行のバイト利用方法（文字終端に'¥0'が入ります）

一続きの長い文字並びで保存すること

プログラム文の内部保存領域の寸法は、30KB の長い文字型配列を予約します。この寸法は、もしプログラム文を詰めて表すと、A4 用紙で約８ページの分量です。Plain_Basic のプログラム文編集とは、「テキスト入力枠からの文字列を保存領域に追加作成すること・必要に応じてテキストウインドウに表示すること・その文字列の修正や削除をすること」を指し、これを Plain_Basic 組み込みのエディタで行います。プログラム文の管理とは、このプログラム文を「外部のテキストファイルに保存すること・そこからプログラム文を読み込むこと」です。内部保存領域のデータ構造（図 3.1）とテキストファイルのデータ構造とは異なります。その変換のコーディングが必要です。

3.2　ラインエディタの機能

行番号とラベルとは利用目的が別であること

かなり昔のプログラミング作業は、テレタイプライタのような入出力装置を使ってテキスト原稿を編集しましたので、1 行単位のデータを扱うラインエディタを利用しました。これは原稿の行単位に番号を振って、編集補助に使う方法です。そのため、Fortran や Cobol では最初から行番号込みのソースコード書式仕様が使われていました。ラベルの方はプログラム文の目印ですので、必ずしも整数である必要はありません。Fortran ではラベルに整数番号を使いますが、行番号とは別仕様ですし、すべての行にラベルは付けません。その数字も、必ずしも昇順に付ける必要がありません。マイコンの Basic もそのような流れで設計されていましたが、すべての行に昇順の整数番号を付け、ラベルとしても利用しました。行番号があると、プログラム文を解説するときに行位置を指定できて便利です。最近のプログラミング言語はそれがありませんので、デバッグなどのときに案外不便になりました。Plain_Basic は、プログラミング教育も兼ねていますので、すべての行に古典的なラベルを付ける方式を採用しています。ラベルは、プログラムの実行時に GOTO 文などで実行位置を変更するときの目印に利用します。

Plain_Basic はラインエディタの機能を持つこと

Plain_basic に組み込んであるエディタは、古典的なラインエディタです。これは、テキストを行(line)単位で扱い、行に昇順の整数番号を付け、番号の若い順に並べます。行番号は一連番号ではなく、10,20,30…のように付けておき、途中行にプログラム文を追加挿入する余裕を持たせます。若い行番号のプログラム文を挿入するときは、その位置以降を後に押し出して場所を空けます。また、或る行番号のプログラム文を削除すると、その場所を詰めるように後のプログラム文を繰り上げます。この処理は **pbEditLine()** で行います。ユーザレベルでの編集用コマンドは「**AUTO, EXIT, LIST, DELETE, RENUM**」および **CLS** です。EXIT を除き、pb の接頭字を付けた関数 pbAuto(), pbList(), pbDelete(), pbRenum() などが使われます。

表 3.1　Plain_Basic のラインエディタ関係の関数

関数名	解　　　説
pbAuto	行番号を昇順に自動発行し、プログラム編集モードに移行します
pbList	テキストモニタ上に番号付きでプログラム文をリストします。
pbDelete	指定した行番号のプログラム文を削除します。
pbRenum	行番号を付替えます
pbEditLine	指定した行番号でプログラム文の追加・削除をします。
pbSortLineNr	テキストを納めたバッファーから指定した行番号位置のアドレスを求めます。
pbSortAddress	上と殆ど同じですが、編集のためのテキスト挿入位置のアドレスを求めます。

プログラム文の編集は行単位で処理する

一般のテキストエディタやワードプロセッサは、テキストの単位は段落(paragraph)です。長い文字並びの表示は、ディスプレイの右端、語の切れ目で自動的に折り返されて複数行にまたがります。これを自動改行(wordwrap)と言います。改行コードは、段落の切れ目で入れますが、このときを強制改行と言います。このテキスト文を編集するとき、文字並びの削除や挿入は、その場所以降の文字数が変りますので、複数行にまたがる文字並びは、次の行への繰り越しか繰り上げが行われ、自動折り返しの位置も変ります。Plain_Basic のプログラム文編集は、テキストウインドウ上の編集作業で行うのではなく、一行(line)単位でテキスト入力枠から行います。このためラインエディタと言います。一行の文字並びで、途中の文字を修正したいときでも、まるまる一行分をテキスト入力枠から入力します。これがテキストエディタとラインエディタとの違いです。これは、編集作業では不便です。Plain_Basic では便利な編集機能を使うことができます。それはテキストウインドウにリストを出力しておいて、そこで修正した一行分の文字並びをコピーして、テキスト入力枠に**貼り付け**る方法を使います。

プログラム編集モードを使うこと

プログラム文を一行単位で入力するときには、文頭に整数番号を付けます。プログラム文の行数が多いとき、番号を割り付ける作業を自動化して入力することができて、その状態をプログラム編集モードと言います。直接モードでコマンド AUTO を入力します。引き数を省略するとデフォルトで初期値が 10、増分が 10 に設定されます。テキスト入力枠のプロンプトラベルが「Console ==>」から「Auto 10 =>」に変りますので、英字で始まるプログラム文をテキスト入力枠から入力して改行します。エコーのリストは、この番号付きでテキストウインドウに表示され、プロンプトの番号は増えて 20 になって次のプログラム文の入力を待ちます。このように作業を進め、プログラム編集モードから抜けるときは EXIT と入力します。エコーのリストには行番号付きで EXIT が表示されますが、これはプログラム文には繰り込まれません。プロンプトのラベルは、元に戻ります。入力したプログラム文の確認は LIST コマンドで行います。

テキストウインドウ自体は独立したテキストエディタである

Plain_Basic は、処理結果の出力用に二つの子ウインドウを持ちます。この内、テキストウインドウは、処理の通知情報と PRINT 文の書き出しに使います。画面は編集機能を持つ RichTextEdit を載せてありますので、このままで独立したテキストエディタとしても使うことができます。ここで編集するテキストは、Plain_Basic の入力とは一切関係しません。ただし、この中の文字列一行をクリップボード経由でコピーして、テキスト入力枠に貼り付ける機能に使うことができます。

テキストファイルのデータ構造とは異なること

Windows システムには、単純なテキストエディタ NotePad があります。データは、印刷表示の補助に使う改行コード、タブコードなどが区切り記号として入った一続きの連続した文字並びです。Plain_Basic の内部で使うプログラム文書は、改行コードを使わず、1 行の区切りは、値 0 のバイト('¥0')を使います。これは C/C++の言語仕様からくる制限です。したがって、プログラム文をテキストウインドウに表示する(LIST)、外部のテキストファイルに書き込む(SAVE)、またテキストファイルからプログラム文を読み出す(LOAD)とき、行末コードの書き換えを必要とします。バイト並び単位での文字の挿入と削除は、注目する文字位置を起点として、長い文字並びの途中から最後までの全体を移動させます。行番号をキーにして文字列を探すときは、データの頭から行の文字数分ごとに文字位置を飛ばしながら、行番号位置を求めます。この処理は、プログラム実行モードのとき、プログラム文を取り出すときにも利用されます。行番号は正の整数の昇順で割り当て、行番号 0 は終端コードとして使います。このプログラム文を外部のテキストファイルに保存するときは、２バイトのバイナリ値の行番号をデコードして数字に直し、文字数のデータを除き、行末のバイト('¥0')を改行復帰コード(0D0A)に直します。

3.3　行番号の更新

行番号とラベルとは利用目的が別であること

プログラムの大きさや複雑さは、ソースコード全体文書の行数で言うのが普通です。文書量が多くなると、全体の見通しが悪くなり易いので、高級プログラミング言語ではさまざまな作文技法が応用されています。行番号は、プログラム文の編集作業のとき、部分的に見ているプログラム文の相対的な位置を知る上で助けになります。ラベルは、主に、プログラムの実行位置を変えたいとき(GOTO 文、または GOSUB 文)の目印に使います。ラベルは、必ずしも数字である必要はないのですが、行番号と兼用してあると、どこに制御が移るかが分かり易くなります。

行番号の付け直し機能がある

プログラム文の行の挿入と削除は、行番号の重複がないように調整します。行番号を連続数でなく、10 刻みで付けておくのは、途中行の挿入作業を助けるためです。この状態では９行までの挿入ができますが、それ以上の追加ができませんので、行番号を新しく 10 刻みに付け直しをします。これが RENUM コマンドです。行の追加や削除をすると行番号の間隔が不揃いになりますので、行番号間隔を揃え、全体の行数が簡単に分かるようにするためにも RENUM コマンドを使います。

ラベルの付け直しが厄介であること

プログラムの実行順序をGOTO文・GOSUB文で変更するとき、飛び込み先に目印、つまりラベルが必要です。Plain_Basic では、行番号をラベルと兼用しますので、編集作業のときに厄介な問題が起こります。それは、飛び込み先の行番号を未だ決めていなくて、仮の行番号を当ててラベルとしておき、全体が完成したときに見直して、ラベルを正しい行番号に付替える手間が掛かることです。そうして調整しても、RENUM コマンドを実行すると行番号が変りますので、ラベルの方も連動して新しい行番号に付替えなければなりません。これはPlain_Basic本体のコーディングで手間が掛かるところです。高級プログラミング言語では、行番号を使いませんので、目印はラベルだけであり、それも英数字を使います。この方式の進化したのがサブルーチンとして独立させたモジュールです。高級プログラミング言語は、コンパイルとリンク作業を経て実行形式のプログラムを完成させるのですが、このとき、変数名・関数名・サブルーチン名のアドレスマップを作成しています。Plain_Basic の場合は、RENUM コマンドを実行するときにラベルのマップを作成して、GOTO文・GOSUB文で参照する行番号を変更します。

行番号の新旧対照表の作成

コマンドRENUMは、関数pbRenum()を呼んで行番号の付替えを行います。この処理の始めと終りに、関数 pbListLines()を呼び、新旧二つの行番号の**対照表**を作成します。この表は、作業領域として renumLabel[]を使いますが、変数と配列の内部保存領域Buf[]と共用するように名前v.をつけてunion宣言で準備しています。RENUMの処理はプログラム文の編集ですので、変数と配列を使わないからです。これを元に、pbRenum2()に制御を渡して、GOTO,GOSUB,RESTOREのキーワードで指定されたラベルを付替えます。ただし、RESTORE のキーワードは、デフォルトで行番号無しの指定もあります。この処理は結構入り組んでいますので、以前のNUCE_BASICでは手を抜いていた部分です。

行番号の差し替え

PBRENUM2()は、プログラム文の実行と同じように、先頭行からプログラム文を一行単位で読み出して、一文字単位で作業用のバッファーprinttBuffにコピーします。このとき、英字の’g’または’r’が現われたら、前記のキーワード文字列と一致するか否かを検査します。一致すれば、そのキーワードに続く数字並びを２バイト整数にエンコードします。これは古い行番号ですので、対照表を参照して新行番号を求めます。さらに、これをデコードして数字並びに直し、printtBuffに転送します。このようにして一行の修正ができれば、この行をpbEditLine()で差し換えします。ただし行番号の変更が無ければ、差し換えなしに、次のプログラム文一行の読み出しを行います。

3.4 プログラム編集の関数

表 3.2　pbAuto()のソースコード

```
void pbAuto(void)                         /* AUTO labelling set */
{                                                // 2006-02-17
     bool b;
     pbText1(',');
     if (bErrorFlag) return;
     if (NrOfOperand >2) {pbErrList(ErrString02); return;};
     iStartNo = NumData[0];
     iIncreNo = NumData[1];
     if (iStartNo == 0) iStartNo = 10;
     if (iIncreNo == 0) iIncreNo = 10;
     iCurrentLabel = iStartNo;
     iAutokey = 1;
}
```

表 3.3　pbList()のソースコード

```
void pbList(void)              /* LIST N1 - N2 */
{
	Char * lineAddress;
	Word len, lineNumber, iFrom, iTo;
	String Text1;
//
	pbText1('-');
	switch (NrOfOperand){
		case 0:
			iFrom = 1; iTo = 32767; break;
		case 1:
			iFrom = NumData[0]; iTo = iFrom; break;
		case 2:
			iFrom = NumData[0]; iTo = NumData[1]; break;
		default:
			pbErrList(ErrString02); return;
	};
	lineAddress = pbSortAddress( iFrom );
	for(;;) {
		lineNumber = *((Word*)lineAddress);
		if( lineNumber == 0 ) break;
		if( lineNumber > iTo ) break;
		Text1 =Format("%d", ARRAYOFCONST(((int)lineNumber)));
		len = *((Word*)(lineAddress + 2)) ;
		memcpy(printBuff,lineAddress + 4,len - 4);
		Write6(Text1 + ' ' + (String) printBuff);
		lineAddress += len;
	};
}
```

表 3.4　pbDelete()のソースコード

```
void pbDelete(void)            /* Delete N1 - N2 */
{
	Char * lineAddress1, *lineAddress2;
	Word len, sizeOfProgram, iFrom, iTo;
//
	pbText1('-');
	switch (NrOfOperand){
		case 0:                       /* delete all */
			progBuff[0] = 0;
			progBuff[1] = 0;
			progBuff[2] = 0;
			progBuff[3] = 0;
			progBuff[4] = '¥0';
			codeTop = 5;
			return;
		case 1:                       /* delete a line */
			iFrom = NumData[0]; iTo = iFrom; break;
		case 2:                       /* delete iFrom - iTo */
			iFrom = NumData[0]; iTo = NumData[1];
			if (iTo == 0) iTo = 32767;
			break;
		default:
			pbErrList(ErrString02); return;
	};
/* DELETE iFrom - iTo: */
	lineAddress1 = pbSortAddress( iFrom );
	lineAddress2 = pbSortAddress( iTo );
	lineAddress2 += *((Word *)(lineAddress2+2));
	len =lineAddress2 - lineAddress1;
	sizeOfProgram = codeTop-(lineAddress2-progBuff);
	memmove( lineAddress1, lineAddress2, sizeOfProgram);
	codeTop -= len;
}
```

表 3.5　pbRenum()のソースコード

```
void pbRenum(void)                    /* Renumbering */
{                                             // 2006-03-30
     Char *lineAddress;
//   Char *lineAddress2;
     bool b;
     Word iStartNo, iFrom, iIncreNo;
     Word lineNumber, lineNumberOld, len;
     pbText1(',');
     if (bErrorFlag) return;
     if (NrOfOperand >3) {pbErrList(ErrString02); return;};
     iStartNo=10; iFrom=1; iIncreNo=10;   // default setting
     switch (NrOfOperand){
          case 0:
               break;
          case 1:
               iStartNo=NumData[0];break;
          case 2:
               iStartNo=NumData[0]; iFrom=NumData[1];break;
          case 3:
               iStartNo=NumData[0]; iFrom=NumData[1]; iIncreNo=NumData[2];
               break;
          default:
               pbErrList(ErrString02); return;
     };
     if (iStartNo==0) iStartNo=10;
     if (iFrom==0) iFrom=1;
     if (iIncreNo==0) iIncreNo=10;
     lineAddress = pbSortAddress(iFrom);
     lineNumber=*((Word*)lineAddress);
     if (lineNumber==0) return;
     lineNumber=iStartNo;
//
     for (;;){
          *((Word*) lineAddress)=lineNumber;
          lineNumber += iIncreNo;
          len = *((Word*) (lineAddress+2));
          lineAddress += len;
          lineNumberOld = *((Word*)lineAddress);
          if (lineNumberOld ==0) break;
     };
}
```

表 3.6　pbEditLine のソースコード

```
//=====================================================================
/* editor for local Basic statements in ProgBuff */
/*********************************************************************
*    progBuff[] is the Buffer area to save local basic program statements.
*    A statement is buile as a series of byte_structures;
*         lineNumber(Word) in 2 bytes              +
*         length_of_line(Word) in 2 bytes      +
*         character strings with '¥0' at end.
*   Length of a statement has nByte
*********************************************************************/
void pbEditLine(const Word newLineNumber)
{                                                          // 2006-02-23
     Char * targetAddress, * nextAddress;
     Char ch;
     Word oldLineNumber;
     Word len, sizeOfProgram;
     targetAddress = pbSortAddress( newLineNumber );
     oldLineNumber = *((Word*)targetAddress);
     len = *((Word*)(targetAddress+2));
     nextAddress = targetAddress + len ;
/* removing the line */
     ch = *execPointer;
     if( ch == EOL) {
          if( oldLineNumber == 0 || oldLineNumber != newLineNumber ) {
               pbErrList2( newLineNumber ); return;};    // line number not found
          sizeOfProgram = codeTop-(nextAddress-progBuff);
          memmove( targetAddress, nextAddress, sizeOfProgram);
          codeTop -= len;
          resumePointer=0;
          return;
     };
/* insert new line */
     if( oldLineNumber == 0 || oldLineNumber != newLineNumber ) {
          if( codeTop + nByte + 5>= SIZE_FOR_PROGAREA ) {
               pbErrList(ErrString12); return;};    // program area overflow
          sizeOfProgram = codeTop - (targetAddress-progBuff) ;
          memmove( targetAddress + nByte + 5, targetAddress, sizeOfProgram);
          codeTop += nByte + 5;
/* replace line */
     } else {
          int delta;
          delta = nByte + 5 - len;
          if( codeTop + delta >= SIZE_FOR_PROGAREA) {
               pbErrList(ErrString12); return;};    // program area overflow
          sizeOfProgram = codeTop - (nextAddress - progBuff);
          memmove( nextAddress + delta, nextAddress, sizeOfProgram);
          codeTop += delta;
     };
//
     *((Word*)targetAddress) = newLineNumber;  // copy line Number
     *((Word*)(targetAddress + 2)) = nByte+5;  // copy text length
     memmove(targetAddress+4, execPointer, nByte ); // text string
     *(targetAddress + nByte + 4) = '¥0';       // add EOL
     resumePointer=NULL;                        // can not continue
}
```

表 3.7　pbSortLineNr のソースコード

```
/* search address of given line number */
Char * pbSortLineNr( Word lineNumber )
{
	Char * ptr;
	Word textLabel;
	ptr = progBuff;
	for(;;) {
		textLabel = *((Word *)ptr);
		if( textLabel == 0 ) { return NULL; };		// not found
		if( lineNumber == textLabel ) { return ptr;	};	// found
		ptr += *((Word *)(ptr+2));
	};
}
```

表 3.8　pbSortAddress のソースコード

```
/* find Label address for List or new Program Line*/
Char * pbSortAddress( Word lineNumber )
{
	Char * ptr;
	Word textLabel;
	ptr = progBuff;
	for(;;) {
		textLabel = *((Word *)ptr);
		if( textLabel == 0 )			{ return ptr; };
		if( lineNumber <= textLabel )	{ return ptr; };
		ptr += *((Word *)(ptr + 2));
	};
}
```

4. 数式の解読処理

4.1 数式の計算が面倒である理由

計算を実行するには幾つもの手順を経ること

プログラミング言語を作成するとき、数式を解読して数値計算を実行させる処理の設計は相当な難題です。それは次のような理由からです。

① 数値計算を表す代数式は、文字で表現されている。
② この表現は、必ずしも実際の計算順序ではない。
③ したがって、式を解読して、実際の計算順序に並べ変える必要がある。
④ 文字で表現されている数字や変数を内部で使う数表現に直す（エンコード)。
⑤ 計算された結果を変数に代入する。
⑥ モニタやプリンタに表示するためには逆変換処理（デコード）が必要である。

式を表す文字並びは必ずしも数値計算の順序ではない

具体的な説明をするため、例として、A=B*(C+D)のような単純な代数式を考えます。この計算では、あらかじめ右辺の変数 B,C,D を準備し、それに数値が入っていなければなりません。左辺の変数 A は、未だ宣言されていない場合もあります。これが①の課題です。式の処理の前に、例えば、B=3, C=4, D=5 のような代入文の処理で、実体の宣言と中身が決まっていなければなりません。つまり、変数の宣言方法と数の保存方法を踏まえなければなりません。これは第 2 章で説明しました。変数名を使わなくて、直接数字を使って、A=3*(4+5)の表現も処理できなければなりません。このときには、内部で仮の変数領域を準備します。これらの作業領域の扱いは、後で説明する<u>リストセル</u>で行ないます。

計算手順を文字や記号で表す学問が代数学であること

単純な代入を表す文字並びの表現、例えば、Y=X の形について説明します。代数学では、Y=X と、逆に書いた X=Y とは同じ意味であって等式と言います。しかしプログラム文では意味が違います。この形を<u>代入文(assignment)</u>と言います。これは右辺の値を左辺に転送します。数の流れで見れば右から左であって、文字表現とは逆順です。Y=X の形は、英語の構文からくる表現であって、例えば「Y is equal to X」を記号化したものです。日本語では、「X を Y に代入する」言い方が自然です。この区別があるため、式 Y=X において、Y を<u>左辺値</u>(lv:left value)、X を<u>右辺値</u>(rv:right value)と言います。関数を使った表現、例えば、A=SIN(X)は許されますが、関数の方を左辺値に使う SIN(X)=A のような表現は許されません。これが②の課題です。

式文と代入文の区別があること

元の例題 A=B*(C+D)の形は、イコール記号を挿みますので、全体は代入文ですが、右辺は演算子を含む表現です。右辺だけを扱うとき、これを<u>式文(expression)</u>または単に<u>式</u>と言って独立に考えます。この場合でも、文字並びと計算順序は同じではありません。計算は、括弧の中(C+D)を先に求めます。これを「括弧内の計算は演算の<u>優先順位</u>が高い」と言います。演算子に注目すると、A+B-C*D のような式では掛算部分 C*D を先に計算する約束とします。これを演算子の優先順位と言います。

括弧を想定して手順を分解すること

コンピュータを使う数値計算においても、内部の計算は電卓の操作順序と同じように組み立てます。計算は、二つの数値間で行ないます。電卓を使うときは、途中の計算をメモなどに記録しながら進めます。Plain_Basic の計算では、リストセルをメモ代わりに使います。数の種類が減り、式の形が変って行き、最後に一つの数値が答として残ります。手順前後を間違えないようにする最も単純な方法は、括弧を使うことです。例えば、A=B*(C+D)の形は括弧を使って、下のように書き変えながら計算を実行することです。括弧を考えることが、この後で説明するリストセルの使い方の原理です。

$$\left.\begin{array}{l}(A)=((B)*(C+D))\\(A)=((B)*(C'))\\(A)=(B*C')\\(A)=(B')\\(A')\end{array}\right\}\qquad 式\ 4.1$$

リストセルを使うデータ管理

上で説明した括弧を使う式を模式的に表す方法として、図 4.1 に示すような数の入れ物を考えます。これを(<u>リストセル</u>：list cell)と呼びます。この個数は、上で説明した括弧の対の数とほぼ同じになります。括弧は、入れ子構造の表現ですが、それを演算子によってリンク（結合）した構造に直します。A=B*(C+D)の計算をするには、変数 A を一旦準備しておき、右辺に移って括弧の中の C+D を先に計算し、それと B との積を求めてから、最初に戻って変数 A に代入します。この計算順序は、式の表現順とは違って、右から解決していることに注意します。この全体が説明項目③です。これには、図に直して理解することと、計算順序の組み立てと解決（右から）とを含みます。計算は原則として左から右の順に処理します。このとき、後のリストセルと前のリストセルとで演算をし、その結果を前のリストセルのデータと差し換え、後ろのリストセルを消去します。したがって、リストセルは全体としては後ろから（右から）消去するように処理します。右辺の計算が済めば、結果の入った一つのリストセルが残りますので、それを左辺のリストセルの変数アドレスに代入します。

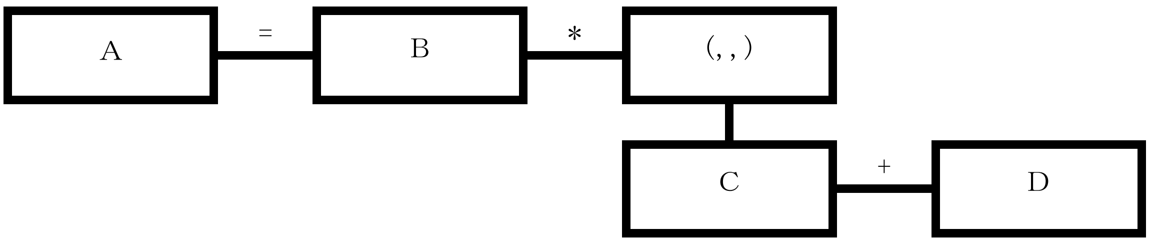

図 4.1　セルを使って数式の関係を模式的に表した図

4.2　リストセルのプログラミング

リストセルのデータ構造

数値計算処理の道具には、Plain_Basic ではリストセルを準備し、それを利用するプログラミングを設計します。リストセルは、演算子の情報を媒介としたリンク構造に組み立て、数のデータを格納すると同時に、変数などではその型やアドレスなどの情報を持たせます。下の表は、リストセル制御用の変数と、リストセルの構造体の構成を示したものです。単位のリストセルは、物理的には、「2 バイト長の整数 10 個・名前保存用 16 バイトの文字領域・作業用変数保存領域内での変数位置を表す 4 バイト整数・倍精度型データ領域 8 バイト×2」(ただし、ここでは幾何言語に拡張することを含みにして倍精度数のデータを 12 個保存する仕様になっています)、それに加えて、FOR-NEXT,GOSUB-RETURN の制御用に二つのポインタ用変数を持たせました。

表 4.1　リストセルの制御用変数とリストセル構造体

```
WORD stackMaxSize;              // = STACK_MAX,  current size is 30
WORD stackNextCell;
WORD stackRemainder;
WORD stackMaxUsed;
struct {
     WORD iMother;        // 1 left side cell
     WORD iDaughter;      // 2 right side cell
     WORD iPar;           // 3 pointer to parenthesis data
     WORD iNextPar;       // 4
     WORD iCode;          // 5 function code etc.
     WORD iSon;           // 6 number of arguments; 0=single Element
     WORD iSenior;        // 7
     WORD iJunior;        // 8
     WORD iOpeCode;            // 9 operation code for +-*/ etc.
     WORD icType;         // 10 INTEGER, DOUBLE, STRING etc.
     Char elementName[16];           // 11 name to a variable or an array
     int elementAddress;       // 12 relative address to a variable
     Char * returnPtr;         // FOR, GOSUB return address
     Char * NextProgPtr;       // save NEXT address
     double dValue[12];        // all variables are managed in double
} ListCell[STACK_MAX];
```

構造体にすると型の違うデータの集合を管理できる

ワードマシンを前提としたFortranのプログラミングでは、リストセル用のデータ領域を準備するために２次元の配列を使いました。このリストセル単位に、型の違うデータの集合をまとめるには面倒な方法を使いました。C/C++では、データ集合の設計を構造体(struct)で表すことができますので、扱いが容易になりました。リンク構造のようなセル相互の参照は配列番号を相対アドレスのポインタとして使います。そうすると、配列の先頭番号は０ですが、相対アドレスとして使うときには、０はアドレスが無いと解釈しますので、実際には配列番号１から利用します。そして、配列番号０は、リストセルの初期値を保存しておいて、新しくリストセルを利用するときには、この先頭のリストセルをコピーして初期化します。

リンクの構成方法

幾つかのリストセル相互の関係を表すため、他のリストセルの番号を保存し、それを手掛りにしてリストセルを繋ぎます。この番号もポインタですが、C のポインタよりも単純に使います。標準的なリンク構造は鎖状に繋ぐもので、自分の左と右のリストセルの番号をポインタとして持ちます。その呼び名としてMotherとDaughterを使いました。図4.1で、Ｂと(,,)のリンク関係が母子関係です。括弧を使う数式は、括弧を仮の親セル(Parent)として作成し、その下に長男が付属するような縦の入れ子方式の父子関係のリンクにします。括弧の英語がparenthesisであることも考慮した名前です。この例は、図4.1 において、(,,)とＣとの上下のリンク関係をつけます。配列名・関数・サブルーチンの括弧の中は複数の引き数をコンマ区切りの引き数で持ちます。その関係は、親の下の長男から次男・三男…の順に左右にリンクを設定します。図4.1において、ＣとＤがこの構造を示した例です。こちらも鎖状ですが、Senior,Junior の名称にしました。リストセルに保存するデータには種々あります。数値の場合、変数名・データ型・アドレス・演算子コード・その数値のコピー、などを保存します。

ランダムにリストセルを利用すること

リストセルは、種々の情報の入れ物です。リストセル構造体は、配列で宣言して準備します。作業に使うセルを一つ予約する関数を pbCellTake()、使用が済んだセルを戻す関数を pbCellBack()とし、再利用ができるような後入れ先出し(last in, first out:LIFO)のスタック管理方式で次の利用に備えます。リストセルは、式の解読と計算だけでなく、FOR-NEXT、GOSUB-RETURNでの制御にも使います。したがって、リストセルを配列として準備してあっても、参照順はランダムです。式の表現が込み入ってくると多くのリストセルが必要になりますが、Plain_Basic では差し当たり30セルを準備しました。プログラムを初期化(NEWコマンド)すると、配列１番から利用するように初期化します。

表4.2　リストセルの初期化で行なわせる処理のソースコード（pbNewの一部分）

```
/* List Cell initialize */
    ListCell[0].iMother = 0;
    ListCell[0].iDaughter = 0;
    ListCell[0].iPar = 0;
    ListCell[0].iNextPar = 0;
    ListCell[0].iCode = 0;
    ListCell[0].iSon = 0;
    ListCell[0].iSenior = 0;
    ListCell[0].iJunior = 0;
    ListCell[0].icType = 0;
    ListCell[0].elementName[0] = '¥0';
    ListCell[0].elementAddress = 0;
    ListCell[0].returnPtr=NULL;
    ListCell[0].nextProgPtr =NULL;
    for (int i=0;i<12;i++){ListCell[0].dValue[i]=0;};
    stackMaxSize = STACK_MAX - 1;
    stackNextCell = 1;
    stackRemainder = stackMaxSize ;
    stackMaxUsed = 1;
```

倉庫からの取りだしと食器棚での一時保存法

リストセルは、お料理のお皿を準備するのと似たような管理をします。大量のリストセルは、最初、言わば、倉庫に準備してあります。必要な量のお皿は倉庫から出して使いますが、利用が済めば、倉庫ではなく、食器棚のような臨時のスタックに積み上げます。この処理が pbCellback() です。次に使うセルは、後入れ先出しで食器棚から取り出して利用します。食器棚にお皿が無くなったら倉庫から補充します。古いリストセルは、お皿を重ねるようにスタックに積み上げるのですが、このとき、一番上に載せるリストセルのアドレスが次に使うセルとして登録され、そのすぐ下にあるリストセルのアドレスをそのリストセルに書き込んでおきます。iDaughter を、その番号保存の場所に使います。原理は多少複雑ですが、プログラムは、表 4.3 に示すように短いソースコードです。

表 4.3　リストセルの予約と返納処理 pbCellTake/Back のソースコード

```
WORD pbCellTake(void)
/*    reserve one list_cell and clear */
{
     WORD unused, ibk;
     if (stackRemainder <=0 ) {pbErrList(ErrString16); return 0;};
     ibk = stackNextCell;
     stackRemainder--;
     unused = stackMaxSize - stackRemainder;
     if ( stackMaxUsed == unused ) {
          ibk = stackMaxUsed;
          stackMaxUsed++;
     }else{
          stackNextCell = ListCell[ibk].iDaughter;
     };
     ListCell[ibk] = ListCell[0];
     return ibk;
}
void pbCellBack(WORD ibk)
/*   return the list cell to the stack  */
{
     if(ibk == 0) return;
     ListCell[ibk].iDaughter = stackNextCell;
     stackNextCell = ibk;
     stackRemainder++;
}
```

4.3　数式文字列の解読

式の表し方には一定の規則があること

文字の並びで数式を表すことは種々の場面で現われます。代入文は A=X…の形で始まり、最初の A は必ず変数名か配列名ですので英字で始まります。配列名の場合にはカギ括弧 ’[’ が続き、例えば二次元配列で A[U,V]のような表現です。U,V も、それぞれ数式(expression)です。数字で表されていても、それは文字式であって、エンコードして数データに直します。上の例では X,U,V は数式の文字並びで表されますが、これを数値データに直すのが数式解析ルーチンです。その文字並びの表し方に約束があります。数式であることを特徴づけるのは下のような性質です。

①　英字で始まる場合、変数名か配列名か関数名である。配列名は[、関数名は(が続く。
②　数字で始まる場合は具体的な数を表すときである。指数記号は英字の E が続く。
③　演算子記号で始まるのは＋または－である。小数点単独は、式文の始めには使わない。
④　記号は左括弧 ’(‘ で始まる場合だけが数式に表れること。

関数 pbSet0()は文字列から、具体的な数字で利用できるように変換する処理です。幾つかの関数を従えた、かなり複雑なソースコードです。

式解読の基本形は代入文である

プログラム文が数式であることをPlain_Basicに知らせる最も標準の書式が代入文です。このときに使われる約束と解読の手順は次のようです。

① 文は英字で始まり、それは変数名か配列名のはずです。

② 配列名ならば、左カギ括弧 ’[’ が続きます。そうでなければエラーです。

差し当たり、説明を急ぐため、その英字名は単純変数であるとしておきます。

名前の解読はpbName()で行ないます。文字のポインタ位置は、英字名の次です。

名前の iKwrdCode は5または6になっています。それを判定して、

③ 処理は、pbLet1()に入ります（表4.4)。

変数名であることが分かっていますので、作業用のリストセルを一つ予約します。

セルの予約はpbCellTake()を使います。

このセル番号を引き数として、数式処理のpbSet0()を呼びます。

このセルには、その変数の型やアドレスなど、左辺値が記録されて帰ってきます。

文字のポインタは英字名の次の位置にあって、スペースが続いていることもあります

次の有効文字がイコール記号でなければエラーです。

ポインタを一つ進めてイコール記号の位置をスキップします。

ここで、右辺値が式文で続くと仮定できますので、別のリストセルを予約します。

このセル番号を引き数として、数式処理のpbSet0()を再度呼びます。

エラーが無ければ、このリストセルに右辺値が設定されて帰ってきます。

二つのリストセルを使って、左辺値を右辺値に代入します。

代入が済めば、二つの作業用セルをpbCellBack()で返却します。

④ 文字位置のポインタは、代入文の直ぐ後ろに在るはずですが、処理はこれで一区切りです。

明示的に代入文を解読するときLETコマンドを使うことがある

文字並びで表される代数式は A=X の形式です。これは、他のプログラム文がキーワードで始まるのと比較すると例外的な文法です。そこで、明示的に、代入文であることを示すキーワードにLETを付けて、「LET A=X」のように書く仕様もあります。普通は LET のキーワードを使う必要がありませんが、配列の成分に順々にデータを代入するとき、少し鬱陶しい手続きになります。そこで、例えば A[3]の配列にデータを入力させるとき、引き数を省略し、イコール記号の後ろに、必要個数の数値のリストを並べる方法を考えます。例えば、DIM A[3]と宣言しておいて、

```
LET A = 4, 5, 6
```

のように使います。この形式は、一種の入力文になっています。そこで、この拡張として、複数の変数名を使う方法を考えることができます。その形式は、

```
LET X,Y,Z = 3, 4, 5
```

のように表す書式です。これは左辺値が複数、右辺値も複数です。ここで、LET を別のキーワードに変え、左辺値の文と右辺値の文とを別々に準備すると、ファイルやキーボードからの入力文になります。また、出力文PRINTも、形式として代入文と親戚関係になります。例えば

- ・キーボードからデータ入力　INPUT X,Y,Z ⇔ キー入力　3, 4, 5
- ・ファイルからデータ入力　INPUT #1 X,Y,Z ⇔ ファイルデータ　3, 4, 5
- ・内部ファイルの読み込み　READ X,Y,Z ⇔ DATA 3, 4, 5
- ・プリンタへの出力文　PRINT X,Y,Z　（表4.5のリスト参照）

なお、Plain_BasicではINPUT文と、PRINT #文を使いません。内部ファイルの扱いは次章で説明します。

表 4.4　代入文を解読する関数 pbLet1 のソースコード

```
/*========================== 840702 === FORTRAN 77S ======= BLET1
      SUBROUTINE BLET1(TEXT,ICTRL)
  EQUATION ANALYZER.
*/
void pbLet1(void)
{
     double DIS,AX,AY;
     Word ist0, ist, ist1, icType, idType,nCnt,mCnt,ntnv,ntnd;
     Word iMod;
     Char ch;
     int iPoint;
/* set l_value of equation */
     if (bErrorFlag) return;
     icType = iVarType;
     ist0 = pbCellTake();
     ist = ist0;
     iMod = 0;                      // array evaluation Mode
     pbSet0(ist, iMod); if (bErrorFlag) return;
/* shall be squal sign '=' */
     ch = pbCharRead();
     if (ch != '=') {pbErrList(ErrString02); return;};
/* set r_value of equation */
     execPointer++;
     ist1 = pbCellTake();
     iKwrdCode = 0;
     iMod = 1;                      // arithmetical calculation Mode
     pbSet0(ist1,iMod); if (bErrorFlag) return;
     icType = ListCell[ist0].icType;
     idType = ListCell[ist1].icType;
     iPoint = ListCell[ist0].elementAddress;
/* assignment */
     switch (icType){
          /* 1: ---- ASSIGNMENT EXPRESSION SCALOR=SOMETHING */
          case INTEGER:  // fall through
          case DOUBLE:
               switch (idType){
                    case INTEGER:  // fall through
                    case DOUBLE:   // fall through
                         vBuff[iPoint] = ListCell[ist1].dValue[0];
                         break;
                    default:
                         pbErrList(ErrString13); return;
               };
               break;
          case STRING:
               switch (idType){
                    case STRING:
                         vBuff[iPoint] = ListCell[ist1].dValue[0];
                         vBuff[iPoint+1] = ListCell[ist1].dValue[1];
                         break;
                    default:
                         pbErrList(ErrString13); return;
               };
               break;
     };
/* return ListCells */
     pbCellBack(ist0);
     pbCellBack(ist1);
}
```

表 4.5　コマンド PRINT を扱うソースコード

```
/*=========================== 840702 === FORTRAN 77S ======= BPRNT
      SUBROUTINE BPRNT(TEXT,ICTRL)
  PRINT CONTROL.
*/
void pbPrint(void)
{                                      // 2006-03-27
    Word ist, iMod, icType, kDim;
    int nCnt, iPoint;
    double dValue;
    const Char Tab = 9;
    const Char Space = ' ';
    Char ch, characterBuff[128], *src, *dst;
    String textForPrinter, text;
    ch=pbCharRead();
    execPointer--;
/* loop for printer */
    for (;;) {
        ch=pbCharNext();       // extract next character
        iKwrdCode = 0;
        switch (cbtype){
            case EOS:
                //execPointer++;   // fall through
            case EOL:          // flush out and clear
                Write6((String) textForPrinter);
                characterBuff[0] = '¥0';
                textForPrinter = "";
                return;
            case SYMBOL:
                switch (ch) {
                    case ' ': continue;     // skip spaces
                    case '+':     // fall through
                    case '-':// fall through
                    case '(': break;   // to equation solver
                    case ';': continue;//{textForPrinter += Space; continue;};
                    case ',': {textForPrinter += Tab; continue;};
                    case '"':     /* string begins with quote " */
                        dst = characterBuff;
                        execPointer++;
                        for (;;){
                            ch = *execPointer++;
                            if (ch == '"') break;
                            if ((ch == '¥0') || (ch == ':')) {
                                pbErrList(ErrString02); return;};
                            *dst++ = ch;
                        };
                        *dst ='¥0';
                        textForPrinter += (String)characterBuff;
                        continue;
                    default: pbErrList(ErrString02); return;
                };
            case DIGIT:
                break;
            case ALPHA:
                pbName(); if (bErrorFlag) return;
                if (iKwrdCode > 2) break;
                if (iKwrdID != KWD_ELSE) {pbErrList(ErrString02); return;};
                pbSkip(); continue;
```

```
        };
      /* LITERAL EQUATION; ENCODE AND PRINT FOR VARIABLE OR ARRAY. */
        ist = pbCellTake();    if (bErrorFlag) return;
        iMod = 1;                                        //  calculation
mode
        if ((iKwrdCode==6)&&(iKwrdNarg ==0))iMod =3;  // print array mode
        pbSet0(ist, iMod); if (bErrorFlag) return;
        nCnt = 1;
        icType = ListCell[ist].icType;
        iPoint = ListCell[ist].elementAddress;
        if (iMod == 3) {
            for (int i=0;i<iVarDim;i++) {nCnt *= iVarElementSize[i];};
        };
        for (int i=0;i<nCnt;i++) {      // send data to printer buffer
            switch (icType){
                case INTEGER:
                    if (iPoint==0 ){
                        text = Format(" %d",
                            ARRAYOFCONST(((int)ListCell[ist].dValue[i])));
                    }else{
                        text = Format(" %d",
                        ARRAYOFCONST(((int)vBuff[iPoint+i])));
                    };
                    break;
                case STRING:
                    UC.dValue[0]=vBuff[iPoint];
                    UC.dValue[1]=vBuff[iPoint+1];
                    text = (String) UC.ElementName;
                    break;
                default:
                    if (iPoint==0){
                        text = Format(" %g",

    ARRAYOFCONST(((double)ListCell[ist].dValue[i])));
                    }else{
                        text = Format(" %g",
                        ARRAYOFCONST(((double)vBuff[iPoint+i])));
                    };
            };
            textForPrinter += text; text = "";
        };
        pbCellBack(ist);
        if (iKwrdCode == 0) {execPointer--; continue; };
        if (iKwrdCode != 1) {pbErrList(ErrString02); return;};
        if (iKwrdID != KWD_ELSE) {pbErrList(ErrString02); return;};
        if (!pbSkip()) {pbErrList(ErrString02); return;};
        continue;
    };
}
```

4.4　数式のエンコード

数一つを納める入れ物がリストセル

数式(expression)は、複数の変数や数値を表す文字並びですが、この目的は、一つの結果数値を求める単位です。したがって、コマンドで複数の引き数を取る場合、コンマで区切られたリストの個々が一つの数解読対象です。数式一つごとに一つのリストセルを割り当て、演算モードを指定して pbSet0()に解析を委ねます。しかし、一単位の数式の中身は複数の変数や数値を含みますので、pbSet0()に制御が入ると、幾つかの追加リストセルを繋ぎ、その個々に変数や数値を納めます。その表し方の一例が**図 4.1** です。式の解読と演算は、**式 4.1** のように進み、追加リストセルは次々に返却され、最後に最初の引き数のリストセル一つが残り、結果数値が得られます。pbSet0()は、内部から幾つかの関数を呼びます。主な名前は pbSet?()です(?記号はワイルドカードの意味で 1,2...などと使い分けます)。また演算子で結合された数の計算は pbAcc?()を使います。この節はこの全体の解説です。数式解読のソースコードは大きいので、pbSet0()のソースコードだけを表 4.6 に示します。

pbSet0()のアルゴリズム解説__(1)

pbSet0()に制御が入るときは、前もって作業用のリストセルを一つ用意します。文字列解読対象が式であることが前提です。また、解析の目的をモード番号 **iMod** で指定します。これは４通りあります。

- 0:　引き数を持つ配列の具体的な位置（アドレス）を求めます（左辺値)。
- 1:　数式を解析して結果数値を持ち帰ります（右辺値)。
- 2:　DIM 文で宣言していない配列があったとき、デフォルト寸法で作成します（左辺値)。
- 3:　引き数を省略した配列名を使うときです。PRINT 文では右辺値、LET 文では左辺値扱いです。

PbSet0()に制御が入るとき、ポインタは、必ずしも解読対象文字列の先頭を指してはいません。コマンドの LET を使わなければ、代入文の左辺値は、英字名の解読で変数名か配列名かが分かってからですので、ポインタは英字名の次の文字を指しています。代入文はイコール記号の後ろが右辺値です。PRINT 文のリストは、右辺値の並びです。右辺値であることが条件であれば、その先頭文字が何であるかは未知の状態で pbSet0()に制御が入ることもあります。この区別は、名前のプロパティ **iKwrdCode** を見て判断します。英字を読んだ直後でなければ、iKwrdCode を 0 に設定しておきます。右辺値を解読する条件で PbSet0()に制御が入り、最初の文字が英字であれば、そこで英字名の解読を **pbName()**で行ないますので、解読条件が同じに揃います。

文字並びの全体が式文(expression)である条件は、式全体の文字並びの先頭文字種から言って三つあります。①　英字で始まる場合、左辺値を求める場合は変数名か配列名です。右辺値では関数名も許されます。配列名であれば'[',関数名であれば'(' が来ます。配列名の場合には、引き数なしで参照する使い方を Plain_Basic の仕様に含めてあります。LET コマンドの左辺値で使うときは、配列の成分数の数値を右辺値から取り込んで連続的に代入する目的に使います。PRINT 文のリストに使うときは右辺値扱いをし、配列の成分数の印刷に必要なデータ数が得られるように結果を持ち帰ります。英字名の場合には、プロパティ **iKwrdCode** の種別によって処理が異なりますので、**pbSet2()**に制御を渡します。これは次の項で解説します。

式全体の文字並びの先頭文字種が数字であるとき②は、具体的な数値ですので右辺値です。これは pbEncd0()に制御を渡して数値に直します。数字並びには指数記号 E を使うことがあります。内部的には単精度型の実数を使いませんが、英字 E に決めました。このときは、＋または－符号と整数が続きます。ただし＋符号は省くことができます。この全体は、スペースを挿みません。数字表現の区切りは、演算子記号か、スペース・コンマ・EOL・EOS などです。

先頭文字種が記号であるとき③も、右辺値の場合です。演算子記号は、符号を表す＋または－だけです。これは、リストセルの符号情報 **iOpeCode** に 0 または 1 を記録します。小数を表すとして、小数点で始める表記は使いません。括弧は、演算順位を明確にする目的がありますので、左括弧 '('で始める場合があります。括弧が現われたときは、新しいリストセルを追加して、括弧内の式を入れ子に構成します。図 4.1 の例を見て下さい。関数の引き数を表すときにも括弧を使いますが、その場合には英字名に直ぐ続けて左括弧がきますので、それを判断して演算用の括弧と区別します。

pbSet2()のアルゴリズム解説__(2)

　数式の文字並びで、英字が現われたら、直ぐに<u>pbSet2()</u>に処理を渡します。このとき、名前の解読が既に pbName()で済んでいる場合と、これから解読をしなければならない場合とがあります。式文の途中で何かの変数名を使っているときに、後者の場合が起こります。この判定は、英字名のプロパティ **iKwrdCode** で判断し、iKwrdCode=0 であれば pbName()を読んでプロパティを得ます。iKwrdCode は整数番号ですが、英字名を定義してあって、ソースコードの読み易さを図りました。ここでは説明に両方を示します。

1(COMMAND), :　数式にコマンド名が現われることはありませんのでエラーとします。

2(STATEMENT):　追加のコマンドです。上と同じ理由でエラーとします。

3(FUNCTION)　:　　関数名です。このときは、括弧が続く仕様です。数の型(**iVarType**)、引き数の数(**iKwrdNarg**)をリストセルに書き込み、このセルを<u>親リストセル</u>として<u>子リストセル</u>を引き数の数だけ作ります。兄弟関係のリンクを設定し、親のリストセルにぶら下げます。この処理は、<u>pbRegister()</u>に渡します。図 4.1 において、C と D とを括弧の下に繋げるのがそうです。この長男リストセルのポインタを pbRegister()の引き数が持ち帰ります。そうして、<u>pbSet0()</u>の頭に戻って文字解読を続けます。このときは、解読する文字位置のポインタが左括弧'('の位置の次に戻っています。関数値は、右括弧 ')'が現われる時点まで待って、<u>pbFunc()</u>で解決します。

4(OPERATOR) :　AND, OR などの文字表現演算子は、Plain_Basic では使用しません。

5(VARIABLE):　単純変数名です。変数が既に宣言済みであれば、変数名の保存位置のポインタ(iKwrdPoint)が 0 以外の値を持っています。0 の場合には、<u>pbDim3()</u>を呼んで、数の型(iVarType)と変数名(**kwName[]**)を元に、変数を変数領域 **vBuff** に作成し、初期値を設定します。変数値の位置は、相対アドレスの値が <u>**iKwrdPoint**</u> に返っていますので、これから変数の値を作業用のセルにコピーします。

6(ARRAY) :配列名です。この処理は、関数の場合と同じようにカギ括弧'['が続く仕様ですので、引き数の個数分の子リストセルを<u>pbRegister()</u>で準備してぶら下げ、長男リストセルのポインタを持ち帰ります。括弧の中を解読するため、<u>pbSet0()</u>の頭に戻りますが、この場合の文字位置のポインタは左カギ括弧'['の次に戻しています。実際の配列の値は、右カギ括弧']'が現われる時点まで待って、<u>pbSet1()</u>で解決します。

PbSet4()のアルゴリズム解説__(3)

　複数のリストセルが繋がっているとき、本文解読の一区切りのところで、<u>pbSet4()</u>が呼ばれます。そこでは、当該リストセルの上位リストセル(**iMother**)が在るかを調べます。例えば、図 4.1 では、D に対して左側 C が iMother になっていて、演算子＋で関連付けられていますので、この計算を実行させます。演算処理は、<u>gbAcc()</u>に処理が渡されます。そうすると、演算の右側（ここでは当該リストセル D)を返納して、計算結果を左側リストセルに移して <u>gbAcc()</u>から抜けます。そうして元の解析の流れに戻って文字解読を続けます。

pbSet0()のアルゴリズム解説続き__(4)

　最初の英字名の解（２）が済んで pbSet0()に制御が帰ると、文字解読のメインループに入ります。このとき、通常は演算子記号か、配列や変数名の場合には左括弧に続く文字列であって、数字か別の変数名や関数名が続き、作業対象のリストセルは、子リストセルに変わっています。したがって、解読対象の文字種は、数字・英字・記号文字の何れかです。数字の場合には、関数 <u>pbEncd0()</u>を呼んで数値にエンコードしてループ解読に戻ります。英字の場合、再度 <u>pbSet2()</u>を呼びます。記号文字の場合には、文字種が多いので、switch 文で個別の処理をさせます。少し入り組んだ判断と処理が行なわれます。個別の説明を次の項に続けます。

pbSet0()のアルゴリズム解説続き__(5)

まず、演算子記号は、加減乗除(+ − * /)・べき乗(^)・論理演算子(& @))・関係演算子(= < >)があります。これらが現われると、新しく子リストセルを作り、現在のリストセルと母子関係（左右）で繋ぎ、演算子の番号を保存し、次に現われる文字の解読を続けるためにメインループに戻ります。引用符は対で使い ″ ″ で挿まれた文字並びは文字型のデータと見なし、数字データと同じように一つのリストセルのデータ領域に保存して演算の対象にします。

pbSet0()のアルゴリズム解説続き__(6)

左括弧 ’(’は、関数名のすぐ後に続く場合と、数式の演算順序を限定する場合に使います。配列名・関数名のときは、当該リストセルに型の情報があります。それが 0 であれば数式に使う左括弧であると判断します。関数と配列は、プロパティを求めた pbName()の処理で、引き数の数をもとに父子関係（上下）の設定が済んでいますので、ここでは数式処理用の括弧として扱います。関数と配列の場合と同じように、**pbDim1()**でコンマ区切りの数を求めますが、その数は 1 であるのが標準です。父子関係を**pbRegister()**で作って、その子リストを持ってメインループの処理に戻ります。

PbSet1()のアルゴリズム解説__(7)

pbSet0()のループ処理で右括弧が見つかると、左括弧で始まった処理のまとめを**pbSet1()**で行なわせます。ここでは当該リストセルは父子関係の子セルでしたので、父セルのコードを調べ、そのコード別の処理に switch させます。そのコードは、(1)単純な数式、(2)配列、(3)関数です。数式の場合には、括弧内の数式を**pbSet4()**で確定し、その値を父セルにコピーして子セルをキャンセルします。関数であるときは **pbFunc()**で関数を計算してから **pbSet4()**で確定させます。配列の場合には多少複雑になります。それは、既に配列が宣言されていて実体が変数領域 vBuff にある場合と、これからデフォルトの寸法を持った配列を作成してから利用する場合とがあるからです。この区別は父セルのデータで判定します。変数領域のアドレス値が 0 であれば、父セルのデータを元に **pbDim3()**を呼んで、デフォルトの寸法を持った配列を作成します。そして、子セルのデータを解読して、配列内の特定したアドレスにある数値を父セルにコピーし、子セルをキャンセルします。

pbSet0()のアルゴリズム解説続き__(8)

pbSet0()のループ処理の終了は、数式としての文字並びに一区切りが済んだ場合であって、記号として文末(EOS)または行末(EOL)も含みます。代入文 A=X の形では、A の解読が済んだ場合と、X の解読が済んだ場合であって、処理の最終に **pbSet4()**が呼ばれます。このとき、作業用セルが一つだけ残り、これが最初に **pbSet0()**に制御を渡したときのセルであれば正常に解読されたことになります。

pbSet0()の動作制御に使うパラメータ iMod__(9)

コマンド DIM を使って或る次元数と寸法を持った配列を宣言するとき、数式解読が必要になります。この場合にも **pbSet0()**を使うのですが、この場合には、iMod=2 と使います。この iMod の違いは、**pbSet1()**のルーチン内で判別されます。

配列名だけを引き数として使う場合__(10)

サブルーチンを組み込んで利用することを考えて、配列名の利用方法を二種類定義できるようにしました。Plain_Basic では、配列内の位置を指定するときの番号は 1,2,3...のようにします。これを配列の**基底**が 1 であると言います。サブルーチンの引き数に配列を使う場合は、例えば A[3]のように表すのですが、特に配列の先頭を使う場合には A[1]と正直に書かなくても、A とだけ書く方法も許すことにしました。サブルーチン側では、原則として引き数はアドレス参照としますので、引き数に配列を要求する場合、A と A[1]とは同じ意義を持ちます。また、例えば A[3]とすると、A[3]を先頭とする配列と解釈します。(ただし、この仕様は幾らか危険を伴います。)

印刷制御に配列名を使う場合__(11)

コマンド PRINT を使って変数や配列をコンソールに書き出すとき、単純に引き数なしの配列名を使うと、その配列の全成分を出力するようにしました。この場合、配列名の参照に使う pbSet0(ist,iMod)では、iMod=3 のモードで使います。

表 4.6　数式解読処理 pbSet0()のソースコード

```
//========================== 840702 === FORTRAN 77S ======== BSETO
/*      SUBROUTINE BSETO( IST, IMODO)
    Arithmethical Accumulator
     ist --- local address to ListCell
     iMod ---(0)=ARRAY_EVALUATION,(1)=ARITHMETICAL_CALCULATION
          (2)=CREATE_ARRAY,                    (3)=PRINT_MODE.
*/
void pbSet0(Word& ist, const Word iMod0)  // 2006-02-10
{
     Char ch;
     Word ist0,istn,iMod,icTyp,idType,nSymb,kSymb, iMother;
     Word kOpeCode, nOpeCode, kDim, iPar, iElementSize;
     double dValue;
     bool bEnd;
     union {
          Char stringData[16];
          double dValue[2];
     } UCC;
     int i;
//
     if (bErrorFlag) return;
     iMod = iMod0;
     ist0=ListCell[ist].iPar;
     if (ist0 == 0) ist0 = ist;
     if (iKwrdCode != 0) {                // begins with ALPHA
          pbSet2(ist,ist0,iMod); if (bErrorFlag) return;
          if (iKwrdCode != 0) return;
          if ((iMod != 1) && (ist == ist0)) {
               pbSet4(ist,0);
               if (ist != ist0) pbErrList(ErrString16);
               return;};
     };
//
     for (;;){
          ch = pbCharRead();
          iKwrdCode=0;
          if ((cbtype == EOL) || (cbtype == EOS)) {
               bEnd = true; break;};
          icTyp = ListCell[ist].icType;
          nOpeCode = ListCell[ist].iOpeCode;
          bEnd = false;

          switch (cbtype){
               case DIGIT:
                    if (icTyp != 0 ) {bEnd = true; break;};
                    dValue = pbEncd0();         if (bErrorFlag) return;
                    ListCell[ist].dValue[0] = dValue;
                    ListCell[ist].iOpeCode = nOpeCode;
                    pbSet4(ist,1);                   if (bErrorFlag) return;
                    continue;
               case ALPHA:
                    pbSet2(ist,ist0,iMod);      if (bErrorFlag) return;
                    if (iKwrdCode != 0) return;
                    if (iMod == 1) continue;
                    if (ist != ist0) continue;
                    bEnd = true; break;
               case SYMBOL:
                    kOpeCode = 0;
```

```
switch(ch){
    case '+':
        kOpeCode=PLUS;
        // fall through
    case '-':
        if (ch=='-') kOpeCode=MINUS;
        if (icTyp != 0){
            pbSet4(ist,0);          if (bErrorFlag) return;
            istn = pbCellTake();if (bErrorFlag) return;
            ListCell[ist].iDaughter = istn;
            ListCell[istn].iMother = ist;
            ist = istn;
            ListCell[ist].iOpeCode = kOpeCode;
            execPointer++; continue;};
        if (nOpeCode ==0) {
            ListCell[ist].iOpeCode = kOpeCode;
            execPointer++; continue;};
        pbErrList(ErrString16); return;
    case '*':
        if (icTyp == STRING){
            pbErrList(ErrString02); return;};
        kOpeCode=MULTIPLY;
        break;
    case '/':
        kOpeCode=DIVIDE;
        break;
    case '&':
        kOpeCode=LOGICAL_AND;
        break;
    case '^':
        kOpeCode=POWER;
        break;
    case '>':
        kOpeCode=GREATER_THAN;
        if (*(execPointer+1) == '='){
            execPointer++;
            kOpeCode=GREATER_OR_EQUAL;};
        break;
    case '<':
        kOpeCode=LESS_THAN;
        if (*(execPointer+1) == '=') {
            execPointer++;
            kOpeCode=LESS_OR_EQUAL;};
        if (*(execPointer+1) == '>') {
            execPointer++;
            kOpeCode=NOT_EQUAL;};
        break;
    case '=':
        kOpeCode= EQUAL;
        break;
    case '@':
        kOpeCode=LOGICAL_OR;
        break;
    case '(':
        if (icTyp != 0 ) {bEnd = true; break;};
        kDim = pbDim1();        if (bErrorFlag) return;
        if (kDim != 1) {bEnd = true; break;};
        pbRegister(ist,0,1);
        continue;
    case ']':     // fall through
```

```
                    case ')':
                        pbSet4(ist,0);
                        pbSet1(ist,ist0,iMod); if (bErrorFlag) return;
                        execPointer++;
                        if ((ist != ist0 )|| (iMod0 == 1)) continue;
                        return;
                    case ',':
                        pbSet4(ist,0);              if (bErrorFlag) return;
                        iPar = ListCell[ist].iPar;
                        if (iPar == 0) {bEnd = true; break;};
                        istn = ListCell[ist].iJunior;
                        if (istn == 0) {pbErrList(ErrString02); return; };
                        ist = istn;
                        execPointer++; continue;
                    case '"':
                        if (ListCell[ist].icType != 0 ) {
                            pbErrList(ErrString13); return; };
                        for (i=0; i<16; i++ ) {
                            execPointer++;
                            ch = *execPointer;
                            if (ch == '"' ) break;
                            UCC.stringData[i] = ch;
                        }
                        if( ch != '"') {pbErrList(ErrString15); return;};
                        UCC.stringData[i] = '¥0';
                        ListCell[ist].iOpeCode = 0;
                        ListCell[ist].icType = STRING;
                        for (int i=0; i<2; i++) {
                            ListCell[ist].dValue[i]=UCC.dValue[i]; };
                        execPointer++; continue;
                    default: bEnd = true; break;
                };                  // end of switch(ch)   symbol selection
                if (icTyp == STRING){  // string addition pending
                            pbErrList(ErrString02); return;};
                if (bEnd) break;
                if (kOpeCode != 0) {
                        if (icTyp == 0) {
                            pbErrList(ErrString02);return;};
                        istn = pbCellTake(); if (bErrorFlag) return;
                        ListCell[ist].iDaughter = istn;
                        ListCell[istn].iMother = ist;
                        ist = istn;
                        ListCell[ist].iOpeCode = kOpeCode;
                        execPointer++; continue;
                };

        };                          // end of Switch(cbtype)
        if (bEnd) break;
    };                              // end of for loop
    pbSet4(ist,0);
    if (ist != ist0) {pbErrList(ErrString16);};
}
```

pbAcc()の解説

リストセルは、一つの数値をデータとして持ち、演算子記号を介して二つ以上のリストセルとリンク構造を形成します。演算は、形式の上では左から右に評価しますが、演算の優先順位があります。乗除計算を先に、加減算を後に行なわせます。基本的には、左右のリストセル間の演算を行なわせて、左側のリストセルに結果を納め、右側のリストセルを返納します。pbAcc()のルーチンは引き数として左右のリストセルへのポインタを持ち、左右の数値の型と、演算子記号を元に演算を行います。論理演算の場合には、倍精度実数を整数型に変換し、ビット単位での論理演算を行なわせ、その整数を倍精度実数に直して true と false の判定に使います。演算子の記号には番号が割り振られ、分かり易いソースコードにするため文字記号を下の表のように#define 文で定義しました。

表 4.7　演算処理を行う pbAcc()のソースコード

```
void pbAcc (Word& istA, Word& istB)
{                                          // 2006-02-08
     Word iMod, ima, imb, iTypeA, iTypeB;
     Word ncnt;
     double RA,RB,RAB,DIS;
     int IA,IB;
     const double TR=1.0e-20;

     if (bErrorFlag) return;
     iTypeA=ListCell[istA].icType;
     iTypeB=ListCell[istB].icType;
     if ((iTypeA == STRING) || (iTypeB == STRING))
          { pbErrList(ErrString02); return;};
     pbAcc2(istA);                // in case of negative sign
     pbAcc2(istB);
     imb=ListCell[istB].iOpeCode;     // 0-20, (+-*/^&=<>....@)

// 1:    First if loop -------------------------------------------
/* mathematical expression between scalors. */
     if ((iTypeA < 1000) && (iTypeB < 1000)) {
          RA=ListCell[istA].dValue[0];
          RB=ListCell[istB].dValue[0];
          RAB = RA - RB;
          switch(imb){
               case PLUS:    // fall through
               case MINUS:
                         RA=RA+RB; break;
               case MULTIPLY:
                         RA=RA*RB; break;
               case DIVIDE:
                         if(fabs(RB) <= TR) {pbErrList(ErrString11);return;};
                         RA=RA/RB; break;
               case POWER:
                         RA=pow(RA,RB); break;
               case EQUAL:
                    RA = 0;
                    if (RAB == 0.) RA = -1;
                    break;
               case LESS_THAN:
                    RA = 0;
                    if (RAB < 0.) RA = -1;
                    break;
               case GREATER_THAN:
                    RA = 0;
                    if (RAB > 0.) RA = -1;
                    break;
```

```
            case LESS_OR_EQUAL:
                RA = 0;
                if (RAB <= 0.) RA = -1;
                break;
            case GREATER_OR_EQUAL:
                RA = 0;
                if (RAB >= 0.) RA = -1;
                break;
            case NOT_EQUAL:
                RA = 0;
                if (RAB == 0.) RA = -1;
                break;
            case LOGICAL_AND:
                IA = (int)RA;
                IB = (int)RB;
                IA = IA & IB;
                RA = (double)IA; break;
            case LOGICAL_OR:
                IA = (int)RA;
                IB = (int)RB;
                IA = IA | IB;
                RA = (double)IA; break;
            default:
                pbErrList(ErrString11);return;
        };
        ListCell[istA].dValue[0] = RA; return;
    };
};
```

表 4.8： 演算子記号の種類と番号(英字を使う場合を含めてありません)

演算子記号	番号	定義する英字名
+	0	PLUS
-	1	MINUS
*	2	MULTIPLY
/	3	DIVIDE
^	4	POWER
&	5	LOGICAL_AND
=	14	EQUAL
<	15	LESS_THAN
>	16	GREATER_THAN
<=	17	LESS_OR_EQUAL
>=	18	GREATER_OR_EQUAL
<>	19	NOT_EQUAL
| または @	20	LOGICAL_OR

5. データリストの解読

5.1 概説

データを扱う論理デバイス

Plain_Basic は、数字に代わる英字名を使った代数式を解読して数値計算ができます。この前段階として、英字名で表された変数や配列の宣言と、それに具体的な数値データを代入する手続きが必要です。データは、プログラム内部にあらかじめ準備するか、ディスクなどの外部装置（デバイス）にファイルとして準備し、それを読み出す処理が必要です。DOS の環境では、ディスクの拡張した概念に、キーボードも含みます。どのデバイスを使う場合であっても、データの集合をレコード、レコードを集めたものをファイルと言う概念で捉えます。レコードは、すべて、文字並びです。ディスクはコンピュータ側から見ると外部装置ですし、その扱いがデバイス毎に異なります。当然、プログラミング技法も異なります。そのことを表す用語がオブジェクト指向プログラミング(object oriented programming)です。初心者レベルの利用を考えた Plain_Basic では、意図的にデバイスの扱い、つまりファイルの扱いを最小限に抑えました。しかし、ファイルからレコードを読み出し、それからデータを解読して取り出す手続きの理解は必要です。Plain_Basic では、その手続きを内部ファイルで扱うようにしました。

データ読み出しには定型化した手続きがあること

ファイルの中身を読み出す処理は、一般的に言うと、下のような定型化した手続きを踏まえます。

(1) デバイスとの接続を行う（ファイルのオープン処理がそうです）
(2) ファイルからレコード単位でデータを読み出す(read, get ,input などの用語を使います)
(3) レコードの中身から個別のデータを解読して変数などに代入する（エンコードです）
(4) 処理が終了したらデバイスとの接続を切る（クローズ処理です）

ディスクからデータを読み込むときには、ユーザのプログラムでは前もってファイルの OPEN と後始末の CLOSE が義務化されています。コンソールからデータを入力するとき、DOS の環境では明示的なオープン処理を必要としませんが、Windows の環境では、デバイスを準備することからプログラミングを始めなければなりません。Plain_Basic では、ユーザレベル向けに、上の(1)と(4)の部分を簡略化してあります。ユーザが留意するのは(2)と(3)であって、データの準備方法と関連を持ちます。

ファイルの読み出し処理も３通り

Plain_Basic は、ディスクからのデータ「読出し・書込み」に関連したステートメントを含めてありません。つまり、高級プログラミングで眼にする OPEN, CLOSE, INPUT, PUT, GET 相当するステートメントがありません。しかし BASIC のプログラム文をファイルに保存して読み出すコマンド(LOAD/SAVE/MERGE)は必須ですので、データファイルの「読出し・書込み」も、この処理に含ませるようにしてあります。デバイスが実体のあるディスクであるとき、そこにあるファイルを外部ファイルと言う呼び方は理解できるでしょう。しかし、内部ファイルは、ディスク装置などの実体がありません。データファイルの中身のレコードがプログラム文の中に一緒に組み込んであって、そのレコードからデータの読み出す方法が外部ファイルの読み出しと似ていますので、このように呼びます。高級プログラミング言語では、内部ファイルを読む方法を使いません。Plain_Basic のプログラム文で宣言した変数や配列にデータを代入する書式は、第 4.3 節で紹介しました。それを整理して再録したのが、下の表 5.1 です。

表 5.1　データ読み込みの書式

入力要求の書式		データ準備の書式	
ステートメント文字列	変数名リスト（例）	データリスト（例）	デバイス
READ	A,B,C	DATA 3, 5, 6.78	内部ファイルから入力

データ代入の手順は５段階で構成される

ファイルのレコードを読み出して変数ごとのデータに代入する処理は、デバイスからの読み出し手順がデバイス毎に異なるだけで、レコード解読部分のアルゴリズムすべて基本的に同じ流れです。その手続きを５段階に分けます。

- 第１段階は、データリストの設計と準備です（表 5.1 の第３欄）。解読の処理手順は、下で説明する変数名の並びと関係しますので、後で詳しく解説します。Plain_Basic では内部ファイルを扱い、そのレコードはプログラム文と同居しますので、プログラム文の構築と同じように、ラベルを付け、データリストの頭にキーワードの DATA を付けて、メモリ上に作成します。データ部分だけを別のファイル名で保存すれば、プログラム文とデータ文の二つを MERGE コマンドでまとめることができます。この場合、ラベルの番号が重ならないようにします。

- 第２段階は、キーワードに続く変数名文字リストの設計です（表 5.1 の第２欄）。Plain_Basic の実行時に READ 文に制御がくると、変数名を解読して、そのアドレスなどの情報を変数用リストセルで積み上げます。これは、pbRead0 の関数名にしました。まだ宣言されていない変数名が使われていれば、ここでその変数名を登録して、そのアドレスを渡します。

- 第３段階は、データ並びの文字列レコードの取りだしです。内部ファイルの場合には、キーワードの DATA が付いた行がレコード行ですので、その場所に解読ポインタを設定します。この場所を、一般にはバッファーと言います。論理的には一続きの連続した文字並びの場所であって、そこに複数のデータが連続します。読み始め解読ポインタは、最初の DATA 行にありますが、データ並びを順に読み出し、その行のデータが終われば次の DATA 文にポインタを移します。途中までデータが読まれているときは、解読ポインタは途中のままになっていて、次の READ 文は、そこから読みます。この処理を含め、残り全体を pbData1()が処理します。

- 第４段階は、レコードの文字列並びから個別にデータをエンコードします。この文字並びの書式には、例えば同じ数値 2.5 を 10 個並べる代わりに 10*2.5 のように表す方法を使うことができます。そのため、データ用リストセルを作成して、必要文の個数のデータをこのリストセルから供給します。

- 第５段階は、代入処理です。変数用リストセルと、データ用リストセルを使います。処理が済んで不要になったリストセルを順次キャンセルしていき、変数用リストセルが無くなったところがデータ読み出し処理の区切りです。未使用のデータ用リストセルが残っていることもあり、これは次の読み出し処理に使います。

変数の個数は一行で 10 個程度に抑える

表 5.1 に例示した変数名は A,B,C の３個ですが、変数名の文字並びは、一行に書くことが許されるステートメントの最大文字数（約 120 バイト）で制限されます。標準的には 80 byte 程度です。変数の個数は、作業用のリストセルの最大個数で制限を受けます。差し当たって 30 セルです。多くの変数にデータを読み込みたいときは、一行の変数を 10 個程度に分けて、何回かに分けて読み出すとよいでしょう。

5.2　変数名リストを解析する関数 pbRead0()

pbRead0()は、代入文の左辺値に使う複数の変数名の解読です。代入文は一つの変数名ですが、ここではコンマで区切られた英字で始まる変数名リストを解読します。変数用リストセルを変数名単位で準備し、pbSet0()を呼びます。この変数用リストセルを変数名順にリンクし、先頭の変数用リストセルの番号を ReadRegister に納めます。直接利用するリストセルの項目を表 5.2 に示します。

表 5.2　変数用リストセルの仕様

ListCell[]の変数名	値	解説
.iDaughter		次の変数用リストセル
.icType	STRING, DOUBLE	データの型
.iSenior	nCnt	成分数の保存に使用
.elementAddress		相対アドレス
.dValue[0] .dValue[1]		代入する数値 または文字列

表 5.3　pbRead0(void)のリスト

```
void pbRead(void)                          // 2006-04-21
{                                              // prepare ListCells for variables
     Char ch;
     WORD ist, ist0, nType, iDim, iMod;
     int nCnt, iPoint;
     ist = 0;                              //
     ReadRegister = 0;                     // top cell register initialize
     for (;;){
          ch = pbCharRead();
          switch(cbtype) {
               case EOL:                   // fall through
               case EOS:                   // end of a statement
                    return;
               case ALPHA:                 // only alphabetical names allowed
                    pbName();              if (bErrorFlag) return;
                    if (iKwrdCode < 5)     {pbErrList(ErrString02);return;};
                    nType = iVarType;      // save work data
                    iPoint = iKwrdPoint;   // address at control data
                    iDim = iKwrdNarg;
                    iPoint = iKwrdPoint;
                    nCnt = 1;
                    ist0 = pbCellTake();   if (bErrorFlag) return;
                    iMod = 0;
                    if ((iKwrdCode == 6 ) && (iKwrdNarg == 0)) iMod = 3;
                    pbSet0(ist0,iMod);          if (bErrorFlag) return;
                    if (ist == 0) { ReadRegister = ist0;}
                         else{ListCell[ist].iDaughter = ist0;};
                    ist = ist0;
                    if (iMod != 1){
                         for (int i=0;i<iDim;i++) {nCnt *=iVarElementSize[i];};};
                    ListCell[ist].iSenior = nCnt;    // use as a work memory.
                    ListCell[ist].icType = nType;

                    ch = pbCharRead();     // serach next variable name
                    if ((cbtype == EOL) || (cbtype == EOS)) return;
                    if (ch != ',' )    {pbErrList(ErrString02); return;};
                    execPointer++;
                    continue;
               default:
                    pbErrList(ErrString02); return;
          };                                   // end switch(cbtype)
     };                                        // end of for(;;) loop
}
```

5.3 データリストの書式

データリストは数字か記号(")で始まる文字並び

数値データは、数字で始まる文字並びを１解読単位として、スペースなどの区切り文字で区切ったリストで与えます。文字列データは、二つの引用符 "" で囲みます。文字列データは文字型の配列に納めます。一つの文字列データの文字数は半角 15 字までです。型が合わなければエラーとします。データリストの書式例を下に示します。ただし、内部ファイルは、キーワードの DATA をデータリスト行の頭に付け、プログラム文と一緒にメモリに保存します。

表 5.4 データリストの書式の例

種別	リストの例	解　　説
①	1 2 5 9.7 2.1E6 "Name1"	データ間をスペースで区切った場合
②	1, 2, 5, 9.7, 2.1E6, "Name1"	コンマで区切った表示法。スペースも挿めます
③	9*0, 8*0.35	0 が９個、0.35 が８個続くデータである
④	1 TO 100 STEP 3, 120 TO 125	FOR-NEXT の書式で表す方法
⑤	10*/ 5/	配列の最初の 10 個をスキップして 11 番目を５とする

上で挙げたデータリストは例ですので項目単位の詳しい構文規則を下のように約束します。

(1) 項目単位では、文字並びの先頭は、数字、引用符記号（"）、またはマイナス符号(－)です。
(2) マイナス符号は次に必ず数字が続きます。プラス記号(+)は使いません。
(3) 引用符記号は必ず対で使い、その内側が任意の文字データであることを知らせます。
(4) 文字データの長さは半角 15 文字以内です。日本語の２バイト文字も扱えます。
(5) 整数は、数字だけで構成する文字並びです。単独に 0 と使う以外に頭に 0 が付きません。
(6) 実数は、小数点を含みます。指数表示を持つ場合もありますが、スペースを挿みません。
(7) 小数点で始まる実数表示はしません。この場合には 0. のように頭に 0 を補います。
(8) 同じ数 N1 を N2 個並べる代わりに **N2*N1** と書くことができます。スペースを挿みません。
(9) 上の書式で、N2 は符号無し整数、N1 は整数または実数で、符号－を持つこともあります。
(10) 同じ文字データを N2 個並べる場合に **N2*"…"** の表記で応用することができます。
(11) {**N1 TO N2 STEP N3**}は一つの項目単位です。昇順または降順の複数の並びを表します。
(12) {**N1 TO N2**}の表記は、N3=1 である場合の省略記法（デフォルト）です。
(13) 上の書式でのデータ個数は、(N2-N1)/N1+1 で計算される正の整数個です（負はエラー)。
(14) 変数（配列も含む）の成分並びで N 個の成分を読み飛ばすときは、 **N*/** でスキップさせせます。
(15) N*/の書式では、先頭１成分を読み飛ばすときには 1*/ と使います。
(16) 代入指定箇所が連続していても、残りはそのままにしたいとき、直後に/を使います。
(17) データ項目の表記単位の区切り文字(delimiter)は、スペースまたはタブを標準とします。
(18) コンマ (,) は明示的に区切り箇所を示す区切り文字に使うことができます。
(19) 改行コードは一つの区切り文字扱いです。

データ個数は入力要求変数名の個数分が必要

データリストのデータ個数は、入力要求の変数の個数と一致しなければなりません。既に宣言済みの配列（例えば DIM A[10])があって、具体的に添え字付きで入力要求リストにあるとき（例えば A[4])、これは一つの変数扱いです。しかし、これが添え字表示無しの配列名（例えば A)で入力要求リストにあるとき、その全要素個数分（この例では 10 個）のデータ個数が必要です。データリストのデータ並びの個数が多くて一行に収まらない場合には、次の行にリストが続くとして読み出します。逆に、当面の変数または配列成分の個数よりもデータリストのデータ個数が多いとき、デバイス毎にその処理の考え方が違います。そのことを次に説明します。

内部ファイルには RESTORE 文を使うことができる

内部ファイルは、通常のテキストファイル各行に DATA のキーワードが頭に付いた形式です。BASIC プログラム文と混ぜて構成しますが、READ 文は DATA 文の行を選択して順に読み出します。内部ファイルを利用するときの利点は、特定の READ 文に合わせて、DATA 文の読み出し位置を明示的に指定できることです。読み込みに必要とするデータの個数が少ないのはプログラムミスですが、読まない場合も起こります。正しいデータグループを使いたいときは、データ読み出しの頭出しの行をラベルで指定できます。これが RESTORE 文です。

連続データを読み込むときの繰り返し指定

幾つかの変数名および配列名の並びにデータを代入するとき、それらは総て一続きの変数並びとして扱います。配列が既に宣言されていて、配列成分の番号（添え字）が省略されている場合には、配列の全成分が並んでいるとみなして、その全個数分のデータを要求します。この仕様は Plain_Basic の特徴の一つです。例えば行列を宣言する A[3,3]は９個分のデータリストが必要です。もし行列要素を総て０でクリアしたいとき、配列名 A でデータを要求します。0 を９個並べる代わりに 9*0 の表記を使うことができます③。この場合には、２次元以上の配列も１次元の並びで扱いますので、メモリ上で成分並びの約束が必要です。Plain_Basic では、例えば２次元配列 A[3,3]では、メモリ上で A[1,1], A[2,1], A[3,1], A[1,2], A[2,2],…の順に並びます。データに規則性があって、FOR-NEXT 文形式表現できるような方法も準備しました④。「1 TO 100 STEP 3」は 1,4,7,10,…97,100 まで、計 34 個のデータリストを表します。この表記法は、必要個数を間違いなく準備しなければなりません。

途中を読み飛ばしてデータを代入させたいとき

配列名だけが指定されているとき、配列の特定の箇所だけにデータを代入し、他の箇所はそのままにしたいことがあります。その拡張として、幾つかの変数と配列並びに、データを変えて何回も使いたきこともあって、特定箇所の位置までの成分個数は、データを代入しないでスキップさせたいことがあります。これを、スラッシュ記号(/)で制御します。スラッシュ１つは１データをスキップする意味です。10*/は、10 データ位置をスキップすることを意味します。特定箇所から後ろに成分の残りがあって、代入を打ちきりたいときは、データリストの最後に(/)を付けます。この制御方法は、Plain_Basic 固有の仕様です。

文字列型の配列に代入するときは FOR-NEXT 文形式を使わない

文字列型の変数は配列に作ることができますので、その場合には連続データの読み込みとスキップ仕様を使うデータ代入ができます。ただし FOR-NEXT 形式は数値だけに利用する書式ですので文字列型では使えません。

5.4 データリストの解読ルーチン pbData1()

データのエンコードに作業にデータ用リストセルを使う

表 6.4 に示したようなデータリストの文字並びは、解読して変数または配列に代入するのですが、一旦作業用のリストセルに整理します。連続データ仕様の場合は、個数分のデータを順に供給しなければなりませんので、一つのリストセル準備し、それを介して代入します。そのリストセル利用方法の仕様を下の表 5.5 に示します。

表 5.5 データ作業用リストセルの仕様

ListCell[]の変数名	値	解説
.icType	STRING, DOUBLE	データの型
.iCode	0,1	データスキップコード
.dValue[0]	N1	代入する数値
.dValue[1]	Ncnt	個数カウンター(正の整数)
.dValue[2]	N3	増分
.dValue[3] .dValue[4]	"string data"	文字型を倍精度数２語に割り当て
.dValue[5]	N2	{N1 TO N2}作業用
.dValue[6]	N3	{N1 TO N2 STEP N3}作業用
.elementName[]		文字データ作業用

データ用リストセル作成とキャンセルのタイミング

- 関数 pbData1 に制御が入ると、差し当たって、現在の文字解読用ポインタ execPointer を recoverPointer に保存します。そして、最も外側の for ループＡとして、ReadRegister が指定する変数用リストセルのリンクを頭から順に読み出します。リストセルが終了すれば、このループから出ます。このセルは、「変数の型、変数位置の相対アドレス、成分個数」が要求項目として入っています。

- ここで、一つ内側に for ループＢを立ち上げます。データ用リストセルは、データ項目１単位の文字並びを解読（エンコンード）するときに一つだけ作成します。DataRegister は共通変数であって、データ用リストセルの番号を保存します。初期値は０です。まず DataRegister を調べます。これが 0 でなければ、作成済みのリストセルが未だあります。これが０であれば、データ用リストセルを一つ予約し、データ項目解読ルーチン pbData1 を呼んで表 5.5 の中身を作成し、DataRegister にその番号を記録します。

- データ用リストセルには、個数カウンタ nCnt があります。変数用リストセルにも要求成分個数がありますので、同期を取って、変数にデータを代入し、それぞれの個数を一つずつ減らします。この作業を二回り下の for ループＣに構成します。このループは、二つの条件で終了します。第一は、変数用リストセルの要求成分個数が 0 になったときであって、for ループＡの先頭に制御を戻します。第二は、データ個数が 0 になったときです。データリストセルをキャンセルし、DataRegister=0 とします。そして、新しくデータ項目を解読するため、制御を for ループＢの先頭に戻します。どちらの場合にも、一旦ループＣから外に出て、ループＢを経てループＡに戻します。

- ループＡから外に出るときは、変数への代入処理が済んだときです。そして、recoverPointer を execPointer に復元します。この場合、DataRegister が 0 でなければ、データ用リストセルに未だデータが残っています。しかし、コンソールからデータを入力するときは、残りのデータがあればそれを無視しますので、DataRegister=0 とし、コンソールにその通知メッセージ"*** extra data are ignored ***"を表示します。

- 関数 pbData1 は、引き数としてリストセルの番号 istD と、擬似デバイスの番号 iDev を使います。デバイスによって、バッファーアドレスは異なりますので、ポインタ用変数名はデバイスに固有するように決めておきます。文字列読み出しのポインタ execPointer は、このアドレスに設定換えします。

- バッファー内の文字並びの解読ループに入ります。バッファーの初期値は空です。読み出した文字がヌル文字'¥0'であれば、ファイルからデータ文字列のレコードを読み出して、ポインタをバッファーの先頭に設定します。iDev によってファイルの読み出し方法が違います。ファイル末に来てレコードが無ければエラー終了です。

- データリストの文字並び

データの文字並びを保存するバッファーのポインタを、ReadRegister を一つずつ順に読み出して、そこで要求されているデータを原則として1個ずつエンコードして作業用のリストセル(DataRegister)に納めます。繰り返しデータの場合には、初期値・増分・回数を解析してリストセルに納めます。

表 5.6　文字の並びの出現順で分類するステータスコード kState

現在の kState	現在の文字並び	次に来る文字	更新文字並び	更新 kState	カウンタ Cnt	
1	無し	数字	N1	2		
		/		→	0	end
		″	″string″	→	1	
		その他				エラー
2	N1	*	N1*	3		
		英字 TO		5		
		del,EOL,EOS		→	1	
		/		→	1	end
		その他				エラー
3	N1*	数字	N1*N2	→	Cnt=N1	
		/	N1*/	→	Cnt=N1	
		″	N1*″string″	→	Cnt=N1	
		その他				エラー
4	定義なし					
5	N1 TO	数字	N1 TO N2	6	Cnt を計算	
		その他				エラー
6	N1 TO N2	del,EOL,EOS		→		
		英字 STEP	N1 TO N2 STEP	7		
		/		→	Cnt を計算	end
		その他				エラー
7	N1 TO N2 STEP	数字	N1 TO N2 STEP N3	→	Cnt を計算	
		その他				エラー

註１：del は、区切り文字であって、コンマ、タブ、スペースである。
註２：マイナス記号(-)、は数字の始まりとみなす。数字はエンコードされる。
註３：英字は、キーワードと比較して調べる。
註４：記号→は return の意。
註５：Cnt は、代入する個数。Cnt 個代入した後で、end の判定をする。
註６：end は、READ 文で代入する変数が残っていても、データの代入を打ち切る。

表 5.7 pbData1()のリスト

```
void pbData1(void)
{
     Char ch, *chptr;
     WORD istR, istn, istD, ivType, nLabel, len, iSlash, iOpeCode;
     int iPoint, nrCnt, ndCnt;
     bool bfound, bEnd;
     if (bErrorFlag) return;
     recoverPointer = execPointer;    // save current Program execution pointer
     istR = ReadRegister;
     istD = DataRegister;
     if (istR == 0) {pbErrList(ErrString02); return;};   // no variables
     execPointer = NULL;
//
     for (;;){                        // for-loop A: create DataRegister
          if (istR == 0) break;
          if (DataRecordID == 0) {
               for(;;) {// local loop-E begins
                    bfound = false;
                    chptr = nextDataPtr;
                    nLabel = *((WORD*) chptr);
                    if (nLabel == 0) {pbErrList(ErrString14); return;};
                    iDataLabel = nLabel;
                    len = *((WORD*) (chptr +2));
                    nextDataPtr += len;
                    execPointer=chptr+4;
                    for(;;){ // local loop-F: is keyword 'DATA' ?
                         ch = pbCharRead();
                         if (ch == ':' ) {execPointer++; continue;};
                         if (ch =='¥0' ) {bfound = false; break;};
                         if (ch == 'd' ) {
                              pbName(); if (bErrorFlag) return;
                              if (iKwrdID == KWD_DATA)
                                   {bfound = true; break;};
                         };
                         pbSkip();
                         ch = * execPointer;
                         if (ch != ':')  break;
                         execPointer++;
                    };       // loop-F next 'DATA'
                    if (bfound){DataRecordID = 1;break;};
               };            // next loop E
          };
          if (bfound == false) {pbErrList(ErrString14); return;};
/* create DataRegister */
          if (istD == 0) {
               pbData2(istD, bEnd); if (bErrorFlag) return;
               if (bEnd){        // end of data file record
                    DataRecordID = 0;
                    DataRegister = 0;
               };                // endif (bEnd)
          };                     // endif (istD==0)
/* assign data */
          if (istD != 0) {
               for (;;){              // for-loop B
                    iSlash = ListCell[istD].iCode;
                    iOpeCode = ListCell[istD].iOpeCode;
                    iPoint = ListCell[istR].elementAddress;
                    nrCnt = ListCell[istR].iSenior;
```

```
                    ndCnt = (int) ListCell[istD].dValue[1];
                    if (iSlash==0)
                        vBuff[iPoint]=ListCell[istD].dValue[0];
                    if (ivType==STRING)
                        vBuff[iPoint+1]=ListCell[istD].dValue[1];
                    nrCnt--;
                    ListCell[istR].iSenior=nrCnt;
                    ListCell[istR].elementAddress++;

                    if (nrCnt == 0) {
                        istn = ListCell[istR].iDaughter;
                        pbCellBack(istR);
                        istR = istn;};

                    ndCnt--;
                    ListCell[istD].dValue[1] = (double) ndCnt;
                    if ((ndCnt !=0) && (iOpeCode ==2))
                        ListCell[istD].dValue[0] += ListCell[istD].dValue[2];
                    if (ndCnt <=0){
                        pbCellBack(istD);
                        DataRegister=0;istD=0;
                        break;};
                };                      // next for-loop B
            };                          // endif istD
        };                              // next for-loop A
/* resume */
        execPointer = recoverPointer;   // recover execution pointer
}
```

表 5.8 pbData2()のリスト

```
/*========================== 840702 === FORTRAN 77S ====== BSCANN
      SUBROUTINE BSCANN(TEXT,ICTRL,IST)
  TEXT SCANNER FOR NUMERICAL DATA.
  end of file when bEnd=true
*/
void pbData2(Word &istD, bool &bEnd)
{
    Char ch, *startPointer, *src, *dst;
    Word icType, nByte, kState;
    int jj, iValue, nCnt;
    double aValue, bValue, sStep;
//
    istD = pbCellTake(); if (bErrorFlag) return;
    kState=1;
    bEnd=false;
    for (;;){
        ch = pbCharRead();
        if (ch=='-') cbtype=DIGIT;
        if (ch==',') cbtype=EOS;
        switch (cbtype){
            case EOS: execPointer++;    // fall through
            case EOL:
                if (cbtype==EOL) bEnd=true;
                switch (kState){
                    case 1:
                        pbCellBack(istD); istD =0; return;
                    case 2: return;
                    case 6: return;
```

```
                default:pbErrList(ErrString02); return;
            };
        case DIGIT:
            if ((kState==1) || (kState==3) || (kState==5) || (kState==7)){
                aValue = pbEncd0();     if (bErrorFlag) return;
                icType=DOUBLE;
                iValue=(int)aValue;
                if (aValue ==(double)iValue) icType=INTEGER;
            };
            switch (kState){
                case 1:                          // N1
                    ListCell[istD].dValue[0]=aValue;
                    ListCell[istD].icType=icType;
                    ListCell[istD].dValue[1]=1.;
                    kState=2;
                    continue;
                case 2:return;                       // next data is DIGIT
                case 3:                          //  N1*N2
                    if (ListCell[istD].icType != INTEGER)
                        {pbErrList(ErrString02); return;};
                    if (ListCell[istD].dValue[0]<0)
                        {pbErrList(ErrString02); return;};
                    ListCell[istD].dValue[1]=ListCell[istD].dValue[0];
                    ListCell[istD].dValue[0]=aValue;
                    ListCell[istD].icType=icType;
                    return;
                case 4: return;
                case 5:                          // N1 TO N2
                    ListCell[istD].dValue[5]=aValue;
                    nCnt=(int) (aValue-ListCell[istD].dValue[0]);
                    sStep=1.;
                    if (nCnt<0) {
                        sStep=-1.;
                        nCnt=-nCnt;};
                    ListCell[istD].dValue[1]=(double)nCnt;
                    ListCell[istD].dValue[2]=sStep;
                    kState=6;
                    continue;
                case 6: return;
                case 7:                          // N1 TO N2 STEP N3
                    ListCell[istD].dValue[6]=aValue;
                    nCnt=(int) ((ListCell[istD].dValue[5]
                        -ListCell[istD].dValue[0])/aValue);
                    if (nCnt<0)   {pbErrList(ErrString02); return;};
                    ListCell[istD].dValue[1]=(double)(nCnt+1);
                    ListCell[istD].dValue[2]=aValue;
                    return;
                default:
                    pbErrList(ErrString02); return;
            };
        case SYMBOL:
            if (ch == ',') {
                execPointer++;
                switch (kState){
                    case 2: return;
                    case 4: return;
                    case 6: return;
                    default:pbErrList(ErrString02); return;
                }; };
```

```
                    if (ch == '/') {bEnd = true; return;};
                    switch (kState){
                         case 1:
                              if (ch == '"'){
                                   pbSkipQ(); if (bErrorFlag) return;
                                   nByte=execPointer - startPointer -1;
                                   if (nByte>15) {pbErrList(ErrString15); return;};
                                   src=startPointer;
                                   dst=ListCell[istD].elementName;
                                   for (int i=0;i<nByte;i++){*dst++ = *src++;};
                                   *dst='¥0';
                                   return;};
                              pbErrList(ErrString02); return;
                         case 2:
                              if (ch == '*') {kState=3;execPointer++; continue;};
                              if (ch == '/') return;
                              pbErrList(ErrString02); return;
                         case 3:
                              if (ListCell[istD].icType != INTEGER)
                                   {pbErrList(ErrString02); return; };
                              nCnt = (int)ListCell[istD].dValue[0];
                              ListCell[istD].dValue[1] = (double) nCnt;
                              ListCell[istD].iCode = 1;        // N1*/
                              return;
                         case 6:
                              if (ch == '/')return;
                              pbErrList(ErrString02); return;
                         default:
                              pbErrList(ErrString02); return;
                    };        // end switch(kState)
               case ALPHA:
                    if ((ch !='t') && (ch !='s'))
                         {pbErrList(ErrString02); return;};
                    if ((kState == 2) || (kState == 6)) {
                         pbName();              if(bErrorFlag) return;
                         if ((iKwrdID == KWD_TO ) && (kState ==2))
                                   {kState = 5; continue;};
                         if ((iKwrdID == KWD_STEP) && (kState ==6))
                                   {kState = 7; continue;};
                    };   // end if
                    pbErrList(ErrString02); return;
          };   // end switch(cbtype)
     };        // end for_loop
}
```

6. 実行制御

6.1 概説

実行制御は文に目印が必要であること
　Plain_Basic は、原則としてコマンド起動型の処理の集合です。テキスト入力枠からコマンドやステートメントを入力させて、直接モードで実行させる機能が基本ですが、能率よく利用するため、入力文をファイルに保存し、それを順に読み出した実行させるバッチ処理モードと、ラベルを付けてプログラム文に編集して実行させるプログラム実行モードが準備されています。プログラム文の場合、プログラム文の実行位置を選択して実行位置を変更することができます。そのときに使う文は、FOR 文、GOTO 文、GOSUB 文、IF 文の四種類です。プログラム文の実行位置を変更するためには、プログラム文に目印が必要です。その最も単純な方法が、プログラム文に整数のラベルを付けておくことです。GOTO 文、GOSUB 文はそのラベルを目印に飛び込み先を探します。目印はラベルだけとは限りません。FOR 文は、キーワードの NEXT を目印に使います。READ 文はキーワードの DATA を目印にしますが、実行制御ではキーワードの REM と同じく読み飛ばしを指定する目印になります。IF 文では、条件次第でキーワードの ELSE を目印とし、ELSE 以降を読み飛ばすか、ELSE までを読み飛ばすかの制御を行わせます。

リストセルを帰りのアドレス保存に使うこと
　GOTO 文は、行き先の文のラベルを元に、次に実行するプログラム文を探して、そこに文字列解読のポインタを設定します。これに使う関数は pbSortLineNr()です（表 3.7 のリスト参照）。IF 文は、条件判定をして GOTO 文で制御位置を変えることがあります。GOTO 文は行きっぱなしですので、飛び出し元の文字列のポインタを保存する必要がありません。この部分のソースコードは、表 1.5 の pbCommand1()に入れてあります。これに対して、FOR 文と GOSUB 文は制御を元の位置に返す必要がありますので、飛び出し元の文字列の位置、それにラベルなどを保存しなければなりません。これにはリストセルを使います。特に FOR 文と GOSUB 文では入れ子式にループを構成し、一番内側から解決しなければなりませんので、後入れ先だし(LIFO:last in first out)でアドレス情報を戻します。FOR-NEXT の処理部分のリストを、表 6.1、表 6.2 に示します。GOSUB 文のソースコードはやや短いので、表 1.5 pbCommand1()に含めました。IF 文は、論理式を計算し、その結果を判定して処理を選択します。ソースコードを表 6.3 に示します。

表 6.1　pbFor()のリスト

```
/*=========================== 840702 === FORTRAN 77S ======= BFOR
       SUBROUTINE BFOR(TEXT,ICTRL)
  FOR-NEXT LOOP INITIALIZATION.
*/
void pbFor(void)
{                                           // 2006-02-25
     Char ch, *src, *dst, Name[16];
     double init,iTo, iStep;
     int iPoint, nCnt;
     Word iNext, iNext0, ivType, ist;
     ch = pbCharRead();
     if (cbtype != ALPHA)             {pbErrList(ErrString02); return;};
/* 'FOR ???' */
     pbName();                              if (bErrorFlag) return;
     if (iKwrdCode != 5)              {pbErrList(ErrString02); return;};
     if ((iVarType != INTEGER) && (iVarType != DOUBLE))
                                            {pbErrList(ErrString13); return;};
     ist = pbCellTake();              if (bErrorFlag) return;
     ivType = iVarType;
     src = kwName; dst=Name;
     for (int i=0;i<16;i++) {*dst++ = *src++;};
     if (iKwrdPoint == 0) pbDim3(Name,0,ivType);    // create a variable ???
                                            if (bErrorFlag) return;
```

```
    iPoint = iKwrdPoint + 4;
/* '=' */
    ch =pbCharRead();
    if (ch != '=')                    {pbErrList(ErrString02); return;};
    execPointer++ ;
    iKwrdCode = 0;
    pbSet0(ist,1);                    if (bErrorFlag) return;
    init = ListCell[ist].dValue[0];
    pbCellBack(ist);
    vBuff[iPoint] = init;
/* 'TO' */
    if (iKwrdID != KWD_TO)        {pbErrList(ErrString02); return;};
    ist = pbCellTake();               if (bErrorFlag) return;
    iKwrdCode = 0;
    pbSet0(ist,1);                    if (bErrorFlag) return;
    iTo = ListCell[ist].dValue[0];
    pbCellBack(ist);
/* 'STEP' */
    iStep=0.;
    if (iKwrdCode != 0){
        if (iKwrdID != KWD_STEP )   {pbErrList(ErrString02); return;};
        ist = pbCellTake();           if (bErrorFlag) return;
        iKwrdCode = 0;
        pbSet0(ist,1);                if (bErrorFlag) return;
        iStep = ListCell[ist].dValue[0];
        pbCellBack(ist);
    };
    if (iStep == 0) iStep = 1;
    nCnt = (int) ((iTo - init) / iStep);
/* create a ListCell for 'FOR-NEXT register' */
    iNext = pbCellTake();         if (bErrorFlag) return;
    iNext0 = nextRegister;        // previous ListCell
    nextRegister = iNext;         // keep the latest ListCell
    if (iNext0 != 0) {
        ListCell[iNext].iMother =iNext0;
        ListCell[iNext0].iDaughter =iNext;};
    ListCell[iNext].icType = ivType;
    src = Name;
    dst = ListCell[iNext].elementName;
    for (int i=0;i<16;i++) {*dst++ = *src++;};
    ListCell[iNext].elementAddress = iPoint;
    ListCell[iNext].dValue[0] = iStep;
    ListCell[iNext].dValue[1] = (double) nCnt;
    ListCell[iNext].dValue[4] = (double) iCurrentLabel;
    ListCell[iNext].returnPtr = execPointer;
    ListCell[iNext].nextProgPtr = nextProgPtr;
}
```

表 6.2　pbNext()のリスト

```
/*========================== 840702 === FORTRAN 77S ======= BNEXT
      SUBROUTINE BNEXT(TEXT,ICTRL)
  NEXT LOOP COUNTER.
*/
void pbNextFor(void)
{                                       // 2006-02-25
     Char ch;
     int iPoint,IX,nCnt;
     Word iNext, iUp, nType;
     String Name1, Name2;
//
     iNext=nextRegister;
     if (iNext == 0) {pbErrList(ErrString02); return;};
     ch = pbCharRead();
     if (cbtype != EOL) {
          if (cbtype != ALPHA)    {pbErrList(ErrString13); return;};
          pbName();               if (bErrorFlag) return;
          if (iKwrdCode != 5)     {pbErrList(ErrString01); return;};
          iPoint = iKwrdPoint + 4;
          IX=iKwrdID;
          if (IX == 0)            {pbErrList(ErrString01); return;};
          for (;;) {
               nType = ListCell[iNext].icType;
               Name1 = (String) kwName;
               Name2 = (String) ListCell[iNext].elementName;
               if ((Name1 == Name2) && (iVarType == nType)) break;
               iUp = ListCell[iNext].iMother;
               nextRegister = iUp;
               if (iUp == 0)           {pbErrList(ErrString01); return;};
               pbCellBack(iNext);
               iNext = iUp;
          };
     };
//
     iPoint = ListCell[iNext].elementAddress;
     nType = ListCell[iNext].icType;
     vBuff[iPoint] += ListCell[iNext].dValue[0];        // iStep
     nCnt = (int) ListCell[iNext].dValue[1];
     if (nCnt <= 0) {
          iUp = ListCell[iNext].iMother;
          pbCellBack(iNext);
          nextRegister = iUp;
          ch = pbCharRead();
          if (ch ==':') execPointer++;
          if ((cbtype != EOS) && (cbtype != EOL)) {pbErrList(ErrString13);};
          return;
     };
     ListCell[iNext].dValue[1] = (double) (nCnt -1);
          iCurrentLabel = (Word)ListCell[iNext].dValue[4];
          execPointer = ListCell[iNext].returnPtr;
          nextProgPtr = ListCell[iNext].nextProgPtr;
}
```

表 6.3　IF 文の処理 pbIfGo()のリスト

```
/*========================== 840702 === FORTRAN 77S ====== BIFGO
      SUBROUTINE BIFGO(TEXT,ICTRL)
  SELECT GOTO BRANCHES BY 'IF-THEN-ELSE-GOTO'.
*/
void pbIfGo(void)                // 2006-02-09
{
     Word ist, nLabel;
     int iTrue, iFound;
     Char ch;
     Char * chptr;
/* analyse IF condition */
     ist = pbCellTake(); if (bErrorFlag) return;
     iKwrdCode = 0;
     pbSet0(ist,1); if (bErrorFlag) return;
     pbAcc2(ist);
     iTrue =(int) ListCell[ist].dValue[0];     // false = 0
     pbCellBack(ist);
     if (iKwrdCode == 0) pbName(); if (bErrorFlag) return;
/* after If condition 'THEN' shall be followed */
     if (iKwrdID != KWD_THEN) {pbErrList(ErrString02); return; };
     if (iTrue != 0) {
          cbtype = EOS;              // continuation specified
          return;                         // statement1 will be executed
     };
/* SKIP TEXT UNTIL 'ELSE' IS FOUND */
     iFound = 0;
     execPointer--;
     for (;;) {
          ch = pbCharNext();
          if ((cbtype == EOL) || (cbtype == EOS)) return;
          switch (cbtype) {
               case EOL:              // fall through
               case EOS:return;
               case ALPHA:
                    if (ch != 'e') continue;
                    pbName();
                    if (bErrorFlag) return;
                    if (iKwrdID != KWD_ELSE) continue;
                    iFound = 1;
          };
          if (iFound == 1) break;
     }
/* ELSE statement */
     cbtype = EOS;                   // continuation specified
     return;                              // statement2 will be executed
}
```

7. グラフィックス処理

7.1　概説

最初はPrint Plotが応用されたこと

プリンタに文字を表示するとき、文字の組合せで簡単な図形を表示する方法があり、一般的に Print Plot と言われます。最も単純な応用は、活字自体の図形の特徴を生かす表示方法であって、E-mail でよく見るface markです。(^^) (>-<) <m(--)m> などで「笑い・困惑・謝罪」の意思表示に使っています。活字の濃淡を利用し、モザイク状に活字を並べてグラフや濃淡図を作成する方法も良く使われます。この場合には等幅フォントを利用し、文字幅と行間の尺度に注意が必要です。この方法は、レーザプリンタなどの作図方式の原点に位置するものです。解像度は、Print Plotの10 DPIに対して300 DPI以上が利用できます。コンピュータのモニタでは尺度がやや曖昧ですが、約100 DPIの精細度があります。

グラフィックスはデバイスに依存すること

図を表示して何かの情報を伝える場合、三つの要素を考えます。形・寸法・意匠です。文字も図形ですが、三つの要素は、字形・寸法・字体デザインです。一般的な図形は線で外形を指定し、線の種類・色・濃淡が意匠的な要素に当たります。数値計算の結果を図に表す目的でコンピュータグラフィックスを利用する場合には、意匠的な要素を低位に扱うことも許されます。しかし、形と寸法の扱いに注意深さが必要です。グラフィックスの結果を最終的に用紙（ハードコピー）に得る装置（デバイス）は、プリンタやプロッタです。その前にコンピュータの画面上でモニタします。また、そのモニタ画面だけで観察して済ます（ソフトコピー）ことは、無駄な印刷を節約するために行われます。そのため、プログラミングの立場からは、モニタ画面自体もグラフィックス装置の一つとして扱い、プリンタやプロッタに出力することは、デバイスの選択を切り換えることで扱います。

オブジェクト指向プログラミングの考え

コンピュータのOSがWindowsに代表されるようなGUIの方式を採用するようになって、デバイスを直接制御するプログラミングに代わって、モニタ上に擬似的なデバイスを表示し、そのデバイス相手に擬似的な制御をする方法になりました。シミュレーションの言葉を当てます。この擬似的なデバイスを総称してオブジェクトと言います。モニタ上に準備するグラフィックス装置は、Plain_Basic ではグラフィックス用子ウインドウを当て、その上に作図専用にビットマップ出力用のオブジェクト Image（コントロール、コンポーネントなどと呼びます）を載せます。子ウインドウの作業領域（クライアント領域）に描きだすこともできますが、一過性の作図ですので、再描画の機能が少し不便です。作図領域の寸法などの幾何学的性質をプロパティ、作図命令がメソッドと言われ、これらを制御することがオブジェクト指向プログラミングです。Plain_Basic のユーザレベルでは、作図はグラフィックスコマンドの形式で扱うように単純化し、オブジェクト指向プログラミングの実践はPlain_Basicのソースコードで消化してあります。グラフィックスコマンドは、基本的な作図ができるものに限定しました（表7.1）。

表 7.1　標準的なグラフィックスコマンド

コマンド名	引き数	説明
DPERAS	なし	グラフィックス画面を消去する
DPWIND	WX,WY,WW	対象画面をカメラ光軸が狙う中心座標と視野の横幅。 デフォルトは、DPWIND 0,0,640 としてある。 縦横比は、デフォルトで3:4である。
DPMOVE	X,Y	線を引かないでペン位置を設定する。 何も指定しなければ、以前に描いた図形の終端。
DPDRAW	X,Y	標準は、DPMOVEで指定した位置から線を引く。 線種、色などは、その前に指定した値が使われる。 デフォルトは、黒の細い実線。
DPCIRC	X,Y,R	中心座標X,Y、半径Rの円を描く。塗り潰しはしない。
DPOINT	X,Y	指定した座標に１ピクセルの点を打つ。
DPMARK	X,Y,IP	指定した座標に、IPで決めた番号の記号を描く
DPENSZ	IP	線の太さをIPで指定する。デフォルトは細線。
DPENTX	IP	実線（デフォルト）、破線、点線などの線種を指定
DPTEXT	X,Y,"text"	最初の文字の左下の座標をX,Yとして文字列を描く

7.2　基礎的なコマンド処理

擬似的なカメラを想定すること

グラフィックス処理は奥行きの深い題材です。一般的な初心者がグラフィックスを楽しむには、最低限の知識として、座標系の理解と、それを利用して位置と寸法を指定して作図するコマンドを覚える必要があります。Plain_Basic は、手書きの感覚でマウスを使って任意の図形を描くお絵描きソフトではありません。コンピュータのモニタ上に或る作図用の枠を準備します。これが Plain_Basic のグラフィックスウインドウです。これ自体も、モニタを部分的に使う図形であって、位置と寸法を変えることができます。そのとき、描きたい図形がいつもその枠に収まるようにするため、仮想のカメラで被写体を撮影し、そのフィルムを引き伸ばしてその枠に一杯に貼り付ける、という操作を考えます。モニタ上の図形の絶対寸法は変化しますが、図形は常に相似に拡大・縮小されて表示されます。

数学座標系を使うこと

被写体は、垂直な壁のような平面に在る図形とし、そこに数学座標系を考えます。ｘ軸は右、ｙ軸は上向きが正とします。コンピュータグラフィックスの用語ではこれを<u>世界座標</u>と言います。擬似カメラのファインダを覗き、図形を視野に納める世界座標の矩形範囲を決めます。これがグラフィックスの用語での<u>ウインドウ</u>です。この決め方には種々の習慣があります。コンピュータグラフィックス関係では、モニタの制御の関係があって、左上を原点としたピクセル単位で世界座標を表しています。またｙ軸は数学座標の習慣とは逆に、下向きを正としています。この約束はプログラマレベルでは使いますが、一般的なユーザ向きではありません。Plain_Basic のウインドウの定義は、擬似カメラの光軸が狙う世界座標(Wx,Wy)と、どれだけの横幅範囲(Ww)を視野に収めるか、の三つの数値で決めることにしました。ウインドウの枠は矩形ですので、幅だけでなく高さも指定するべきですが、縦横比をデフォルトで 3:4 に決めてあります。子ウインドウであるグラフィックスウインドウの縦横比は自由に変更できますが、このファインダの視野枠がウインドウ枠の中央に一杯に接するように、Plain_Basic が調整します。このコマンドが表 6.1 の DPWIND です。デフォルトの設定は DPWIND(0,0,640)としてあります。これは、世界座標の原点を狙い、(-320<x<320),(-240<y<240)の範囲が作図範囲になります。

作図の基本は線を引くこと

コンピュータグラフィックスのデバイスは、ペンを機械的に移動させて作図させるプロッタに出発しましたので、グラフィックスコマンドはその制御方式を引継いでいます。基本的な命令は、ペンを上げたままで移動させる命令(move)と、ペンを下ろして線を引く(draw)の二つです。これを組み合わせればすべての図形を描くことができます。しかし、ユーザレベルの便利さを考えて、矩形・円・多角形・簡単な曲線、さらには文字を書く、などのコマンドが工夫されています。ただし、Plain_Basic では、円を描くコマンドだけしか準備していません。矩形は簡単に作図できますので意図的に省きました。線の太さペンを変えることで対応しますが、コマンドは DPENSZ としました。実線・破線・点線などはペンの上げ下げの間隔を約束して線を引けば得られますが、コマンドとして DPENTX を準備しました。ただし、この制御は Windows API を間接的に利用していますので、ペンの太さを大きくすると破線・点線などの指定が無効になって、すべて実線で描いてしまいます。

デバイスの制御

グラフィックス装置は、コンピュータに対して外部装置ですので、ファイルの操作と同じように、デバイスの準備、接続のための open/close の手続き、その間に機械的な制御命令を使います。この全体がオブジェクト指向プログラミングに当たります。つまり Plain_Basic のソースコードのプログラミングでは、デバイスの準備はグラフィックス子ウインドウを使うことにあたります。作図命令の他に、プロッタの場合には用紙の取替がありますが、これは画面の消去コマンド DPERAS が当たります。作図成果の保存は、メニューの方で対応させます。ユーザレベルでは、これらのプログラミングを消化して使い易いコマンドで準備しました。この部分のソースコードを表 7.2 に紹介しておきます。

表 7.2　グラフィックス関係のソースコード（Form に直接描くバージョン）

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "FormCanvas.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormGraphics *FormGraphics;
//---------------------------------------------------------------------------
double xCwidth, yCheight;       // form window size
double xswidth, ysheight;       // user window size
double xsleft, ystop;           // user coordinates at top left
double wWx, wWy, wWw;           // keeps graphics window constants
double Gaspect0 = 0.75;         // default aspect ratio
int lCountSecond;
int currentGx, currentGy;       // used in dpMove etc.
//---------------------------------------------------------------------------
__fastcall TFormGraphics::TFormGraphics(TComponent* Owner)
     : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void dpWind(double xx, double yy, double ww)
{
     double asp, wx, wy;
     wWx = xx; wWy = yy; wWw = ww;             // keeps window constants
     xCwidth = FormGraphics->ClientWidth; // Canvas width in pixels
     yCheight = FormGraphics->ClientHeight;    // Canvas height in pixels
     asp = yCheight / xCwidth;                 // Canvas aspect ratio Y/X
     wx = ww;
     wy = Gaspect0 * wx;
     if (asp > Gaspect0) {
          wy = wx * asp;
     }else{
          wx = wy / asp;
     };
// Normal Position
     xswidth = wx;
     ysheight = -wy;
     xsleft = xx - wx / 2;
     ystop = yy + wy / 2;
}
//--------------------------------------------- DP-21 ----------------------
/* MOVE PEN POSITION WITHOUT DRAWING */
void dpMove(double xx, double yy)
{
     currentGx =int(xCwidth * (xx - xsleft) / xswidth );
     currentGy =int(yCheight * (yy - ystop) / ysheight );
     FormGraphics->Canvas->MoveTo (currentGx, currentGy);
}
//--------------------------------------------- DP-22 ----------------------
/* DRAW A LINE ONTO THE SPECIFIED POSITION */
void dpDraw(double xx, double yy)
{
     currentGx =int(xCwidth * (xx - xsleft) / xswidth );
     currentGy =int(yCheight * (yy - ystop) / ysheight );
     FormGraphics->Canvas->LineTo (currentGx, currentGy);
```

```
}
//---------------------------------------------------------------------------
void dPenSz(int ipensz)
{
int ipen;
     ipen = ipensz;
     if (ipen <1) ipen =1;
     if (ipen >10) ipen =10;
     FormGraphics->Canvas->Pen->Width = ipen;
}
//---------------------------------------------------------------------------
void dPenTx(int ipen)
{
     switch (ipen){
          case 0:
          case 1:FormGraphics->Canvas->Pen->Style = psSolid;break;
          case 2:FormGraphics->Canvas->Pen->Style = psDash;break;
          case 3:FormGraphics->Canvas->Pen->Style = psDot;break;
          case 4:FormGraphics->Canvas->Pen->Style = psDashDot;break;
          case 5:FormGraphics->Canvas->Pen->Style = psDashDotDot;break;
          default:FormGraphics->Canvas->Pen->Style = psSolid;
     };
}
//---------------------------------------------------------------------------
void dpMark(int ipen)
{
     // not specified in this verison
}
//---------------------------------------------------------------------------
void dpEras(void)
{
     TRect R;
     FormGraphics->Canvas->Brush->Color = clWhite;
     R.Left=0;  R.Top=0;
     R.Right= FormGraphics->ClientWidth;
     R.Bottom=FormGraphics->ClientHeight;
     FormGraphics->Canvas->FillRect(R);
}
/*==================================== DP-27 ============== DPCIRC
      SUBROUTINE DPCIRC(CIRCL)
  CIRCL(1)=X, CIRCL(2)=Y, CIRCL(3)=R
*/
void dpCirc(double xx, double yy, double rr)
{
     int X1,X2,X3,Y1,Y2,Y3;
//
     X1 = int(xCwidth  * (xx - rr - xsleft)/ xswidth );
     Y1 = int(yCheight * (yy - rr - ystop) / ysheight);
     X2 = int(xCwidth  * (xx + rr - xsleft)/ xswidth );
     Y2 = int(yCheight * (yy + rr - ystop) / ysheight);
     X3 = X2;
     Y3 = int(yCheight * ( yy - ystop) / ysheight );
     FormGraphics->Canvas->Arc(X1, Y1, X2, Y2, X3, Y3, X3, Y3);
}
//---------------------------------------------------------------------------
void dpPoint(double xx ,double yy)
{
     currentGx = xCwidth  * (xx - xsleft)/ xswidth;
     currentGy = yCheight * (yy - ystop) / ysheight;
     FormGraphics->Canvas->MoveTo (xx, yy);
     FormGraphics->Canvas->LineTo (xx, yy);
```

```
}
//---------------------------------------------------------------------------
void dpText(double xx, double yy, String text )
{
    currentGx = xCwidth  * (xx - xsleft)/ xswidth;
    currentGy = yCheight * (yy - ystop) / ysheight;
    FormGraphics->Canvas->TextOut(currentGx, currentGy, text);
}
//---------------------------------------------------------------------------
void __fastcall TFormGraphics::FormLoad(TObject *Sender)
{
    FormGraphics->Canvas->Pen->Mode = pmCopy;
    FormGraphics->Canvas->Pen->Color = clBlack;
    FormGraphics->Canvas->Pen->Style = psSolid;
    FormGraphics->Canvas->Pen->Width = 1;
    dpWind(0,0,640);
}
//---------------------------------------------------------------------------
void __fastcall TFormGraphics::OnResize(TObject *Sender)
{
    dpWind(wWx,wWy,wWw);
}
//---------------------------------------------------------------------------
```