J3W for Linux 入門講座

水谷純
http://www.nk.rim.or.jp/~jun/index.html

第8回(最終回)　J3Wのアセンブラ

これまでJ3C言語を使ったプログラミングに関して解説してきましたが、最終回の今回はJ3Wのネイティブ言語であるJ3Wアセンブリ言語のプログラミングを簡単に解説します。

これまでにも、J3C言語のソースは、j3cコマンドでコンパイル後、j3dasmでアセンブルされ、j3wで実行されることは見てきました。コンパイル後に生成されるJ3Wのアセンブリ言語とはいったいどんなものなのでしょうか。

■J3C言語とJ3Wのアセンブリ言語の比較

リスト1に示すコードcount1.j3cは、これまで見てきたJ3C言語によるプログラムです。実行すると、1～9までの数字を表示して終了します。

このcount1.j3cをj3cコマンドでコンパイルすると、count1.j3mというファイルが生成されます。count1.j3mは、テキストファイルでcatやlessコマンドで内容を表示できます(リスト2)。これを見ると、J3C言語のcount1.j3cがコンパイルされて、count1.j3mというJ3Wのアセンブリ言語のソースファイルが生成されることが確認できます。

count1.j3cの動作をアセンブリ言語で直接プログラムしたのがリスト3です。J3C言語のソースより短く、たった6行のプログラムで同じ動作をします。リスト3の動作を言葉で書くと次のようになります。

1．変数に1を代入
2．値を表示
3．変数に1を加算
4．変数が10より小さければ2．に戻る
5．終了

短い例ですからリスト3を詳しく見ていきましょう。1行目の「LOAD R1 1」では、R1というレジスタ(変数のようなもの)に1を格納しています。2行目の「label:」は「ラベル」と呼び、行の先頭部分にコロンを付けた名前を置いて、特定の場所(この場合2行目)に名前を付けています。「OUTNUM R1」はJ3C言語のNumber組み込み関数と同じで、R1の値を表示します。3行目の「INC R1」でR1の値を1つ増やし、4行目の「CMP R1 10」は、R1の値を10と比較します。内部的には「R1－10」を実行します。5行目の「BLS label:」は、直前の比較(演算)の結果が「小さい」ならば、ラベルの位置に分岐(この場合2行目)します。R1の値が10以上の場合は分岐しないで、6行目の「STPALL」を実行して終了します。

どうですか？　単に処理を順に並べているだけですね。「LOAD」や「BLS」という命令(ニーモニック)さえ覚えてしまえば、処理内容を日本語で箇条書きするのとあまり変わりません。

リスト3をアセンブルすると生成される

【リスト1】count1.j3c

```
class main {
      int INIT() {
            volatile int i;

            for (i=1; i<10; i=i+1) Number(i);
            Stop();
      }
      int RUN() {}
      int EVENT() {}
}
```

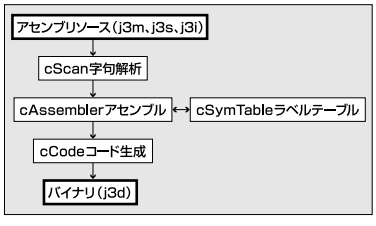
Column　j3wの内部構造の解説

■アセンブラj3dasm

J3Wのアセンブラ(j3dasm)は、アセンブリ言語のソースコードを2度読み込んで処理する2パスアセンブラになっています。ソースコードを1度だけ読んで(1パス)、分岐命令の飛び先となるアドレスのラベルが前方にあってアドレスが分からない場合、飛び先アドレスを未設定のままアセンブルを続行し、後になってラベルのアドレスが確定した時点で元に戻ってアドレスを設定する方法も可能です。この場合は、アドレスが未確定のまま残った命令の位置をすべて覚えておいて、最後に未確定のアドレスを修正する必要があり、処理が煩雑になります。j3dasmでは、1パス目ではアドレスが未確定でも無視してアセンブルを続け、ラベルが宣言されたアドレスを記録します。2パス目では、1パス目で分岐先のラベルのアドレスがすべて確定しているため、アセンブラはすべてのラベルの参照が可能となります。1パス目は、ラベルテーブルの作成だけが仕事になります。命令の長さが可変長であるため、実際にコードを生成していますが、2パス目で再生成しています。

j3dasmの簡単な構造を図Aに示します。cAssembler::assemble(ソースファイル名)を起点としてアセンブルを開始します。cScan::NextWord()が1単語ずつに分離してアセンブリソースを読み込みます。アセンブラは複数行にわたって処理する必要がなく、1行ごとに処理が完結するため、ラベルがあればラベルをテーブルに登録し、翻訳した命令と引数(オペランド)をcCodeに登録、という作業を繰り返しています。命令の翻訳は700行もある長大なswitch文1つが担当しています(素人にはお薦めできない;-))。最後にcCodeのバイナリを拡張子がj3dのテキストファイル(16進数)に出力してアセンブル終了となります。このcCodeはj3wが仮想CPUのコード領域として使用しているものと同じものです。　(水谷純)

【図A】j3dasmのクラス構造



count2.j3dは、8桁までの16進数値が並んだリスト4のテキストファイルです。これがいわゆる「マシン語」で、j3wはこれをメモリに数値として読み込んで実行します。

count1.j3cとcount2.j3sを実行してみると、どちらも一致することが確認できます(実行例1、実行例2)。

■仮想CPU

2001年10月号で「J3C言語は仮想CPUで実行される」ということを説明しましたが、この仮想CPUの構造を詳しく見ていきましょう。図1は、J3Wの中心となるCPUのレジスタとデータメモリです。

J3WのCPUは、同時に複数存在できますが、コードメモリ領域(プログラム領域)と共有メモリ領域は、すべてのCPUで共有されます。J3C言語のクラスのインスタンス1つにつき1つのCPUを使います。左側に数字が付いているレジスタがプログラムから直接読み書きできるレジスタです。

「プログラムカウンタ」は、CPUが実行する命令のコードメモリ中のアドレスを示しているレジスタです(図1の上側の矢印)。分岐命令は、このプログラムカウンタの値を書き換えることで分岐を実現しています。「コードメモリ」には「プログラム」として実行する命令(と引数)が並んでいます。コードメモリは複数のCPUが共有して使います。

「スタックポインタ」は、データメモリのアドレスを示しています(図1の下側の矢印)。PUSH命令を実行すると、スタックポインタから1が減算され、その位置のデータメモリに値を格納します。POP命令では、逆にスタックポインタの示すデータメモリから値を取得した後にスタックポインタに1が加算されます。PUSH命令とPOP命令を対応して使うことで、レジスタ内容の一時退避と復帰が簡単に実現できます。「ベースポインタ」はENTER、LEAVE命令で内部的に使用されます。コンパイラは引数と局所変数をLOADBP、STORBP命令を通してアクセスします。

「フラグレジスタ」は演算の結果に従って分岐するかどうかを決めるため、演算命令と分岐命令を内部的に使用します。演算結果が0かどうかを示すゼロフラグと演算結果が負であることを示す符号フラグの2bitを使っています。

RQ～R6レジスタまでの14個のレジスタは、

【図1】J3WのCPUの構造

【リスト2】count1.j3m

```
;+---------------------------------+
;|  This file was compiled by j3c.
;+---------------------------------+

      BRA     main:
main:
      CALL    main_INIT:
      CALL    main_RUN:
      DELPRC
main_INIT:
      ENTER   1
      LOAD    R4  1
      LOAD    R1  R4
      STORBP  R1  -1#
main_INIT_0:
      LOADBP  R4  -1#
      LOAD    R1  R4
      LOAD    R4  10
      CMP     R1  R4
      BLS     main_INIT_4:
      LOAD    R1  0
      BRA     main_INIT_5:
main_INIT_4:
      LOAD    R1  -1
main_INIT_5:
      CMP     R1  0
      BEQ     main_INIT_3:
      BRA     main_INIT_2:
main_INIT_1:
      LOADBP  R4  -1#
↘     LOAD    R2  R4
      LOAD    R4  1
      ADD     R2  R4
      LOAD    R1  R2
      STORBP  R1  -1#
      BRA     main_INIT_0:
main_INIT_2:
      LOADBP  R4  -1#
      LOAD    R1  R4
      OUTNUM   R1
      BRA     main_INIT_1:
main_INIT_3:
      STPALL
      LEAVE
      RETURN
main_RUN:
      RECEIV  RL
      CMP     RL  0
      BEQ     main_RUN_BODY:
      CALL    main_EVENT:
main_RUN_BODY:
      ENTER   0
      LEAVE
      THROW
      BRA     main_RUN:
main_EVENT:
      ENTER   0
      LEAVE
      THROW
      RETURN
```

【リスト3】count2.j3s

```
       LOAD    R1  1
label: OUTNUM  R1
       INC     R1
       CMP     R1  10
       BLS     label:
       STPALL
```

【リスト4】count2.j3d

```
40090000
1
98090000
8090000
19090000
a
25000002
63000000
```

【実行例1】count1.j3cの実行例

```
jm:~$ j3c count1.j3c
 J3C ver. 1.04 2001 Jun Mizutani
     Source File : count1.j3c
     Assembly File : count1.j3m

jm:~$ j3dasm count1.j3m
 j3dasm  ver. 1.42  2002 Jun Mizutani
     Source File : count1.j3m
     Object File : count1.j3d
       PASS 1
       PASS 2
     Assemble 52 steps

jm:~$ j3w count1.j3d
 J3W ver.6.44 2002 Jun Mizutani
 Execute file : count1.j3d
123456789
 [finished]
```

【実行例2】count2.j3sの実行例

```
jm:~$ j3dasm count2.j3s
 j3dasm  ver. 1.42  2002 Jun Mizutani
     Source File : count2.j3s
     Object File : count2.j3d
       PASS 1
       PASS 2
     Assemble 8 steps
jm:~$ j3w count2.j3d
 J3W ver.6.44 2002 Jun Mizutani
 Execute file : count2.j3d
123456789
 [finished]
```

プログラムから直接アクセスできます。「RQレジスタ」はプログラムで設定しますが、メッセージの送り先のCPUを識別するために使用します。メッセージの送り先は、RQの値をチェックして決定されます。

「RLレジスタ」は、LOOP命令の繰り返しのカウンタとして使用しますが、ループ命令外では汎用レジスタとしても使用できます。J3C言語では、メッセージの保持に使用しています。

RX、RY、RZ、RH、RP、RBの6個のレジスタは、XYZ座標値の設定や取得、ヘッド、ピッチ、バンクそれぞれの回転角度の設定や取得用として特定の命令で使用します。J3C言語で、引数をレジスタ変数で渡す組み込み関数と同じです。それらの命令を使わない場合には、汎用レジスタとしても使用できます。

R1、R2、R3、R4、R5、R6の6個のレジスタには特別な機能はなく、値の保持や演算用の汎用レジスタとして使用します。Cなどのプログラム言語のグローバル変数に相当する使い方をします。

【表1】J3Wにのみ用意されている命令

	命令	説明
演算命令（13個）	INC	指定レジスタに1を加算
	DEC	指定レジスタから1を減算
	ADD	レジスタにレジスタ、メモリ、定数を加算
	SUB	レジスタからレジスタ、メモリ、定数を減算
	MUL	レジスタにレジスタ、メモリ、定数を乗算
	DIV	レジスタをレジスタ、メモリ、定数で除算
	SHL	レジスタをレジスタ、メモリ、定数で左シフト
	SHR	レジスタにレジスタ、メモリ、定数で右シフト
	AND	レジスタにレジスタ、メモリ、定数をビット積
	OR	レジスタにレジスタ、メモリ、定数をビット和
	NOT	レジスタをビット反転
	CMP	レジスタをレジスタ、メモリ、定数と比較
	NEG	レジスタの符号反転
	命令	説明
分岐命令（12個）	BRA	無条件にジャンプ
	BEQ	演算結果が0か等しいならジャンプ
	BNE	演算結果が非0か等しくないならジャンプ
	BGR	より大きいならジャンプ
	BGE	より大きいか等しいならジャンプ
	BLS	より小さいならジャンプ
	BLE	より小さいか等しいならジャンプ
	BMI	演算結果が負ならジャンプ
	BPL	演算結果が0以上ならジャンプ
	LOOP	RLレジスタの回数分の繰り返し
	CALL	副手続きの呼出し
	RETURN	副手続きからのリターン
	命令	説明
レジスタ転送命令（15個）	PUSH	レジスタのプッシュ
	POP	レジスタのポップ
	CLEARG	汎用レジスタの一括クリア
	PUSHG	汎用レジスタのスタックへの一括プッシュ
	POPG	汎用レジスタのスタックから一括ポップ
	ENTER	スタックフレームの作成
	LEAVE	スタックフレームの解放
	CALLTB	共有メモリのテーブルで呼び出し
	LOAD	レジスタに値を設定
	LOADM	データメモリに値を設定
	XLOAD	レジスタ相対でデータメモリの値を取得
	LOADBP	ローカル変数をレジスタにコピー
	STORE	レジスタをデータメモリにコピー
	XSTORE	レジスタ相対指定でレジスタをデータメモリにコピー
	STORBP	レジスタをスタック上のローカル変数にコピー

■仮想CPUの命令

アセンブリ言語は「難しい」言語ではありません。x86などの一般的なCPUでも、OSを作るのでない限り、基本的に「演算」、「分岐」、「レジスタの操作」という3種類の命令に入出力関連の命令を使うだけでプログラムが作成できます。ただ、複雑な式や繰り返し構造を実現する場合でも、単純な命令を組み合わせる必要があるため「面倒な」言語ではあります。

J3C言語には約110の組み込み関数が用意されていて、それらはすべてj3wの命令に1対1で対応しています（J3W付属のj3c2j3w.html参照）。j3wには約150の命令があるので、40の命令がJ3Cには用意されていないことになります。J3Cに用意されていない命令は、表1に示すような演算命令、分岐命令、レジスタとスタックの操作命令です。

J3C言語では、これらの命令は式や文をもとに自動で生成されます。表1の命令は、ほとんど英単語そのままか、それを省略したものになっています。分岐命令は省略されすぎていて覚えにくいかもしれません。「BRAnch」、「Branch on Not Equal」、「Branch on LeSs」とモトローラ系のCPUの分岐命令の命名法を採用（特に理由はない）しています。

■j3dasmの仕様

アセンブリ言語で書かれたソースを、CPUが理解して実行できる数値列に翻訳（アセンブル）するプログラムを「アセンブラ」と呼びます。j3w用のアセンブラであるj3dasmは、アセンブラとしての基本的な機能だけを実装しています。アセンブリ言語のソースは図2の形式で記述します。行の先頭から空白を区切りとして、ラベル、命令、引数、セミコロンに続くコメントの順に並べます。正式には、命令はニーモニック（mnemonic）、引数はオペランド（operand）と呼びますが、ここでは「命令」、「引数」と呼ぶことにします。

ラベルはコロンで終わる文字列で最大31文字、命令とレジスタ名は必ず英大文字で表記します。行は、セミコロンか物理行末まで（1行当たり最大255文字まで）です。セミコロンがあると、セミコロン以降から物理行末まで読み飛ばします。ラベル名、定数名の英文字の大文字、小文字は区別します。

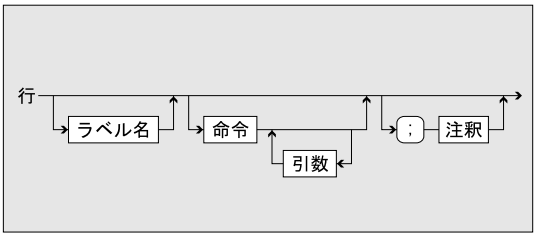
数値は0～9で始まれば10進数、「0x」で始まれば16進数を表します。文字定数は「'A'」のようにシングルクォートで囲みます。文字列は「"String"」のようにダブルクォートで囲みます。全角文字も可能です。「#」で終わる数値はデータメモリのアドレスを示します。

行頭から、空白やタブを入れずに「%インクルードファイル名%」と記述すると、指定したファイル名のファイルがその位置でインクルード（挿入）されます。このとき、パスを含めずにファイル名だけを指定すると、ソースファイルと同じディレクトリのファイルが使用されます。別のディレクトリのファイルを「%/home/jun/j3w/work/lib/library.j3i%」のように絶対パスで指定することも、「%../lib/library.j3i%」のように相対パスで指定することもできます。

インクルードファイルから、さらにファイルをインクルードする操作（「ネスト」または「入れ子」という）は、8レベルまで可能です。同じ名前のラベルが複数宣言されている場合はエラーにならず、参照した場合には行番号の若いラベルが使用されます。

命令に変換されない擬似命令として、定数

【図2】j3dasmの行形式



に名前を付けるDEFINE擬似命令があります。定数名には次の制限があります。

1．定数にラベルを定義することはできない
2．定数名にはニーモニックと同じ名前を使用しない
3．定数は使用前（ソースコード中の位置）に必ず定義する必要がある
4．定数の再定義はできない
5．定数名は最大31bytes

次に、DEFINE擬似命を使った有効な定数宣言の例を示します。

```
DEFINE DataMemory 1234#
DEFINE Constant   123
DEFINE HexConst   0x100
DEFINE Character  'A'
```

実際の使用例はj3w_script以下のディレクトリに用意していますので、参考にしてください。

■プログラミング

例によって、三角形を表示するプログラムを作ってみます（リスト5）。このプログラムでは、J3C言語と同様に「視点となる物体」と「三角形となる物体」の2つが必要となります。プログラムの先頭部分で、DEFINE擬似命令で定数に名前を付けけていますが、この部分では命令は生成されません。回転速度や三角形の大きさをここで変更できるようにしています。プログラムの実行は最初の命令（「Start:」ラベルの位置）から始まり、すぐに三角形用のCPUを生成します。その後、空の物体を生成して、位置、背景色の設定後、グラフィックウィンドウを開いてプログラムが終了するまで待ちます。三角形用のCPUは、GENPRC命令で指定されているように、「Triangle:」の位置から実行を始めます。頂点と面を登録した後に位置を設定し、1分間回転して終了します。

J3Wの仮想CPUは、三次元グラフィックス専用の命令を多く持っているため、計算や条件分岐をあまり使わないプログラムは、アセンブラを使うほうがむしろプログラムを短くすることができます。J3C言語のほうがプログラムは読みやすくなりますが、メッセージを細かく処理する必要がある場合ではアセンブリ言語のほうが楽な場合もあります。J3Wには、アセンブリとJ3Cの両方で書かれた多くのプログラムサンプルが付属しています。アセンブリ言語のマニュアルも付属しているので参考にしてください。

【画面1】triangle.j3sの実行結果

■終わりに

J3Wは、MS-DOS版を最初に公開してから6年、J3C言語の追加から3年が経過しました。開発環境はWindowsからLinux中心に変わりました。Windows版の開発（Delphi ver.2）もソースをLinuxに持ち込んで、diff／patchを使ってます。私のWebサイトでJ3WのWindows版とLinux版のページのアクセス比は2：1程度となっていて、OSの普及の比率から見てLinuxユーザーの健闘が目立ちます。Vector（記事末のResource［1］参照）からのダウンロード数は20：1ですが……。

今回でJ3Wの紹介を終了しますが、バージョンアップは私のサイト（［2］）でお知らせします。興味のある方は、たまに見に来てください。

【リスト5】triangle.j3s

```
      DEFINE Size  700      ; 三角形のサイズ
      DEFINE Size2 -700     ; 三角形のサイズ
      DEFINE Time  30000    ; 30秒間
      DEFINE Angle 28800    ; 10回転

Start:
      GENPRC  Triangle: 50  ; CPUを生成してTRIANGLEから実行開始
      GENOBJ  0  0          ; 視点用物体の生成
      CLEARA                ; RX～RBを0クリア
      LOAD    RX   -2000    ; 位置の指定20m後方
      SETPOS                ; 位置と角度を視点に設定
      SETEYE                ; この物体が視点を持つ
      BCOLOR  0x000         ; 背景色を黒に設定
      GRAPHM                ; グラフィックウインドウを開く
      WAIT                  ; メッセージ受信を待つ（何もしない）
      DELOBJ                ; オブジェクトの削除
      DELPRC                ; プロセスの削除

Triangle:
      GENOBJ  6  2          ; オブジェクトの生成　6頂点2面
      DEFPNT  0  0    Size  ; 頂点0　三角形の頂点座標を登録
      DEFPNT  0  Size Size2 ; 頂点1
      DEFPNT  0 Size2 Size2 ; 頂点2
      DEFPNT  0  0    Size  ; 頂点3　頂点0と同じ
      DEFPNT  0  Size Size2 ; 頂点4　頂点1と同じ
      DEFPNT  0 Size2 Size2 ; 頂点5　頂点2と同じ
      LOAD    R1 13         ; 面の色指定
      DEFPLN  R1 3 0 1 2    ; 面の定義
      LOAD    R1 14         ; 面の色指定
      DEFPLN  R1 3 5 4 3    ; 面の定義
      CLEARA                ; RX～RBのゼロクリア
      SETPOS                ; 初期位置と角度の設定
      ROTH    Time Angle    ; ヘッド回転
      ROTP    Time Angle    ; ピッチ回転
      DELOBJ                ; 物体の削除
      STPALL                ; 全プロセス終了
```

Resource

［1］ Vector
http://www.vector.co.jp/

［2］ Jun's Homepage
http://www.nk.rim.or.jp/~jun/

J3C 構文リファレンス ③

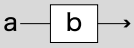
J3Cは、3Dアニメーションキット J3W のコンパイラです。J3Cは、Java風の文法を持つオブジェクト指向言語として設計されています。J3Cの構文リファレンスを掲載します。

構文図式の見方

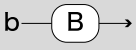
■構文規則（syntactic rule）

コンピュータ言語の文法は、構文規則によって決定されます。構文は、左から右へ、上から下へ矢印をたどって読んでいきます。

aはbである

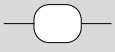


bは「B」である



構文規則の左側にある名前のことを、「非終端記号(nonterminal symbol)」と呼びます。また、`class`や`for`などの予約語、＝や＋などの記号で表される単語そのもののことを「終端記号(terminal symbol)」と呼びます。終端記号と非終端記号は、構文図では次のように表します。

終端記号
(terminal symbol)
囲みの中には予約語または記号が入ります



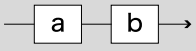
非終端記号
(nonterminal symbol)
囲みの中には、他で定義しているものが入ります

構文規則を順次適用していくと、すべての非終端記号は終端記号に行き着きます。

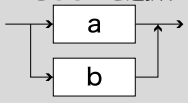
■構文図式の基本形

構文図式の基本形は次のようになっています。「連続」、「選択」、「繰り返し」を組み合わせて文法を記述します。

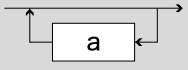
連続
aの後にbが続く



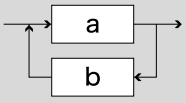
選択
aかbのどちらかを選択する



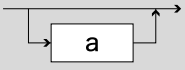
繰り返し
aの0回以上の繰り返し



繰り返し
aの1回以上の繰り返し。bは区切りの役割を持つ

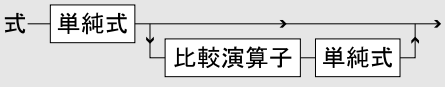


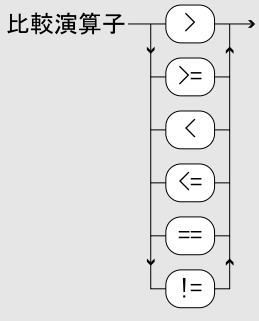
任意
aか何もなし



式

式は C、C++、JAVA に比べると演算子の種類が少ないですが、実用上は充分な種類を持っています。演算子の優先順位は、Pascalとほぼ同じです。





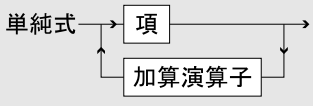
演算子の優先順位は、通常の四則演算（加減乗除）と同様、乗除は加減算より先に評価されます。優先順位は以下の順序で低くなります。

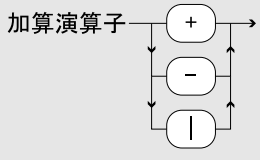
優先順位		演算子
高	1	- ()
	2	* / &
	3	+ - |
低	4	> >= < == !=

※優先順位が最も高い「-」は、「A = -A;」のように、符合を逆転する「単項マイナス」です。

単純式

四則演算の加減算を行います。演算子「|」はビット OR です。



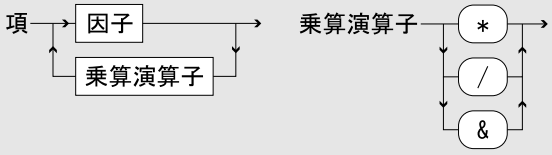


単純式Aと単純式Bは、表に示すような大小関係によって返り値が変化します。

式	返値	
	真（非0）の条件	偽（0）の条件
A > B	AよりBが小さい	AよりBが等しいか大きい
A >= B	AよりBが等しいか小さい	AよりBが大きい
A < B	AよりBが大きい	AよりBが等しいか小さい
A <= B	AよりBが等しいか大きい	AよりBが小さい
A == B	AとBが等しい	AとBが等しくない
A != B	AとBが等しくない	AとBが等しい

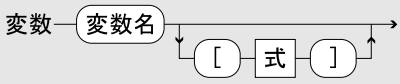
項

四則演算の乗除算を行います。演算子「&」はビットANDです。



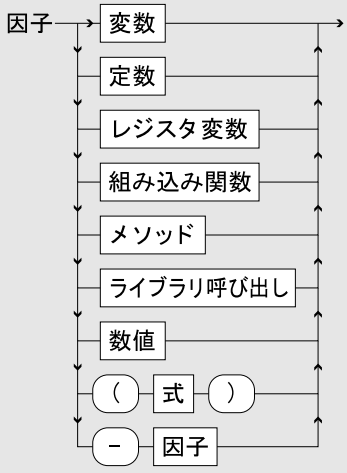
変数

変数には、変数名だけで使う「単純変数」と変数名に［式］を伴う「配列変数」があります。変数の型は32ビットの符号付整数のみです。



因子

因子は、変数、関数、数値のように、値を持つ式の最小の構成要素です。数値は10進数、16進数、キャラクタで記述できます。

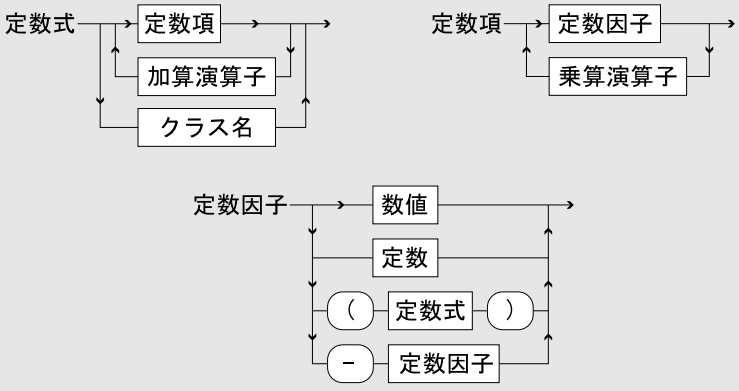


定数

定数の値を参照します。配列の要素を指定することもできますが、配列の添え字は定数式の必要があります。配列定数の添え字に変数を使用することはできません。ライブラリで宣言された定数はどのクラスからでもグローバルに参照できます。

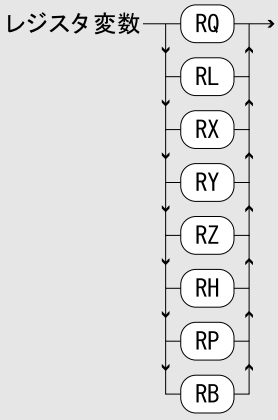
定数式

定数式はコンパイル時に値が決定できる必要があります。従って、実行時に値が変化する可能性のあるメソッドや変数を使用することはできません。変数のうち`final`宣言された変数（定数）は、実行時に値が変化しないため使用できます。`new`と`child`組み込み関数で、宣言済みのクラス名を置くことができます。



レジスタ変数

`RX`、`RY`、`RZ`、`RH`、`RP`、`RB`レジスタ変数は、組み込み関数のうち30種類程度の関数の呼び出し時の設定または返り値の取得に用いられます。また、クラスインスタンス生成時の`RX`、`RY`、`RZ`、`RH`、`RP`、`RB`レジスタ変数の値は生成されたインスタンスに引き継がれます。



このJ3C構文リファレンスは、2002年2月号から3回にわたって掲載しています。これまでの掲載分は、第1回は`diagram1.pdf`、第2回は`diagram2.pdf`として、今月号の本誌付録CD-ROMに収録しています。併せてご覧ください。

――お詫びと訂正――

3月号掲載の構文図（PDFファイルは`diagram2.pdf`）の「ライブラリ呼び出し」の図中に誤りがございました。次の図が正しいものです。お詫びして訂正いたします。