COBOL 利用技術のご紹介

COBOL コンソーシアム利用技術分科会
小林 純一(マイクロフォーカス株式会社 技術部)

第 4 回　算術演算の精度

1. COBOL による計算

過去40年来、COBOLは事務計算向き言語で Fortranは科学技術計算向き言語と言われてきています。これはもちろん、それぞれの言語がサポートする機能がそのような目的に設計されているからです。具体的にもっとも異なる点として、それぞれの言語における計算式・四則演算の機能の相違を一番に挙げることができます。

COBOLによる計算の特徴は、明示的な10進演算です。プログラマはそれぞれのデータ項目に対して、小数点上・下の10進桁数を明示的に定義することができ、これらの間での四則演算およびべき乗の計算を自由に記述できます。これは、COBOLという言語を特徴付ける便利で強力な機能です。

しかし、便利である反面、計算の結果に対するプログラマの責任は大きくなります。Fortran や Cのように言語として整数型と浮動小数点型しか持たない場合には話が簡単です。整数型は 2の補数形式で表現される 2バイト、4バイト、8バイトのレジスタ上で計算され、その範囲内を越えればオーバーフローが発生します。除算の剰余は切り捨てられます。浮動小数点型は、丸めの誤差があることを前提で使用するものです。大雑把に言えば、プログラマもコンパイラも計算結果の末尾の桁には責任を持たない世界です。

ところが、COBOLプログラマは、算術式の対象データ項目の桁数を自分で定義します。これは、どの桁で切り捨てや丸めが起きるかがプログラマの責任で決められるということです。

一方、3項以上の複合的な算術式の場合にCOBOLコンパイラが一時的に使用する中間結果では、COBOLコンパイラが適宜定めた桁数が使用されます。ANSI 2002 ではこれに関する規格化が完成しますが、現在のCOBOLコンパイラはそれぞれコンパイラのベンダーが定めた方式を使用しています。このために、コンパイラによって計算結果の精度に相違が出てくることがあります。

本稿では、このような話題を扱い、プログラミング上の注意点を解説します。

2. ANSI 85 COBOL の規定する 10 進演算

ここではまず、COBOLによる数字データ項目の定義のしかたと、それに対する操作をおさらいし、ANSI 85規格で規定されているCOBOLの計算規則について説明します。

2.1 数字項目の定義

COBOLではデータ項目に対して PICTURE文字列に “9” を使用して数字項目を定義することができます。これで項目の10進桁数が決まります。これとは別に、USAGE句を使用して、数字項目の物理的な表現形式を指定することができます。

2.1.1 USAGE DISPLAY

```
01  項目-1     PIC 9(4)V9(2)  USAGE DISPLAY.
```

は、小数点以上4桁、小数点以下2桁のデータ項目を定義しています。ANSI 85 COBOL規格では、小数点上下合計で 18桁までの定義を許しています。USAGE DISPLAY は、この項目の「用途が表示用」であることを示します。具体的にはこのとき、「項目-1」には 6バイト長の領域が割り付けられ、先頭4バイト分が小数点以上、末尾2バイト分が小数点以下をあらわすようになります。

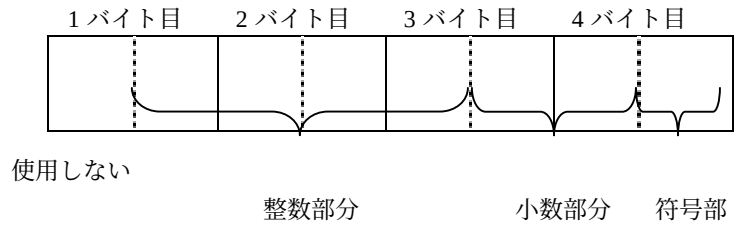
USAGE DISPLAYの項目に対しては、この他に SIGN SEPARATE句を使って符号の表現形式を定義することもできます。

2.1.2 USAGE PACKED-DECIMAL

```
01  項目-1     PIC S9(4)V9(2)  USAGE PACKED-DECIMAL.
```

は、小数点以上4桁、小数点以下2桁の符号付数字項目をパック10進形式で定義しています。パック10進形式の表現方法は、コンピュータのマシンアーキテクチャやCOBOLコンパイラの実現方式によって変わりますが、ほとんどの実装では、10進 1桁につき4ビットが使用されるようになります。

たとえば上記の「項目-1」では 4バイト長の領域が割り付けられ、以下のように使用されます。末尾バイトの後半の4ビットは、符号部として正負をあらわすために使用されます。



パック10進形式が用意された背景には歴史があります。コンピュータの主な用途が事務用のデータ処理であった時代には、コンピュータのアーキテクチャは現在のように 32ビットや64ビットのバイナリレジスタを使用する命令セットだけではありませんでした。アーキテクチャのレベルで10進演算向けに設計されたコンピュータもありました。

現在でもほとんどのメインフレームは、ハードウェア命令として10進演算をサポートしており、パック10進形式の四則演算を 1命令で実行します。メインフレームのCOBOLコンパイラは、このようなハードウェア命令を駆使することによって COBOLの10進演算を高速に実行しています。

2.1.3 USAGE BINARY

```
01  項目-1    PIC S9(4)V9(2)  USAGE BINARY.
```

は、小数点以上4桁、小数点以下2桁の符号付数字項目を 2進形式で定義しています。数値の2進表現は、コンピュータのアーキテクチャに依存するため、機種、OS、コンパイラによって異なっています。

基本的にはいわゆる「2の補数」表現によって整数値を表現します。小数点以下の桁数がある場合にはその分の「下駄履き」をして表現します。上記の場合ですと、小数点以下2桁がありますので、実際に表現したい値を 100倍した整数値を 2の補数で表現します。

コンピュータの機種に依存する相違としては、2進数値をメモリに格納する際のバイト順序の問題があります。多くのメインフレームでは、上位のビットを最初のバイトに格納する「ビッグエンディアン」方式です。たとえば数字の 1 を4バイトの2進で表現すると X'00000001' というバイトならびになります。これに対して、Intel x86 系のCPUではこの逆の「リトルエンディアン」方式です。数字の 1 を4バイトの2進で表現すると X'01000000' です。ところが COBOLコンパイラによっては、メインフレームとのデータ互換性を保つために、Intel CPUであってもビッグエンディアンで格納するものがありますから注意が必要です。

また、PICTURE句で指定された10進の桁数に応じて、必要となるバイト長が変わりますが、この長さを決定する規則もCOBOLコンパイラによって相違があります。しかし、多く

は桁数が増えるのに応じて 1、2、4、8、16バイトを使用するような方式を取っています。3バイト、7バイトといった半端な長さの 2進データ項目を扱えるコンパイラもあります。

2.1.4 その他の USAGE

以上、DISPLAY、PACKED-DECIMAL、BINARYという3つのUSAGEを見てきましたが、これらはすべて ANSI 85規格で定められた構文であり、すべてのCOBOLコンパイラがサポートしているものです。通常のCOBOLコンパイラはこの他に COMP、COMP-1、COMP-2、COMP-3、COMP-4、COMP-5 など、さまざまなUSAGEをサポートしており、実装方法にも若干の相違がありますので、プログラミングにおいては注意が必要です。

とくに ANSI 85規格の範囲外の USAGEを使用すると、場合によっては PICTURE文字列で定義した10進桁数と無関係な扱いができたり、後述するCOBOLの桁よせ規則が適用されないことがあります。

2.2 演算のプログラミング

数字データ項目と数字定数を使用して、これらの算術演算をプログラミングできます。ANSI 85規格では、加減乗除の四則演算とべき乗が規定されています。DIVIDE文のREMAINDER指定で除算の剰余を得ることもできます。ビット単位の論理演算は規格では定められていません。

COBOL文としては、ADD文、SUBTRACT文、MULTIPLY文、DIVIDE文があり、それぞれ加算、減算、乗算、除算ができます。また、A * (B + 1) / (C – 1) のような複合的な書き方を含む算術式を、COMPUTE文、比較条件などで使用することができます。

2.2.1 二項演算の規則

ANSI 85規格では、A + B のようなニつのデータ項目または定数間の演算結果は完全に定義されており、桁あふれがない限りCOBOLコンパイラによる相違はありません。ニつの数値が純粋に数学的に計算され、その結果が受け側のデータ項目のPICTUREで定義される桁数に桁落としして格納されます。ROUNDED指定を明示的に書かない限り四捨五入されることはなく、必ず切り捨てされます。

COBOLプログラマはこのことを熟知していますから、これを意図的に使って必要なだけの小数点以下精度への切り落としを行ったりします。以下のようなプログラミングは、数学的には一見不可思議に見えるかも知れませんが、下2桁を切り落とすために意図的に書かれたものです：

```
COMPUTE A = B / 100.
COMPUTE B = A * 100.
```

2.2.2 桁あふれの扱い

計算結果が、受け側のデータ項目の桁数を越える場合（オーバーフロー）は、ON SIZE ERROR指定に書かれたエラー処理が実行されます。ON SIZE ERROR指定が書いていない場合、ANSI 85規格は何も規定していません。コンパイラによって実行時エラーでアベンドしたり、数学的に正しくない結果を返してそのまま処理を続けたりしますので注意が必要です。

特別な場合として、ゼロによる除算も桁あふれ条件に合致します。この場合も ON SIZE ERROR指定が書かれていないと結果はコンパイラに依存します。

2.2.3 三項以上の演算

三項以上の演算がなされるのは、複合的な算術式の場合だけではありません。たとえば、

```
ADD A B C TO D.
```

は、

```
COMPUTE D = D + ( A + B + C ).
```

に相当します。

三項以上の演算に対して、コンパイラは計算の中間結果を格納するための暗黙的なデータ項目を自動的に作成します。上の例では、プログラム上には現われないデータ項目 T1、T2 が暗黙的に用意され、

```
COMPUTE T1 = A + B.
COMPUTE T2 = T1 + C.
COMPUTE D = D + T2.
```

の順番で計算がなされます。

3. 中間結果の桁数の相違による結果の相違

COBOLプログラム中に複合的な算術式を書くと、コンパイラが演算の中間結果にどれだけの精度を与えるかによって、プログラムの実行結果に相違が出る場合があります。

10進演算を高級なハードウェア命令としてサポートするメインフレームと異なり、マイクロプロセッサを使用するオープンシステムでは、10進演算はほとんどの場合ソフトウェアによって実行されます。コンパイラの作成者は、常にもっとも高速に計算が実行される

ことを念頭におきますので、マシンアーキテクチャが異なれば最適な実装方式も自ずと異なってきます。

例として以下のようなプログラムを考えます。

```
WORKING-STORAGE  SECTION.
01  ITEM-1       PIC 9V999.
01  ITEM-2       PIC 9   VALUE 1.
01  ITEM-3       PIC 9   VALUE 7.
01  ITEM-4       PIC 9   VALUE 3.
    COMPUTE      ITEM-1  =  ITEM-2 / ITEM-3 * ITEM-4.
```

数学的には 1 ÷ 7 × 3 = 0.428571428... ですから最終結果の ITEM-1 は 0.428 になるように思えます。しかし、ほとんどのメインフレームでは、この結果は 0.426 になります。これは、最終結果を受け取る ITEM-1 が PIC 9V999 であるため、コンパイラが割り当てる中間結果に小数点以下 3桁しか取られないことによります。上の計算は、

```
WORKING-STORAGE  SECTION.
01  ITEM-1       PIC 9V999.
01  ITEM-2       PIC 9   VALUE 1.
01  ITEM-3       PIC 9   VALUE 7.
01  ITEM-4       PIC 9   VALUE 3.
01  T            PIC 9V999.
    COMPUTE      T  =  ITEM-2 / ITEM-3.
    COMPUTE      ITEM-1  =  T * ITEM-4.
```

のように計算されます。従って最初の COMPUTE文の結果は 1 ÷ 7 = 0.14285714... の小数点以下 4桁以下を切り落として 0.142 となってしまいます。これに 3 をかけて結果は 0.426 となります。

ANSI 85規格では、中間結果に割り当てる桁数の取り方はコンパイラの作成者が自由に決めて良いが、その取り方の規則を公開しなければならないことになっています。この機会にお使いのCOBOLコンパイラのマニュアルを読んでみてください。必ずどこかに中間結果の精度に関する記述があるはずです。

上の例で説明したのは、多くのメインフレームのCOBOLが採用している方法です。詳細な説明は各コンパイラのマニュアルに記載されていますが、簡単に言えば「最終結果の受け取り側項目の小数桁数に合わせる」方法です。しかしオープン系のCOBOLではもっと高い精度を許容するものもあり、上の例で結果が 0.428 になるものもあります。

メインフレームからのダウンサイジングを考えている方からときどき、「パソコンでメインフレーム並みの高い精度のある計算ができるのか」というご質問を受けることがありますが、単に精度だけを言えば、上の例のようにむしろメインフレームの方が低いこともあります。同じプログラムをパソコンのCOBOLで再コンパイルすることによって、メインフレームより正確な計算結果が返ることもあります。しかし、精度が高くなることによって計算結果が違ってしまうと、ダウンサイジングの結果、既存システムとの非互換が発生します。このため、COBOLコンパイラによっては、高精度の計算のほかに互換性オプションを用意しており、メインフレームと同じ精度での計算結果も選択できるようにしています。

4. プログラミングの作法

以上のようなことがらを踏まえて、COBOLプログラミング中での算術計算について、プログラマが注意しなければならないことについてまとめてみます。

4.1 コンパイラの限界を知る

これまで説明してきたように、算術計算の書き方によって、どのCOBOLコンパイラでも同じ結果が返るものと、コンパイラによって異なる結果が返る可能性のあるものがあります。本来プログラマはこのことを意識するべきです。

一般に、加減算と整数の乗算、および剰余切り捨ての整数除算の場合には、このような問題はありえません。実際、通常の COBOLプログラムではほとんどはこのような問題の無い計算だけです。

問題は、3項以上の複合演算で、中間結果の小数点以下桁数が、最終結果の精度を越える場合にのみ発生します。この問題に対する対策は以下の3段階がありえます：

(1) ANSI 85規格が規定する範囲内でのプログラミング

2項演算しか使用せず、複合的な演算については、すべての中間結果に対して自分で必要な桁数分を明示的に定義します。こうしておけば将来コンパイラがバージョンアップされても、その結果が変わることは決してありません。しかし、プログラムの読みやすさが犠牲になります。

(2) コンパイラの仕様を理解した上でのプログラミング

3章で説明したように、各COBOLコンパイラは中間結果の精度の取り方を公開しています。従って、プログラマは自分のコーディングの結果、どのような精度の値が返ってくるかを事前に知ることができます。これを意識して設計しておけば、(1) のような

面倒なコーディングの必要はありません。ただしこの場合、将来コンパイラが変わった場合には必ずしも同じにならないことを覚悟しなければなりません。
この対策のひとつとして、複合演算の中で演算の順序に注意するということがあります。例えば 3章で例示した

```
COMPUTE      ITEM-1  =  ITEM-2 / ITEM-3 * ITEM-4.
```

という計算は、

```
COMPUTE      ITEM-1  =  ITEM-2 * ITEM-4 / ITEM-3.
```

のように除算を最後に移動することによって、ほとんどの場合コンパイラによる非互換を回避することができます。

(3) 一切気にしない

浮動小数点演算を使った場合のように、ある程度の精度でしか結果が保証されないことを前提としてコーディングします。業務によってはこれでかまわないものもあるでしょう。

4.2 桁あふれ対策

2.2.2 で述べましたが、算術演算で桁あふれが生じた場合、ON SIZE ERROR指定が書かれていない場合の結果はコンパイラの実装によって異なります。しかし、対策として ON SIZE ERROR指定を書くことは必ずしも正しいものではありません。
COBOLで算術演算を使用する場合、ほとんどは結果の桁数の上限を予測することができます。従って正しい作法は、必要な桁数を自分で宣言し、桁あふれが絶対に起こり得ないようにすることです。

4.3 コンパイラの最適化を信じる

コンピュータアーキテクチャを知り尽くしたプログラマは、さまざまなテクニックを駆使して高速に実行されるコードを書こうとします。例えば、以下のコードは除算を使用することなしに数字項目の 1の位の数字を取り出しています。

```
WORKING-STORAGE  SECTION.
01  ITEM-1       PIC 9999.
01  ITEM-2       REDEFINES ITEM-1.
    05  ITEM-3   PIC 999.
    05  ITEM-4   PIC 9.
```

```
MOVE       1234  TO   ITEM-1.
DISPLAY   “ITEM-1 の1の位は “  ITEM-4.
```

このようなコーディングは､時としてアーキテクチャに依存するものになってしまうことに注意するべきです。とくに REDEFINES は、データ項目のメモリ上の配置に依存したコードになりますので、処理効率を目的とした使用は危険です。

COBOLコンパイラもさまざまな最適化を行っており、ストレートにプログラミングすれば､プログラマが意識することなしに効率的なコードを生成してくれます｡コーディングテクニックを駆使することで､かえって読みにくく保守性の低いプログラムにするよりも､プログラムが処理したいことを論理的に正しく記述することを心がけるべきです。

－　第4回　終　－