COBOL を学ぶ意義
～プログラミング言語の現状をふまえて～

COBOL コンソーシアム会長　高木　渉　（株式会社日立製作所）

1. プログラムの書き方の多様性

少し古い本で，「プログラム書法」という名著[1)]がある。1970 年代に世の中に出回っていたさまざまな教科書に載っているプログラムを取り上げ，良くないところを改善してみせては，良いプログラムを書くための教訓を抽出している。

「プログラム書法」の 2 章の冒頭に FORTRAN のスパゲッティープログラムが掲載されいる。10 行しかないのに GOTO 文が 6 つもあって，やっていることがすぐには理解できない。続く本文で，このプログラムは変数 X，Y，Z のうち，一番小さいものが変数 SMALL に入る，と説明があり，次の改善例を示している。

```
SMALL = X
IF (Y .LT. SMALL) SMALL = Y
IF (Z .LT. SMALL) SMALL = Z
```

巧みである。10 行が 3 行になり，GOTO 文は消えている。場合分けをする元のスパゲッティープログラムとは論理構造が異なり，次へ次へと制御が移るうちに最小値が求まる仕掛けになっている。

この本をさらに 50 ページほど読み進むと，先のスパゲッティープログラムの場合分けの論理構造はそのままに，PL/I に用意されている IF-THEN-ELSE の構文を使って書き直している。

```
IF X >= Y THEN
  IF Y >= Z THEN
    SMALL = Z;
  ELSE
    SMALL = Y;
ELSE
  IF X >= Z THEN
    SMALL = Z;
  ELSE
    SMALL = X;
```

その上で，このプログラムは，IF 文の条件判定の中にさらに IF 文が入れ子になった「繁りすぎた木」になっていて分かりにくいと評している。確かに，場合分けの構造は，IF-THEN-ELSE という構文のおかげで明快になったが，結局何がしたいのか，意図が透けて見えない。

書籍を離れて別の工夫をしてみる。変数 SMALL に値を入れる変数の候補は 3 種類しかないのだから，3 つの動作のどれかに落ちるように場合分けする。ここでは COBOL で書いてみる。場合分けには EVALUATE 文を使う。

```
EVALUATE TRUE
WHEN X < Y AND X < Z
   MOVE X TO SMALL
WHEN Y < X AND Y < Z
   MOVE Y TO SMALL
WHEN OTHER
   MOVE Z TO SMALL
END-EVALUATE
```

今度は条件判定が入れ子にならず，繁りすぎた木ではない。動作別のまとまりで条件をくくっているので，正しく意図が実現されていることを確かめるのは難しくない。

このように，3 つの変数から一番小さな値を見付けるという単純な処理でも，何種類もの方針でプログラムが書ける。プログラムを書く局面は，書き方の工夫ができて創造的だが，プログラムを読む局面は，どんな書き方が出てくるか予想がつかない状態で臨まなければならないことを意味する。

2. プログラムのライフサイクル

プログラムは，使い捨てでなければ，何年も，何十年も使われ続ける。プログラムを作り始めてから使われなくなるまでのライフサイクルの中で，最初の開発期間は短く，続く保守期間は長い。誤りの修正や部分的な改造をしていく保守では，既存のプロ

グラムを読んで理解できなければ手が出せない。

長い保守期間を考えれば，プログラムを書くときに重視すべきなのは読みやすさである。難しい処理でもないのに，能力に溢れた人にしか理解できないような疑ったプログラムは駄作である。

3. プログラムを読むという知的作業

プログラムを読むとはどのような作業だろうか。本稿では，ソースプログラムのうち，100 行くらいの部分を，ほぼ予備知識なしで読む場面を考える。

このような狭い範囲のプログラムを読む作業は，専ら，変数の値の変化を思い浮かべながら，頭の中でプログラムの実行をトレースしていき，処理内容を理解する作業となる。

多くのプログラミング言語で，プログラムの実行は，メモリ上のデータ値を変化させながら進む。また，変数の値を判定することで，プログラムの流れが変わる。

例えば，本稿の最初に示した 3 行の FORTRAN プログラムなら，それまでに調べた変数の中での最小値を変数 SMALL に格納しながら次へ次へと進む。本稿の 2 番目と 3 番目のプログラムなら，条件が真になる箇所にプログラムが進み，変数 SMALL に値を設定したら，それ以外の箇所は実行しない。

しかし，トレースできただけでは理解したことにはならない。さらに抽象化が必要である。冒頭のプログラム例なら，変数 X，Y，Z のそれぞれの値を与えられ，変数 SMALL に設定される値を言い当てるだけでは理解としては不十分である。「3 つの変数のうちの最小値を求める処理である」と，ひとまとまりを抽象化して，ようやく理解したことになる。

抽象化して，3 つの変数の最小値を求める処理だと捉えていれば，もっと大きなプログラムの一部にこの処理が書かれたときにも，周りの処理との関係で理解できる。プログラムを予備知識なく読んで理解する作業では，書かれたプログラムから部分部分を抽象化し，さらに抽象化の部分を組合せて，より大きな単位に抽象化していく。

4. プログラムの読み易さ

実行をトレースすることからして困難なプログラムというのは存在する。処理に使われているアルゴリズムが理解し難いというのでなければ，原因は，読みながら頭の中でメモしておかなければならない事象が多いことに求めることができる。

(1)値の変化を同時に意識しなければならない変数の数が多い

(2)変数の値を変化させる箇所と，その変数を参照する箇所があちこちに散らばっている

(3)実行の流れの向かう先が多い

プログラムの中のある処理部分を読んでいるときは，その部分に集中していたいし，他の部分は考慮の外に置きたい。しかし，(1)のように変数が多ければ，それぞれの値を追い続けるのは大変である。(2)のように変数の値が広範囲に影響したり，(3)のように制御が広範囲に飛んだりするなら，考慮の範囲を広げなければならない。

プログラムを読むときに，頭の中で記憶するメモの数が少なくて済むためには，独立した小部分が，独立を保ちながら組合わさっていればよい。

構造化プログラミングやオブジェクト指向プログラミングというのは，独立した部分を作る方法であり，それらを組合せる方法であるという見方もできる。また，定石やパターンと呼ばれるプログラムの組み方も同じような目的で使える技術である。自然言語の英語でさえ，一般的な英文の 7 割以上はコロケーションからなっているという研究があるということであり[2]，まったく新しい表現の使用頻度は少ない。プログラミングでも，パターンという先人の知恵に沿って書けば楽に書けるし，誤る可能性も低くなる。プログラムを読むときに，特定のパターンだと思って読めれば理解の助けになる。

5. 硬直化するプログラム

初めは分かりやすかったプログラムも，機能を追加したり，誤りを修正したりと，変更を繰り返すうちに，いびつになってくる。いびつになったプログラムは変更しにくい。プログラムを読んで理解するのも，適切な変更箇所を特定するのも，確認のためのテストケースを用意するのも難しくなる。こうなると，一見簡単な機能追加でも，複雑な作業となり，思いのほか工数が掛かる。

プログラムが硬直化していく原因に，必要最低限の変更を繰り返すことがある。変更を最小限にする方針は一面で正しい。変更していない部分に誤りが入り込む可能性を下げることができる。

プログラムの硬直化に万能薬はない。むしろ，読みにくいプログラムはどうしても存在し，それを読

んで改変していくという仕事も存在するということを受け入れるべきなのだろう。

プログラムの硬直化は，2つの側面に分かれる。

⑴処理の組み立て方がつぎはぎになる

⑵扱うデータの構造がつぎはぎになる

このうち，データ構造の硬直化については，体験しないと実感しにくい。

業務プログラムでは，数百種類ものデータを扱うことも少なくない。ビジネスルールの中核部分は変化しなくても，少しずつ新しいことを取り入れていく。新しく区別が必要になったり，活用したいデータの種類が増えたりする。しかし，ファイルなど，永続的に保存するデータでは，構造の変更は簡単ではない。将来の変更を見越して，あらかじめデータの持ち方をじっくり設計しておく必要がある。

一方で，プログラムの振舞いは変更せずに全体を書き直して整理してしまうリファクタリングという手法がある。オブジェクト指向の性質を活用する。ただ，今後どのどの程度多くのプロジェクトにリファクタリングが浸透していくかは未知数である。

6. プログラミング言語 COBOL の設計方針

COBOL は，1959 年に設計が始まって，1960 年に仕様書が公開されたプログラミング言語である。当時は，プログラミングの一般的な概念として，構造化もオブジェクト指向も確立する前と考えてよい。それどころか，ようやくコンピュータのハードウェアの命令語を直接書くのではなく，FORTRAN のような人間向きの言語が出てきた黎明期である。

こうした時期に，アメリカで，コンピュータのユーザ企業，軍，政府機関，研究機関，コンピュータメーカー等から人が集まり，後に COBOL と名付けられるビジネス向けプログラミング言語の仕様を開発する委員会を作った。その動機が興味深い。当時，幾つかのビジネス向けの言語の実装が出始めていた。あるメーカーのハードウェア上で稼働するプログラムを，他のメーカーのハードウェアに移植しようとすると，言語が異なるので，再プログラミングすることになる。これは問題だということで，共通の言語を作ろうとしたのであった[3)]。

3 点，重要な点を挙げる。

⑴英文として読めるプログラムを目指した。

⑵異なる機種のコンピュータ間で，プログラム資産を移行できるようにすることを目指した。

⑶処理の記述とデータの宣言とを分離した。

6.1 英文として読めるプログラム

簡単な英語を使ってプログラムを書くという COBOL の設計方針を実装することは，当時簡単ではなかったに違いない。それでも，自然言語で読めるプログラムになることを目指した。

この方針は，成功していない面もある。プログラムの記述に使う構造は，英語の文書の構造とは異なる。例えば，構造化プログラミングでは，枠構造を使う。逐次，反復，分岐という制御の流れのそれぞれ開始位置と終了位置を示して枠を作り，これらを入れ子にする。英語にはこのような構造はない。COBOL にも最初はなかった。1985 年制定の国際規格で，枠構造の終わりを示す，END-IF というような人工的な語を導入して枠を表現できるようになった。

しかし，英語表現にこだわったことで，一つの文に処理を詰め込めるようにはしない，という COBOL の文化を今日まで受け継いでいる。

少ない記述量で済むプログラミング言語で書く方が保守しやすいという言説を聞くことがある。確かに，適切な記号によって簡潔に書ける方が理解し易い。しかし，紙一重ではあるが，詰め込んで書けるということなら，これを読む局面では，たった一行の中で起こる多くの振る舞いをイメージすることを強いられる。誰でも実行をトレースできるという訳にはいかなくなって，却って保守しにくくなる。ある程度，記述がバラけている方が分かりやすい。

6.2 異なる機種間で移行性のある言語

COBOL の初期の設計で，プログラムを他のプラットフォーム（コンピュータハードウェアやオペレーティングシステムなど）に移行しても稼動できることを目指したのは，今振り返ると，有効な方針だったと評価できる。プラットフォームは，時間と共に旧式となり，提供元のベンダのサポートがなくなってしまうなどの理由で，いずれ新しいものに更新しなければならなくなるからである。

COBOL は，最初の設計方針の一つに，互換性を掲げた言語である。COBOL のソースプログラムを構成する 4 つの部（見出し部，環境部，データ部，手続き部）のうち，環境部がプラットフォーム固有の仕様差異を吸収する役目を担っている。現実には環境部だけでは差異を吸収しきれないのであるが，

実際に様々なプラットフォームで，ほぼ仕様の共通する COBOL が使える。

また，データ部は，機種に依存しない形でデータ項目（変数）の特性を宣言する。数値を表す変数なら，基本的に十進数の桁数と固定の小数点位置を指定する。こうして宣言する変数は，多くの場合，COBOL の言語処理系がソフトウェア的に実現して，どの機種でも共通に使えるようにしている。

6.3 処理とデータの宣言との分離

COBOL のソースプログラムは，4 つの部に分かれている。中でも，変数を宣言するデータ部が，処理を記述する手続き部と分離していて，データ構造の全体像を見通しやすく，データ構造をじっくり考えることを促す。長期にわたって使用し，改変を繰り返すプログラムを作るなら，データ構造をじっくり検討し，プログラムが硬直化していくことを予防する必要がある。

COBOL では，データのフィールドを並べてレコードとし，データの永続的な保管場所であるファイルを，レコードを単位として扱う言語仕様が組み込まれている。COBOL でデータ構造を考えるのは，ファイルの持ち方を考えることに近い。また，COBOL のレコードは，リレーショナルデータベースの属性（表における列）にも似ており，親和性がある。

7. COBOL プログラミングを経験する意義

COBOL は 1960 年代から使われ続け，膨大な量のプログラムが現役で活躍している。技術的に安定期に入ったためか，情報は流布していない。しかし，次の表のように，国内の調査で，Java に続いて COBOL が使われていることが分かる。

同じ文献に，COBOL が新規の開発プロジェクトで使われていることが分かる統計もある。他の言語で書き直していない既存の COBOL 資産があるから仕方なく，というような後ろ向きの理由ばかりで COBOL を使っているのではない。

表　開発に用いた言語

第 1 回答だけを集計
プロジェクト数 : 2,417/2,584 件

Java	25.4%
COBOL	16.8%
VB	14.1%
C	11.8%

出典：(独)情報処理推進機構　ソフトウェア開発データ白書 2010-2011

長期間稼働を続け，長期間保守していくようなプログラムの開発・保守を想定して，COBOL のプログラミングを経験していることは，他の言語での開発プロジェクトに入っても，役に立つはずである。

COBOL で経験できることは他の言語でも経験できる。そうは言っても，結局，使用するツールは思考を誘導する。プログラムを詰め込んで書けることを特徴とする言語を使って詰め込まない努力をするのも，変数の宣言が不要な言語で多種類のデータを扱うプログラムのデータ設計をするのも，目的に沿わないツールを選択している。

同じ理由で，一つの言語だけを経験するより，系統の異なる他の言語でのプログラミングにも触れて，考え方のいろいろなをクセを経験していると，引き出しが増える。

8 実務を意識したプログラミング学習

将来，プログラミングに携わる場合に，意識して経験して欲しいことを列挙してみる。

(1)プログラムを書いている本人以外の人に分かり易いプログラムを書くようにしよう。
(2)多くの種類のデータを扱うプログラムで，処理を書く前に，データを設計してみよう。
(3)他人のプログラムに，最小限の変更で処理を追加してみよう。
(4)他人のプログラムを，全体の動作は変えずに，自分のやり方で書き直してみよう。
(5)系統の異なる複数の種類のプログラミング言語でプログラミングしてみよう。

1) Brian W. Kernighan, P. J. Plauger 著，木村泉訳 : プログラム書法 第 2 版，共立出版（1982）
2) 堀正広 : 例題で学ぶ 英語コロケーション，研究社（2011）
3) Jean E. Sammet: The Early History of COBOL, History of Programming Languages, ACM（1978）