COBOLに関する諸問題

三井信雄

緒言

COBOLが1959年に提案されてから今日までCOBOL実用化の研究は着実に進展し，米国の計算機メーカーでは優秀なCOBOL・コンパイラーをユーザーに提供する段階にまで発展した．

NHKではEDPS導入決定当時よりCOBOLが将来事務計算のプログラミング手法として有用であることを認め，38年10月IBM 7044 System導入を機会に，全面的なCOBOLによるプログラミングの実施に踏み切った．

約一年間の実績を通し，COBOLにより処理速度の速いプログラムを作ることは，かなり手法的な面があること，使用する計算機の特性，コンパイラーの機能をある程度理解しておく必要があること，また一方COBOL特有のプログラミング上のむずかしさがあることが明確になってきた．

これは特に，われわれがBinary-Word計算機を事務計算に適用したことからくる矛盾点かもしれないが，プログラム上の難易さを別にすれば，でき上がった結果は事務計算といえどもBinary-Word計算機のほうが利とするところが多い．また最近のCOBOLコンパイラーのVersionでは，この点を十分に考慮して設計されている．

本文はIBM 7070/44 COBOL実用化の段階で，われわれが経験したCOBOL運用上の諸問題と，実行処理時間をはやめるため，検討した手法上の問題を紹介し，終りに簡単にコンパイラーの機能についても触れてみた．

結論的にいって，現在COBOLは実用上完成の段階にもきており，将来，科学計算のFORTRAN，ALGOLとならんで事務計算の共通語として普及し発展していくであろう．

1. COBOL 使用上の総合検討

この節ではCOBOLの使用上からみた総合的な機能について触れる．

1.1. プログラミングの問題

われわれがCOBOLを採用した理由は，(1) プログラムの作成維持に要する時間と費用を最少にし，作業管理を有効に実施する，(2) システム設計側とプログラム作成側との情報の交流を確実にし，ロジクのミスコーディングを防止する，の二点にあった．

(1) の問題については，事実われわれの経験からみてもプログラムの作成時間は，Symbolicなプログラム言語を使用した場合に比べて，1/10程度に短縮は容易である．しかし，もっと重要なことは，一つのプログラムの作成がグループ編成で集団的に分担して進められること，他人の作ったプログラムの修正が，企業の中で特別の運用ルートを作ったり，規格を作らなくともスムーズに行なわれるようになったことである．

このほか，コンパイル上のdebuggingについても，エラー・メッセージが比較的親切にできていることから，Symbolicなプログラム言語までレベルを下げたリストを必要とせず，数回のテストでCOBOLの知識だけで終了でき，テスト時間が短縮できる．

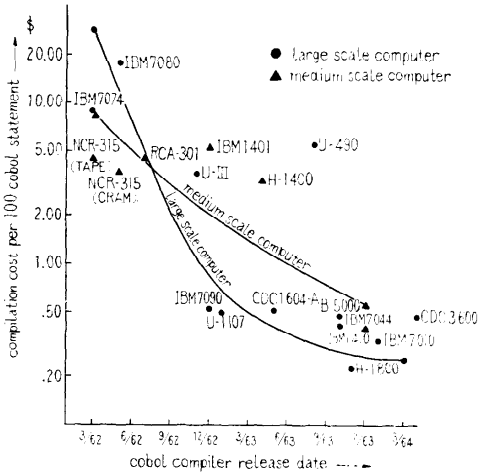
(2) については，特に問題なく情報の交流がシステム設計者との間に行なわれるようになったが，複雑なプログラムでは，フロー・チャートの助けなしにロジクを追うことはかなり困難なことが多い．

1.2. コンパイル時間と実行時間

コンパイル時間は，COBOLを採用するとき，よく問題にされる点であるが，最近のコンパイル手法の発達はすばらしく，7040/44，7090などの大型計算機では，COBOL-Subroutineを有効に利用して時間短縮を計っており，所要時間はソースカード500枚程度で3分以内でコンパイル・アセンブルまで処理を終ってしまう．この状況を各種計算機の能力とコンパイラーの評価のためコンパイル時間に関係したコンパイル・コストを一つの評価値として，コンパイラーが供給された日付を対応させた図表を参考として**第1図**に示す．

この図からもわかるように，最近提供されてきたCOBOLコンパイラーは，コンパイル時間が短縮され，

* Several problems on COBOL by Nobuo Mii (The 4th Planning Bureau, NHK (Japan Broadcasting Corp.)
** 日本放送協会　経営第4部



第1図 Compilation cost vs. Compiler release date

これに比例してコストも低減する傾向にあり，同時に大型計算機のほうに利があることがわかる．

これは従来 COBOL はコンパイル時間がかかりすぎるという批判を打ち消すもので，プログラム作成に必要なコスト面でみても，Symbolic なプログラム言語で書いた場合の人件費と debugging に要する計算機自体のコストを含めて考えると，COBOL を採用したほうが利益が多く，この差は開く方向にある．

だが，ここでユーザーとして考えたいことは，コンパイル時間を短縮するため，Symbolic で書けば，単純な Step ですむものまで，共通の Subroutine を使用することになると，実行時間にこの思想が影響してきて，好ましくない傾向を示すことである．

これは，科学計算で一つのプログラムを Compile-GO して終りにしてしまうという思想と事務計算で，何度も同一プログラムを使用するものとの差異からくる重要な問題である．

事実，コンパイラーの提供者側も，この点を考慮して，最近のコンパイラーでは，実行時間の短縮に対しての配慮が払われるようになってきた．

7040/44 COBOL コンパイラーでも Version 4 Mod 01, Version 5 Mod 01 の間では character 単位の MOVE，編集を伴う MOVE について 50～40% の時間短縮が計られ，Subscript の機能が Index Register の用法改善で 40% 程度時間短縮が得られることが確認された例がある．しかし，この代償はコンパイル時間に関係し，相対的に2割程度長くなっている．

このような配慮の下に最近の優秀な COBOL コンパイラーは，通常のプログラマーの平均以上の能力を有しており，COBOL 用法の条件を十分に検討してあればでき上がったプログラムの実行時間は，Symbolic で直接書かれたものに匹敵して差異はない．

1.3. 入力，出力の問題

事務計算のプログラムのむずかしさは，入，出力の処理にあるといわれる．この処理は COBOL の問題というより IOCS の良し悪しにあるが，COBOL の File Description Entry で簡単に Blocking や Variable Record の処理ができることは，大きな利点である．

また最近 Binary の機械が事務計算に用いられるようになり，Binary-BCD のモードを任意に FD Entry で取れるようになったことは一つの大きな進歩であり，Sequential には Disk も使えるようになってきた．事実事務計算だといって File がすべて BCD である理由はなにもない．

1.4. DATA-DIVISION の重要性

COBOL の一つの特徴である DATA DIVISION は，プログラムを書く場合に一つの大きな負担であるといわれている．しかし，この DATA DIVISION はプログラムを照査したり，将来修正する場合有効な手助けとなる．

われわれの経験からいうと，プログラムで PROCEDURE DIVISION を書く前にフロー・チャートを書き，その段階で必要な Work Area なり FD のフォーマットを決めてシンボルを明確にしておくべきであって，PROCEDURE を書きながら，DATA DIVISION の項目を決めてゆくことは，プログラム作成の手法として好ましいとはいえない．

よって PROCEDURE を書く前に，DATA DIVISION を完成させることを必然的な理由として推賞している．

また COBOL を実際使ってみて，この DATA DIVISION の設定は，ある程度の技術を要することがわかってきた．一般に新しく養成したプログラマーにとっては，PROCEDURE DIVISION を書くことはロジクさえ明確に規定してあれば，さほど困難ではないが，この DATA DIVISION はかなり計算機の処理内容を理解しないと間違いを起こしやすい．

特に後述する計算機の実行時間を決定する一つの大きな要素ともなるので，計算機一般の常識プラス，該当計算機の構成についても十分な知識を要求する．

McCracken なども，彼の著書で，この DATA DIVISION は，専門のレベルの高いプログラマーが書いたほうがよいと強調している[2].

われわれの得たデータでは，この設定の上手下手で実行時間において 10：1 程度の開きが簡単にでてくることが確認された.

以上 COBOL を総合的にその能力について述べたが，当初予想したように，プログラム作成効率の向上，運用管理の確立，容易なプログラミングなど，すぐれた利点が立証され，COBOL プログラム作成のポイントが明確になってきた.

2. COBOL プログラム手法上のポイント

COBOL プログラムの上で考慮すべき諸点は，DATA DIVISION で設定した Record が，どのような形で計算機の Area としてとられているか，またその Record に与えられた data-name を，PROCEDURE DIVISION で設定する. Verb Statement なり，Conditional Expression と組み合わせて，いかなる動きをするかを理解することである.

この組合わせを理解することは，また直接実行時間の短縮に関係しているのでプログラム手法上重要なポイントである.

2.1. Record Area の設定と Verb Statement, Conditional Expression との関係

COBOL コンパイラーがよくできていることから，プログラマーは通常次のような動きを知らないでプログラムを書くことが多い.

（a） Word 単位の計算機でも，Record の桁数が任意にとれるので，プログラムの中ではデータが character 単位で Pack された状態で取り扱われがちで，演算実行中に，演算，比較，移動のたびごとに Character ⟷ Word の変換を行なっている.

（b） Record が BCD でとられた場合，計算機によっては処理条件で BCD ⟷ Binary 変換がコンパイラーで自動的にとられている.

この第一の例として，7040/44 COBOL で

$$\text{MOVE}\left\{\begin{matrix}\text{data name}-1\\ \text{literal}\end{matrix}\right\}\ \text{TO data name-2,}$$

の Verb Statement について，data-name の条件を変えて，その動作をみると**第 2 図**のような構成となる.

この図からもわかるように data name-1 の Source-Area と data name-2 の Receiving Area との間には，DATA DIVISION の設定条件が異なると，たんに MOVE Verb だけでも 12 通りの組合わせができる.

SOURCE-AREA ＼ Receiving-area	グループ項目(a)	英字	英数字(非報告)	英数字(報告)	数値(表示)	数値(計算)
グループ項目 (a)	5	5	5	—	—	—
英字	5	5	5	—	—	—
英数字（非報告）	5	5	5	—	—	—
英数字（報告）	5	—	5	5	—	—
数値（表示）	5	—	5	1,4	1	1,3
数値（計算）	5	—	2,5	1,2,4	1,2	1
ZERO	5	—	5	1,4	1	1
SPACE	5	5	5	5(b)	—	—
LOW-VALUE	5	—	5	—	—	—
HIGH-VALUE	5	—	5	—	—	—
QUOTE	5	—	5	—	—	—
ALL	5	5	5	—	—	—

※ (a) グループ項目はすべて X とみなして，5 と同じ MOVE とする.
(b) WARNING MESSAGE がでる.

この表中に表示してある 1,2,3,4 は次の意味をもっている.

1. 小数点の位置でそろえられ，状態により，高位，下位の桁に ZERO がはいったり Truncate されたりする.
2. Binary → BCD 変換をする.
3. BCD → Binary 変換をする.
4. 編集作業が行なわれる.
5. RECEI VING-AREA に高位の桁から下位の桁（右 → 左）に 1 char ごとにはいっていく．残りの下位の桁は SPACE, SOURCE のほうの桁が長いと残りは切れる.

第 2 図　MOVE の組合せ（IBM 7040/44 COBOL MANUAL による）

この組合わせを前提として 7040/44 COBOL で実行時間を測定したのが**第 3 図**である．この測定の結果からもわかるように，7040/44 の機械は 6 桁を 1 Word とする（36 bit Binary）計算機であることから，英数字（BCD）で 6 桁，またその倍数(註)にとった場合と数値(計算)の条件で，COMPUTATIONAL SYNCHRONIZED RIGHT にとった場合が最もはやい処理時間になることがわかる.

ここで，一つの変形として興味深い事実は**第 4 図**に示す A-AREA を truncate して B-AREA の image を作る場合の手法で，次の二通りが考えられる.

（i） A-AREA PICTURE 9(6).
　　B-AREA PICTURE 9(3).

として

(註) 7040/44 では TMT instruction で，一度に 250 word を 1.3 ms で動かすことができる.

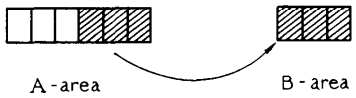
〔Data Division 77〕

data name-1 \ data name-2	9(3)	9(3) COM.	X(3)	9(5)	9(5) COM.	X(5)	9(6)	9(6) COM.	X(6)	9(9)	9(9) COM.	X(9)	9(10)	9(10) COM.	X(10)	9(12)	9(12) COM.	X(12)	Z(*)
		S-R			S-R			S-R			S-R			S-R			S-R		
Literal 1	410	10					435	10								750	15		860
9(3)	38	491					578	490											890
9(3) COM. S-R	411	10	448				435	10											
X(3)			38																860
9(5)				383	661		633	650											
9(5) COM. S-R				448	10		435	10											
X(5)						61													
9(6)	715	745		711	745		10	746	10	760	745		785	745		828	751		1233
9(6) COM. S-R	411	10		488	10		435	10	446	571	10		588	10		751	15		
X(6)			36				10	755	10										
9(9)							896	1033		375	1035								
9(9) COM. S-R							436	10		571	10								
X(9)												375							
9(10)							950	1116					418	1120					
9(10) COM. S-R							435	10					588	10					
X(10)															418				
9(12)							1056	1436								20	1445		2021
9(12) COM. S-R							456	15								733	20		
X(12)																		20	

COM ····· computational
S-R ····· synchronized right

(*IBM 7044 について NHK で調査した値)
(単位: μs)

第3図 MOVE { data name-1 / literal } TO data name-2



第4図 Source data 高位の桁の truncate

```
MOVE A-AREA TO B-AREA.
```

(ii)

```
.....
02 A-AREA
    03 FILLER      PICTURE X(3).
    03 A-AREA-E    PICTURE X(3).
.....
.....
02 B-AREA  PICTURE X(3).
```

として

```
MOVE A-AREA-E TO B-AREA.
```

(i), (ii) について比較してみると，前者が 715 μs かかるのに対し，後者は 3.8 μs で処理し，1/20 に実行時間が短縮できることがわかる．

また character として処理することが不得手であることから，Zero Suppress には 1 ms 以上の時間が平均かかっており，1401 クラスの計算機より処理は一般に遅い．

NHK では，これを解決するため，7044 から出るプリント用のテープの頭に，特別に開発した 1401-プリント・プログラムに必要な Zero Suppress 動作のパラメーターを与え，COBOL では Zero Suppress 機能は定常プログラムでは使用しないよう指示している．

次に Verb Statement のもう一つの例として四則演算の

ADD { data name-1 / literal } TO data name-2.

について MOVE ····· 同様，桁数を変えた組合せ

〔Data Division 77〕

data name-1 \ data name-2	9(3)	9(3) COM. S-R	X(3)	9(5)	9(5) COM. S-R	X(5)	9(6)	9(6) COM. S-R	X(6)	9(9)	9(9) COM. S-R	X(9)	9(10)	9(10) COM. S-R	X(10)	9(12)	9(12) COM. S-R	X(12)
Literal 1	898	15		1106	15		1178	15					1875	166		2353	210	
9(3)	1381	495																
9(3) COM. S-R	901	15																
X(3)			1390															
9(5)				1760	666													
9(5) COM. S-R				1110	15													
X(5)																		
9(6)							1918	751										
9(6) COM. S-R							1181	15										
X(6)									1925									
9(9)										2630	1040							
9(9) COM. S-R										1605	15							
X(9)																		
9(10)													2988	1276				
9(10) COM. S-R													1876	168				
X(10)																		
9(12)																3773	1623	
9(12) COM. S-R																2348	200	
X(12)																		

(*IBM 7044 について NHK で調査した値)
(単位 μs)

第5図　ADD { data name-1 / literal } TO data name-2

について，演算時間を測定した結果を**第5図**に示す．

この図から 1 word (36 bit＝10 進 9 桁＋SIGN) の間では，Binary の data (COMPUTATIONAL SYNCHRONIZED RIGHT) については，演算時間は 15 μs で一定であり，他の組合せでは Character ⟷ WORD, BCD ⟷ BIN の変換 Subroutine を呼んで，かなりの演算時間を要していることがわかる．

第6図に ADD…… の Test で得た MAP のリストを参考として示し，同じ演算でも data-name の設定条件によっては，いかに COBOL Subroutine が呼ばれてくるかを示した．

最後の例として Conditional Expression について述べよう．

IF { data name-1 / literal-1 } { = / IS EQUAL TO } { data name-2 / literal-2 }

では，**第7図**に示すように，2桁の比較までは Character 単位で行なわれ，3桁以上になると桁を高位から下位にシフトさせて差を求める方式をとっており，3桁の Equal 比較がもっとも実行時間がかかることがわかる．

また First Operand と Second Operand の間には，前に述べた例のように data-name の設定条件で，BCD ⟷ Binary 変換を伴うことについても留意しておく必要がある．次に Binary 計算機の Conditional Expression で配慮されるべき問題に Collating Sequence の取り方がある．これは Equal 比較の場合には考慮する必要がないが，

IF { data name-1 / literal-1 } { IS GREATER THAN / IS LESS THAN } { data name-2 / literal-2 }

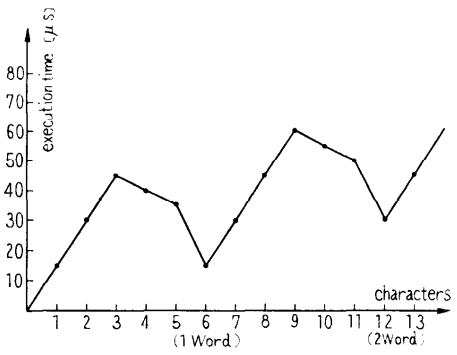
の大小比較では大きく実行時間に関係している．

第8図はこの条件の測定の一例で，Commercial

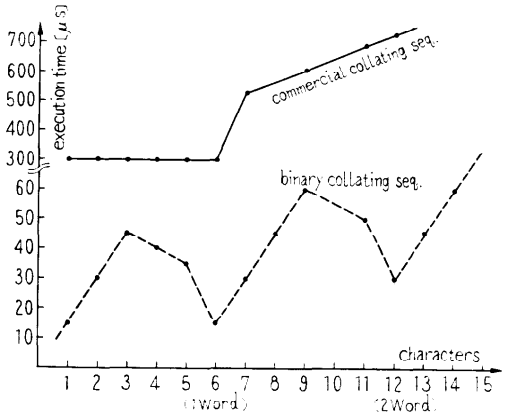
```
DATA DIVISION
    77 5A PICTURE 9(5).
    77 5B PICTURE 9(5). COMPUTATIONAL SYNCHRONIZED RIGHT.
    77 5C PICTURE 9(5).
    77 5D PICTURE 9(5). COMPUTATIONAL SYNCHRONIZED RIGHT.

ADD 5A TO 5C
    G. 0531   TRA  *+1
    S. 0414   TSX  C. MOV 2, 4 ····· MOVE Subroutine (Source Address 基準)
              PZE  S. 0132
              TXI  C. XDID, 1, 5 ····· 数字 (表示) → 数字 (計算)
                                       BCD          BIN
( 注                )  STO  G. 1031
( S. 0132 ····· 5C  )  TSX  C. MOV 2, 4 ····· MOVE Subroutine (Source Address 基準)
( S. 0102 ····· 5A  )  PZE  S. 0102
                       TXI  C. XDID, 1, 5 ····· 数字 (表示) → 数字 (計算)
                                                BCD          BIN
                       STO  G. 1031+1
                       ADD  G. 1031
                       TSX  C. MOV 1, 4 ····· MOVE Subroutine(target Address 基準)
                       PZE  S. 0132
                       TXI  C. IDXD, 1, 5 ····· 数字 (計算) → 数字 (表示)
                                                BIN          BCD
ADD 5B TO 5D
    G. 0561   TRA  *+1
    S. 0422   CLA  S. 0104
              ADD  S. 0134
              STO  S. 0134
( 注                )
( S. 0104 ····· 5B  )
( S. 0134 ····· 5D  )
```

第6図 ADD 処理の例 (IBM 7040/44 の MAPLIST の例)



第7図 Execution time per characters: equal compare (Commercial, Binary collating Seq. とも同じ)



第8図 Execution time per characters: high low compare

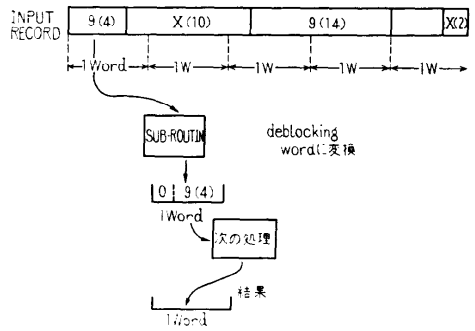
Collating Sequence と Binary Collating Sequence の間に大きな差があることを示している．この例でもわかるように Commercial の場合は，すべてコンパイラーが Subroutine で処理しようとしているのに対し，Binary では高位の桁を Character 比較して大小関係が判定できないときだけ，シフト Word に変換して（＊3桁以上のとき）比較する方式をとっており，両者の間にはかなりの開きを生じている．

事務計算では，この大小比較の実行時間が全体の運用時間を左右することが多く，Binary 計算機を事務処理に適用した場合，COBOL の能力に Mask されて見逃されるポイントで，batch 処理の場合コード体系の設定と関連して十分配慮されるべき問題であろう．

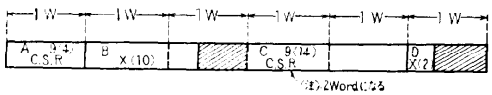
2.2. FILE, WORKING-STORAGE の取り方

前項で述べたように，COBOL といえども使用計算機にもっとも適合した形で data を設定することが望ましく，業務処理上の効率に関係してくる．

たとえば Word 単位の機械であれば，FILE 自体を**第9図**のように Packing されている形で，Word 単位で区切ることなく，一つの必要な data 項目が Chain されている場合，この data の基本項目を用いて処理を行なうと必ず図のように deblock の処理が COBOL コンパイラーで自動的にとられる．



第9図　File が pack されている場合の処理



第10図　File が unpack されている場合

このようなむだな処理をさけるためには，Word 単位の機械で COBOL を使用する場合，**第10図**のような FILE 形式として Record の記述は

```
01 RECORD-NAME
   02 A PICTURE 9(4) USAGE COMPUTATIONAL SYNCHRONIZED RIGHT.
   02 B PICTURE X(10) SYNCHRONIZED LEFT.
   02 C PICTURE 9(14) USAGE COMPUTATIONAL SYNCHRONIZED RIGHT.
   02 D PICTURE X(2) SYNCHRONIZED LEFT.
```

と書いたほうがよい．

同様のことが WORKING-STORAGE についてもいえる．ただし基本項目 77 で Area を規定した場合は，Word 単位に自動的にとられる．

このほか Binary の計算機で COBOL を用いる場合，むだな Binary ⟷ BCD 変換をさけるため，FILE についても，WORKING-STORAGE についても，Binary Mode を指定するほうがよい．

仮に Data が必然的に BCD Mode でなければならない場合には，一度だけ BCD ⟷ Binary 変換をして WORK AREA に置く方式をとるほうがむだなステップをさけることができる．このことは出力 FILE にもいえる．

しかし，Data をたんに比較して Relation Test のみ行なうのであったら，無意味な BCD ⟷ Binary 変換をとるような MOVE の組合せは用いないほうがよい．

2.3. Condition-name の活用

COBOL でロジクの条件をチェックする場合，通常のプログラムのように，条件を示すコードを直接開くような方式は COBOL の特長を生かしたとはいえない．特に複雑なプログラムで種々の条件をたびたび開く場合には，Condition-name を活用したほうが間違いが少なく，見やすい COBOL 的なプログラムを書くことができる．

たとえば

```
IF SYOKUIN=5 MOVE X TO Y ELSE ……
```

と書くなら，

```
IF SHINSAI MOVE X TO Y ELSE ……
  02 SYOKUIN PICTURE X.
  88 SHINSAI VALUE 5.
```

としたほうがプログラムを照査したり，修正するとき便利である．

しかし，この使用にあたっては，通常コンパイラーの処理 phase が一般の Relation Test と異なるので実行時間を別に検討する必要がある．たとえば，7040/44 では，上の例のような Condition-name を用いた条件チェックを行なうと，他の Conditional Expression と同じルーチンをとらず，Word 単位のものを除いて，すべて COBOL の Relation Test の Subroutin で処理し，300 μs 以上の実行時間がかかることが測定の結果わかった．これは逆にいうと，実行時間を問題にするときは，無条件で Condition-name が使えないことを指摘しており，現在変更を IBM に依頼している．

2.4. TABLE-SEARCH の一つの方法

事務計算では，よく TABLE を用いて，この引き出した値を基にして処理をすることが多い．

また，この TABLE は COBOL では通常 CONSTANT SECTION と REDEFINE で指定された SUBSCRIPT を伴う Record Description Entry で指定することが多い．

この条件で TABLE-SEARCH を迅速に行なう一つの手法を次に示す[3]．

まず TABLE を Collating Sequencne に並べるよう DATA DIVISION で規定し，次のように PROCEDURE DIVISION で BINARY-SEARCH を試みるとよい．

```
BINARY-SEARCH.
  COMPUTE HI=TABLE-SIZE+1.
  COMPOTE LO=O.
LOOK-AGAIN.
  COMPUTE N=(HI+LO)*0.5.
  MOVE TABLE-ARGUMENT(N) TO
    DUMMY-ARGUMENT.
  IF SEARCH-ARGUMENT=DUMMY-
    ARGUMENT GO TO FOUND-ITEM.
  IF SEARCH-ARGUMENT LESS THAN
    DUMMY-ARGUMENT, MOVE N TO HI
    ELSE MOVE N TO LO.
  IF LO+1 NOT=HI, GO TO LOOK-AGAIN.
  CANNOT-FIND. .....
      .....
      .....
  FOUND-ITEM. .....
```

このほか TABLE-SEARCH の方法は種々考えられるが，三次元まで許される．SUBSCRIPT を上手に利用して実行処理時間を短くすることを考える必要がある．また同時にコンパイラーの SUBSCRIPT 処理の優劣が大きく影響することも留意しておくべきで，簡単な Area まで DATA-DIVISION の記入が短くてすむということだけで使用することは危険である．

事実われわれの経験でも入力，出力 Device の処理速度内で実行時間が決まるものが，SUBSCRIPT の乱用で時間をむだにしている例があった．

3. COBOL コンパイラーの構成[4]

COBOL コンパイラーの構成にはいろいろ考えがあり，使用計算機によって翻訳手法も異なっている．

参考として本文では，IBM 7040/44 System の COBOL コンパイラーの働きを要約し紹介してみよう．

IBM 7040/44 COBOL コンパイラーは，次の四つの Phase に機能が分かれている．

(1) Phase I Language Reduction
(part 1, part 2)

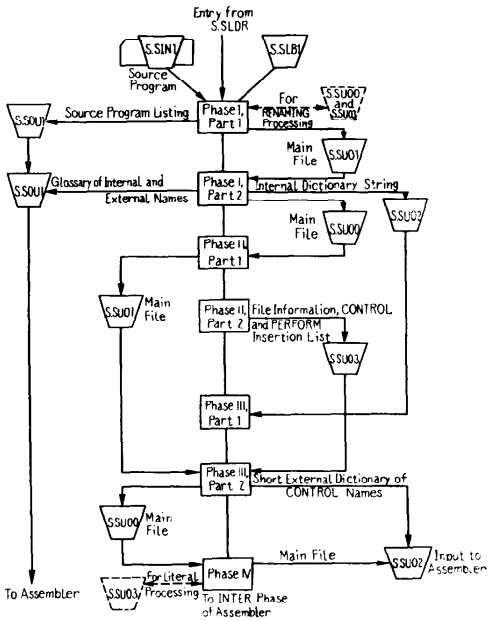
(2) Phase II Syntax Analysis
(part 1, part 2)

(3) Phase III Data Reduction
(part 1, part 2)

(4) Phase IV Procedure Generation.

以上の処理は System Library (S. SLB 1), System Input (S. SIN 1). System Output (S. SOU 1) と，ほかに四つの utility Unit を必要とし，

第 11 図のフローに従って進められる．



第 11 図 Input/output unit Allocation for COBOL compiler

3.1. Language Reduction (Phase I)

Phase I は二つの部門 part 1: Initial Edit Part 2: Qualification Reduction の機能から成り立っている．まず part 1 でソース・プログラムを COBOL の 4 division のおのおのについて識別し，個々の Statement の内容を interpret して，後続する処理のため Concise Form に部分的に encode し，Main SUO 1 に書き出す．またすべての Source-name を External Dictionary の形で記憶する．

ソース・プログラムのリストは，part 1 でソース・プログラムを読んだ時点に編集され，S. SOU 1 に output される．

part 2 では，S. SUO 1 の Main File と External Dictionary を照合し，ロジクの条件に従って Source-Name に Internal Identifier を割当て，完全に encode された形で S. SUO 3 (S. SUOO) に Main File を output する．

このほか part 2 では，DATA-DIVISION で指定してあるすべての Name については Unique な Identifier を与え，同時に encode された形で data の設定条件を規定した Internal Dictionary String を別 File として S. SUO 2 に書き出す処理を行なっている．以上 part 2 で処理された結果は，Generate された name とソースプログラムの name の対応としてプログラマーのチェックのため，S. SUO 1 にソース・プログラムリストに引き続いて output される．

3.2. Syntax Analysis (Phase II)

Phase II は part 1: Syntax Analysis. part 2: Table Processing の二つの機能から構成されており，part 1 は大略，さらに次の二つの機能に分離される．

(a) Main File を読んで，PROCEDURE DIVISION の Statement を Phase IV の個々の Verb Analyzer に適応するよう Elementary Operation の Sequence に変換する．

(b) ソース・プログラムで規定した File 関係の情報，ENVIRONMENT DIVISION に記入してある CONTROL, FILE REFERENCE Statement および PROCEDURE DIVISION で ENTRY Statement が指定してある場合の External Name, PERFORM Statement の return linkage の情報を摘出し table の形で記憶する．

以上 (a), (b) の処理を行なって Main File を S. SUO 1 に書き出し，Phase III の part 2 に渡す．

次に part 2 では part 1 で (b) の処理によって摘出された Table の情報を FILE, LABEL の Pesudo Operation Record, PERFORM CONTROL の Insertion Record の形にまとめて S. SUO 3 に移しかえるための処理を行なう．

3.3. Data Reduction (Phase III)

Phase III も二つの part に分かれ，まず part 1 において Phase 1 で encode した形で S. SUO 2 にはいっている．DATA DIVISION の VALUE を除く Clause をさらに Analyse し，USAGE, CLASS, SIZE, SYNCHRONIZE, Scaling, Decimal Integer Count などの情報から data の構成条件を規定し Internal Dictionary として Phase III の処理の間，記憶しておく．

そこで，Phase III の part 2 では，

(a) Internal dictionary としての data の構成条件

(b) File 情報 CONTROL PERFORM の insert 条件 …… (S. SUO 3)

(c) 基本 operation に還元した PROCEDURE, DIVISION, encode された形でまとめられた DATA DIVISION (File, Working-Storage, Constant Section に区分されている)．を含む Main-File. …… (S. SUO 1)

の三つの情報が合流する．

よって，DATA DIVISION の FILE SECTION が Main File にあると，おのおのの File について，この phase で IOCS に必要な Calling Sequence を generate する．

WORKING-STORAGE CONSTANT SECTION があると，internal dictionary を対応して，その character 数に適合する AREA を確保する．また VALUE が指定してあると，ここで OCT の Pseudo-Operation で必要な値を Generate する．

PROCEDURE DIVISION の data name は DATA-DIVISION で規定された条件によって性格付が行なわれる．また PERFORM Verb に関連する Section または paragraph に対しては insert 条件をチェックし，処理ルーチンを Generate する．このほか，この phase では，Object Time ルーチンの name, File に関連した name, CONTROL SECTION の name, ソース・プログラムの ENTER ASSEMBLY-PROGRAM Section にある name で直接アンセンブラーに関係するものを External-Dictionary として Generate する．

3.4. Procedure Generation (Phase IV)

この phase では，アンセンブラー処理ができる形に，変換する最後の処理が行なわれている．よって Phase IV では Procedure の General Statement を特定のアセンブラー命令の形に Coding する．しかし，この Coding のやり方は Data の組合せによって異なる．

たとえば MOVE A TO B では A, B の Data 設定条件で Coding のやり方が違う.

(i) A, B が両者とも 1 word と定義づけられているとき,

```
CLA A
STO B
```

(ii) A, B が n word であれば

```
CAL LABEL
TMT  n
 ⋮
LABEL PZE B, A
```

(iii) A, B が両者とも numeric で Computational で

A ····· 99V9 B ····· 99V99 ならば

```
LDQ A
MPY=10
STQ B
```

と幾通りもの組合わせができることになる.

このような一連の Macro Assembly Program への変換は通常 (MOVE, ADD, IF, GO TO などの Analyzer を介して行なわれる.

その一つの特殊なケースとして, subscript symbol をもった MOVE を考えてみる.

```
MOVE A(ij) TO B
```

この場合, Phase II でまず, 二つの statement に分解する.

```
(i)  SUBS A, i, j, S
(ii) MOVE S, B
```

ここで SUBS, MOVE は Analyzer と呼ばれるもので, Phase IV でまず SUBS が条件によって S を決定し, MOVE が S の条件により, アセンブラー命令の形式に変換する.

このほか Phase IV では, Index Register, AC, MQ の用法が最適なものかどうかを, つねに考慮して処理が行なわれている.

以上 Phase I～IV までの処理は COBOL 文法上のエラーをチェックしながら進められ, 途中でエラーが発見されても, 仮定条件をおいて, 一応最後まで処理を行なう方式を IBM 7040/44 COBOL では採用している.

結 言

以上われわれが IBM 7040/44 COBOL を使用した経験から考察した諸問題と, COBOL コンパイラーの動作を述べたが, 結論として COBOL 自体は十分に実用に供する能力をもっていることが確認された.

また従来 COBOL に与えられた批判は, 裏返してみれば利点といえることが多く, たとえば DATA DIVISION も含めて各 statement が冗長で書きにくいという批判は, 企業内の規程の変更で, 計算機の処理条件が修正される事務計算では恩恵を受けるケースのほうが多い.

したがって, COBOL に対する評価は, 採用する企業のおかれている状態を中心にして総合的に論ぜられるべきで, 現在の時点ではわれわれは COBOL 自体の能力に対する疑問は解決したと考えている.

しかし将来, Disk 関係の Random Access(註), ON LINE 処理はどのように取り扱われるのか, また現在使用計算機に効率のよいプログラムを書いたときに, 他の機種では能率の悪いプログラムになるのではないかという問題には, COBOL の Compatibility とも関連して非常に関心を払っている.

終りに本稿の作成にあたり御助言をいただいた, 松浦経営第4部長, 実行時間の測定などに協力された, 園田, 川原, 染谷の諸君に感謝の意を表します.

(註) 現在 NHK では特別の Random Disk Subroutine を作って COBOL の中で使用している.

参考文献

1) ROYDEN A. COWAN: IS COBOL Getting Cheaper? DATAMATION(P46), June 1964.
2) Daniel D. McCracken: A Guide to COBOL Programming 1963, John Wiley Sons.
3) F. Olsen: COBOL Coding Techniques Manual SHARE SECRETARY DISTRIBUTION, SSD. 116, 1964.
4) IBM 7040/7044 Operating System (16/32 K) System Programmer's Guide File No. 7040-36, 1963.
(IBM Systems Reference Library)

(昭和39年9月21日受付)