論　文

トップ・ダウン・プログラミング用言語系の提案*

寺　島　信　義**

Abstract

This paper discusses the language system which provides top-down programming by the following means.

(1) To describe each decomposition process in the extensible programming language EPL and the structured programming language SPL in order to eliminate ambiguity of each decomposition representations which are in general written in natural languages.

(2) To describe not the whole system but the program segments which are necessary for each decomposition process.

(3) To generate the whole system which runs in the actual programming environment by this system.

1. まえがき

プログラミングの信頼性や生産性向上の手段として，トップ・ダウン・プログラミングが注目されている[1),2),7),8),12),13)]．トップ・ダウン・プログラミングの基本的な考え方は，つぎのように要約することができよう[7),8),13)]．

(1) 問題の解 (Problem Solution) となるプログラムは，いくつかの階層のプログラムからなる．

(2) (1)のようなプログラムの階層化は，つぎのようなプロセスを経て実現される．

(a) あるステップでは，ある階層のプログラムが詳細化され，その下の階層のプログラムに分解 (decompose) される．

(b) (a)の分解のプロセスが繰り返される．そして問題の解が，特定のプログラミング言語だけで記述されるプログラム（これを最終分解のプログラムという）に分解された所で，分解のプロセスは終了する．

このような分解のプロセスは，自然言語に近い形で行われるのが一般的である．このような方法では，分解のプロセスにおける表現に曖昧さが出てくる．大きなプロジェクトになればなるほど，表現の問題が顕在化し，スムーズなプロジェクトの進捗が困難になると考えられる．この場合表現法をより厳密に行う手段が用意できれば，このような問題は除外できる．またトップ・ダウン・プログラミングでは，それぞれのステップで完全に閉じたプログラムを構成してゆかなければならない[7),12)]．一般にあるステップでは，その前のステップの一部の処理が詳細化される．その他の処理は，そのままそのステップに引き継がれる．そのまま引き継がれる分の記述が省略できれば，分解の省力化が達成される．またこのような分解のプロセスは，一般に人手で行われており工数がかさむ．それゆえ必要最小限の分解を人手で行い，プログラム全体の分解をシステムで自動的に行うことができれば，やはり省力化が可能となる．それゆえ本研究では，表現の曖昧さを除去するために，自然言語に代るものとして拡張型言語を利用する．ここに拡張型言語は，Leavenworth 流の文マクロ (Smacro) や関数マクロ (Fmacro) と同じ概念に属する[9),16)]．拡張型言語を利用することにより，自由な表現ができるだけでなく，分解のプロセスが厳密に定義できる．さらに分解の結果，得られたプログラムの信頼性を高めるために，このプログラムが記述されるプログラミング言語として，ストラクチャ

* The Top down Programming Language System by Nobuyosni TERASHIMA (Planning and Coordination Office, Yokosuka Eectrical Communication Laboratory, N.T T.)

** 日本電信電話公社横須賀電気通信研究所企画管理室

ード・プログラミング言語の概念を導入する[4),5)]．また各ステップでは，分解を必要とする部分のみの記述を行えばよいようにする．そして最終分解のプログラムは，これらの分解の記述に基づいて，この言語系により生成されるようにする．これにより分解作業の省力化をはかる．

本論文では，このようなトップ・ダウン・プログラミングの省力化を行うと同時に，得られるプログラムの信頼性を高めるための言語系の概念，この言語系の構成要素であるストラクチャード・プログラミング言語（SPL と呼ぶ）および拡張型言語（EPL と呼ぶ）の仕様概要，この言語系を用いたトップ・ダウン・プログラミングの例およびこの言語系の特徴について述べる．

2. この言語系の概念

2.1 分解の定義

分解のプロセスを，プログラミング言語の概念を用いてつぎのように表現する．

$$P=\{ST,\ SUB_{(C)},\ MACRO_{(C)}\}$$
$$SUB_{(B)}=\{ST,\ SUB_{(C)},\ MACRO_{(C)}\}$$
$$MACRO_{(B)}=\{ST,\ SUB_{(C)},\ MACRO_{(C)}\}$$
$$\cdots(1)$$

ここに $\{\alpha,\beta\}$ は，α と β からなる集合を示す．P は設計の対象となる全システム動作が記述されるプログラム，$SUB_{(C)}$ および $MACRO_{(C)}$ は，それぞれサブルーチン（関数サブルーチンも含まれる）およびマクロ呼び出し，$SUB_{(B)}$ および $MACRO_{(B)}$ は，それぞれサブルーチンおよびマクロ本体，ST は文（プログラム構造文，宣言文，実行文など）の意である．P の分解のプロセスは，(1)式で $SUB_{(B)}$ や $MACRO_{(B)}$ が ST のみからなるプログラムになる所で終了する．

2.2 この言語系の構成

たとえば PL/I[10] では，$SUB_{(C)}$ は CALL 文あるいは関数呼び出しの形で記述され，$MACRO_{(C)}$ は，プリプロセサ文で記述される．これらは，いずれも記述性や読解性が悪い．それゆえこれらを自然言語に近く，しかも厳密な形で書きたい．このアプローチとして，表現に自由度があり，厳密に記述できる拡張型言語 EPL を導入する．このときサブルーチン呼び出しやマクロ呼び出しは文に対応させ，関数サブルーチン呼び出しは式に対応させて記述するのが適当である．このような文や式の定義を EPL の文法則記述で行う．また PL/I では，$SUB_{(B)}$ は手続き，$MACRO_{(B)}$ は文の列である．これは PL/I に限定されることではなく，一般的に成立する事項であることは明らかであろう．これらの記述を EPL の意味則記述で行う．ST は，プログラム P が最終的にこの言語系により展開される文である ST をストラクチャード・プログラミング言語 SPL で記述することにより，得られるプログラムの信頼性を高める．（SPL の信頼性評価については，**6.** で述べる．）

step 1 $P=\{ST,\ SUB_{1(C)},\ MACRO_{1(C)}\}$
step 2 $SUB_{1(B)}=\{ST,\ SUB_{2(C)},\ MACRO_{2(C)}\}$ and/or $MACRO_{1(B)}=\{ST,\ SUB_{2(C)},\ MACRO_{2(C)}\}$
⋮
step n $SUB_{n-1(B)}=\{ST\}$ and/or $MACRO_{n-1(B)}=\{ST\}$

Fig. 1 Decomposition process

さて SPL と EPL の関係をプログラム P の解析処理の観点から明らかにする．P について SPL 文法による解析処理を行う．このとき文の出てくる文脈では，まず EPL で定義された文の文法による解析を行い，その後 SPL の文の文法による解析を行う．また式の出てくる文脈では，まず EPL で定義された式の文法による解析を行い，その後 SPL の式の文法による解析を行う．このようにこの言語系は，EPL と SPL からなる．

トップ・ダウン・プログラミングの分解のプロセスを **Fig. 1** に示す．Fig. 1 で分解の記述はつぎのように行われる．(1)まず P が通常 EPL で定義された文法と SPL の文法で記述される．(2)つぎに step i から step $i+1$ への分解は，EPL を用いてつぎのように行われる．ここに $i=1,\cdots,n-1(n>=2)$ である．step i の分解を必要とする $SUB_{i(C)}$ や $MACRO_{i(C)}$ の記述は EPL の文法則記述で，分解された step $i+1$ の $SUB_{i(B)}$ や $MACRO_{i(B)}$ の記述は EPL の意味則記述で行われる．そしてこの言語系が，(1)および(2)に基づいて P を SPL のみからなるプログラムに展開する．

3. SPL 言語仕様

SPL は，つぎのような特徴を持つ．

(1) データ宣言部と手続き部の明確な分離．

(2) GO TO 文は使用できない[3)]．

(3) BLISS[4)] や PASCAL[5)] と同等の制御構造(IF ～, ループ制御文，ループからの脱出文）を持つ．このほか GCASE 文や割込み制御機能などの SPL 特有の制御構造を持つ．

(4) イン・ライン・コード記述機能[6]を持つ.

ここでは SPL の特徴的な制御構造である GCASE 文と割込み制御機能について述べる. SPL の仕様詳細は, 本論文の主題ではないので省略する. なお SPL のプログラム例については, Fig. 8 を参照されたい.

3.1 GCASE 文

ネストした IF 文の構造を簡略化するために, 3つの制御構造が提案されている[11]. 実際のプログラムでは, これらの3つの制御構造とこれらの拡張された構造が組合わされて使用されることが多い (たとえば **Fig. 2** 参照). ここではこれらを記述できる GCASE 文を提案する. これを **Fig. 3** に示す.

Fig. 3 において, 〈 〉で囲まれたものは, 構文要素 (超言語変数ともいう)を示す. [α] は, 構文要素 α が出現してもしなくてもよいことを示す.

⟨expression$_i$⟩ ($i=1, 2, \cdots, n$) は, その結果が長さ1のビット列となるスカラー式である.

GCASE 文の機能は, つぎのようである.

(1) ⟨expression$_1$⟩ の値が真ならば, ⟨statement list⟩$_1$ が実行され, つぎのいずれかの動作がとられる.

(a) NEXT オプションがなく, COMMON 部が指定されていれば, COMMON 部が実行され, 他の ⟨statement list⟩ は実行されずに, 制御は ENDCASE 文のつぎの文に移る. (b) NEXT オプションが指定されれば, つぎの構文である (⟨expression$_2$⟩) : ⟨statement list⟩$_2$ [NEXT] [ELSC ⟨statement list⟩$_{2e}$] が実行される. (c) NEXT オプションも COMMON 部もなければ, ENDCASE 文のつぎの文に制御が移る.

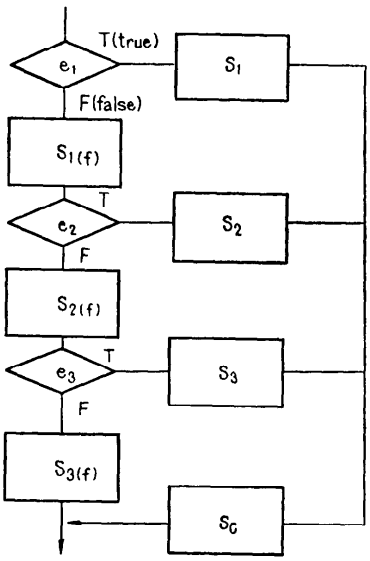

Fig. 2 IF Statement nested structure.

```
GCASE;
( <expression_1> ): <statement list>_1 [ NEXT ]
   [ ELSC <statement list>_1e ]
   .
   .
( <expression_i> ): <statement list>_i [ NEXT ]
   [ ELSC <statement list>_ie ]
   .
   .
( <expression_n> ): <statement list>_n
   [ ELSC <statement list>_ne ]
   [ COMMON <statement list>_c ]
ENDCASE;
```

Fig. 3 GCASE statement.

(2) ⟨expression$_1$⟩ の値が偽ならば, ELSC 部 (⟨statement list⟩$_{1e}$) が実行され (ELSC 部があれば), つぎの構文 (⟨expression$_2$⟩) : ⟨statement list⟩$_2$ [NEXT] [ELSC ⟨statement list⟩$_{2e}$] に制御が移る. この構文がなければ, ENDCASE 文のつぎの文に制御が移る.

(1)でも(2)でも, つぎの構文に制御が移れば, 上述と同様の動作が繰り返される.

GCASE 文の使用法を Fig. 2 を用いて示す.

Fig. 2 で e_1, e_2, e_3 はスカラー式, S_1, S_2, S_3, $S_{1(f)}$, $S_{2(f)}$, $S_{3(f)}$, S_C は文の列 ⟨statement list⟩ である. この制御構造で $S_{1(f)}$ および $S_{2(f)}$ が空文の時, CASE 文[11]で記述できるが, これ以外の場合, CASE 文では一般には記述できない. それゆえこれらの場合には IF 文で記述するのが適当である. しかるに SPL では, この構造を GCASE 文によりつぎのように簡潔に記述することができる.

```
GCASE;
(e1): S1
      ELSC S1(f)
(e2): S2
      ELSC S2(f)
(e3): S3
      ELSC S3(f)
      COMMON SC
      ENDCASE;
```

なお GCASE 文により, 文献 11) の制御構造を全て記述することができる.

3.2 割込み制御機能

PL/I では, 割込み制御の流れは, 逐次制御の流れとは独立に扱われる. たとえば領域のオーバフロ条件や EOF (end-of-file) 条件が発生した時, その時 active

な ON 文が実行される．つまり ON 条件が発生すると，不連続な制御移行が行われる．

たとえばつぎのような PL/I で記述されたプログラムがあるとする．

```
M: PROC;
DCL A AREA (4096);
DCL X CHAR (16) BASED;
DCL P PTR;
    ⋮
ON AREA BEGIN;          ……(1)
A=EMPTY; RETURN; END;
    ⋮
ALLOCATE X SET (P) IN (A);
                        ……(2)
END M;
```

このとき文(2)で AREA 条件が発生すると，文(1)の ON 文が実行される．このような不連続な制御移行は，プログラムの well-structured な制御の流れに妨害を与える．それゆえ SPL では，制御の流れを well-structured にするために，ON ユニットを割込みの発生する可能性のある文に，その都度指定させる．

たとえば上述のプログラムを SPL ではつぎのように記述する．

```
ALLOCATE X SET (P) IN (A)
ON AREA BEGIN; A=EMPTY;
RETURN; END;
```

この文で AREA 条件が発生すると，この文の ON ユニットが実行され，制御の流れは乱されない．

4. EPL 言語仕様

EPL には，文と式の拡張機能がある．ここでは，本論に必要な範囲すなわち文の拡張法についてのみ示す．EPL では，文の拡張定義に超言語機能[17]を導入する．このときつぎのような超言語機能の表記法を採用する．

(1) バッカス記法や COBOL 記法の超言語連結詞 (metalinguistic connectives) に相当するものとして，OPTIONAL(‥) や ITERATION(‥) などの直観的に理解し易いニモニック記法を導入する．

(2) AN (Algol N) 記法[17]では，構文の結合の順序を括弧で表現するが，EPL ではレベル番号で表現する．すなわち前者は1次元的な表記法であるのに対して，後者は2次元的な表記法である．したがって後者は前者に比べて，記述性が良いばかりでなく，直観的にも理解し易い．

4.1 文の拡張法

(1) 文の文法則記述法

これを **Fig. 4** に示す．Fig. 4 において {α}… は，構文要素 α が1回以上繰り返し出現することを示す．{α|β} は，構文要素 α か β のいずれか一方が出現することを示す．[α]… は，構文要素 α が0回以上繰り返し出現することを示す．〈identifier〉1 は，文の文法則記述を識別する名前である．〈minor structure〉(以後項目という)の直前の数字はレベル番号を示す．レベル番号 n (n>=2) の項目には，つぎの n より大きい項目で，はじめて n あるいは n より小さいレベル番号を有する項目が出現するまでの全ての項目が含まれる．レベル番号 n の項目の子を，これに含まれるレベル番号 $n+1$ の項目と定義する．またこのとき，レベル番号 n の項目を，これに含まれるレベル番号 $n+1$ の項目の親と定義する．ある親の項目に含まれる子の項目は，その親が RANDOM か ONE-OF でなければ，これらの記述順序に従がって出現する（もし出現するとすれば）ことを示す．〈character string constant〉1 は，拡張される文あるいは文の1部（キーワードや区切り記号など）を示し，そのまま出現することを示す．ITERATION (‥) は，これに含まれる項目が n (n>=1) 回繰り返し出現することを示す．繰り返しの回数が〈parameter〉2 に設定される．RANDOM は，これに含まれる子の出現順序が任意であることを示す．OPTIONAL(‥) は，これに含まれる項目が全て出現してもしなくてもよいことを示す．そしてこれらの項目の出現状況（出現すれば '1'B，出現しなければ '0'B）が〈parameter〉3 に設定される．〈character string constant〉2 DEL は，ITERATION (‥) の最後の子として指定でき，〈character string constant〉2 が，ITERATION (‥) に含まれる項目の最後の繰り返しを除いて繰り返しの度に出現することを示す．この機能をつぎの例で示す

```
2  ITERATION ($1),
  3  'ABC',
  3  ',' DEL,
```

この例で $1 が3の時，文字列 'ABC, ABC, ABC' を表わす．

ONE-OF は，これに含まれる子のどれか1つが出

```
〈statement syntax rule description〉
:= 1 〈identifier〉1 SYNTAX-OF-STATEMENT{,2 〈minor structure〉2ss
[,3 〈minor structure〉3ss[,..[,n+1 〈minor structure〉nss]...]...]...}...;

〈minor structure〉1ss~nss
:= {〈character string constant〉1 | ITERATION( 〈parameter〉2)
|RANDOM | OPTIONAL( 〈parameter〉3) | 〈character string constant〉2 DEL
|ONE-OF | 〈identifier〉2 | 〈parameter〉1{VARIABLE | EXPRESSION | STATEMENT}}
```

Fig. 4 Statement syntax rule description.

現することを示す.

〈identifier〉$_2$ は,項目を識別する名前である.

VARIABLE は,〈parameter〉$_1$ が SPL で許される変数の形式を持つことを示す. EXPRESSION ならびに STATEMENT は,それぞれ〈parameter〉$_1$ が SPL の式および EPL で拡張された式ならびに SPL の文および EPL で拡張された文の形式を持つことを示す.

(2) 文の意味則記述法

これには,2つの形式がある. すなわち文の列で与えられるものおよび手続きで与えられるものがある. ここでは前者についてのみ示す. この意味則記述法を **Fig. 5** に示す.

Fig. 5 で,ある親の項目に含まれる子の項目は,その記述順序に従がって出力される(もし出力されるとすれば)ことを示す. 〈identifier〉は,文の意味則記述の名前である. 〈identifier〉$_1$ は,対応する文法則記述を識別するために使用される. すでに定義されている意味則を使用したい時には,そのレベル1の項目のみを指定すればよい. 〈character string constant〉$_1$ は, SPL の文(の列)(あるいは/および) EPL により拡張された新しい文(あるいは/および)これらの文の1部を示し,そのまま出力されることを示す. 〈parameter〉$_1$, 〈parameter〉$_2$ および〈parameter〉$_3$ は,対応する〈statement syntax rule description〉から引渡される情報である(文法則と意味則の間で引渡されるパラメタに関する規則は,(3)で記述する). このうち〈parameter〉$_1$ は,そのまま出力される. ITERATION (‥) は,〈parameter〉$_2$ の値の回数だけ,これに含まれる項目が繰り返し出力されることを示す. OPTIONAL (〈parameter〉$_3$) は,〈parameter〉$_3$ の値が '1'B なら,これに含まれる項目が出力され, '0'B なら出力されないことを示す. 一方 OPTIONAL (¬〈parameter〉$_3$) は,〈parameter〉$_3$ の値が '0'B ならば,これに含まれる項目が出力され, '1'B なら出力されないことを示す.

〈character string constant〉$_2$ DEL は, ITERATION (‥) に含まれる最後の子として指定でき,〈character string constant〉$_2$ が, ITERATION (‥) に含まれる項目の最後の繰り返しを除いて繰り返しの度に出力されることを示す. **Fig. 6** に,文法則記述法と意味則記述法の具体例を示す.

(3) 文法則記述と意味則記述間のパラメタの授受法

```
〈statement semantic rule description〉
:= 1 〈identifier〉 SEMANTICS-OF-STATEMENT( 〈identifier〉 1)
[,2 〈minor structure〉 1sm[,3 〈minor structure〉 2sm[,...[n+1 〈minor
structure〉 nsm]...]...]...];
〈minor structure〉 1sm-nsm
:= { 〈character string constant〉 1 |〈parameter〉 1 | ITERATION( 〈parameter〉 2)
|OPTIONAL {( 〈parameter〉 3)|(¬〈parameter〉 3)}|〈character string constant〉 2DEL}
```

Fig. 5 Statement semantic rule description.

```
1 FOR-ST SYNTAX-OF-STATEMENT,
  2 ONE-OF,3 'FORDO',3 'FOR',3 'DOFOR',2 $1 VARIABLE,2 '=',
  2 ITERATION($2),3 $3 EXPRESSION,3 RANDOM,4 OPTIONAL($4),
  5 'TO',5 $5 EXPRESSION,4 OPTIONAL($6),5 'BY',5 $7 EXPRESSION,
  3 ',DEL,2 $8 STATEMENT;
1 SEM-OF-FOR SEMANTICS-OF-STATEMENT(FOR-ST),
  2 ITERATION($2),3 'DO',3 $1,3 '=',3 $3,3 OPTIONAL($4),
  4 'TO',4 $5,3 OPTIONAL($6),4 'BY',4 $7,3 OPTIONAL(¬$6),4 'BY 1',
  3 ';',3 $8,3 'ENDDO;';
```

Fig. 6 Syntax and semantic description example.

文法則記述で用いられるパラメタは,意味則記述の対応するパラメタに引き渡される. パラメタは $ 記号を前置された数字で表現される. 文法則記述と意味則記述間のパラメタの対応関係を Fig. 6 を用いて示す. Fig. 6 で文法則記述 FOR-ST のパラメタ $1 は,意味則記述 SEM-OF-FOR の対応するパラメタ $1 に引き渡される. 同様に FOR-ST の $2, $3, $4 などが,それぞれ SEM-OF-FOR の $2, $3, $4 などに引き渡される.

(4) EPL の使用例

入力文字列 FOR A=B TO C, D BY E TO F, G TO H BY I S(A)=T(A)+U(A); が原始プログラムに現われた時,この文字列は Fig. 6 の定義に照合され, FOR-ST とみなされ,この文法則記述の $1 から $8 とつぎのように対応づけられる.

(a) $1 はAとみなされる.

(b) $2 は3とみなされる. それゆえ ITERATION ($2) に含まれる項目は,全て1次元の配列で配列要素の個数は3である. これらの項目と原始プログラムの対応はつぎのようになる.

(i) $3(1), $3(2), $3(3) はそれぞれ B, D, G とみなされる.

(ii) RANDOM の子 $4(1), $4(2), $4(3) には,いずれも '1'B(存在する)が設定され,その子である $5(1), $5(2), $5(3)は,それぞれC, F, H とみなされる. 一方 $6(1), $6(2), $6(3)には,それぞれ '0'B(存在しない), '1'B, '1'B が設定され,その子である $7(1), $7(2), $7(3) は,それぞれ空(存在しない), E, I とみなされる.

(c) $8 は，SPL の代入文 S(A)=T(A)+U(A); とみなされる．

このようにして FOR-ST のパラメタ $1～$8 と入力文字列との対応が与えられる．

これらのパラメタが対応する意味則記述 SEM-OF-FOR に渡され，つぎのような文の列が生成される．ここでは理解を助けるために2段階の展開を行う．

(d) 第1段階 まず ITERATION ($2)，OPTIONAL ($4)，OPTIONAL ($6) および OPTIONAL (¬$6) の値に従がって意味則を生成すると，つぎのような文の列が生成される．

```
DO $1=$3(1) TO $5(1) BY 1; $8 ENDDO;
DO $1=$3(2) TO $5(2) BY $7(2); $8 ENDDO;
DO $1=$3(3) TO $5(3) BY $7(3); $8 ENDDO;
```

ここに $6(1) に対しては，OPTIONAL (¬$6) の指定が適用され，これに対応する文脈に "BY 1" が出力される．

(e) 第2段階 $1, $3, $5, $7 および $8 を，それぞれ対応する値で置き換えると，つぎのような文が生成される．

```
DO A=B TO C BY 1;
S(A)=T(A)+U(A); ENDDO;
DO A=D TO F BY E;
S(A)=T(A)+U(A); ENDDO;
DO A=G TO H BY I;
S(A)=T(A)+U(A); ENDDO;
```

4.2 曖昧性の除去

EPL で拡張された文について，その文の解釈に曖昧性が生ずる場合がある．この言語系では，曖昧性の除去対策を講じている．これについて例を用いて説明する．EPL で拡張された文の1例を，つぎに示す．

⟨statement⟩::=EIF⟨expression⟩ ETHEN⟨statement⟩₁ [EELSE ⟨statement⟩₂]

入力文字列が EIF E1 ETHEN EIF E2 ETHEN A=B; EELSE A=C; のとき，**Fig. 7** に示すような2通りの解釈が生ずる．この言語系では解釈 2 の解釈をとることにより，曖昧性を除去している．すなわちネスト・レベルの深い位置にある構文則を優先的に適用することにより，構文解析を行う．

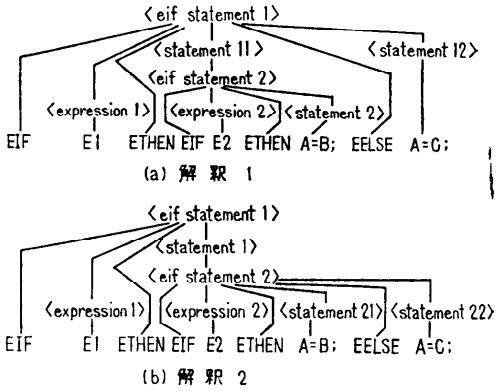

Fig. 7 Ambiguous statement example.

5. 記述例と特徴

文献 12) のトップ・ダウン・プログラミングの例 (Reservation Program) を分解し，EPL による記述を行う．なおここでは文献 12) との対応を明らかにするために，GCASE 文は使用しない．

(Step 1) Reservation Program の基本的構成を決める．

すなわち

(1) Transaction-File を読み，Transaction-Code により，Reservation-File の創成や更新を行う．

(2) (1)の処理を，Transaction-File が空になるまで繰り返す．この処理を EPL により記述するとつぎのようになる．

```
1 RESERVATION-PROGRAM SYNTAX-OF-STATEMENT,
  2 'RESERVATION-PROGRAM;';
1 SEM-OF-RESERV SEMANTICS-OF-STATEMENT(RESERVATION-PROGRAM),
  2 'REPEAT FOREVER; READ FILE(TRANSACTION-FILE) ON(EOF) DO;
CALL PRINTA(RESERVATION-FILE); STOP; END;
IF TRANSACTION-CODE='1'B
 THEN CALL CREATE(RESERVATION-FILE);
 ELSE UPDATE(RESERVATION-FILE); ENDREP;';
```

(Step 2) (Step 1) で定義された UPDATE の分解を行う．

```
1 UPDATE SYNTAX-OF-STATEMENT,
  2 'UPDATE(',2 $1 VARIABLE,2 ');';
1 SEM-OF-UPDATE SEMANTICS-OF-STATEMENT(UPDATE),
  2 'IF UPDATE-CODE='0'B THEN DO; FIND(',2 $1,2 '-RECORD)INVALID-KEY
DO; ADD(',2 $1,2 '-RECORD); INVALID='1'B; END; IF INVALID='0'B THEN
MODIFY-ADD(',2 $1,2 '-RECORD); ELSE INVALID='0'B; END; ELSE DO; FIND(',
2 $1,2 '-RECORD)INVALID-KEY DO; CALL PRINT('NON EXISTENT RECORD',
TRANSACTION-RECORD); INVALID='1'B; END;IF INVALID='0'B THEN MODIFY-DELETE
(',2 $1,2 '-RECORD) ELSE INVALID='0'B; END;';
```

(Step 3) (Step 2) で定義された FIND, ADD, MODIFY-ADD および MODIFY-DELETE の分解を行う．

```
1 FIND SYNTAX-OF-STATEMENT,
  2 'FIND(',2 $2 VARIABLE,2 ')INVALID-KEY';
1 SEM-OF-FIND SEMANTICS-OF-STATEMENT(FIND),
  2 'CALL FIND(',2 $2,2 ');IF INVALID-KEY='1'B THEN';
```

```
1 ADD SYNTAX-OF-STATEMENT,
  2 'ADD(',2 $2 VARIABLE,2 ');';
1 SEM-OF-ADD SEMANTICS-OF-STATEMENT(ADD),
  2 'CALL INSERT(',2 $2,2 '); CALL PRINT('NEW RECORD ADDED',',
  2 $2,2 ');';

1 MODIFY-ADD SYNTAX-OF-STATEMENT,
  2 'MODIFY-ADD(',2 $2 VARIABLE,2 ');';
1 SEM-OF-MODIFY-ADD SEMANTICS-OF-STATEMENT(MODIFY-ADD),
  2 'IF PASSENGER-NAME-1 OR PASSENGER-NAME-2='    ' THEN DO;
  CALL MODIFY(',2 $2,2 '); CALL PRINT('RESERVATION ADDED',',
  2 $2,2 '); END; ELSE DO; CALL REWRITE(',2 $2,2 '); CALL PRINT(
'NO ROOM ON FLIGHT',',2 $2,2 '); END;';

1 MODIFY-DELETE SYNTAX-OF-STATEMENT,
  2 'MODIFY-DELETE(',2 $2 VARIABLE,2 ');';
1 SEM-OF-MODIFY-DELETE SEMANTICS-OF-STATEMENT(MODIFY-DELETE),
  2 'DO; CALL MODIFY(',2 $2,2 '); IF PASSENGER-NAME-1 AND
PASSENGER-NAME-2='    ' THEN DO; CALL DELETE(',2 $2,2 '); CALL
PRINT('RECORD DELETED',',2 $2,2 '); END; ELSE DO; CALL REWRITE
(',2 $2,2 '); CALL PRINT('RESERVATION DELETED',TRANSACTION-RECORD
); END; END;';
```

(Step 1), (Step 2) および (Step 3) の分解において，$1=RESERVATION-FILE, $2=RESERVATION-FILE-RECORD を与え，EPL で処理すると，**Fig. 8** に示すような SPL で記述された Reservation Program が生成される．

```
REPEAT FOREVER;
READ FILE(TRANSACTION-FILE) ON(EOF) DO;
CALL PRINTA(RESERVATION-FILE);STOP;END;
IF TRANSACTION-CODE='0'B
THEN CALL CREATE(RESERVATION-FILE);
ELSE IF UPDATE-CODE='0'B
THEN DO; CALL FIND(RESERVATION-FILE-RECORD);
IF INVALID-KEY='1'B
THEN DO; CALL INSERT(RESERVATION-FILE-RECORD);
CALL PRINT('NEW RECORD ADDED',RESERVATION-FILE-RECORD);
INVALID='1'B; END;
IF INVALID='0'B
THEN IF PASSENGER-NAME-1 OR PASSENGER-NAME-2='
THEN DO; CALL MODIFY(RESERVATION-FILE-RECORD);
CALL PRINT('RESERVATION ADDED',RESERVATION-FILE-RECORD); END;
ELSE DO; CALL REWRITE(RESERVATION-FILE-RECORD);
CALL PRINT('NO ROOM ON FLIGHT',RESERVATION-FILE-RECORD); END;
ELSE INVALID='0'B; END;
ELSE DO; CALL FIND(RESERVATION-FILE-RECORD);
IF INVALID-KEY='1'B
THEN DO; CALL PRINT('NON EXISTENT RECORD',TRANSACTION-RECORD);
INVALID='1'B; END;
IF INVALID='0'B
THEN DO; CALL MODIFY(RESERVATION-FILE-RECORD);
IF PASSENGER-NAME-1 AND PASSENGER-NAME-2='    '
THEN DO; CALL DELETE(RESERVATION-FILE-RECORD);
CALL PRINT('RECORD DELETED',RESERVATION-FILE-RECORD); END;
ELSE DO; CALL REWRITE(RESERVATION-FILE-RECORD);
CALL PRINT('RESERVATION DELETED',TRANSACTION-RECORD); END; END;
ELSE INVALID='0'B; END;
ENDREP;
```

Fig. 8 Generated SPL statement list.

Table 1 Comparison of this method and Dijkstra's method

項番	この方式	Dijkstra 流の方式
1	分解のプロセスを EPL により記述することにより，自由な表現ができるだけでなく，表現の曖昧さを除去できる．	特に規定されてはいないが，最終分解のプログラム以外は，自然言語に近い形で行われるのが普通である．それゆえ表現の曖昧さが生ずる可能性がある．
2	それぞれのステップで，プログラム全体の記述を行う必要はなく，分解を必要とする部分のみの記述を行えばよい．	それぞれのステップで，プログラム全体の記述が必要である．
3	step $i+1$ では，step i の処理単位を分解し，これらを EPL の文法則記述と意味則記述で与える．これにより分解の対象となる処理単位について，step i と step $i+1$ の対応関係が明示されるばかりでなく，これらのモジュラリティが向上する．	step $i+1$ のプログラムは，step i が分解された結果のプログラムであり，分解の対応関係は，個々の分解に対しては与えられない．
4	問題の解から，最終分解のプログラムは，EPL で記述された各ステップの分解を用いて，この言語系が処理することにより得られる．	それぞれのステップのプログラムは，そのステップの機械上で動作する完全なプログラムである．これらはプログラマが記述する．
5	分解の過程は形式的にのみ行われる．そして最終分解で実際のプログラムが得られる．それゆえ第1ステップで与える実パラメタに自由度がある．	それぞれのステップで，そのステップの機械上で動作する実際のプログラムが得られる．それゆえ左記のような自由度はない．

このように，この方式では各ステップで必要な分解を EPL により記述する．そして最終分解のプログラムは，各ステップの分解の記述をこの言語系が処理することにより得られる．この方式を，Dijkstra 流のトップ・ダウン・プログラミングの方式[7),12)]と対比して表示すると，**Table 1** のようになる．

Table 1 より，この方式は，Dijkstra 流の方式に比べて，つぎのような特徴を持つ．

(1) 項番1より，自由な表現ができ，かつ曖昧性を除去できることにより，読解性の向上が期待できる．ただし，フロー・チャート記法のような表記法はできない．

(2) 項番2より，プログラムの記述量が減少する．

(3) 項番3より，プログラム仕様の変更に伴なう修正作業や保守作業の省力化が期待できる

(4) 項番4より，プログラムの生産性の向上が期待できる．

(5) 項番5より，作成されるプログラムに自由度がある．すなわちパラメタの置き換えにより，1個以上のプログラムを容易に得ることができる．

6. あとがき

この言語系の概要ならびに記述例と特徴について述べた．この言語系の運用に基づく効果測定は今後の課題であると考えている．なお SPL については定量的な評価を行っており，つぎのような結果を得ている．

（1） 信頼性（バグ発生件数の逆数）では，SYSL[14),15)]に比べて2倍程度であること．

（2） 性能は，SYSL に比べて 2～3% のオーバヘッドであることが判明した．

このことより SPL の信頼性は高いことがわかった．信頼性向上の要因としては，GO TO 文を除去し，それに代る制御構造を導入したこと，データ部と手続き部のセグメント化を行ったことなどが挙げられよう．性能が劣化しているのは，GO TO 文を除去したためである．

参 考 文 献

1) B. H. Liscov: A Design Methodology for Reliable Software Systems, AFIPS 1972 FJCC, 41, Part 1, pp. 191～199, Spartan Books. New York
2) E. W. Dijkstra: The Structure of the "THE" Multiprogramming System, CACM 11, 5, pp. 341～346 (1968)
3) E. W. Dijkstra: Go To Statement Considered Harmful, CACM 11, 3,, pp. 147～148 (1968)
4) W. A. Wulf et al.: BLISS: A Language for System Programming, CACM 14, 12, pp. 780～790 (1971)
5) N. Wirth: The Programming Language PASCAL, Acta Informatica, 1, pp. 35～63 (1971)
6) R. D. Bergeon: Language for System Development, SIGPLAN notices 6, 9, pp. 50～72 (1971)
7) E. W. Dijkstra: Notes on Structured Programming, Structured Programming, pp. 1～72, Academic Press, London and New York (1972)
8) N. Wirth: Program Development by Stepwise Refinement, CACM 14, 4, pp. 221～227 (1971)
9) N. Solntseff & A. Yezerski: A Survey of Extensible Programming Language, Annual Review in Automatic Programming 7, pp. 267～307 (1974)
10) IBM System/360 Operating System: PL/I Language Specifications, C28-6571-4, IBM Corp. (1966)
11) G. M. Weinberg et al.: IF-THEN-ELSE Considered Harmful, SIGPLAN notices 10, 8, pp. 34～44 (1975)
12) C. L. Maclure: Top-Down, Bottom-Up, and Structured Programing, IEEE Transactions on Software Engineering SE-1, 4, pp. 397-402 (1975)
13) O. Dahl and C. A. R. Hoare: Hierarchical Program Structues, Structured Programming, pp. 175～219. Academic Press, London and New York (1972)
14) 寺島：DIPS-1 における高能率システム製造用言語の実用化，情報処理 16, 6, pp. 492～498 (1975)
15) 寺島他：システム製造用言語 SYSL-2 の設計，情報処理 16, 8, pp. 692～697 (1975)
16) R. M. Leavenworth: Syntax Macros and Extended Translation, CACM 9, pp. 790～793 (1966)
17) 島内：プログラム言語論，共立出版 (1972)

（昭和51年9月6日受付）
（昭和52年7月15日再受付）