アルゴリズム・レベルの プログラム文書自動化ツール

落水浩一郎　酒井三四郎　八木文治　岡本勝男
（静岡大学　工学部　情報工学科）

1. はじめに

ソフトウエア開発，保守の効率化をはかるには、設計者やプログラマの

(1) 知的生産活動を積極的に支援したり

(2) 思考活動の阻害要因を除去するために

方法論，道具（言語，支援プログラム群など），標準化技術，組織化技術を有機的関連をもつ体系として整える必要がある。

この問題に対して，我々はプログラム文書化技術の再編成を中心としたシステム作りをおこなっている。

ソフトウエア・ライフサイクルにおけるプログラミングチームの活動は，見方を一段深い観点に置いてみると、各種情報（各種仕様書，マニュアル，プログラムなど）の生成，蓄積，利用のプロセスとして、とらえることができる。

この立場からは、プログラミング支援システムは、上記情報の基本単位，表現形式，流通経路を確立した上で、情報の生成，蓄積，利用の流れを円滑にする支援ツールを経路上に系統的に配置したものとして定義できる。

また、プログラム文書化に関する従来の方法は、作成したプログラム一つ一つに対して、手作業により付属文書を作成するというものである。　この方法は表現に柔軟性があり，必要に応じて完全性の高いドキュメントを作りうる利点をもつ。　反面、更新に弱い，検索が容易でないという欠点もあわせもつ。

以上の考察に基づいて，本報告では、対象をソフトウエア・ライフサイクルの一つの局面，つまり，モジュールの内部論理の設計・コーディング段階として，プログラム文書の自動化ツールについて述べる。

その一つとして「リーダビリティ・チェッカ」，「アナライザ」方式による自動化ツールがある[1]。　この設計原理は、H.D.Mills の Symtax Directed Documentation [2]の考え方に基づいている。

アナライザとは、図1に示すように蓄積されたソースプログラムから，要求に応じた種々の情報（内部仕様，外部仕様レベル）を作成し，提示するツールである。

しかし、通常ソースプログラムにはこのような情報は存在しない。　リーダビリティ・チェッカは、ソースプログラムの制御構造に基づいて、意味的階層構造のレベルの情報を質疑応答によって収集し，ソースプログラムの適切な場所にコメントの形で，もりこむものである。

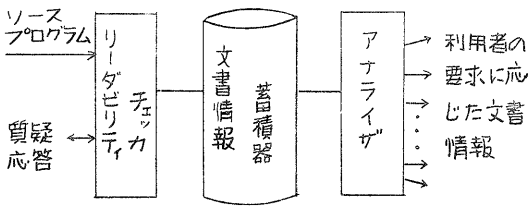


図1　リーダビリティ・チェッカ　アナライザ方式

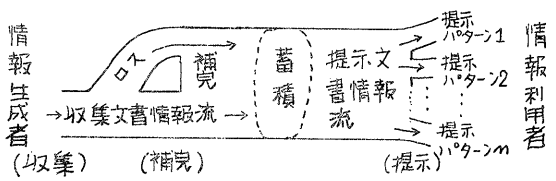
しかし、この方法は，質問がプログラムの制御構造に依存するため、質問量が多い。　また，プログラム作成との時間的ずれなどもあって，プログラマにとって，大きな負担であった。

これらの問題点を解決する方法とし

て、プログラム作成と同時に文書情報を収集すること。　さらに、収集，補完を，プログラマがモジュール実現の過程で行なった段階的詳細化の意志決定を基本に行なうことなどを採用し、生成器，補完器，提示器の三つのツールからなるアルゴリズム・レベルの文書情報の自動化ツールを実現した。

2．システムの概要

図2に生成器，補完器，提示器の関連を示す。



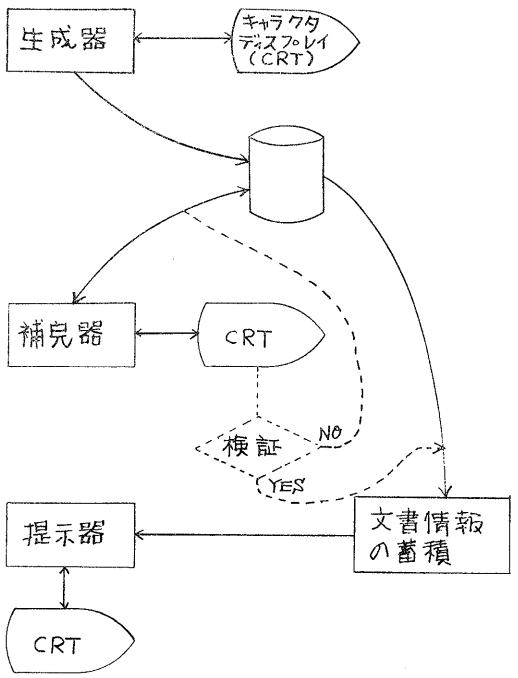


図2　システムの構成

生成器は、プログラム作成者のプログラミングの作業を助けながら、その過程で行なわれる意志決定を文書情報として自動収集する。　しかし、プログラム作成者の本来の作業である設計やプログラミングを妨げないように収集するため、十分な文書情報を収集することができにくい。

補完器は、この文書情報の不足を補い、加えて文書情報の質を高める。そして、文書情報を蓄積する。

提示器は、蓄積された文書情報から，文書情報利用者に対して、プログラムモジュールの利用や内部論理の理解に必要かつ、十分な情報を提供する。

3．生成器

3.1　基本設計

一つのプログラム・モジュールの実現過程と文書化の関連について考察する。

モジュールの外部仕様書を基に、データ構造やアルゴリズムの検討を行ない、紙などに記録する。　そして、外部仕様書に示された抽象的機能を段階的に詳細化し、実現しようとする言語のステートメントになるまでくりかえす。　この過程において、詳細化をたすける手段として図や表などを利用する。　ほとんどの場合それらは紙に書かれ、何度か書きなおされ、後に文書作成上の重要な資料となる。

また、上述のモジュール実現過程において、プログラム作成者は、ソースリスト中にコメントという形で、文書情報の一部を入れることがある。　しかし、これらは各自自分だけのためにつけることが多い。　そして、プログラム作成者は、プログラム完成後、ソースリストとモジュール実現過程で生成された情報をもとに、新めて内部仕様書を作成したり、ソースリスト中にコメントを付加することになる。

以上の考察から、次の問題点が指摘される。

(1)　段階的詳細化をたすける手段と

して利用される「紙に書く」という作業は、変更のたびに何度も書きなおしたり、清書したりするために、非常に煩わしい。

(2) モジュール作成と文書の作成に時間的ずれが生ずるために、文書作成時に、自分の作成したはずのプログラムの解読に苦労したり、モジュール作成の各段階で苦労して考えた事項をもう一度再現せざるをえない。

この問題点を解決するために図3に示すようなエディタと階層構造記述言語を設計した。以下に詳細を述べる。

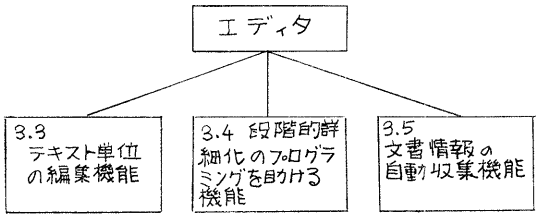


図3 エディタの機能

3.2 階層構造記述言語

この言語に要求されることは、以下のとおりである。

(1) ある機能単位をサブ機能単位に分解する作業を表現できること。

(2) 各機能単位について自然語の説明が記述できること。

(3) 記述に柔軟性があり、プログラム作成者に負担をかけないこと。

これらの要求を満たすように、図4に示す言語の設計を行なった。(1)のDCL, COND, BLOCKは、ある機能単位を表わし、その単位に名前をつけ、説明を自然語で書く。各キーワードは機能単位の性格を表わし、DCLは宣言、CONDは条件、BLOCKは手続に関するものを表わす。

```
(1) 機能単位の表現
  DCL[機能単位名：機能単位に関する自然語による説明]
  COND[  〃   :        〃                ]
  BLOCK[  〃   :        〃                ]
(2) 機能単位間の関係を表わすキーワード
  IF THEN ELSE DO BEGIN END WHILE
(3) エントリー文(プロセジャー文)とリターン文(エンド文)の表現
  ENTRY[エントリー文自身：自然語による説明]
  RETURN[リターン文自身：   〃      ]
(4) ソース・ステートメントを直接記述する場合
  〔ソース・ステートメント〕
```

図4 階層構造記述言語

3.3 テキスト単位の編集機能

図5に示すように、QEDXのコマンド体系に準ずる体系をもつ。

I,A,C,D	テキストの挿入,付加,交換,消去
M	テキストの移動
S	文字列の交換および消去
J	ポインタの移動
#XB,#XU,#XS,#XF	バッファ・ファイルの定義拡能出力
BB,BC,BD	バッファの定義,交換,解放
BF	ファイルの定義
R,W	バッファとファイル間でのテキストのリード/ライト
#Q	初期化

図5 テキスト編集のコマンド体系

3.4 段階的詳細化を助ける機能

この機能は図6に示すようなエディタ内の複数のバッファを利用して実現される。主な機能をあげる。

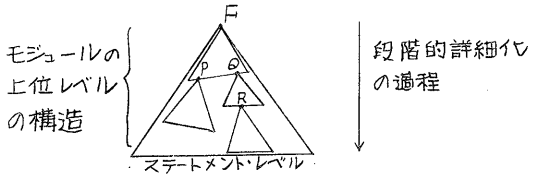
(1) 各バッファを階層構造記述用に宣言し、階層構造記述言語を用いて詳細化のあるレベルの階層の記述を可能にする。また、そのバッファには機能単位名と同じ名前をつけて各階層間の関連を保持する機能。

(2) 段階的詳細化を行なっているプログラム作成者に必要な情報を提供する機能。たとえば、階層間の関連や詳細化の進行状況の提供。

(3) 各バッファの内容を入力として

定義にそってソース・プログラムを構築するトランスレータの機能。

(4) 階層定義用に宣言されたバッファ内の編集にも図5のコマンドが自由に使用できる機能。 つまり、この機能によって、バッファ内にモジュールの上位レベルの構造を構築したり、変更したりできる。



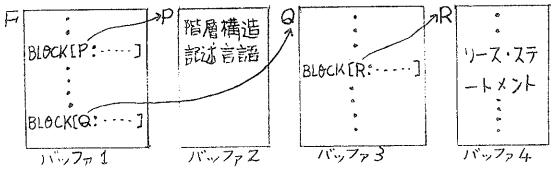


図6 階層構造のエディタ内での表現

以上の機能のために図7に示すコマンドが用意されている。

1) @S(機能単位名) ( )内に示された機能を詳細化することを宣言し、そのために必要なバッファの開設を行なう。

2) @T(〃) 1)と同様。ただし、( )内に示された機能は、プログラム言語のステートメントに詳細化されることを指示。

3) @D(〃) ( )内に示された機能単位以下のレベルの詳細化を無効にする。使用されていたバッファの解放。

4) @R(〃) ( )内に示された機能単位名のクロス・リファレンスの表示。

5) @P(出力バッファ名) 定義されている階層構造の概観を( )内に示されたバッファに出力する。

6) @E(〃)PL 定義されている階層構造に従ってプログラム・モジュールを組みたてて、( )内に示されたバッファに出力する。同時に、文書情報をコメントとして、ソース・プログラム中にうめこむ。

7) #X@ 階層構造定義用に使用されているバッファに関する情報の表示。

8) &S(出力バッファ名) エディタ内に展開されている階層構造の定義を( )内に示されたバッファに退避する。

9) &R カレント・バッファの内容を8)のコマンドで退避したデータとみなして、元の状態に復旧する。

図7 階層構造処理用のコマンド体系

3.5 文書情報の自動収集機能

図6からわかるように、モジュールの上位レベルの構造がエディタ内に保持されている。 この情報は、作成されるモジュールの内部論理に関する文書情報として重要である。 つまり、これを整理してコメントの形式でソーステキスト中にうめこむことによって文書情報の自動収集を可能にしている。

また、付加されるコメントには、詳細化のレベルを表わす番号が、最上位のレベルを"1"として、詳細化が進むにつれて、2,3,…というようにつけられている。

4. 補完器と提示器

4.1 補完器の基本設計

補完器の役割として、以下の二つがあげられる。

(1) 生成器で収集される文書情報と提示器で必要な文書情報のギャップを埋める(不足を補う)。

(2) 文書情報の質を高める。

こうした補完器を実現するには、以下の問題を解しなければならない。

(1) 検証の方法(文書情報の質をどのようにして判断するか)。

(2) 補完器利用者の負担の軽減。

(1)については、補完の対象である生成器の出力を、そのまま解析し、提示器の出力様式で表示し、後に提示器を利用する者の立場で文書情報の検証を行なう方法を採用した。

また、検証の結果、必要な補完、修正を行なうために、以下のオペレーションを用意した。

① 文書情報の変更
② 文書情報の追加
③ 文書情報の消去
④ 文書情報の選択

(2)については、上記①～④のオペレーションの負担を軽減するために、ライトペン、ファンクションキーによる

操作を積極的に採用した。　さらに、変更，追加されるドキュメント情報の入力操作の負担を軽くするため，ローマ字→カナ文字変換機能を組込んだ。

4.2　提示器の基本設計

従来の文書情報は，紙に書かれており、以下のような問題点をもっている。

(1) 量の膨大化にともなって，必要とする情報を参照するのが困難になる。

(2) 文書情報の書式が一定していない（一つのシステムにおいては、標準化されているかもしれないが複数のシステム間では書式が一定していない場合が多い）。

(3) 仕様の変更に文書情報の変更がついていけない。

このような事態を打開する一方法として、提示器が考案された。　この提示器は、文書情報利用者の要求に応じて、プログラム・モジュールの外部仕様書，内部仕様書レベルの情報を提示するシステムである。　なお、文書情報利用者とは，プログラム・モジュールの利用者，保守者，管理者である。

したがって、提示器に要求されることは、以下の点である。

(1) 使い易いこと。

(2) 情報が見易いこと。

(3) 利用・保守・管理時に必要かつ十分な情報が得られること。

(4) プログラム，仕様の変更に強いこと。

(1)については、端末としてキャラクタ・ディスプレイを使用することとし，ライトペンやファンクションキーの積極的な利用をはじめ，操作性を向上させる設計を行なった。

(2)については、情報を視覚的に訴える工夫を行なった。

(3)については、プログラム・モジュールの利用・保守・管理に必要かつ十分な情報の洗い出しを行なった結果、以下の情報を提示することにした。

プログラム・モジュールの利用時には、

① モジュールの入口とその機能についての情報。

② インターフェースの情報。

③ モジュールが使用するファイルに関する情報。

④ 制限事項。

プログラム・モジュールの保守・管理時には，

⑤ 局所的なデータ構造についての情報

⑥ 特記事項

⑦ 名標に関する情報

⑧ 内部論理に関する情報

⑨ ソースリスト

(4)については，文書情報はソースプログラムと共に蓄積され，ソースプログラムを変更する度に，生成器，補完器を通過して蓄積される。　提示器はそれを参照するので、常に最新の文書情報を提示できる。

4.3　操作コマンドと機能

補完器と提示器は、図8に示すコマンド体系をもつ。

1～8は文書情報の提示のコマンドで、提示器で使用する他，補完器で文書情報を検証する時に使用する。

11～15は文書情報の補完に使用される。

以下に1～7の各コマンド毎に提示内容について述べる。

1) EF (Emtry & Function)
- 外部入口名。
- 入口が一次入口か二次入口か。
- 各入口に対応する機能。

2) II (Interface Information)
- 呼出し形式
- パラメータ

・入力・出力・更新パラメータの区別
・名標
・属性
・用途
・関数値の属性

3) FI (File Imformation)
・名標
・属性
・環境 (ENVIRONMENT属性)
・用途
・ファイル名
・ボリューム名

4) DR (Data structure & Restriction)
・データ構造に関する説明
・制限事項
・特記事項

5) LC (Level Chart)

レベル・チャートのレベルとは、プログラム・モジュールのアルゴリズム(内部論理)の詳細化の段階(階層)を表わす。生成器でモジュールを作成するときに、モジュールを一つの機能をもつブラックボックスとして考えて定義した機能にレベル番号"1"をつける。そのブラックボックスを複数のサブブラックボックスに分けたとき、アルゴリズムの詳細化が一段階進んだ(階層のレベルが一つさがった)と考え、それぞれの機能にレベル番号"2"をつけ、さらに各機能を複数個の機能に分割し、…… というように、アルゴリズムの詳細化が進む度にレベル番号を一つずつ増やしていく。

そして、最後には言語のステートメントの集合になる。

このレベル番号と、そのレベルで実現される機能を説明したコメント文をレベル番号に応じて、インデンテーションをつけて表わしたものが、「レベル・チャート」である。

レベル・チャートに提示される情報は次の三つである。
・レベル番号
・コメント文
・コメントに対応するソースプログラムのステートメントの上限と下限(これによって、コメントに対応するソース・テキストを切り出して見ることが可能)

6) ID (IDemtifier)
・定義(宣言)されているステートメント番号
・名標
・属性
・用途
・クロス・リファレンス

7) SL (Source List)
・ソースリスト(ただし生成器が自動作成したり、補完器が補った、コメントをのぞいた純粋のソースリスト)

1	EF	プログラム・モジュールの機能
2	II	インターフェース情報
3	FI	ファイル情報
4	DR	データ構造の説明と制限事項
5	LC	内部論理に関する情報(レベル・チャート)
6	ID	名標に関する情報
7	SL	ソース・リスト
8	LP	1～7の情報のラインプリンタ出力
9	MS	モジュール選択
10	KL	サービス終了
11	PS	補完の省略
12	C	文書情報の変更
13	A	文書情報の追加
14	D	文書情報の消去
15	OK	補完終了

図8 操作コマンド

5. システムの実現

本システムは、本学科に設置されているFACOM 230-45S上で実現されて

いる。　言語は、PL/Iとアセンブリ言語（FASP)を使用し，概要は図9の通りである。

		生成器	提示器・補完器	
			リース解析・提示パターン作成フェーズ	提示・補完フェーズ
ステップ数	PL/I	10,600	3,400	1,500
	FASP	1,000	1,000	300
モジュール数		146	76	30
プログラム・サイズ		20ページ	41ページ	25ページ

図9　システムの実現　　1ページ＝2Kバイト

6. システムの使用例

以下の例題を用いて，使用例を示す。

例題

FORTRANのリースプログラムのキーワード（IF, CALL, READ, ･･etc）の種類別頻度の統計をとるプログラムの作成。

本例題のプログラムは三つのモジュールからなる。　処理の概要は，

(1) TOKEI：メインルーチン。(2)のモジュールを起動してトランザクション・データを作成し，ファイルを更新して，(3)のモジュールを起動して報告書を作成する。

(2) PROCESS：メインルーチンから与えられたキーワードの表を基に，FORTRANプログラムを読みこんで，種類別頻度のデータを作成する。

(3) OUTPUT：報告書を印刷する。

まず生成器を使用して，PROCESSというモジュールを作成する。　はじめに、@S(PROCESS)コマンドで、これから階層を定義することを宣言し，その名前をPROCESSとする。　そうすると、生成器は、同名のバッファを開設する。

そのバッファ内に、図5に示したコマンドを利用して、図4に示した言語で一つの階層を定義していく。　図10は、最上位の階層が定義されたところである。

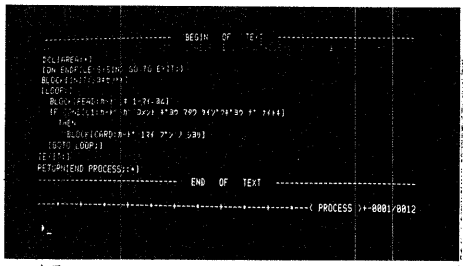


図10

次に、図10の9行目にあるCARDという機能単位を詳細化する。（図11）

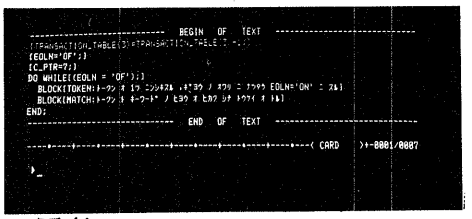


図11

図11の5行目のTOKENを同様に詳細化する。（図12）

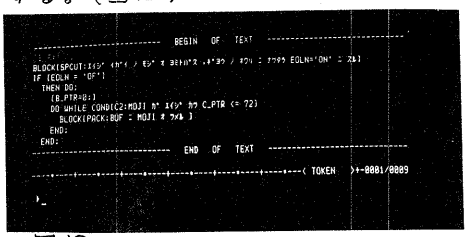


図12

また、図11の6行目のMATCHという機能単位は、プログラム言語のステートメント列に詳細化されると判断した場合は、@T(MATCH)のコマンドで詳細化を宣言する。　生成器の用意したMATCHというバッファにプログラム言語のステートメント列を記述する。

図13

以下同様にして，各階層を定義していく。

図14は、@Rコマンドで機能単位名のクロス・リファレンスを表示しているところである。

図15は、#X@コマンドで、階層構造定義用に使用されているバッファに関する情報を表示しているところ。

図16は、@P(STRC)コマンドで、階層構造の概観を表示しているところ。

これらのコマンドで、段階的詳細化をすすめるのに必要な情報を得ることができる。

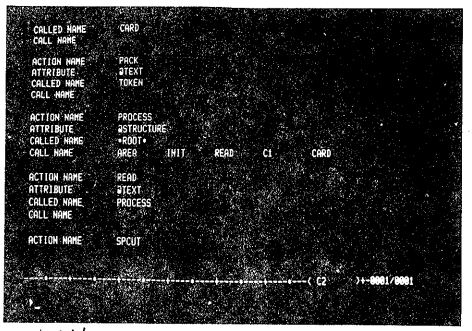


図14

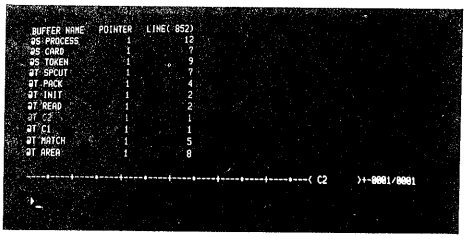


図15

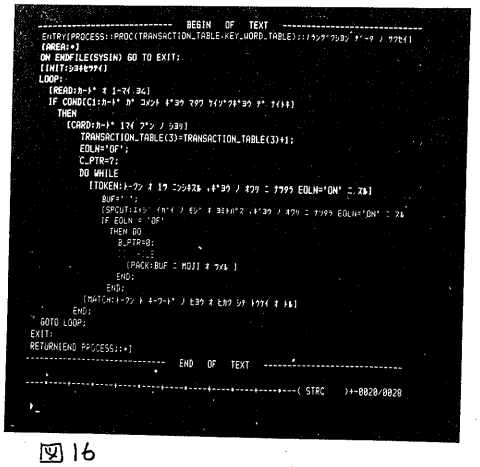


図16

すべての階層の定義が終了したら、

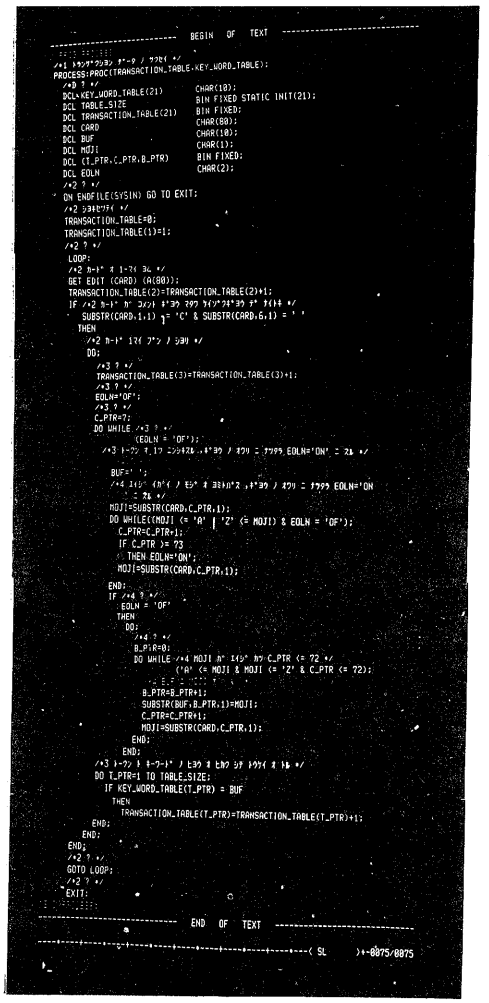


図17

@E(SL)PLコマンドでプログラム・モジュールを構築する。（図17）

次に、補完器を使用する。　図17のモジュールを入力して、そのまま解析を行ない、提示器の出力様式で、表示し、検証を行なう。

図18～図21に、提示パターンの一部を示す。　このパターンを見ながら、図8の11～14のコマンド、ライトペン、ファンクションキーを使用して、文書情報を補完する。　なお、補完が必要と判断された部分は、画面上で「・・・・」の部分が点滅している。

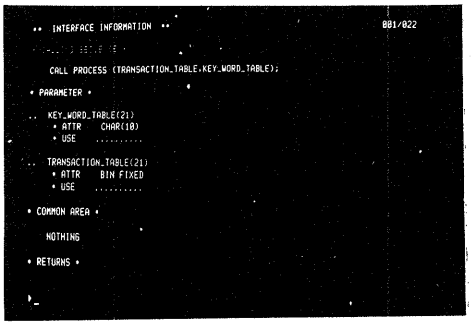


図18 IIコマンド

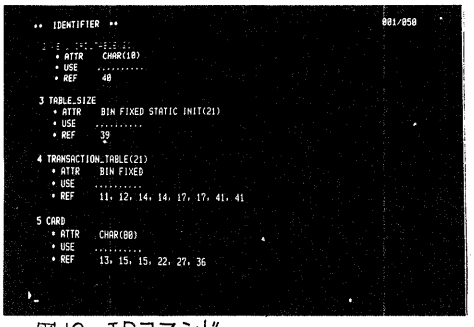


図19 IDコマンド

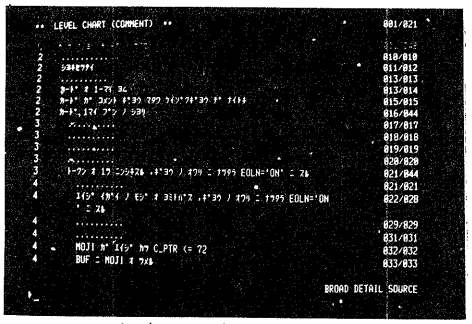


図20 LCコマンド

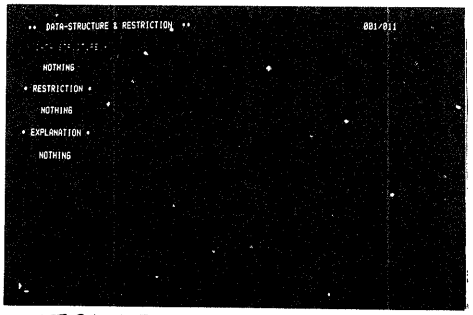


図21 DRコマンド

補完が完了したら，OKコマンドで，文書情報を含んだソースを蓄積する。
次に，提示器を使用する。(図22~30)

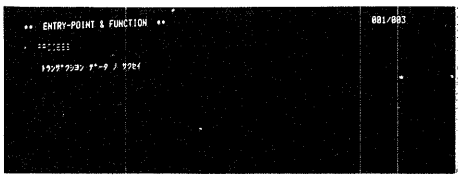


図22 EFコマンドの提示パターン

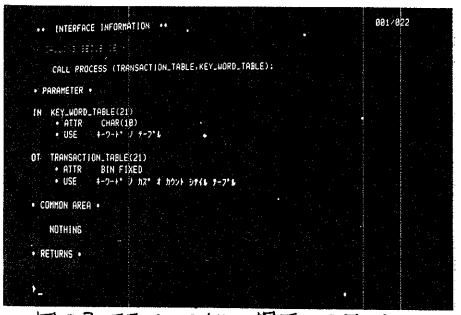


図23 IIコマンドの提示パターン

図24 DRコマンドの提示パターン

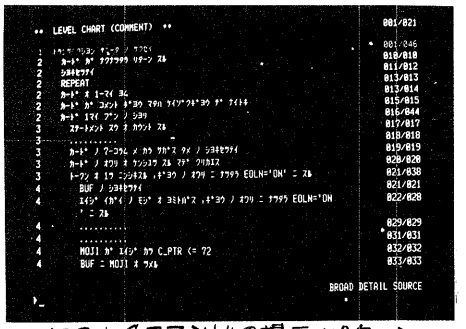


図25 LCコマンドの提示パターン

図25のレベルチャートは，レベル4まで表示されているが，これは任意に選択できる。 図26では，レベ

ル2まで表示されている。　また、図25の5行目の「カード　オ　1ーマイヨム」に対応するソースリストを見たい時は、5行目にポインタを移動させ画面右下の「SOURCE」をライトペンで指示すると、図27のようにソースリストが表示される。

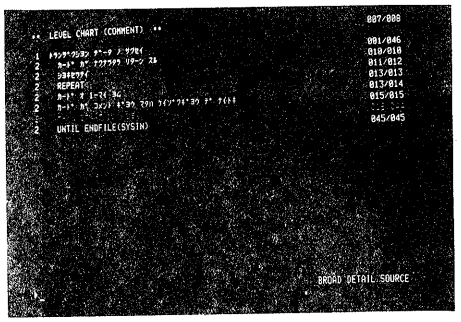


図26 LCコマンドの提示パターン

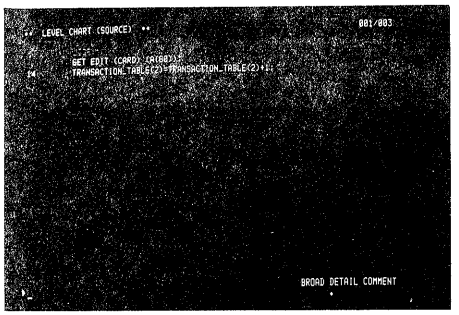


図27

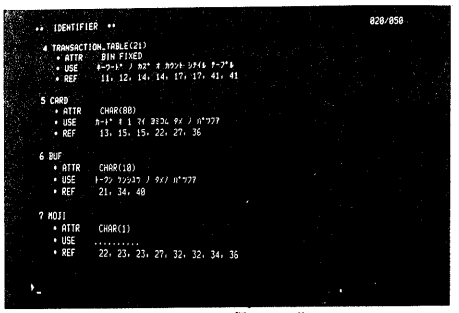


図28 IDコマンドの提示パターン

図30は、モジュール TOKEIで使用されているファイルに関する情報である。

7. おわりに

本システムの実現によって、段階的詳細化法によるプログラミングの支援

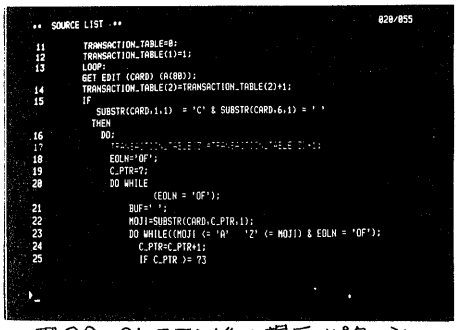


図29 SLコマンドの提示パターン

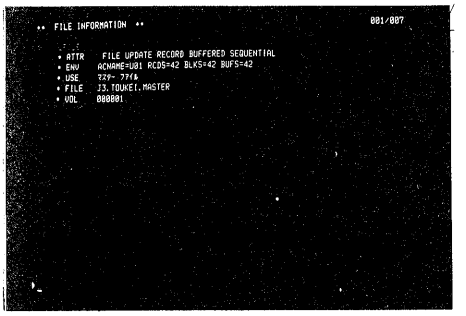


図30 FIコマンドの提示パターン

と文書情報の自動収集。　また、必要かつ十分な内容の文書情報を軽い負担で、作成、補完、参照でき、プログラム理解時の立ち上がり時間を大幅に短縮することができた。

しかし、ドキュメント情報の質をあるレベル以上に保つことに関しては、生成、補完とも、人間の意志決定にまかされており、客観的な尺度が存在しないという問題がある。　また、提示情報がキャラクタ・ディスプレイとラインプリンタにしか出力できないなど、手書の文書に比べて表現の柔軟性に欠ける。

今後は、本報告で述べた方式に基づいて、当面、システムの設計段階の文書情報を対象として、ツールの開発を行なっていく。

〔参考文献〕

[1] 落水浩一郎、福村和悦：「リーダビリティ・チェッカ、アナライザ方式によるプログラム文書化」情報処理学会第18回全国大会(1977)

[2] H.D.Mills：「Syntax-Directed Documentation For PL360」CACM 13,4 P216 (1970)