解　説

ソフトウェア開発における人間的要素

プログラミング言語とその会話型支援環境†

中　所　武　司††

1. はじめに

プログラミング言語[1]は，最も古くからあるプログラミングツールの1つであり，これまで種々の言語が開発されてきたが，その人間工学的側面としては，次の2点が重要である．

（1）　記述水準：対象とする問題や解法の記述容易性．

（2）　支援環境：そのプログラムの開発容易性．

まず，プログラミング言語の第1の目的は，人間と計算機とのインタフェースをできるだけ人間側に近づけることである．そのため，言語は記述水準の高級化という形で発展してきたが，それは実用的視点でみると次の3段階に分けて考えることができる．すなわち，第1段階では記述量の削減，第2段階ではプログラミング方法論の反映，第3段階では非手続き的問題記述形式の実現を主目的としてきた．詳細は次章に譲るが，例えば各々の代表的言語として，Fortran, Ada, Prolog を掲げることができる．

一方，このような言語の発展に比して，プログラム開発支援環境の立遅れがソフトウェア工学の分野で問題となっている．すなわち，プログラムの開発はプログラムの記述を以って終了するものではなく，その後に開発費用の約半分を占めるプログラムの検証作業が必要である．そのため，言語の高級化によるプログラムの生産性向上には限度があり，むしろその開発支援環境が重要なわけである[2]．そこで，本文では，言語と支援環境の現状の問題およびその解決のための特定言語向き統合プログラム開発支援環境の動向について述べる．

2. 現状の問題

2.1 言　語

1950年代後半から60年代前半にかけて，Fortran, Cobol, Algol, PL/I などの手続き向き言語が次々と開発され，記述量の削減によるプログラムの生産性向上をもたらした．しかしながら，70年代にソフトウェアの大量化と大規模化が進み，その保守，拡張の費用が増大するにつれて，プログラムの書きやすさより読みやすさの方が重要になってきたが，この点では従来言語は不十分なものであった．

そこで，最近の言語設計では，これまでのような機械語への展開率を向上させる「量的高級化」よりも，構造化プログラミングに代表されるようなプログラミング方法論を反映した「質的高級化」を実現している．すなわち，制御構造に関しては，goto 文のような無条件分岐を排するとともに，プログラムの段階的詳細化によるトップダウン開発を支援するモジュール構造の導入により，またデータ構造に関しては，豊富な基本データ型や新しいデータ型の定義機能，あるいはデータ型とその操作手続きをまとめて定義するデータ抽象化機能の導入などにより読みやすいプログラムを作成できるようにしている．例えば，過渡的な Pascal の他，CLU, Mesa, Iota, Ada などが抽象化機構[3]を備えており，筆者らも SPL[4]を開発した経験がある．

したがって，このような言語はそこに具現化された方法論も含めて修得する必要がある．ところが，例えば米国国防総省が標準言語として開発している Ada は，抽象化機構としてのパッケージ機能のほか，リアルタイム処理のための非同期処理や例外処理機能が充実している反面，その理解は容易でない．このように言語が複雑化する一方で，プログラマの増加とともにその平均的能力はむしろ低下の傾向にあると思われるため，言語とその使用者の間の教育的ギャップが大きくなるという問題がある．このような人間工学的矛盾

† Languages and Programming Support Environments by Takeshi CHUSHO (Systems Development Laboratory, Hitachi Ltd.).

†† (株)日立製作所システム開発研究所

への対策として，3で述べるような，統合的プログラム開発支援環境の充実が必要であると思われる．

2.2 プログラミングツール

プログラムの開発過程では種々のツールが用いられる．その基本的なものとしては，プログラムの作成と修正に用いるエディタ，そのソースプログラムを計算機で実行可能な機械語プログラムに変換するコンパイラ，それを実行しながら誤り検出するためのデバッガなどがあるが，その機能はまだ十分ではない．

エディタは従来の行エディタに代り，最近は画面エディタが普及しはじめて操作性が向上した．しかしながら，これらはプログラムを単なる文字列とみなして編集するテキストエディタであるため，そのユーザはプログラムを思考対象とするときは意味のある文章とみる一方，操作対象とするときは文字列とみる必要がある．このような意味的ギャップはプログラミング効率の低下や誤り発生の原因となっており，最近では構文要素単位の編集機能を有する構造エディタが研究，開発されているが，詳細は後述する．

次にコンパイラについては，今まで処理性を重視してアセンブラで記述していたようなプログラムが生産性，移行性の良い高級言語で記述されるようになるとともに，最適化処理の強力なコンパイラが開発されている．しかしながら，プログラム開発時には，頻繁にプログラムの修正と再コンパイルを繰り返すため，最適化機能よりもむしろデバッグ機能やわかりやすいエラーメッセージ出力，あるいはプログラム解析による誤りの自動検出などの機能が有用であるが，現在のコンパイラにはこのような開発支援機能が不十分である．

またデバッガについては，従来の機械語レベルでのデータ値の表示と設定，実行の中断や命令トレースなどの機能を備えたものに代って，最近ではソースプログラム内の変数名や名札を用いて指示できるシンボリックデバッガが普及してきた．しかし，これらは誤りの原因究明のためのデバッグ支援が主であり，アセンブラから高級言語への移行とともにより重要になってきた系統的テスト支援機能が弱い．

プログラム開発時に用いるツールとしては，このほかにもプログラムの静的解析や動的解析を行うもの，ドキュメント生成用ツール，変更歴管理ツールなど数多いが，個々のユーザがプログラム開発時に用いる計算機システムや記述言語を特定したときに使用可能なツールはまだ少ない．また複数のツールが使用可能な場合でも，各ツールが個別に開発されているため，対象とする言語の仕様が少し異なるとか，双方の入出力ファイルの形式が違うとか，ユーザインタフェースの不統一などの問題が生じやすい．

3. 統合的プログラミング環境

3.1 統合化の条件

2で述べたような言語とプログラミングツールに関する現状の問題を解決するためには，

（a） 各ツールの有機的結合による1システム化

（b） 各ツールの言語適応化による高機能化

の方針に沿った統合的プログラミング環境が望ましい．すなわち，これまで個々に開発されてきた単品ツールの結合と対象プログラムの記述言語に適応した支援機能の付与により強力な支援環境を構築できる．特に第1方針の「有機的結合」は次に述べるような，方法論，技法，データ，ユーザインタフェースの共有によって実現しうる．

（1） プログラミング方法論の共有

プログラミングの主要な作業であるプログラムの設計，作成，検証の間に一貫性を持たせるために，各部分の支援ツールに同一のプログラミング方法論を反映させる．例えば，段階的詳細化法やデータ抽象化法に基づいて設計されたプログラムモジュールはその支援機能を有する言語で記述できること，さらにその設計法を誘導する構造エディタや導かれたプログラムのモジュール構成に適した構造テスト機能を利用できることが望ましい．

（2） プログラム解析技法の共有

特定言語向きツールはプログラム解析処理を伴うが，ツール間で共有な部分も多い．例えば，構文解析はコンパイラの他にも，構造エディタやテスト網ら率計測用コード埋込みツールでも必要である．そこで，構文解析やフロー解析などの処理を共有化することにより，各ツールの高機能化が可能になる他，対象とする言語の仕様がツール間でくい違うのも防止できる．

（3） データの共有

プログラムや各ツールの入出力データなどをデータベース化し，それらの間の関係を保持しておくことにより，ツール間インタフェースの不一致を防止できる他，プログラム変更に伴う再コンパイルや再テストの自動化，あるいは関連データの問合せなどの機能を付与できる．

（4） ユーザインタフェースの共有

ツールごとにコマンド言語や端末操作法が異なっていてはシステムとしては使いづらい．ユーザインタフェースの一様性は統合化の必須条件である．

以上に述べたプログラミング環境の単一システム化の4条件から，統合化の第2方針である言語適応化は必然的に導かれる．なお，統合化の条件としてのユーザインタフェースとデータベースの重要性については，W. E. Howden[5] や A. I. Wasserman[6] らも指摘している．

3.2 事　例

3.2.1 汎用プログラミング環境

前節で述べた統合化条件の一部を満たし，よく普及している既存システムとして Unix☆ がある[7]．これは米国の Bell 研究所で1970年頃に開発されたミニコン用タイムシェアリングシステムである．基本的にはプログラミングツールは個別に開発したものの集りであるが，ユーザインタフェースとファイル構造の統一化によって強力な会話型プログラミング環境を構築しており，70年代の優れたシステムと言える．すなわち，Shell と呼ばれるコマンドインタプリタを経由して個々の機能を実行する方式とコマンド間の入出力ファイル指定を不要にするパイプと呼ばれるコマンド結合方式およびバイト列形式のファイルを木構造で表現する簡潔なファイルシステムなどにより使い勝手の良いシステムとなっている．

3.2.2 特定言語向きプログラミング環境

次に特定の言語を対象とした統合プログラミング環境をいくつか紹介する．これらのほとんどは構造エディタとデバッガを備えているが，その特徴を**表-1**にまとめておく．

（1） **Interlisp**[8),9)]

リスト処理用言語 LISP の処理系の1つとして，MACLISPなどとともに広く用いられている Interlisp は単なる言語処理系ではなく，構造エディタやデバッガを内蔵した統合プログラミング環境である．ここでは幾つかの特徴的な開発支援機能について述べる．

まず，先駆的な構造エディタは構文要素単位の操作機能を有する．すなわち，プログラム内の操作対象はつねにその部分式であり，ネスト構造をしたリストの上位や下位の式への移動や式単位の削除，挿入，置換，探索ができる．またプログラム構造を表現している括弧の削除，挿入，移動コマンドではつねに左右の括弧の対が保たれるようにしている他，プリティプリンティングの機能も用意されている．

例えば，次のようなプログラムを考えよう．

```
[LAMBDA (X) Y (COND ((NULL X) Z)
    (T (CONS (CAR) (APPEND (CDR X Y]
```

これは2つのリストを統合する関数 append の定義を誤ったもので，次のような構造エディタのコマンド列で修正できる．

① (3)　　——第3要素Yの削除
② (2 (X Y))　——第2要素(X)を(X Y)で置換
③ (R Z Y)　——すべてのZをYで置換
④ F CAR　——CAR の探索
⑤ (N X)　——リスト(CAR)にXを追加挿入
⑥ NX　——次の要素に移動
⑦ (RI 2 2)　——第2要素の第2要素右に）挿入
⑧ PP　——プリティプリンテング指示

このとき，修正されたプログラムは次のようになる．

```
[LAMBDA (X Y)
    (COND
        ((NULL X)
```

表-1 特定言語向きプログラミング環境とその構造エディタの一覧表

No.	システム名*	開発元	開発時期	対象言語	内部表現	言語処理系	その他の特徴
1	Interlisp	Xerox BBN	1973	Lisp	テキスト	インタプリタ コンパイラ	・プログラム構造（括弧の対）の変更容易
2	MENTOR	INRIA	1975	Pascal	抽象構文木	インタプリタ	・ユーザインタフェースは木操作言語
3	Cornell Program Synthesizer	Cornell Univ.	1978	PL/CS	構文木と逆ポーランド記法	インタプリタ	・段階的詳細化支援 ・未完成プログラムのテスト実行
4	IPE (ALOE)	CMU	1979	GC	構文木	コンパイラ	・プログラム作成は BNF 文法規則の置換のみ
5	Iota Programming System	京大	1980	Iota	構文木	インタプリタ コンパイラ	・テキストエディタと機能分担
6	APSE	DOD	開発中	Ada	?	インタプリタ コンパイラ	・要求仕様のみで詳細不明
7	CROPS (PARSE)	日立	開発中	SPL	構文木	コンパイラ	・文レベル以下はテキスト入力，編集も可能

*：（ ）内は構造エディタ名

☆ Unix は Bell 研究所の登録商標

```
      Y)
   (T (CONS (CAR X)
            (APPEND (CDR X) Y]
```

次にデバッグ機能として，関数引用時や関数内のラベルの前後などに中断点を設定し，ユーザ指定の中断条件成立時に一時中断したり，関数引用時にその引数やユーザ指定値，また関数値計算後にはその値をプリントするトレース機能などがある．さらに，綴り字誤りやLISPに多い括弧誤りなどを自動修正するDWIM (Do-What-I-Mean) 機能がある．

（2） **MENTOR**[10),11)]

MENTOR は，プログラムの設計，作成，検証，保守を支援する会話型プログラミング環境を目ざしたシステムで，最初に Pascal 用構造エディタが開発されている．これは，抽象構文木をその木操作言語を用いて編集する方式で，例えば，次のような Pascal プログラムは**図-1** の抽象構文木として保存される．

```
if X>0 then P(X, Y)
        else begin Y:=Y*2; X=0 end
```

木操作言語はコマンド形式で，操作対象とする節の位置指定やその上下左右への移動，および指定位置での部分木の削除，挿入，置換ができ，構文誤りを生じるような指示は受けつけない． 例えば，図-1 の抽象構文木を操作して次のようなプログラムに変更する場合を考える．

```
if X>0 then begin
                Y:=Y*2; P(X, Y); X:=0
                end
        else  Z:=0
```

編集コマンドは次のようになる．

① S2 X S3 ——then 句と else 句の交換
② S2 S1 I S3 ——then 句に else 句を複写挿入
③ S3 C & ——else 句を次の 端末入力で置換
④ Z:=0; ——端末入力テキスト
⑤ P ——アンパースしてプリント出力

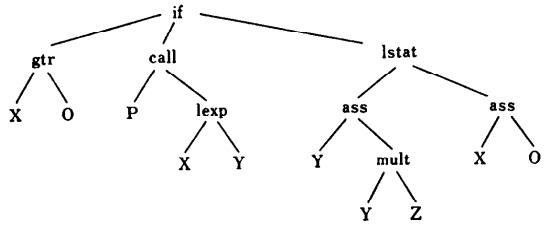

図-1 MENTOR の抽象構文木の例

なお，Sn は n 番目の子供を指すので，②は then 句の1番目の文の後への else 句の複写を意味する．

このほかにも，抽象構文木を上位のnレベル分だけアンパースしてテキスト表示したり，あるパターンを探索する機能がある．

（3） **Cornell Program Synthesizer**[12),13)]

これは，PL/I 風の教育用言語 PL/CS を対象にプログラムの作成，編集，実行，デバッグを支援する会話型プログラミング環境で，すでにコーネル大のほかにも幾つかの大学で教育用に使われている．プログラムの作成は，代入文などの一部の文を除いた文や宣言以上の構文要素は構文テンプレート挿入用コマンドを用い，その他は直接テキストを入力する．これらの入力可能位置は placeholder と呼ばれる非終端記号の表示位置である．構文要素単位の編集機能として，このほかにも要素の削除や削除した要素をファイルに保存しておき任意の所へ挿入するためのコマンドや隣接する要素，同一レベルでの前後の要素，上位または最上位の要素へのカーソル移動用コマンドがある．

次に本システムの主な目的の1つである段階的詳細化の例として，以下のようなプログラムの作成手順を示しておく．

```
/* sum up two data */
adder: PROCEDURE OPTIONS (MAIN);
    DECLARE (x. y) FIXED;
    GET LIST (x. y) ;
    DO WHILE (x>0) ;
        {statement}
        END;
    END adder;
```

まず，.ed adder と .main をコマンド入力すると次のテンプレートが表示される．

```
/*comment*/
adder: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END adder;
```

そこで以下のコマンド列を入力する．

① sum up two data——ヘッダコメント
② .fx ——FIXED 型の宣言テンプレート挿入
③ x, y ——宣言する変数名の指定
④ 空行 ——宣言終了（非終端記号の削除）
⑤ .g ——GET LIST 文のテンプレート挿入
⑥ x, y ——その入力変数リストの指定

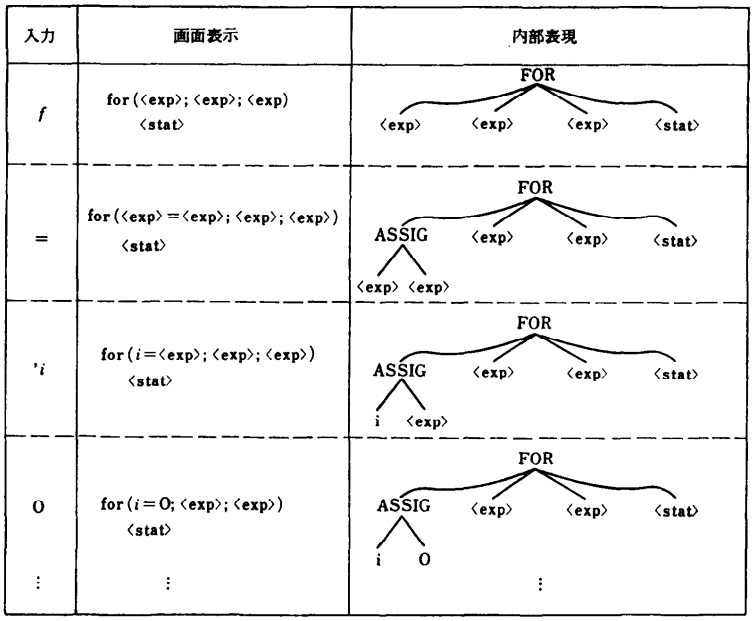

図-2 IPE によるプログラム作成例

⑦ .dw ——DO WHILE 文のテンプレート挿入
⑧ x>0 ——条件式のテキスト挿入

このほか，コメント用コマンドを介して文の列を挿入すると桁下げ表示したり，その文の列の代りにコメントだけ表示する機能や対応する宣言がないと変数参照部分が高輝度表示になるような型チェック機能なども特徴的である．また，デバッグ機能として，未完成プログラムを実行しながらプログラムを詳細化する機能や，制御トレースやデータトレースの常時画面表示，構文要素単位の部分実行などの機能がある．

(4) Incremental Programming Environment[14),15)]

本システムは，CMU のソフトウェア開発環境プロジェクト Gandalf の中で開発された，単一プログラムを扱う単一プログラマ用プログラミング環境である．対象言語はもともとは Ada で，プログラムの内部表現には Ada 用 TCOL が用いられているが，最初に開発されたものはCを変形した GC である．

その構造エディタは ALOE と呼ばれ，プログラムの作成はプログラムテンプレートの中のメタノード(非終端記号) にテンプレートまたは識別子や定数を挿入することを繰り返して行う．図-2 に for 文を詳細化していく例を示す．編集機能としては構文木の部分木を構成する構文要素に対する削除，挿入，探索コマンドなどがある．この ALOE は構造エディタ生成システムによって各言語対応に作られるようになっており，そのための文法記述には，構文の他に各メタノードに挿入可能な構文要素集合，プリティプリンティング情報，アクションルーチン名などが含まれる．

プログラムの実行は構文木をコンパイルして行うが，そのときに条件付中断設定や変数値表示用コードあるいは不完全な手続き用スタブコードなどを埋め込む方式でデバッグを支援する．

(5) Iota プログラミングシステム[16),17)]

本システムは京大の中島らによって開発された抽象化機構を持つモジュラ言語 Iota用の会話型プログラミング環境である．これはテキストエディタと構造エディタの両方を有し，それぞれ構文解析系および意味解析系と連動してその検出エラーの修正に用いられる．以下に最小値を選ぶ IF 文の修正手順の例を示す．

① s IF l ——'IF' を探索してプリント
IF lesseq (x, y) THEN z : =x ELSE z : =x END IF
② 3 ——3番目の部分木へポインタ移動
③ m ——その修正を指示
/\z : =x
④ 3 f d iy$——3文字先の文字削除と 'y' 挿入
z : =y/ \
⑤ z ——テキスト編集終了

ここで①～③は構造エディタコマンド，④と⑤はテキストエディタコマンドで /\ はポインタを示す．この後，構造エディタがこの修正コードを受け取り，その終了時に意味解析系が呼ばれてまとめて処理される．構造エディタ用コマンドとしては構文木上のポインタ移動用に6種類，構文木の修正用に5種類のほか，部分木のプリティプリング機能などがある．

プログラム実行支援機能としては，分割コンパイル方式の処理系のほか，インタプリタ方式のデバッガがある．

（6） **Ada プログラミング支援環境（APSE）**[18]

APSE は，Ada の開発方法と同じようにまず米国国防総省（DOD）がその要求仕様を明確にする作業を行い，その最終版の要求仕様書 STONEMAN に基づき，幾つかの組織で開発が行われている．この仕様書によれば，APSE は3階層から成り，最下層に核部分 KAPSE，中間に最小ツールセット MAPSE がある．

構造エディタは最後の章で簡単に言及されており，言語要素のテンプレートや入力チェックにより構文的に正しいプログラムの作成を支援する．なおテキストエディタは MAPSE に含まれる．またソーステキストのレベルで使用できるテストデバッグツールを必要としている．

（7） **SPL プログラミング環境（CROPS）**[24]

筆者らも1975年に段階的詳細化とデータ抽象化を支援する構造化プログラミング言語 SPL[4] を開発後，種々の支援ツールを開発してきたが，最近その統合化システムのユーザインタフェースとなる構造エディタ PARSE (Production And Reduction Screen Editor) を開発中である．当初は構造化プログラミングの視点からのプログラム構造の再構成とソースレベルでの最適化を支援する SPL 専用エディタを考え，第3回 UJCC (1978) 等で報告[19]したが，当時はまだビデオ端末があまり普及しておらず，実現に至らなかった．

現在開発中の PARSE は，これまでの SPL の適用経験を踏まえて，言語に反映させたプログラミング方法論を支援することを主目的としている．主な機能は，文法誘導による段階的詳細化，構造化コーディング，構文要素単位の編集機能などである．前述の他システムと比べて，文のレベル以下のプログラム入力はテキストもテンプレートも許していることやコードレビュ用の問合せ機能などに特徴がある．

3.3 考 察

統合プログラミング環境の事例を幾つか紹介したが，実際に大規模ソフトウェアへの適用実績を有するものは，Interlisp だけと思われる．その理由は，対象言語の Lisp はインタプリタ方式のシステムが多く統合化が容易であること，言語構造が単純で構造エディタに適していることなどから早期開発が可能であったためと思われる．

一方，手続き向き言語を対象としたものはそのコンパイラが単独で開発されてきたため統合化が遅れている．ここに掲げた事例も既開発のものはまだ十分に技術が熟しているとはいえず，研究課題は多い．

例えば構造エディタについてみると，

（1） プログラム構造の認識容易なビデオ端末の使用，

（2） ユーザには木構造形式の内部表現を意識させない，

（3） あるレベル以下の構文要素の入力には初心者用のテンプレート方式と熟練者用のテキスト方式がともに可，

などが必須条件と思われるが，既存のもの（表-1(1)～(5)）で3条件を満たすものはない．

一方，**3.1** で述べた統合化の4条件についてみると，ユーザインタフェースの一様化は当然として他の条件はまだ不十分である．その理由は統合システムの必須機能であるエディタとデバッガを含んでいるが，ソフトウェア工学的ツールをあまり備えていないためである．この点では，現在開発中の APSE が本格的な統合プログラミング環境になることが期待されるが，要求仕様書からは不明である．

4. おわりに

以上，プログラム開発における人間と計算機とのインタフェースである言語とその支援環境について，人間工学的側面からの問題点とその解決のための特定言語向き統合プログラミング環境について述べた．特に後者については構造エディタを中心に先駆的システムを幾つか紹介したが，まだ機能的に熟成しているとはいえず，議論[20)~23)]も多い．しかしながら，3.1で述べたように，種々のプログラミングツールの統合化と言語適応化による高機能化が1つの解決策となるのは確かであると思われるので，今後の実用化研究が注目される．

参 考 文 献

1) 特集—プログラミング言語の最近の動向，情報処理，Vol.22，No.6 (1981)．

2) Standish, T. A.: The Importance of Ada Programming Support Environments, Proc. NCC 82, pp. 333-339 (1982).
3) 鳥居宏次, 二木厚吉, 真野芳久: プログラミング方法論の展望, 情報処理, Vol. 20, No. 1, pp. 22-43 (1979).
4) 中所武司, 野木兼六, 林 利弘, 森 清三: 段階的詳細化, データ抽象化を支援する言語 SPL のコンパイル技法, 情報処理学会論文誌, Vol. 21, No. 3, pp. 223-229 (1980).
5) Howden, W. E.: Contemporary Software Development Environments, Comm. ACM, Vol. 25, No. 5, pp. 318-329 (1982).
6) Wasserman, A. I.: Automated Development Environments, Computer, Vol. 14, No. 4, pp. 7-10 (1981).
7) Kernighan, B. W. and Mashey, J. R.: The UNIX Programming Environment, *ibid.*, pp. 12-22.
8) Sandewall, E.: Programming in an Interactive Environment: the "Lisp" Experience, Comput. Surv., Vol. 10, No. 1, pp. 35-71 (1978).
9) Teitelman, W.: Interlisp Reference Manual: Xerox PARC (1975).
10) Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B. and Levy, J. J.: A Structure-oriented Program Editor, Research Report 114, IRIA-Laboria, France (1975).
11) Donzeau-Gouge, V., Huet, G., Karn, G. and Lang, B.: Programming Environments Based on Structured Editors: the MENTOR Experience, Research Report 26, INRIA, France (1980).
12) Teitelbaum, T. and Reps, T.: The Cornell Program Synthesizer: a Syntax-directed Programming Environment, Comm. ACM, Vol. 24, No. 9, pp. 563-573 (1981).
13) Teitelbaum, T.: *The Cornell Program Synthesizer: a Tutorial Introduction*, Cornell Univ., New York (1980).
14) Medina-Mora, R. and Feiler, P. H.: An Incremental Programming Environment, IEEE Trans. Softw. Eng., Vol. SE-7, No. 5, pp. 472-482 (1981).
15) Medina-Mora, R. and Notkin, D. S.: *ALOE Users' and Implementors' Guide*, CMU, Pittsburgh (1981).
16) Nakajima, R., Yuasa, T. and Kojima, K.: The Iota Programming System—a Support System for Hierarcgical and Modular Programming, Proc. IFIP 80, pp. 299-304 (1980).
17) Yuasa, T. and Nakajima, R.: Iota Modular Programming System, IEEE Trans. Softw. Eng. (to appear).
18) Requirements for Ada Programming Support Environments "STONEMAN", US Dept. of Defense (1980).
19) Chusho, T. and Hayashi, T.: Two-stage Programming: Interactive Optimization After Structured Programming, Proc. 3rd UJCC, pp. 171-175 (1978).
20) Waters, R. C.: Program Editors Should Not Abandon Text Oriented Commands, ACM SIGPLAN Notices, Vol. 17, No. 7. pp. 39-46 (1982).
21) Cohen, E.: Text-Oriented Structure Commands for Structure Editors, ACM SIGPLAN Notices, Vol. 17, No. 11, pp. 45-48 (1982).
22) Meyrowitz, N. and Dam, A. V.: Interactive Editing Systems, Comput. Surv., Vol. 14, No. 3, pp. 321-415 (1982).
23) 和田英一: エディタとテキスト処理 10, bit, Vol. 15, No. 1, pp. 96-102 (1983).
24) Chusho, T., Watanabe, T. and Hayashi, T.: A Language-Adaptive Programming Environment Based on a Program Analyzer and a Structure Editor, 9th World Computer Congress IFIP '83 (to appear).

(昭和58年2月7日受付)