総　論

構造エディタ†

原田賢一††

1. はじめに

プログラムの開発において，プログラムテキストの入力と修正のためのツールとして，行あるいは画面を編集単位としたテキストエディタ[1]が，現在，最も広く使用されている．これらのエディタは，基本的には，文字の列という簡単な構造をもつテキストを編集の対象としている．そのために，用途はプログラムの編集だけでなく，入力データの作成，さらには一般の文書の編集など広範囲に及んでいる．このような汎用を目的としたエディタに対して，編集の対象を特定のプログラミング言語，あるいは設計用の言語や図式を用いて記述されるものに限定し，専用のエディタを開発する試みが最近，活発になされている[21),34),35)]これらのエディタにおいては，テキストを単なる文字の列としてではなく，言語や図式に固有な構造，あるいはそれに近い形の構造をもつものとして扱っている．使用者から見ての編集操作は，内部的にはこれらの構造に対する操作として実現される．このような特徴をもつエディタを，以下，一般に**構造エディタ** (structure editor) と呼ぶ．

構造エディタの具体例については，各論で詳しく述べられるので，ここでは構造エディタを特徴づけるテキストの入力方式と表示方法，編集機能，および実現について解説する．

2. 構造エディタの分類

エディタは，使用目的，編集対象，編集方式，表示方式，あるいは実現の形態（単独のツールあるいは環境の一構成要素かどうか）など，種々の側面から分類することができる．まず，支援の目的によって大別するのが一般的であろう．構造エディタの場合には次のように分けることができる．

（1）特定のプログラミング言語によるテキストの作成を支援するエディタ：Pascal 用の MENTOR[7)-10)]および Gandalf[13),14)]プロジェクトで開発された PASCAL．aloe, INTERLISP システムのエディタ[17)-18)], PL/I のサブセット用の Cornell Program Synthesizer (以下，C. P. S. と略す)[2)-4)]などがある[23),24),27)]．

（2）段階的詳細化によるプログラムの作成を支援するエディタ：各詳細化のレベルにおいて記述されるテキストを1つの単位として，抽象的な記述とその詳細の記述との関係を保持する．処理系から独立して単独のツールとして存在する場合には，原始プログラムを生成するためのコマンドが用意されている．それによって，抽象的な記述は手続き呼出し，あるいは対応する詳細の記述で置き換えられ，特定の言語で書かれたプログラムが得られる．このようなエディタには，FORTRAN 用の TDSS[28)], SPL 用の PARSE[25),26)], Iota システム[31),32)], Pascal および Basic 用の DIMED[19),20)]などがある[22)]．

（3）プログラム設計を支援するエディタ：特定の言語によるプログラムテキストの作成よりも，論理設計やモジュール設計に関する文書の作成と管理を主な目的とする．その多くは，トップダウン設計法や段階的詳細化によるプログラム設計を支援するために開発されている．これらには，πエディタ[16)], DUAL[15)], HCP[29)]チャートプロセッサ[30)]などがある．詳細化の最下位のレベルには任意の言語のテキストが記述でき，テキストの置換や変形用のコマンドを用いて，機械的にプログラムテキストを生成できる機能をもつものもある．

以上のエディタは，特徴のとらえ方によって，**プログラムエディタ**，**言語向きエディタ** (language oriented editor)，**構文主導型エディタ** (syntax directed editor) と呼ばれることがある．

構造エディタの具体例については，各論で詳しく述べられるので，ここでは構造エディタを特徴づけるテ

† Structure Editors by Ken'ichi HARADA (Institute of Information Science, Keio University).

†† 慶応義塾大学情報科学研究所

キストの入力方法と表示方法，編集機能および実現について解説する．以下の章では，タイプの異なる2つの代表的なプログラムエディタ，MENTOR と C.P.S. を例としてとり上げる．まず，両者について簡単に紹介する．

MENTOR はフランスの INRIA (Institut National de Recherche en Informatique et en Automatique) において，Pascal のためのプログラム作成支援環境を構成する1つのツールとして開発されたものである．種々の型の端末で利用できるように，行単位の入力および表示方式を採用している．現在のところ，このエディタだけが完成し，使用されている．

一方，C.P.S. は Cornell 大学の T. Teitelbaum らによって開発されたプログラム作成支援システムで，PL/I のサブセットである PL/CS によるプログラムを対象としている．このシステムはマイクロコンピュータ TERAK (LSI11) 上に実現され，プログラミングの教育に使用されている．このシステムはエディタだけでなく，プログラムの実行とデバッグのための通訳系 (interpreter) を含む．編集には多重窓 (multi-window) をもつ画面方式を採用している．

次章以降で詳しく述べるが，両者の特徴の比較を**表-1** に示す．これらの特徴は，多くの構造エディタ (とくに，特定のプログラミング言語を対象としたもの) に共通したものである．

3. テキストの入力方式

構造エディタは，入力方式に関して，テキストエディタと同様に原始テキスト (source text) を直接，与える方法と，テンプレート (template) と呼ぶ一種の型紙を指定する方法とに分けることができる．MENTOR は前者，C.P.S. は後者に属す．

3.1 テキストを直接入力する方法

このタイプのエディタは言語処理系と同様に，原始プログラムのテキストを文字列として読み込んで，内部ではそれを構文解析し，構文木を作成していく．この過程で構文上の誤りが検出されれば，ただちにメッセージが表示される．

MENTOR では，編集したい個所が簡単に指定できるように，また内部の処理が簡単にできるように，Pascal の構文規則によって定義される構造よりも簡単な構造をもつ木を生成している．この木をとくに**抽象構文木** (abstract syntax tree) という．**図-1** にその例を示す．編集中に，プログラムの一部あるいは全部のテキストを表示することが必要になった場合には，この抽象構文木からもとのテキストのイメージを組み立てて (unparsing)，表示する．

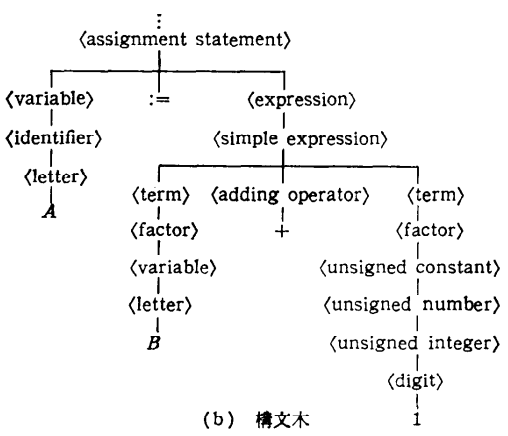
3.2 テンプレートを用いる方法

対象とする言語の各構文規則に対する，一種の型紙あるいは型板に相当するものを**テンプレート**という．使用者は，エディタに用意されたテンプレートを選択し，必要な個所にそれを埋め込むことによって，プログラムを作り上げていく．例として，C.P.S. における IF 文のテンプレートを**図-2** に示す．この中で，

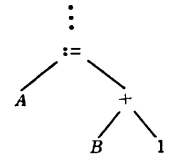
表-1 構造エディタの比較

	C.P.S.	MENTOR
形 態	環境の一要素	単独ツール
使用環境	個 人	TSS による多人数
主な使用目的	教育用	研究用
計算機	マイクロコンピュータ	中型機
プログラムの作成	テンプレートの置換	テキストを直接入力
表 示	画 面	行
表示の省略技法	注釈で代表	ホロフラスティング
文法チェック	構文と意味	構文だけ

(a) 入力テキスト：$A:=B+1$



(b) 構文木



(c) 抽象構文木

図-1 構文木と抽象構文木の例

```
IF (condition)
   THEN statement
   ELSE statement
```

図-2 IF 文のテンプレート

condition, statement は，構文規則における非終端記号 (nonterminal symbol) に相当し，それらの書き換えができる構文要素のクラスを示す．テンプレート中のそのような要素のことを**記入子** (placeholder) という．1つの文や宣言を完成させるには，記入子の部分を別のテンプレートあるいは終端記号で置き換える操作を繰り返す．図-1 の構文木の例からも明らかであるが，すべての入力をテンプレートだけで行おうとすると，簡単な式を入力するのにも，使用者はいくつものテンプレートを指定しなければならなくなる．そこで，操作性を改善するために，C.P.S. を含めて，テンプレート方式のエディタの多くは，条件式や入出力並びといった個所には直接，文字の列として句 (phrase) を書き込むようにしている．C.P.S. では，代入文を入力するときには，そのためのテンプレートを指定しないで，テンプレート statement に対して，直接，そのテキストを書き込むようにしている．句の入力に対しては，その部分についてだけ構文解析が行われ，もし誤りがあれば，句の再入力が要求される．

C.P.S. におけるプログラムの入力過程の例を**図-3** に示す．この図において，(a), (b), …はテンプレートまたは句の入力による画面の推移を示す．

注釈 (comment) は多くの言語において，任意の場所に記述できるようになっているが，そのことをテンプレートの中でいちいち表示していると，画面は見づらくなり，注釈を省略するのにもキー操作が必要となる．そこで，C.P.S. では注釈が記入できる場所を制限し，注釈を書くときには，そのためのテンプレートを別に指定させるようにしている[3]．注釈は次の場所に書くことができる．

(1) PROCEDURE テンプレートの注釈欄

(2) 文，文の列，宣言部，パラメタ宣言の直前

(3) 宣言，およびパラメタ宣言中の変数並びの終り

(2)の場合には，それらの記入子にカーソルを移し，コマンド ".c/" を入力して，注釈テンプレート付きの記入子に置き換える．例えば，

{statement}

に対して，".c/" を入力すると，この部分は次のようになる．

/＊＊comment＊/

{statement}

ここで，記入子 comment を使用者の注釈文で置き換えればよい．このテンプレートは2行で1つのものであり，注釈は {statement} 全体で何を行うのかを説明するのに用いられることを仮定している．

初期の構造エディタとして，米国の Argone National Laboratory で開発された Emily システム (1971年) がある[12]．このエディタは PL/I 用で，BNF で書かれた構文規則をほぼそのままテンプレートとして採用している．グラフィックス端末を用い，画面に表示されるテンプレート中の記入子に対しては，使用者へのガイドとして，その非終端記号に対する定義を画面の下方に表示するようにしている．

3.3 両者の比較

構造エディタは，上述のように，入力方式から見て，MENTOR のような構文解析 (parsing) 型と，C.P.S. のような合成 (synthesis) 型とに分けることができる．合成型のエディタは構文解析型のものと比べ，次のような特徴をもつ．

(1) テンプレートの種類を指定することによって，文の一般形が表示されるので，**if, then, else** といった予約語の入力は必要ない．

(2) 記入子の示す個所に入力できる構文要素のクラスはエディタが管理していて，正しいものだけが受理される．入力や修正の誤りによって，構造が破壊されることはない．

(3) 記入子は，表示される順に具体化していく必要はない．カーソルの位置を任意の記入子に移動することによって，望ましい部分から具体化を行うことができる．C.P.S. は，記入子が残っている不完全なプログラムでも実行が可能である．実行中に記入子に出会うと，エディタに制御が戻り，その部分が画面に表示される．その記入子を他の文あるいはテンプレートで置き換えれば，再び実行を続けることができる．

(4) テンプレートの表示により，文法を完全に理解していない者でも，言語の学習をしながら，プログラムを作ることができる．プログラミングの入門教育の実習にとくに効果的である．

職業的プログラマにとっては，テンプレートの指定のために特別なキー操作をするよりも，構文解析型のように順次，原始テキストを入力していく方が使い易いという意見もある (Teitelbaum 談)．

構文解析型のエディタは，既存のプログラムでも，そのファイルを入力することによって，構文木を作り，編集を行うことができる．しかし，合成型の場合には，エディタの中に専用の構文解析部が用意されていない限り，他のツールを用いて作成されたプログラ

```
(a)  input  ".main/"
.main                                          editing abs
-------------------------------------------------------
object


(b)  input "/"
                                               editing abs
-------------------------------------------------------
/* comment */
abs: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END abs;


(c)  input a comment
                                               editing abs
-------------------------------------------------------
/* print the absolute value of an input integer  */
abs: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END abs;


(d)  input "/"
                                               editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END abs;


(e)  input ".fx/"
                                               editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( list-of-variables ) FIXED;
    {statement}
    END abs;


(f)  input "k/"
                                               editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    {declaration}
    {statement}
    END abs;


(g)  input "/"
                                               editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    {statement}
    END abs;
```

図-3(a)

```
(h)  input ".g/"
                                                editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    GET LIST ( list-of-variables );
    END abs;


(i)  input "k/"
                                                editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    GET LIST ( k );
    {statement}
    END abs;


(j)  input ".i/"
                                                editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    GET LIST ( k );
    IF ( condition )
       THEN statement
       ELSE statement
    END abs;


(k)  input "k<0/"
                                                editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    GET LIST ( k );
    IF ( k<0 )
       THEN statement
       ELSE statement
    END abs;

(l)  input ".pl//.pl/"
                                                editing abs
-------------------------------------------------------
/* print the absolute value of an input integer */
abs: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    GET LIST ( k );
    IF ( k<0 )
       THEN PUT LIST ( list-of-expressions );
       ELSE PUT LIST ( list-of-expressions );
    END abs;
```

図-3(b) C.P.S. におけるプログラムの入力過程" (□はカーソル, "/" は return キーを示す).

ムを編集することはできない.

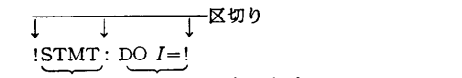
4. テキストの表示方法

テキストを端末に表示するときには, エディタが保持している構文木の中の必要な部分から, 原始テキストを組み立てなおすことは前に述べたとおりである. このとき, プログラムの制御構造は既知であるから, テキストは制御構造に着目して字下げ (indentation) して表示されるのが一般的である. また, 限られた画面を有効に利用するため, そして低速のデータ転送を

カバーするために，テキストの一部を省略して表示する技法がいくつか試みられている．

Emily システムでは，**図-4** に示すように，PL/I の構文要素を示す記号 (holophrast) と，その原始テキストの最初の数文字だけを表示するようにし，簡潔な表示を試みた[12]．

MENTOR では，この技法を発展させ，構文木の根から，使用者が指定した下位の n レベルまでの部分で構成されるテキストだけを表示するようにしている




```
DO I=1 TO 20;
   S=!ARITH: S+A(I!;
END;
```

図-4 Emily システムにおけるテキストの表示例

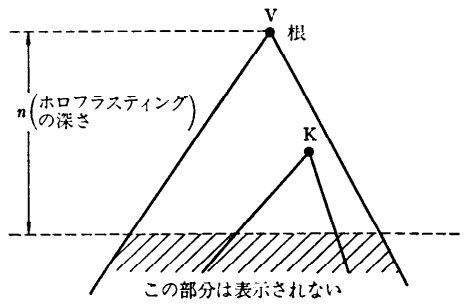


図-5 ホロフラスティング

```
                                    holophrasting depth
if # then #                                  n=1

if A = B+2*C then
       begin
       ...                                   n=2
       end

if A = B+2*C then
       begin
       #;#;#                                 n=3
       end

if A = B+2*C then                  {source text}
       begin
       I:=I+1;                               n=4
       J:=0;
       WRITE(A,B,C)
       end
```

図-6 ホロフラスティングの例 (…および#は，それぞれ並び，および文または式を表わす記号である)．

(**図-5**)[7]．この技法を**ホロフラスティング**という．**図-6** にその例を示す．式はレベル n にあっても，ホロフラスティングは適用されず，そのままの形で表示される．

MENTOR では，必要なたびに，表示用の原始テキストを生成しているが，これは CPU への負荷が大きくなる．また，画面方式のエディタでは，表示の制御と管理が複雑になる．その処理を簡単にするために，C.P.S. では注釈に着目して，具体的な宣言や文は省略 (elipsis) し，それらに付けられた注釈だけを表示する方法を採用している．□はカーソルを表わすものとして，次のような画面を想定する[3]．

```
/** exchange x and y */
temp=x;
x=y;□
y=temp;
```

このとき，省略のために "…" キーを押すと，画面は次のようになる．

```
/** exchange x and y */
□...
```

ここで，再び "…" キーを押せば，もとの画面が再現される．省略の指定は，各種のデバッグ機能を使って実行を追跡するときも有効に働く．

筆者らは，画面方式の構造エディタを試作しているが[33]，ここでは，ホロフラスティングを採用し，省略部分にカーソルを移すと画面を改め，その詳細を表示する方法（自動ズーミング）をとり入れている．そのとき，表示されているテキストの部分からカーソルが離れると，前の画面が再現される．

5. 編集機能

テキストエディタを用いる場合には，行や行の中の文字列に対して，挿入，削除，あるいは置換といった操作を繰り返すことによって，プログラムの編集を行う．これに対して，構造エディタの場合には，編集対象の構造を考慮した操作が必要である．ここでは，その特徴的な機能だけを述べる．

5.1 編集位置の指定

画面方式のテキストエディタでは，カーソルを上，下，左，右に移動させるキーを用いて，また行方式の場合には行番号を指定することによって，編集位置を指定する．構造エディタの場合には，表示されているテキストに対する構文木の上

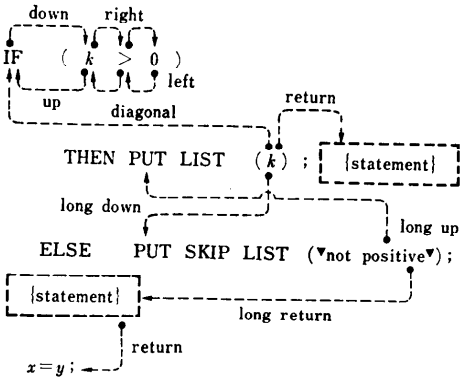


図-7 C.P.S. におけるカーソルの移動（点線で囲んだ部分は，カーソルの移動によって，その中のテンプレートが画面に表示されることを示す．テンプレートの置換をしないで，カーソルを他へ移せば，消える）．

で，カーソルがどの節点に対応しているかを考えて位置を指定しなければならない．

C.P.S. には，カーソルの移動のために専用のキー up, down, left, right, long, return が用意されている（図-7）．これらの働きを以下に示す．カッコ内のキーは，逆方向のカーソルの移動を表わす．

（1） down (up)：次のテンプレート，句，あるいは記入子にカーソルを移動させる．

（2） right (left)：カーソルが句を指しているときは，次の文字へカーソルを移す．それ以外の場合は down (up) と同じ．

（3） return：カーソルが {statement} のような記入子に対する並びの要素の1つを示しているものとする．そのとき，このキーを押すと，その要素の直後に同じクラスの要素 (statement) が挿入できることを示すための記入子テンプレートが表示され，カーソルはそこへ移る．

（4） diagonal：現在の要素を直接含む，テンプレートへカーソルを戻す．

（5） long right (left)：句の最後へカーソルを移す．

（6） long down (up)：構造上，現在の要素と独立していて，同じレベルにある次の要素へカーソルを移す．

（7） long return：long の意味は上と同じであり，次に挿入可能な並びの要素の位置にカーソルを移す．

（8） long diagonal：プログラムの先頭へカーソルを移す．

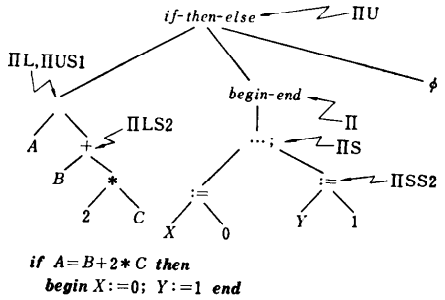


図-8 MENTOR における編集位置の指定

MENTOR は画面方式を採用していないために，構文木上における現在位置あるいは名前をつけた節点からの相対位置を式で指定する方法を用いている．構造上の位置 (structual address) を保持する変数をマーカ (maker) という．マーカはエディタに用意されているもの以外に，使用者自身でも定義することができる．マーカを移動させるには，次の記号を組み合わせたものを用いる（図-8）．

U_n：n 個上位の節点
R_n：n 個右側の節点
L_n：n 個左側の節点
S_n：n 番目の子供

MENTOR では，式についても構文木を作成しているので，式中の一部を修正する場合でも，これらの記号を用いてその位置を指定しなければならない．このエディタの使用者は，Pascal の構文および対応する抽象構文木について，正しい理解をもっていることが要求される．

5.2 主な編集操作

（1） 削　除

カーソルあるいはマーカによって編集位置を指定し，削除コマンドを入力する．C.P.S. の場合には，削除された部分が，“delete” と呼ぶファイルに移されるので，あとからそれを挿入に用いることができる．削除された部分には，場所によって（たとえば IF 文の条件式や，並び全体），もとの記入子が入れられる．プログラムの一部分を切り取って，別の場所に保存する操作を**クリッピング** (clipping) というが，C.P.S. には，“delete” キー以外に，クリッピング用のキー “clip” が用意されている．両者は機能的に同じである．“clip” キーを押したときには，“clip” という名前のファイルに移される．また，いくつかの文

や宣言の列を一度にクリップするために，long clip がある．

MENTOR は，削除された部分木を保持しておくためのスタックを備えている．そのための専用のマーカを操作することによって，以前，削除した部分木をアクセスすることができる．

（2） 挿 入

C. P. S. では前述の (long) return キーを用いることによって，必要な位置にテンプレートを表示させればよい．それを他のテンプレートや句，あるいは "delete" や "clip" ファイルの内容で置き換えることによって，挿入ができる．

MENTOR の場合には，挿入したい位置にマーカを移動させ，挿入コマンドを用いて，テキストをそのまま入力する．このとき，他の部分木を複写して入れることもできる．

（3） 交 換

文や式どうしの交換は，構文木における対応する部分木の交換によって簡単に実現することができる．MENTOR には，それら2つの構造上の位置を指定して，それらを交換するためのコマンドが用意されている．

（4） 構造の変更

複合文：DO S_1 ; S_2 ; … ; S_n END

を次のように変更する場合を考える．

IF (x>0) THEN
  DO S_1 ; S_2 ; … ; S_n END

通常のテキストエディタを用いるときには，複合文の前に，行 "IF (x>0) THEN" を挿入するだけでよい．構造エディタの場合には，制御構造，それ自身を変更することになるので，複合文をまず取り去り，そこに新たに IF 文を挿入するようにしなければならない．このとき，複合文の再入力を避けるために，クリッピングの機能が有効となる．

（5） そ の 他

その他の機能としては，名前のつけかえ (renaming)，パターンによる構文木の探索，構文木を保持するファイルの入出力，原始テキストファイルの生成などがある．

6. 実 現

（1） 作 成 方 法

構造エディタを開発する場合に，言語ごとに構文解析部と原始テキストの生成部を最初から設計し，作成するのは，時間と費用がかかり，保守においても同じ問題が起る．そこで，構文と内部表現の形式，原始テキストの表示の形式を仕様化し，それらから自動的に，あるいは半自動的にエディタを作り出す試みがなされている．たとえば，MENTOR では，詳細な構文と抽象構文とを記述し，それらから構文解析部を生成するための言語 METAL[10]，抽象構文木に対する操作（コマンド）を定義するための言語 MENTOL を使って，エディタを作成している[11]．Teitelman らは，C. P. S. での経験をもとに，属性文法 (attribute grammar) を利用した合成系成生プログラム (Synthesizer Generator) の開発を試みている[5),6)]．また，Gandalf プロジェクトでも，字句の定義，構文の記述とそれに対する処理ルーチン (action routine) の記述，原始テキストの表示形式の指定などから，合成型のエディタを生成するための ALOE 成生系を開発している．

（2） 意味チェック

構文的な正しさと同時に，意味 (semantics) 的な正しさも常に保証しようとすると，たとえば，次のような場合の取り扱い方で問題が出てくる：未定義の名前の参照，型の不一致，宣言済みの構文要素の削除と属性の変更．MENTOR では，この問題を避けるために意味チェックは行わず，処理系にまかせる方針をとっている．

一方，C. P. S. では，宣言を絶対的なものとして，常に意味チェックを行い，正しい句だけを受理するようにしている．宣言に対して編集が行われたときには，その参照個所すべてについて意味チェックを行う．その結果，未定義名の参照や型の矛盾を検出した場合には，それらの個所を光らせて，使用者に修正を促すようにしている[2)]．

（3） プログラム作成支援環境

多くの言語処理系は，原始プログラムの中間形として，構文木あるいはそれに近い形の表現を使用している．したがって，言語処理系に編集機能を付加したり，エディタにコード生成や通訳実行の機能を加えたりすることによって，1つのツールにまとめ上げることができる．C. P. S. は1つの内部表現を中心に，編集，実行，デバッグといった機能を関連づけることによって，プログラム作成支援環境を実現している．

7. おわりに

以上，構造エディタについて，機能的な側面から特徴を述べてきた．使用者から見て，構造エディタは次

のような問題が指摘できよう.

1) 多重窓を用いた利用者インタフェースをもつ構造エディタを,通常の CRT 端末の上で実現する場合には,原始テキストの表示のために 20 行程度の場所しか確保できない.これは,テキストエディタにも共通した問題であるが,利用者にとっては視野がひじょうに制限されることになる.そのために,ホロフラスティングの深さの変更,画面の移動 (scrolling) を頻繁に行わなければならない.

2) 編集をするときには,表示されているテキストの"構造"を考慮しなければならない.とくに,編集位置を指定する場合である.大きなプログラムを編集するときには,位置を指定するために多くのキー操作が必要となる.この点,テキストエディタの場合には,単に行番号を指定するだけで,任意の場所に位置づけができる.

3) ホロフラスティングの深さの変更,画面の移動によって,原始テキストの生成が必要となり,また画面とカーソルの制御と管理が複雑になることから,構造エディタは CPU に対する負荷が大きくなる.TSS 環境の下では,使用者数に比例して,極度に応答性が悪くなる.速やかな応答性を保つには,C. P. S. のように個人用の計算機を使って,中央の CPU に対する負荷を吸収することが望まれる.

4) 原始テキストは常に一定の形式で表示されるので,コーディングの標準化には役立つ.しかし,自分のコーディングスタイルというものをもっている使用者にとっては,馴じめない場合が多いであろう.きめの細かい表示をしようとすると,原始テキストの生成に時間がかかり,カーソル位置の管理も面倒になる.

構造エディタは,現状では,従来のエディタに代るものとは言えないが,特定の言語によるプログラミング教育というように限定された場面では有効なツールである."構造"を取り扱うことから,単にプログラムテキストの入力と編集だけでなく,設計情報を構造化し,互いに関連づけて,設計からプログラミングまでを支援するツールとして,さらに発展することが望まれる.

最後になりましたが,本稿をまとめるにあたって,多くの資料を提供して下さいました方々,ならびに内容について熱心に議論して下さった慶応義塾大学理工学部大学院博士過程,野呂昌満君に深く感謝します.

参考文献

1) Meyrowitz, N. and van Dam, A.: Interactive Editing Systems: Part I and II, ACM Comp. Surv., Vol. 14, No. 3, pp. 321-416 (1982). (邦訳, 石井 博, 瀬川 清: コンピュータ サイエンス, bit 12月号別冊, 共立出版, pp. 75-166 (1983)).

2) Teitelbaum, T. and Reps, T.: The Cornell Program Synthesizer: a Syntax-directed Programming Environment, Comm. ACM, Vol. 24, No. 9, pp. 563-578 (1981).

3) Teitelbaum, T.: The Cornell Program Synthesizer: A Tutorial Introduction, p. 47, Dept. of C. S., Cornell Univ. (1981).

4) Teitelbaum, T. et al.: The Why and Wherefore of the Cornell Program Synthesizer, SIGPLAN Notices, ACM, Vol. 16, No. 6, pp. 8-16 (1981).

5) Reps, T.: The Synthesizer Editor Generator: Reference Manual, p. 9, Cornell Univ. (1981).

6) Demers, A. et al.: Incremental Evaluation for Attribute Grammars with Application to Syntax-oriented Editors, 8-th Anual ACM Symp. on POPL, pp. 105-116 (1981).

7) Donzeau-Gouge, V. et al.: A Structure-oriented Program Editor: a First Step towards Computer Assisted Programming, Research Rep. 114, p. 8, INRIA (1975).

8) Donzeau-Gouge, V. et al.: The MENTOR Program Manipulation System, p. 24, INRIA (1979).

9) Donzeau-Gouge, V. et al.: Programming Environments Based on Structured Editors: The MENTOR Experiences, Research Rep. 26, p. 14, INRIA (1980).

10) Donzeau-Gouge, V. et al.: Recent Publications about MENTOR, INRIA (1983).

11) Klint, P.: A Survey of Three Language Independent Programming Environments, Research Rep. 257, p. 16, INRIA (1983).

12) Hansen, W. J.: User Engineering Principles for Interactive Systems, FJCC '71, pp. 523-532 (1971).

13) Medina-Mora, R. and Feiler, P. H.: An Incremental Programming Environment, IEEE Trans. Softw. Eng., Vol. SE-7, No. 5, pp. 472-482 (1981).

14) Habermann, N. et al.: The Second Compendium of Gandalf Documentation, Carnegie-Mellon Univ. (1982).

15) Petrone, L. et al.: DUAL: An Interactive Tool for Developing Documented Programs by Step-wise Refinements, 6th Int. Conf. on

Softw. Eng., pp. 350-357 (1982).
16) Nakamoto, Y. et al.: An Editor for Documentation in π-system to Support Software Development and Maintenance, 6th Int. Conf. on Softw. Eng., pp. 330-339 (1982).
17) Teitelman, W.: Interlisp Reference Manual, XEROX Palo Alto Research Center (1974).
18) Sandwall, E.: Programming in an Interactive Environment: the "LISP" Experience, ACM Comput. Surv., Vol. 10, No. 1, pp. 35-72 (1978).
19) Witty, R.: Dimentional Flowcharting, Softw. Pract. Expr., Vol. 7, pp. 553-584 (1977).
20) Jönsson, A.: DIMED: Dimentional Flowchart Editor Manual, Internal Rep., Linköping Univ., p. 20, Sweden (1980).
21) Morris, J. M.: The Design of a Language-Directed Editor for Block-Structured Languages, SIGPLAN Notices, ACM, Vol. 16, No. 6, pp. 28-33 (1981).
22) 酒井三四郎，落水浩一郎：プログラム階層構造の生成，処理，文書化能力を有するテキスト・エディタ，情報処理，Vol. 23, No. 5, pp. 487-494 (1982).
23) 宮本衛市，浅見可津志：プログラム編集機能をもつテキスト・エディタ，情報処理学会論文誌，Vol. 20, No. 6, pp. 474-480 (1979).
24) 菅井賢三他：対話型修正編集機能をもつ PASCAL 構文解析システム，情報処理学会論文誌，Vol. 22, No. 2, pp. 148-154 (1981).
25) 田中 厚他：言語適応型プログラミング環境用マシン インタフェースとしての構造エディタ PARSE，情報処理学会ソフトウェア工学研究会資料 34, pp. 43-48 (1984).
26) Chusho, T. et al.: A Language Adaptive Programming Environment Based on a Program Analyzer and a Structure Editor, 9th World Computer Congress, IFIP '83, pp. 621-626 (1983).
27) 海尻賢二：構文主導型プログラム開発システムについて，情報処理学会ソフトウェア工学研究会資料 27-4, p. 10 (1982).
28) 情報処理振興事業協会：TDSS (Top-down Design Support System) 利用者マニュアル，p. 224 (1982).
29) 花田収悦：プログラム設計図法，企画センター，p. 119 (1983).
30) 内田輝夫他：HCP チャートプロセッサ，情報処理学会第 28 回全国大会論文集，pp. 529-530 (1984).
31) 湯浅太一他：モジュラー・プログラミングのための支援環境，情報処理，Vol. 23, No. 5, pp. 433-441 (1982).
32) 湯浅太一，中島玲二：IOTA プログラム支援環境，情報処理学会プログラム設計技法の実用化と発展シンポジウム論文集，pp. 151-165 (1984).
33) 野呂昌満他：構文向きプログラムエディタにおける原始プログラムの表示方法について，情報処理学会第 26 回全国大会論文集，pp. 567-568 (1983).
34) 中所武司：プログラミング言語とその会話型支援環境，情報処理，Vol. 24, No. 6, pp. 715-721 (1983).
35) Henderson, P. (Ed): Software Engineering Symposium on Practical Software Development Environments, ACM SIGSOFT/SIGPLAN, Vol. 9, No. 5, p. 197 (1984).

(昭和 59 年 5 月 14 日受付)