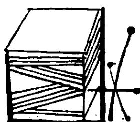
論　説



思考の道具としてのエディタ†

平　賀　　譲††

1. はじめに

文書やプログラムにはさまざまな構造が輻輳して存在しており，われわれはそれらを手がかりに作成・編集作業を頭の中で組みたてている．これを「自然な」形で実行するには，道具であるエディタはその構造やそれに対する操作を反映した機能をもつ必要がある．

一方システムの内部では，これらの情報は最終的には「ファイル」という形で単なる文字の並び（あるいはそれを適当に区切ったもの）として表現されるので，システムとしてのエディタはこの文字の並びを操作する機能をもっていれば十分である．初期のエディタはこの観点に基づいて作られていたため，人間の側からの見方との間にギャップを生じ，それを埋めることはユーザの負担となり，エディタの使いづらさの原因の一つとなっていた．

本稿ではまず文書やプログラムがもつ種々の構造とそれがどのようにエディタ機能に反映されているかを概観する．最近の先進的なエディタは人間の思考との整合性という点からみると他のシステム，特に既存のプログラミング言語をすでに追越していると考えられるが，その知見は必ずしも他に生かされているとはいえない．これに関連する問題として2点，syntax-directed な構造エディタ（以下構文エディタとよぶ）のもつ問題点と，人間の思考とのインタフェースからみた既存の言語のもつ問題点をとりあげる．

2. テキストの構造

文書やプログラム（以後「テキスト」と総称する）は，最終的には文字の並びとして書下すことができるから，この文字列は，のっぺらぼうな基底構造と考えることはできる．しかしわれわれはテキストを単なる文字列として見ることはなく，いくつかの文字をまとめた単位，あるいはさらにそれらを組み合わせた高次の構造により把握している．そのような構造の例として，テキストが物理的に表示される形式に基づく表示構造，単語・文・段落，…といった（連接的で syntactic な）階層構造，意味的なつながりに基づく意味構造などが考えられる．

2.1 表示構造

文字列という基底構造を機械的に分割し，適当に結合したものが表示構造である．表示構造はエディタがユーザに示すテキスト像とみることができる．図-1に4種の表示構造を示した．(a)は各文字を分割の単位としたもので，TECO がこれにあたる．(b)は行を単位としたもので，行同士は（前後関係以外に）直接は関係ない．MULTICS の CTSS エディタをはじめ，初期のエディタは皆この行エディタである．(c)は単一バッファのスクロールエディタ，つまり，vi などの画面エディタであり，(b)とは行内，行間の往来が自由な点が異なっている．ただし，実際に画面に表示されるのは「ウインドウ」で区切られた部分だけである．(d)はマルチバッファエディタで，普通はマルチウインドウでもある．これは普通の本（codex）のイメージに近く，本同様，複数の「ページ」を同時に参

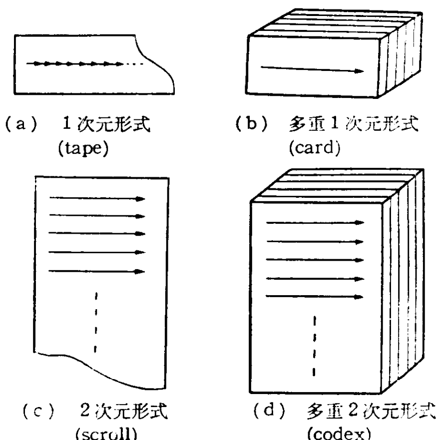


図-1　テキストの表示構造

† Editor as a Tool of Thought by Yuzuru HIRAGA (University of Library and Information Science).

†† 図書館情報大学

照できる．このようなエディタの代表的な例は Star, Smalltalk-80 などの bit-mapped display を用いたシステムにあり，これらにはハードウェア，ソフトウェア共，現在の最高水準の技術が導入されている．

編集操作について簡単にふれておくと，(a)，(b) では基本コマンドが対象とするのは字，行といった分割の単位であり，一方(c)，(d)ではウインドウに表示されたテキストの全体またはその一部である．これには字，行だけでなく，次節でふれる単語，文なども含まれており，テキストに対する視野の拡大と合わせて，エディタでの作業を紙の上での校正のイメージに近いものにしている．

2.2 階層構造

表示構造がシステム主導的な概念であるのに対し，単語・文・段落といった要素から成る階層構造は人間主導的な概念である．プログラムでは名前，演算子などのトークン，その列としての文などがこれに対応する．階層構造は人間の思考の中では最も浅い部分にあって意識の対象であり，文字列表現であるテキストへは比較的容易に変換できる．これに対し，思考の深い部分にあると考えられる文の構文構造などは複雑であり意識の上には上らない一方，表示構造に属する事項，たとえばある単語が行末にあるか否かなどはエディタコマンドを組みたてる作業以外では思考には登場しないと考えられる．

階層構造は各要素が連接的で，テキストから読みとることは比較的簡単であり，これを機械的な文字列検索によって認識できることも多い．たとえば単語は「区切り記号に囲まれた英字列」，文は「前のピリオドの次から次のピリオドまで」といった具合である．これは常にうまくゆくわけではないが，実用的には十分であり，vi, emacs 等，多くのエディタで利用されている．特に emacs の Lisp パッケージなどは Lisp 構造エディタと比べても遜色ない[5]．

上は擬似的に構造を扱う方法だが，システムの内部で直接構造を扱うこともできる（＝構造エディタ)．いわゆる document editor では章・節といった階層構造はシステム内で木構造として表現される一方，テキストそのものは文字列として扱うという hybrid な構成をとっている．これに対し，プログラムを対象とした構造エディタは基本的にはプログラム全体を構文木という構造として扱っている．構文構造は単純な階層構造と違ってネストが許され，テキストも連接的ではなくなるので擬似的な構造として扱うのは難しい．

構造エディタはテキストエディタに比べるとシステムとして複雑になり*，また有用性についてはいろいろな議論がある[4),5)]．

2.3 意味構造

テキスト中に散らばった要素から構成された，あるいは直接字面に現われないような抽象的な構造を一括して意味構造と呼ぶことにしよう．文書の脚注や図表参照，プログラムのサブルーチンの参照関係などもこの部類に含めることはできるが，ここでは意味にかかわる抽象的なものを中心に考える．文書では文同士や章同士の意味的なつながり，プログラムでは初期化，例外処理などの機能上の単位や，もっと抽象的にアルゴリズムなどがこれにあたる．

意味構造は人間の思考の深層に存在するものであり，それ自体を明示的に表わすことは難しい．われわれは言語などの道具を使ってその投影を表現できるに過ぎない．ここで投影といったのはこの変換の過程で情報が保存されなかったり，逆に余分な要素が付加されたりすると考えられるからである．簡単な例でいうと，「配列の総和をもとめる」という意味をプログラムで表わすには

$a:=0$; for $i:=1$ to n do $a:=a+b[i]$;

などと書かねばならない．ここで a, i, n といった変数，a のゼロ化や for 文による繰返しなどは言語の都合で付加された余計な要素である．

意味構造は複雑なネットワーク的形態をとると考えられるが，これを表現する言語は（プログラミング言語も含めて）基本的には直列的である．したがって意味的には近い関係にある概念でもテキスト上では散らばってしまうことが多い．特にプログラムでは実行順序という制約も加わるので，意味的には別個の処理がプログラム上では入り混ることになり，これを読みとるにはプログラムの「断面」を抽出するなどしなければならない[6]．

このような意味構造をエディタで直接扱うのは極めて難しい．せいぜいわれわれにできるのは，頭の中で手がかりとなる階層構造などの要素を組みたてて，それを利用するぐらいである．

3. 構文エディタについて

構文エディタというのはプログラミング言語の構文規則によって top-down にプログラムを作成する

* Lisp エディタの場合は事情が少し異なるが，ここではとりあげない．

構造エディタのことで，最近では Cornell Program Synthesizer や Gandalf/Aloe などがある*．構造エディタの最大の利点として，構文的に正しいプログラムが自動的に作れることがあげられる．とはいえ，構文自体はコンパイラでチェックできるし，簡単な parser を備えたテキストエディタも多く，これが際だった特徴といえるか，疑問が残る．また意味的なエラーには無力である．

人間の思考とのインタフェースという点から考えると，構文エディタにはいくつか問題点がある．一般に構造エディタの視野は意外に狭いもので，たとえば簡単な Lisp エディタでは

a)))))(((*b*…

という形があったとき，*a* から *b* へ注視点を移すのはわれわれの直観に反して面倒な操作が必要となる．この原因は構造エディタが自身で定義した構造のみをユーザに示し，表示構造などわれわれが利用する他の構造を切り捨てている点にある．もちろんテキストエディタでも事情は同じだが，文字列の場合は 2.2 でみたようにかなり融通がきく．単一の構造の強制は編集操作もわずらわしいものにすることがある[4]．

より重要な問題が「top-down 展開」という処理手順の強制，及びその過程中での非終端記号の明示から生じる．すでにみたように，われわれの思考の中にある「プログラム」の構造は，構文規則で律せられる性質のものではない．例として「数値データを読み，平均値を計算する」思考プログラムを考えよう．これは「データの個数を数える」「データの総和をもとめる」「割り算する」という処理に意味的には分割できるが，実際のプログラムでは入力の逐次的な性質のため，前二者の処理が構文的に混り合った形で現われる．

一般に段階的詳細化を明示的に示せば木構造ではなくグラフとなり，それは構文木とはかなり異なった形になる可能性が大きい．プログラムが整った木構造をもつのはあくまで言語の都合から決められたことで，頭の中の意味構造や作成過程とは必ずしも関係はない．出来上ったものが美しいからといって，それを作る過程も同様に美しいとは限らない．

さらに，自然言語の場合同様，プログラムでもわれわれは文を意識下で断片的に生成しており，構文規則は局所的に（場合によっては不完全に）意識下で適用されていると考えられる．われわれがエディタに向かったときにはトークン列という階層構造がすでに出来ているわけで，この段階でエディタが構文規則の処理を要求してくるのは，同じ仕事を，しかも意識下で済んでいたものを意識的に繰返させることになる．このへんが多くの人が構文エディタに抵抗感をもつ原因ではないだろうか．

細かい問題点はまだいくつか残っているが，もっと本質的な問題が一つある．それは，このようなエディタの必要性が感じられるほどの大規模な構文構造が作られてしまう必要性があるのかという点で，これについてはプログラムの分割という側面から次にふれる．

4. エディタから見たプログラミング言語

自然言語とプログラミング言語の重要な違いは，後者は作り直すことが可能な点にある．ここまでエディタについてみてきた事項を材料に，既存の言語を少し見直してみよう．本来エディタは従属的なシステムであり，処理対象に適合するように作られてきた．しかしこれではその欠点がそのまま持込まれることになるし，一方，ことユーザインタフェースに関しては，先進的エディタはプログラミング言語よりはるかに進んでおり，言語側へのフィードバックをする意義はありそうである．

紙数の都合で，話題を2次元情報の活用とページ単位のモジュール分割の2点に絞り，他の話題（たとえば図形表示の利用）については割愛する．

4.1 2次元情報の活用

言語が本質的に1次元・直列的なのに対し，画面エディタは2次元的視野を提供してくれる．これを活用しなければ，情報をムダにすることになる．昔からある話として，ネスト構造を begin…end 等の区切り記号を使わずにインデントによって表わす方法がある．OCCAM[3] ではこれは仕様の一部である．行末のセミコロン省略（bcpl）は，このもう少し簡単な例である．テーブル操作言語などは言語そのものが2次元的といえる．

並行プログラムも2次元的表現のよい対象である．並行動作の記述機能には，これを用いて直列的プログラムを凝似的並列プログラムに変換し[3]，プログラムの意味構造をすっきりと表現するための道具に使えるという別の利点もある．

4.2 ページ単位のモジュール分割

ページ大のウインドウ表示は理解の対象としても手頃な大きさであり，プログラムをこの大きさに分割することは discipline としては提唱されている[2]．リス

* 詳細は本号の紹介記事と文献を参照してほしい．

トではこれが直列に並べられるので読むのは大変だが，システム上では各単位を有機的に管理しておけるので必要な単位を楽に表示できる．この意味で，われわれは「文書」としてのプログラムを放棄し，すべての仕事をシステム上で行ってしまうように発想を切換える時期ではないだろうか．

モジュール分割はLispなどでは抵抗なく行えるが，多くの言語では手続き化がその唯一の道具であり，これには心理的抵抗感が大きく，すぐ長い手続きができてしまう．この原因の一つは，言語にいろいろな制御構造が用意されており，それをネスト化できる点にあるだろう．しかしすでにみたように，制御構造（特に繰返し構造）で表現される処理は頭の中にある処理と食い違うことが多いし，ネストを重ねることは（自然言語の場合は当然の話だが）理解を妨げる大きな要因になる．制御構造はいろいろな動作の「概念」を表現する手段だが，これをいちいち基本的動作に分解して示すよりは，これに名前をつけ，その意味を別個所で説明する方が自然であり，分割も容易になる．一度定義した概念は蓄積してゆくことが可能で，これを言語の構成要素に含めれば，言語は open-ended な構成をもつことになる．

このような言語を使うには概念の結びつけ方を記した「文法書」だけでなく，各概念の意味を記した「辞書」も必要となる．これは自然言語ではあたりまえであり，適切な言語の枠組と環境さえあればプログラミング言語でも有効だろう*.

実在する言語（というよりシステム）でこれにあてはまる例としては UNIX/C（これは辞書がヒドイ！）Lisp などがあるが，(視点は少し変るが) object を前面に出した Smalltalk-80[1] がその代表格だろう．Smalltalk についてふれる余裕はないけれども，ここにその名が出てきたこととそのエディタが最も先進的機能をもつこととは，決して偶然の符合ではないだろう．

もちろんプログラミング言語としては使い易さ，習い易さ，効率等々，他に考慮しなければならない点が多い．しかしあくまで人間が使うものである以上，エディタというインタフェースを通じて得られる知見は決して無意味ではないだろう．最後に次の言葉をもって結びとしよう…

「新しい革袋には新しいぶどう酒を入れるべきである.」　　(…ん？)

* これは Winograd[7] などと共通な考え方である．

参考文献

1) Goldberg, A. and Robson, D.: Smalltalk-80, the Language and its Implementation, Addison-Wesley (1983).
2) Kernighan, B. W. and Plauger, P. J.: The Elements of Programming Style, Bell Telephone Laboratories (1978).
3) May, D.・OCCAM, SIGPLAN Notices, Vol. 18, No. 4, pp. 69-79 (1983).
4) Meyrowitz, N. and van Dam, A.: Interactive Editing Systems: Part 1, 2, Comput. Surv., Vol. 14, No. 3, pp. 321-415 (1982).
5) Sandewall, E.: Programming in an Interactive Environment: The LISP Experience, Comput. Surv., Vol. 10, No. 1, pp. 35-71 (1978).
6) Weiser, M.: Programmers Use Slices When Debugging, Comm. ACM, Vol. 25, No. 7, pp. 446-452 (1982).
7) Winograd, T.: Beyond Programming Languages, Comm. ACM, Vol. 22, No. 7, pp. 391-401 (1979).
8) Wood, S. R.: Z-The 95% Program Editor, Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation, pp. 1-7 (1981).

(昭和59年4月9日受付)