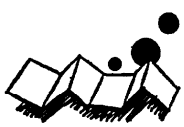
解　説



# プログラミング環境としてのUNIX†

岸田孝一††　栗原正利††

UNIXには，ASCIIコードの文字列を扱うための多数のコマンドがあり，プログラミングに限らず種々のアプリケーションで使用できる．

これらのツールの利用分野は多岐にわたり，プログラミングにおいても文字列のサーチや簡単な置換に応用することが可能である．しかし利用方法があまりに多種多様であるため，ここではそれらの利用方法については割愛し，プログラミング・サポートを目的としたツールについてのみ述べる．

周知のごとく，UNIXはOSを含めほとんどがC言語によって記述されている．このためC言語を対象としたプログラミング・ツールは，LISP, Pascal, FORTRANなどの言語処理系に比べ豊富に用意されている．そこでここではC言語用のツールを中心に取り上げ，C言語のプログラミング環境について述べる．

## 1. C言語処理系

### 1.1 C言語の特徴

C言語はBCPLから発展し，UNIXシステムを記述するために設計された，アセンブラの柔軟性と高級言語の読みやすさを備えた言語である．現在UNIXはカーネルを含めほとんどのコマンドがC言語によって記述されており，UNIXシステムの高い移植性に寄与している．

C言語は一般の手続き型高級言語と同じように，いわゆるストラクチャードなプログラムを記述するための制御構造を有している．しかしながらシステム・プログラムを目的とした言語であるため，一般のアプリケーションで使用される入出力や文字列の操作のための構文は特に用意されておらず，これらの処理は標準ライブラリとして提供される関数の呼び出しによって記述する．

一般にC言語で記述されたプログラムは，ほかの高級言語で記述されたプログラムと比較してコンパクトなプログラムになる．これはC言語における，式の豊富な表現機能による．たとえば，プログラム中によく現れるカウンタに1加える操作は，FORTRANの

```
I=I+1
```

形式の代入字に対し，C言語では

```
i++
```

形式の式として記述できる．式は他の式あるいは文の中に記述することが可能であるため，加算した値をさらに式の中で使用するなどの記述が可能である．変数の参照前あるいは使用後のカウントアップ・ダウンのためのC言語特有の記法は，C言語によって記述されたUNIXシステムが最初に動作したPDP 11の機械語を意識したものであると思われる．

またC言語の処理系にはプリプロセッサが標準装備されており，マクロの展開，プログラム中への別ファイルの挿入，条件コンパイル制御などの機能を提供している．この機能により，複数のバージョンを持った同一機能のプログラムを一つのファイルで管理することが可能である．

### 1.2 C言語処理系の構成

C言語処理系は，モジュール化された処理の結合によって実現されており，高い移植性を実現している．

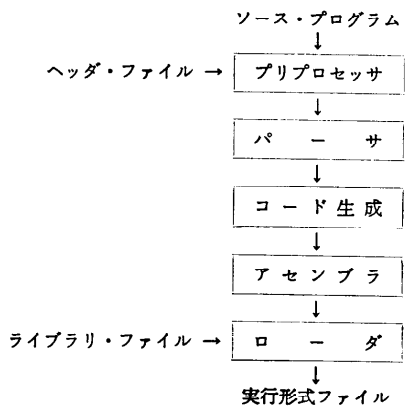



図-1　C言語処理系の構成

† UNIX OS was Developed by Bell Laboratories and Licenced by AT&T.

†† (株)ソフトウェア・リサーチ・アソシエイツ

処理系は，プリプロセッサ，パーサ，コード・ジェネレータ，アセンブラ，ローダの独立したプログラムから構成される．言語処理の流れを図-1 に示す．

このようなモジュール化された処理系は，一般の高級言語の一体化された処理系に比べ処理速度の点で不利であるため，バークレー版においてはパーサとコード・ジェネレータを一体化させている．しかしながらこのモジュール化による構成は，柔軟性，移植性において有利であるため，UNIX システム自身の流布，あるいは UNIX 上での新しい言語への対応など多くのメリットを作り出している．

## 2. プログラム編集

### 2.1 vi—端末独立型スクリーン・エディタ

（1） vi の特徴

vi は，カリフォルニア大学バークレー分校において作成された端末に依存しないスクリーン・エディタであり，現在ではバークレー版 UNIX に限らずほとんどすべての UNIX で利用可能である．バークレーで開発されたエディタには，vi のほかに ex という名前の行エディタがある．実際には vi と ex は同じプログラムであり，名前の違いは行モードで動作するかスクリーン・モードで動作するかの違いである．もちろん実行中にそれぞれのモードに移行することも可能である．

vi は画面の最後の行をステータス表示や，一部のコマンドの入力に使用し，それ以外の部分をテキストの表示に使用する．一般の端末では 23 行がテキスト表示に使用される．vi エディタの画面表示例を図-2 に示す．

vi は種々の CRT に対応するため，端末の種々の属性，たとえば画面の大きさ，カーソルを移動させるための制御コードなどを，プログラム本体とは別のファイルで記述し，vi はこれを参照しながら動作するように作成されている．このような一般化は，ある端末専用に作成されたプログラムに比べて処理速度の点で不利であるが，それまで高価なインテリジェント端末でのみ可能だったスクリーン・エディットを，安価な不特定のターミナルで実現した点で注目に値する．なお現在に至ってはマイクロ・プロセッサの著しい発展により，処理効率の問題は解決されていると考えられる．

この端末に独立した CRT 画面の制御手法は，vi から独立した termlib あるいは curses というライブラリとして提供され，現在広く使用されている．

```
extern  char stdbuf[BUFSIZ];
extern  int  bflg, eflg, nflg, sflg, tflg, vflg;
extern  int  spaced, col, lno, inline;
copyopt(f)
        register FILE *f;
{
        register int c;
top:
        ch = getc(f);
        if (c == EOF)
                return;
        if (c == '\n') {
                if (inline == 0) {
                        if (sflg && spaced)
                                goto top;
                        spaced = 1;
                }
"copy. c" line 11 of 52 --19%--
```

図-2 vi エディタの画面表示例

（2） vi のコマンド群

vi は，前項のように画面の内容についても制御を行うプログラムであるため，これまでのインテリジェント端末を利用した画面エディットと比較して，非常に高機能な編集能力を持つ．vi のコマンド群は大別して以下のように分類できる．

- 画面制御
- カーソル制御
- 変更，削除
- 挿入
- 取り消し，再実行
- 検索
- 行モード・コマンド
- マーク，レジスタ

画面制御は，画面に表示する部分を上下にスクロールする機能で，カーソル制御は，画面内あるいは画面制御と連動してカーソルの位置を変更する機能である．

よく vi のコマンドはあまりに多いため覚えにくいという話を聞くが，実際には上記のように分類して考えると簡単に修得できる．変更，削除は，それぞれのコマンドに続けてその範囲をカーソル制御のコマンドによって指定する形式になっている．コマンドがあまりに多いという感想は，これらすべての組合せを覚えようとしていることに起因するものと思われる．

挿入は，一般のスクリーン・エディタと同様に，文字列あるいは行に対して行うことが可能である．また文字列の挿入が複数行にわたってもよい．

vi は直前に行った修正の取り消し，およびその修正

処理と同じ修正処理，つまり再実行を行うことができる．たとえば文字列や行の削除を行う場合に，その数を目で数えなくてもだいたいの値で実行し，多かった場合は取り消し，少なかった場合は再度残りの部分を削除するなどの方法で行うことができる．このように取り消し機能によって，こわごわエディタを使うという世界から解放される．また再実行の機能は，あるパターンを見つけて，確認しながら変更を加える場合に有効である．

検索は，エディット対象のファイル全体に対して，ある文字列を検索する機能である．UNIX には，文字列検索のために正規表現という指定形式を採用している．この正規表現は vi の中での文字列検索だけではなく，UNIX の文字列検索を行うほとんどのコマンドで利用できる．この正規表現では文字列のパターン，たとえばスペースがいくつかあってその後にある文字があるなどのような，非常に柔軟性が高い指定を行うことが可能であり，vi を有効に活用できるか否かの一つのキーとなる機能である．

行モード，つまり ex で使用できるコマンドを vi の中から実行することが可能である．この機能はエディット対象のファイル全体に対して，一括した変更を加える場合に使用される．

マークはエディット対象のテキストに目印を付ける機能であり，別なコマンドでの範囲指定や，画面表示部分を切り替える場合に利用される．レジスタ機能は，文字列を一時的に vi のワーク・エリアに保存する機能であり，文字列の離れた部分への移動や，コピーに利用される．マークおよびレジスタには，それぞれ a から z まで 26 個の名前を付けることが可能であり，正規表現によるサーチとともに vi を有効に活用できるか否かと一つのキーとなっている．

（3） プログラミングのための機能

これまで述べた vi の機能は，プログラミング専用の機能ではない．vi にはこれらの機能のほかに，プログラミングのために用意されたショートハンド，および関数検索の機能がある．

ショートハンドは，入力された文字列を指定された文字列に展開する機能であり，展開処理はスペースや特殊記号によって，入力された文字列がワードとして認識された時点で行われる．残念ながらC言語は，長いキーワードを持っていないので，変数名などに対してしか有効でないが，COBOL のような長いキーワードを持った言語については有効に利用できると考えられる．

関数検索は，指定した関数名のエントリの行へのジャンプ機能である．このジャンプは異なったファイルへのものであってもよく，その場合は vi が自動的にエディット対象のファイルを変更する．関数名の指定は，キーボードから行う方法と，画面上に検索したい関数名がある場合はその部分にカーソルを移動し実行する方法がある．この機能は特にプログラムを読む場合に有効であり，プログラムをメインの部分から順次下位の関数にたどっていき，目的の部分を検索する場合に実に有効な機能である．

関数検索で使用される関数名の情報は，vi が内部的に作成するものではなく，ファイルで指定できるようになっている．Cプログラムについては，プログラムを解析してこのファイルを作成するツールが vi とは別に提供されている．またC言語の標準ライブラリについてもこのファイルが用意されているため，検索する関数は自分が定義したものに限らず，標準ライブラリで提供された関数でもよい．

vi にはこのほかにも，自動インデンテーション，行番号の表示，括弧の対応の自動表示，エディタの一時実行停止，そのときの自動書き戻しなどの機能があり，オプション指定によって使用することが可能である．

### 2.2 emacs—エディタ機能を有するユーザ・インタフェース

（1） emacs の特徴

emacs の歴史は古く，UNIX の誕生以前にさかのぼる．UNIX 版 emacs にもいくつかのバージョンがあるが，特に有名なものはカーネギーメロン大学の James Gosling によって作成されたバージョン，そして Richard Stallman の gnuemacs である．emacs は以前は手数料程度の金額でリリースされていたが現在は販売されているものもあり，バークレー版や System V とともにリリースされているソフトウェアでない．

emacs はウィンドウをサポートするテキスト・エディタであるが，単なるエディタの機能にとどまらず，一つのプログラミング環境をサポートするユーザ・インタフェースと考えられる．

emacs のコマンド群は，非表示文字から構成される．前述の vi ではコマンド・モードとテキスト・モードがあり，ユーザはこのモードを意識的に制御しなければならないが，これに対し emacs では表示可能な

文字列は常に挿入対象の文字列として扱われる．このため emacs に対するコマンドは，コントロール文字あるいは ECS キーとほかの文字との組み合わせを利用する．emacs のコマンド処理部分はきわめて柔軟性が高いように作成されており，任意のコントロール文字列を emacs の機能に対応させることが可能であり，利用者自身が自分の使いやすいように定義することができる．

また emacs は LISP 処理系を内蔵しているため，LISP で記述したプログラムを追加することにより，emacs に新しい機能を追加し拡張することが可能である．

(2) ウィンドウ制御

emacs では，一つの CRT 画面を複数のウィンドウに分割して使用することができる．それぞれのウィンドウは内部的なバッファに対応し，さらにこのバッファは一般にファイルに対応されるため，一つのファイルの異なった部分を同時に表示したり，複数のファイルを表示し編集作業を行うことが可能である．この機能はプログラムのデータを定義参照しながら命令部分を作成したり，あるいは別のプログラムを参考にしながら編集作業を行う場合などに有効である．

しかしながら一般の端末は画面の大きさが 20 数行程度しかないため，三つ以上の分割表示はあまり実用的ではなく，ビットマップ端末におけるウィンドウとは比較すべきものではない．emacs の画面分割例を図-3 に示す．

(3) コンパイル処理

emacs のバッファは，ファイルへの対応ばかりでなくプロセスに対応させることも可能である．特にプログラムのコンパイル処理を制御する make コマンドについては，これを制御するための専用のコマンドが用意されている．

プログラムの作成が終了した場合，一般のコンピュータではエディタを終了し，またバークレー版 UNIXではサスペンドによってエディタを一時停止させてコマンド・インタプリタのレベルに戻りコンパイルを実行するが，emacs 環境下ではこの操作は emacs 内部からバッファをとおして実行することが可能である．

emacs からコンパイル実行を行うと，自動的に make コマンドの出力を受け取るためのバッファが生成され，このバッファが画面上に表示される．make から実行されたコンパイラが出力したエラーメッセージは，このバッファ上に保持される．さらに make の実行終了後，エラーメッセージに対応するソース・コード部分に順次カーソルを移すことができる．この機能によって，コンパイラの出力したメッセージを書き取ったり，ファイルに保存したりして，ソースコードと対応させるなどといったつまらない作業から解放され，プログラムの作成のみに集中することができる (図-4)．

emacs のバッファを使用したプロセスとの通信は，make コマンドだけに限らず画面を直接制御しないプログラムであればなんでもよい．たとえばログイン時に実行される UNIX のコマンド・インタプリタであるシェルを，emacs のウィンドウをとおして実行することも可能である．実際にはウィンドウでの実行とログイン時のシェルは必要に応じて使い分けることにな

```
CFLAGS  = -g
OBJECT  = main.o copy.o
type: $(OBJECT)
        cc -o type $(CFLAGS) $(OBJECT)
$(OBJECT):
 Buffer: Makefile  File: Makefile  (Normal)  Top
extern  char  stdbuf[BUFSIZ];
extern  int   bflg, eflg, nflg, sflg, tflg, vflg;
extern  int   spaced, col, lno, inline;
copyopt(f)
        register FILE *f;
{
        register int c;
top:
        ch = getc(f);
        if (c == EOF)
                return;
 Buffer: copy.c  File: copy.c  (elec-c)  17%
```

図-3 emacs の画面分割例

```
cc -g -c main.c
cc -g -c copy.c
"copy.c", line 13: ch undefined
*** Error code 1
 Buffer: Error log  File: [None]  (Normal)  13%
extern  char  stdbuf[BUFSIZ];
extern  int   bflg, eflg, nflg, sflg, tflg, vflg;
extern  int   spaced, col, lno, inline;
copyopt(f)
        register FILE *f;
{
        register int c;
top:
        ch = getc(f);
        if (c == EOF)
                return;
 Buffer: copy.c  File: copy.c  (elec-c)  23%
```

図-4 make の実行例

が，すべての処理を emacs から起動／制御することも可能である．

プロセスとの入出力はバッファに保存され，ウィンドウへの表示はファイルのエディットの場合と同様に操作できるため，実行したプロセスの出力が画面から流れてしまい再実行を行うなどの無駄な操作から解放される．また emacs はウィンドウ間のテキストの移動もサポートしており，プロセスの出力を取り込んだ編集なども容易にできる（**図-5**）．

（4） LISP 処理系

emacs は LISP 処理系を内蔵しており，この機能を使って emacs を拡張することが可能である．

emacs は，一般的なコマンドについては標準的なコントロール文字列に対応させており，利用者が何の設定も行わなくてもこのコントロール文字列によって

```
% ls
Makefile    copy.c    main.c    shell.CKP
% uptime
10:04pm up 4 days, 9:04, 13 users, load: 4.83
%
 Buffer: shell* File: [None] (Normal) 97%
extern  char  stdbuf[BUFSIZ];
extern  int   bflg, eflg, nflg, sflg, tflg, vflg:
extern  int   spaced, col, lno, inline;
copyopt(f)
        register FILE *f;
{
        register int c;
top:
        ch = getc(f);
        if (c == EOF)
                return;
 Buffer: copy.c File: copy.c (elec-c) 17%
```

**図-5** シェルの実行例

```
)n  Mon, 16 Jun  kurihara        Quiq Jobs
 N  Mon, 16 Jun  junko@sravd     Project Schedule
 N  Mon, 16 Jun  megumi@sravd    Project Schedule
 N  Wed, 18 Jun  kondo@tokyo     Schedule Control Tool
 N  Wed, 18 Jun  youichi@keio    JEIDA Domain Network
Mail from /usr/kurihara/Messages [0%]

From kurihara Mon Jun 16 14:12:16 1986
To: kurihara
Subject: Quiq Jobs
Status: R

1. Paper
        OHP for UNIX Seminar
2. Project Management
        Registration and Planning Management Rule

Current message [Top]
```

**図-6** rmail マクロの実行例

emacs の機能を選択実行できるが，emacs のコマンドと文字列との対応は LISP 処理系をとおして自由に再設定することが可能である．

また LISP 処理系によって拡張された機能の中でも，一般的に利用できる機能が標準マクロとして提供されている．有名なマクロとしては，以下のようなものがある．

- elec-c……Cプログラム入力
- dired ……ファイル名操作
- rmail ……電子メイル送受信
- spell ……スペル・チェック
- info……マニュアル表示

rmail マクロによる電子メイル処理の例を**図-6** に示す．

## 3. 静 的 解 析

### 3.1 lint—Cプログラムの詳細な文法解析

lint はCプログラム用の文法チェック・ツールである．一般にプログラムの文法チェックはコンパイラによって行われるが，UNIX のCコンパイラはいわゆる UNIX 流のツールであり口数が少なく，またコード生成を目的としているためコード生成に不可欠な必要最小限のチェックしか行っていない．

lint とCコンパイラは同じCプログラムを解析するツールであるが，Cコンパイラのコード生成に対して，lint はCプログラムの詳細な文法チェックを目的としたツールである．lint の行う主なチェック項目には以下のようなものがある．

- 代入文の型の一致
- 実行されない文
- ループ内へのジャンプ
- 使用されていない変数
- エリアが定義されていない変数
- 変数のない論理演算式
- 関数の引数の数とタイプ
- 関数値のタイプ

これらのチェック項目の中で，特に関数の引数に対するチェックは有効であり，自分が定義した関数だけでなくC言語の標準ライブラリで用意されている関数についてもチェックが行われる．C言語のような関数を多用するプログラムでは，この種のチェックは不可欠である．UNIX においてC言語でプログラムを作成した場合は，コンパイルではなく，まずはじめに lint を実行すべきである．

一般に静的解析のツールは無駄なことまでレポートするため，その出力を解析するのに時間を要してしまう．lint も例外ではないが，自分で定義した関数については引数のチェックの省略するように指定することもできるため，型変換のキャスティングと併せて使用することにより，誤りの入り込みにくいプログラムを作成するのに役立つ．

### 3.2 cflow—関数構造の分析

cflow はCプログラムの関数呼び出し関係を解析しコール・グラフを作成するツールである．

cflow は System V で提供されているコマンドであり，バークレー版 UNIX ではリリースされていない．しかしバークレー版 UNIX を使用しているサイトはほとんどが AT & T とソース契約をしていると思われるので，利用者自身の手で移植して利用することもできる（図-7）．

### 3.3 cxref—変数，関数名のクロスレファレンス

cxref はCプログラム用のクロス・レファレンス作成ツールである．

cxref は，cflow と同様に System V で提供されているコマンドである．クロス・レファレンス情報は一般にプリントしておき，変数を参照している部分を検索するために使用されるが，UNIX のような端末で生活をする環境下では紙に出された情報の利用価値は低いかもしれない．

### 3.4 indent—ブロック構造の解析

indent は，バークレー版 UNIX で提供されるCプログラムのプリティプリンタであり，ブロック構造に従ったインデンテーションを行う．

Cプログラムのインデンテーションを行うツールとして cb があるが，cb はブロック構造に従って各行の先頭にタブを付加するだけであり，プログラム・テキストそのものについての編集は一切行わない．これに対し indent は，マージン設定による行の分割，選択可能なブロックを意味する左括弧での行の分割，オペレータの前後へのスペースの挿入，コメント位置の統一など編集を行うことが可能である．

### 3.5 ctags—検索用タグ生成

ctags は，バークレー版 UNIX で提供されるCプログラムの関数エントリを抜き出すコマンドである．

ctags は roff 形式でプリントするための出力を行うことも可能であるが，一般には vi エディタで関数検索のために利用される関数名とその検索パターンの入ったファイルを生成するために利用される．このコマンドによってC言語の標準ライブラリだけではなく，自分が現在作成しているプログラムについても vi の関数検索機能を容易に利用できる．

```
% cflow cat.c
   1            main: int( ), <cat.c 19>
   2            setbuf: < >
   3            fstat: < >
   4            fopen: < >
   5            perror: < >
   6            fprintf: < >
   7            fclose: < >
   8            copyopt: int( ), <cat.c 101>
   9        printf: < >
%
```

図-7 cflow の実行例

## 4. コンパイル制御

### 4.1 make—コンパイル/リンクの自動実行

make は，コンパイル処理に必要なあるいはコンパイルにより生成されるファイルの時間関係から，必要なコンパイル処理の再実行を制御するツールである．

一般にある程度の規模を超えたプログラムを作成する場合は，プログラムの機能分割を行い，分割された機能別にそれぞれ開発を行うため，作成されたソース・コードも機能別にファイルに分割されることになる．特に UNIX では，大きなファイルの処理は得意ではなく，またソース・コードを再利用することも容易になるので，比較的小さな単位に分割されることが多い．しかし複数のファイルで一つのシステムを構成すると，逆に全体をまとめる作業が煩雑になりがちである．

一般にこのようなケースでは，汎用大型コンピュータでいうとカタログ・プロシージャ，あるいは UNIX のシェル・スクリプトのように，コンパイル作業を指示する手順をファイル化しておくことが多い．しかしこれでは，一つのファイルを修正した場合でもすべてのコンパイルが行われることになり，無駄な待ち時間，CPU 時間の浪費につながる．

make ではコンパイル作業手順のファイル化だけでなく，その作業によって生成されるファイルの依存関係を記述することにより，指定されたファイルの更新時刻を比較し必要最小限な処理だけを選択実行することを可能にしている．たとえば図-8 のようなファイルの依存関係があり，make の実行によってファイル exe を生成した後ファイル x. c を修正した場合は，再度 make を実行することにより，ファイル x. o の

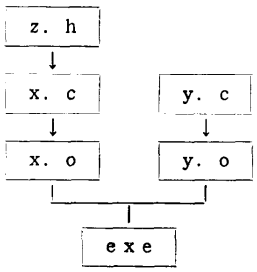



図-8 ファイルの時間的依存関係例

再生成と，さらにそれによってファイルexeの生成だけが実行できる．

makeにおける，ファイルの依存関係の定義は非常にシンプルであり，容易に作成できる．またファイル再生成のための処理方法の定義は，UNIXにある標準的な言語処理系については makeに内蔵しているため，特に記述する必要はない．たとえば上記ファイルの依存関係は，以下のように指定すればよい．

```
exe: x. o y. o
  cc -o exe x. o y. o
x. o: z. h
```

makeは，内蔵しているコンパイルのためのルールのほかに新しいルールを追加定義したり，内蔵されているルールを変更したりすることも容易にできる．このため makeは，コンパイル作業以外でもファイルの更新時刻に依存する定型的な作業であればどんなものにでも利用することができる．UNIXはドキュメント作成の環境としても有名であるが，もちろん roff を使った文書清書処理などにも利用することが可能である．

### 4.2 error—エラー・メッセージ挿入

errorはバークレー版で提供される，Cコンパイラおよび lintのエラー・メッセージとソース・コードをマージするコマンドである．

前述のように emacsではコンパイル時のエラー・メッセージは emacsの中で活用することができるが，errorコマンドはこれと同様な機能を viエディタの利用者にも利用可能にするコマンドである．errorコマンドはCコンパイラのエラー・メッセージを解析しそのエラーが発生した行にC言語のコメント形式でエラー・メッセージを挿入する．これによって viエディタにおいても紙に出されたエラー・メッセージを見ながら修正作業をするという煩わしさから解放される．

errorは本来Cコンパイラおよび lintの出力を処理するコマンドであるが，makeの出力と makeから起動されたCコンパイラの出力を合わせて入力しても問題なく動作する．

## 5. 履歴管理

### 5.1 RCS—ソース・ファイルの世代管理

RCSはパデュー大学で開発され，バークレー版UNIXとともにリリースされているバージョン管理ツールである．

UNIXのバージョン管理ツールとしては，PWB UNIXで開発され System Vでリリースされている SCCSが有名であるが，RCSはユーザ・インタフェースや最新バージョンを取り出す場合の早さなどの点で SCCSよりも優れている．

RCSの主たるコマンドは以下に示す二つだけで，非常に明快なユーザ・インタフェースとなっている．

- ci……ファイルの登録
- co……ファイルの取り出し

RCSは一つの時間列にそった差分管理だけではなく，一つのファイルから複数のブランチを生成し，それぞれの差分管理を行うことが可能である．この機能

```
% ci string. c ---------- RCS への登録
RCS/string. c, v <-- string. c
initial revision: 1. 1
enter description, terminated with ^D or '. ':
> > Sample Program for RCS ---------- コメント入力
> > .
% co -l string. c ---------- ファイルの取り出し
RCS/string. c, v --> string. c
revision 1. 1 (locked)
% vi string. c ---------- エディタで修正
% ci string. c ---------- 差分の登録
RCS/string. c, v <-- string. c
new revision 1. 2; previous revision: 1. 1
enter log message, terminate with ^D or '. ':
> > Add RCS Header ---------- コメント入力
> > .
% rlog string. c ---------- 履歴の表示
RCS: RCS/string. c, v; Working: string. c
head: 1. 2
total revisions: 2; selected revisions: 2
description: Sample Program for RCS
----------------------------
revision 1. 2
date: 86/02/11 13: 02: 36; added/del: 1/0
Add RCS Header
----------------------------
revision 1. 1
date: 86/02/11 13: 01: 29; author: kurihara
Initial revision
%
```

図-9 RCSの実行例

はSCCSでもサポートされているが，RCSではさらに複数のバージョンを一つのマージを補助するためのコマンドが用意されている．また現在SCCSを利用しているユーザのために，SCCSで管理されていたファイルを，RCSに移行するためのコマンドも用意されている（図-9）．

### 5.2 SCCS—PWB UNIX 版世代管理

SCCSはPWB UNIXで開発され，現在System Vでリリースされているバージョン管理ツールである．

前述のようにRCSと比較して特に欠けている，あるいは特筆すべき機能はないが，RCSが最新バージョンを基準に持っているのに対し，SCCSは最初のバージョンとそれぞれの差分を持っているため，バージョンの数が増えるにつれて最新版の取り出しや登録に時間がかかるようになることに注意したい．

SCCSの主なコマンドを以下に示す．

- admin……ファイルの初期登録
- delta ……ファイルの修正登録
- get ……ファイルの取り出し
- edit……ファイルの修正

## 6. テスト実行

### 6.1 dbx—シンボリック・デバッガ

dbxはバークレー版UNIXでリリースされているシンボリック・デバッガであり，C言語およびFORTRANプログラムのデバッグに利用できる．

UNIXにはsdbというシンボリック・デバッガがあり，System Vでリリースされており，バークレー版UNIXでも4.2BSD以前のバージョンではdbxではなくsdbが利用されていた．dbxはsdbの改良版と考えて差し支えない．sdbと比較して特に，実行のトレース表示，構造体変数の表示，サブコマンドの再定義機能などの点について改良が加えられている．

dbxの主な機能を以下に，実行例を図-10に示す．

- スタック表示
- 宣言型による変数内容表示
- 構造体の変数内容表示
- メモリ番地表示
- 変数への値設定
- ソース表示
- ブレーク・ポイントの設定
- シングル・ステップ実行
- トレース表示

シンボリック・デバッガを利用するためには，プログラムのコンパイル時にデバッグのためのオプションを指定する．この種の指定はUNIXに限らず一般のOSでも同様であるが，デバッグ用の特殊な命令の埋め込みの方法については一般のOS下で動作しているデバッガと異なる．

一般のOSでは，コンパイル時にデバッグのオプションを付けた場合，デバッグ用の特殊なライブラリを使用することによって，作成されたオブジェクトの中にデバッガを組み込む場合が多い．このためデバッグ・オプション付きでコンパイルされたオブジェクトは，デバッグ以外に使用することができず，デバッグ

```
% dbx string
(dbx) list 3,20 ---------------------------------- ソースの表示
    3  main (argc, argv)
    4  int argc;
    5  char *argv[ ];
    6  {
    7      static char msg[80] = "ARGS is ";
    8      strcat (msg, argv[1]);
    9      puts (msg);
   10  }
   11
   12  strcat (dest, source)
   13  char *dest, *source;
   14  {
   15      while (*dest)
   16          dest++;
   17      while (*dest++ = *source++);
   18  }
(dbx) stop at 17 --------------------------------- ブレーク・ポイントの設定
(dbx) run args ----------------------------------- 実行
  stopped in strcat at line 17
   17  while (*dest++ = *source++) ;
(dbx) where -------------------------------------- スタックの表示
  strcat (dest="", source="args"), line 17 "string.c"
  main (argc=2, argv=.....), line 8 "string.c"
(dbx) delete 1 ----------------------------------- ブレーク・ポイントの解除
(dbx) trace source ------------------------------- 変数のトレース設定
(dbx) cont --------------------------------------- 実行再開
  initially at line 17: source = "rgs"
  after line 17: source = "gs"
  after line 17: source = "s"
  after line 17: source = ""
  ARGS is args
  execution completed, exit code is 10
(dbx) quit
%
```

図-10 dbxの実行例

終了時に再度デバッグ・オプションをはずしてコンパイルする必要がある．これに対し UNIX のデバッガは，テストするプログラムとは別に存在し，コンパイル時のデバッグ・オプションはデバッガとテスト・プログラムを連結するための命令ならびに情報を埋め込むだけであるため，デバッグ・オプション付きでコンパイルされたオブジェクトでもデバッガを使用せずに実行することが可能である．

### 6.2 prof—関数別の実行時間計測

prof は関数ごとの実行回数と実行時間を解析するツールである，C言語，FORTRAN，さらにバークレー版 UNIX では Pascal のプログラムで使用できる．

prof を使用するためには，シンボリック・デバッグの場合と同様にコンパイル時に実行時間計測のためのオプションを指定する．実行時間計測オプション付きでコンパイルされたプログラムを実行すると，mon.out という名前のファイルが作成され，この中に関数実行の情報が書き込まれる．prof はこのファイルを解析し関数別の集計表示を行う．

バークレー版 UNIX ではさらに gprof というツールがあり，関数のコール・グラフを表示させることができる．

## 7. プログラム生成

### 7.1 lex—字句解析プログラム生成

lex は字句解析を行うCプログラムを生成するツールである．

lex は，字句解析時にワードとして認識したい文字列の指定は，UNIX が採用している文字列検索法である正規表現をさらに拡張した文法によって記述することができるため，キーワードだけでなく変数名などもそのパターンによって容易に記述することができる．たとえば，変数名を「先頭に英文字がありその後に任意個の英数字が続く」と定義したい場合は，以下のように指定すればよい．

[A-Za-z] [A-Za-z 0-9]*

lex における文字列表現の代表的な例を**表-1** に示す．

lex への入力は，ワードして認識すべき文字列とともに，そのパターンを見つけた場合に行う処理をC言語で記述し与えることができる．一般に lex の生成した関数は，文法解析を行うような上位の関数から呼ばれた形で使用されるため，この部分には見つけ出したパターンの情報を上位関数に返すようなコードを指定する．

**表-1** lex の文字列表現例

| 特殊記号 | 記号の意味 | 使 用 例 | マッチするパターン |
|---|---|---|---|
| " " | 文 字 列 | xyz"++" | xyz++ |
| ^ | 行 頭 | ^abc | 行の先頭にある abc |
| $ | 行 末 | abc$ | 行の最後にある abc |
| [ ] | 選 択 | [abcdefg] | abcdefg のどれか1文字 |
| - | 範 囲 | [a-z0-9] | 小文字の英字または数字 |
| ^ | 範囲否定 | [^a-zA-Z] | 英字でない文字 |
| . | 任意文字 | ab.d | ab, b の間に何か1文字 |
| ? | 省 略 可 | ab?c | abc または ac |
| * | 繰り返し | aa* | a, aa, aaa … |
| \| | 選 択 | ab\|cd | ab または cd |
| ( ) | グループ | (ab\|cd+)* | |
| { } | マ ク ロ | {DIGIT} | |

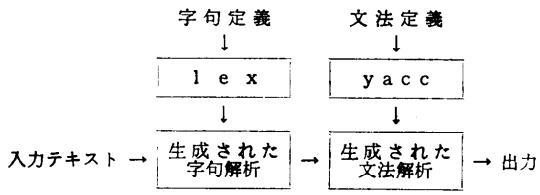



**図-11** yacc と lex の連結利用

### 7.2 yacc—コンパイラ・コンパイラ

yacc はコンパイラ・コンパイラであり，UNIX 上ではコンパイラに限らずある，シンタックスを持った文字列を解析するツールでは，ほとんどすべてのコマンドにおいて利用されている．

yacc への入力は，バッカス記法で記述された文法と，lex の場合と同様に指定された文法に合致した場合に実行すべきC言語で書かれたプログラムである．また当然のことであるが，yacc の生成したプログラムは，lex の生成したプログラムと連結して実行することが可能である（**図-11**）．

しかしながら，字句解析処理自体は一般にそれほど難解なものではなく，かつ lex の生成するプログラムは人間がコーディングしたプログラムより処理効率が悪いので，UNIX の言語処理系の中で lex を利用しているものはほとんどない．lex の利用価値は，字句解析プログラムの短期間での実現にある．このためコンパイラのように CPU 時間を消費し，かつ頻繁に使用されるツールの場合は，最終的には人間がコーディングした字句解析ルーチンに置き換えるとよい．

実際に yacc を利用するには，このほかのいろいろな指定についても知らなければならないが，残念ながら yacc の詳しい利用方法をお話するには，私自身力

不足であり，かつかなりの紙面を必要とすると思われるので，詳細については各自マニュアルやほかの文献を参照してほしい．

## 8. プログラミング環境の構築

前章までで，UNIX に備わっている主なプログラミングのためのツール群について一とおり説明した．

これらのツールを適宜使用することによって，効率的なプログラミングが可能になるはずであるが，その利用方法は利用者自身が自分の努力によって獲得しなければならない．UNIX はプログラミングの環境として有名であるが，この環境という言葉が意味するものは，

「ある利用方法が与えられて，それに従って作業すれば誰にでも効果的に利用できる環境が存在する」

ということではなく

「共通に利用できるコンポーネントと，それを組み合わせて利用できる手段を与えることによって，利用者自身が自分のための環境を構築することができる環境を提供する」

ということである．たとえば UNIX をプログラミング・サポートのために導入した場合でも，

「UNIX のエディタとデバッガを利用しCプログラム開発作業を効率的に行う」

という単純な利用方法だけではなく，

「文字列処理のための豊富なコマンド群と SHELL プログラミングによって有効に活用することができる」

「提供されたソースを活用し，再利用，技術移転に積極的にすすめることができる」

「ネットワークを使って UNIX コミュニティに参加し，最新情報あるいはパブリック・ドメイン・ソフトウェアの流通を促進することができる」

などの多彩な利用形態が考えられる．

誰かが与えてくれることを待つことに慣れてしまった人には，おそらく UNIX 導入の効果は薄いと考えられる．UNIX という環境は，新しいものを積極的に取り込んでいきたいと考えるプログラマにとって，ネットワークという情報交換の手段とソフトウェア実行の共通の基盤を提供する天国の世界なのである．

プログラミング環境が持つべき機能を階層的に整理すれば，下から順に：

（1） 基盤環境：エディタ，コンパイラなどの基本的プログラミング・ツールの集まり．

（2） ユーティリティ・ツール・ボックス：電子メイルそのほか，日常的な雑務(?)を支援するツール群．

（3） ソフトウェア・データベース：ソース・コードやドキュメントを体系的に管理するツール群．

（4） 開発支援システム：単にプログラミング工程だけにとどまらず，要求分析・設計から保守・管理にいたる幅広い開発活動のスペクトラムを支援するツール群．

（5） 方法論ガイド：特定のプロジェクト向けに，ユーザの技術レベルや職務別に相異なるインタフェースで，仕事のやり方（その中での各種ツールの使い方）をガイドするシステム

といったピラミッド構造になる．

UNIX は，この階層構造のうちの，少なくと第 1〜2 層までをかなりの範囲カバーしてくれている．しかし，それより上位の階層を整備し，より快適な環境を作り上げるのは，あくまでユーザ自身の責任なのである．

(昭和 61 年 8 月 6 日受付)