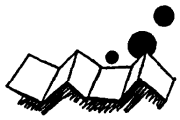
解　説

Shell-コマンド通訳系†



橋爪　宏達††

1. はじめに

UNIX のコマンド通訳系は shell すなわち外殻と呼ばれている．UNIX の OS 核と外部の利用者環境の境界に位置することを象徴する名称といえる．UNIX は他の多くの TSS システムとは異なり，コマンド通訳系を OS 組込み機能とはしていない．単なるユーザ・プロセスの一つである．また以下で述べるように，各コマンドの実行も，fork システムコールにより，すべて個別のプロセスとして実行される．この UNIX の単純なコマンド↔プロセス概念が，shell の機能を強力なものにしている．

本解説では shell の操作仕様についても最小限の記述をするが，むしろパイプラインやフィルタ概念など UNIX の基本的特性と密接した機能を考察しようと試みる．これは UNIX について，明快，単純，柔軟，強力などの美点を見出せるが，その多くがこれらの shell 機能に由来すると考えるからである．

2. Shell の系譜

UNIX は 1969 年に PDP-7 上で原型が成立しているが，このシステムのために K. Thompson が作成したコマンド通訳系を shell の創始と見なすことができる[1]．以来 shell は Thompson の管理下にあり，Bell 研内での UNIX の成長とともに機能拡張が加えられてきた．文献 1) には入出力の切換え (I/O redirection)，バックグラウンドでのコマンド実行，パイプラインなど現在の shell の基本機能が次々に着想された経緯が述べられており興味深い．

その後 PWB/UNIX 用に shell の機能強化が計られたが，これは J. R. Mashey, A. L. Glasser, R. C. Haight らによるものである．さらにこれに基づいて第 7 版 UNIX 用 shell が 1978 年に発表されたが，その開発は S. R. Bourne によるものである[4]．開発者の名をとって Bourne-shell ないし B-shell と呼ばれることがある．B-shell は以後の AT&T 系 UNIX である System Ⅲ，V に継承されている．本解説では特に断わらない限り B-shell に即して shell 仕様を論ずることにする[2),3)]．

一方，カリフォルニア大バークレー校の UNIX プロジェクトでは，W. N. Joy が第 6 版 UNIX の shell を下敷きにして新しい shell を作成した[5]．これは C-shell と呼ばれる．コマンド履歴機能などを装備して会話性向上に力が注がれており，人気が高い．OS 本体は AT&T UNIX を採用するワークステーション・メーカも，C-shell はバークレー系 UNIX から移植している例が多い．実質的に C-shell も標準 shell に準じた立場にあるため，第 5 章でこれに触れることにする．

3. Shell の基本機能

UNIX shell はユーザ・プログラムとして作成できることから無数の実現例があるが，それらはほぼ例外なく以下に述べるコマンド↔プロセス概念，引数展開，入出力の切換え，パイプラインなどのオリジナル shell の基本機能を踏襲している．それは，単にこれらの機能が成功したからというだけでなく，むしろ上記のような基本機能が UNIX の本質にかかわっているためと思われる．

3.1 Fork-exec によるコマンド実行

shell は，ユーザからコマンドが投入される度に，次のようなステップでコマンドを実行する．

(Ⅰ) fork システムコールにより，親プロセス (shell) から子プロセスを生起する．

(Ⅱ) 子プロセスは exec システムコールで，コマンドに対応するプログラムを起動する．

(Ⅲ) 親プロセス (shell) は子プロセスの終了を wait システムコールで待合わせる．

以上のステップを図-1 により順を追って見てゆく．まず shell (sh) が fork コールを実行した直後は，fork を呼出した shell プロセスのコピーが子プロセ

† Shell-the command interpretor by Hiromichi HASHIZUME (National Center for Science Information System).

†† 学術情報センター

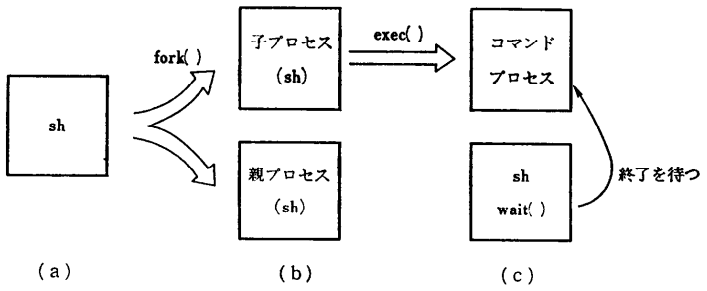


図-1　コマンドの実行過程

スとして生成され，fork コールの戻りの位置から動きはじめる．システム中には二つの shell が親子プロセスとして存在する状態になる．子プロセスは，親プロセス shell の完全なコピーであり，また fork 時点で親がオープンしていた標準入力，標準出力などのファイル記述子，ワーキング・ディレクトリ，プロセス所有者 ID などの属性を引き継いでいる．したがって子プロセスが親と違った機能を果すためには，自分が子であることを判断し，積極的に動作を変える必要がある．

この判定は fork の戻し値により行う．親プロセスの fork コールには生成した子プロセスのプロセス番号（非零値）が返るのに対し，子プロセスの fork は値 0 が返ることを利用する．

この判定後，子プロセスは，プロセスのコア・イメージを目的とするコマンド・プログラムのものに変更し，実行を行う．これには Exec システムコールを使用する．これでコマンドが開始されたことになる（図-1（c））．

親プロセスは子プロセスの終了を wait システムコールで待ち合わせ，その後次のコマンド実行に移る．しかし，親プロセスは待ち合わせを省略して次のコマンド実行に移ることも可能である．この場合，子プロセスはバックグラウンド・ジョブとして実行されている状態になる．コマンドが

Command &

のように & つきで起動された場合の挙動である．

図-2 に以上で述べた shell の動作を簡単な shell をC言語で構成した例を示す．(smplsh と命名してみた．) smplsh にはコマンド引数の処理機能もファイル・ディレクトリのパス検索機能もなく，とうてい実用になるものではないが，fork, exec, wait の機能を理解する上では適当と思う．ステップ(Ⅰ) が 10 行目，ステップ(Ⅱ) が 11 行目，ステップ(Ⅲ) が 13 行目に対応する．

```
 1  #include <stdio.h>
 2
 3  main ( )
 4  {
 5      char command[32];
 6      char *prompt="$ ";
 7
 8      while (printf("%s", prompt),
 9              gets(command) !=NULL) {
10        if (fork( )==0)
11              execl(command, command, 0);
12        else
13              wait(0);
14      }
15  }
```

（a）ソースプログラム

```
1  % smplsh
2  $ /bin/date
3  Fri Oct 17 13:42:30 JST 1986
4  $ /bin/who
5  hasizume tty00 Oct 17 12:45
6  adachi   tty01 Oct 17 11:03
7  $ ^D
```

（b）実行例

図-2　簡単な shell の実現と実行例

3.2　ファイル名の展開

shell はコマンド行を構成する語中に *，?，[などの総称文字（ワイルドカード文字）を見出すと，総称文字と対応するファイル名を求めてファイル・ディレクトリのサーチを行い，マッチングのとれた文字列に置換，展開する機能を持つ．

たとえば現ディレクトリ中に a, b, c の三つのファイルがあった場合，

echo *

というコマンドでは*の位置にこの三つのファイル名が展開され，結果として echo コマンドは

echo a b c

として起動されたのと同価になる．

3.3 入出力の切換え

UNIX 上のプロセスで使用するファイルには，通番で指示されるファイル記述子が与えられる．コマンド実行の際，標準的には，ファイル記述子0で表される「標準入力 (standard input)」は端末キーボードからの入力ファイルに，ファイル記述子1の「標準出力 (standard output)」は端末の画面への出力ファイルにそれぞれ割当てられる．さらにファイル記述子2は，標準出力と同じく端末の画面に接続されるが，主にエラーメッセージの出力に使用され，「標準エラー出力 (standard error output)」と呼ばれる．

コマンド投入時に，標準入出力ファイルに関し，〈，〉の文字で他のファイルに出入口の変更を指定できる．これを「入出力の切換え」と呼んでいる．たとえば，

sort 〈abc 〉xyz

というコマンドは，ファイル abc 内の行を ASCII コード順にソートし，結果をファイル xyz に格納する．機能的には shell 内では，次のような動作を行っている．まずファイル abc と xyz をオープンする．(もし出力ファイル xyz が存在していなければ新規作成する．) このオープン操作で得られた abc と xyz に対応する新しいファイル記述子を，おのおの標準入出力のファイル記述子 0，1 にコピーする．コピー操作には dup システムコールを使用する．

入出力切換え機能は，従来の多くの OS のコマンド言語でも，論理入出力ファイル名と物理入出力ファイル名の対応機能として用意されていたものである．しかしこの機能自体をたとえば allocate コマンドなどの名称で，複雑なシンタクスを持つ個別コマンドにする場合が多かった．shell の入出力切換えは，〈，〉などの記号のみで手軽にかつ日常的にこの機能を利用できるようにしていることが重要である．その結果，すべてのコマンドにつき，たとえファイル入出力を行うのが普通と想定されるものについても，基本的に標準入出力（端末キーボードと画面）を取扱い対象ファイルとしてプログラムしておくことが UNIX 上での慣習となった．さらに shell の入出力切換えの概念は，次節のパイプ機能に発展する．

3.4 パ イ プ

従来の OS では，一つのデータを各種のプログラムで次々に加工していく場合には，中途に中間ファイルを設けて各ステップを順次起動していた．中間ファイルの準備，中間ファイルへの入出力割付けなどに個別のコマンドを必要とする OS 上で，このような形態でシステムを利用する場合には，ユーザは大変な心理的負担を要求されることになる．UNIX ではファイルの新規作成に特に面倒なことはなく，shell の入出力切換えも手軽であるから，他の OS より条件は良いが，それでも中間ファイル名を個々に考案したり，使用済みファイルを消去したりする負担が残る．UNIX では入出力切換えを発展させ，一つのコマンドの標準出力を直接に次のコマンドの標準入力にするというアイデアを採用した．これでユーザは中間ファイルの取扱い

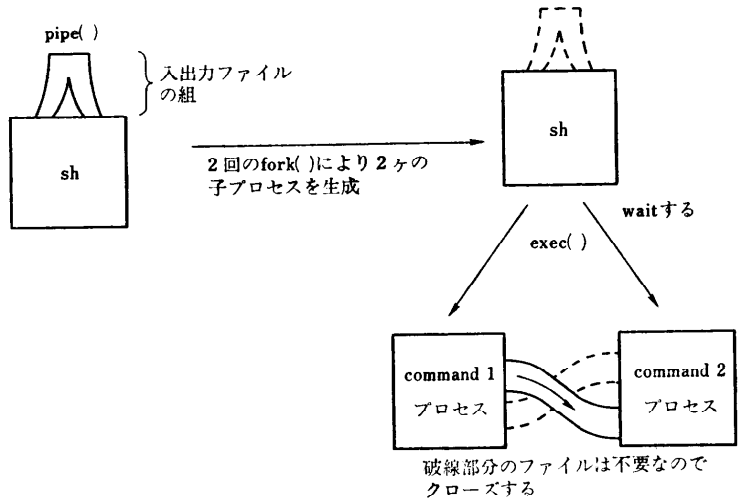


図-3 パイプによるプロセス間通信

に関する一切の作業から解放されることになった．これが shell のパイプ機能である．

パイプ機能は，

Command 1|*Command* 2|*Command* 3

のように，個々のコマンドの間に縦棒記号を置いた簡単な記法で起動される．上記のような形式のコマンド行を「パイプライン」と呼び，また縦棒自体は「パイプ」と呼ばれる．

shell 内部では，パイプは pipe システム・コールにより実現されている．図-3 で，shell がコマンド間でパイプを形成する様子を模式的に示す．まず pipe システム・コールにより OS にプロセス間通信路（パイプ）を用意させる．この段階ではパイプはまだプロセス間を接続する形態にはなっていない．shell プロセス中にパイプの出入口が2個のファイル記述子として与えられているだけである（図-3（a））．次に2個のコマンドに対応する子プロセスを順次 fork により生成する（図-3（b））．fork により得られた子プロセスには，パイプも親プロセスから遺伝するプロセス属性の一部として引渡される．子プロセス内ではおのおの，パイプの入口，出口を，標準入力，標準出力に，dup システム・コールを使って対応させる．これでパイプラインが完成する．その後は，おのおののコマンド・プロセスを exec システム・コールで実行すればよい．

以上では，パイプラインの概念を説明するために，パイプを中間ファイルに対応するものとして紹介した．事実，UNIX 第6版当時に小規模 PDP コンピュータ用として存在した mini-UNIX では，主記憶サイズの制約から，パイプ動作を実際に中間ファイル生成により模擬していた．しかし他の UNIX ではパイプはプロセス間通信として実現しており，中間ファイルは実質的に存在しない．各コマンド・プロセスはマルチプロセス環境下で並行動作している．したがってパイプラインの起点で発生したデータは，起端のコマンド・プロセスの終結を待たずに次々と後続のプロセスへ伝達される．以上のような特徴は，中間ファイルを使用した多段コマンド実行では得られないものである．たとえば通信プロトコルで使用される応用層，プレゼンテーション層…などをパイプで連結させ個別コマンド・プロセスとして実現することが可能である．

パイプの着想自身も，実は中間ファイルとは無関係のもので，ベル研の M. D. McIlroy の「コマンドを2項演算子として考える」という概念から発生している[1]．すなわち，たとえば通常

sort *inputfile outputfile*

という記法をとる sort コマンドは，sort という2項演算子を prefix notation で用いたものと考える．そしてさらにこの infix notation である

inputfile sort *outputfile*

を考えてみると，sort 自身は左側の入力ストリームを加工し右側出力ストリームへ廻す作用素とも考えられる．これがパイプの着想の原形である．UNIX システムでパイプラインを使用するユーザも，コマンド入力時にパイプを中間ファイルに対応する機能として使用しているという意識は少ないであろう．むしろ「いくつかの基本コマンドを組合せて，その場で自分の必要とする高次元機能を果すコマンドを作る」といった意識であろう．

3.5 フィルタ

フィルタは機能として shell ないし UNIX に存在する実体ではない．以上に述べた shell の入出力切換え機能やパイプ機能から導かれた UNIX 固有のプログラミング・スタイルを指す．

パイプを使用するユーザは，個々のコマンドを自分の必要とする高次機能を実現する要素機能としてとらえる．このような場合のコマンドは，データ全体を引込んで複雑・大規模な処理加工を施すといったものでは不適当で，むしろ標準入力からのデータストリームに単純な処理加工を施し，次々と標準出力へ送るようなものであって欲しい．そのような使用法を意図したコマンドをフィルタという．信号処理のフィルタから連想された名称である．UNIX のコマンド・セットの大部分がフィルタとして使用しうるよう考慮されているが，中でも特にフィルタに好んで使用される代表的なコマンドには，

grep—ファイルの中から，特定パターンの文字列を持つ行を捜す．

sort—ファイル内の行のソート．

tail—ファイルのおしまいの数行を表示する．

sed—ストリーム・エディタ．入力ファイルの内身に適当な編集加工を施す．

などがある．単純かつ明解な機能仕様を持つコマンド群である．

コマンドをフィルタの概念で組合せて使用することで，UNIX はユーザの複雑かつ変化に富む要求に柔軟に対応できる．またユーザの作成するプログラムも，フィルタの概念に適合させると，必然的に標準入出力

を使った単純機能の小規模なものとなる．現在ソフトウェア工学の一環としてプログラムのモジュール化技法が研究されているが，UNIXのフィルタはこの先駆かつ典型と見ることができよう．

なお，UNIXのコマンドはフィルタとして使用するため，余計なメッセージを極力出さないように注意して作られている．一部のユーザにとってはこの寡黙さが不気味に感じられるらしく，UNIXに対する批判の一部となっている．

4. プログラム言語としてのshell

3.1で見たようにshell自身も一般のプロセスであることから，shell自身を明示的に'sh'という名称のコマンドとして呼出して実行できる．実行させたいコマンド群をファイルcommfileに用意しておき，

```
sh commfile
```

として，shell自身を子プロセスとして呼び出す☆．このshellはcommfile中のコマンドを次々と孫プロセスを作ることによって実行する．このようにshell自身を再帰的に実行することは，UNIXがコマンド通訳系をOS組込み機能にせず，ユーザ・プロセスで実行していることから容易に実現できる．

shellではパイプラインの段階でコマンド相互を組合わせられるので，高レベルのプログラミングが可能と言える．しかしさらに，shellには変数機能や制御構造が導入されており，本格的なプログラム言語として機能する．shellのプログラミング機能を用いて，ひとまとまりのコマンド・プロシージャとして作成されたファイルは，'shell script'と呼ばれる．なお，通訳系はパソコンのBASICのようにコマンド処理機能とプログラム言語を兼ねている例が多いが，shellの場合は通訳系自体を再帰的に呼び出せる点がユニークである．

4.1 Shell変数

プログラム言語としてのshellが扱うデータは文字列型のみであり，これを格納する変数がshell変数で

表-1 Shellの組込み変数

$#	引数の数
$*, $@	shellに与えられた全引数
$-	shellに与えられたオプション群
$?	最後に実行されたコマンドの返した値
$$	shellのプロセス番号
$!	最後に実行されたバックグラウンドコマンドのプロセス番号
$ HOME	cd (change directory) コマンドの標準引数値（ホームディレクトリ）
$ PATH	コマンド・ファイルを検索するパス名称
$ MAIL	メールの到着の通知，メールボックスなどを指定する
$ MAILCHECK	
$ MAILPATH	
$ PS1	コマンド入力促進文字．標準は'$ '．
$ PS2	継続コマンドの入力促進文字，標準は'> '．

☆) 実際にはcommfileについて，ファイル属性を「実行可能」と設定しておく．これによりshell scriptの実行は，わざわざ'sh～'とするまでもなく，ファイル名そのものがコマンド名称になる．

```
case $# in
0)      set 'date'; m=$2; y=$6;;        # no args: use today
1)      m=$1; set 'date'; y=$6;;        # 1 arg: use this year
*)      m=$1; y=$2;;                    # 1 args: month and year
esac
case $m in
jan* | Jan*)        m=1;;
feb* | Feb*)        m=2;;
mar* | Mar*)        m=3;;
apr* | Apr*)        m=4;;
may* | May*)        m=5;;
jun* | Jun*)        m=6;;
jul* | Jul*)        m=7;;
aug* | Aug*)        m=8;;
sep* | Sep*)        m=9;;
oct* | Oct*)        m=10;;
nov* | Nov*)        m=11;;
dec* | Dec*)        m=12;;
[1-9] | 10 | 11 | 12) ;;                  # numeric month
*)                  y=$m; m='' '';;     # plain year
esac
/usr/bin/cal $m $y                      # run the real one
```

図-4 Shell scriptの例

表-2 Shell の制御構造

```
for name [in word...] do list done
case word in [pattern [|pattern]...list ;;]...esac
if list then list [elif list then list]
    ...[else list] fi
while list do list done
```

ある．shell 変数は shell script 中で

$ 名前

の形式で参照され，参照位置で内容の文字列に展開されコマンド行の一部として評価される．変数の初期化，消去，値の代入，参照などの操作ができる．なお**表-1** のような定義せずに使える shell 変数（組込み変数）がある．これらは shell の内部状態を直接反映するものになっている．

shell 自身は shell 変数に対する演算機能を持たない．演算は通常のコマンドを呼出すことで実施する．しかしshell は shell 変数の内容に対する評価機能と強力なパターン・マッチング機能を持っている．**図-4** は，shell script の例であり，コマンド起動時の引数で指定された年月のカレンダを出力する．引数の解析にパターン・マッチング機能を使っている典型例と言える．

4.2 Shell 制御構造

shell は高級プログラム言語に見られるような制御構造を持っている．その代表的なものを**表-2** に示す．なお，B-shell では以前の shell にあった **goto** *label* の制御構造が削除されたのが注目される．

4.3 Make による高高レベル・プログラミング

UNIX の make コマンドは，プログラム群の維持管理を目的としたユーティリティである．Makefile という名称のファイル中にソース・ファイル群から目的ファイルを作成するまでの条件，手順をプログラムしておくと，make コマンドはこれに基づき動作する．Makefile 中の個々の目的ファイル導出手続きは shell script で記述されており，その意味で make は shell よりさらに上位の高高レベルプログラム言語と言える．

5. C-shell

C-shell は第2章で紹介したようにバークレー系 UNIX の標準 shell であり，B-shell とは幾分異なった機能及び性質を持つ．C-shell の名称自体は，**表-3** に示すように，C言語をまねた制御構造を持つよう設計されていることによる．しかし，C-shell が評価を受けているのはプログラム言語としてではなく，むしろ C-shell を直接会話的に使用した場合に，マン・マシン・インタフェースへの配慮が行き届いている点である．

表-3 C-shell の制御構造

```
∘ foreach name (list)
      ⋮                          break, continue を使える.
  end
∘ if (expression) command
∘ if (exprssion) then
      ...
  else if (expression) then
      ...
∘ endif
∘ while (expression)
      ...                        break, continue を使える.
  end
∘ switch (string)
    case str 1
          ...
          breaksw
      ...
  default
      ...
  endsw
```

この面での C-shell の特徴を代表するのは，

コマンドの履歴・参照機能（history 機能）

ジョブ・コントロール機能

の二つであろう．

history とは C-shell が過去に実行したコマンドを覚えている機能である．ユーザは必要に応じて以前実行したコマンドを検索，修正して，再び投入することができる．一般端末上で，インテリジェント端末と似たコマンド操作が可能である．

ジョブ・コントロール機能とは，現在実行中のフォアグラウンド・ジョブ☆(C-shell が終了を待合せているコマンド）やバックグラウンド・ジョブに中断をかけ，中断中のジョブを強制終了させたり，フォアグラウンドないしバックグラウンドで続行させる機能である．コマンドをフォアグラウンドで投入したときに，そのコマンド実行が思いのほか長時間を要し，別のことができなくなり困ることがある．ジョブ・コントロール機能によれば，このようなコマンドは自在にバックグラウンドへ回すことができ便利である．

その他，ワーキング・ディレクトリのネスト化機能などもある．B-shell に比べて高い会話性能を持ち，多くの AT&T 系 UNIX マシンの上でも使用されている．AT&T でも上述のような C-shell の優位性は認めており，これに対抗しうる shell として 1983 年に Korn-shell ないし K-shell を開発した．名称は開発者の David G. Korn にちなむものである．

☆) この場合の「ジョブ」とは，現在実行中のコマンド・プロセスあるいは実行を中断されているコマンド・プロセスを指すものである．

6. 新しい shell への試み

shell は公開された OS のインタフェースのみを使用して完全なユーザ・プログラムとして作成できることから，無数の変種が作成されている．おそらく UNIX システム・プログラミングに興味を持った人の大部分が自己流の shell を作った経緯があると思われる．その中で，報告されている代表的なものを紹介する．

まず shell を LISP 系言語仕様のコマンド通訳系にしようとするいくつかの試みが報告されている[6]．shell の基本動作は文字列処理とパターン・マッチングにあるから，これから LISP の諸機能を連想するのは自然であろう．また記号処理専用言語である SNOBOL 4 の仕様で shell を作る試み[7]，shell を関数型言語のパラダイムでとらえる試み[8] なども報告されている．いずれも，高レベル・プログラム言語としての shell の特性改良を目的としている．

一方，直接のマン・マシン・インタフェースの道具としての shell に関しては，最近のワークステーションの動向が今後を示唆している．このような機器では，ビットマップ表示とポインティング・デバイスを使用した OS オペレーションが標準になりつつある．ビットマップ上のマン・マシン・インタラクションについてはいまだ決定的手法はないが，現在のところオブジェクト指向のインタフェースが注目されている．これは従来からの shell とはかけ離れたパラダイムに基づくものである．ワークステーションの本格的活用の時代を迎えている現在，shell のようなコマンド/応答言語は転換期にあると言えよう．

謝辞 本稿をまとめるにあたって貴重なご助言をいただいた学術情報センターの安達淳氏，図書館情報大学の長谷部紀元氏に深謝いたします．

参考文献

1) Ritchie, D. M.: The Evolution of the UNIX Time-Sharing System, AT & T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, pp. 1577-1593 (1984).
2) UNIX™ System V User Reference Manual, AT & T (1984).
3) UNIX™ System V User Guide, AT&T (1984).
4) Bourne, S. R,: An introduction to the UNIX Shell, UNIX™ Time-Sharing System UNIX Programmer's Manual, Seventh Edition, Vol. 2 A (1979).
5) Joy, W.: An Introduction to the C Shell, UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Vol. 2 C (1983).
6) Ellis, J. R.; A Lisp Shell, SIGPLAN Notices 15, 5, pp. 24-34 (1980).
7) Fraser, C. W. and Hanson, D. R.: A High-Level Programming and Command Language, SIGPLAN Notices, Vol. 18, No. 6, pp. 212-219 (1983).
8) Shultis, J.: A Functional Shell, SIGPLAN Notices, Vol. 18, No. 6, pp. 202-211 (1983).

(昭和61年11月7日受付)