and analysis of performance models applicable to man–machine systems evaluation," Bolt Beranek and Newman Inc., BBN Rep. 3446, Mar. 1977.
[32] E. C. Poulton, *Tracking Skill and Manual Control*. New York: Academic, 1974.
[33] E. A. Riddle, "Comparative study of various text editors and formatting systems," Air Force Data Services Center, The Pentagon, Rep. AD-A029 050, Aug. 1976.
[34] T. B. Sheridan and W. R. Ferrel, *Man-machine Systems: Information, Control, and Decision Models of Human Performance*. Cambridge, MA: MIT, 1974.
[35] D. G. Hoecker and R. W. Pew, "User input to the design and evaluation of computer-assisted service delivery," BBN, Tech. Rep. 4358, Mar. 1980.
[36] A. Cakir, D. J. Hart, and T. F. M. Stewart, "The VTD manual," Inca-Fiej Research Association, Tech. Rep. 1979.

Multiparty Grammars and Related Features for Defining Interactive Systems

BEN SHNEIDERMAN

Abstract—**Multiparty grammars are introduced which contain labeled nonterminals to indicate the party that produces the terminal string. For interactive person–computer systems, both the user commands and system responses can be described by the linked BNF grammars. Multiparty grammars may also be used to describe communication among several people (by way of computers or in normal dialogue), network protocols among several machines, or complex interactions involving several people and machines. Visual features such as underlining, reversal, blinking, and color, window declarations, and dynamic operations dependent on cursor movement are also covered.**

I. Introduction

INTERACTIVE computer systems are increasingly popular, but the design tools available for specifying systems do not meet the complex needs of interaction. The Backus–Naur form (BNF) [1] for specifying grammars, and therefore the language generated, is geared towards parsing statements in batch oriented programming languages such as Fortran. The metanotation used for Cobol [19] or the syntax diagrams used for Pascal [11] are reasonable alternatives to BNF, but they do not contain facilities for interactive systems design.

Programming languages such as APL and Basic were created with facilities which emphasized the writing of interactive systems. Input control routines had simple but rigid rules for valid input, and then a user's program had to respond to the input. This approach is effective, but it is difficult to standardize, document, modify, comprehend, and debug the interactive dialogue. Checks for completeness and consistency must be done by hand in an *ad hoc* manner.

Manuscript received January 16, 1981; revised February 11, 1981.
The author is with the Department of Computer Sciences, University of Maryland, College Park, MD 20742.

Computer-assisted instruction systems such as Planit [6] or Tutor [20] enable authors to prepare a frame with some text followed by a question and then to list possible human responses with accompanying messages to be displayed by the computer. This approach simplified the course author's task, helped in checking for completeness and consistency, and facilitated debugging and modification. Unfortunately the scope of application and the flexibility of computer-assisted instruction systems is limited. Supplements to common programming languages have also been offered as tools for constructing interactive systems. These supplements include Fortran subroutine packages, extensions to standard languages, and Pascal data types [8].

Recognizing the inadequacy of the available tools, Parnas suggested transition diagrams for defining interactive computer systems [16]. Transition diagrams have labeled nodes which indicate an initial state, possibly multiple terminal states, and possibly multiple intermediate states. The directed arcs are labeled with a possible input string followed by the system response to that string. Feyock [7] described transition diagrams in the context of computer-assisted education and help systems, and Denert offers a variant of these ideas [5]. Wasserman and Stinson, like Feyock, emphasized that the system response on the arc may involve the invocation of another transition diagram [26]. This subroutining idea is essential if transition diagrams are to be modularly organized and comprehensible. Wasserman and Stinson demonstrated a machine processable encoding of the transition diagram and are more attentive to details of interfacing with a procedural language to carry out computations. They give a realistic evaluation of the advantages and disadvantages of the transition diagram approach. Transition diagrams have also been suggested for

0018-9472/82/0200-0148$00.75

describing interprocess communication activity between computers in a network [2], [23], [24].

Variants of Petri nets [17] have been used to describe process coordination, but they are similar to augmented transition diagrams when used to describe interaction. The use of BNF to describe human input in interactive dialogues has been suggested by Colmerauer [4] and by Hanau and Lenorovitz [9], but both resort to other notational mechanisms for describing the machine's response.

Moran offers an ambitious alternative [15] called the command language grammar which provides four levels of definitions for completely specifying syntax and semantics. The ideas in Moran's work are important but the method appears to be extremely complex. Reisner [18] extends the classic BNF to create an *action grammar* for specifying the operation of a color graphics system called Robart. She uses a plus sign between terminals and nonterminals to indicate the passage of time. Reisner's action grammar is used to specify the complex actions of a user in pushing buttons, making lightpen touches, and entering commands or data from the keyboard, but does not include the specification of the system response.

The idea of an action grammar suggested a natural generalization to a *two-party action grammar* where the action of both parties could be specified using familiar BNF grammar tools. Generalizing further yields *multiparty action grammars* which describe the actions of several parties, people or machines, using the same notation. Such *multiparty action grammars* would be useful in describing simple interactive person–computer systems, more complex teleconferencing, game playing, commodity exchange, and other social interaction systems involving several people and one machine, or network protocol interaction among several machines (without human intervention).

These multiparty grammars might also be useful to psycholinguists in modeling communication among two or more humans. Research on natural language grammars focuses on parsing a single sentence at a time and treats a dialogue as a sequence of disconnected sentences. A more inclusive model of human communication would treat a dialogue as a connected sequence of sentences. Artificial intelligence researchers have used this approach to construct natural language question answering systems which conduct clarification dialogue [3], [25]. Even early systems such as Doctor [27] constructed a dialogue by taking phrases from the human input and using them to generate the output. For example, the human entering "YOU ARE AFRAID OF ME" will cause the computer to respond "DOES IT PLEASE YOU TO BELIEVE THAT I AM AFRAID OF YOU." Heidorn's transformational question answering system [10] describes a decoding phase during which the human input is parsed and an encoding phase during which the results of the parse are used to generate the machine's response. Mann *et al.* [13] shed further light on human dialogue with their comprehension model.

This paper proposes extensions of BNF grammars to describe the actions of several parties involved in a dialogue. After these basic ideas are presented, solutions to specific problems of describing person–computer interactions are offered.

II. Multiparty Grammar

Given the hard copy printout from a person–computer interaction, we could produce a grammar which parsed the entire printout, but such a parse should indicate which party produced each line of output. This suggests that there are really two independent grammars; one for parsing the machine output and one for parsing the human entries. A more precise description is that there are two grammars, one for each party, but that there is some interplay between the two grammars. For example, the command:

DELETE GRADES

might produce the machine response:

FILE GRADES HAS BEEN DELETED

or the diagnostic message:

DELETE FAILS BECAUSE GRADES HAS PROTECTION KEYS.

The human commands may be described in the user manual with a BNF grammar, but the machine responses are often merely listed in the last pages of the reference manual. The linkage between the human entries and the machine responses is defined by a lengthy and inaccessible program.

In a multiparty grammar, a BNF grammar can be used to describe the human entries, machine response (acknowledgment or diagnostic), and some aspects of the interaction. Usually, the human-related BNF grammar is used to *parse* the input while the computer-related BNF grammar is used to *generate* the output. In other circumstances the human-related BNF grammar may be used to generate test data.

Nonterminals of a multiparty grammar are labeled by the party. Human nonterminals might be distinguished by an "H" immediately after the left angle bracket, and computer nonterminals might have a "C." For example, ⟨H: VALID-ACCT⟩ is a nonterminal for a human entering a valid account number and ⟨C: ACCEPT-ACCT⟩ is a nonterminal for the computer's response when accepting an account number. Nonterminals without an H or a C describe complex actions involving one or more parties which are specified in other BNF productions. Productions with H nonterminals on the left are used for recognition parsing of input strings, and productions with C nonterminals on the left are used for generating strings. In the simple account acceptance example there might be a production of the form:

⟨ACCOUNT ACCEPTANCE⟩: : =
⟨H: INVALID-ACCT⟩ ⟨C: ACCEPT-ACCT⟩|
⟨H: INVALID-ACCT⟩ ⟨C: DIAGNOSTIC-FOR-ACCT⟩.

Since multiparty grammars deal with interactions, it will be necessary for one party to respond directly to the contents of the message sent by another party. When two humans meet the dialogue might be

Party 1: GOOD MORNING MY NAME IS GEORGE
Party 2: HELLO GEORGE.

Party 2 has taken the name GEORGE from Party 1's statement. To represent this idea in multiparty grammars, nonterminals acquire the value of the most recent parse. Surrounding a nonterminal with square brackets indicates that the value of the nonterminal is to be used in the generation of output. Thus the human dialogue would be represented as

⟨DIALOG⟩ : : = ⟨1: GREET⟩⟨2: RESPOND⟩
⟨1: GREET⟩ : : = GOOD MORNING MY NAME IS ⟨1: NAME⟩
⟨1: NAME⟩ : : = ⟨1: IDENTIFIER⟩
⟨2: RESPOND⟩ : : = HELLO [⟨1: NAME⟩]

where ⟨1: IDENTIFIER⟩ is any string of characters. The square brackets indicate that whatever name was given by Party 1 is used by Party 2. Subsequent parses involving ⟨1: NAME⟩ would assign new values.

BNF grammars, production systems, and other metanotation schemes emphasize the parsing of correct syntactic forms and have meager facilities for recognizing errors, but a large part of interactive systems design is coping with incorrect syntactic forms. A simple approach is to extend the grammar to describe all possible inputs, so that the grammar recognizes all strings. Familiar typographic errors or common user mistakes might be described in the grammar, and then specific diagnostics could be easily produced. Good systems designers recognize that illegal entries deserve responses that are as well designed as responses to legal entries. Unfortunately, writing BNF productions for each possible set of input tokens leads to complex grammars which obscure the normal process. It would be convenient to have a nonterminal which matches any string, if all other parses fail. In this notation ⟨H: *⟩ will indicate such a nonterminal. For example, a highly simplified command language might contain

⟨COMMANDS⟩ : : =
⟨H: OPEN⟩⟨C: OPEN-ACKNOWLEDGE⟩|
⟨H: CLOSE⟩⟨C: CLOSE-ACKNOWLEDGE⟩|
⟨H: *⟩⟨C: OPEN-CLOSE-DIAGNOSTIC⟩

⟨H: OPEN⟩ : : = OPEN⟨H: FILENAME⟩|
O⟨H: FILENAME⟩

⟨C: OPEN-ACKNOWLEDGE⟩ : : =
[⟨H: FILENAME⟩] IS NOW OPEN

⟨H: CLOSE⟩ : : = CLOSE ⟨H: FILENAME⟩|
CL⟨H: FILENAME⟩|
C⟨H: FILENAME⟩|
END⟨H: FILENAME⟩

⟨C: CLOSE-ACKNOWLEDGE⟩ : : =
[⟨H: FILENAME⟩] HAS BEEN CLOSED

⟨C: OPEN-CLOSE-DIAGNOSTIC⟩ : : =
[⟨H: *⟩] CANNOT BE RECOGNIZED
AS AN OPEN OR CLOSE.

The ⟨H: *⟩ nonterminal must be used cautiously, and every effort should be made to parse as much of the input as possible so as to provide the best possible diagnostic message. Good diagnostics should avoid accusatory terms such as ERROR, INVALID, or ILLEGAL but should be constructive and suggest what needs to be done to set things right. The value of ⟨H: *⟩ is the last value assigned to it; allowing echo printing of unparsable statements.

1. ⟨LOGON⟩ : : = ⟨START⟩ ⟨ACCT⟩
2. ⟨START⟩ : : = ⟨H: INITIATE⟩ ⟨C: READY-ACCT⟩ |
⟨H: INVALID-INITIATE⟩⟨C: CRLF-REQUEST⟩
3. ⟨H: INITIATE⟩ : : = I◯
4. ⟨H: INVALID-INITIATE⟩ : : = ⟨H *⟩◯
5. ⟨C: READY-ACCT⟩ : : = READY FOR ACCOUNT NUMBER◯
6. ⟨C: CRLF-REQUEST⟩ : : = TO SIGNON TYPE AN "I" AND HIT ENTER
7. ⟨ACCT⟩ : : = ⟨H: VALID-ACCT⟩ ⟨C: ACCEPT-ACCT⟩ |
⟨H: *⟩ ⟨C: ACCT-REQUEST⟩
8. ⟨H: VALID-ACCT⟩ : : = ⟨H: NUM⟩ ⟨H: NUM⟩ ⟨H: LETTER⟩◯
9. ⟨C: ACCEPT-ACCT⟩ : : = LAST SIGNON FOR [⟨H: VALID ACCT⟩] WAS ⟨LAST-SIGNON-INFO⟩
10. ⟨C: ACCT-REQUEST⟩ : : = ACCOUNT NUMBERS ARE TWO DIGITS FOLLOWED BY A LETTER◯ ⟨C: READY-ACCT⟩

where ⟨H: *⟩ is a pattern match nonterminal which is attempted only after other parses have failed.

⟨H: NUM⟩ : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨H: LETTER⟩ : : = A | B | C . . . | Z
◯IS THE CARRIAGE RETURN AND LINE FEED CODE

Fig. 1. Example of log-on procedure using multiparty grammar.

To recapitulate, three features have been introduced which distinguish multiparty grammars from the traditional BNF:

1) labeling nonterminals with a party identifier;
2) assignment of values to nonterminals and the use of square brackets to output the value;
3) a nonterminal which matches any string if no other parse succeeds.

Fig. 1 contains an example log-on procedure described with a multiparty grammar. Production 1 shows that a log-on consists of a starting phase and an account number phase. Production 2 shows a proper and an invalid human initiation (described in productions 3 and 4) with appropriate computer response (described in productions 5 and 6). Production 7 shows a proper and an invalid account number entry (described in production 8) and the appropriate computer responses (described in productions 9 and 10). In production 9 the computer repeats the user's account number by displaying the value assigned to ⟨H: VALID-ACCT⟩ in production 8 and shows information about the last sign-on that is generated on other productions.

Multiparty grammars describe the syntax of interactions and a small portion of what might be called the semantics. Since programming operations or database functions may

be very complex, the complete description of an interactive system would require use of some programming language description. The approach taken in compiler generators such as XPL (McKeeman *et al.* [14]) or YACC (Johnson [12]) is suitable. Each production which is used for recognition of input might be followed by a series of programming language statements which perform computations, compare values, search databases, etc. Productions which generate text do not need further elaboration.

Implementing the square bracket which allows for assignment of values to nonterminals presents some problems. A nonterminal appearing in several productions or in a recursive production may receive multiple values during a parse. Two approaches seem possible. A stack could be implemented for each nonterminal and each value would be placed on the top of the stack. Reference to the nonterminal would include an optional number to indicate which value was desired. For example, [⟨H: ENTRY⟩(−1)] would indicate the entry below the top of the stack. A simpler implementation would require only a single value for each nonterminal. The user would be required to write code associated with each production which copied the value into temporary storage areas for later use.

The ⟨H: * ⟩ feature also presents implementation problems. Its use should be restricted to the parsing of input, and it is applied only if all other parses fail. A delimiter such as a ":" or " * /" or carriage return to the right of this nonterminal is necessary to avoid a parse which takes all the remaining text.

The advantage of using an automatic compiler generator system is that some aspects of completeness and ambiguity can be checked. Nonterminals which are undefined or unreferenced can be detected by the automatic system and unique parse trees can be guaranteed. These features are a tremendous aid to the design of large interactive systems where the volume of detail may be enormous.

Having a machine-readable specification which is used to produce the final system may lead to fewer implementation problems and delays. Variants in user command formats can be quickly and easily implemented, thereby facilitating pilot testing and rapid improvement [22]. The machine specification may also aid in the automatic construction of user aids such as HELP facilities, UNDO commands, and status requests.

Multiparty grammars have the advantage of familiarity: the similarity to BNF makes this notation natural to a large community of researchers and system developers. Productions can be made readable to humans while preserving the precision necessary for machine processing. The presence of numerous nonterminals which have meaningful names aids human comprehension. Level structuring and modular organization of productions permit meaningful sections of the grammar to be grouped and facilitate comprehension. Labeling of nonterminals with the name of the party is a further aid.

A potential advantage of multiparty grammars is that standards for interaction can be established by specifying the grammatical forms which are permissible. Grammars, when prepared in a regular way, can be used as the basis for complexity metrics for the induced language. For example, in a system for novices, the designer may be forced to restrict commands so that no more than three productions are necessary or that no more than two operands are permitted. Comparative metrics would be useful in assessing the relative merits of two proposed command languages. Simplicity might be measured by merely counting the number of productions necessary to describe the language.

The basic multiparty grammar concept describes syntactic features of an interactive dialogue, but extensions could be made to include more semantic features. A guard expression containing Boolean qualifiers might be used to indicate when a particular production is invoked. Integer counters might be initialized, incremented, and tested to control the number of times a particular production was applied. This technique would be used to limit the number of sign-on attempts that were permitted or the number of responses a user could give in a computer-assisted instruction application.

III. Defining Visual Features

The multiparty grammar described thus far is limited to hardcopy line-by-line types of interaction. To accommodate soft copy displays, graphics, lightpen input, touchscreens, sonic pens, special programmed function keys, joysticks, knobs, buttons, etc., we must include additional tools. These tools are not extensions of the multiparty grammar metanotations, but employ the grammar description approach to specify features common to interactive systems.

It seems straightforward to include some features of line-oriented cathode ray tube or plasma displays which merely change the visual presentation of characters, such as underlining, reversal, blinking, fonts, typesize, intensity, or color. By including a special marker, these visual features can be specified. This approach is similar to commands included in text editors or document processors. For example, to underline a portion of output the following production could be used:

⟨C: RESPONSE⟩ : : = ⟨C: FIRSTPART⟩⟨C: UNDERLINE⟩
⟨C: IMPORTANTPART⟩⟨C: OFFUNDERLINE⟩.

The ⟨C: UNDERLINE⟩ nonterminal indicates that the next set of characters should be underlined until a ⟨C: OFFUNDERLINE⟩ is encountered. This approach can be applied to at least the following features:

1) underline	UNDERLINE, OFFUNDERLINE underlines text
2) reversal	REVERSE, OFF REVERSE switches from white on black to black on white

3) blinking	BLINKING(⟨RATE⟩) rate is specfied in blinks per second OFFBLINKING, equal to BLINKING(0)
4) font	FONT(⟨typefont name⟩),
5) typesize	TYPESIZE(⟨size⟩),
6) intensity	INTENSITY(⟨level⟩),
7) colors	COLOR(⟨color choice⟩).

It is possible to switch to underlining, a different typefont, and a higher intensity by making three successive specifications. The permissible features depend on the capabilities of the hardware and software systems. These *static feature nonterminals* may be invoked by any party in the interaction. For humans at a keyboard, there may be a special character to specify underlining, a toggle switch to indicate reversal or a knob for intensity; for the computer, signals may be sent to the terminal to invoke these static features or the nonterminal may expand to a null character. In any case, instructions must be written by the system developer to execute the visual feature change.

IV. Window Declarations for Screens

Designers have found it convenient to design interactive systems so that the screen is divided into several work areas called windows. Typically, status information indicating the time of day, system availability, program name, or frame number appears on the top. The center portion of the screen may have one or two work windows with a bottom window listing available commands or options. The number, shape, and size of windows should be alterable by human commands or computer program instructions.

If we restrict ourselves to character-oriented screens with a fixed number of horizontal and vertical positions, then windows can be specified by grid points. The upper left position is position (1, 1), and the lower right position is (R, C) where R is the number of rows and C is the number of columns. For example, a screen might be separated into three windows: a top window showing status information, a middle window providing a user workspace, and a bottom window listing the available commands. Users should be able to describe such a screen layout by indicating the rectangular shaped windows as a two-dimensional array of characters and the name of the starting symbol which defines the contents of the window. The declaration might be part of a screen definition module which is part of the interactive system development package:

```
DECLARE SCREEN CONTAINING
    (WINDOW STATUS (1:4, 1:72) WITH STATUS-INFO,
    WINDOW WORKSPACE (5:34, 1:72) WITH WORK-INFO,
    WINDOW AVAILABLE (35:40, 1:72) WITH AVAIL-INFO).
```

The row declarations in this simple case do not overlap and do cover the full 40 row and 72 column screen. Extensions to this simple layout might include the following.

1) *Coverage of Less than the Full Screen*—The undefined areas are left blank.

2) *Coverage of More than Full Screen*—The defined areas that are off screen are maintained but not shown. This allows using sliding window techniques in which only a small portion of a large data area is visible at one time.

3) *Overlapped Windows*—The window declaration order indicates which windows are on top of previously defined windows.

4) *Nested Window Declarations*—A window may contain several windows. For example, the workspace window may be broken down into several windows for errors or future status information specific to a particular application. The declaration might be as follows:

```
DECLARE SCREEN CONTAINING
 (WINDOW STATUS (1:4, 1:72) WITH STATUS-INFO,
 WINDOW WORKSPACE (5:34, 1:72) CONTAINING,
   (WINDOW TEXTSPACE (5:27, 1:72) WITH WORDPROCESS,
   WINDOW ABBREVIATIONLIST (28:29, 1:72) WITH ABBREVS,
   WINDOW ERRORS (30:34, 1:72) WITH DIAGNOSTICS),
WINDOW AVAILABLE (35:40, 1:72) WITH AVAIL-INFO).
```

5) *Multiple Screen Layouts*—Several screens may be defined for applications such as utility plant monitoring or air traffic control, but the information on each screen is independent. In other applications such as teleconferencing, cooperative program debugging, game playing, psychological experimentation, commodity exchange or stock market systems, and newspaper editing, parts of a screen display may be shared by several screens. This can be arranged by referencing the same starting symbol. For example, in a commodity exchange system, each viewer may have a common window with the current trades, a common window with the latest price for each commodity, a personal window showing current holdings, and a personal window for a work area to make computations in planning future purchases and sales.

V. Dynamic Operations on Screens

One of the exciting advantages of visual display screens is that the information on the screen can be dynamically modified. In text editing it becomes possible to make insertions and have the text move on the screen to accommodate the additional words. In data entry, prompting patterns such as "READY FOR DATE: MM. DD. YYYY" can guide the user and then be overwritten as data is entered: "READY FOR DATE: 03.24.1973." In command languages, the user may move a cursor back a few lines to repair an incorrect command without having to retype the entire command. In commercial practice, familiar forms may be shown on a screen and the user simply moves a cursor to the proper position to make an entry or replace a previous entry.

The key to dynamic operations is the capacity to move a cursor to specify the placement of the new entries. Cursor movement can be specified in several ways:

1) by a cursor placement device such as a lightpen, sonicpen, touchscreen or rotating click wheels;
2) by a cursor movement device such as four keys with

directional arrows (↑→↓←), a mouse, joystick, or rotating track ball;
3) by specifying the absolute physical address within the screen, for example, row 12, column 49;
4) by specifying the relative screen address within the window, for example row 3, column 19 within the WORKSPACE window;
5) by giving a command to move the cursor horizontally or vertically, for example, UP 1, or LEFT 3.

Methods 1 and 2 are generally easiest for humans, but the output of these special purpose devices must be converted into the absolute screen address for machine processing. Method 3 ignores the user's perception of windows. Methods 4 and 5 can both be implemented, or to keep matters simple, one can be implemented in terms of the other. A cursor, currently at row 3, column 1, could be moved to row 1, column 6, of the window with the starting symbol WORKINFO by the command

a) WORKINFO CURSOR TO (1, 6)

or by the command

b) WORKINFO CURSOR UP 2 LEFT 5

or by hitting the following keys:

c) ↑ ↑←←←←←

or by using a special purpose device such as the lightpen.

In all these forms the cursor address must be transformed to indicate the absolute physical screen address of line 5 column 1. This transformation must be done by the interactive control program handling the productions that describe the movement operations. Those operations could be written for each of the above three examples:

a) ⟨H: CURSOR MOVE⟩ : : =
⟨H: STARTING SYMBOL NAME⟩
CURSOR TO (⟨H: INTEGER⟩, ⟨H: INTEGER⟩)

b) ⟨H: CURSOR MOVE⟩ : : =
⟨H: STARTING SYMBOL NAME⟩
CURSOR ⟨H: VERTICAL⟩ ⟨H: HORIZONTAL⟩
⟨H: VERTICAL⟩ : : =
UP ⟨H: INTEGER⟩ | DOWN ⟨H: INTEGER⟩
⟨H: HORIZONTAL⟩ : : =
LEFT ⟨H: INTEGER⟩ | RIGHT ⟨H: INTEGER⟩

c) ⟨H: CURSOR MOVE⟩ : : =
⟨H: SINGLE MOVE⟩ |
⟨H: SINGLE MOVE⟩ ⟨H: CURSOR MOVE⟩
⟨H: SINGLE MOVE⟩ : : = ↑| ↓ |→| ← .

Once the cursor is in place, then the normal rules of interactive systems apply until a new cursor movement instruction is given. Old text is replaced by the new text or eliminated by blanks.

The productions indicating human initiated cursor movement are only one part of the specification of the interaction. The cursor may be moved by the interactive program specification, for example in an interactive customer order entry application, the system may automatically move the cursor from one part of the form to another as data items are entered.

Another problem that must be dealt with is illegal cursor movements. The user may inadvertently try to move the cursor outside its window. An error message could be shown, the cursor could be stopped at the border of the window, the cursor could "wrap around," emerging at the opposite side of the window, or the cursor could revert to the upper left-hand corner of the window. Application specific conditions on cursor movement might be included; for example, in text editing an attempt to move beyond the right side of the window might move the cursor to the beginning of the next line of text.

VI. Summary

The major contribution of this paper is to introduce the notion of multiparty grammars for describing the actions of several participants in a dialogue. Multiparty grammars have labeled nonterminals to indicate the source of the string. Nonterminals acquire values which can be referenced by any party. Since error handling is such a critical component of interactive systems design, a nonterminal which matches an unparsable string is introduced.

Secondary notions which permit a more complete description of an interactive terminal session are also introduced. Common visual features such as underlining, reversal, or color were accommodated with a set of BNF productions. Window definitions for screen-oriented systems were made in a simple declaration statement. Cursor movements which are necessary to describe dynamic screen features were handled with another set of BNF productions.

The use of multiparty grammars and these secondary features for describing interaction is especially appealing if a compiler–compiler is available to provide partial verification of consistency and completeness, and to facilitate the writing of procedural language statements to perform required operations. Multiparty grammars may also facilitate standardization, preparation of comparative metrics of simplicity, documentation, generation of help facilities and verification of correctness. The reader should be cautioned that this is a proposal which must be implemented, refined through experience, and tested in a controlled environment to determine its effectiveness.

Extending these ideas to include special hardware or full graphics is a natural next step. Grammars to describe pictures have been investigated, and these should be studied to expand the generality of this approach. Additional grammar features to describe specific application environments such as line-oriented text editing may substantially reduce the implementor's task by simplifying or eliminating the need to write procedural language statements to handle each production. A text editor generating system seems feasible.

Acknowledgment

The author wishes to thank M. Brodie, J. Gannon, R. Hamlet, G. Nagy, J. Sowa, and G. Thomas for sugges-

tions and encouragement after reading versions of this paper. The referees provided additional references and the idea of adding a colon to the multiparty grammar notation.

REFERENCES

[1] J. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference," in *Information Processing*. Paris, France: UNESCO, 1960, pp. 125–132.

[2] G. V. Bochmann, "Finite state description of communication protocols," *Comput. Networks*, vol. 2, pp. 361–372, Oct. 1978.

[3] E. F. Codd, "HOW ABOUT RECENTLY? (English dialog with relational databases using RENDEZVOUS Version 1)," in *Databases: Improving Usability and Responsiveness*, B. Shneiderman, Ed. New York: Academic, 1978, pp. 3–28.

[4] A. Colmerauer, "Metamorphosis grammars," in *Natural Language Communication with Computers*, L. Bolc, Ed. Berlin, Germany: Springer-Verlag, 1978, pp. 133–189.

[5] E. Denert, "Specifications and design of database systems with state diagrams," in *Proc. Int. Computing Symp. 1977*. Amsterdam, The Netherlands: North-Holland, 1977, pp. 417–424.

[6] S. L. Feingold, "PLANIT—A flexible language designed for computer-human interaction," in *Proc. Fall Joint Computer Conf. 1967*. Montvale, NJ: AFIPS Press, 1967, pp. 545–552.

[7] S. Feyock, "Transition diagram-based CAI/HELP systems," *Int. J. Man-Machine Studies*, vol. 9, pp. 399–413, 1977.

[8] D. Gries and J. M. Lafuente, "Language facilities for programming user-computer dialogues," *IBM J. Res. Devel.*, vol. 22, pp. 145–158, Mar. 1978.

[9] P. Hanau and D. R. Lenorovitz, "Prototyping and simulation tools for user/computer dialogue design," in *Proc. ACM SIGGRAPH '80 Conf.*, July 1980, pp. 271–278.

[10] G. E. Heidorn, "Automatic programming through natural language dialogue: A survey," *IBM J. Res. Devel.*, vol. 20, pp. 302–313, July 1976.

[11] K. Jensen and N. Wirth, *PASCAL User Manual and Report*, 2nd ed. New York: Springer-Verlag, 1975.

[12] S. C. Johnson, "YACC—Yet another compiler-compiler," Bell Lab. Tech. Rep., Murray Hill, NJ, 1977.

[13] W. C. Mann, J. A. Moore, and J. A. Levin, "A comprehension model for human dialogue," in *Proc. 5th Int. Joint Conf. Artificial Intelligence*, 1977.

[14] W. McKeeman, J. J. Horning, and D. Wortman, *XPL: A Compiler Generator*. Englewood Cliffs, NJ: Prentice-Hall, 1970.

[15] T. P. Moran, "Introduction to the command language grammar: A representation for the user interface of interactive computer systems," XEROX PARC Rep. SSL-78-3, Palo Alto, CA, Oct. 1978.

[16] D. L. Parnas, "On the use of transition diagrams in the design of a user interface for an interactive computer system," in *Proc. 24th Nat. ACM Conf.* New York: ACM, 1969, pp. 379–385.

[17] J. L. Peterson, "Petri nets," *ACM Computing Surveys*, vol. 9, pp. 223–252, Sept. 1977.

[18] P. Reisner, "Using a formal grammar in human factors design of an interactive graphics system," IBM Res. Rep. RJ2505, San Jose, CA, Apr. 11, 1979.

[19] J. Sammet, "A Definition of the COBOL 61 Procedure Division Using ALGOL 60 Metalinguistics," *Proc. 16th Nat. Conf. ACM*, New York, 1961.

[20] B. A. Sherwood, *The TUTOR Language*. Urbana, IL, Computer-Based Education Lab. Univ. of Illinois, June 1974.

[21] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop, 1980.

[22] _____, "Human factors issues in designing interactive systems," *IEEE Comput.*, vol. C-12, pp. 9–19, Dec. 1979.

[23] C. A. Sunshine, "Survey of protocol definition and verification techniques," *Comput. Networks*, vol. 2, pp. 346–350, Oct. 1978.

[24] _____, "Formal techniques for protocol specification and verification," *IEEE Comput.*, vol. C-12, pp. 20–27, Sept. 1979.

[25] D. Waltz, "An English language question answering system for a large relational database," *Commun. ACM*, vol. 21, pp. 526–539, July 1978.

[26] A. I. Wasserman and S. K. Stinson, "A specification method for interactive information systems," in *Proc. Conference Specifications of Reliable Software*, IEEE Cat. 79 CH 1401-9C.

[27] J. Weizenbaum, "ELIZA—A computer program for the study of natural language communication between man and machine," *Commun. ACM*, vol. 9, pp. 36–45, Jan. 1966.