This interface will normally then prompt the user to enter a statement at his terminal. If he types a statement without specifying that it should be saved as a line in a program, then the system will translate and execute that statement; otherwise, it is saved as a line in some program as specified. By entering a direct statement which calls a saved program, the user gives control to it. It may later call USER directly or indirectly (such as the occurrence of an error). Other parts of the system also may interface with the user: examples are data requests by the program, program output, etc.

This control is maintained by certain actions of the IPS. USER is known to the system components, and they will call it in order to handle errors, special conditions, or in response to an

are other data structures in an interactive system which users could make use of within their programs.

An immediate candidate is the symbol table, which associates names with objects (variables, programs, etc.) in the IPS. The ability to access and sequence through this structure would allow the user to display the names of all variables; ask whether a given variable had already been used by him; or, using the symbol table and the program text, create an index of all the statements using a given variable, for instance.

Many of these features could replace those already available in JOSS, APL, and LC^2, and some, such as creating an index of the uses of a variable, or simply listing the names and number of lines in individual functions are not even available in those systems. Hence, users could augment the available interface and context facilities of the systems by their own programs, thus tailoring the systems to their individual tastes and needs.

2C The Concept of the User as a Function

One further mechanism can allow this molding of the system/user interface to be completely under user control. That is to extend the notion of the user's representative to the system as suggested by the PART 0 concept in LC^2 to be a procedure (we will call it USER) which is expressible in the language of the IPS, and which is considered as the interactive control manager of the system.

Hence, USER is the face of an "execution facility" and acts as the agent (though not necessarily the only one) which allows the human user to enter statements which can be saved or directly executed by the IPS. Thus, when a user logs into or begins a dialogue with the IPS, some initial interface will "get" his version of USER — which may, of course, be some standard one — and call it, making it the controlling program for the system.

2. Design Considerations for an IPS

Now that we have given a brief survey of some interactive systems and some criteria for judging them, we will investigate further the implications of both the history and the criteria for designing and implementing an IPS.

2A Direct and Stored Statements

The notion, due to JOSS, that each statement of the language should be able to be used both directly and indirectly (i.e., stored as a line in a program) gives rise to a number of consequences. Some of these have already been mentioned but are repeated here for completeness.

(1) Since the language must contain statements for modifying program text, therefore, program text is a data structure of the language.

(2) Given that program text is data, it is desirable to make available those operations which the system possesses for treating it. Among these are the following:

2(a) to translate a statement as if it had been entered by the user at his terminal, executing it if direct or storing it if indirect;

2(b) to sequence through the data structure in the lexical order (by line or step number) of the program; and

2(c) to access the text of a given statement as data (probably a string) which can be stored as the value of a variable.

An interesting side effect of 2(a) is that any parts of the language which only allowed constant information in certain places can now have variable information in those places, since one can create a string - from the values of variables - which may then be executed.

2B Accessing Other Data Structures

Once we have allowed access to the system data structure defining the text of programs, one is led to inquire whether there

user some indication that this is happening in LC^2, the system, which normally prompts the user for a line by spacing the typing element a few places, increases this spacing at each nested level of PART 0 -- thus as the user nests more conversations, the system gives a longer prompting string whenever a line is to be entered by the user.

BASIC: simple and easy to use, but restrictive.

JOSS: very good

APL: very good.

the part in which a declaration occurred pops the top value from the stack for that variable. This means that any part which uses a variable, A, for instance, but which executes no declaration for A will use the same incarnation of A as its caller did. The base of this inductive definition is supplied by the system's always providing a value for each variable at the outermost level.

BASIC: poor JOSS: fair APL: excellent

Human Interfacing:

By considering the user as a program, PART 0, in the system, and by making it possible to express the actions of PART 0 in LC^2 itself, the system allows users to make its interface behave in any number of ways. Unfortunately, this feature was not well-developed in the system, or better string replacement operators might have been available to do things such as macro expansion of program text or even some parsing at that interface to achieve some language extensibility. Then a user could readily tailor the language -- insofar as he could see it through that interface -- to suit his needs and abilities.

The only use made of the typewriter terminal which is not already a feature of APL was prompted by the ease with which one can unknowingly and incorrectly generate many nested conversations in APL. When an error occurs in APL, one is often tempted simply to fix it and then re-call the function in which it occurred. Because of the ability to nest suspended executions in APL, this procedure will cause a large nesting of function calls if it is repeated on each error, without deleting the previous execution. In order to give the

Response Suitability:

LC^2 runs under a general time-sharing system, TSS/360, and has little control over the scheduling within the system, although it normally gives good response. Its main drawback is that it is interpretive and is therefore unsuitable for programs requiring large amounts of computer time or efficient execution.

BASIC: good

JOSS: excellent APL: excellent

Language Power:

LC^2, like Algol, only has operations which deal with scalar operands, except for the case of program text, which can be operated upon in groups of statements. It could benefit greatly from APL's operator philosophy applied to its other data forms. Some additive, Algol-like features of note are the inclusion of strings as a data type throughout the language, reference variables, and the ability to pass parametrizable expressions as actual parameters to a function -- Algol's call-by-name mechanism can be viewed as the ability to pass unparametrizable expressions to functions.

LC^2 added locally declared variables to an interactive environment by (1) making all declarations executable statements which can only have an effect if actually executed in the course of some part; and (2) making each variable a stack and stating that any use of a variable means that the value on the top of its stack is to be used; a declaration places a new value on a variable's stack (unless it is a redeclaration at the same level as the previous one), and leaving

BASIC: poor

JOSS: fair

APL: good

Assumed Context:

While LC^2 does allow a user to save objects (parts, variables and values) on files, it restricts him from saving control information which would allow the type of long term storage of "active" program available in APL. Another feature relating to context is the ability to display type, structure and declaration-level information about variables at the user's terminal.

The ability to treat programs very much in the manner of normal data is a feature of LC^2 not present in the other systems and does add a useful sector to the universe of discourse available to the human.

Also of interest is the ability to execute incomplete programs in LC^2, adding parts as required when programs attempt to use them. This is made possible by a control statement, RECOVER, which instructs the system to retry an operation which caused an error. In this way, a program can proceed after an error involving, for instance, an undefined variable or program part, once it has been defined, as if the error had never occurred. The system can thus prompt the user to define parts of his program as they are needed in the development of the program -- an automated critical path method for system building.

BASIC: fair

JOSS: good APL: excellent

This gives one control in both the forward, or calling direction, and the reverse, or exiting direction. Also, changes are allowed to any statements except those pending a function return which is not a return from PART 0.

BASIC: the user must follow a lock-step cycle: edit-run-edit-etc.

JOSS: very good; first system to allow interleaving of execution and alterations.

APL: good.

Plasticity:

Not only can programs be accessed as in APL, they can be accessed and changed from within programs (all statements may be used in both direct and stored mode). Operations exist to fetch the text of a statement or group of statements as a string value which can then be stored in a variable and operated upon. The reverse operation of making a string a statement in a program is also provided and acts just as if that string had been typed by the user at his terminal: if it is an immediate (direct) statement, it is translated and executed; if it is a statement which is to become a step in a part, then it is translated and stored into that part. This allows existing text to be changed by a program and also makes it possible to create program text for execution or to be saved.

Although it allows fairly general, heterogeneous data structures, it is impossible for a program to get and save information about any structures, a regrettable deficiency. Its correction could well use APL as a stepping off point for describing data structures which are descriptions of other data structures.

availability of explicit operations, normally part of the system, for use on programs (such as translation and execution);

(4) a view of declarations for variables as dynamic instead of dependent on the lexical structure of the program;

(5) control over the execution of the program; and

(6) the use of typeless variables, variables not requiring a declaration, but able to take on values of different types.

These departures and other points to be mentioned we will comment on within the scope of evaluating LC^2.

Interactive Control

LC^2's basic control is very JOSS-like, at least in terms of execution/alteration interleaving. One significant point on which they differ is that LC^2 considers the user to have a program called PART 0 (parts are the basic program groupings used in LC^2) which acts as his representative to the system. As a part, PART 0 is call-able and recursively usable. Indeed, whenever the system needs to make contact with the user, it does so either by calling PART 0 anew or returning to an already existing incarnation of it if the statement under execution was direct. Moreover, as we shall see, PART 0 is expressible within the LC^2 language.

The LC^2 user also has control over the flow of control in his programs. Not only can he start executions at any time, including the period when a previous execution has called PART 0, thereby giving him control; but he can also close (partially or entirely) previous executions and conversations, since PART 0 acts like any other program.

1D The Interactive Criteria Applied to LC^2

We will now give a sketch of LC^2 (Language for Conversational Computing) [MPV 68], a system which has prompted much of this research. The outline will be divided and described under the interactive just stated. We will "rate" three other systems, BASIC, JOSS and APL under those headings also.

1D1 LC^2: Language for Conversational Computing

The LC^2 research [MPV 68, LM 69, VZ 69] was originally conceived as the task of creating a conversational Algol system using many ideas from JOSS as a starting point for the interactive qualities desired. However, this was soon abandoned because it was thought that the total bindings inherent in Algol were too much in opposition with the notion of flexibility in an interactive environment.

The result was an Algol-like language with some extra syntactic sugar and semantic extensions in terms of data types and actual parameters to procedures. The most radical departures from Algol were

(1) the ability to execute incompletely specified programs and, in fact, intermingle development and initial debugging and programming;

(2) the ability to execute parts of programs under different environments and parametrizations as an aid in program development;

(3) the ability to modify, copy, and delete program text (and its executable representation at the same time) with the same ease as corresponding operations on more conventional data, plus the

1C4 Some Criteria for Interactive Programming Systems

The above two sections lead us to compile the following list of criteria or design principles for an IPS. We will use these to evaluate some existing systems and, in the next chapter to justify and prompt some features to be developed for an IPS.

Interactive-Control: the ability of the user to initiate, interrupt, and generally interject himself into the control of the system.

Plasticity: the extent to which a system is changeable, especially in its appearances and behavior to the user. Directly associated with this is the extent to which the system is self-referenceable: i.e., the amount of accessibility of programs and system data from within the language of the IPS itself.

Assumed-Context: the amount and duration of the universe of discourse which is available when conversing with the IPS. Examples of this range from the ability to use typeless variables to being able to stop and restart programs with an intervening delay of days or months.

Language-Power: this criterion is not specific to interaction but is certainly important in the utility of any programming system.

Human-Interfacing: Although unmotivated by the preceding batch/ interactive comparisons, there is one further criterion which must be applied to any interactive system (whether used for programming or driving a car): namely, the smoothness of the interactive controls and their suitability for human use. This encompasses the face of the system which the user sees and the ease with which he can manipulate and roam around in the information space which is of interest to him when he wishes.

These criteria along with response-suitability discussed earlier, will be used to evaluate some existing interactive programming systems. The criteria are not completely pair-wise orthogonal, and hence some features of the systems to be reviewed, while treated under only one of these headings, may also lie in the realm of one or more of the others. The four systems to be analyzed are, in roughly chronological order, Dartmouth's BASIC, RAND's JOSS, Iverson's APL\360, and Carnegie-Mellon's LC^2.

This can be viewed as the imposition of symmetry into an asymmetrical control structure, and is extremely important for debugging and development of programs by preventing one from getting into action cul-de-sacs in which nothing can be done. Other mechanisms of control such as goto's and loops are essentially idempotent with respect to symmetry; so also is coroutine control of which more will be discussed shortly.

1C3D System Commands

Since programs are to be viewed as data, the statements available for manipulating them, once a prime concern of control languages, become part of the language itself. One consequence of this is that the execution of programs may be interleaved with changes to those same programs, either by other programs or the user (which amounts to the same thing since it is really a program which does the alterations for him). Above, the user was mentioned as an active control element in an IPS. Therefore, what programs can do so may he: call other programs, execute statements, be called from a program, etc. He is a function in the system, of great flexibility, unpredictable, but without his inclusion in the control, the word "interactive" becomes inapplicable. In some sense, the amount of interactive control of an IPS is proportional to the number of places at which the user may interpose himself into the operations of the system. This interleaving of execution and alteration is of central concern to the bulk of the rest of this thesis.

1C3B Typeless Variables

JOSS [Ba66] was one of the earliest interactive systems to allow so-called typeless variables. It is not that such variables have no type but rather that their type is dependent only on the value which they represent at a given moment -- their type varies when their associated values change in type or form. One reason for their existence is that they allow the user to dispense with certain unnecessary overhead such as the declarations of Algol. After all, the user knows what is meant by a given variable, and that is what really matters. What is more, since variables may be treated differently as a program is developed, not having to declare them has obvious advantages for the user. However, it is crucial that one be able to find out what the current semantics of a variable are at any time, in order to keep track of variables, information being a prerequisite for control.

Typeless variables have been one of the main reasons that JOSS, LC^2 [MPV 68] and APL\360 [IF 68] are implemented as interpretive systems. If we are ever to counteract this variability at the system level in order to obtain efficient implementation, we have to find a means of measuring and recording the changes in type which a given variable incurs.

1C3C Control Symmetry

Putting the user into the control of the system as an active control agent imposes some design considerations on the mechanisms used for control. He must have a means of unwinding or killing part or all of an execution in order to retry certain parts of the program without necessarily completely restarting the execution. That is, he needs a means of reestablishing some previous context and control state from another, later one. One possible way of implementing this is to have a means of closing or exiting from subroutine or function calls, possibly returning an error indication to those routines being closed should any of them need to perform some clean-up before exiting to their caller in turn.

1C3A Programs as Data

Perlis [MPV 68] says:

> "The program text, being under direct execution, is understood to be 'immediately' subject to change and hence the system statements, known as editing statements, are part of the language. Put another way, the text is data of the language."

Indeed, the object code of a program has always been partly accessible (though not necessarily changeable) -- that is what control instructions do. However, the notion that the program is a changeable, fluid object is -- aside from languages such as LISP and IPL-V -- peculiar to interactive programming. Perlis's statement may be changed to read "put another way, the programs are data of the language." Since a program may have more than one representation, each may be considered more or less accessible by the user. In fact, we hope to make the point that any of the data structures in an IPS which are potentially useful to the user ought to be accessible to him (and therefore to his programs).

in order to minimize the frustation level of the user. This last exhortation is known as human engineering, or egonometrics in fields such as the design of airplane control systems, automobile design,etc.

1C2E Types of Program Control

The main control mechanisms in batch system and languages designed for the batch environment have been simple, such as subroutine calls and returns, and coroutine control in a few systems. Some recent multiprogramming systems have also allowed the parallel execution and synchronization of independent processes. These controls are fixed in the sense that the structure of the source programs describes the execution behavior of those programs. Making control variable then means allowing control paths to be established and used which were not pre-defined in the program. The ability to exit from some part of a sequence of nested calls is a feature which is very useful in an interactive situation but would be difficult to make good use of in a batch environment. Even ignoring its relative utility, it is a feature for which there is little motivation in a non-interactive world, but whose absence is noticed in a conversational system.

1C3 The Implications of Variability in an IPS

The sketch just given of a comparison of batch and interactive systems has some interesting implications for the designers of an IPS. As well as trade-off decisions about generality versus the ability to implement and so on, there are some issues which can be directly dealt with.

for instance) of a variable can be considered attributes of the variable. In batch systems the attributes of a variable whose values are needed in order to use the variable in a program must either be declared fully, as in Algol, or be deducible, as in Fortran. This is partly so because the user is not available to supply such bindings if they are needed. Allowing such attributes to be variable (as opposed to unspecified) has more meaning in an environment in which the user, because he is creating and debugging programs on-line, may decide to change those attributes and the way in which he uses certain variables as the program develops.

1C2D User/System Interface

The face of a batch system seen by a user usually consists of a highly specialized language called a control language, whose statements are interpreted by the system to determine what functions,such as compiling, loading or executing, are to be performed in a given job. Allowing this to become variable can be done in two senses. The first is in making the control language truly a programming language with the control mechanisms and program structure of such languages. The second involves changing that language to allow it to be altered by the user to suit himself. In a batch system, control statements are of secondary importance to the task of running programs; in an interactive system, however, they become almost paramount, because of the change in their frequency of use and because of the close association of the user and the system across that interface -- when a facility is used a great deal, it is important that it be smooth and functional

1C2A Program Text

Decks of cards have been a way of life in computing since (and in fact before) the genesis of electronic digital computers. Any changes to programs or data in card oriented systems thus involves insertion, deletion, or replacement within card decks and is normally done off-line by the user with a keypunch machine. Those handling larger programs and files of data generally have had the ability to perform exactly these same functions with a computer and magnetic tapes. Once the text of a program becomes accessible as a file within an interactive system, however, the card-image view of the world breaks down. Contextual modification of files and program-directed file searches become a part of the normal operations derived from the handling of card decks.

1C2B Program Representation

Programs in a batch system can normally be considered to be in one of a small number of states: source, object deck, or loaded in memory. The transitions between these states are known as compiling and loading. When a program is accessible as a file or data structure, these distinctions become unimportant and the user may view his source text as the only representation of his programs. Hence, the changing of a line of program text should also result automatically in changes in the "machine code" for that program.

1C2C Attributes of Variables

The type (integer, real, etc.) and the structure (array [1:10]

we also have an argument for allowing a single user to own more than one operating process which he can link up to or detach to run separately, as desired.

We prefer to replace Simon's and Miller's considerations by the following:

> Response suitability: the responses to the user should be within acceptable time limits for the type of task performed (this includes (1) and (2) above), and the more tasks which can be called trivial (i.e., give immediate, two second response), the better.

This is the first of a number of criteria which we will develop for interactive programming systems. Much can be discovered about conversational programming by attempting to understand what differentiates an IPS from a batch oriented system. This will in turn lead to some more criteria similar to that above.

1C2 From Batch to Interactive Systems

Most of the differences between interaction and batch processing are of the form: make variable something which was usually considered as fixed in batch systems (see, for instance [Ba 66] and [MPV 68]). The introduction of the human into the system loop is the main reason for the introduction of such variability. If such variability is not available, the main advantage of using a terminal system is the time saved in walking to the computer to have a program run. Therefore, we will examine a number of the quantities (real and abstract) which are manipulated in both batch and interactive systems, with a view to investigating the consequences of making variable what formerly was not.

full duplex terminal. Almost no delay can be allowed between depressing a key and the corresponding character's being printed (0.1 to 0.2 seconds) due to the computer's echoing it. When a line is completed, however, he will accept a much larger wait (about 2 seconds) until the system prompts for the next line, even for the most trivial of tasks.

Simon's second design principle essentially defines a maximal wait time (he estimates about ten minutes) beyond which the user could switch tasks and use the time to advantage. The ideal, of course, would be for the system to tell him when a task will take that long and possibly even allow him to ask for the answer when he is ready for it rather than have the system interrupt him in the middle of another task. If the other task requires using the system, then

a conversation is not well balanced. The first is that the more powerful conversant will simply remain in the situation, utilizing his time poorly. The second disfunction is the result of information not arriving at an appropriate rate or in an appropriate form. In this case, a conversant may process that information inefficiently; eg., displaying numbers in octal could create such a situation when the user is only able to add in decimal and must perforce perform conversions on the output in order to do simple calculations.

As a result, Simon proposes the following design principles for conversational systems:

(1) "neither the human nor the machine component of the system [should] respond at rates either above or very much below their processing capacity";

(2) "the human, like the computer, has minimum swap times† which we can estimate roughly".

The first principle is a direct result of his statements about a balanced conversation. A similar point is made by Miller [Mi 68]: response times are important when the task being done is a transition to a goal of some sort. The concept of "goal" in this context is really a subjective sense of completion called "closure" by psychologists; the rule is that more extended delays are acceptable after a closure than in the process of obtaining a closure. A good example of this occurs in the echoing of characters typed by a user using a

---

† The human swap time is the time it takes to stop one task and "get up to speed" on another.

(1) any time-sharing system will become saturated by its community of users; and

(2) the definition of what is a trivial task depends solely on whether or not the system can perform it and give immediate response to the user.

It must be pointed out that (1) does not mean that research into the development of more efficient and powerful time-sharing systems is futile: on the contrary, it is the carrot held before all who work in the field. Moreover, such research is the primary way that a man over computer imbalance stress is nullified. Similarly, (2) implies that a good way to make better use of human "nerve-ware" is to find better and faster ways to perform the tasks they request. In this way more and more tasks will enter the class of trivial tasks thus allowing the human to work at an ever higher level. The building of theoretical systems in mathematics by proving theorems using previously proven statements instead of always invoking a set of axioms and inference rules is an analog of this process. Indeed, the prime motivation for this thesis has been the fact that a large number of tasks which ought to be trivial in interactive programming (by this author's judgement as a user) are not, and could be made so by research into the properties of execution in a conversational environment.

1C1 Human Psychology and Interactive Programming

We have spoken several times of "immediate response." A good definition of this term has been worked on by very few people. One notable paper by Miller [Mi 68] and another by Simon [Si 66] have dealt with this problem. We will not delve into Miller's results here, except to mention that he has classified a large number (17 to be exact) of interaction situations and given time bound estimates in those circumstances. On the other hand, Simon's paper really deals with relative time: it was his description of a balanced system as one in which the two conversants "are approximately matched intellectually and are taking symmetric roles in the conversation" which prompted the preceding analysis of processing power imbalance.

Simon mentions two possible disfunctions which can result if

complement the other's inefficiency. That is, it will be used for more and conceivably larger tasks in order to utilize its wasted potential. And, to the extent that it is used for more by the weaker element, the more the weaker's definition of trivial will expand to include those tasks. This will ultimately saturate the more powerful processor; in the case of a computer, this has resulted (historically) in the development of bigger and faster computers. So far, men's need for information has been the prime stress keeping this cycle going. The human's power, then, can be measured in terms of his ability to propose more complex tasks whenever an idle computer can be found. The computer's power takes the form of performing those tasks in less and less time. In any case, there are two main consequences of this argument:

1C Considerations and Criteria for an IPS

An IPS can be thought of as a small, closed society of man and system, with mutual feedback. Because of the problems in developing time-sharing systems and the cost of hardware, the user's needs have only been met insofar as they have not adversely affected the computer's efficiency. Simon's statement which we quoted in section 1B implies that some of the initial impetus for time-sharing was that the user was sitting idle a great deal of the time when using a batch system. The concern over idle time is clearly symmetrical, and whenever there is an imbalance in the man-computer system, the system can be expected to move (or be moved) in such a way as to nullify that imbalance. This statement is reminiscent of LeChatellier's Principle in chemistry. This imbalance is not one of processing power pre se; rather, it is concerned with the complexity of the tasks which the man can request the computer to perform, relative to the computer's processing ability. The converse, of course, is also true (as witness the amount of work which may be initiated by the user in response to an error message from the system).

It is a generally held belief that interactive systems should give "immediate" response to trivial requests [We will delve further into the meaning of "immediate" shortly]. The crux of the matter, however, is the definition of a trivial task; for it is clear from the analog to LeChatellier's Principle above that if either the human or the computer element of an IPS is idle, more processing load will be placed on the more powerful processor in order to com-

Interactive systems do not, however, "guarantee" good programming, higher programmer productivity, or even less frustation in using a computer. Simon [Si 69] has said that a computer is a very powerful tool; and one quality of tools is that they are not goal-specific. It follows that one can use an interactive system to disadvantage (in many ways!) Moreover, instead of relieving the early computer user from having having to push buttons and throw switches, we have simply altered the protocol: now he must type commands to the system in order to get it to perform for him.

Of course, these commands may be much more powerful in their actions than were the buttons on computer consoles. And, if the user could alter his protocol, or interface, with the system, hopefully that is better than requiring him to follow a standard format of interaction. We will develop these notions more fully below.

Control over the interaction has also advanced: many systems include the user in the interaction whenever an error occurs; he presses an interrupt button at his remote terminal or the program which he has written requests his intervention. This flexibility in man-machine interaction has benefitted a great deal from the pioneer effort of JOSS. It was the first system which allowed one to edit, add, or delete programs within the framework of the conversational language. In JOSS, one could also suspend program execution and examine the state of control of the program and then change it to suit one's needs. This control over execution of a program has added a great deal of flexibility to the task of creating and debugging programs. The user need not completely restart a computation because he forgot to initialize a variable for instance, but can assign a value to it when the system discovers the value is missing, and then tell the system to continue from where the error occurred. If such an error occurs in most noninteractive systems, either the error is not detected, in which case the value of the variable is meaningless, or the user's program is terminated with an error message, after which he has to re-submit the entire program (after fixing the error, hopefully). The disparity between this procedure and that available in a language such as JOSS is one of the main reasons for the interest in interactive systems.

One last item of note is that variables in many conversational systems are considered "typeless"; i.e., it is not necessary for the program to specify that X, say, is real. The fact that the value of X at some moment is real is sufficient for the system to use it correctly. Moreover, the type of X may be allowed to change as the value of X is changed; this it may be an integer one moment, and a real or string value at some later time. The prime motivation for this facility is that such "internal bookkeeping" is part of the common context of the interaction - thus, explicit reference to such items ought to be unnecessary.