Elephant 2000: A Programming Language Based on Speech Acts

John McCarthy, Stanford University

I meant what I said, and I said what I meant.
An elephant's faithful, one hundred percent!

moreover,

An elephant never forgets!

Abstract

Elephant 2000 is a proposed programming language good for writing and verifying programs that interact with people (e.g. transaction processing) or interact with programs belonging to other organizations (e.g. electronic data interchange)

1. Communication inputs and outputs are in an I-O language whose sentences are meaningful speech acts identified in the language as questions, answers, offers, acceptances, declinations, requests, permissions and promises.
2. The correctness of programs is partly defined in terms of proper performance of the speech acts. Answers should be truthful and responsive, and promises should be kept. Sentences of logic expressing these forms of correctness can be generated automatically from the form of the program.
3. Elephant source programs may not need data structures, because they can refer directly to the past. Thus a program can say that an airline passenger has a reservation if he has made one and hasn't cancelled it.
4. Elephant programs themselves can be represented as sentences of logic. Their extensional properties follow from this representation without an intervening theory of programming or anything like Hoare axioms.

5. Elephant programs that interact non-trivially with the outside world can have both *input-output specifications*, relating the programs inputs and outputs, and *accomplishment specifications* concerning what the program accomplishes in the world. These concepts are respectively generalizations of the philosophers' *illocutionary* and *perlocutionary* speech acts.

6. Programs that engage in commercial transactions assume obligations on behalf of their owners in exchange for obligations assumed by other entities. It may be part of the specifications of an Elephant 2000 programs that these obligations are exchanged as intended, and this too can be expressed by a logical sentence.

7. Human speech acts involve intelligence. Elephant 2000 is on the borderline of AI, but the article emphasizes the Elephant usages that do not require AI.

1 Introduction

One stimulus for writing this article was a talk entitled "Programming languages of the year 2000" that I considered insufficiently ambitious.

It has long been said that programming languages need more of the features of natural language, but it has not been clear what the desirable features are, and there have been few significant conceptual advances in the last 20 years. It is rather clear that the surface syntax of natural language doesn't offer much; COBOL did not turn out to be an advance.

This article proposes several new features for programming languages more or less taken from natural language. We also propose new kinds of specifications that make it easier to be confident that the specifications express what is wanted of the program. As a vehicle we propose Elephant 2000, a language that would embody them. Some features are not yet well enough defined to be included in the program examples.

1. Elephant programs will communicate with people and other programs in sentences of the Elephant I-O language which have meanings partially determined by the Elephant language and partially implicit in the program itself. Thus Elephant input and output include, and the I-O language distinguishes, *requests*, *questions*, *offers*, *acceptances of offers*, *permissions* as well as *answers to questions* and other assertions

of fact. Its outputs also include *promises* and statements of *commitment* analogous to promises.

2. Some of the conditions for correctness of an Elephant program are defined in terms of the meaning of the inputs and outputs. We can ask whether an Elephant program fulfilled a request, answered a question truthfully and responsively, accepted an offer or fulfilled a commitment. We can also ask whether the program has authority to do what it does. Some of such correctness conditions are *intrinsic*, because the text of the program determines them. Expressing these intrinsic correctness conditions as sentences of logic requires a formal theory of what it means to fullfil a commitment, etc. This theory doesn't have to correspond exactly to human behavior or social customs; we only need analogs useful for program correctness. They are somewhat analogous to the grammaticality conditions of present programming languages, but they are semantic rather than syntactic.

   For example, when a program "promises" someone to do something, it needn't believe (as Searle (1969) suggests it should) that fulfillment of the promise will do its recipient some good. Indeed many programs that promise won't have any beliefs at all. We expect to be able to generate the intrinsic correctness sentences automatically from the text of the program. Thus the text of the program that designates certain outputs as answers to questions, determines a logical sentence asserting that the answers are truthful.

3. Elephant programs do not require data structures, because program statements can refer directly to past events and states. An Elephant interpreter keeps a history list of past events, and an Elephant compiler constructs whatever data structures in the object language are needed to remember the information needed for compiled program to behave as specified in the Elephant source program. However, it seems unlikely that it will be convenient to omit data structures completely from the language.

4. An Elephant program is itself a logical sentence (or perhaps a syntactic sugaring of a logical sentence). The extensional correctness properties of the program are logical consequences of this sentence and a theory of the domain in which the program acts. Thus no logic of programs is

needed. Any sugaring should be reversible by a statement-by-statement syntactic transformation, thus getting the logic back.

5. Requests, permissions and promises such as those we want Elephant programs to perform are called *speech acts* by philosophers and linguists. The idea is that certain sentences don't have only a declarative significance but are primarily actions. A paradigmatic example is a promise, whose utterance creates an *obligation* to fulfill it and is therefore not merely a statement of intention to do something. For some purposes, we can bypass the the philosophical complexities of *obligation* by considering only whether a program *does* fulfill its promises, not worrying about whether it is obliged to.

   In the customary philosophical terminology, some of the outputs of Elephant programs are *performative* sentences, commonly referred to just as *performatives.* Indeed Elephant 2000 started with the idea of making programs use performatives. However, as the ideas developed, it seemed useful to deviate from the notions of speech act discussed by J. L. Austin (1962) and John Searle (1969). Thinking about speech acts from the *design standpoint* of Daniel Dennett (1971) leads to a view of them different from the conventional one. We now refer to *abstract performatives* which include purely internal actions such as *commitments* not necessarily expressed in output, but on whose fulfillment the correctness of the program depends. Taking the design stance in the concrete way needed to allow programs to use speech acts tends to new views on the philosophical problems that speech acts pose.

   Notice that it isn't necessary for most purposes to apply moral terms like *honest*, *obedient* or *faithful* to the program, and we won't in this paper. However, we can incorporate whatever abstract analogs of these notions we find useful. The philosophical investigations have resulted in ideas useful for our purposes. This is partly because programs belonging to one organization that interact with those belonging to other organizations will have to perform what amounts to speech acts, and the specifications of these programs that have to be verified often correspond to what Austin calls the *happy* performance of the speech acts.

   (McCarthy 1979a) discusses conditions under which computer programs may be ascribed beliefs, desires, intentions and other mental qualities. It turns out that some specifications require ascription of beliefs, etc. for their proper statement, and others do not.

Allowing direct reference to the past may also permit easier modification, because the program can refer to past events, e.g. inputs and outputs, directly rather than via data structures whose design has to be studied and understood. Since referring directly to past events is characteristic of natural languages, we expect it to prove useful in programming languages.

6. The theory of speech acts distinguishes between *illocutionary* acts, such as telling someone something, and *perlocutionary* acts, such as convincing him of it. The distinction between illocutionary and perlocutionary can be applied to speech inputs as well as outputs. Thus there is an input distinction between *hearing that* and *learning that* analogous to the output distinction between telling and convincing.

   Procedures for human execution often specify perlocutionary acts. For example, a teacher might be told, "Have your students take the test on Wednesday". However, including perlocutionary acts in programs is appropriate only if the program has good enough resources for accomplishing goals to make it reasonable to put the goal in the program rather than actions that the programmer believes will achieve it. One would therefore expect perlocutionary statements mainly in programs exhibiting intentionality, e.g. having beliefs, and involving some level of artificial intelligence.

   Even without putting perlocutionary acts in the program itself, it is worthwhile to consider both input-output and accomplishment specifications and for programs. These correspond to illocutionary and perlocutionary speech acts respectively. For example, an air traffic control program may be specified in terms of the relations between its inputs and its outputs. This is an input-output specification. Its verification involves only the semantics of the programming language. However, our ultimate goal is to specify and verify that the program prevents airplanes from colliding, and this is an accomplishment specification. Proving that a program meets accomplishment specifications must be based on assumptions about the world, the information it makes available to the program and the effects of the program's actions as well on facts about the program itself. These specifications are external in contrast to the intrinsic specifications of the happy performance of speech acts.

It will often be worthwhile to formulate both input-output and accomplishment specifications for the same program and to relate them. Thus an argument based on an axiomatic theory of the relevant aspects of the world may be used to show that a program meeting certain input-output specifications will also meet certain accomplishment specifications. Apparently the dependence of program specifications on facts about the physical world makes some people nervous. However, the problem of justifying such axiomatic theories is no worse than that of justifying other formalized theories of the physical world in applied mathematics. We are always trusting our lives to the physical theories used in the design of airplane wings.

7. The most obvious applications of Elephant are in programs that do transaction processing and refer to databases in more general ways than just answering queries and making updates. Abstract performatives will also be important for programs involved in business communication with programs belonging to other organizations. (McCarthy 1982) suggests a "Common Business Communication Language".

This article is exploratory, and we are not yet prepared to argue that every up-to-date programming language of the year 2000 will include *abstract performatives*. We hope that programs using performatives will be easier to write, understand, debug, modify and (above all) verify. Having a standard vocabulary of requests, commitments, etc. will help.

2 Speech Acts and Abstract Performatives

Philosophers and linguists mostly treat speech acts naturalistically, i.e. they ask what kinds of speech acts people perform. Even when they intend to approach the problem abstractly, the analyses seem to me to be too close to human behavior. Our approach is to ask what kinds of speech act are useful in interactive situations. The nature of the interaction arises from the fact that the different agents have different goals, knowledge and capabilities, and an agent's achieving its goals requires interaction with others. The nature of the required interactions determines the speech acts required. Many facts about what speech acts are required are independent of whether the agent is man or machine.

Often our programs can perform successfully with a small repertoire of speech acts, event though these speech acts don't have all the properties Searle (1969) and other philosophers have ascribed to human speech acts.

Here are some of the kinds of speech acts we plan to provide for.

1. Assertions. One correctness condition is that the assertions be truthful. Of course, proving them true is based on axioms about the domain in which the program operates. A weaker correctness condition is that the assertions be sincere. This requires a theory of the beliefs of the program. If proving the truth or sincerity of certain kinds of statements is too difficult, the programmer may choose to omit such correctness conditions and ask the user to rely on his intuitions.

2. Questions. The user can question the program, and the program can question the user.

3. Answers to questions. These are assertions and should be sincere and truthful. However, they should also be responsive to the questions. If the question calls for a yes or no answer, responsiveness is easier to formulate than otherwise. However, there may still be difficulties of the sort discussed by Grice in the papers collected in (Grice 1989). For some questions, the appropriate answer is "I don't know". Other questions make assumptions that the questionee may know to be false, and a literal answer may be inappropriate.

   Suppose Pat asks for Mike's telephone number. It is unresponsive to say that Mike's telephone number is the same as Mike's wife's husband's telephone number. One way of requiring responsiveness is to require that the answer be a sequence of digits and not just an expression whose value is a sequence of digits. This may be too narrow. To say that Mike's number is the same as Mike's wife's number might be responsive under some circumstances. Another approach is described in section 10, which is about responsiveness.

   The assertions and questions of an Elephant program involve certain predicate and function symbols common to the user and the program. These make up the *I-O language* of the program and are to be combined using the tools of first order logic. Natural language front ends might be provided if found helpful. The I-O language of a particular program is described in the manual for the program. However, certain predicates and functions are common to all Elephant 2000 programs, whereas many others are close enough to ordinary language words to be usable without study.

4. Commitments and promises. A *simple promise* is an internal commitment to do something, i.e. to make some sentence true, and an output assertion that the commitment exists. Correctness requires that the assertion be truthful.

   However, Austin (1962) points out that uttering a promise creates an obligation in a public sense. This will be important also for computer programs, since a promise by the program may create an obligation, perhaps legal, on the part of the organization operating the program. Sometimes, it will be convenient to settle for specifications in terms of fulfilling internal commitments, and sometimes the public character of promises will have to be taken into account in the specifications.

   Internal commitments constitute one kind of *abstract performative*; presumably there are others. It is abstract in that its content is independent of how it is expressed, and it need not be externally expressed at all.

   We also propose to handle external obligations in an abstract way. Suppose a notions of type 1 and type 2 obligations are introduced. We specify that a reservation program incurs a type 1 obligation when it makes a reservation and imposes a type 2 obligation on the organization that operates it. What type 1 and type 2 obligations are, e.g. the legal requirements they impose, how they are to be treated when the conflict with other considerations, and the consequences of their nonfulfillment, is subject to institutional, e.g. legal, definition. These kinds of obligation are created via social, commercial or legal means rather than being discovered by philosophers or pyschologists.

3 Referring to the past

Algolic programs refer to the past via variables, arrays and other data structures. Elephant 2000 programs can refer to the past directly. In one respect,this is not such a great difference, because the program can be regarded as having a single virtual history list or "journal". It is then a virtual side-effect of actions to record the action in the history list. Calling the history list virtual means that it may not actually take that form in a compiled program which may use conventional arrays and not record information that is sure not to be referred to. However, we propose to provide ways of referring to the past that are semantically more like the ways people refer to the past in natural languages than just having an index variable into a list

We consider the past as a history, a function of time that gives the events and states that have occurred. Referring to the past involves using some function of this history. Here are some examples.

1. The simplest function of the past is the value of some parameter at a given time, say the account balance of a certain person on January 5, 1991. References to the past are rarely this simple.

2. Next we may consider the time of a certain event, say the time when a person was born.

3. Slightly more complex is the first or last time a certain event occurred or a certain parameter had a certain value, say the most recent time a certain person was overdrawn at his bank.

4. More generally, we may consider the unique time or the first or last time a certain proposition was true.

5. Still more generally, we will be interested in time-valued functions of the whole past, e.g. an average time.

6. Whether a person has an airplane reservation for a certain flight is determined by whether one has been made for him and not subsequently cancelled.

7. An interpreter for a programming language with subroutines might avoid explicit mention of a stack by saying that a program returns from a subroutine to the statement following the entry that corresponds to the current execution of the return statement. It must also say that when the return occurs the variables have the values they had before the subroutine was entered. It isn't obvious that avoiding explicit mention of a stack would be a good idea, but it might have the advantage that a compiler would have more flexibility in how it chose to remember the necessary information than if an explicit data structure were specified.

8. The wages of an hourly worker for a week is obtained by adding up the lengths of the intervals during which he was on the job, each multiplied by the pay rate for that time, e.g. more for working on the graveyard shift. We shall see that this is ontologically more complex than the previous examples, because it requires sets of intervals as objects and not just individual times.

   For the purposes of the Elephant language we shall devise a set of ways of expressing these functions of the past that are general enough to obviate the need for many references to data structures and are still computationally feasible.

4 The Structure of Elephant 2000 programs

A simple version of Elephant 2000 will be described in terms of an interpreter. Compiled programs are to have the same input-output behavior, but the object program uses ordinary data structures instead of referring directly to the past.

We suppose that only one input reaches the program at a time; the run-time system is supposed to achieve this. We also suppose that inputs not of the required form are rejected, so that the Elephant program itself doesn't have to say what is done with them.

The program responds to each input as it is received. Thus it can be regarded as a stimulus-response machine. However, in deciding on a response the program may inspect the entire past of its previous inputs and responses. There may also be a permanent database to which the program refers.

5 Examples of Programs

1. Here is an airline reservation program accompanied by an explanation of the notations used. The identifiers in bold face are defined in the Elephant language; identifiers in italics belong to of the particular program.

accept.request is an action defined in the Elephant language. It means to do what is requested.

answer.query is the action of answering a query.

In general, $admit(psgr, flt)$ means admitting the passenger $psgr$ to the flight flt. It is an action defined in the particular program. However, it has two possible interpretations, (1) telling the agent at the gate to let the passenger in and (2) actually letting the passenger in. The first is an illocutionary act, the second is a perlocutionary act.

commitment is a function taking a future action as an argument and generating from it an abstract object called a commitment. The internal actions **make** and **cancel** and the predicate **cancel** apply to commitments. **make**, **exists** and **cancel** are part of the Elephant language and have meanings and be axiomatized independently of the particular abstract objects being made, tested and destroyed. The notion of faithfulness involving the fulfillment of commitments will not necessarily extend to other abstract entities.

$$\textbf{if}\neg full\ flt\ \textbf{then accept.request make commitment}\ admit(psgr, flt).$$

In Elephant 2000, we associate to the right, so the above statement is equivalent to

$$\textbf{if}\ \neg full(flt)\ \textbf{then accept.request(}\ \textbf{make commitment}\ (admit(psgr, flt))).$$

$$\textbf{answer.query exists commitment}\ admit(psgr, flt).$$

$$\textbf{accept.request cancel commitment}\ admit(psgr, flt).$$

$$\begin{array}{l}\textbf{if}\ \ \textbf{now} = \textbf{time}\ flt \wedge\ \textbf{exists commitment}\ admit(psgr, flt) \wedge \neg\ full1\ flt \\ \qquad \textbf{then accept.request}\ admit(psgr, flt)\end{array}$$

$$full\ flt \equiv card\{psgr|\textbf{exists commitment}\ admit(psgr, flt)\} = capacity\ flt.$$

Even for this simple reservation program it is worthwhile to distinguish between input-output and accomplishment specifications. The distinction comes up in the interpretation of $admit(psgr, flt)$. If we interpret it as ordering the admission, then it's an illocutionary act. If we interpret is as making sure the passenger can actually get on, then it is a perlocutionary act.

It is an input-output specification of the program that it not order the admission of more passengers than the capacity of the flight. It is an external fact that the plane will hold its capacity and not more.

2. Similar considerations apply to a minimal program for a control tower.

It receives requests to land. It can perceive the positions of the airplanes. It can issue instructions like "Cleared to land" or "You are number 3 following the red Cessna" or "Extend your downwind leg" or "Turn base now". Its input-output specifications involve only its perceptions and its inputs and outputs. They include that it should not say "Cleared to land" to an airplane until the previous airplane is perceived to have actually landed. The verification of the input-output specifications involve facts about the program itself.

The accomplishment specifications provide that the airplanes should land safely and as promptly as is compatible with safety. Their verification involves assumptions about the correctness of the program's perceptions and the behavior of airplanes in response to instructions.

(The word "perceive" is ambiguous in English usage. The *Webster's Collegiate* dictionary uses "become aware of", and this implies that if one "perceives that an airplane has left the runway", then it really has left it. However, a poll of colleagues found that most considered that a person could perceive it without it actually being true. For our purposes we need both notions, so we'll add adverbs to make the meaning clear.)

A specialist in airplanes may verify that if the program meets its input-output specifications, then it will also meet its accomplishment specifications.

The concepts of input-output and accomplishment specifications apply to programs that interact with the outside world in general and not just Elephant programs. However, we expect them to be easier to state and verify for Elephant programs.

6 Abstract Objects

Consider airline reservations. What is an airline reservation? That's not the right question. Maybe we want to ask how we should define an airline reservation for the purposes of an Elephant reservation program. This is still too definite. What do we want to say about airline reservations to our program? As little as possible. We need to say enough so that the program can make, cancel, answer questions about, and honor reservations. Saying more than what is necessary unnecessarily restricts the implementation. It also increases what has to be known in order to modify the program.

So what do we need?

7 Implementation

We contemplate that Elephant 2000 will have two kinds of implementation, interpreted and compiled.

The interpreted implementation has a single simple data structure—a history list of events. Inputs will certainly be included as events. In principle, this is all that is required. However, including actions by the program in the history enables the interpreter to avoid repeating certain computations. These actions are both external and internal.

The interpreter then examines inputs as they come in and according to the rules decides what to do with each one. The version of Elephant 2000 presently contemplated only provides for certain requests, etc. and assumes they arrive in some order. The necessary synchronization is performed by the runtime system outside of the program itself, and so is the rejection of inputs that don't match any program statement.

Matching event patterns to the history of events is done explicitly each time an input is interpreted. The actions that are logged are those that are part of the Elephant 2000 language. The simple form of matching done by Prolog may be adequate for this purpose.

The interpreter will therefore be somewhat slow, but for many purposes it will be adequate.

An Elephant 2000 compiler will translate programs into an ordinary programming language, e.g. Common Lisp or C. It would put data structures in the object program that would permit reference to these structures in order to decide what to do and would update the structures according to the action taken. In a full translation, there would be no explicit history in the object program.

8 Specifying and Verifying Elephant 2000 Programs

Since Elephant 2000 programs will be sugared versions of logical sentences, their properties will be logical consequences of the sentences expressing the program, the axioms of Elephant 2000 and a theory of the data domains (if any) of the program. In this Elephant resembles Algol 48 and Algol 50 described in a later section.

(more to come)

9 Levels of Intentionality

We discuss philosophical work on speech acts with two objectives. First, we consider what the computer language use of speech acts can learn from the extensive work by philosophers. Second, considering speech acts as we want computers to do them sheds light on the philosohical problems. The two aspects of the philosohical treatments are so interrelated that we discuss them together. Here are some remarks.

1. Philosophers treat speech acts as natural phenomena to be studied. However, they propose not to treat them from the point of view of anthropology or linguistics. Instead they study their essential characteristics in terms of what they accomplish. This makes their work more relevant to computer science and artificial intelligence than linguistic or anthropological work would be.

2. My view of speech acts is that they are necessary in the *common sense informatic situation* in which people interact with each other to achieve their goals. The most important features of this informatic situation are independent of the fact that we are humans. Martians or robots with independent knowledge and goals would also require speech acts, and many of these would have similar characteristics to human speech acts.

3. The point of this paper is that speech acts are valuable when we design computer systems to interact with humans and with each other.

4. However, only some of the characteristics that philosophers have ascribed to speech acts are valuable for our purposes. Which ones they are will depend on the purposes.

5. Besides the speech acts that are common in human society, it is convenient to invent others. Indeed human institutions often involve the invention of speech acts. An example is the airplane reservation discussed in this paper.

6. A particular kind of speech act is an entity in an approximate theory in the sense of (McCarthy 1979a). For this reason attempts at precise definitions, e.g. of an airplane reservation, are likely to be beside the

point. Instead we will have nonmonotonic axioms (McCarthy 1986) that partially characterize them.

7. Regarding speech acts as events of execution of program statements may be useful for philosophers also.

8. Performatives that are not really speech acts because they don't result in external output are also useful. Our main example is the commitment. When a program makes a commitment, its correctness requires the fulfillment of the commitment.

9. A key question we share with philosophers is that of what must be true in order that a speech act of a given kind be successfully performed.

10. This paper plays some role in the controversy between John Searle (1984) and the artificial intelligence community about whether computers can really believe and know. As I understand Searle's position, a natural extension would say that computers can't really promise either. Our position is that Elephant 2000 programs can perform some kinds of speech acts as genuinely as do humans. The difference between what Elephant 2000 programs do and what some humans to in this respect will be similar to the differences among humans. We expect that the programming community will for a long time be interested in speech acts of a more limited sort than Searle has discussed. However, human speech acts when performed in certain institutional settings also have a limited character. For example, giving a reservation usually does not involve an opinion that it will benefit the person to whom it is given.

    It's not clear that a difference of opinion on this point has practical consequences for programming.

11. It will often be possible to regard a program not written in Elephant as though it were by regarding its inputs as questions and requests and its outputs as promises, etc. This is the Elephant analog of Newell's (1982) logic level or my (1979a) about ascribing mental qualities to machines.

12. Austin and Searle distinguish *illocutionary* from *perlocutionary* speech acts. An example is that ordering someone to do something is illocutionary, but getting him to do it is perlocutionary. The same sentence

may serve as both, but the conditions for successful perlocutionary acts don't just involve what the speaker says; the involve its effect on the hearer. Both philosophers mention difficulties in making the distinction precise, but for the purposes of Elephant it's easy.

The correctness conditions for an illocutionary act involve the state of the program and its inputs and outputs. The correctness conditions for a perlocutionary act depends also on events in the world. An airline reservation program may reasonably be specified in terms of the illocutionary acts it performs, i.e. by its inputs and outputs, whereas the correctness of an air traffic control program is essentially perlocutionary, because stating the full correctness of the latter involves stating that it prevents the airplanes from colliding.

In this connection it may be worthwhile to go beyond philosophical usage and apply the term *perlocutionary* to inputs as well as outputs. Namely, perlocutionary conditions on the inputs state that they give facts about the world, e.g. the locations of the airplanes. The correctness of an air traffic control program depends on assumptions about the correctness of the inputs as well as on assumptions about the obedience of the pilots and the physics of airplane flight.

10 Responsiveness

Requiring that answers to questions be responsive as well as truthful raises questions involving intentionality and/or metamathematics.

Suppose I ask someone at a party, "Who is that man over there?" and he replies "Tom Jones". That would normally be considered a responsive answer. Suppose that instead I ask, "Who is Tom Jones?" and the replies, "That man over there". This is also normally considered responsive. This indicates that the general problem of responsiveness is difficult, and we shall want to consider some special cases.

Stating and proving that answers to questions and other statements are responsive seems to require a substantially larger logical apparatus than merely proving that the answers are truthful. It turns out that the logical apparatus proposed in (McCarthy 1979b), which uses separate notations to denote objects and concepts of objects is suitable for the task.

An answer is responsive provided the questioner will *know* the answer to the question after he receives it. Suppose Pat asks for Mike's telephone number. An answer like "Mike's telephone number is the same as Mike's wife's husband's telephone number" is unresponsive, and can be characterized as by noting that it doesn't make Pat know Mike's telephone number. Using (McCarthy 1979b) we can define

$$knows\text{-}what(pat, Telephone\ Mike) \equiv$$

$$\exists x(telephone\text{-}number\ x \wedge knows\text{-}that(pat,$$

$$Equal(Telephone\ Mike, Concept1\ x))).$$

Here the function $Concept1$ maps a telephone number into a standard concept of that telephone number, i.e. one that permits the person who has it to dial the number. The range of $Concept1$ needs to be suitably axiomatized.

11 Connections with Artificial Intelligence

The full use of speech acts requires intelligence on the human level. Artificial intelligence has made progress, and there is a useful expert systems technology based on this progress, but human-level intelligence is still an unknown distance ahead of us. Therefore, it is important to analyze the proposed uses of speech acts and decide how much intelligence is required for each use. It is a contention of this paper that many uses of speech acts do not require many of the intellectual capabilities of humans. We further hope that Elephant 2000 will permit using whatever level of AI is feasible for the designers of any particular system. Care will be needed to ensure that a program doesn't require too much intelligence of the programs with which it interacts.

(more to come)

12 Domain of Application

A very large fraction of office work involves communicating with people and organizations outside the office and whose procedures are not controlled by the same authority. Getting the full productivity benefit from computers depends on computerizing this communication. This cannot be accomplished by approaches to distributed processing which assume that the communicating processes are designed as a group.

Automating such communication was easiest when it is absolutely standard what is communicated and when one of the organizations is in a position to dictate formats. For example, in the 1950s the IRS dictated a magnetic tape format on which it was prepared to receive reports of wages and deductions.

The initial approaches to electronic data interchange, EDI, have involved exchanges of information between a large manufacturer and its suppliers, where the customer could dictate the form of the interchange. This works best when the supplier has only this one customer.

A next step is provided by standards like X12 which provide standard electronic formats for a fixed set of commercial documents, e.g. invoices. Presumably, X12 forms have fixed collections of slots in which can be inserted numbers, e.g. quantities and prices, or labels, e.g. names of people, places and things.

(McCarthy 1982) proposes a "Common Business Communication Language" that would provide for a language of business messages. These messages could include complex descriptions of various entities, e.g. price formulas, schedules of delivery, schedules of payment, configurations of equipment and terms for contracts. That paper envisages that firms may issue electronic catalogs and advertisements that cause thems to be listed automatically as suppliers of certain items. Programs looking for good prices and delivery conditions for the items might automatically negotiate with the programs representing the sellers and even issue purchase orders within their authority to do so. (McCarthy 1982) did not discuss the programs that would use CBCL.

Programs that carry out external communication need several features that substantially correspond to the ability to use speech acts. They need to ask questions, answer them, make commitments and make agreements. They need to be able communicate assurance that their agreements will be honored by the organizations they belong to. This is best assured by some kind of authority tree extending up from programs that issue purchase orders through the programs that decide what items are to be purchased and through the human hierarchy of the organization.

Because these programs will often operate with only minimal human supervision, they need to be carefully verified. The features of Elepant 2000 are important for this, because the forms of speech act provided for have definite meanings. They will also need frequent modification, often by people other than those who wrote them—by non-programmers as much as possible.

As people acquire more home computers and use them more and more for doing business with firms, transaction processing programs will become more important, and will be more used by people other than employees of the organizations operating the programs. This will also put requirements on verification and modifiability.

Present reservation programs have many limitations that using Elephant 2000 features will encourage correcting. Often they don't even emit strings that are supposed to have long term meaning. Instead they emit display updates. These display updates cannot be conveniently used by programs belonging to others, because changes in output intended to make the display prettier can destroy the ability of other programs to decipher them. The result is that when a ticket has to be changed, the airline counter clerk often retypes the name of the passenger, because it cannot be taken from the screen showing the reservation. Elephant speech acts are a step above strings, because the higher levels of the speech acts have a meaning independent of the application program.

13 Algol 48 and Algol 50

This section is a warm-up for the next section.

We introduce the "programming languages" Algol 48 and Algol 50 to illustrate in a simpler setting some ideas to be used in Elephant 2000. These are the explicit use of time in a programming language and the representation of the program by logical sentences. The former permits a direct expression of the operational semantics of the language, and the latter permits proofs of properties of programs without any special theory of programming. The properties are deduced from the program itself together with axioms for the domain.

We use these names, because the languages cover some of the ground of Algol 60 but use only a mathematical formalism— old fashioned recursion equations—that precedes the development of programming languages. They are programming languages I imagine mathematicians might have created in 1950 had they seen the need for something other than machine language. Algol 48 is a preliminary version of Algol 50 just as Algol 58 was a preliminary version of Algol 60.

Consider the Algol 60 fragment.

$$
\begin{array}{lll}
0 & \mathit{start}: & p := 0; \\
1 & & i := n; \\
2 & \mathit{loop}: & \textbf{if } i = 0 \textbf{ then go to } \mathit{done}; \\
3 & & p := p + m; \\
4 & & i := i - 1; \\
5 & & \textbf{go to } \mathit{loop}; \\
6 & \mathit{done}: &
\end{array}
$$

The program computes the product mn by initializing a partial product p to 0 and then adding m to it n times. The correctness of the Algol 60 program is represented by the statement that if the program is entered at *start* it will reach the label *done*, and when it does, the variable p will have the value mn. Different program verification formalisms represent this assertion in various ways, often not entirely formal.

Its partial correctness is conventionally proved by attaching the invariant assertion $p = m(n - i)$ to the label *loop*. Its termination is proved by noting that the variable i starts out with the value n and counts down to 0. This proof is expressed in various ways in the different formalisms for verifying programs.

13.1 Algol 48

In Algol 48 we write this algorithm as a set of old fashioned recursion equations for three functions of time, namely $p(t)$, $i(t)$ and $pc(t)$, where the first two correspond to the variables in the program, and $pc(t)$ tells how the "program counter" changes. The only ideas that would have been unconventional in 1948 are the explicit use of a program counter and the conditional expressions. We have

$$
\begin{aligned}
p(t+1) = \quad & \textbf{if } pc(t) = 0 \textbf{ then } 0 \\
& \textbf{else if } pc(t) = 3 \textbf{ then } p(t) + m \\
& \textbf{else } p(t),
\end{aligned}
$$

$$
\begin{aligned}
i(t+1) = \quad & \textbf{if } pc(t) = 1 \textbf{ then } n \\
& \textbf{else if } pc(t) = 4 \textbf{ then } i(t) - 1 \\
& \textbf{else } i(t),
\end{aligned}
$$

and

$$
\begin{aligned}
pc(t+1) = \quad & \textbf{if } pc(t) = 2 \wedge \ i(t) = 0 \textbf{ then } 6 \\
& \textbf{else if } pc(t) = 5 \textbf{ then } 2 \\
& \textbf{else } pc(t) + 1.
\end{aligned}
$$

The correctness of the Algol 48 program is represented by the sentence

$$\forall m \ n(n \geq 0 \rightarrow \forall t(pc(t) = 0 \rightarrow \exists t'(t' > t \wedge pc(t') = 6 \wedge p(t') = mn)))$$

This sentence may be proved from the sentences representing the program supplemented by the axioms of arithmetic and the axiom schema of mathematical induction. No special theory of programming is required. The easiest proof uses mathematical induction on n applied to a formula involving $p(t) = m(n - i(t))$.

Algol 48 programs are organized quite differently from Algol 60 programs. Namely, the changes to variables are sorted by variable rather than sequentially by time.

13.2 Algol 50

However, by reifying variables, Algol 50 permits writing programs in a way that permits regarding programs in this fragment of Algol 60 as just sugared versions of Algol 50 programs.

Instead of writing $var(t)$ for some variable var, we write $value(var, \xi(t))$, where ξ is a state vector giving the values of all the variables. In the above program, we'll have $value(p, \xi(t))$, $value(i, \xi(t))$ and $value(pc, \xi(t))$.

The variables of the Algol 60 program correspond to functions of time in the above first Algol 50 version and become distinct constant symbols in the version of Algol 50 with reified variables. Their distinctness is made explicit by the "unique names" axiom

$$i \neq p \wedge i \neq pc \wedge p \neq pc.$$

In expressing the program we use the assignment and contents functions, $a(var, value, \xi)$ and $c(var, \xi)$, of (McCarthy 1963) and (McCarthy and Painter 1967). $a(var, value, \xi)$ is the new state ξ' that results when the variable var is assigned the value $value$ in state ξ. $c(var, \xi)$ is the value of var in state ξ.

As described in those papers the functions a and c satisfy the axioms.

$$c(var, a(var, val, \xi)) = val,$$

$$var1 \neq var2 \rightarrow c(var2, a(var1, val, \xi)) = c(var2, \xi),$$

$$a(var, val2, a(var, val1, \xi)) = a(var, val2, \xi),$$

and

$$var1 \neq var2 \rightarrow a(var2, val2, a(var1, val1, \xi)) = a(var1, val1, a(var2, val2, \xi)).$$

The following function definitions shorten the expression of programs. Note that they are just function definitions and not special constructs.

$$step(\xi) = a(pc, value(pc, \xi) + 1, \xi),$$

$$goto(label, \xi) = a(pc, label, \xi).$$

We make the further abbreviation $loop = start + 2$ specially for this program, and with this notation our program becomes

$$
\begin{array}{ll}
\forall t & (\xi(t+1) = \\
& \quad \textbf{if } c(pc, \xi(t)) = start \\
& \qquad \textbf{then } step\ a(p, 0, \xi(t)) \\
& \quad \textbf{else if } c(pc, \xi(t)) = start + 1 \\
& \qquad \textbf{then } step\ a(i, n, \xi(t)) \\
& \quad \textbf{else if } c(pc, \xi(t)) = loop \\
& \qquad \textbf{then } (\textbf{if } c(i, \xi(t)) = 0 \textbf{ then} \\
& \qquad\quad goto(done, \xi(t)) \\
& \qquad\quad \textbf{else } step\ \xi(t)) \\
& \quad \textbf{else if } c(pc, \xi(t)) = loop + 1 \\
& \qquad \textbf{then } step\ a(p, c(p, \xi(t)) + m, \xi(t)) \\
& \quad \textbf{else if } c(pc, \xi(t)) = loop + 2 \\
& \qquad \textbf{then } step\ a(i, c(i, \xi(t)) - 1, \xi(t)) \\
& \quad \textbf{else if } c(pc, \xi(t)) = loop + 3 \\
& \qquad \textbf{then } goto(loop, \xi(t)) \\
& \quad \textbf{else } \xi(t+1))
\end{array}
$$

In Algol 50, the consequents of the clauses of the conditional expression are in 1-1 correspondence with the statements of the corresponding Algol 60 program. Therefore, the Algol 60 program can be regarded as an abbreviation of the corresponding Algol 50 program. The (operational) semantics of the Algol 60 program is then given by the sentence expressing the corresponding Algol 50 program together with the axioms describing the data domain, which in this case would be the Peano axioms for natural numbers. The transformation to go from Algol 60 to Algol 50 would be entirely local, i.e. statement by statement, were it not for the need to use statement numbers explicitly in Algol 50.

Program fragments can be combined into larger fragments by taking the conjunction of the sentences representing them, identifying labels where this is wanted to achieve a **go to** from one fragment to another and adding sentences to make sure that the program counter ranges don't overlap.

The correctness of the Algol 50 program for multiplication by addition is expressed by

$$\begin{aligned}\forall t \xi_0 (c(pc, \xi(t)) = start \quad & \wedge \xi(t) = \xi_0 \\ & \rightarrow \exists t'(t' > t \wedge c(p, \xi(t')) = mn \\ & \quad \wedge c(pc, \xi(t')) = done \\ & \quad \wedge \forall var(\neg(var \in \{p, i, pc\}) \rightarrow c(var, \xi(t')) = c(var, \xi_0))))\end{aligned}$$

Note that we quantify over all initial state vectors. The last part of the correctness formula states that the program fragment doesn't alter the state vector other than by altering p, i and pc.

We have not carried the Algol 50 idea far enough to verify that all of Algol 60 is conveniently representable in the same style, but no fundamental difficulties are apparent. In treating recursive procedures, a stack can be introduced, but it would be more elegant to do without it by explicitly saying that the return is to the statement after the corresponding procedure call and variables are restored to their values at the time of the call. This requires the ability to parse the past, needed also for Elephant 2000.

We advocate an extended Algol 50 for expressing the operational semantics of Algol-like programming languages, i.e. for describing the sequence of events that occurs when the program is executed. However, our present application is just to illustrate in a simpler setting some features that Elephant will require. In particular, proper treatment of calling a function procedure with side-effects will require a state that can have a value during the evaluation of an expression.

Nissim Francez and Amir Pnueli (see references) used an explicit time for similar purposes. Unfortunately, they abandoned it for temporal logic.

While some kinds of temporal logic are decidable, temporal logic is too weak to express many important properties of programs.

14 Elephant Programs as Sentences of Logic

This is the most tentative section of the present article. At present we have two approaches to writing Elephant programs as sentences of logic. The first approach is analogous to Algol 50. It is based on updating a state of the program and a state of the world. Of course, the functions updating the state of the world will be only partially known. Therefore, unknown functions will occur, and the knowledge we have about the world will be expressed by subjecting these functions to axioms. The second approach expresses the program and tentatively what we know about the world in terms of events. The events occur at times determined by axioms. We describe the events that we assert to occur and rely on circumscription to limit the set of occurrences to those that follow from the axioms given. We begin with the Algol 50 approach.

As with Algol 50, in our first approach we use a state vector $\xi(t)$ for the state of the program during operation. The program is expressed by a formula for $\xi(t+1)$. Since the program interacts with the world, we also have a state vector $world(t)$ of the world external to the computer. Because we can't have complete knowledge of the world, we can't expect to express $world(t+1)$ by a single formula, but proving accomplishment specifications will involve assumptions about the functions that determine how $world(t)$ changes.

We have

$$\xi(t+1) = update(i(t), \xi, t),$$

$$i(t) = input\ world(t),$$

and

$$world(t+1) = worldf(output\ \xi(t), world, t).$$

Notice that we have written ξ and $world$ on the right sides of the two equations rather than $\xi(t)$ and $world(t)$, which might have been expected. This allows us to let $\xi(t+1)$ and $world(t+1)$ depend on the whole past rather than just the present. (It would also allow equations expressing dependence on the future, although such equations would be consistent only when subjected to rather strong conditions.)

We have written part of the reservation program in this form, but it isn't yet clear enough to include in the paper.

The second approach uses separate functions and predicates with time as an argument. In this respect it resembles Algol 48.

$$
\begin{array}{ll}
(\forall psgr\ flt\ t) & (input\ t = \mathbf{request\ make\ commitment}\ admit(psgr, flt, ?seat) \\
& \qquad \wedge \neg \mathbf{holds}(t, full\ flt) \\
& \rightarrow (\exists seat)(\mathbf{holds}(t, available(seat, flt)) \wedge \\
& \quad \mathbf{arises}(t, \mathbf{commitment}\ admit(psgr, flt, seat)) \\
& \quad \wedge \mathbf{outputs}(t, \mathbf{promise}\ admit(psgr, flt, seat)))).
\end{array}
$$

Note the use of *admit* with a variable $?seat$. We may suppose that *admit* actually has a large set of arguments with default values. When an $admit(\ldots)$ expression is encountered, it is made obvious which arguments are being given values by the expression and which get default values. The signals for this are the name of the variable or an actual named argment as in Common Lisp. In the present case, the compiler will have to come up with a way of assigning a seat from those available.

$$
\begin{array}{ll}
(\forall psgr\ flt\ t) & (input\ t = \mathbf{query\ exists\ commitment}\ admit(psgr, flt) \\
& \rightarrow \mathbf{outputs}(t, \mathbf{if\ exists}(t, \mathbf{commitment}\ admit(psgr, flt))\mathbf{then} \\
& \mathbf{confirm}(\mathbf{commitment}\ admit(psgr, flt, seat))\ \mathbf{else\ deny}(\mathbf{input}\ t))
\end{array}
$$

$$
\begin{array}{ll}
(\forall psgr\ flt\ t) & (\mathbf{input}\ t = \mathbf{request\ cancel\ commitment}\ admit(psgr, flt) \\
& \rightarrow \mathbf{revoke}(t, \mathbf{commitment}\ admit(psgr, flt))
\end{array}
$$

$$
\begin{array}{ll}
(\forall t\ x)(& \mathbf{exists}(t, \mathbf{commitment}\ x) \equiv (\exists t')(t' < t \wedge \mathbf{arises}(t', \mathbf{commitment}\ x)) \\
& \wedge (\forall t'')(t' < t'' < t \rightarrow \neg \mathbf{revoke}(t'', \mathbf{commitment}\ x)))
\end{array}
$$

$$
\begin{array}{ll}
(\forall psgr\ flt\ t) & (\mathbf{input}\ t = \mathbf{request}\ admit(psgr, flt) \\
& \rightarrow \mathbf{if\ exists}(t, \mathbf{commitment}\ admit(psgr, flt) \\
& \qquad \mathbf{then\ outputs}(t, command(agent, admit(psgr, flt, seat)) \\
& \qquad else\ \mathbf{outputs}(t, command(agent, \mathbf{don't}\ admit(psgr, flt))))
\end{array}
$$

Asserting that certain outputs occur and that certain propositions hold doesn't establish that others don't occur. Therefore, the program as given is to be supplemented by circumscribing certain predicates, namely **arises**, **outputs** and **revoke**.

15 Remarks

Proving that a program fulfills its commitments seems to be just a matter of expressing a commitment as making sure a certain sentence is true, e.g. that the passenger is allowed on the airplane. In that case, proving that the program fulfills its commitments is just a matter of showing that the sentences expressing the commitments follow from the sentence expressing the program. The problem now is to decide what class of sentences to allow as expressing various kinds of commitments. If the commitments are to be externally expressed as promises, then they have to belong to the i-o language.

Commitments are like specifications, but they are to be considered as dynamic, i.e. specific commitments are created as the program runs. It makes sense to ask what are the program's commitments when it is in a given state. Indeed some programs should be written so as to be able to answer questions about what their commitments are.

An Elephant interpreter need only match the inputs against the program statements with inputs as premises. It then issues the outputs, performs the actions and asserts the other conclusions of the statement. The circumscriptions should not have to be consulted, because they can be regarded as asserting that the only events that occur are those specified in the statements. Here the situation is similar to that of logic programming. The circumscriptions *are* used in proving that the program meets its specifications, e.g. fulfills its commitments.

There is a theorem about this that remains to be precisely formulated and then proved. Making it provable might involve some revisions of the formalism.

Many kinds of human speech act are relative to social institutions that change. For example, a challenge to a duel in societies where dueling was customary was not just an offer. It generated certain obligations on the part of the challenger, the person challenged and the community. Suppression of dueling was accomplished partly by intentional changes in these institutions. The exchange of speech acts among computer programs will often involve the design of new institutions prescribing the effects of speech acts. For example, the kinds of external obligation created by business promises is partially specified by the Uniform Commercial Code that many states have enacted into law. Programs that make commitments will have to be specified in some way that corresponds to something like the Uniform Commercial

Code. One point is to be able to prove that (subject to certain assumptions) the programs do what is legally required. Another is that the effects of commercial speech acts should be defined well enough for programs to keep track of the obligations they have incurred and the obligations incurred to them. The simplest case is keeping track of the effects of the speech acts on accounts receivable and accounts payable.

Perhaps we will need three levels of specification, internal, input-output and accomplishment. Internal specifications may involve computing a certain quantity, regardless of whether output occurs.

Some communications among parts of a program may also usefully be treated as speech acts.

We may need to consider joint speech acts such as making an agreement. For some purposes an agreement can be considered as an offer followed by its acceptance. However, we may know that two parties made an agreement without knowing who finally offered and who finally accepted.

It seems that (Searle and Vanderveken 1985) improves on (Searle 1969) in distinguishing successful and non-defective performance of speech acts. It isn't clear whether these distinctions will play a role in computer speech acts. Perhaps the logic of illocutionary acts proposed in that book will be useful in writing specifications and proving that programs meet them.

Writing this paper began with the simple notion of a program that makes commitments and the notion of proving that it fulfills them. Then it became interesting to consider speech acts with more elaborate correctness conditions. However, we expect that the simple cases will be most useful in the initial applications of Elephant 2000 and similar languages.

Dorschel (1989) proposes certain conditions for the "happy" performance of directive speech acts, e.g. orders, promises and declarations. It turns out we want some of them and not others. By this I mean that when someone verifies an Elephant program, he will want to show that some of Dorschel's conditions are satisfied and not others.

For example, Dorschel proposes that a promise being fulfilled necessitates that making the promise be a cause of the act that fulfills it. The Elephant programmer in verifying his program need not show that the promise will be fulfilled because it was made. It is enough that he show it will be fulfilled. On the other hand, Dorschel proposes to require that an order is properly made only if the speaker has authority to give the order. We'll surely want that—especially for purchase orders, which include both an order component and a promise component.

These considerations illustrate our right to pick and choose among the concepts proposed by philosophers.

16 Acknowledgements

I am indebted to Vladimir Lifschitz, Leora Morgenstern and Carolyn Talcott for useful suggestions.

Support for this work was provided by the Information Science and Technology Office of the Defense Advanced Research Projects Agency.

17 References

Austin, J. L. (1962): *How to Do Things with Words*, Oxford.
Dorschel, Andreas (1989): "What is it to Understand a Directive Speech Act?", *Australasian Journal of Philosophy*, Vol. 67, No. 3, September 1989.
Francez, Nissim and Amir Pnueli (1978): "A Proof Method for Cyclic Programs", *Acta Informatica* 9, 133-157.
Francez, Nissim (1976): *The Analysis of Cyclic Programs*, PhD Thesis, Weizmann Institute of Science, Rehovot, Israel.
Francez, Nissim (1978): "An Application of a Method for Analysis of Cyclic Programs", *IEEE Transactions on Software Engineering*, vol. SE-4, No. 5, pp. 371-378, September 1978.
Grice, Paul (1989): *Studies in the Way of Words*, Harvard University Press. This is a collection of his papers.
McCarthy, John (1963): "Towards a Mathematical Theory of Computation", in Proc. IFIP Congress 62, North-Holland, Amsterdam.
McCarthy, John (1967): "Correctness of a Compiler for Arithmetic Expresions" (with James Painter), *Proceedings of Symposia in Applied Mathematics, Volume XIX*, American Mathematical Society.
McCarthy, John (1979a): "Ascribing Mental Qualities to Machines" in *Philosophical Perspectives in Artificial Intelligence*, Ringle, Martin (ed.), Harvester Press, July 1979.
McCarthy, John (1979b): "First Order Theories of Individual Concepts and Propositions", in Michie, Donald (ed.) *Machine Intelligence 9*, (University of Edinburgh Press, Edinburgh).

McCarthy, John (1982): "Common Business Communication Language", in *Textverarbeitung und Bürosysteme*, Albert Endres and Jürgen Reetz, eds. R. Oldenbourg Verlag, Munich and Vienna 1982.
McCarthy, John (1986): "Applications of Circumscription to Formalizing Common Sense Knowledge" *Artificial Intelligence*, April 1986
Newell, Allen (1982): "The Knowledge Level," *Artificial Intelligence*, **18**, 87-127.
Searle, John R. (1969): *Speech Acts* Cambridge, Eng., Univ. Press.
Searle, John R. (1984): *Minds, Brains, and Science*, Cambridge, Mass. : Harvard University Press, 1984.
Searle, John R. and Daniel Vanderveken (1985): *Foundations of Illocutionary Logic*, Cambridge, Eng., Univ. Press.

This draft of /u/jmc/w93/elephant.tex TEXed on 1998 Nov 24 at 2:49 p.m..
This file originated on 10-Jun-89

/@steam.stanford.edu:/u/jmc/w93/elephant.tex: begun 1993 Mar 2, LATEXed 1998 Nov 24 at 2:49 p.m.