The Dialogue Designing Dialogue System

Tai-Ping Ho

Department of Computer Science
California Institute of Technology

5137:TR:84

THE DIALOGUE DESIGNING DIALOGUE SYSTEM

Thesis by

Tai-Ping Ho

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

Computer Science Department

California Institute of Technology

Pasadena, California

5137:TR:84

1984

(Submitted May 17, 1984)

敬獻 家父母

何志先生夫人

Acknowledgments

I wish to thank my adviser, Professor Frederick B. Thompson, whose guidance, philosophy and encouragement made this thesis possible.

I wish to thank Professor Bozena H. Thompson, whose linguistics courses made it possible for me to walk into the world of natural language. I also wish to express my grateful appreciation to Professor James T. Kajiya for his valuable comments and suggestions on this thesis.

Thanks are also due to Caltech, as a whole, that provided an ideal academic environment throughout my stay. In addition, I wish also to thank my former officemate, Kwang-I Yu, for his helpful discussions on my work; and Mr. and Mrs. J. L. (Paul) Hurschler for their friendship.

And most of all, I wish to thank my parents, my wife and her parents for their love and support.

For the financial support for this thesis, I wish to thank the Hewlett Packard Company, Desktop Computer Division.

Abstract

This thesis presents an interactive system, the Dialogue Designing Dialogue System, that integrates natural language programming of user dialogues with a natural language system, the ASK system. This interactive system satisfies the basic criteria of a general programming language.

The system presented in this thesis may be referred to as a "meta-dialogue" system. Using this meta-dialogue, the user implements domain specific dialogues which he and others can then use, providing highly succinct and efficient interfaces for interaction with the computer.

The system combines the use of a syntax-directed and a semantic-directed system, which gives the user flexibility in specifying additional capabilities, and thus in turn gives the system itself a much broader domain of application. Further, the system integrates natural language programming, dialogue directed user interface, underlying data base, and text handling capabilities, so that it does not require users to have programming background in order to establish an application system for themselves.

Table of Contents

CHAPTER 1

INTRODUCTION

1.1: About the Thesis

1.1.1: Objectives of the thesis

This thesis presents an interactive system that integrates natural language programming of user dialogues with a natural language system. This interactive system satisfies the basic criteria of a general programming language.

A user interface system is usually designed to support user interaction relative to a specific task, for example, word processing, medical diagnosis, graphic design. It allows the user to interact with the computer in a convenient way but to a rather limited extent. Here we emphasize a user interface system not limited in its domain, but one that allows a user's decision structure to be specified in a much more flexible format. It supports the user's capability to design an application specific to his/her special interest by himself*.

In using an interactive system, the quality of the user interface determines the success of the system's man-machine communication. A natural language system environment goes a long way toward a habitable interface. However, for repetitive tasks, it may require tedious interaction. A knowledgeable dialogue, in a natural language system environment, that can carry out such a task using only a few user inputs provides higher productivity. An interactive system should be easily extendable to include such dialogue systems. A user's own knowledge as well as the user's decision structure for such an application can be represented best by the user himself under the system's guidance.

The system presented in this thesis may be referred to as a "meta-dialogue" system. Using this meta-dialogue, the user implements domain specific dialogues which he and others can then use, providing highly succinct and efficient interfaces for interaction with the computer.

Work on the meta-dialogue system of this thesis was preceded by the design of two specific Dialogue Systems for forms processing and bulk data input process-

*Throughout this thesis, the pronouns he, his, himself are used to refer to both sexes; this choice was made purely on the basis of the minimum of letters in "he".

ing. This experience has suggested the fundamentals of the meta-dialogue system. Therefore, based on this successful experience, the integration of natural language programming and friendly user interface became our research focus, resulting in the development of this meta-dialogue system.

1.1.2: Contributions of the thesis

The thesis addresses the problem of the design and the implementation of the meta-dialogue system. It is an interactive system, using non-procedural methods to access knowledge structures. It is based on a natural language system, the ASK System (see Chapter 3), which is the "abstract machine" on which this meta-dialogue system runs. The concrete result of this thesis is the Dialogue Designing Dialogue System, which is now an integral part of the ASK System.

The main contribution is that the system integrates natural language programming, dialogue directed user interface, underlying data base, and text handling capabilities, so that it does not require users to have programming background in order to establish an application system for themselves. The user interface for the underlying ASK System is natural language. With the addition of the Dialogue Designing Dialogue, of this thesis, natural language becomes only the primary language for ASK. The user can now design and implement his own interfaces and complex semantic responses, involving data base manipulations and report generation, tailoring his interactions to specifically fit his immediate needs.

1.1.3: Organization of the thesis

This thesis consists of ten chapters. The remainder of the first chapter contains an overview of the basic criteria of a programming language, natural language programming and the user interface, some terminology, and a review of what has been done by others.

Chapter 2 gives some basic notions involved in a Dialogue System. A simple computer dialogue will illustrate these notions. The chapter mainly discusses some properties (e.g., prompt, response, field, node, etc.) used in the Dialogue Designing Dialogue System.

Chapter 3 introduces relevant parts of the ASK System environment (e.g., knowledge base categories, syntax-directed grammar, etc.), the prefix/prompt mechanism, the parsing and the evaluation, and the the Bridge procedure.

Chapter 4 to Chapter 7 describe the design of the Dialogue Designing Dialogue.

Chapter 8 consists of two examples to demonstrate how each application Dialogue System proceeds in the design phase and the run phase.

Chapter 9 discusses the facilities for examining, editing and deleting a Dialogue System that has been designed by the Dialogue Designing Dialogue.

Finally, concluding remarks on the system are in Chapter 10. Comparisons are made between the programming language criteria, stated in Chapter 1, and the system presented in this thesis. Suggestions for future work are given.

The design phase protocol with all help messages is included in the Appendix A. A third example of designing and using a Dialogue System is included in Appendix B.

1.2: Criteria of a Programming Language

A programming language is a notation with which people can communicate algorithms to computers and to one another. Since high level languages are preferable to machine or assembly languages in regard to the aspects of ease of understanding, naturalness and efficiency of use, we concentrate on the nature of a high level programming language. While there is no widely accepted definition of a programming language, the following are the basic criteria that we believe a good programming language should satisfy:

- data type (data structure): basically there are four standard (unstructured) types: integers, reals, characters and booleans.

  A good programming language must have these. The extended types may include structures like pointers, lists, trees, arrays, queues, stacks, strings, graphs and records. The selection of these extended types for a given programming language depends on the nature of the expected application of that language.

- operation set: the set of operators that function on the specified set of data types;

  Basically it includes the operators:

  - for real and integer operands, e.g., +,-,*,/;
  - for comparison (to return a boolean value): relational: <,=,>, etc; logical: and,or,not;

- for string-wise (the extension of characters): concatenation, substring (e.g., character selection).

The data type and its associated operations constitute a mutually inclusive integral pair rather than two independent pieces in the design of a programming language. There must exist some means in the operation set to perform the creation and deletion of instances of each data type. If a constant is considered a type, it should only be used globally and not be modified. Operations are primitively defined if and only if their associated data types are primitively defined. Problems of either redundant types or insufficient operations would otherwise occur.

- statement: either simple or compound, is for:
  - computation (e.g., assignment, by copying or by sharing)
  - sequence control (e.g., call)
  - structural (e.g., end)
  - declaration (e.g., the declaration of type, variable)
  - i/o statement (e.g., format)
- parameter passing: the choice of calling by address (i.e., call by reference), by value or by name. Note that each programming language, according to the nature of its application, may adopt one or more methods of the parameter passing. For instance, Pascal uses call by value and call by address, Algol uses call by value and call by name (where Algol 68 uses call by value only), and (most) Fortran uses call by address only [Wulf81].
- storage management for storage allocation:
  - static: allocated at compile time
  - dynamic: allocated at run time. It has two methods: the stack for handling recursive functions, and the heap for handling frequently created, destroyed and modified data whose size varies as the program is running (e.g., for garbage collection). Storage and associated storage management may have special structures, such as an available space list for list processing, or stack or queue processing.
- recursive functions: for a procedure that calls itself, directly or indirectly. The use of recursion (e.g., to traverse a tree) often permits a more natural, lucid and concise description of algorithm than would be possible without recursion.

Issues of variable binding (thus, the parameter passing) and consequently, storage management, require careful attention.

- program control structure:
  - branch constructs (e.g., goto, conditional branch),
  - selection constructs (e.g., case),
  - iteration constructs (e.g., repeat, while, for),
- variables and scope rules: global, local; scope of program: modularity, block structure.
- language support: e.g., connection to I/O devices (including graphics), file handling, data base access facility, a host of other library programs, etc.

These programming language criteria are stated here as they apply to traditional programming languages. In this thesis, the natural language programming system being presented is, comparatively, a very high level language, yet must satisfy these same criteria to guarantee all of the functions required. In Chapter 10, we will return to examine the above list of criteria as it applies to our meta-dialogue system.

1.3: Natural Language Programming

A major research direction in computer science is aimed at reducing the increasingly serious programming bottleneck and improving currently available software facilities. Therefore natural language programming systems which can shift the growing burden of programming to the computer itself have been receiving a great deal of attention. They allow users to express their problems to the computer in problem oriented languages, instead of specifying how to solve the problem in procedure oriented languages. Query languages for data base systems are a good example of problem oriented languages.

For instance, the LADDER system uses natural language as a query language. However, since the system uses a "semantic" grammar which implies that its grammar (therefore the corresponding semantics, interpreted by the LIFER system [Hendrix77]) is tightly coupled to the semantic concepts represented in the data, any application other than to the Navy database requires the writing of a new grammar. In contrast, the Dialogue Systems discussed in this thesis, including the meta-dialogue system that is the result of this thesis, are based on a "syntax"

grammar natural language system. In addition, we combine the use of a syntax-directed and a semantic-directed system. While in using the Dialogue Designing Dialogue, the designer can specify a wide range of syntactic forms which are used to direct and control the resulting dialogues, in this case we use "syntactic-directed" approach. *While the user runs the Dialogue System, since its decision structure* depends on the semantics of the responses occurring in the resulting dialogues and the underlying data base, we then use the "semantic-directed" approach (also referred to as "state driven"). It is because of this mixture that the system gives the user flexibility in specifying his intention, and thus in turn gives the meta-dialogue system itself a much more broader domain of application.

The non-procedurality, high level operations and abstract data structures become the main features in the problem oriented language. That is, in a very high level language, a task is described in terms of the desired results and is independent of any specific way of accomplishing the task. Some natural language systems translate the source language into an intermediate language program which can then be executed. The NLPQ system [Heidorn75], for instance, is such an example. It uses natural language dialogue to specify part of a simple queuing-simulation problem and then the system generates a GPSS program for solving the problem. The system we present is of this form. It will not synthesize the problem specification into a target programming language in the usual sense. Instead, the user's problem specifications (including decision structure) for a Dialogue System are transformed into a corresponding Dialogue Data Structure. *No effort is spent on program* optimization, in contrast to most of these systems. Efficiency is obtained since the translation of the user inputs is, abstractly, into a very high level virtual language whose operations (e.g., access to data records) have already been highly optimized. Incomplete information and inconsistent information will also be handled properly.

1.4: Terminology

The following terminology will be used in the remainder of this thesis.

Dialogue: A series of interrelated computer-user interactions.

Dialogue Element: a single computer-user interaction that is part of a dialogue.

Dialogue System: An application system that, once initiated by a user, proceeds in a structured way to hold a dialogue with that user. We will call such a dialogue an instance of the Dialogue System.

Meta-Dialogue System: A Dialogue System which is used to design or generate other Dialogue Systems.

Dialogue Designing Dialogue System: The meta-dialogue system of this thesis.

1.5: Comparison of the Results of This Thesis With Related Work in the Literature

In this section we will be making two kinds of comparisons:

other Dialogue Systems	VERSUS	Dialogue Systems designed using the Dialogue Designing Dialogue System
other Meta-Dialogue Systems	VERSUS	the Dialogue Designing Dialogue System

The distinction between these two kinds of comparisons should be kept in mind.

1.5.1: Other Dialogue Systems (with the exception of Expert Systems)

The use of dialogues to provide a friendly communication interface has been increasing. In this area, we found that most of the systems using dialogues do not really allow users to express their intent beyond simple responses: a single word or number. For example, the ambient quality monitoring system [Halpern81] is one such system.

The TEAM system [Grosz83], a natural language interface system, is an advanced one. This is a special purpose dialogue system for attaching a new data base "back end" onto the natural language "front end" called Dialogue. It builds this interface with a given database by conducting a dialogue [Robinson82] with a database expert who provides information about files, fields in the database, and the database query language. To translate a natural language query into a database query, it first translates the query into a "logical form" in the acquisition component, and then translates the logical form into a formal database query in the data-access component. The system separates information about the language, the application domain and the data.

Some systems employ dialogues in a more flexible way. Users can express their intent in a natural language statement which will be interpreted by the system. For example, the SCHED system [Heidorn78] is one such dialogue system.

The SCHED system allows a user to schedule a meeting using a dialogue approach. The AYPA system [Gershman81] helps users to search the yellow page information concerning automobile parts and related objects by transforming a user's responses during a dialogue into a database query. Another example in [Marburger81] connects a dialogue to a scene analysis system (traffic monitoring at a busy intersection) to study the interaction between natural language and image understanding, especially the verbal description of motion. These systems are, of course, specific systems designed for specific tasks. Dialogue processing is tightly coupled with their semantics, therefore the applications for such systems are just over specific domains, as their titles express.

How do these systems compare with the Dialogue Systems that can be defined using the Dialogue Designing Dialogue? A Dialogue System can be analyzed as having three parts: (a) the "dialogue" aspects by which the system elicits information from the user, (b) the "decision structure" aspect which decides what actions should be taken as a result of these inputs, (c) the "action" aspects by which it carries out a resulting complex set of actions. The Dialogue Designing Dialogue could be used to implement each of the Dialogue Systems discussed above as far as (a) and (b) are concerned. The actions available to a Dialogue System designed by the Dialogue Designing Dialogue are querying, updating, extending a database and communicating via texts. Some of the Dialogue Systems mentioned above can be completely implemented using the Dialogue Designing Dialogue when the actions to be taken are of these kinds, for example, SCHED and AYPA. Others would require the addition of specialized capabilities. For example, for the traffic monitoring system, both a television camera input and image processing capabilities would be needed. In the case of TEAM, communication capability to access a foreign data base would be required (see, however [Papachristidis83]).

We know of no research on meta-dialogue systems other than in the expert system area. We know of no meta-dialogue systems to compare with the Dialogue Designing Dialogue.

1.5.2: Expert systems

Good examples of advanced user interfaces are the expert systems (or, knowledge engineered systems) [Barr82]. They are viewed as "intermediaries between experts who interact with the system in **knowledge acquisition** mode, and human users who interact with the systems in **consultation** mode." [Feigenbaum77] The expert systems are built through the painstaking interaction of domain experts,

who may find it difficult to articulate all of their knowledge, and system designers [Barr81]. The resulting system is thus built with particular and profound knowledge of the domain. The user, who also has some of this knowledge, can expect that the system will be able to interpret his queries in terms of its deeper knowledge, assisting him in decision making.

The MYCIN system [Shortliffe76], for instance, acts as a medical consultant, aiding in the diagnosis and selection of therapy for patients with bacteremia or meningitis infections. It carries on an interactive dialogue with a physician and is capable of "explaining" its reasoning. Another example is the PROSPECTOR [Duda79] system which assists geologists working on certain problems in "hard-rock" mineral exploration. The user supplies information about the most significant features of his prospect and the system will help to confirm the best matching model.

However there exist disadvantages in the use of such specific systems. First, a given system is designed for one special domain only, therefore its application is inherently restricted. Second, the user can only respond according to the system's needs in computer-questions/user-answers style. This implies that he can just "passively" supply the answers requested, and has no means to "actively" express his own intention of how to treat some data or how to do some aspect of the task, if his ideas deviate from those programmed into the system by some distant experts. Therefore he has no way to update the old knowledge. Third, since the system is likely to have been designed by experts other than the practitioners who use the system, when some old information in the system needs to be updated, or deleted, or new information is to be added, the delay in communicating with the designer to "fix" the system is inevitable. The delay may take days, weeks, months or forever [Brooks78].

Expert systems are radically different from dialogue systems designed using the Dialogue Designing Dialogue in that expert systems, by and large, have strong capabilities for drawing inferences from user inputs and making use of these inferences in the actions they support. This is not the case for dialogue systems designed using the Dialogue Designing Dialogue. More will be said about this difference below. By using the decision structure capabilities provided by the Dialogue Designing Dialogue, weak inference capabilities can be achieved. There are many such "small size" local knowledge tasks to be done which could utilize the computer, and a user would like to do so if he could only arrange for such "mini expert systems" as he foresaw the need.

The difficulty is that each user's environment has its own idiosyncrasies, and

so standard packages of "services" never completely meet his immediate needs. We therefore would like to have a system for experts to use to satisfy such needs. The system only requires a user to be expert in his application domain but does not require him to be expert in "building" his expertise into the system. In contrast to an expert system, such a system for experts is not created with local knowledge on any particular application. Instead, it has its own global knowledge to "elicit" information from the designer to build into itself any particular knowledge structure. In other words, in addition to the information retrieval and decision making as in an expert system, the system is specifically designed to be updated by the user, who may indeed be the designer. There is no unpleasant delay for communication with a distant designer. In addition, since our system will be integrated with the wider system environment, therefore all the constituents of this wider system will be available, enriching the interactive capabilities of the system significantly.

There are several meta-dialogue systems for designing expert systems. Expert systems are, superficially, much like the dialogue systems that can be designed using the Dialogue Designing Dialogue System. Thus systems that are used to build expert systems [Hayes-Roth83] have similar characteristics to that of the Dialogue Designing Dialogue in the sense that all such systems provide tools for designers to build various interfaces for users to solve problems. Among these, the following two representative systems: EMYCIN [van Melle80] and ROSIE [Fain81] will be discussed. A third such system is the "knowledge acquisition" subsystem of MYCIN, TEIRESIAS [Davis76], which helps expert physicians expand or modify the rule base, which contains hundreds of production rules representing human-expert-level knowledge about the domain.

There are two major differences between expert systems and expert system building systems, on the one hand, and database systems and the Dialogue Designing System, on the other hand. The first is that, in the case of an expert system, with its knowledge base consisting of production rules, the expert building system is just a mechanism for building the knowledge base, for adding new production rules; whereas in our case, the knowledge base exists as a separate entity, the ASK system providing many ways to build and update it, and the Dialogue Designing Dialogue providing the way to add more productive means for the user to interact with the data base, namely dialogues, including succinct means of adding new data. The second major difference is in the basic nature of the algorithm for processing queries. In the data base case, access to the data base is controlled by the syntactic structure of the query sentence in the manner of a syntax directed interpreter, thus makes

use of only local aspects of the data base. In the expert system case, this algorithm is driven by a rule matching control structure which requires a more global search of the knowledge base. It is in terms of this second difference that one can see the impact of the meta-dialogue system. In the Dialogue Designing Dialogue case, the resulting dialogue is closely related to a definition. In a definition, a more extensive syntactic construction is abbreviated by a less extensive one. A dialogue is designed to replace a much longer free form interaction by a much more succinct interaction. Thus the role of the Dialogue Designing Dialogue in the context of the ASK System is completely different from the expert system building systems, even though they may superficially appear to be similar.

This superficial resemblance is most noticeable in the case of ROSIE. In using ROSIE, the two striking features are its English-like syntax, which facilitates the creation and manipulation of the ROSIE database; and its pattern-matching capability, which provides part of the ability to monitor or control off-site computation which in turn provides the interrupt mechanism to accept new data immediately. Its database actually keeps track of all ROSIE English sentences; the insertion or the deletion of a data is in terms of the entire sentence. It is hard for a ROSIE model to add or modify its own rules. It is also impossible to change its control structure to fit new problem domains.

Another meta-system for building expert systems is EMYCIN. EMYCIN is a general purpose system that evolved from TEIRESIAS, which in turn was an attempt to add to MYCIN a capability to add production rules. Thus some features (e.g., metarules and the how/why explanation facility) are from TEIRESIAS. The strength of EMYCIN lies in the realm of human engineering in building an expert diagnostic system. Its principal method of control is backward-chaining, with limited provision for forward-chaining. In using EMYCIN, just as in ROSIE, the designer builds both the resulting Dialogue System and also at the same time the underlying knowledge base. The result of using EMYCIN is a complete Dialogue/Knowledge Base, i.e., the single, self contained Expert System, but there is nothing else. It may be used in a mode where it extends an existing Expert System previously designed with its use, but the above distinction still holds.

CHAPTER 2

BASIC NOTIONS

2.1: Additional Terminology

Initiating Statement of a Dialogue System: The unique statement used to initiate an instance of a Dialogue System by a user.

Dialogue Data Structure: a data structure of a Dialogue System, produced by the Dialogue Designing Dialogue System. The Initial Dialogue Data Structure is produced by the design phase of the Dialogue Designing Dialogue System, and is stored on a disk. Upon the initiation of a dialogue, a copy of its Dialogue Data Structure is reconstructed in main memory, and is dynamically modified as the Dialogue System progresses. All the information necessary for the continuation of the Dialogue System is retained in this structure.

Modes: Two modes exist: the **ASK Mode** and the **Dialogue Mode**. The **Dialogue Mode** occurs only when a Dialogue System is currently in control of the user dialogue. All other situations are considered to be in the **ASK Mode.** Typically, switching from ASK Mode to Dialogue Mode happens when a user types the initiating statement of a Dialogue System; from Dialogue Mode to ASK Mode when the Dialogue System has completed the current dialogue.

User and Designer: A **designer** is a person who clearly understands his application area and how the computer can be used to support those users who wish to use the computer in their work in that area. The designer seeks to achieve this support by designing a Dialogue System. A **user** can accomplish some aspect of his task by applying this Dialogue System. Thus, a designer functions as an application programmer for his group of users. The designer, in many cases, will also be a user. Indeed, it is our intent to so facilitate the process of design that users who have repetitive tasks to be done can themselves, without delay, design their own Dialogue Systems and immediately make use of these systems.

Each Dialogue System is accomplished through two phases: a **design phase** and a **run phase**. During the design phase, a designer builds an application Dialogue System for his users. When a user subsequently makes use of this Dialogue System to carry out a particular task, he is said to be in the run phase of this Dialogue

System. Although the designer and the user of a Dialogue System may be the same person, we will use "designer" when referring to the design phase and "user" when referring to the run phase.

2.2: A Simple Example of a Dialogue

Here is an example of a simple dialogue for adding items to a bibliography. The protocol includes:

- an initial query (i.e., ">What are books?"),
- an instance of a simple Dialogue System, initiated by "new bibliography item",
- a final query (i.e., ">Who is the author of each book?"), illustrating the effect of this dialogue.

(The symbol ">" initiates lines in which the computer will wait for user input. User inputs are in italics, computer responses are in typewriter fonts.)

```
> What are books?
 Aspects of the Theory of Syntax
 Conceptual Information Processing
> new bibliography item
>title:  The Natural Language of Interactive Systems
>author:  H. Ledgard
>key word:  interactive languages
>key word:  programming
>key word:
 The new bibliography item The Natural Language of Interactive
 Systems has been entered.
> Who is the author of each book?
 book                                        author
 Aspects of the Theory of Syntax             N. Chomsky
 Conceptual Information Processing           R. Schank
 The Natural Language of Interactive Systems H. Ledgard
> .......
```

Comparing the final query with the initial one, we observe that a bibliography item (i.e., "The natural Language of Interactive Systems" and other associated data) has indeed been added into the database. This Dialogue System contains three different dialogue elements. The user is initially in ASK Mode. When a user types in a valid initiating statement for a Dialogue System, he is then in the Dialogue

Mode. After the user responds, the system will continue to give the next prompt and wait for a response. The result of running the Dialogue System may involve the modification of the database, text handling, etc. In this simple example, words are added to the vocabulary and data recorded in the database. When the dialogue is finished, the user is returned to ASK Mode.

2.3: Prompt and Response

There are three parts in each dialogue element within a dialogue. The first part, initiated by the symbol ">", is a "prompt" message, issued by the Dialogue System, calling for a user's reply. We hereafter refer to this message as a **prompt**. The second part of the interaction is the user's "response" to this prompt. We hereafter refer to this reply as **response**. For instance, in the previous example, the message ">author: " is the prompt, and the reply "H. Ledgard" is the response. The third part will be referred to as the **decision structure** of the dialogue element. It is this decision structure, using the response and also responses to previous dialogue elements, that decides on the next dialogue element to be activated. This third part is, of course, not directly visible to the user.

A prompt may be one of the following:

1) declarative: asking the user to give information; e.g.,

```
>Transfer to what node?
```

2) selective: asking the user to select information; e.g.,

```
>What type of user response is expected? nu(mber), no(un phrase),
 ti(me), te(xt), se(ntence) or st(ring)?
```

In the latter case, a string of distinguishable leading characters is expected. For example, the response, either "nu" or "numb", is acceptable for "number".

For each prompt, the response can be classified as one of the following:

- a responsive response: giving the information desired;
- a non-responsive response: e.g.,

  ```
  >port of entry:several
  ```

  this is usually considered as a "bad" response;
- a nil response, that is, only a carriage return. Usually this will be interpreted as using the system-provided default value.
- a service response: three services may be requested, namely:

- a "help" response, which allows the user to get more detailed information concerning the nature of the response that is desired.
- a "wait" response, which allows the user to leave the dialogue for a time, and then to resume the suspended Dialogue System at the same dialogue element where he left it. This may happen, for example, if the user finds, in the course of a dialogue, that he needs additional information from the underlying knowledge base or wishes to add new information or vocabulary.
- an "exit" response, which allows the user to escape from the current Dialogue System.

2.4: The Concept of a Field

In a Dialogue System, a **field** corresponds to the notion of a local variable in programming languages. It is a primitive structure for communicating information between the dialogue elements in the Dialogue System. A field is established by the dialogue designer specifying its field number, its field type (equivalently, its ASK part of speech), and assigning it a value. A field is referred to by a field symbol: "<n>", where n is its field number. For example, one may want to use field 3 and field 7 to represent the author and title of a book, and then indicate a database update by the statement: "The author of <7> is <3>."

The value of a field is either given directly by user response, during run time, or evaluated from an expression specifically assigned to the field. For example, suppose field <3> is to hold the author's name, then the response "N. Chomsky" to the prompt ">author: " becomes the value of field <3>. On the other hand, suppose field <5> is to hold the tax of an item whose price is represented by field <9>, then the value of field <5> will be obtained only after the value of field <9> is known in the specified expression. For example, field <5> would be assigned to the expression "0.065 * <9>". The assignment of a value to a field is typically made in an action statement (see section 5.1.3.2).

Any field may also function as a stack or queue. Just as a field may be assigned a value by an action statement, a value may be pushed on top of a field (treating it as a stack), pushed on the bottom of a field (treating it as a queue), or popped off of a field. (see section 5.1.3.2 for specifics concerning the use of fields as stacks or queues).

2.5: The Concept of a Node

2.5.1: An interaction node

There are three prompts in the simple dialogue in section 2.2, where a nil response to the third prompt (i.e., >key word:) terminates the dialogue with the issuing of a terminal message. To the designer, each prompt can be considered as one **interaction node** (which we will often abbreviate to **node**). If we sequentially label those prompts in the simple dialogue with node numbers starting with one, then we get a node flow as in the following Figure 2.2:

```
            +-------+       +-------+       +-------+  nil
start ->    | node  |  ->   | node  |  ->   | node  |  ->  end
            |   1   |       |   2   |    ->|   3   |>-
new         +-------+       +-------+    |  +-------+   |  The new
biblio-     title:          author:      |  key word:   |  biblio-
graphy                                   |              |  graphy ...
item                                      -----<------    entered.
```

Fig. 2.2. A simple dialogue node flow, where node 3 has a loop

In designing such a dialogue, one must be able to specify in a natural way such instructions as:

"Repeat node 3 until there is a nil response, then complete actions that have been indicated and terminate the dialogue."

"The response to node 2 is to be entered into the knowledge base as the author of the response to node 1."

These, and other similar instructions, should be tied to the appropriate nodes (and fields) in such a way that they will be activated at the time the dialogue is subsequently used. The Dialogue Designing Dialogue system provides just the tools for these tasks. The procedures for prompting for and gathering all these instructions and other properties of any node becomes the nucleus of the system.

2.5.2: Properties of a node

In the Dialogue Designing Dialogue, a node is the embodiment of a single dialogue element: the prompt, and the associated decision structure that accepts and acts on the user response to this prompt. An entire Dialogue System is viewed as a node flow process. In the flow of the dialogue, a node obtains control when the user has responded to the prompt corresponding to this node. Processing of this

node according to its internal decision structure results in issuing a new prompt and prefix, and thus in the transfer to a new node.

Each node should be equipped with the following properties:

- a node identifier
- a node prompt
- a node type
- a node response processor to process the various kinds of user responses: "help", "exit", "wait", nil, responsive and non-responsive
- access to responses that have been entered at previous nodes.

Consider each of these properties in turn. A node identifier is simply for distinguishing one node from another (a unique number as we have implemented it). A node prompt is a succinct message that conveys to a user an idea of the response that is expected of him, e.g., "author: ", "P(aper), B(ook) or A(rticle): ". The node type is the data type of the expected user response to this node; it may be a number, a noun phrase, a time phrase, etc.

2.5.3: Node response processor

What the node response processor is designed to do for the first three kinds of user responses, namely "help," "exit," and "wait," is rather obvious. A help message is provided as part of the design. The exit subprocessor allows the user to quit the design at any time. The wait subprocessor allows the user to temporarily leave the dialogue design, then on the command: "resume", go back to where it was put in recession.

The remaining three subprocessors: nil, responsive and non-responsive processors, are the central part of the node response processor. The nil response subprocessor deals with the case of a nil response, i.e., where a single carriage return key is pressed. The processing of a viable response of the correct type may involve a complex decision structure. The dialogue will also need to know what to do if the user's response is not understandable or not of the correct type; whether, for example, to add the new word to the vocabulary with its associated data structure or simply to reprompt possibly with a diagnostic message.

Finally, a node needs access to the responses that have been made by the user prior to reaching the given node. This requires careful record keeping during the

design process to ensure that all such responses which the designer anticipates at a given node will have been made available along all possible paths to that node. The concept of field was introduced for precisely this purpose.

The dialogue designer indicates the transition to another node in a node decision structure. The design process keeps track of which nodes have been designed and which nodes have been named in the transition parts of such a decision structure. When all nodes so named have been designed, the design of the application dialogue is finished. However, in using such an application dialogue later on, the dialogue may end at any node depending on the user's responses and the decision structure specified by the designer. The previous bibliography system, for instance, may very likely be finished at either a nil response to ">title: " or to ">key word: ", depending upon how it was designed.

CHAPTER 3

THE UNDERLYING ASK ENVIRONMENT

3.1 The ASK System

The host system for the Dialogue Designing Dialogue system, and its Dialogue Systems, is called the ASK System, A Simple Knowledgeable System [Thompson83a, Thompson83b, Thompson84]. It is a total system for the structuring, manipulation and communication of information. It is being implemented in an extension of the Pascal programming language. In this section, we will briefly introduce four relevant aspects in ASK: features of its knowledge base categories, syntax-directed grammar, list data structure, and global area.

ASK maintains its knowledge base in a semantic net. There are four types of nodes: Classes, Objects, Attributes and Relations. The objects are single entities, e.g., "Alamo", "John Jones", "Boston", also "budget memo", "letter to John". The classes are sets of objects, e.g., "ship", "city", "letter file". Attributes and relations relate one object to another, with the difference that attributes are single valued, e.g., "homeport", "father", while relations may be multiple valued, e.g., "cargo", "child", "comment". The following ASK English question exemplifies these types:

```
>Is Boston the home port of some ship whose cargo is coal and steel?
```

ASK is a syntax directed system. It contains grammar rules, and each rule is associated with a semantic procedure. When a user enters a sentence, it is parsed according to the grammar available and a tree structure is produced, associating the various grammar rules with the segments of the input to which they apply. A simple example is:

```
        Grammar Rule                                  Associated Semantic
                                                            Procedure
R1: <sentence> => "What is " <number> "?"                   OUT_PROC
R2: <number> => "(" <number> "+" <number> ")"               ADD_PROC
R3: <number> => "(" <number> "*" <number> ")"               MULT_PROC


      <sentence>..............................
              <number>........................
                         <number>..............
               <number>   <number>  <number>

      What is (3       * ( 5       + 7       ))?
```

This tree structure then is used to compose the associated semantic procedures to produce the desired response: "36" = OUT_PROC(MULT_PROC(3, ADD_PROC(5, 7))). The various semantic procedures are independent of one another, communicating via the arguments corresponding to their constituent phrases.

Rules of grammar are stored in the ASK system's dictionary. Each rule is specified in the format shown below.

```
RULE
.. general rewrite rule ..
execution-sequence     semantic-procedure
```

Where the execution-sequence indicates at what point in the processing the procedure is to be executed: during semantic processing as a postfix (POST) or prefix (PRE) procedure, or during syntax processing (SYN). Here is an example of this kind of specification for the above rule R2.

```
RULE
<number> => "(" <number> "+" <number> ")"
POST ADD_PROC
```

The ASK internal data structures are recorded in lists. Each list has the following format:

(list identifier, flag, f1, f2, f3)

where the list identifier consists of three characters which identify the type of the list, the flag records a number, and the type of the element f1, f2 or f3 is determined

by the corresponding character in the list identifier. For example, the character "l" points to a list, "p" points to a disk page, "s" points to a literal, etc. Thus, a list identifier "lsp" (with flag 2) implies the following list data structure:

(lsp, 2, list_pointer, literal_pointer, page_pointer)

The global area GLOBAL is a work space in the ASK system, and is protected from the list processor's garbage collector. The global variable GLOBAL^[current_dialogue] is preserved for the use of dialogues in maintaining local contexts in the form of lists. A second global variable GLOBAL^[dialogue_stack] is used by the wait-resume and call mechanism as a push-down stack.

3.2: Prefix/Prompt Mechanism

"Prompt" and "prefix" are strings that are used as follows. An OUT-response is in the format: (OUT, flag, lines to be output, prompt, prefix). The system, upon returning from a call to a semantic procedure and finding that an OUT-response was returned, first outputs to the user the lines indicated. It then starts a new line with the character ">", followed by the prompt (if any), then does a read, leaving the cursor just following the prompt. For example, (OUT,0,("aaa","bbb"),"more","ccc"), would result in:

```
 aaa
 bbb
>more_
```

Suppose the user responded:

```
 aaa
 bbb
>morexxx
```

Thus the read issued by the system would pick up the string "xxx". It would concatenate the prefix (if any) onto the beginning of this string, and deliver it to the sentence analysis portion of the system; in the case of this example, the string: cccxxx.

The purpose of this prompt-prefix mechanism is precisely to facilitate dialogues; specifically, to allow the implementor to ensure that when a specific semantic procedure gains control, it can be certain of the status of the user computer dialogue. This mechanism is typically used in the following way. Consider the following two rules:

```
RULE
<sentence> =>......
POST start_proc

RULE
<sentence> = "@ddd@" <noun_phrase>
POST end_proc
```

Suppose start_proc returns:

(OUT,0,("New article."),"Author: ","@ddd@")

Thus at the end of start_proc, the user sees:

```
New article.
>Author: _
```

and responds:

```
New article.
>Author: Smith
```

The system then passes to sentence analysis the string: "@ddd@Smith". Once "Smith" has been parsed to a noun phrase, the second of the above rules will apply, and the semantic procedure end_proc will be called. It will have been written with the anticipation that the input is a response to the prompt: "Author", and thus will know how to proceed. The reason that the roles of prompt and prefix have been separated is so that several prompts may lead to the application of the same rule, using a common prefix.

3.3: Evaluation and Parsing

The function "evaluate" evaluates a given string or phrase, returning the syntactic and semantic evaluation and an error code (similar to the LISP "eval"). Thus if the statement: "Scott is the author of Ivanhoe." is successfully evaluated, then the specified information will have been added to the database. If "Scott" were not in the vocabulary, or some other fault were detected, evaluate would return an appropriate error code. Similar to but rather simpler than the function "evaluate" is the function "parse_list". It parses an ASK system expression to construct a parsing graph without further evaluating that expression.

We use both "evaluate" and "parse_list" in different situations. While in the design phase, user input may very likely contain some fields which will not be instantiated until run time. However, the input should be examined to check its

legitimacy. Then we use "parse_list". In the run phase, when all fields have been instantiated with proper values, we use the function "evaluate" to obtain the final value. Illegitimate statements will result in diagnostics in both the design phase and the run phase.

3.4: The Bridge Procedure

Under a variety of conditions, the user may be expected to use words that the system has never seen before. This will be the case when adding new words to the vocabulary, or introducing definitions. The function Bridge, whose format is:

bridge (a character, boolean)

allows the application programmer to pick up such otherwise unrecognized input in a convenient form. It is a system function, called during syntax analysis, since it must modify the parsing graph so that sentence correction analysis procedures will not be called.

Bridge is used in the Dialogue Systems in the following way. Typically, the grammar rule associated with a dialogue element is of the form:

```
RULE
<sentence> => "@dddaction@"
SYN dddaction
```

Bridge is then used to obtain the user response to the prompt.

The boolean argument of Bridge is used to help determine the appearance of field symbols. If the user input contains field symbols, then the specification of the true boolean for the second argument will result in a linked list that separates field symbols and non-field-symbols, while a false will simply return one list element with no such field symbols separation.

In the Dialogue Systems, for example, if the second argument is set to true while in calling the function Bridge for the following dialogue element:

```
>Action: <3> is the author of <1>.
```

where <3> and <1> are field symbols, the result is a list of four elements containing successively: "<3>" as a field symbol, the string: "is the author of", "<1>" as a field symbol, and ".".

CHAPTER 4

THE DESIGN: INITIATING STATEMENT

The initiating statement for the Dialogue Designing Dialogue System itself is:

```
>New Dialogue
```

The ASK system responds:

```
You are now in the dialogue designing dialogue. You may, at any
time, type "help", "exit" or "wait".
>What user input should initiate this dialogue? _
```

At this point, the user should enter the initiating statement for the Dialogue System he is beginning to design.

The format of an initiating statement may be a fixed one, e.g.:

```
"load ship"
```

or may contain field parameters, e.g.:

```
"load <3>"
"send load plan of <1> to <2>"
```

containing, respectively, the field parameter <3> and the two field parameters <1> and <2>. If the initiating statement prototype contains parameters, the values of these parameters will be provided by the user at run time, e.g.:

```
>load the Alamo
>send load plan of the Alamo to the task force commander
```

The inclusion of the parameter(s) will not only make the initiating of a Dialogue System more natural for the user but also will provide a more flexible format for potential internal calls, i.e., one dialogue calling another dialogue, providing for the passing of information to the called dialogue.

If the prototype statement contains parameters, the character of each of these parameters must be declared by the dialogue designer. First, the type of the response for each of the parameters must be declared:

```
>What user input should initiate this dialogue? load <3>
>What type of user response is expected for field <3>? nu(mber),
 no(un phrase), ti(me), te(xt), se(ntence) or st(ring)? noun phrase
>What type of noun phrase response should be expected (e.g.,
 individual, text class, number attribute, etc.):_
```

Second, the designer may wish to have the Dialogue System check to see that the values entered by the user are appropriate, and to issue meaningful diagnostics if they are not. To this end, the system prompts the designer for this information:

```
 What conditions should the value of field <3> satisfy?
>Condition: <3> is a ship?
>Enter a diagnostic message for the user in case the condition is
 not satisfied: ....
>Condition:

 What conditions should the values of fields <1> and <2> satisfy?
>Condition: <1> is a ship?
>Enter a diagnostic message for the user in case the condition is
 not satisfied: ....
>Condition: Is <1> in <2>'s command?
>Enter a diagnostic message for the user in case the condition is
 not satisfied: ....
>Condition:
```

Note that a condition is stated as a "yes"/"no" ASK English question. At run time, each condition is checked. The diagnostic messages for all conditions that are not satisfied are output to the user. If any of the conditions is not satisfied, the dialogue is not initiated and control stays in ASK mode. After all of the conditions have been specified, or if there are no conditions or no parameters, the design phase issues the following:

```
When the user enters "...the initiating statement..." s/he will
be at node 1 of the dialogue you are designing. Now define each
node in turn.
```

and starts the design of the first node.

CHAPTER 5

THE DESIGN: DESIGNING A NODE

5.1: Characterization of the Node

After the initiating statement of the Dialogue System is specified, the designer will be asked to design the structure of each node, starting with node 1:

```
Designing node 1.
```

Suppose we are starting the design of node j. The system will first display all fields that have been assigned, indicating the assignment statements by which they were introduced. For example:

```
Designing node 4.
The fields that have been assigned on entering the node are:
<1> : Add to what bibliography?
<2> : Title:
<3> : Author:
<10>: address of <3>
```

Thus the designer sees immediately which fields are available and what they contain. If in designing this new node, he uses a field that is not in this list, he is prompted to include the assignment. All prompts for designing the node will appear in the following sequence. (Note that at any point in the sequence, the designer can ask for help, issue a wait thus suspending the design process, or exit aborting the design session.)

5.1.1: The node prompt

The first step in node design is to specify its prompt:

```
>What is the prompt message for this node?
```

The designer's response can be any string, including field symbols. At run time, these field symbols will be replaced by the literal for the value of the prompt. Examples of prompt messages are:

```
>Author:
>Destination of <1>:
>Should <3> be on the L(eft) or R(ight) of <4>?
```

The system proceeds to find out what should be done with the user's response, and the nature of the expected user response.

```
>If you wish the response to this prompt to be assigned to a field,
 then give the field number here:
>What type of user response is expected? nu(mber), no(un phrase),
 ti(me), te(xt), se(ntence) or st(ring):
```

If the type of user response is expected to be a noun phrase, then the system needs to ascertain what to do if it can not parse the user's response into a phrase of the form specified. It may be that this response is to be treated as a new word to be added to the vocabulary; on the other hand, unknown words may not be acceptable at this point, therefore the system just reprompts. To this end, the system asks the designer:

```
>If the user's response is not understandable, presumably not in the
 vocabulary, what should be done: a(ccept) or r(eprompt):
```

Finally, the system needs to ascertain in what form the user response will be treated:

```
>What type of noun phrase must the response be? (e.g., individual,
 text class, number attribute, etc.):
```

When the system issues the prompt:

```
>What is the prompt message for this node?
```

the designer may give a nil response (i.e., only carriage return). In this case, at run time when a transfer is executed to this node, no prompt message will be shown on the user's terminal, the application Dialogue System proceeds directly to the underlying decision structure. This feature is useful when, for example, several nodes transfer to a common point and common actions are to be taken. The system responds to a designer's nil response with:

```
No user interaction for this node.
```

and proceeds directly to the design of the node's decision structure (see section 5.1.3).

Suppose a designer, in designing a "Dialogue System", were to uniformly, for every node, give a nil response to the prompt:

```
>What is the prompt message for this node?
```

The result would be a computer program without user interaction. In this way, we see that the Dialogue Designing Dialogue subsumes, as a special case, a pure automatic programming capability, without dialogue elements.

5.1.2: The help message

Suppose a user comes to a given node but finds that he needs more information in order to proceed. He should be able to ask for "help" and receive some assistance. Therefore, the designer should provide such help messages. To this end, the system will ask the designer what message should be displayed:

```
>Enter a "help" message for your user at this node:
```

If the designer responds with the nil response, then if the user types "help", this message will be displayed:

```
No help available
```

5.1.3: Designing the node decision structures

The final two steps in designing the node are to specify two node decision structures, one for the case where the user has responded to the prompt with a nil response, the other where he has responded with a good response. The dialogues for each of these cases are essentially the same, except for the initial prompts:

```
>What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
 a(ction-transfer), or c(ondition-action-transfer):

>What should be done if the user gives a good response? f(inish up),
 t(ransfer), a(ction-transfer), or c(ondition-action-transfer):
```

If there is no prompt for a given node, then the designer is asked:

```
>What should be done at this node? f(inish up), t(ransfer),
 a(ction-transfer), or c(ondition-action-transfer):
```

The available design options are:

- reprompt: reissue the prompt for this node;
- finish up: mark this node as a terminal node for user interaction, schedule the completion of all indicated actions, and return of control to ASK mode;
- transfer: transfer directly to another node; the designer will be prompted for its number;
- action-transfer: the designer will be prompted for actions (see section 5.1.3.2) that should be scheduled at this node, and then for the number of the node to which to transfer;

- condition-action-transfer: the designer will be prompted for a condition, for a message to be issued if the condition is not satisfied, and for the actions to be taken and the transfer to be taken if the condition is satisfied; he will then be prompted for a second condition, and so forth.

5.1.3.1: Condition-action-transition sequences

Both a nil response and a good response allow the designer to specify a condition-action-transition (CAT) sequence. These sequences are the principal part of node design in using the Dialogue Designing Dialogue system. Essentially, they are a dialogue form of the standard "IF-THEN-ELSE" program segment. When such a sequence is selected as a response option, the Dialogue Designing Dialogue system will prompt the designer to build up his decision structure on the node by specifying first an "IF" part, then the corresponding "THEN" part, cycling back until the "IF" part is answered with "otherwise" or a nil response.

The prompt to the "IF" part is simply:

```
>Condition:
```

and expects an ASK English "yes/no" question, possibly containing field symbols, or the single word "otherwise", which represents the "none-of-the-above" condition. Examples are:

```
>Condition: <5> is a ship?
 .....
>Condition: <6> is less than or equal to the inventory of <7>?
 .....
>Condition: otherwise
 .....
```

There is one additional response the designer may give to the "Condition: ". This response concerns the use of a field as a stack or queue. As we will see below, actions may be specified for popping elements off a field, when it is used as a stack or queue. Thus, to control looping, the designer must be able to indicate a test to see if the stack or queue is empty. To this end, he may enter the condition: "is stack/queue <n> empty?", e.g.:

```
>Condition: is stack <3> empty?
```

If any condition is not satisfied, the designer may wish an interpretation of the situation, or diagnostic information, to be given to the user. It may include field symbols, which will be replaced by the literals of the associated user responses. If the designer enters a nil response, no diagnostic will be given the user.

```
>Enter a diagnostic message for the user in case the condition is
 not satisfied:
```

The first prompts to the "THEN" part are for actions:

```
>Action:
```

Each action statement is followed by another "Action: ", until the designer gives a nil response; thus the designer can specify as many actions as he desires. (The actions available to the designer are described in section 5.1.3.2.) For example:

```
>Action: <2> is the author of <1>.
>Action: Send <1> to <3>.
>Action: <3> is the recipient of <1>.
>Action: (nil)
```

Finally, the designer is prompted for the number of the next node (see section 5.1.3.3).

After the specification of the first CAT sequence is completed, the Dialogue Designing Dialogue system, under the assumption that this current condition is not satisfied, prompts again for another condition specification, so on and so forth, until all conditions (and associated actions and transitions) have been specified.

A CAT sequence ends after the "Action: "and "Transfer to what node? " statements following a "Condition: " response of "otherwise" or nil. At this point, the design of the node is completed with the message:

```
Node j is completed.
```

The designer is informed by the system to design the next node k where k is not necessarily to be j+1. In fact, the node that is to be designed next is determined by the system by the depth-first method (see section 5.2).

By such a repetitive use of the CAT sequences, all of the five patterns of "IF-THEN-ELSE" program segments can be achieved. This is illustrated in Figure 5.1 with simplified CAT sequences (where actions are omitted and conditions, except the "condition: otherwise", are not explicitly specified; and the "next: " means "Transfer to what node? ".)

```
(a)        (b)          (c)            (d)            (e)

1 --> 2    1 --> 2      1 --> 2 --> 4  1 --> 2        1 --> 2 --> 5 --> 4
           |            |     |        |              |           |
            -> 5        |      -> 5    |-> 5           <----------
                        |              |
                         -------> 5     -> 4
```

```
(a)                 (c)                 (d)            (e)
node 1:             node 1:             node 1:        node 1:
  next: 2             cond: --            cond: --       next: 2
                      next: 2             next: 2
                      cond: otherwise     cond: --     node 2:
(b)                   next: 5             next: 5        next: 5
node 1:                                   cond: --
  cond: --          node 2:               next: 4      node 5:
  next: 2             cond: --                           cond: --
  cond: otherwise     next: 4                            next: 1
  next: 5             cond: otherwise                    cond: otherwise
                      next: 5                            next: 4
```

Fig. 5.1 The comparison of a dialogue form (lower part) with the general pattern of the IF-THEN-ELSE structure (upper part); the one-one corresponding is for (a)unconditional, (b) if-then-else, (c) if-then-if, (d)case, (e) while.

5.1.3.2: Available Actions

The following actions are available to the designer. In each case, the way the actions are to be specified is shown and examples given.

a) Updating the database: an ASK declarative sentence, possibly including field symbols, e.g.:

```
>Action: Publisher of <5> is <6>.
>Action: Change the length of <2> to <7>.
```

b) Display the updating: same as above with the addition that the ASK System's response to the declarative sentence will be displayed. e.g.:

```
>Action: Display: Publisher of <5> is <6>.
```

thus at run time:

```
>Title: Let's talk LISP
 .....
>Publisher: Prentice-Hall
 ......
 Prentice-Hall has been added as publisher of Let's talk LISP.
```

c) Display information to user: any ASK English query, e.g.:

```
>Action: What ships carry <5> or <6>?
>Action: List the length and width of each <5>.
```

d) Display message to user: any message enclosed in double quotes (including field symbols) will be replaced by their literal values; e.g.:

```
>Action: Display: "When you finish, please send a copy of <5> to <4>."
```

e) Save information as text: a statement of the form:

```
Save in -name of a text object-: -an ASK English query-
```

The information, as text, will be placed at the end of the indicated text; e.g.:

```
>Action: Save in load plan of <2>: List cargo type, assignment
 and destination of each cargo of <5>.
```

f) Save message as text: a statement of the form:

```
Save in -name of a text object-: message in double quotes
```

The information, as text, will be placed at the end of the indicated text; e.g.:

```
>Action: Save in load plan of <2>: "Cargo Loading Plan for <5>:"
```

g) Field assignment: a statement of the form:

```
Assign to <n>: -an ASK expression-
```

This may be used to reassign a field's value, or to introduce a new field whose value does not come directly from a user response. e.g.:

```
>Action: Assign to <5>: home port of <2>
>Action: Assign to <8>: <8> + <4>
```

h) Add an element into a stack: a statement of the form:

```
 Action: push on --a field-- : -- any ASK English phrase --; e.g.:
>Action: push on <3>: <5>
```

In most cases, this phrase will be a simple field that is to be stacked. The statement inserts the content of field <5> on the top of the stack that is pointed by field <3> (after the push, field <3> points to the newly added element).

i) Add an element into a queue: a statement of the form:

```
 Action: push under --a field-- : -- any English phrase --; e.g.:
>Action: push under <3>: <5>
```

In most cases, this phrase will be a simple field that is to be queued. The statement inserts the content of field <5> into the tail of a queue whose head is pointed to by field <3>.

j) Remove an element from a stack or from a queue: a statement of the form:

```
 Action: pop --a stack field--
 Action: pop --a queue field--   e.g.:
>Action: pop <3>
```

The action takes an element off the top of the stack or the queue (both are pointed by field <3>); after the pop, field <3> points to the new top of the stack or queue, or to nil.(The designer is not protected from programming errors in using push and pop.)

k) Call another Dialogue System for use: a statement of the form:

```
 Call: -the Initiating Statement of the Dialogue System that
       is to be called (including the call of itself)-; e.g.:
>Action: Call: load <8>.
```

l) Quit the action prompt: upon the specification of one of the above actions, the Dialogue Designing Dialogue system will reprompt for another action; a nil response will make this system advance to a node transition prompt.

Note that following each condition the designer may specify any number of actions of these various types.

Actions specified during a CAT sequence may include such ASK English statements as:

```
>Action: Create a text named ...
>Action: Mail ..text.. to ..authorized person...
>Action: File ..text.. in ..text file..
```

These action specifications, together with the "Save in ..." actions, allow the designer to provide convenient protocols for communication between databases and members of the team or staff using the system. The following segment of a dialogue design links the user to the outside world:

```
>Condition: <5> is an authorized person?
>Action: Mail <4> to <5>.
>Action:                 (default, indicating no more actions)
>Transition: -next node-
```

```
>Condition: otherwise     (final condition, the ELSE of an IF-THEN-ELSE)
>Action: File <4> in my secretary's Please Forward file.
   (note: "my" will be interpreted as the user at run time)
```

The "Create a text named ..." actions together with the "Save in ..." actions provide the capabilities of a standard report generator.

5.1.3.3: Transfer to the next node

The response to a node transition prompt sets up the transfer of control along the node path. The actual node transfer is done only after the actions are performed. A direct transfer, one of the options available for a nil or responsive response, does not traverse a CAT sequence at all.

A nil response to a node transition means that the node is a terminal node of the newly designed Dialogue System. That is, when a user invokes this Dialogue System, if he reaches this node, then after carrying out all of the indicated actions, his dialogue is completed. To have multiple terminal nodes in one Dialogue System is not unusual.

Suppose the designer, in the process of designing a node, indicates a transition to a following node that has already been designed. The system checks to see that all fields referenced in this following node either are available to the current node by previous assignments or have been assigned in the current node. If a field is found that is referenced by the indicated following node but has not been assigned along the node path leading to and including the current node, the designer is prompted with this fact to include this assignment.

5.1.4: The last node

Finally, when the last node, say node m, has been built, the epilogue of the Dialogue System will show the following completion message:

```
>Node m is completed.
 All nodes have been designed.
 You have completed the design of the new dialogue "load <3>".
```

If, at this time, some Dialogue Systems were referenced in the "Action: call: ..." statements but not designed yet (see Chapter 6), then these to be designed Dialogue Systems (in terms of their initiating statements) will appear in the following message:

```
The following dialogues remain to be designed:
 Dialogue System "...instance of the initiating statement..."
 Dialogue System "...instance of the initiating statement..."
 ....
Proceed with the design of one of these dialogues.
```

then followed by the system prompt:

```
>What user input should initiate this dialogue?
```

5.2: Issues in Node Construction

In designing a dialogue, a designer may possibly reference a field that is not yet assigned in the given node path; or may possibly form an entirely closed cyclic node path (potentially an infinite loop). In general, in a programming language, there is no way to check whether such an occurrence is an actual error; there may be mitigating circumstances. However, the potential for such problems can be detected. We have chosen to do so, giving the user a warning, thus providing additional support since the system is designed for use by users with little if any programming experience. In this section, examples of these problems are given, and the means used for solving these issues discussed.

Each node corresponds to a dialogue element and its underlying decision structure, therefore a node flow is a conceptual representation of the relationships among dialogue elements. A node flow is a directed graph, possibly including cycles. A typical one looks like the following (where "T" means a terminal node):

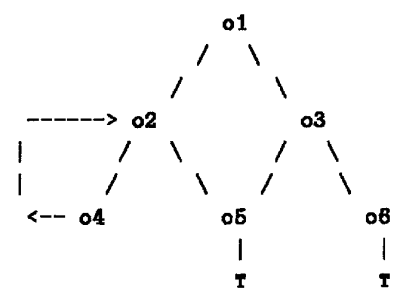

Fig. 5.2. A typical node structure, possibly with cycles

To describe how a node flow is performed, we introduce the term "node path" in the format: (... nodes sequence ...). The node path (1 2 5), for example, describes

that node 1 will be performed first, followed by node 2, then node 5. Two kinds of node paths exist:

- **Node execution path:** the sequence of nodes traversed in the run phase.
- **Node construction path:** the sequence of nodes traversed in the design phase.

It is in the node execution path where problems may occur; it is in the node construction path that problems must be anticipated.

The node construction path is developed using the depth-first method, which is similar to the "preorder traverse" [Knuth73] in searching a tree structure (if we disregard cycles). In the case of Figure 5.2, for example, the resulting path will be constructed in the (1 2 4 5 3 6) sequence.

In building a node, we must be concerned with the following cases. First, a field that is used while not yet defined (i.e., **an undefined field**). Suppose, in Figure 5.2, that field <10> is to be used in node 5. If field <10> has been defined in node 2 and has not been defined in node 3 during the node construction, then field <10> is such an undefined field. In run time the node execution path (1 2 5) presents no problem; however, the other path (1 3 5) may cause a problem if <10> is used indiscriminately. Second is the case in which nodes that form a closed node transition where none of these nodes will eventually transfer to a terminal node. If the transfer of node 2 to node 5 does not exist in the node flow of Figure 5.2, for instance, then node 2 and node 4 form a closed cycle. The algorithm for node construction should give the user warning of the existence of either of these potential conflicts.

To detect these problems, the algorithm keeps track, for each node, of (1) the fields that have been defined in every node path leading to and including this node, (2) the fields that are used in condition and action statements of this node, and (3) the nodes to which this node transfers. This information is sufficient to identify many of these potential errors.

The following two figures of simple node flow are exemplified to illustrate the system's response when the above mentioned problems occur.

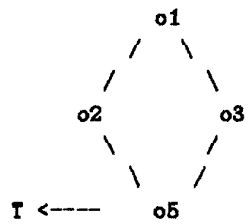

Fig. 5.3

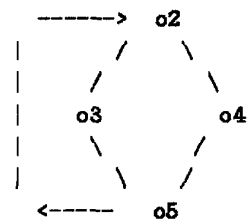

Fig. 5.4

Consider an example of an "undefined" field. In Figure 5.3 where the node construction path is (1 2 5 3), node 2 defines field <10> and node 5 uses field <10>. When node 3 sets up a transfer to node 5, those fields used in node 5 (and its successors, if any) are checked against the fields defined at node 3; thus in this example, since field <10> is not defined for node 3, the system will warn the designer. The message will be in one of the following formats:

```
>Transfer to what node? 5
 Warning: Field <10> is used in node 5 or its successors,
 but in at least one path leading to this node. it will
 not have been defined.
>Do you want to retain this transfer? (y/n):

>Transfer to what node? 5
 Warning: Fields <10> ... are used in node 5 or its
 successors, but in at least one path leading to this
 node, they will not have been defined.
>Do you want to retain this transfer? (y/n):
```

Similarly, in Figure 5.3 where the node path is (1 2 5 3), where node 2 does not define field <10>. When node 5 sets up an action or condition statement referencing field <10>, the system will warn the designer. The message will be in the following format:

```
>Action: <10> is a ship.
 Warning: Along at least one path leading to this node,
 field <10> .... will not have been defined.
>Do you want to retain this action? (y/n):
```

An example for the case of an infinite loop is given in Figure 5.4. When node 5 sets up a transfer to node 2, the path (2 3 5 2) forms a loop, but it need not be an infinite one. This is because node 2 also transfers to node 4, which has not been constructed yet. However, when node 4 sets up a transfer to node 5 and its design is

completed with no other transfers, then the path (5 2 4 5) has no way to terminate. This is because no node in the path leads to an unconstructed node or a terminal node. Thus, the system will warn the designer with the following message:

```
>Transfer to what node? 5
 Warning: the following nodes may potentially form an
 infinite loop: (2 3 5 4)
>Do you want to retain this transfer? (y/n):
```

In view of the fact that a Dialogue System may be quite complex, and what currently appears to be a source of error may not actually cause an error, the algorithm plays only an informative role. It leaves the final decision to the designer with the options that he may respond to the prompt either with a "no" to enter another input response, or a "yes" to continue the processing. He may later on modify those aspects of the design that are responsible for the current error by using the "dialogue editing" facility (see section 9.2), or "exit" to start over the entire design.

CHAPTER 6

THE DESIGN: CALLING A DIALOGUE

There are three ways that control can pass between the ASK Mode and the Dialogue Mode, as shown below in Figure 6.1.

```
+-----------------------------------------------+
|       ASK System                              |
|                                               |
|                ASK interactive mode           |
|                                               |
|              |                                |
|        initiating a dialogue from ASK mode    |
|              |                                |
|        +-----|-----------------+              |
|        |     v                 |              |
|        | Dialogue      ..wait----->........   |
|        | System A              |         .    |
|        |               <----------resume...   |
|        |                       |              |
|        | Action:call:...       |              |
|        |   |                   |              |
|        |   |      ^   ..complete---->         |
|        |   |      |            |              |
|        +---|------|------------+              |
|            |      |                           |
|        +---|------|-----------+               |
|        |   v    complete      |               |
|        |     ...............  |               |
|        | Dialogue  System B   |               |
|        |                      |               |
|        +----------------------+               |
+-----------------------------------------------+
```

Fig. 6.1 Three links between ASK mode and Dialogue mode

- a dialogue may be initiated by the user from the ASK Mode; when this dialogue is completed, control returns to ASK Mode;

- a user can type "wait" in answer to a dialogue prompt; control then passes to the ASK Mode until the user types: "resume", at which time control returns to the Dialogue System at the point where it was put in recession;
- one dialogue can call another dialogue, in which case the calling dialogue goes into recession, much as in a wait/resume sequence; when the called dialogue is completed, control passes back to the calling dialogue at the point where it was put into recession.

Initially, when the user is in the basic ASK interactive mode, both GLOBAL^[current_dialogue] and GLOBAL^[dialogue_stack] are nil. When a dialogue is initiated by a user, the GLOBAL^[current_dialogue] will point to the Dialogue Data Structure of the initiated Dialogue System. When the Dialogue System is finished, and the GLOBAL^[dialogue_stack] points to nil, then there is no Dialogue System waiting for processing and the GLOBAL^[current_dialogue] will again point to nil:

	GLOBAL^[current_dialogue]	GLOBAL^[dialogue_stack]
	nil	nil
user initiates dialogue DS1	DDS1	nil
dialogue DS1 completes	nil	nil

6.1: The Wait - Resume Mechanism

When the user responds to a prompt with "wait" in a Dialogue System, say DS1, its current Dialogue Data Structure, DDS1, is then put into a push down stack, in GLOBAL^[dialogue_stack], and the control of processing is transferred to the ASK Mode where the user can either stay with normal ASK processing for a while or initiate another Dialogue System DS2 and thus activate its corresponding Dialogue Data Structure. Later on, when in the ASK Mode, he can get back to where he was in DS1, resuming its execution, by issuing a "resume" command. The system responds by taking DDS1 off GLOBAL^[dialogue_stack] and returning it to GLOBAL^[current_dialogue], thus reactivating it.

Wait Algorithm

Is the user in Dialogue Mode, i.e., is GLOBAL^[current_dialogue] non-nil?

If not, then output the diagnostic message:

You are not in a dialogue.

Otherwise:

Add the following element to the top of GLOBAL^[dialogue_stack]:

WL[i]^: (lll, 0, DDS1, LL, WL[i+1])
LL^ : (ssl, 0, current-prompt, current-prefix, prompt-line)

where DDS1 is the Dialogue Data Structure that was in GLOBAL^[current_dialogue], and prompt_line contains the message:

Resuming the xxx dialogue.

where xxx is the name of the current_dialogue. If the prompt message consists of more than one line, then prompt_line also contains all but the last of these lines.

Set GLOBAL^[current_dialogue] to nil;

Resume Algorithm

Is any dialogue in recession? i.e., is GLOBAL^[dialogue_stack] non-nil?

If no, then output the diagnostic message:

No suspended dialogue.

Otherwise:

Take the top element:

WL[1]^: (lll, 0, DDS1, LL, WL[2])
LL^ : (ssl, 0, prompt, prefix, prompt-line)

off the GLOBAL^[dialogue_stack], leaving it pointing to WL[2];

Set GLOBAL^[current_dialogue] to DDS1;
Build an OUT_phrase with the following format:
(OUT, 0, prompt-line, prefix, prompt)

ASK will then reactivate this dialogue.

For example, suppose a user wishes to use a Dialogue System "new item". The sequence of events is shown in Figure 6.2 (where DDS1 is the Dialogue Data Structure of "new item").

```
                          GLOBAL^[current_   GLOBAL^[dialogue_
                                  dialogue]          stack]

                                   nil                nil
> new  item
                                   DDS1               nil
     o
     o
 Do you wish the new item to be entered into the
>vocabulary? (yes/no):   wait
                                   nil                WL[1]
    where
      WL[1]^: ( 111, 0, DDS1, LL, nil )
      LL^   : ( ssl, 0, "vocabulary? (yes/no):", prefix,
                "Resuming the new item dialogue.
                Do you wish the new item to be entered into the" )
     o
     o
> resume
 Resuming the new item dialogue.
 Do you wish the new item to be entered into the
>vocabulary? (yes/no):
                                   DDS1               nil
```

Fig. 6.2 Stepwise changes of two global variables in the wait - resume events

From the above discussion, it is clear that many dialogues can arbitrarily be in recession in GLOBAL^[dialogue_stack].

6.2: Dialogue Calling Mechanism

A Dialogue System, say DS1, may call another Dialogue System, including itself. This call function may be achieved by the designer by specifying in the design phase of DS1 the statement "Action: call: ...". For example,

```
>Action: call: DS2
```

During the run phase of DS1, when the "call" is being executed, the following call algorithm (similar to the wait algorithm) applies:

Call Algorithm

Add the following element to the top of GLOBAL^[dialogue_stack]:

WL^: (lll, 1, DDS1, nil, WL[1])

where DDS1 is the Dialogue Data Structure that was in GLOBAL^[current_dialogue] and WL[1] the previous top of GLOBAL^[dialogue_stack];

Set GLOBAL^[current_dialogue] to DDS2, the Dialogue Data Structure, from disk, for the called Dialogue System DS2.

When the dialogue DS2 has just about been completed, as the last step, it checks to see if it was called by another dialogue, i.e., whether the top element in GLOBAL^[dialogue_stack] has a flag of 1. If not, it simply returns to the ASK Mode. If so, it returns DDS1, the Dialogue Data Structure for the calling Dialogue, to GLOBAL^[current_dialogue] and pops it off the GLOBAL^[dialogue_stack].

For example, suppose a user wishes to use a Dialogue System "dispatch item" which may call a Dialogue System "load ship". Relevant pieces of their design specifications are shown below (in the left and the right side respectively):

```
          o
 Designing node 5.
          o
>node prompt: ship:
>assign to field: <3>
          o
>Action: call:load ship
>Action: Display: "<3> is
 being loaded"
          o
```

Fig. 6.3A: Dispatch Item Dialogue

```
 initiating statement: load ship
 Designing node 1.
          o
>node prompt: cargo space:
          o
 Designing node 6.
          o
>node prompt: stack height:
          o
          o
```

Fig. 6.3B: Load Ship Dialogue

Suppose DDS1 is the Dialogue Data Structure of the Dialogue System "dispatch item" and DDS2 is the Dialogue Data Structure of the Dialogue System "load ship". Then this call-wait-resume works in the following sequence of events when a user wishes to run the Dialogue System "dispatch item".

```
                                GLOBAL^[current_   GLOBAL^[dialogue_
                                        dialogue]          stack]

                                          nil                nil
> dispatch item
                                          DDS1               nil
    o
    o
>ship:   Maru
                                          DDS2               WL[1]
          where
          WL[1]^: ( lll, 1, DDS1, nil, nil )
>cargo space:
    o
    o
>stack height:   wait
                                          nil                WL[2]
          where
          WL[2]^: ( lll, 0, DDS2, LL, WL[1] )
          LL^   : ( ssl, 0, "stack height: ", prefix,
                    "Resuming the load ship dialogue." )
          WL[1]^: ( lll, 1, DDS1, nil, nil )
    o
    o
> resume
 Resuming the load ship dialogue.
                                          DDS2               WL[1]
>stack height:
    o
    o
 You have finished the load ship dialogue.
                                          DDS1               nil
 Maru is being loaded.
    o
    o
 You have finished the dispatch item dialogue.
                                          nil                nil
```

Fig. 6.4 Stepwise changes of two global variables in the call-wait-resume event

6.3: Parameter Passing in the Calling Mechanism

Suppose the Dialogue System DS1 calls the Dialogue System DS2. This call is designed into DS1 during the design phase of the Dialogue Designing Dialogue as an action statement. Then the parameter passing in this call may be in one of the following cases:

Case 1: Suppose the initiating statement for DS2 does not have any parameters, e.g., "load ship". Then the action statement is simply:

```
>Action: Call: load ship
```

Case 2: Suppose the initiating statement for DS2 contains parameter(s), e.g., the initiating statement form for DS2: load <3>, and that DS1 wishes to issue a call by value. Then the action statement of DS1 may be either

```
>Action: call: load Alamo
```

where the parameter is replaced by a constant (e.g., "Alamo"); in this case, the instantiation of DS2 field <3> will be the value of Alamo; or

```
>Action: call: load <5>
```

where the parameter is a variable (i.e., field <5> of DS1); in this case the instantiation of DS2 field <3> will be the current value of the DS1 field <5>.

Case 3: Suppose the initiating statement for DS2 contains a parameter, e.g., "load <3>", and it may be that the value of DS2 field <3> is re-assigned in DS2, and DS1 wishes to issue a call to DS2 in which the value of the parameter is returned to DS1. Then the action statement in the design of DS1 will be:

```
>Action: call: load <V5>
```

In case 2 where the value is given by a field number (i.e., <5>) or in case 3 (i.e., <V5>), the call processing procedure in Dialogue Designing Dialogue will recognize this "<number>" or "<Vnumber>". It does not access directly to where field <5> is allocated in DS1, as the usual call by address may do. Instead, it first instantiates the DS2 field <3> by the value in DS1 field <5>, then runs DS2. It is this value of field <3> in DS2, not field <5> in DS1, that will be referenced during the processing of DS2. In case 3, on completion of DS2, it reassigns the value in Dialogue Data Structure DDS1 for the DS1 field <5> by the value in the terminal Dialogue Data Structure DDS2 for the DS2 field <3> and continues the processing of DS1.

6.4: Addressing Problem in Calls

6.4.1: Problem and sketch of solution

In the Dialogue Designing Dialogue system, an expression in an "Action: call: ..." statement is expected to be an initiating statement for a Dialogue System (i.e., the name of a dialogue) that has already been defined, and thus is in the dictionary with the part of speech <dialogue>. Therefore it presents no problem in the parsing process for the called dialogue. For example, dialogue DS1 calls dialogue DS2 and DS2 can be found in the dictionary.

However, a problem will occur if DS2 is not defined in the dictionary during the design stage of DS1 prior to the "Action: call: DS2". In such a case, we may use the "delaying compilation" technique to solve the problem. In other words, we may temporarily mark this call statement in some way as incomplete, continue the rest of the design stage (the compilation, in the Dialogue Designing Dialogue's viewpoint) of DS1, and then start the design of DS2. As soon as DS2 is completed and stored in the dictionary, we may then go back to finish the marked incomplete call statement with the address of DS2.

Normally, in the stack-based language system (e.g., PASCAL) this addressing problem can be solved in a so called "call forward" technique. The call forward assures the compiler in advance the existence of the forwarded procedure and helps the type checking when the yet undefined procedure is called. The relative address of the call statement to the calling procedure is also recorded somewhere, then with the aid of the linkage editor the absolute addresses of all the procedures involved are resolved. However, this technique does not apply to our particular call situation in the Dialogue Designing Dialogue and the ASK environment.

The nucleus of the problem in our call statement concerns the type of the address that we need to record. It is not the relative address of the called Dialogue System to the calling Dialogue System, it is the absolute address of the called Dialogue System's data structure in the disk storage. This is because when a Dialogue System DS2 is about to be finished, its corresponding Dialogue Data Structure DDS2 will be stored in the hard disk permanently by means of the ASK function LIST_OUT, which in turn returns the disk address of the first element of the just stored DDS2. This returned address will then be stored in the dictionary together with the dialogue name, e.g., DS2. Therefore, when in the design of DS1 the action statement: "Action: call: DS2" is entered, the disk address of DS2 can be found in the dictionary and this address of DDS2 can thus be recorded in the successfully parsed "Action: call:..." phrase of the Dialogue Data Structure of DS1.

However the racing problem occurs in the recursive case. When DS1 calls itself, for example, the disk address of DS1 can be obtained (and then stored into the dictionary) only after its Dialogue Data Structure DDS1 is listed out, while at this moment DDS1 still contains an unresolved (disk) address of DDS1 in the action phrase. The problem is that the dictionary and action phrase race for the disk address. A technique of listing out DDS1 first to get its disk address, say DA1, modifying the action phrase with DA1, and then LIST_OUT this modified DDS1 will not help because this second list out does not write DDS1 back on the same place. It returns a different address from the first one, and therefore puts inconsistent links in the dictionary. An alternative would be to modify the pages on which the original form of DDS1 had been written; however this would violate the structured programming aspects of the ASK system design, which provides access to paged lists only through the LIST_OUT/LIST_IN mechanism.

Our solution to this problem is to provide for an indirect addressing scheme for accessing the Dialogue Data Structures of Dialogue Systems. This indirect address is a page address which we will call the Dialogue Indirect Address. (Since a page may hold many such indirect links, a simple mechanism is used to obtain a new Dialogue Indirect Address that economizes on pages.) Whenever a user initiates the Dialogue Designing Dialogue to design a new Dialogue System, say DS1, a new Dialogue Indirect Address is obtained for it, and at that time the initiating statement for DS1 is put into the dictionary with a pointer to this indirect address. Thus in subsequent processing of action statements calling DS1, (i.e., Action:Call:DS1), either in DS1 itself or in other Dialogue System designs, DS1 appears just as other Dialogue Systems that have been completely defined, no incomplete reference need be left to be resolved. Upon completion of the design of DS1, i.e., when its Dialogue Data Structure is completed, the page address of this DDS1 can be put into its Dialogue Indirect Address location. When DS1 is subsequently initiated, it is a simple matter to take the extra step through this indirect address to obtain DDS1. The details of how this mechanism solves the race problem for recursive calls will now be explained.

6.4.2: Indirect addressing mechanism

During a dialogue design session, i.e., when the Dialogue Designing Dialogue has been activated, there are three kinds of Dialogue Systems:

- Those that have already been **completely defined**; their dialogue names will already be in the dictionary, the dictionary entry pointing, indirectly, to their

Dialogue Data Structures.

- Those that are **partially defined**, that have incomplete "Action:call:..." definitions because the called dialogue has not been defined. Although their dialogue names will also already be in the dictionary, their Dialogue Data Structures must still be completed.
- Those that have **not yet been defined** at all, but have been named in "Action:call:..." statements of dialogues of the second kind, dialogues that must be defined in this Dialogue Designing Dialogue session so that others may be completed.

We will refer to Dialogue Systems of first, second and third kinds.

The solution is accomplished in three parts: at the start of the design of a new Dialogue System, during the design when an "Action:call:..." statement is issued, and at the end of the design.

At the beginning of designing a Dialogue System, say DS1:

- Push this DS1 on GLOBAL^[dialogue stack] with the format: WL^: (lll, 2, DDS1, nil, WL') where DDS1 is the Dialogue Data Structure for DS1. The fourth field, now "nil", will later be replaced by a link to dialogues of the third kind that are called by DS1.
- When all parameters in the initiating statement have been recorded with their respective parts of speech, the system will:

  1) get a Dialogue Indirect Address for DS1. (A current partially used page will be kept in a system variable from which a new address can be allocated.)

  2) initialize the content of the element to be nil,

  3) enter the initiating statement into the dictionary with the part of speech <dialogue> and the address that is found in 1) as its semantic content.

Suppose we are now **in the midst of designing the Dialogue System DS1**, and, in response to the prompt: "Action:" we respond:

```
>Action: call: DS4
```

where the Dialogue System DS4 has not yet been defined. Then DS4 is a Dialogue of the third kind, and is put in the GLOBAL^[dialogue_stack] with the following format of the entry L[i,j] that is linked to the WL[i] entry for DS1:

```
   WL[i]^ : ( lll, 2, DDS1,  L[i,1],    WL[i+1]  )
L[i,j]^: ( sll, 0, =>"DS4", action address in DDS1, L[i,j+1])
```

where =>"DS4" is a literal (string) pointer to the calling statement to DS4 in the "Action: call: ... " statement; and the action address in DDS1 contains a dummy value, waiting for the completion of DS4.

If, for example, the last stored L[i,j] is L[i,3], then DS1 calls Dialogue Systems of the third kind three times. It could be that the same Dialogue System is called several times but in different places; it could also be that more than three dialogues are called but some of them are of the first kind.

When the design of DS1 has been completed, except for completion of "Action:call: ..." statements, the Dialogue Designing Dialogue system proceeds as follows:

- Check the GLOBAL^[dialogue_stack] stack to see if DS1 has indeed been completed, i.e., DS1 has only the form:

  WL[i]^: (lll, 2, DDS1, nil, WL[i+1])

  If so, the Dialogue Designing Dialogue system will:

  1) use list_out to put DDS1 on a permanent page, getting its disk address;

  2) *evaluate the initiating statement of DS1; this evaluation will provide the* Dialogue Indirect Address in terms of page number and offset;

  3) update the content of the Dialogue Indirect Address with the disk address of DDS1 obtained from 1);

  4) change the flag of WL[i], the DS1 entry, from 2 to 3 in GLOBAL^[dialogue_stack];

- For every Dialogue System remaining in GLOBAL^[dialogue_stack], check each of the call statements that have been put in the L[i,j] sub-stacks: for example:

  WL[i]^ : (lll, 2, DDSh, L[i,1], WL[i+1])
  L[i,j]^: (sll, 0, =>"DSk", action address in DDSh, L[i,j+1])

  check "DSk" by evaluating the string "DSk". If DSk is now in the dictionary, the evaluation will be successful and the DDSk disk address will be returned (even though the parameters of DSk will have been instantiated in "DSk"); in this case DDSk will be put into DDSh at the action address and the L[i,j] entry will be deleted. If DSk has not been completed, thus the evaluation is not successful, the L[i,j] entry will be left and the statement "DSk" will be put aside in a "to be completed" list.

- The GLOBAL^[dialogue_stack] is checked to see if there are now any entries of the form:

WL[i]^: (lll, 2, DDSh, nil, WL[i+1])

If so (i.e., all its L[i,j] have been deleted during the above step), then:

1) use list_out to put DDSh on a permanent page, getting its disk address;

2) evaluate the initiating statement of DSh;

3) update the content of the Dialogue Indirect Address with the disk address of DDSh obtained from 1);

4) change the flag of WL[i], the DSh entry, from 2 to 3 in the GLOBAL^[dialogue_stack];

- Send to the terminal the "to be completed" list of calls to Dialogue Systems of the third kind, i.e., those which still must be defined. Issue the message:

  Proceed with the design of one of these dialogues.

  and loop back to the starting prompt for designing a new Dialogue System:

  >What user input should initiate this dialogue?

Warning diagnostics guide the designer to design all required Dialogue Systems. When this has been done, that is, when there are no more WL[i] entries in GLOBAL^[dialogue_stack] with a flag of 2, then all WL[i] entries with a flag of 3 are purged from the GLOBAL^[dialogue_stack].

The entire Dialogue Designing Dialogue design session may be aborted. This happens if the designer issues an "exit" to one of the design prompts. In that case, for all entries of the GLOBAL^[dialogue_stack] with flags of 2 or 3:

1) the definition of this Dialogue System is deleted from the dictionary;

2) the entry is purged from the GLOBAL^[dialogue_stack].

This algorithm applies to any Dialogue System call, including recursive calls. The Dialogue Systems designing sequence becomes immaterial.

CHAPTER 7

THE DESIGN: DIALOGUE DATA STRUCTURE, TYPE CHECKING, AND PREFIX/PROMPT GENERATOR

7.1: Dialogue Data Structure

The Dialogue Data Structure for a Dialogue System in the design stage has been implemented in the ASK-style list data structure (see section 3.1), namely, in the following format: (list identifier, flag, f1, f2, f3). The structure, as shown below, has the following five parts: the header list, the parameter list, the node list, the CAT (condition-action-transition) list, and the field list.

```
{header list}
GLOBAL^[current_dialogue].l : = L1 (the DDS list);
L1          : (lll,  current node,  L2, L3,                P1)
L2          : (lli, current field,  N1, V1, cond_ind/cond_rep)
L3          : (bbb, 0, GVB, GNB, GNR)
               GVB  {global_field_built},
               GNB  {global_node_built},
               GNR  {global_node_reserved}      : set of 1..32;

{parameter list}
P1          : (lll, 0,            P2,   para_cond,        dialogue_name)
P2          : (llp, 0,           nil,     P2 link,      caller dialogue)
para_cond   : (lll, 0, para_cond link, cond_phrase,             cond_msg)

{node list}
N1          : (lll, node#,        N1 link,             N2,           N3)
N2          : (lll, field# ,     nil_resp,       norm_resp,          N4)
N3          : (bbb, 0, [nodes transferred],[fields defined],[fields used])
N4          : (lll, 0,                nil,        help_msg,  node_prompt)
nil_resp    : (lli, option,            C1,             nil,  next node/0)
norm_resp   : (lii, next_node,         C1,     accept or reject / option,
                                                  nodetype/catag_opt)
help_msg    :   line list

{condition-action-transition list}
C1          : (lll, 0,                C2,         C1 link,  cond_phrase)
```

```
C2           : (lll, next node, action_list,          cat_msg,          C3)
C3           : (bbl, 0,    [fields defined],   [fields used],      cond_st)
action_list  : (lll, actopt, action_list link,     action_ph, action_param)
cat_msg      :   line list

{field list}
V1           : (lll, field#,              V2,         V1 link,     field_ph)
V2           : (ili, 0, field type/catag_opt,         field_st,     0/node#)
```

The header part is created at the time an initiating statement for the new Dialogue System is given by the designer. It contains pointers to all other parts.

The parameter part is created at the same time as the header part is created. It records the initiating statement in such a form that any parameters are explicitly referenced. It also records the page addresses of the Dialogue Data Structures of all the dialogues that call the dialogue. The field list elements corresponding to these parameters are also created at this time. The condition statements and their corresponding diagnostic messages are added to this parameter list.

Each node part is created at the time the node is introduced in the node construction path. It will be gradually completed later on as the node is being constructed. It contains components such as the field that is assigned to the node, node number, boolean sets for node transitions, fields defined and fields used, pointers to the next node part, the help message, and options for the nil response and normal response, etc.

The CAT sequence part is created at the time the condition-action-transition sequence is introduced. It may exist for both the nil response and the normal response. The components each sequence contains are: a parsed condition statement and/or parsed action statements, and links to the next CAT sequence, etc.

Each field list element is created at the time the field is assigned. It contains components such as field number, field type, the node number in which this field is assigned, field option and category option (in case the field type is a noun phrase), the parsed phrase of the field assignment (e.g., in "Action: assign to ..."), etc.

At the beginning of the run phase, the Dialogue Data Structure is consolidated *to just that information needed during the run phase.* The Dialogue Data Structure in the run stage will be as shown below:

```
{header list}
GLOBAL^[current_dialogue].l : = L1 (the DDS list);
L1            : (lll, 0,          N1,  V1, P1)

{parameter list}
P1            : (lll, 0,            P2,    para_cond,          dialogue_name)
para_cond     : (lll, 0, para_cond link,     cond_ph,                cond_msg)
P2            : (lll, return node#, action-call return address,  nil,     P3)
P3            : (ill, 0, caller V-field#/called V-field#, P3 link, nil, nil)

{node list}
N1            : (lll,     node#,     N1 link,              N2,  node_prompt)
N2            : (lll,    field#,    nil_resp,       norm_resp,     help_msg)
nil_resp      : (lli,    option,          C1,             nil,  next node/0)
norm_resp     : (lii, next_node,          C1,     accept or reject / option,
                                                     nodetype/catag_opt)
help_msg      :   line list

{condition-action-transition list}
C1            : (lll,         0,          C2,         C1 link,  cond_phrase)
C2            : (lll, next node, action_list,         cat_msg,          nil)
action_list   : (lll,    actopt, action_list link,  action_ph, action_param)

{field list}
V1            : (lll,    field#,     inst_ph,         V1 link, field_phrase)
inst_phrase   : (lll,         0, stack/queue link,        nil,   val_phrase)
where val_phrase is the value assigned to the field.
```

Note that in the run phase, the list P2 (and its subsequent lists: P3s) in the parameter list within the Dialogue Data Structure of the calling dialogue has been replaced for parameter passing. It is created whenever a call occurs. The list P2 records the return information: the node number and the address of the current action statement that issues the current call, so that the calling dialogue can resume from the proper place when the called Dialogue System has completed. In the call by address case, each P3 records a pair of the field number: the <Vfield> in the "Action: call: ..." statement of the calling dialogue and the corresponding field in the initiating statement of the called dialogue. When the call has completed, P2 will then be reset to nil.

7.2: Type Checking

We take the notion of "type" from the notion of the type of a variable in many

programming languages. In the ASK System, this notion coincides with that of part of speech (together with features). Since all field types must be known at compile time, as in most programming languages, only static binding applies. Type checking in the Dialogue Designing Dialogue system is performed at both compile time – design phase, and run time – run phase. (It is performed only at compile time in most programming languages.)

At compile time (i.e., the design phase):

Case 1: If the user's input is a normal ASK statement (including fields), then the ASK system function "parse_list" is responsible for rejecting any inconsistent use of types by returning an unsuccessful parsing of the statement. For example, the statement, "Captain of <3> is <8>." where type of field <8> is number, will be rejected.

Case 2: If a field is assigned in an action statement, e.g., "Action: assign to <n>:...action phrase...", then the type of <n> is taken as the part of speech of the action phrase. If this is a reassignment of <n>, then this part of speech must be consistent with the previous assignments. For example, suppose that we assign <3> to be "captain of <7>", later on we assign <3> as "length of <9>". Then the resulting field <3> of the second expression (for which parse_list returns type: number) is in conflict with that of the first one (which returns type: noun phrase), therefore the second expression will be rejected.

Case 3: In Dialogue System call case. The type of each parameter in the "Action: call: ..." statement of the calling Dialogue System will be checked to have the same type as that of each corresponding field in the initiating statement of the called Dialogue System. This type checking holds true for either the call by value or the call by address. For example, suppose the Dialogue System DS1 includes field <3> in the action statement:

```
>Action: call: load <3>
```

where "load <1>" is the initiating statement for Dialogue System DS2. Then field <3> of DS1 must have the same type as field <1> in the initiating statement "load <1>" for the Dialogue System DS2. The initiating statement with its parameters will have been put into the dictionary as a rule of grammar. The above initiating statement, for example, will have become the rule:

```
<dialogue> => "load " <noun_phrase>
```

Thus a call by value instance of this initiating statement, e.g., "load Alamo", or a call by address, e.g., "load <3>", will be type checked by using parse_list, checking that it parses to the part of speech <dialogue>.

At run time (i.e., the run phase):

Case 4: If the user's response to a prompt is not defined in the dictionary and its node type is noun phrase, then the node type and the category option designed for the response will be used to define the response; otherwise its part of speech (obtained from the dictionary) will be checked (by the ASK system function "evaluate") with its field type (obtained from the node type). The response will be rejected if these two types are not the same. For example, suppose in the dialogue element:

```
>length: Alamo
```

the response "Alamo" is defined in the dictionary with part of speech of noun phrase; but the corresponding node type defined in the design phase is a number, then this response will be rejected.

7.3: Implicit Prefix/Prompt Mechanism in the Run Phase

Only two grammar rules are needed to support all Dialogue Systems designed using the Dialogue Designing Dialogue. Namely they are:

```
RULE  initial dialogue rule
<sentence> => <dialogue>
SYN init_dialogue

RULE  continue dialogue rule
<sentence> => "0ddd0"<whole_number>
SYN cont_dialogue
```

Recall that the initiating statement for a Dialogue System is added to the dictionary with the special part of speech: <dialogue> at the first step in the Dialogue Designing Dialogue design phase. Its semantics will consist of a pointer to its Dialogue Data Structure. When a user enters the statement that initiates a dialogue, the statement will, therefore, be parsed with the part of speech <dialogue>, with its Dialogue Data Structure as semantics. This in turn will be parsed by the first of the above rules, and the semantic procedure init_dialogue will thus be called. The first action of the init_dialogue procedure is to get the page address of the Dialogue Data Structure from the <dialogue> constituent, bring a copy of it into

main memory and put it into GLOBAL^[current_dialogue] where it will remain throughout the dialogue. It then obtains from this structure the prompt for the first node of the dialogue, and returns the OUT-phrase:

(OUT, 0, message for node 1, the prompt for node 1, "@ddd@1")

i.e., the output message, the prompt, and the prefix: @ddd@1.

For example, if the dialogue is initiated by the phrase: "new bibliography item", and the first prompt is: "Author:", the following dialogue elements would occur:

```
>new bibliography item
>Author:
```

When the user responds "Jensen", the string "@ddd@1Jensen" would be passed to the parser. Applying the second of the above grammar rules, we would get the parsing:

```
<sentence>---------------
          <whole_number>
@ddd@     1               Jensen
```

and the semantic procedure cont_dialogue would be called. Using the system function Bridge, it would pick up the string "Jensen". From its <whole_number> constituent it will pick up the node number 1; and from GLOBAL^[current_dialogue] its Dialogue Data Structure. With these it can carry out the designed decision structure for the node 1.

This same course of events is followed for each subsequent dialogue element. Thus the response from node n, transferring to node m, would be:

(OUT, 0, –lines for node m–, prompt for node m, "@ddd@m")

the second of the above rules will again apply, the cont_dialogue procedure will again use Bridge to pick up the user's response, the number m from the <whole_number> constituent, and the Dialogue Data Structure from GLOBAL^[current_dialogue], and proceed to carry out the designed decision structure for node m. At the very end of the Dialogue System, i.e., either when a node is reached whose transfer is to the "terminal node", or when the response option is designed to select to "finish up" the dialogue, a "nil" will be recorded in GLOBAL^[current_dialogue], and the control of execution will then be determined by the contents in GLOBAL^[dialogue_stack].

CHAPTER 8

TWO EXAMPLES

Two examples are presented here to illustrate how the Dialogue Designing Dialogue system is utilized to produce Dialogue Systems. We will describe the design phase and the run phase for each Dialogue System.

The first example is a simple Dialogue System for adding books and articles to a bibliography, somewhat more complex than that shown in section 2.2. The second example is about building a family tree into the data base. Two versions of this are given. Both contain only a few prompts. The first version involves a series of recursive calls with parameters. The second version uses a queue technique.

Appendix A consists of the transcription of an instance of the Dialogue Designing Dialogue in which the help messages for all prompts are displayed. This dialogue thus is a complete, succinct documentation of the functions of each dialogue element in the Dialogue Designing Dialogue system.

Appendix B comprises a third example, a complex Dialogue System for handling business invoice transactions. In using this business invoice transaction Dialogue System, both the information entered through the dialogue and information retrieved from the data base are to be distributed to different places in different forms, changing the data base, and composing and dispatching appropriate messages through the electronic mail system. This invoicing system is based on the "invoicing operation" system presented in [Tsichritzis82], a system designed for a quite complex transaction processing application. It was originally presented as an example of the application of the Office Forms System [Kornatowski80] (a system similar in nature to our Forms Designing Dialogue, a precursor system to the Dialogue Designing Dialogue). The example in Appendix B shows the ease in which a Dialogue System to accomplish all of these actions can be designed using the Dialogue Designing Dialogue system. It shows the capabilities of the Dialogue Designing Dialogue system in the areas of data base retrieval, information and report generation.

8.1: A Simple Dialogue Example

The following is a copy of the actual listing from the computer of both the

design phase, using the Dialogue Designing Dialogue System, and the run phase of the resulting Bibliography Dialogue System. It includes query statements showing the starting state of the data base, followed by the design phase, the run phase and final query statements showing the final state of the data base. The mark "***" has been inserted to indicate the transition between the four parts of this illustration. (User inputs are in italics, computer responses are in typewriter fonts.)

```
*** define relevant vocabularies in the ASK system

> classes: biblio,article,book
 The following new classes have been added:
   biblio  article  book
> Books and articles are biblios.
 books  articles  have been added to the class biblios.
> attributes: title,place of publication
 The following new individual attributes have been added:
   title  place of publication
> relation: author,keyword
 The following new individual relations have been added:
   author  keyword
> text: book list
 Enter text for book list:

       Book List

 The new text entity book list has been added.

*** the design phase of the Bibliography System

> new dialogue
 You are now in the dialogue designing dialogue. You may, at any
 time, type "help", "exit" or "wait".
>What user input should initiate this dialogue? new item
 When the user enters "new item" s/he will be at node 1 of the
 dialogue you are designing. Now define each node in turn.

 Designing node 1.
>What is the prompt message for this node? Title:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 1
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
```

```
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): f
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: Title of <1> is <1>.
>Action:
>Transfer to what node? 2
 Node 1 is completed.

 Designing node 2.
 The fields that have been assigned on entering the node are:
 <1> : Title:
>What is the prompt message for this node? Author:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 2
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:
   Enter the author of this new item.
>
 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: author of <1> is <2>.
>Action:
>Transfer to what node? 3
 Node 2 is completed.

 Designing node 3.
 The fields that have been assigned on entering the node are:
 <1> : Title:
 <2> : Author:
>What is the prompt message for this node? Is it an article? (yes/no):
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 3
```

```
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): st
>Enter a "help" message for your user at this node:
   If <1> is an article, you will be
> prompted for the name of the journal or
> anthology in which it appeared.
>
 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is <3> an initial segment of 'yes'?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action: <1> is an article.
>Action:
>Transfer to what node? 4
>Condition: otherwise
>Action: <1> is a book.
>Action: save in book list:"<2>, <1>"
>Action:
>Transfer to what node? 5
 Node 3 is completed.

 Designing node 4.
 The fields that have been assigned on entering the node are:
 <1> : Title:
 <2> : Author:
 <3> : Is it an article? (yes/no):
>What is the prompt message for this node? Journal:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 4
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done? a(ccept), r(eprompt): a
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
```

```
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: place of publication of <1> is <4>.
>Action: save in book list:"<2>, <1>, <4>"
>Action:
>Transfer to what node? 5
 Node 4 is completed.

 Designing node 5.
 The fields that have been assigned on entering the node are:
 <1> : Title:
 <2> : Author:
 <3> : Is it an article? (yes/no):
 <4> : Journal:
>What is the prompt message for this node? Keyword:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 5
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:
  Enter a keyword for this item. A simple
> carriage return will complete this item.
> You will be prompted for as many keywords
> as you wish.
>
 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): t
>Transfer to what node? 1
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), c(ondition-action-transfer): a
>Action: keyword of <1> is <5>.
>Action:
>Transfer to what node? 5
 Node 5 is completed.
 All nodes have been designed.
 You have completed the design of the new dialogue "new item".

*** the run phase of the Bibliography System

> new item
>Title: Ivanhoe
>Author: Scott
```

```
>Is it an article? (yes/no): help
  If Ivanhoe is an article, you will be prompted for the name
  of the journal or anthology in which it appeared.
>Is it an article? (yes/no): no
>Keyword: history
>Keyword: love
>Keyword: England
>Keyword:
>Title: Partitioned Networks
>Author: Hendrix
>Is it an article? (yes/no): yes
>Journal: Journal of the AAAI
>Keyword: semantic nets
>Keyword: natural language
>Keyword:
>Title: Tale of Two Cities
>Author: Dickens
>Is it an article? (yes/no): no
>Keyword: history
>Keyword: France
>Keyword:
>Title:
 You have finished the new item dialogue.

*** Query Statement

> List title, author and keywords of articles and books.
```

articles and books	title	author	keywords
book	Ivanhoe	Scott	history
	---	---	love
	---	---	England
	Tale of Two Cities	Dickens	history
	---	---	France
article	Partitioned Networks	Hendrix	semantic nets
	---	---	natural language

```
> Display book list.

    Book List

 Scott, Ivanhoe
 Hendrix, Partitioned Networks, Journal of the AAAI
 Dickens, Tale of Two Cities
 Display of book list has been completed.
>
```

8.2: Recursive Case

Our purpose in this example is to show the capability of handling recursion in the Dialogue Designing Dialogue System. To this end, suppose a user wishes to add to the data base the family tree for a person and his offspring. In order to keep the issue simple, we limit our considerations to just one side of the family; thus we will not include wives and sons-in-law. A simple family tree is shown in Figure 8.1.

```
          Adam
         /    \
       Bob     Bill
      /  \       \
  Clark  Clint    Cray
```

Fig. 8.1. A family tree of Adam

We present two example dialogues. The first builds the family tree depth first. It illustrates how one dialogue may recursively call another dialogue, possibly itself. The second builds the family tree breadth first. It illustrates the use of the queue mechanism for accomplishing the recursions.

8.2.1: Dialogue recursively calling dialogue

Suppose the family tree is to be built in depth-first order, that is: for any given family member, one of his sons, say John, is selected, then one of John's sons, etc., till we reach a member of the family that does not have a son. Then we back up one level and consider a second son, etc. When this member has no more sons, we back up one more level, etc. Referring to Figure 8.1, the order in which it will be built is: Adam, then Bob, Clark, Clint, Bill and Cray.

The following listing is the complete design phase for two Dialogue Systems: "family tree" and "family of <parameter>"; where the first one calls the second one, and the second one calls itself recursively. At each point where we "back up" a level, i.e., at the end of a recursion, we print out the context, namely, all the sons of the higher level person that have been entered so far so that the user will not get lost in the midst of building a big family tree. After the design phase, we include an example run phase and a query statement.

***** define relevant vocabularies in the ASK system**

```
> class: person
> relation: son
> Who are the sons of each person?
 There are no sons.
```

***** the design phase of the first dialogue system**

```
> new dialogue
 You are now in the dialogue designing dialogue. You may, at any
 time, type "help", "exit" or "wait".
>What user input should initiate this dialogue? family tree
 When the user enters "family tree" s/he will be at node 1 of
 the dialogue you are designing. Now define each node in turn.

 Designing node 1.
>What is the prompt message for this node? Who is the patriarch of
  this family?
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 1
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): r
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: <1> is a person.
>Action: call: family of <1>.
>Action: display: "You have finished <1>'s family tree."
>Action:
>Transfer to what node?
 Node 1 is completed.
 All nodes have been designed.
 You have completed the design of the new dialogue "family tree".

 The following dialogues remain to be designed:
  Dialogue System "family of <1>"
 Proceed with the design of one of these dialogues.
```

***** the design phase of the second dialogue system (depth-first)**

```
>What user input should initiate this dialogue? family of <1>
 What type of user response is expected for field <1>? nu(mber),
>no(un phrase), ti(me), te(xt), se(ntence) or st(ring)? no
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
 What conditions should the value of field <1> satisfy?
>Condition:
 When the user enters "family of <1>" s/he will be at node 1 of
 the dialogue you are designing. Now define each node in turn.

 Designing node 1.
>What is the prompt message for this node? Does <1> have a son?
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 2
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): st
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is <2> an initial segment of 'yes'?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action:
>Transfer to what node? 2
>Condition: otherwise
>Action:
>Transfer to what node?
 Node 1 is completed.

 Designing node 2.
 The fields that have been assigned on entering the node are:
 <1> : family of <1>
 <2> : Does <1> have a son?
>What is the prompt message for this node? Name of <1>'s son:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 3
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
```

```
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: <3> is a person.
>Action: display: <3> is son of <1>.
>Action: call: family of <3>.
>Action: display: "Sons of <1>:"
>Action: Who are sons of <1>?
>Action:
>Transfer to what node? 3
 Node 2 is completed.

 Designing node 3.
 The fields that have been assigned on entering the node are:
 <1> : family of <1>
 <2> : Does <1> have a son?
 <3> : Name of <1>'s son:
>What is the prompt message for this node? Does <1> have another son?
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 4
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): st
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is <4> an initial segment of 'yes'?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action:
>Transfer to what node? 2
>Condition: otherwise
>Action:
>Transfer to what node?
 Node 3 is completed.
 All nodes have been designed.
```

You have completed the design of the new dialogue "family of <1>".

***** the run phase of the family tree (depth-first method)**

```
> individual: Adam
> family tree
>Who is the patriarch of this family: Adam
>Does Adam have a son? yes
>Name of Adam's son: Bob
 Bob has been added as son of Adam.
>Does Bob have a son? yes
>Name of Bob's son: Clark
 Clark has been added as son of Bob.
>Does Clark have a son? no
 Sons of Bob:
 Clark
>Does Bob have another son? yes
>Name of Bob's son? Clint
 Clint has been added as son of Bob.
>Does Clint have a son? no
 Sons of Bob:
 Clark
 Clint
>Does Bob have another son? no
 Sons of Adam:
 Bob
>Does Adam have another son? yes
>Name of Adam's son: Bill
 Bill has been added as son of Adam.
>Does Bill have a son? yes
>Name of Bill's son? Cray
 Cray has been added as son of Bill.
>Does Cray have a son? no
 Sons of Bill:
 Cray
>Does Bill have another son? no
 Sons of  Adam:
 Bob
 Bill
>Does Adam have another son? no
 You have finished Adam's family tree.
 You have finished the family tree dialogue.
> Who are the sons of each person?
 person  son
 Adam    Bob
```

```
        Bill
Bob     Clark
        Clint
Bill    Cray
```

8.2.2: Using a queue to handle the recursion

The example presented in Section 8.2.1, above, used recursive calls to build the family tree in a depth first manner. It could just as well have used a stack or queue method to accomplish the recursion. We will illustrate the use of a queue by building the family tree in a breadth first fashion.

Suppose the family tree is to be built in breadth-first order, that is: for any given family member, all of his sons will be named; then this will be repeated for each of these sons in turn until all of the original person's grandsons are named, etc. Referring to Figure 8.1, the order in which it will be built is: Adam, then Bob, Bill, Clark, Clint and Cray.

```
> class: person
> relation: son
> Who are the sons of each person?
 There are no sons.

*** the design phase of the first dialogue system

> new dialogue
 You are now in the dialogue designing dialogue. You may, at any
 time, type "help", "exit" or "wait".
>What user input should initiate this dialogue? family tree
        o
        o
        o
(The remainder of the family tree design is the same as in Section 8.2.1)
        o
        o
        o
 The following dialogue remains to be designed:
 Dialogue System "family of <1>"


*** the design phase of the second dialogue system (breadth-first)
```

```
>What user input should initiate this dialogue? family of <1>
 What type of user response is expected for field <1>? nu(mber),
>no(un phrase), ti(me), te(xt), se(ntence) or st(ring)? no
 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): individual
 What conditions should the value of field <1> satisfy?
>Condition:
 When the user enters "family of <1>" s/he will be at node 1 of
 the dialogue you are designing. Now define each node in turn.

 Designing node 1.
>What is the prompt message for this node? Does <1> have a son?
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 2
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): st
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is <2> an initial segment of 'yes'?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action:
>Transfer to what node? 2
>Condition: otherwise
>Action:
>Transfer to what node? 4
 Node 1 is completed.

 Designing node 2.
 The fields that have been assigned on entering the node are:
 <1> : family of <1>
 <2> : Does <1> have a son?
>What is the prompt message for this node? Name of <1>'s son:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 3
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
 What type of noun phrase must the response be? (e.g., individual,
```

```
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: <3> is a person.
>Action: display: <3> is son of <1>.
>Action: push under <4>: <3>
>Action:
>Transfer to what node? 3
 Node 2 is completed.

 Designing node 3.
 The fields that have been assigned on entering the node are:
 <1> : family of <1>
 <2> : Does <1> have a son?
 <3> : Name of <1>'s son:
 <4> : push under <4>: <3>
>What is the prompt message for this node? Does <1> have another son?
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 5
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): st
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is <5> an initial segment of 'yes'?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action:
>Transfer to what node? 2
>Condition: otherwise
>Action: display: "Sons of <1>:"
>Action: Who are sons of <1>?
>Action:
>Transfer to what node? 4
 Node 3 is completed.
```

```
 Designing node 4.
 The fields that have been assigned on entering the node are:
 <1> : family of <1>
 <2> : Does <1> have a son?
 <3> : Name of <1>'s son:
 <4> : push under <4>: <3>
 <5> : Does <1> have another son?
>What is the prompt message for this node?
 No user interaction for this node.
 What should be done at this node? f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is queue <4> empty?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action:
>Transfer to what node?
>Condition: otherwise
>Action: assign to <1>:<4>
>Action: pop <4>
>Action:
>Transfer to what node? 1
 Node 4 is completed.
 All nodes have been designed.
 You have completed the design of the new dialogue "family of <1>".


*** the run phase of the family tree (using breadth-first method)

> individual: Adam
> family tree
>Who is the patriarch of this family: Adam
>Does Adam have a son? yes
>Name of Adam's son: Bob
 Bob has been added as son of Adam.
>Does Adam have another son? yes
>Name of Adam's son: Bill
 Bill has been added as son of Adam
>Does Adam have another son? no
 Sons of Adam:
 Bob
 Bill
>Does Bob have a son? yes
>Name of Bob's son: Clark
 Clark has been added as son of Bob.
```

```
>Does Bob have another son? yes
>Name of Bob's son: Clint
 Clint has been added as son of Bob
>Does Bob have another son? no
 Sons of Bob:
 Clark
 Clint
>Does Bill have a son? yes
>Name of Bill's son: Cray
 Cray has been added as son of Bill.
>Does Bill have another son? no
 Sons of Bill:
 Cray
>Does Clark have a son? no
>Does Clint have a son? no
>Does Cray have a son? no
 You have finished Adam's family tree.
 You have finished the family tree dialogue.
> Who are the sons of each person?
 person  son
 Adam    Bob
         Bill
 Bob     Clark
         Clint
 Bill    Cray
```

CHAPTER 9

EXAMINING, EDITING AND DELETING A DIALOGUE SYSTEM

For any existing Dialogue System, it is very likely that a user may occasionally desire to investigate its content and review how it was designed, to modify it without starting over the entire design procedure, or to simply delete it. To this end, there are three facilities: first, one that allows him to review the design of a given Dialogue System; the second, a Dialogue System for editing those Dialogue Systems that have been designed using the Dialogue Designing Dialogue; the third that allows him to delete a Dialogue System entirely. Though these three utilities are in parallel with the Dialogue Designing Dialogue, they are rather straightforward in their implementations. This is because the design of any existing Dialogue System is completely defined by its Dialogue Data Structure, which is the only structure that needs to be referenced, modified or deleted.

9.1: Examining a Dialogue

To initiate the utility for examining an existing Dialogue System, a user only needs to type the statement "Examine <dialogue> dialogue" where <dialogue> is the initiating statement of the Dialogue System to be examined. After that, the utility will display all the design information of the requested Dialogue System in an easily understood format. To illustrate this format, we examine the first dialogue example in Chapter 8 to show the complete listing provided by this utility.

```
> Examine new item dialogue
Following is the design listing of the "new item" Dialogue System:

Initiating statement: new item

Node 1:
    prompt: Title:
    assign to field: <1>, type: noun phrase
      when not in vocabulary: accept
      expected response: individual
```

```
    help message:
    nil response option: finish up
    good response option: action-transfer
      action: Title of <1> is <1>.
      transfer: 2

Node 2:
    prompt: Author:
    assign to field: <2>, type: noun phrase
      when not in vocabulary: accept
      expected response: individual
    help message: Enter the author of this new item.
    nil response option: reprompt
    good response option: action-transfer
      action: Author of <1> is <2>.
      transfer: 3

Node 3:
    prompt: Is it an article? (yes/no):
    assign to field: <3>, type: string
    help message: If <1> is an article, you will be
                  prompted for the name of the journal or
                  anthology in which it appeared.
    nil response option: reprompt
    good response option: condition-action-transfer
      condition: is <3> an initial segment of 'yes'?
      fail message:
      action: <1> is an article.
      transfer: 4
      condition: otherwise
      action: <1> is a book.
      action: save in book list: "<2>, <1>"
      transfer: 5

Node 4:
    prompt: Journal:
    assign to field: <4>, type: noun phrase
      when not in vocabulary: accept
      expected response: individual
    help message:
    nil response option: reprompt
    good response option: action-transfer
      action: reference of <1> is <4>.
      action: save in book list:"<2>, <1>, <4>"
      transfer: 5
```

```
Node 5:
    prompt: Keyword:
    assign to field: <5>, type: noun phrase
      when not in vocabulary: accept
      expected response: individual
    help message: Enter a keyword for this item. A simple
                  carriage return will complete this item.
                  You will be prompted for as many keywords
                  as you wish.
    nil response option: transfer
      transfer: 1
    good response option: action-transfer
      action: keyword of <1> is <5>.
      transfer: 5

You have completed the examination of the dialogue "new item".
>...
```

9.2: Editing a Dialogue

To initiate the second service utility for editing an existing Dialogue System, a user needs to type: "Edit <dialogue> dialogue", where <dialogue> is the initiating statement of the desired Dialogue System. In this editing dialogue, the user is asked which node he wishes to edit; he then enters the Dialogue Designing Dialogue node design sequence. The previous design of the indicated node is displayed as he progresses through the design sequence. Any default response (carriage return only) is interpreted as the maintenance of the previous design, thus only changes need to be entered. In this dialogue utility, all the "help", "exit", and "wait" still apply. If, when the user is asked which node he wishes to edit, he gives a number that is not the number of an existing node, then he is informed that this is a new node, and then enters the node design sequence in the normal way.

Again, we exemplify the first dialogue example in Chapter 8 and show the complete listing of the use of the utility.

```
> Edit new item dialogue
>What node do you wish to edit:  1
>prompt: Title:
>assign to field: <1>
>type: noun phrase
>when not in vocabulary: accept
>expected response: individual
>help message:
```

```
>nil response option: finish up
>good response option: action-transfer
>action: Title of <1> is <1>.
>transfer: 2   6
 Node 1 is completed.

>What node do you wish to edit:   6
 A new node is being added.
>What is the prompt message for this node:  Publishing date:
    o
    o
    o
>Transfer to what node:   2
 Node 6 is completed.

>What node do you wish to edit:
 You have completed the editing of the dialogue "new item".
```

After the modification, the new Dialogue Data Structure will be "saved" back in another disk area due to the nature of LIST_OUT. The utility will then use the new actual disk address to update the address content of the dialogue in the Dialogue Indirect Address. No modification on the calling addresses of the existing dialogues that call this modified Dialogue System is needed.

When the user responds with "*0*" to the "What node do you wish to edit? ", it means that the initiating statement is to be modified, and the editing dialogue will consider this case as the creation of a new Dialogue System. In such case, the editing dialogue will add the new initiating statement to the dictionary with the part of speech <dialogue>, write this new dialogue onto disk and add its actual disk address into a new Dialogue Indirect Address.

9.3: Deleting a Dialogue

To initiate the third service utility for deleting an existing Dialogue System, a user needs to type: "Delete <dialogue> dialogue", where <dialogue> is the initiating statement of the desired Dialogue System. After that, the system will indicate if the desired dialogue has been deleted or not. For example, if the user wants to delete the first and the second dialogue example in Chapter 8, he may utilize this facility and experience the following:

```
> Delete new item dialogue
 The new item dialogue has been deleted.
> Delete family of Adam dialogue
 The family of Adam dialogue has been called by the following dialogues:
 family tree
 The family of Adam dialogue thus can not be deleted.
>Next dialogue to be deleted:
```

The second of the above two examples has been rejected. This is because when the initiating statement <dialogue> is specified, its Dialogue Data Structure is checked. If this dialogue is called by other dialogues, then the current delete request will not be carried out. Rather, all the initiating statements of the dialogues that call this dialogue will be displayed instead. By trying to delete each of these displayed dialogues in turn, the user may eventually find out all the "root" dialogues and, deleting them, in a forward fashion, finally he will be able to delete the original one.

CHAPTER 10

CONCLUDING REMARKS

The technical achievement of this thesis is a problem oriented, natural language programming system, that allows a person with very little programming knowledge to implement quickly an efficient interface for the accomplishment of his specific, immediate task.

We know of no research that is related to the general Dialogue Designing Dialogue capability in the natural language and database area.

In the research area of using natural language as a programming language, we found that other research in this area has not been very successful because of the generality of the goals and the weakness of the underlying supporting environment. Since the Dialogue Designing Dialogue is based on a powerful natural language database system, when we limit the application of Dialogue Designing Dialogue to the programming of more responsive ways to use the system, and impose the dialogue techniques to control the use of natural language (particularly in the creation of loops), we then have achieved, in this circumscribed framework, a working natural language programming capability.

In the area of building expert systems, we found that research includes meta-dialogue systems, for example, EMYCIN and ROSIE. These systems do indeed superficially resemble the Dialogue Designing Dialogue; however, they arise from a very different technology. First, a goal oriented, production system is used in these systems. Its system structure is radically different from the grammar rules in a natural language data base system. Second, the knowledge base consists only of the production rules for the specific domain and specific application, and is therefore not available for other applications. Third, in an expert system dialogue, the system asks the questions and the user replies (with the exception that the user may ask why a particular decision was made), the user has no way to direct the dialogue into areas the system has not anticipated. Finally, due to the different expertise required in the design phase, it is very unlikely that users will also be the designers.

In a sense, a Dialogue System designed by the Dialogue Designing Dialogue is an expert system. For example, the Dialogue Data Structure retains user responses to be used in its decision structure. One can argue that ASK plus the Dialogue

Designing Dialogue provides a significant extension of expert systems. However, the production rule technology has been developed with efficiencies for its intended applications that the grammar rule/parsing technology can not achieve in similar applications. The Dialogue Designing Dialogue system might be said to produce "mini expert systems", but beyond that the technology is distinctly different. Of course, the Dialogue Designing Dialogue itself can rightfully be considered an expert system, though it does not use the production rule technology.

In Chapter 1, a list of criteria for programming languages was presented. We review that list here, showing that the Dialogue Designing Dialogue System meets these criteria.

- data type (data structure): the data structures available in the Dialogue Designing Dialogue are derived from those of the underlying ASK system. Over and above numbers, strings and truth value (i.e., boolean), they also include noun phrases naming individuals and texts, and corresponding classes, attributes and relations. The underlying data structures are highly complex; however, this complexity is entirely below the level of attention to the user of the system.

  The Dialogue Designing Dialogue includes a variety of data structures of its own, including nodes, fields, conditions, actions and messages.

- operation type: the operators of the system coincide with the semantic interpretive procedures of the ASK system plus a family of operators specific to the Dialogue Systems. The latter included: wait-resume, help, exit, call (of a dialogue) and the components of the condition-action-transfer sequences.

- statement: the prompt/response forms constitute a major class of statement types specific to the Dialogue Designing Dialogue and the Dialogue Systems designed by its use. Underlying these forms are complex procedures that give them meaning.

- parameter passing: a Dialogue System may contain parameters as part of its initiating statement. Such a Dialogue System may be called from other Dialogue Systems (including itself) by passing parameters through the technique of either call by address or call by value.

- storage management for storage allocation: the Dialogue Data Structure provides a highly formatted, versatile data structure for maintaining the current status of an ongoing dialogue. This Dialogue Data Structure has both static and dynamic properties, static in abstract format, but dynamic in that variable instantiations, etc., are maintained during run time in this structure.

It also contains a specially formatted stack for dialogues in recession. During the Dialogue Designing Dialogue, this stack is also used in implementing the call of one dialogue by another, including recursive calls. In this "dialogue_stack", the Dialogue Data Structures are maintained as in a push-down stack.

The Dialogue Designing Dialogue **Field** acts both as a local variable, stack or queue. The operations of "assign", "push" and "pop" on fields automatically build the appropriate data structures.

- recursive functions: the Dialogue Indirect Address mechanism allows a Dialogue System to call itself, directly or indirectly.
- program control structure: the node transfer (under the control of the designer), the "exit" subprocessor, and the prefix/prompt mechanism (internally to the system) provide the mechanisms for program control; the condition-action-transfer sequence provides the selection constructs (e.g., case) and iteration constructs (e.g., while). In addition, the proper use of node transfer also provides the iteration constructs (e.g., repeat).
- variables and scope rules: fields are for local use within a Dialogue System. No field is global. When a field is part of an initiating statement of a Dialogue System DS1 (e.g., initiating statement: "load <1>"), it is local to DS1. When another Dialogue System DS2 calls DS1 by address (e.g., calling statement in DS2: "Action: call: load <V5>"), the lack of global variables is handled by explicit transfer of values between the local variables involved. The ASK variables: GLOBAL^[dialogue_stack] and GLOBAL^[current_dialogue] are for global use in the general Dialogue Systems context. Type checking, thus scope, is done in many places, using the parse_list utility of the underlying system.
- language support: the rich functions in the ASK system provide numerous utilities to connect the Dialogue Designing Dialogue to the outside world.

In using the Dialogue Designing Dialogue, a user is always guided by the system's dialogue, thus he need only enter the highest level information needed to specify his program. Automatic type checking eliminates ambiguous assignments. The "programming language" is brief, one word responses or, in the case of condition and action statements, essentially natural language. In this latter case, the domain specific vocabulary and definitions are directly available to him. The system has no knowledge of any specific application; the user specifies such knowledge (including jargon) for himself in his own way, where he only needs to specify what to do rather than how to do in designing his problem oriented Dialogue System. Further,

discrepancies between the desired system and the produced system are far less likely to arise since the design and the use of a Dialogue System can be accomplished easily by the same person and thus no misinterpretation of the knowledge about the application should occur. Furthermore, a user can verify the Dialogue System immediately to see whether the system produced is what he desired (this has been called in the literature: "rapid prototyping"), and make modifications easily for unsatisfactory portions, using the facility of editing an existing dialogue. The resulting dialogue is automatically integrated into the ASK System and becomes available to a user immediately. The other two facilities for examining and deleting a dialogue provide for the user capabilities to review a complete dialogue design and to eliminate an unwanted dialogue and its associated dialogues, both are presented in a simple, clear format.

The two most likely design errors, in using such a high level system, are the use of undeclared variables and the introduction of infinite loops. Although these can not be entirely protected against, the Dialogue Designing Dialogue System gives informative warnings to the designer in both of these cases.

Clearly, the range of application can be wide and complex. The examples shown in the thesis demonstrate the versatile capabilities for handling simple work (e.g., adding new bibliography), recursive constructions (e.g., building a family tree), up to the data base retrieval and report generation (e.g., distributing complex invoice transaction activities). The existence of the underlying data base system, with its ability easily to add vocabulary and data, adds to the levels of complexity that may be handled.

For future work, graphics handling should be added to improve the capabilities of the current Dialogue Designing Dialogue system. For example, an action of "display: ...picture... " or "save in ...:...picture..." should be available in the design phase so that graphic images can either be output to the user or included in a document, where the locations, colors, display types (e.g., a pie distribution, a bar chart, etc.), picture processing, and the associated data that is to be displayed are specified just as action statements involving texts are handled currently.

"Spots" on touch screens, mouse or other cursor control should be specifiable so that these techniques can be used for prompt/responses. The designer should be able to specify simple, one word voice prompts and voice responses.

Special purpose programs and new object types can be easily added to ASK; such extensions could be simply and immediately adapted to a Dialogue System as long as the condition-action-transfer mechanisms contain corresponding grammar

rules to invoke these programs. The Dialogue Designing Dialogue System should be extended so that such extensions to the underlying system are easily incorporated in the designing of dialogues.

REFERENCES

[Barr81] Barr, Avron and Edward A. Feigenbaum, ***The Handbook of Artificial Intelligence,*** Volume 1, William Kaufmann Inc., Los Altos, CA., 1981.

[Barr82] Barr, Avron and Edward A. Feigenbaum, ***The Handbook of Artificial Intelligence,*** Volume 2, William Kaufmann Inc, Los Altos, CA., 1982.

[Brooks78] Brooks, F.P. Jr., ***The Mythical Man-Month,*** Addison-Wesley Publishing Co., 1978.

[Carlson80] Carlson, E.D. and W. Metz, "Integrating Dialog Management and Data Base Management," IBM Research Report RJ2738 (35013), Computer Science, 2/1/80.

[Davis76] Davis, R., ***Applications of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases,*** Ph.D. Thesis, Computer Science Department, Stanford University, Stanford, CA., 1976.

[Duda79] Duda, R.O., Gaschnig, J., and P. E. Hart, "Model Design in the PROSPECTOR Consultant System for Mineral Exploration," ***Expert Systems in the Micro-Electronic Age*** (ed. by D. Michie), Edinburgh: Edinburgh University Press, 1979.

[Fain81] Fain, J., Gorlin, D., Hayes-Roth, F., et al, ***The ROSIE Language Reference Manual,*** Technical Note N-1647-ARPA, Rand Corp., Santa Monica, CA., 1981

[Feigenbaum77] Feigenbaum, Edward A., "The Art of Artificial Intelligence: Themes and Case Studies In Knowledge Engineering," ***Proceedings of the 5th International Joint Conference on Artificial Intelligence,*** Cambridge, Massachusetts, pp.1014-1029, 1977.

[Gershman81] Gershman, A., "Figuring Out What The User Wants - Steps Toward an Automatic Yellow Pages Assistant," ***Proceedings of the 7th International Joint Conference on Artificial Intelligence,*** Vancouver, pp.423-425, August 1981.

[Grosz83] Grosz, B.J., "TEAM: A Transportable Natural-Language Interface System," ***Proceedings of Applied Natural Language Processing,*** Santa Monica, CA., pp.39-45, February 1983.

[Halpern81] Halpern, P. and J.W. Rettberg, "User-Definable Software Applied To a Real-Time Ambient Air Quality Monitoring System," ***IBM Systems Journal,*** V20, N1, pp.86-103, 1981.

[Hayes-Roth81] Hayes-Roth, F., Gorlin, D., Rosenschein, S., Sowizral, H. and D. Waterman, ***Rational and Motivation for ROSIE***, Technical Note N-1648-ARPA, Rand Corp., Santa Monica, CA, 1981

[Hayes-Roth83] Hayes-Roth, Frederick, Waterman, Donald A. and Douglas B. Lenat, ***Building Expert Systems***, Addison-Wesley, 1983.

[Heidorn75] Heidorn, George E., ***Simulation Programming Through Natural Language Dialogue***, Amsterdam: North-Holland, 1975.

[Heidorn76] Heidorn, George E., "Automatic Programming Through Natural Language Dialogue: A Survey," ***IBM Journal of Research and Development***, pp.302-313, July, 1976.

[Heidorn77] Heidorn, George E., "The End of the User Programmer?" ***The Software Revolution, Infotech State of the Art Conference***, Copenhagen, Denmark, 1977.

[Heidorn78] Heidorn, George E., "Natural Language Dialogues for Managing an On-Line Calendar," ***Proceedings of 1978 ACM Annual Conference***, Washington, D.C., pp.45-52, December 1978.

[Hendrix77] Hendrix, Gary G. and E. D. Sacerdoti, "The LIFER Manual: A Guide To Building Practical Natural Language Interface," AI Center Technical Note 138, SRI International, Menlo Park, CA., February 1977.

[Hendrix81] Hendrix, Gary G. and E. D. Sacerdoti, "Natural-Language Processing: The Field in Perspective," ***BYTE***, pp.304-352, September 1981.

[James80] James, E.B., "The User Interface," ***Computer Journal***, V23, N1, pp.22-28, 1980.

[Knornatowski80] Knornatowski, J. and G. Cheung: ***Users Manual***, CSRG Technical Note, University of Toronto, 1980.

[Knuth73] Knuth, Donald E., ***The Art of Computer Programming,***, Volume 1, Second Edition, Addition-Wesley, Reading, Massachusetts, 1973.

[Lenner82] Lenner, Eric J., "Automating Programming,", pp.28-33; "Programming for Nonprogrammers," pp.33-38; both are in ***IEEE Spectrum***, V19, N8, August 1982.

[Marburger81] Marburger, H., Neumann, B., and H. Novak, "Natural Language Dialogue About Moving Objects in an Automatically Analyzed Traffic Scene," ***Proceedings of the 7th International Joint Conference on Artificial Intelligence***, Vancouver, pp.49-51, August 1981.

[Martin73] Martin, James, ***Design of Man-Computer Dialogue***, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[Papachristidis83] Papachristidis, Alexender, "Heterogeneous Data Base Access", Ph.D. Thesis, California Institute of Technology, December 1983.

[Robinson82] Robinson, J. J., "DIAGRAM: A Grammar for Dialogues," ***Communications of the ACM***, V25, N1, pp.24-47, January 1982.

[Shneiderman79] Shneiderman, B., "Human Factors Experiments in Designing Interactive Systems," ***Computer***, V12, N12, pp.9-19, 1979.

[Shortliffe76] Shortliffe, E. H., ***Computer-Based Medical Consultations: MYCIN***, New York: North-Holland, 1976.

[Thompson83a] Thompson, Bozena H. and Frederick B. Thompson, "Introducing ASK, A Simple Knowledgeable System," ***Proceedings of Applied Natural Language Processing***, Santa Monica, CA., pp.17-24, February, 1983.

[Thompson83b] Thompson, Bozena H., Thompson, Frederick B. and Tai-Ping Ho, "Knowledgeable Contexts for User Interaction," ***Proceedings of National Computer Conference***, Anaheim, CA., pp.29-38, May 1983.

[Thompson84] Thompson, Bozena H. and Frederick B. Thompson, ***ASK: A Simple Knowledgeable System (documentation of the ASK System)***, forthcoming.

[Tsichritzis82] Tsichritzis, D., "Form Management," ***Communications of ACM***, V25, N7, pp.453-478, July 1982.

[van Melle80] van Melle, W., ***A Domain Independent System That Aids in Constructing Consultation Programs***, Ph.D. thesis, Computer Science Department, Stanford University, 1980.

[Waldinger69] Waldinger, R. J. and R. C. T. Lee, "PROW: A Step Towards Automatic Program Writing," ***Proceedings of the First Joint Conference on Artificial Intelligence***, Washington, D.C., pp.241-252, 1969.

[Wulf81] Wulf, William A., Shaw, Mary, Hilfinger, Paul N. and Lawrence Flon, ***Fundamental Structures of Computer Science***, Addition-Wesley, Reading, Massachusetts, 1981.

APPENDIX A

The Design Phase Protocol with all Help Messages

```
>What user input should initiate this dialogue? help
 You are starting the design of a new dialogue. As you proceed with
 your design, you may at any time type "help" (to find out what to
 do next and what the consequences of your action will be), "exit"
 (which will immediately terminate this design), or "wait" (in which
 case you will exit this design dialogue; you can then re-enter the
 design dialogue again at the point where you typed "wait" by typing
 "resume").
 At this point you should enter the words the user should use when
 he wishes to initiate the dialogue you are designing. This
 initiating statement may contain parameters, indicated by field
 symbols, e.g., <n>. The user, of course, will replace them by actual
 values; these user entered values will then become the values of the
 indicated fields. Examples of initiating statements:
          "new item",   "load <1>".

>What is the prompt message for this node? help
 The system issues a prompt message to the user, indicating the nature
 of the response it expects. You are to enter the prompt message which
 will be issued to your user at this node. It may include field
 symbols which will be replaced by the literals of the associated user
 responses. Examples are: "Author: ", "More? (y/n): ", "Please enter
 text for <2>: ". If you wish no user interaction at this node,
 respond with nil (only carriage return).

 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: help
 You may wish to make use of the user's response to this prompt in
 specifying how it and further responses are to be processed. For
 example, you may wish to check that the response satisfies certain
 conditions, such as that it is a member of a certain class. For
 these purposes, you will want to assign it to a "field". A field is
 specified by "<n>" where n is an integer, e.g., "<8>" indicates
 field 8. If you assign the user's response to field <n> here, then
 you can subsequently refer to this response, for example: "<n> is a
 ship.". The selection of the field number n is up to you.

 What type of user response is expected? nu(mber), no(un phrase),
```

```
>ti(me), te(xt), se(ntence) or st(ring): help
 A user response must be of one of the following types:
   o number, e.g., "3.6", "5 feet";
   o noun phrase, e.g., "Boston", "home port of the Maru";
   o time, e.g., "July 16, 1981", "tomorrow", "3 seconds from now";
   o text, in which case whatever is entered by the user will be
      accepted as a body of text, available for subsequent filing,
      editing, etc.;
   o sentence, e.g., "List the destination of each ship.";
   o string, so that you can make further tests such as in the
     selection of an option, e.g., as in the response to the prompt:
     "B(ook), A(rticle) or P(aper): ".

 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): help
 Under some circumstances, you may wish to consider the user's
 response as possibly a new word to be added to the vocabulary. If so
 enter a(ccept); you will then be asked for the type you expect this
 new word to be, e.g., individual, class, etc. If you enter r(eprompt),
 the user will be informed that his response is not understood and he
 will be reprompted.

 What type of noun phrase must the response be? (e.g., individual,
>text class, number attribute, etc.): help
 Since the user's response may be added as a new vocabulary item,
 enter the expected type which you would use if you were entering the
 new item yourself, as for example: "Create a class named xxx". What
 type of noun phrase response should be expected >(e.g., individual,
 text class, number attribute, etc.): individual

>Enter a "help" message for your user at this node: help
 If the user responds to this node's prompt by "help", what message
 should be displayed? If you respond here with nil (only carriage
 return), then the help message will be "No help available".
 Terminate your help message by (exec) (cont).

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): help
   If the user gives a nil response, then the design options
 available to you are:
   o reprompt: cycle back, reissuing the prompt for this node;
   o finish up: mark this node as a terminal node for user
     interaction, schedule the completion of all indicated actions,
     and return of control to normal processing.
   o transfer: transfer directly to another node, you will be
```

prompted for its number.

- o action-transfer: you will be prompted for actions that should be scheduled at this node, and then for the number of the node to which to transfer.
- o condition-action-transfer: you will be prompted for a condition and for the actions and transfer to be taken if the condition is satisfied.

What should be done if the user gives a good response? f(inish up), >t(ransfer), a(ction-transfer), or c(ondition-action-transfer): *help*
If the user gives a good response, then the design options available to you are:

- o finish: mark this node as a terminal node for user interaction, schedule the completion of all indicated actions, and return of control to normal processing.
- o transfer: transfer directly to another node, you will be prompted for its number.
- o action-transfer: you will be prompted for actions that should be scheduled at this node, and then for the number of the node to which to transfer.
- o condition-action-transfer: you will be prompted for a condition and for the actions and transfer to be taken if the condition is satisfied.

What should be done at this node? f(inish up), t(ransfer), >a(ction-transfer), or c(ondition-action-transfer): *help*
Since there is no user interaction at this node, the design options available to you are:

- o finish: mark this node as a terminal node for user interaction, schedule the completion of all indicated actions, and return of control to normal processing.
- o transfer: transfer directly to another node, you will be prompted for its number.
- o action-transfer: you will be prompted for actions that should be scheduled at this node, and then for the number of the node to which to transfer.
- o condition-action-transfer: you will be prompted for a condition and for the actions and transfer to be taken if the condition is satisfied.

>Condition: *help*
Enter an ASK English "yes/no" question, possibly including field symbols; for example: "Is <2> a ship?", "Is inventory of <3> at least <5>?", "Is <3> an initial segment of 'yes'?"; for checking fields that are being used as stacks or queues: "Is stack <4> empty?". You will then be prompted for the actions to be taken

if the answer to this question is "yes", i.e., if the condition is satisfied. Subsequently, you will be prompted for what to do if the condition is not satisfied. If you wish to specify actions to be taken without any further condition, or this is the last clause of an "if ... then ... , otherwise ...." sequence, then just enter: "otherwise".

Enter a diagnostic message for the user in case the condition is
>not satisfied: *help*
If the condition, in reference to user responses, is not satisfied, you may wish an interpretation of the situation, or diagnostic information, to be given to the user; enter such a message here. It may include field symbols, which will be replaced by the literals of the associated user responses. If you enter a nil response, no diagnostic will be given the user.

>Action: *help*
Indicate the actions you wish the system to take. After entering an action, you will be prompted for another action until you give a nil (carriage return only) response. There are eleven types of actions which you can specify:

1: Updating the database: an ASK declarative sentence, possibly including field symbols, e.g., "Color of <3> is <5>.", "Change age of <2> to <7>.".
2: Display the update: same as (1) with the addition that the confirmation of the update will be displayed, e.g., "Display: Author of <3> is <5>.".
3: Display information to user: any ASK English query, e.g., "List the length and width of each <3>.", "What is age of each <4>?".
4: Display message to user: any message enclosed in double quotes; included field symbols will be replaced by their literal values; e.g., "Display:"When you finish, send a copy of <3> to <4>." ".
5: Save information as text: a statement of the form: "Save in ...name of text object...:...an ASK English query...". the information, as text, will be placed at the end of indicated text; e.g., "Save in load plan of <2>: List cargo type of each cargo of <2>.".
6: Save message as text: same as 5, except text to be saved is in quotes; e.g., "Save in load plan of <2>:"CARGO LOADING PLAN FOR <2>:" "
7: Field assignment: a statement: "Assign to <n>: ... an ASK expression ... "; this may be used to reassign a field's value or introduce a new field.
8: Add an element into a stack: a statement: "Push on --a stack field -- : -- any ASK English phrase --"; e.g., "Push on <3>: <5>"
9: Add an element into a queue: a statement: "Push under --a stack

field-- : -- any ASK English phrase --"; e.g., "Push under <3>: <5>"

10: Remove an element from a stack or from a queue: a statement: "Pop --a stack field--"; e.g., "Pop <3>"

11: Call another Dialogue System: a Statement: "Call: -- The initiating statement of the Dialogue System to be called --", e.g., "call: load Alamo". If the initiating statement of the dialogue you wish to call contains parameters, e.g., "load <4>", you may either call by value, e.g., "load Alamo", "load <6>", or call by address, e.g., "load <V6>" (where <6> is a field of the dialogue you are designing, the V indicating that the value of field <4> of the called dialogue will be passed back to field <6> of the calling dialogue).

>What condition should the value of fields <n>... satisfy? *help*

When someone types in the initiating statement for this new dialogue, he will use specific names in place of the field symbols. You may wish to specify conditions these values should satisfy. For example, in the initiating statement: "load <1>", it may be that the value given for field <1> must always be a ship. If you wish to impose such a condition, state it here as a yes/no English question, e.g., "Is <1> a ship?".

APPENDIX B
Data Base Retrieval and Report Generation

Sometimes a single event triggers other events. An invoicing system, for example, is such a case. When we file an invoice for a customer's order, we would also like a whole series of actions to be taken, e.g., reduction in inventory records, shipping and warehouse instructions, billing and accounting. Based on the "invoicing operation" system shown in [Tsichritzis82], a Dialogue System to accomplish all of these actions can be designed using the Dialogue Designing Dialogue system. The example shows the capabilities of the Dialogue Designing Dialogue system in the areas of data base retrieval, information and report generation.

To start with, the formats of each of the files to be manipulated by the dialogue are presented. This is followed by the design phase and the run phase of the Invoice Transaction Dialogue System.

B1. Listing of Files (Some of the contents are filled in for illustrative purposes.)

File A: Invoice Template

```
* Invoice #: (AA1234)   Date: (today's date)   Clerk: (operator)
  Customer: (ABC Company)
  Address: (307 S. Wilson Ave   Pasadena, CA 91106)

  Item: (19" color TV with remote control)    Quantity: (10)
  Manufacturer: (RCA)
  Unit Cost: $(...)   Unit Weight: (...)  Shipping Charge: $(...)
  Tax: $(...)  Cost: $(....)          Weight: (...)

  Item: (SL-5800 video recorder)              Quantity: (8)
  Manufacturer: (Sony)
  Unit Cost: $(...)   Unit Weight: (...)  Shipping Charge: $(...)
  Tax: $(...)  Cost: $(....)          Weight: (...)

  Total Tax: $(...)      Total Amount: $(...)
```

File B: Order Template

```
* From (ABC Company)    Reference (AA1234)
                        Date (today's date)

  Order the following items:
     (10) of (19" color TV with remote control) produced by (RCA)
     (8) of (SL-5800 video recorder) produced by (Sony)

  Please bill and ship merchandise to following address:
     (307 S. Wilson Ave    Pasadena, CA 91106)                    *
```

File C: Shipping Template

```
*  (ABC Company)
   (307 S. Wilson Ave      Pasadena, CA  91106)

   Dear Sir,

   Following your order (AA1234) you will find enclosed the
   following items:
     (10) of (19" color TV with remote control) produced by (RCA)
     (8) of (SL-5800 video recorder) produced by (Sony)

   We will bill you separately. Thank you for being our customer
                                                  XYZ Distributing  *
```

File D: Warehouse Template

```
*  Load (10) of (19" color TV with remote control) produced by
        (RCA)  for (ABC Company) reference (AA1234)
   Load (8) of (SL-5800 video recorder) produced by
        (Sony) for (ABC Company) reference (AA1234)               *
```

File E: Billing Template

(too long to copy all; please reference original format in [Tsichritzis82])

File F: Accounts Payable Template

"Customer: (ABC Company) Invoice: (AA1234) Date: (order date)
Amount Payable: $(...)

File G: Purchase Template

"Please purchase (quantity) of (item) produced by
(manufacturer). Reported by (logon id - operator) on (date)."

B2. Design Phase

```
>new dialogue
 You are now in the dialogue designing dialogue. You may, at any
 time, type "help", "exit" or "wait".
>What user input should initiate this dialogue? invoice system
 When the user enters "new item" s/he will be at node 1 of the
 dialogue you are designing. Now define each node in turn.

 Designing node 1.
>What is the prompt message for this node? Invoice number:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 1
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
 What type of noun phrase must the response must be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:
 Enter a valid invoice number here.
>
 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: assign to <8>: today
>Action: assign to <9>: logon identifier
>Action: assign to <16>: 0
>Action: assign to <17>: 0
```

```
>Action: save in file A:"Invoice #: <1>     Date: <8>      Clerk: <9>"
>Action:
>Transfer to what node? 2
 Node 1 is completed.

 Designing node 2.
 The fields that have been assigned on entering the node are:
 <1>: Invoice number:
 <8>: today
 <9>: logon identifier
 <16>: 0
 <17>: 0
>What is the prompt message for this node? Customer:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 2
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): a
 What type of noun phrase must the response must be? (e.g., individual,
>text class, number attribute, etc.): individual
>Enter a "help" message for your user at this node:
 If this is a new customer, his address will be asked; if this
>is an old customer, his credit status and address will be shown.
>
 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: is <2> not a customer?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action: assign to <19>: 0
>Action: Credit of <2> is 0.
>Action: <2> is a customer.
>Action: save in file B:"From <2>              Reference <1>"
>Action: save in file B:"                      Date <8>"
>Action: save in file B:" "
>Action: save in file B:"Order the following items:"
>Action:
>Transfer to what node? 3
>Condition: otherwise
>Action: assign to <3>: address of <2>
>Action: assign to <19>: credit of <2>
```

```
>Action: display: "<2> has a credit balance of $<19>."
>Action: display: "Address of <2> is <3>."
>Action: save in file A:"Customer: <2>"
>Action: save in file A:"Address: <3>"
>Action: save in file A:" "
>Action: save in file B:"From <2>              Reference <1>"
>Action: save in file B:"                      Date <8>"
>Action: save in file B:" "
>Action: save in file B:"Order the following items:"
>Action: save in file C:"<2>"
>Action: save in file C:"<3>"
>Action: save in file C:" "
>Action: save in file C:"Dear Sir,"
>Action: save in file C:" "
>Action: save in file C:"Following your order <1> you will
          find enclosed the"
>Action: save in file C:"following items:"
>Action:
>Transfer to what node? 4
 Node 2 is completed.

 Designing node 3.
 The fields that have been assigned on entering the node are:
 <1>: Invoice number:
 <2>: Customer:
 <8>: today
 <9>: logon identifier
 <16>: 0
 <17>: 0
 <19>: 0
>What is the prompt message for this node? Customer address:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 3
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): te
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: save in file A:"Customer: <2>"
>Action: save in file A:"Address: <3>"
>Action: save in file A:" "
>Action: save in file C:"<2>"
```

```
>Action: save in file C:"<3>"
>Action: save in file C:" "
>Action: save in file C:"Dear Sir,"
>Action: save in file C:" "
>Action: save in file C:"Following your order <1> you will
          find enclosed the"
>Action: save in file C:"following items:"
>Action:
>Transfer to what node? 4
 Node 3 is completed.

 Designing node 4.
 The fields that have been assigned on entering the node are:
 <1>: Invoice number:
 <2>: Customer:
 <3>: Customer address:
 <8>: today
 <9>: logon identifier
 <16>: 0
 <17>: 0
 <19>: 0
>What is the prompt message for this node? Item:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 4
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): r
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): t
>Transfer to what node? 7
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): t
>Transfer to what node? 5
 Node 4 is completed.

 Designing node 5.
 The fields that have been assigned on entering the node are:
 <1>: Invoice number:
 <2>: Customer:
 <3>: Customer address:
 <4>: Item:
 <8>: today
```

```
 <9>: logon identifier
 <16>: 0
 <17>: 0
 <19>: 0
>What is the prompt message for this node? Manufacturer:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 5
 What type of user response is expected? nu(mber), no(un phrase),
>ti(me), te(xt), se(ntence) or st(ring): no
 If the user's response is not understandable, presumably not in the
>vocabulary, what should be done: a(ccept) or r(eprompt): r
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): a
>Action: assign to <10>: unit cost of <4> produced by <5>
>Action: assign to <11>: unit weight of <4> produced by <5>
>Action: assign to <18>: inventory of <4> produced by <5>
>Action: display: "Unit cost of <4> produced by <5> is $<10>."
>Action: display: "We currently have <18> of them on hand."
>Action:
>Transfer to what node? 6
 Node 5 is completed.

 Designing node 6.
 The fields that have been assigned on entering the node are:
 <1>: Invoice number:
 <2>: Customer:
 <3>: Customer address:
 <4>: Item:
 <5>: Manufacturer:
 <8>: today
 <9>: logon identifier
 <10>: unit cost of <4> produced by <5>
 <11>: unit weight of <4> produced by <5>
 <16>: 0
 <17>: 0
 <18>: inventory of <4> produced by <5>
 <19>: 0
>What is the prompt message for this node? Quantity:
 If you wish the response to this prompt to be assigned to a field,
>then give the field number here: 6
 What type of user response is expected? nu(mber), no(un phrase),
```

```
>ti(me), te(xt), se(ntence) or st(ring): nu
>Enter a "help" message for your user at this node:

 What should be done if the user returns a nil (carriage return only)
 response to this node? r(eprompt), f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): r
 What should be done if the user gives a good response? f(inish up),
>t(ransfer), a(ction-transfer), or c(ondition-action-transfer): c
>Condition: <18> is greater than <6>?
 Enter a diagnostic message for the user in case the condition is
>not satisfied:

>Action: assign to <18>: <18>-<6>
>Action: The inventory of <4> produced by <5> is <18>.
>Action: assign to <14>: <10>*<6>
>Action: assign to <13>: <14>*0.06
>Action: assign to <15>: <11>*<6>
>Action: assign to <12>: shipping charge whose weight is <15>
>Action: assign to <16>: <16>+<13>
>Action: assign to <17>: <17>+<14>+<12>
>Action: display: "The cost is $<14>, tax is $<13>, and the
                  shipping charge is $<12>."
>Action: assign to <19>: <19>-<14>-<13>-<12>
>Action: display: "The customer <2> has a credit balance of
                  $<19>."
>Action: save in file A:"Item: <4>          Quantity: <6>"
>Action: save in file A:"Manufacturer: <5>"
>Action: save in file A:"Unit Cost: $<10>    Unit Weight: <11>
     Shipping Charge: $<12>"
>Action: save in file A:"Tax: $<13>   Cost: $<14>
          Weight: <15>"
>Action: save in file A:" "
>Action: save in file B:"   <6> of <4> produced by <5>"
>Action: save in file C:"   <6> of <4> produced by <5>"
>Action: save in file D:"Load <6> of <4> produced by"
>Action: save in file D:"  <5> for <2> reference <1>"
>Action:
>Transfer to what node? 4
>Condition: otherwise
>Action: display: "Insufficient inventory, this item has only <18>."
>Action: display: "ask the customer to reduce the order."
>Action: assign to <20>: regular purchase quantity of <4> produced
                     by <5>
>Action: save in file G:"Please purchase <20> of <4> produced by"
>Action: save in file G:"<5>. Reported by <9>, on <8>"
>Action:
```

```
>Transfer to what node? 6
 Node 6 is completed.

 Designing node 7.
 The fields that have been assigned on entering the node are:
 <1>: Invoice number:
 <2>: Customer:
 <3>: Customer address:
 <4>: Item:
 <5>: Manufacturer:
 <6>: Quantity:
 <8>: today
 <9>: logon identifier
 <10>: unit cost of <4> produced by <5>
 <11>: unit weight of <4> produced by <5>
 <12>: shipping charge whose weight is <15>
 <13>: <14>*0.08
 <14>: <10>*<5>
 <15>: <11>*<5>
 <16>: 0
 <17>: 0
 <18>: inventory of <4> produced by <5>
 <19>: 0
 <20>: regular purchase quantity of <4> produced by <5>
>What is the prompt message for this node?
 No user interaction for this node.
 What should be done at this node? f(inish up), t(ransfer),
>a(ction-transfer), or c(ondition-action-transfer): a
>Action: Credit of <2> is <19>.
>Action: assign to <17>: <17>+<16>
>Action: display: "Total amount payable is $<17>"
>Action: save in file A:"Total Tax: $<16>   Total Amount: $<17>"
>Action: save in file B:" "
>Action: save in file B:"Please bill and ship merchandise to
          following address:"
>Action: save in file B:"   <3>"
>Action: save in file C:" "
>Action: save in file C:"We will bill you separately. Thank
          you for being our customer"
>Action: save in file C:"                XYZ Distributing"
>Action: save in file F:"Customer: <2>            Invoice: <1>
          Date: <8>"
>Action: save in file F:"Amount Payable: $<17>"
>Action:
>Transfer to what node?
 Node 7 is completed.
```

All nodes have been designed.
You have completed the design of the new dialogue "invoice system".

B3. Run Phase

```
>invoice system
>Invoice number: help
 Enter a valid invoice number here.
>Invoice number: AA1234
>Customer: help
 If this is a new customer, his address will be asked; if this is
 an old customer, his credit status and address will be shown.
>Customer: ABC Company
 ABC Company has a credit balance of $9000.
 Address of ABC Company is 307 S. Wilson Ave Pasadena, CA 91106.
>Item: 19" color TV with remote control
>Manufacture: RCA
 Unit cost of 19" color TV with remote control produced by
 RCA is $350.
 We currently have 48 of them on hand.
>Quantity: 10
 The cost is $3500, tax is $210, and the shipping charge is $200.
 The customer ABC Company has a credit balance of $5090.
>Item: Sony SL-5800 video recorder
>Manufacture: Sony
 Unit cost of SL-5800 video recorder produced by Sony is $500.
 We currently have 28 of them on hand.
>Quantity: 8
 The cost is $4000, tax is $240, and the shipping charge is $200.
 The customer ABC Company has a credit balance of $650.
>Item:
 Total amount payable is $8350.
 You have finished the invoice system dialogue.
```