TEXAS INSTRUMENTS

NATURAL LANGUAGE MENU SYSTEM USER'S GUIDE

EXPLORER™ NATURAL LANGUAGE MENU SYSTEM USER'S GUIDE

MANUAL REVISION HISTORY

Explorer™ Natural Language Menu System User's Guide (2243202-0001)

Original Issue ..... June 1985

The computers, as well as the programs that TI has created to use with them, are tools that can help people better manage the information used in their business; but tools—including TI computers—cannot replace sound judgment nor make the manager's business decisions.

Consequently, TI cannot warrant that its systems are suitable for any specific customer application. The manager must rely on judgment of what is best for his or her business.

The system-defined windows shown in this manual are examples of the software as this manual goes into production. Later changes in the software may cause the windows on your system to be different from those in the manual.

Texas Instruments Incorporated
Data Systems Group
P.O. Box 2909 M/S 2151
Austin, Texas 78769

Printed in U.S.A.

THE EXPLORER™ SYSTEM SOFTWARE MANUALS

Mastering the Explorer Environment

Explorer Technical Summary	2243189-0001
Explorer Operations Guide	2243190-0001
Explorer Zmacs Editor Tutorial	2243191-0001
Explorer Glossary	2243134-0001
Explorer Communications User's Guide	2243206-0001
Explorer Diagnostics	2533554-0001
Explorer Master Index to Software Manuals	2243198-0001
Explorer System Software Installation	2243205-0001

Programming With the Explorer

Explorer Programming Primer	2243199-0001
LISP, 2nd Edition, by Winston and Horn	2249963-0001
Explorer Lisp Reference	2243201-0001
Explorer Zmacs Editor Reference	2243192-0001
Explorer Programming Concepts and Tools	2243130-0001
Explorer Window System Reference	2243200-0001
Explorer Command Interface Toolkit User's Guide	2243197-0001

Explorer Toolkits

Explorer Natural Language Menu System User's Guide	2243202-0001
Explorer Relational Table Management System User's Guide	2243203-0001
Explorer Graphics Toolkit User's Guide	2243195-0001
Explorer Grasper User's Guide	2243135-0001
Explorer Prolog Toolkit User's Guide	2243204-0001
Programming in Prolog, by Clocksin and Mellish	2249985-0001
Scribe® User's Manual	2249981-0001

System Software Internals

Explorer System Software Design Notes	2243208-0001

THE EXPLORER™ SYSTEM HARDWARE MANUALS

System Level Hardware Publications

Explorer Unpacking and Inventory 2243216-0001
Explorer 7-Slot System Installation 2243140-0001
Explorer System Field Maintenance 2243141-0001

System Enclosure Hardware Publications

Explorer 7-Slot System Enclosure General Description 2243143-0001
Explorer 2-Megabyte Memory General Description 2243147-0001
Explorer Memory General Description 2533592-0001
Explorer Processor General Description 2243144-0001
Explorer Display Unit General Description 2243151-0001
Explorer System Interface General Description 2243145-0001

Mass Storage Hardware Publications

Explorer Mass Storage Enclosure General Description 2243148-0001
Explorer Winchester Disk Formatter (ADAPTEC) Supplement to Explorer Mass Storage Enclosure General Description 2243149-0001
Explorer Winchester Disk Drive (Maxtor) Supplement to Explorer Mass Storage Enclosure General Description 2243150-0001
Explorer Cartridge Tape Drive (Cipher) Supplement to Explorer Mass Storage Enclosure General Description 2243166-0001
Explorer Cable Interconnect Board (2236120-0001) Supplement to Explorer Mass Storage Enclosure General Description 2243177-0001
Explorer NuBus™ Peripheral Interface General Description (NUPI board) 2243146-0001

Mass Storage Hardware Vendor Publications

Series 540 Cartridge Tape Drive Product Description, Cipher Data Products, Inc., Bulletin Number 01-311-0284-1K (¼-inch tape drive) .. 2249997-0001
MT01 Tape Controller Technical Manual, Emulex Corporation, part number MT0151001 (formatter for the ¼-inch tape drive) 2243182-0001
XT-1000 Service Manual, 5¼-inch Fixed Disk Drive, Maxtor Corporation, part number 20005 (5¼-inch Winchester disk drive, 112 megabytes) 2249999-0001
ACB-5500 Winchester Disk Controller User's Manual, Adaptec, Inc., (formatter for the 5¼-inch Winchester disk drive) 2249933-0001

Optional Equipment Hardware Publications

Explorer NuBus Ethernet® Controller General Description 2243161-0001
Model 855 Printer Operator's Manual 2225911-0001
Model 855 Printer Technical Reference Manual 2232822-0001
Model 855/856 Printer Maintenance Manual 2225914-0001

ABOUT THIS MANUAL

Introduction

This manual contains information about the Texas Instruments Natural Language Menu system, which is designed to operate on the Explorer system.

Purpose

The intent of this manual is to provide substantial reference information on the utilities that comprise the Natural Language Menu (NLMenu) system.

This document is intended for application designers who will use the Explorer system and the NLMenu software to create and maintain natural language interfaces to specific application packages. The individuals who use the interfaces created by the designer are referred to in this manual as *interface users*. However, it is the nature of the NLMenu software system that interface users can themselves become designers of their own interfaces or tailor existing interfaces to meet changing needs.

Scope

The scope of this manual assumes that you:

- Are familiar with computers and computer languages, and with Lisp or Lisp machines
- Have read the *Explorer Technical Summary* and the *Explorer Operations Guide*
- Have access to an installed and working Explorer system

Knowledge of linguistic concepts is not assumed, and familiarity with context-free grammars is very much to the reader's advantage.

Contents of This Manual

This manual includes an index and a glossary and is organized into the following sections.

Section 1: Overview — Describes an interface-user view of a natural language interface and the main components of the NLMenu toolkit that the application designer uses to create such interfaces.

Section 2: Using Natural Language Menu — Discusses the default components of the NLMenu system; describes the operation of a natural language interface from the application designer's perspective and highlights the steps taken by the designer in developing natural language interfaces.

Section 3: Creating Natural Language Menu Sentences — Explains how to devise a grammar that produces all and only the set of sentences that are allowed in the interface language.

Section 4: Application Designer Inputs to NLMenu — Explains the format of NLMenu grammar and lexicon files; describes the process whereby a natural language input sentence is translated into the language of the application and provides examples of how the application designer develops translations; also explains developing screen configuration descriptions and implementing experts.

Section 5: Natural Language Menu Grammar-Testing Tools — Describes the tools that the application designer can use to automatically test and debug an NLMenu grammar.

Section 6: Natural Language Menu Lexicographer — Describes the NLMenu utility that assists the application designer in the development or modification of lexicon files and the construction of selection windows.

Section 7: Database Interface — Describes how application designers can use the NLMenu database interface to generate NLMenu interfaces to database applications automatically.

Suggestions for Using This Manual

For an overview of the NLMenu system and natural language technology, all should read the first two sections. Section 3 introduces the fundamentals of grammar writing. Experienced linguists may choose to skim this section, but since NLMenu interfaces are grammar-driven, an understanding of these principles can help the novice linguist create more efficient grammars. Section 4 discusses the various types of input that the application designer must supply to the NLMenu system to create an interface. These are 1) a grammar file, 2) a lexicon file, and 3) a screen description. The sections dealing with the grammar-testing tools (Section 5) and the lexicographer (Section 6) are of general interest. These tools can help the application designer test and debug the grammar and complete the lexicon. Section 7 is of interest to anyone intending to use the NLMenu system with database applications. It details the process of automatically generating interfaces to database applications. The glossary contains linguistic terms that are unique to this member of the Explorer software document set.

Notational Conventions

This manual uses certain notations to indicate the keystroke sequences used to invoke a command, the process of using the mouse, and portions of Lisp code. The following paragraphs describe the notational conventions used in this manual.

Keystroke Sequences

Many of the commands used with the Explorer are executed by a combination or sequence of keystrokes. Keys that should be pressed at the same time, or *chorded*, are listed with a hyphen connecting the name of each key. Keys that should be pressed in a particular sequence are listed with a space separating the name of each key. The following table illustrates the conventions used in this manual to describe keystroke sequences.

Keystroke Sequence	*Interpretation*
META-CTRL-D	Hold the META and CTRL keys while pressing the D key.
SYSTEM HELP	Press and release the SYSTEM key, then press and release the HELP key.
CTRL-X CTRL-F	Hold the CTRL key and press the X key, release the X key, and then press the F key. Alternatively, press CTRL-X, release both keys, and press CTRL-F.
META-X Find File	Hold the META key while pressing the X key, release the keys, type the letters F, i, n, d, space, F, i, l, e, and then press the RETURN key.
TERM - SUPER-HELP	Press the TERM key and release it, press the - key and release it, then press and hold the SUPER key while pressing the HELP key.

Mouse Clicks The optical mouse features three buttons that enable you to execute operations from the mouse without returning your hand to the keyboard. Pressing and releasing a button is called *clicking*. The following table illustrates the abbreviations used online to describe clicking the mouse.

Abbreviation	*Action*
L	Click the left button (press the left button once and release it).
M	Click the middle button (press the middle button once and release it).
R	Click the right button (press the right button once and release it).
L2 M2 R2	Click the specified button twice quickly. (Press the button, release it, then press it again quickly.) Alternatively, you can press and hold the CTRL key while you click the specified button.
LHOLD MHOLD RHOLD	Click the specified button and hold it down. (Press and hold the specified button.)

Lisp Code Three fonts are used in this manual to denote Lisp code:

- System-defined words and symbols are in **boldface**. System-defined words and symbols include names of functions, variables, macros, flavors, methods, packages, and so on—any word or symbol that appears in the system source code.
- Examples of programs and output are in a special `monowidth` font. System-defined words in an example are also in this font.
- Sample names are in *italics*. Names in italics can be any name you choose to substitute. (Italics are also used for emphasis and to introduce new terms.)

For example, the following sentence contains the word **setq** in boldface because **setq** is defined by the system:

> The purpose of the **setq** special form is to assign a value to a variable.

Some function and method names are very long—for example, **get-ucode-version-from-comment**. When the function is first discussed, the function name is given followed by a list of arguments. Within the text, long function names may be split over two lines due to typographical constraints, as in the following example:

> For this reason, you can use either **get-ucode-version-from-comment** or **get-ucode-version-of-band** to obtain the version of the loaded microcode.

When you type the function name **get-ucode-version-from-comment**, however, you should not split it or include any spaces within it.

Within each example of actual Lisp code, all names are shown in the monowidth font. For instance:

```
(setq x 1 y 2) => 2
(+ x y) => 3
```

> The form `(setq x 1 y 2)` sets the variables `x` and `y` to integer values; then, the next form `(+ x y)` adds them together.

In this example of Lisp code with its explanation, `setq` appears in the monowidth font because it is part of a specific example.

Words for which you can substitute another value are shown in italics, as in the following example:

> The variables *vars* contained in the lambda list of some function *foo* are bound to the argument values of the function invocation.

Occasionally, in examples where you could substitute a specific value, the boldface and italics fonts are used together.

CONTENTS

	Figure	Title	Page

Figures

OVERVIEW 1

Highlights of This Section

- Natural Language Menu Toolkit
- An interface user's view of NLMenu interfaces
  - Interface screen and windows
  - Building a sentence
    - Moving items into view
    - Selecting an item
    - Entering specific information
    - Changing a sentence
  - Sending a sentence to the application program
- Description of the NLMenu system
  - Creating an NLMenu interface
  - NLMenu loop
  - Grammar formalism
  - Parser
  - Grammar compiler
  - NLMenu tools
    - Grammar tests
    - Lexicographer/screen builder

Natural Language Menu Toolkit

1.1 The Texas Instruments Natural Language Menu (NLMenu) system enables you to design a natural language interface for a computer user with little or no programming background. You can use the NLMenu tools to produce a series of windows and menus from which the interface user can select an option, consisting of a natural language word or phrase, to perform some task or to construct input sentences. You can tailor options to the requirements of specific interface users.

Using the NLMenu software, you also control the nature and relationships of words or phrases selected by the interface user during sentence construction. The menu system guides the interface user so that he forms only valid sentences. The scope and limitations of the system become immediately clear to the interface user. This ease of use alleviates the interface user's anxiety about dealing with a new technology, and also saves him time by eliminating error checks and the need to reformulate sentences. The software translates interface-user input into the target language of the application and passes the input to the computer system or application program.

This simplifies the interface and increases both interface-user and system productivity.

An Interface User's View of NLMenu Interfaces

1.2 The following paragraphs describe the interface user's view of a natural language interface that you, the application designer, can create using the NLMenu tools.

Interface Screen and Windows

1.2.1 The interface screen is made up of several windows that contain lists of items, or menus, that the interface user can select. These windows can contain the following types of items:

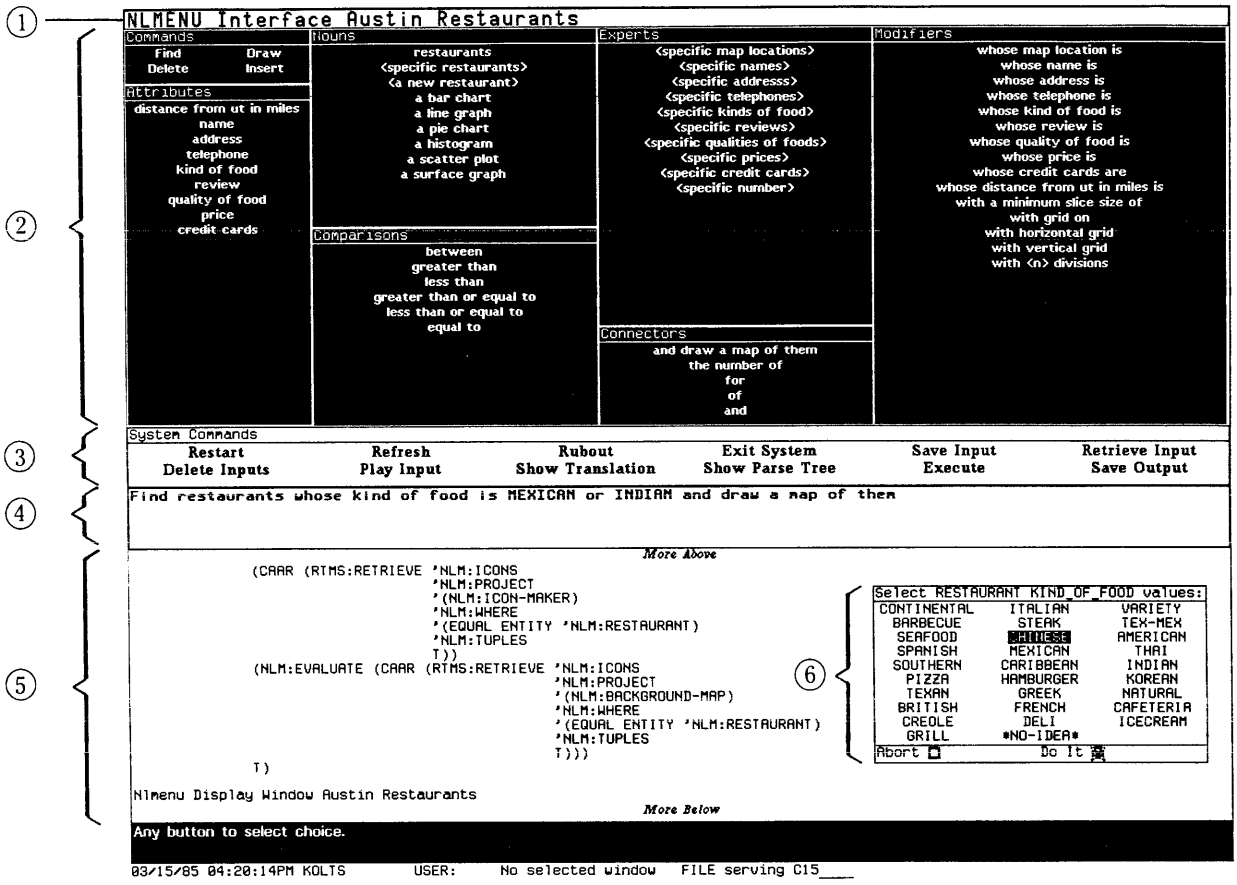
- The name of the natural language interface
- The items the interface user can select to build sentences
- The options that can be applied to sentences the interface user builds
- A display of the sentence as it is being built
- A display of any results or messages once the sentence is completed and sent to the application program for processing

When the interface user makes a selection from one of the menus, the windows containing the next valid choices are automatically highlighted. The interface user can make selections only from highlighted, or active, menus. Highlighted items are displayed in black on a white background. Highlighting and the use of menus provide visual cues that guide the interface user during each step of the sentence-building process. In addition, only the items that are appropriate choices in the existing context are displayed in the highlighted menu. Items that might otherwise appear in the menu but that are inappropriate in the present context are not displayed. As a result, the interface user cannot build invalid sentences. Grammar mistakes, punctuation omissions, lexical errors, unusual paraphrases, and simple typing errors are not possible. Moreover, inputs not within the scope of the application are not possible, so the conceptual limits of the system are immediately apparent.

The interface used in the following series of illustrations is the Austin Restaurants Interface. It is an interface to an RTMS (Relational Table Management System) database containing information about various restaurants in the Austin, Texas area. The interface was constructed automatically with the database interface generator.

Figure 1-1 is an example of an interface screen during a typical sentence-building process. The numbers in the sample screen correspond to the explanatory notes below.

Figure 1-1 Austin Restaurants Interface Screen



① This window contains the name you, the application designer, give to the natural language interface.

② Each of the selection windows contains a list of items (a menu) that the interface user can choose when building sentences. The interface highlights menus as the words or phrases in that menu become eligible for selection. Only the highlighted items can be chosen. This ensures that each completed sentence has the correct format. If there are more items in the menu than can fit in the selection window, the interface user can move through the item list. The process of moving, or scrolling, items into view is described below.

③ This window (System Commands window) displays the operations that the interface user can choose to apply to a sentence at any stage in the building process. The Execute command appears only when the interface user's sentence is complete and indicates that the sentence can be sent to the application program for processing.

④ This window (Input window) displays the interface user's sentence as it is built.

⑤ An application program can produce certain results or send certain messages when it processes the interface user's sentence. This window displays these results or messages. The phrases Beginning-of-Output and End-of-Output appear at the appropriate positions in the display. If there is more output than can be contained within the size limits of the window, the interface user can move additional portions of the output into view. The phrases More Above and More Below appear to indicate the current position within the output display. The process of moving, or scrolling, items into view is described below.

⑥ Additional windows appear, as needed, to provide help information, or to request filenames, dates, and so forth. These are called *pop-up windows*, and they are superimposed over the interface screen.

Building a Sentence

1.2.2 To build a sentence, the interface user places the mouse cursor on the desired word or phrase in a highlighted menu and clicks left once with the mouse. The chosen item is added to the display of the developing sentence in the Input window, and highlighting indicates the next valid menu item(s). The interface user repeats this selection process until the sentence is complete. The following paragraphs describe in greater detail the individual aspects of building a sentence.

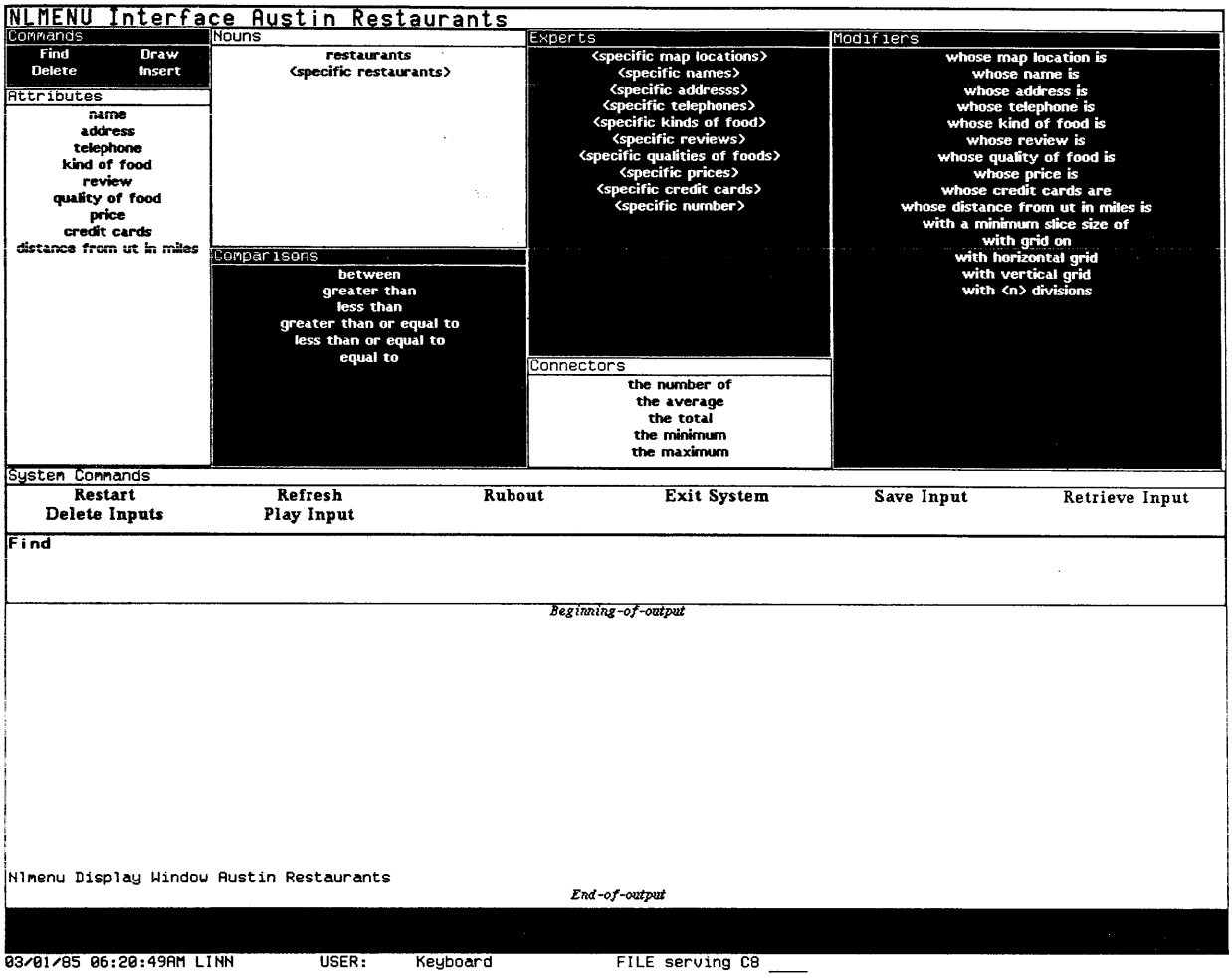
Moving Items Into View

1.2.2.1 When a window contains more menu items or output than can be displayed within the window, the interface user can move the remaining items into view by bumping the mouse cursor against the appropriate edge of the window (depending on the present position of the window in the list, there can be more items above, below, or both above and below the present window position).

Selecting an Item

1.2.2.2 To select a menu item, the interface user moves the mouse cursor to the desired item and clicks left once on the mouse. Figure 1-2 presents a sample screen as the interface user is about to make a selection.

Figure 1-2 Before Selecting the Restaurants Item

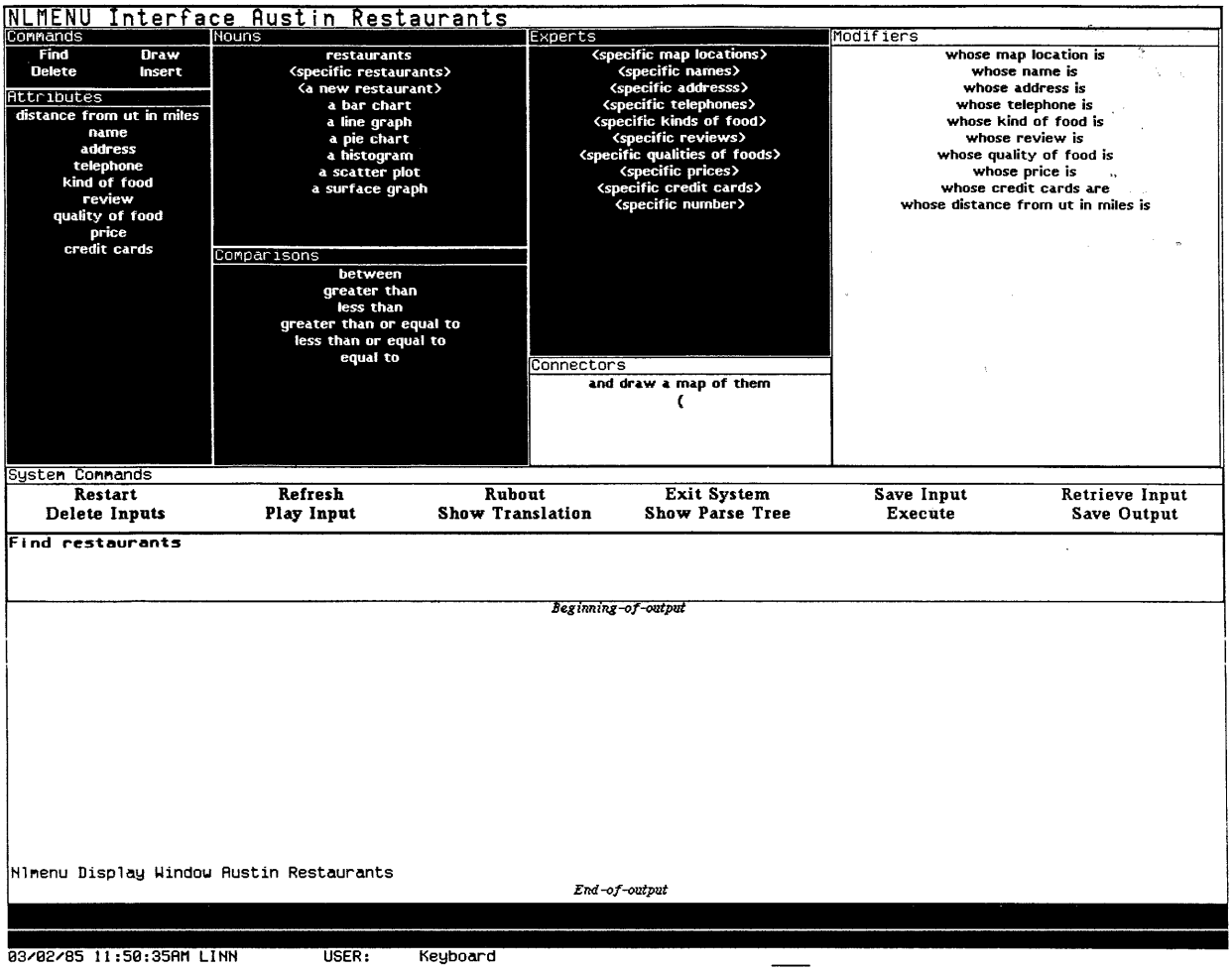


Each time the interface user makes a selection:

- The word or phrase chosen is added to the display of the developing sentence.
- Highlighting indicates the new menu(s) with valid choices.

In the following sample screen (Figure 1-3), the interface user has now made a selection. Note how the window displaying the interface user's sentence and the selection windows have changed.

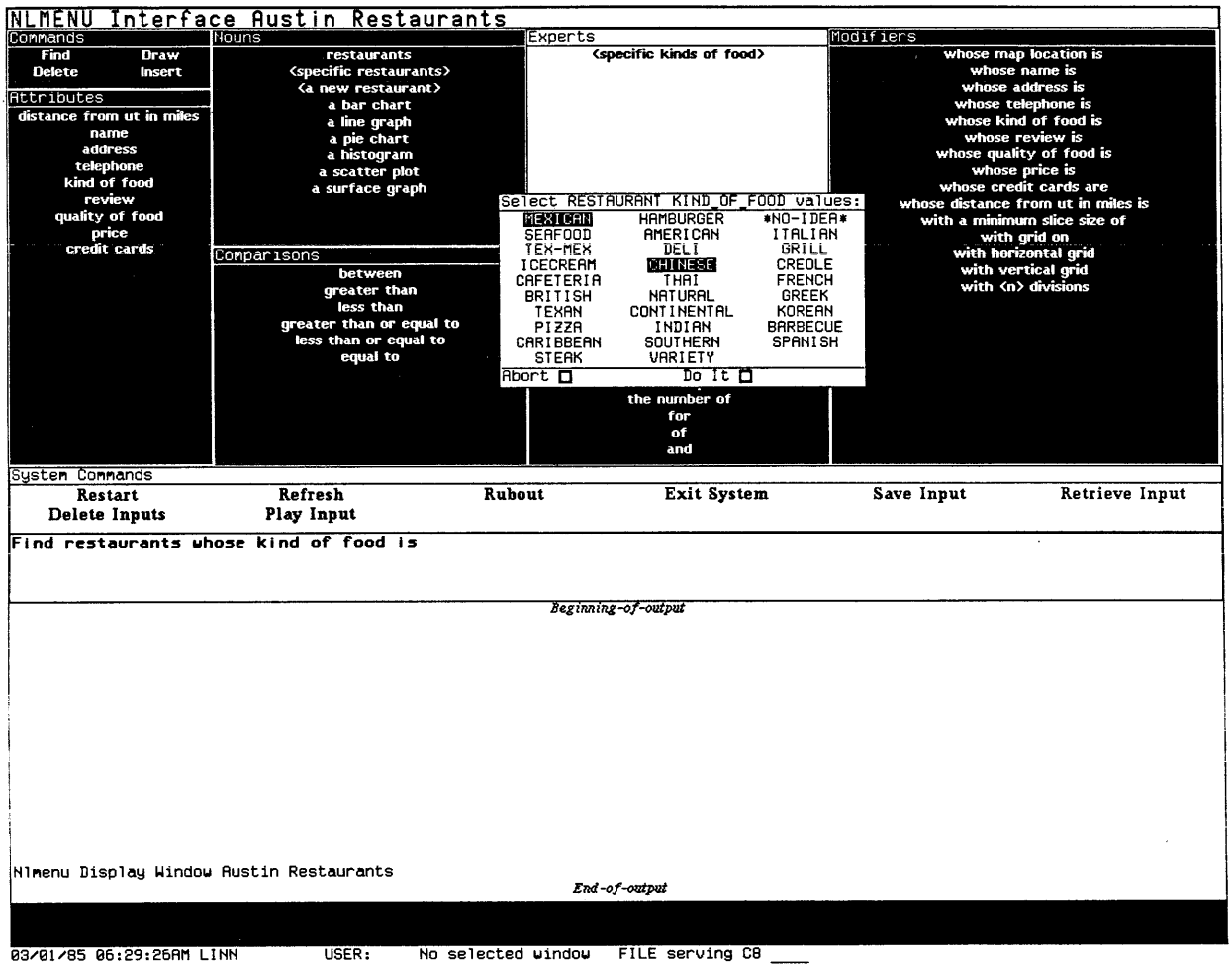
Figure 1-3 After Selecting the Restaurants Item



Entering Specific Information

1.2.2.3 Some sentences that the interface user wants to build can require specific information such as a filename, a date, and so on. The interface requests this information from the interface user by superimposing a pop-up window (see Figure 1-4) over the interface screen. There are several types of pop-up windows, and the different types require different responses from the interface user. Some pop-up windows require the interface user to type the information requested and then press the RETURN key. Other pop-up windows are exclusively for entering numerical data. These windows function like handheld calculators. The interface user selects numerical values from the window with the mouse. The pop-up window shown in Figure 1-4 limits the selection options to a set of default values. To select items in this type of pop-up window, the interface user places the mouse cursor over a desired item and clicks left once on the mouse.

Figure 1-4 Selecting Food Types Via a Pop-Up Window



Changing a Sentence **1.2.2.4** If the interface user wants to change a sentence after making a selection or after building a complete sentence (but before submitting the sentence for processing by the application program), it is possible to reverse the selection process either partially or totally:

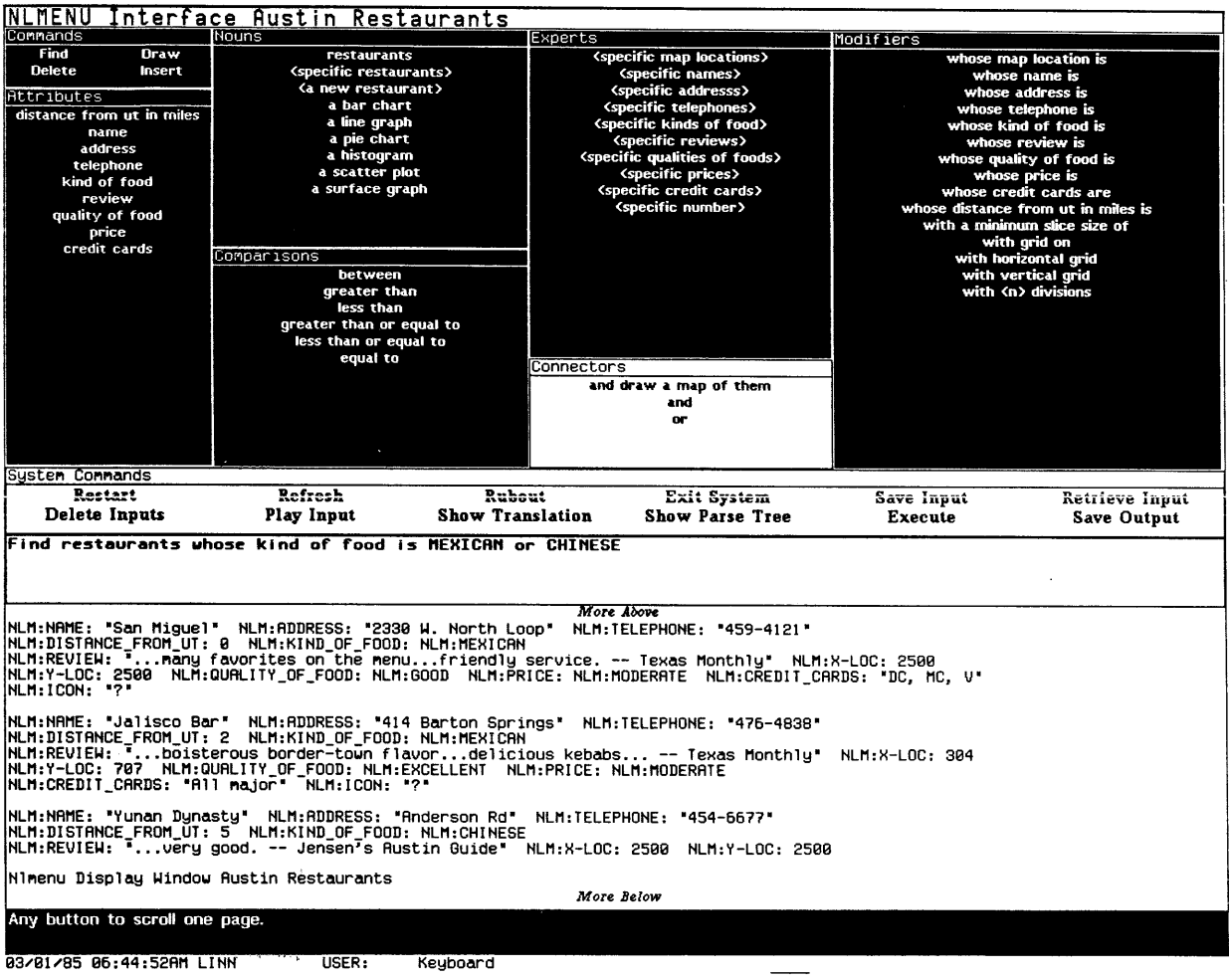
- To back up one selection at a time, either move the mouse cursor to the Rubout option in the System Commands window and click right once on the mouse or press the equivalent keystroke (see the mouse-documentation line).

- To erase the entire sentence and start the selection process over from the beginning, either move the mouse cursor to the Restart option in the System Commands window and click right once on the mouse or press the equivalent keystroke (see the mouse-documentation line).

Sending a Sentence to the Application Program

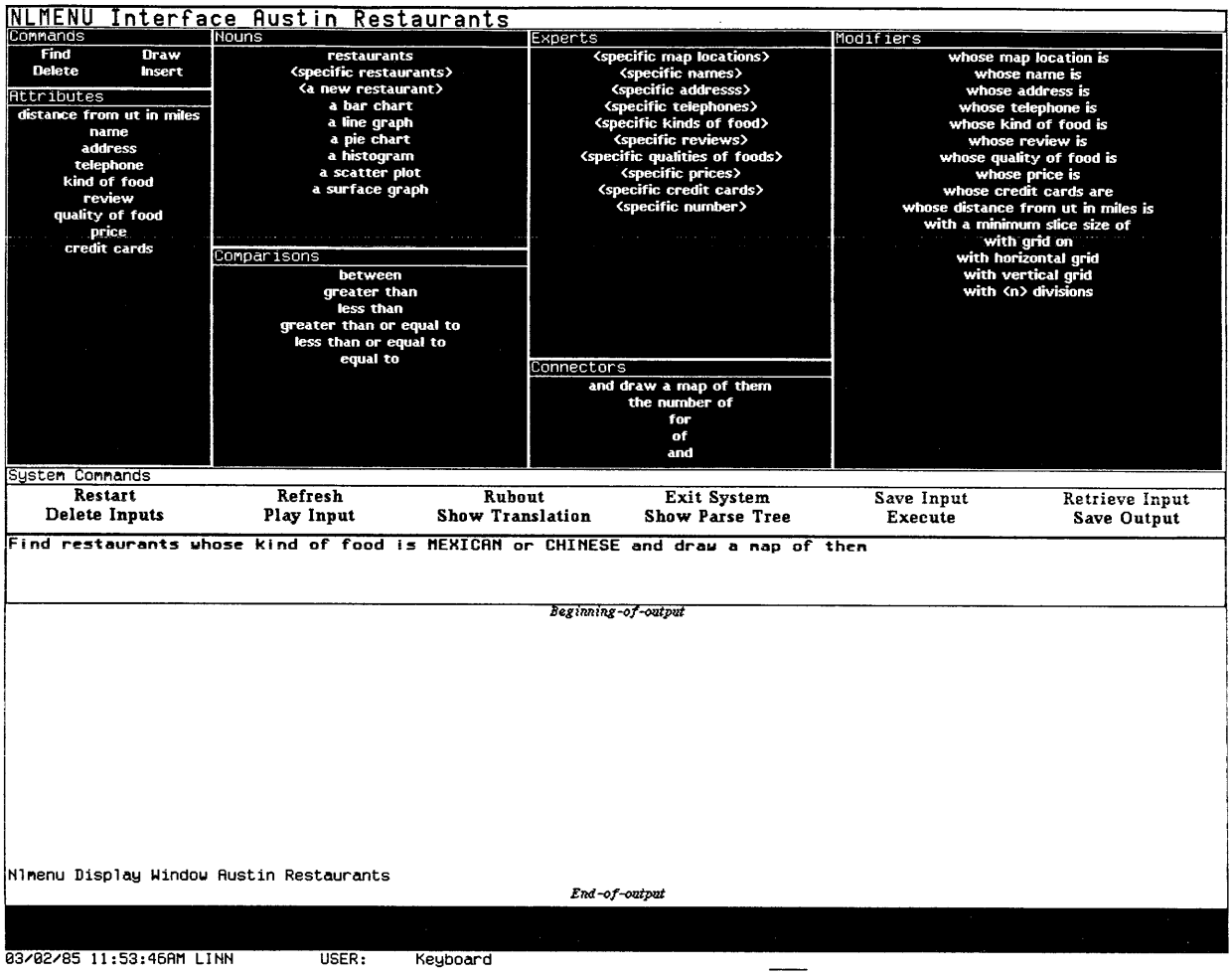
1.2.3 When the input sentence is complete, the interface visually cues the interface user. The interface user then sends the sentence to the application program by moving the mouse cursor to the Execute option in the System Commands window and clicking left once with the mouse. In some instances, the interface user can either submit the sentence as it stands or continue adding items to the sentence until it is again complete. If the interface user's sentence is complete and no possibility for further expansion exists, none of the selection menus are highlighted, and the Execute command appears in the System Commands window. If the sentence is complete but can still be expanded, the Execute command appears, and selection menus representing valid next choices are highlighted. Figures 1-5 and 1-6 contrast these two situations:

Figure 1-5 A Complete, But Expandable Information Request



Note in Figure 1-5 that the interface user has executed the query and obtained results in the Output window. This does not prevent the user from continuing to expand the query.

Figure 1-6 A Complete, Nonexpandable Information Request



As the user makes selections from the menu, the interface translates the input into the language of the application program. When the user selects the Execute command, this translation is sent to the application, and any results, instructions, or messages produced when the application processes the sentence are displayed on the video monitor. Figures 1-7, 1-8, and 1-9 illustrate various results that the interface user can obtain by submitting the query shown in Figure 1-6.

Figure 1-7 The Application Draws the Requested Map

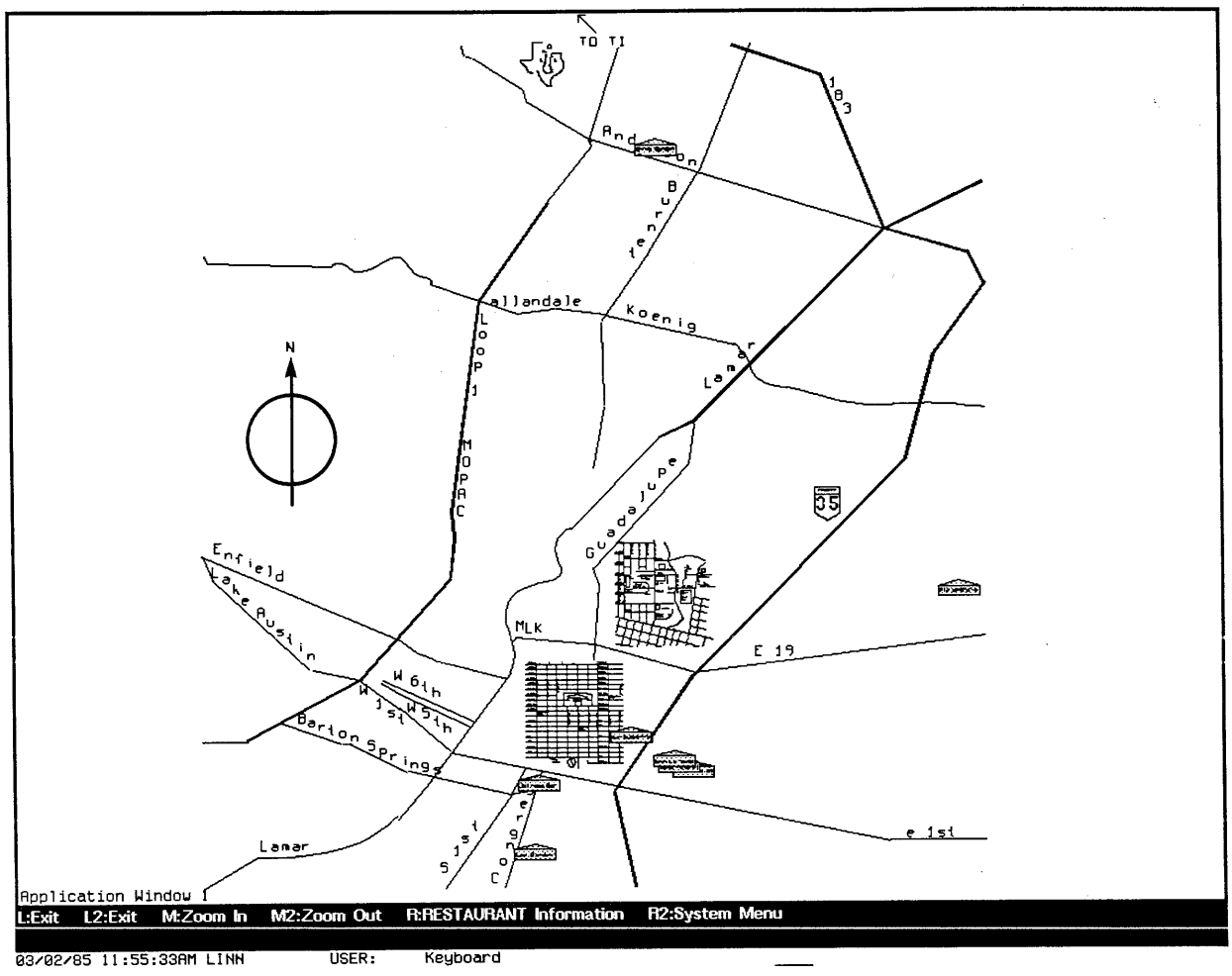


Figure 1-8 The User Chooses to Zoom In

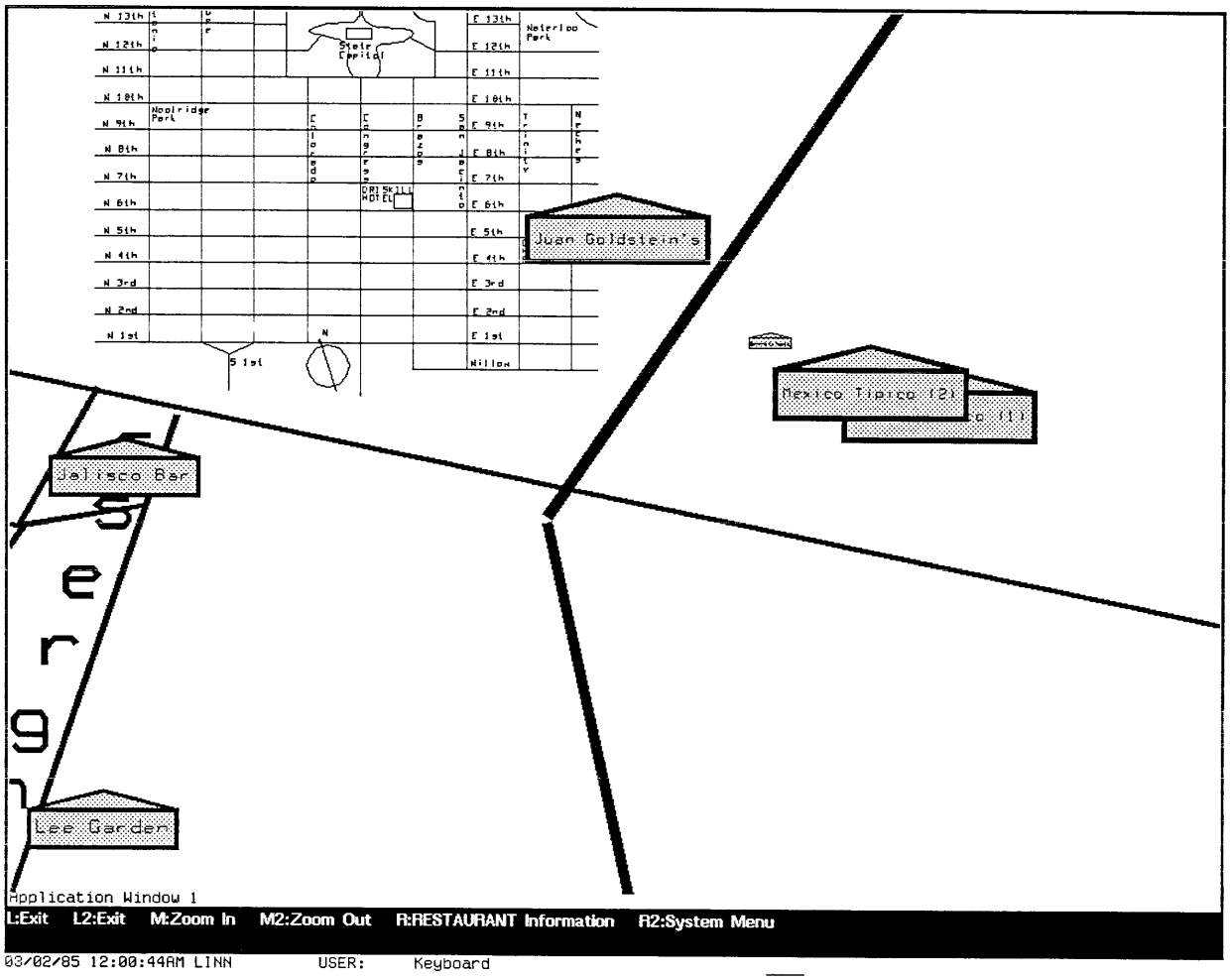
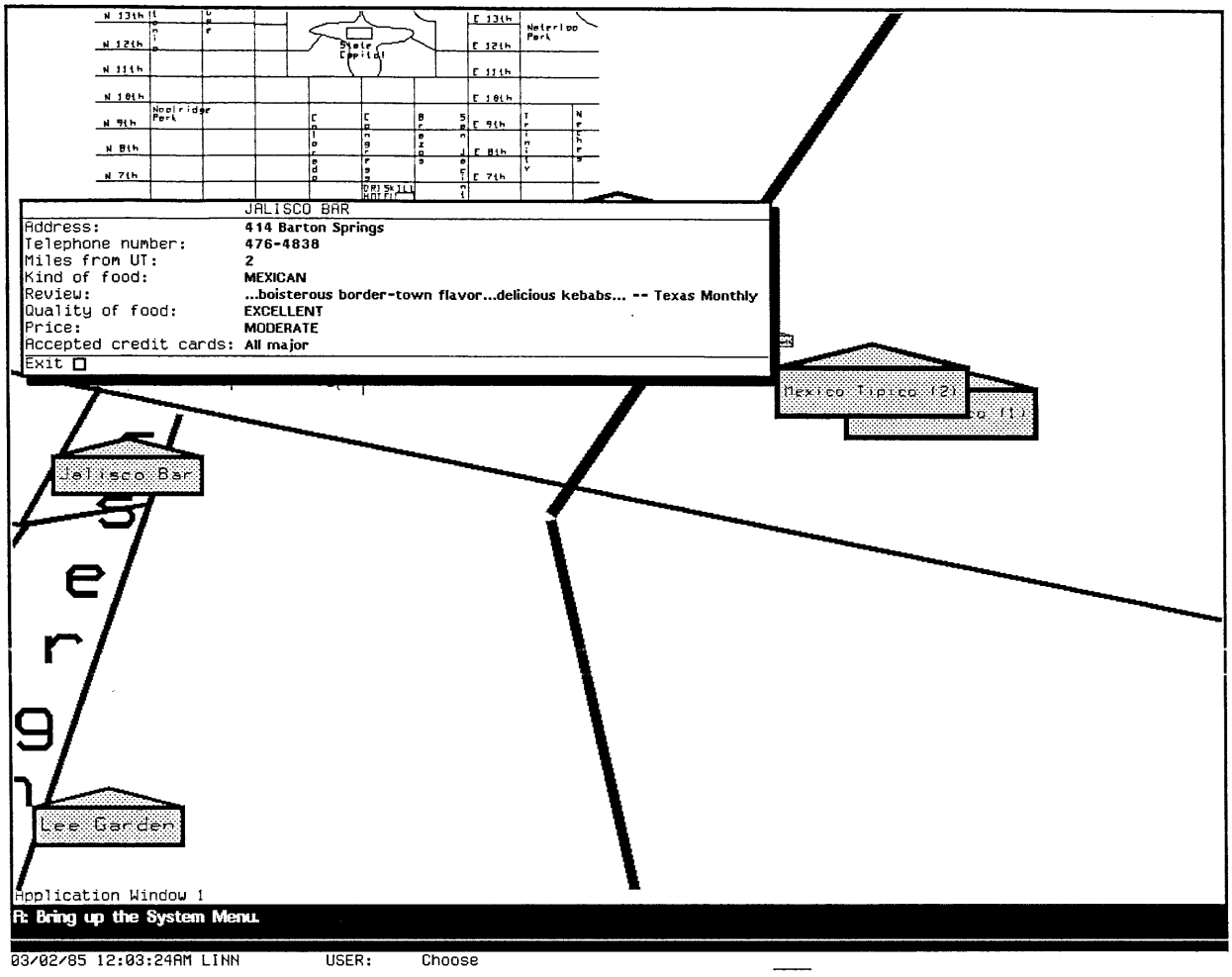


Figure 1-9 Getting Specific Restaurant Information



Description of the NLMenu System

1.3 The following paragraphs contain brief overviews of the key NLMenu components, features, operations, and required input from you, the application designer.

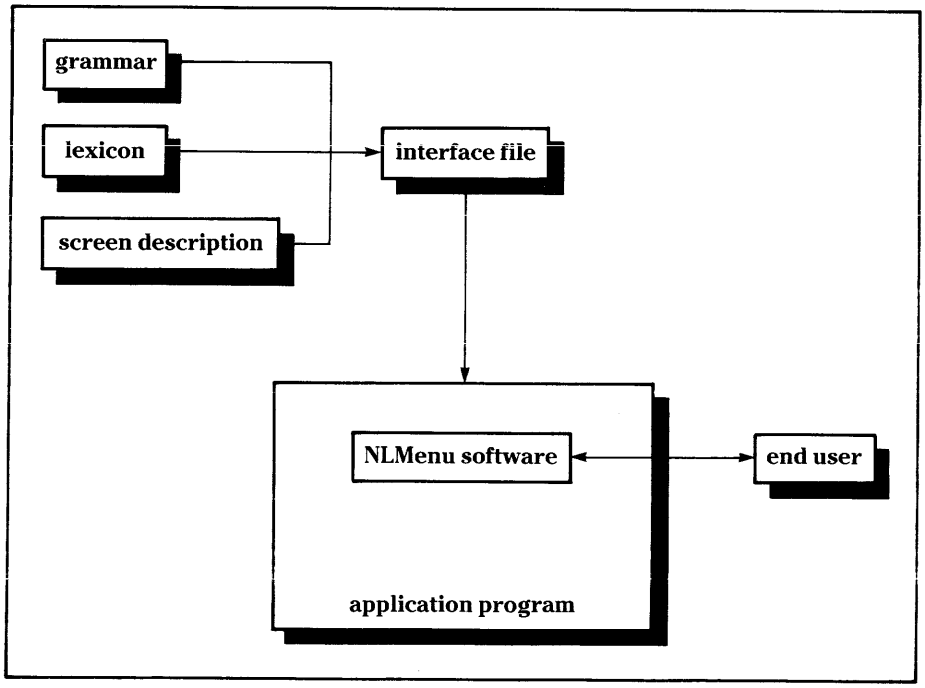
Creating an NLMenu Interface

1.3.1 To determine which windows are active and what data items are displayed in each active window, you must provide the following three types of input to the NLMenu system:

- A *grammar* that defines the valid combinations of words and phrases in a selected natural language, the order of such words and phrases, and the appropriate mapping of these words and phrases into the target language of the application
- A *lexicon* that provides translations of the words or phrases used in the mapping
- A *screen description* that tells the system where to place windows and what words or phrases to insert in them

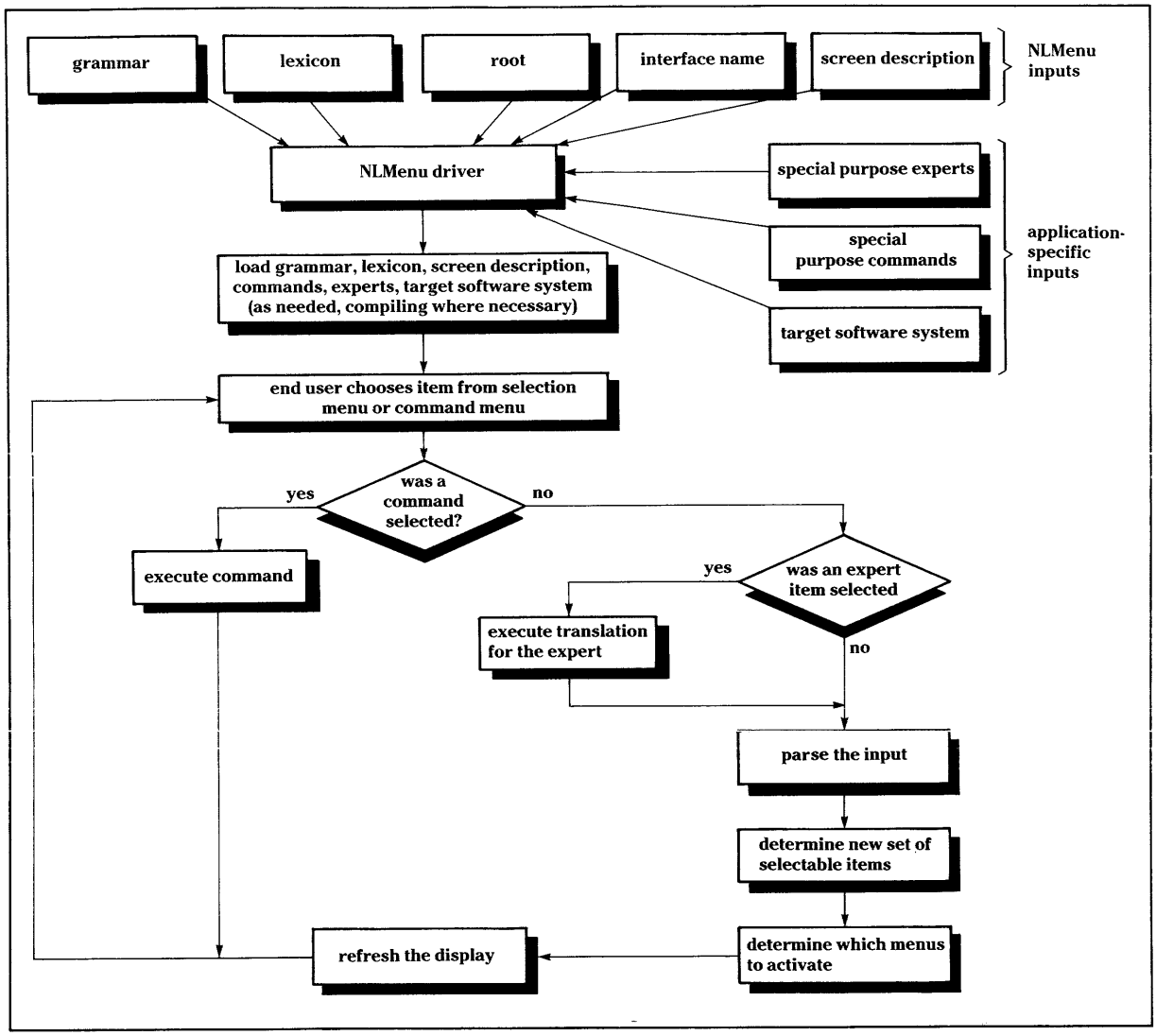
These three elements enable you to create and test NLMenu interface files. Other NLMenu utilities use these files for interaction with your particular application. Figure 1-10 shows how the NLMenu software —linked with the application—accepts natural language input from an interface user, translates that input into a target string, and sends it to the application.

Figure 1-10 **NLMenu Interface**



NLMenu Loop **1.3.2** The NLMenu loop is the main input loop of the NLMenu system, controlling the interaction of other system components. The overall flow of control is shown in Figure 1-11.

Figure 1-11 NLMenu Driver Input Loop



The input to the loop forms a complete specification of an NLMenu interface and consists of the following:

- Semantic grammar
- Lexicon
- Set of experts (handlers for specific information)
- Screen-configuration description
- Set of commands
- Target software system

Each of these types of input is described in detail in a subsequent section. The basic cycle of the loop is as follows:

1. An array of menus is displayed to the interface user.
2. The interface user selects a menu item (a word or phrase of a sentence).
3. The choice is parsed.
4. Using a reachability matrix and the new parser state, the set of next legal grammar terminals is predicted.
5. The screen is refreshed to display the next legal choices, ready for the interface user's next input.

Grammar Formalism

1.3.3 The grammars used in the NLMenu system define the allowable sentences in the interface language. Sentences and sentence fragments are defined by rules that indicate the components of the sentence or fragment and the proper arrangement(s) of the component parts. These rules can contain the standard abbreviatory conventions used by linguists for collapsing (combining) grammar rules. In addition to word order information, the grammar formalism also contains information that determines the meaning of each input sentence. A lexicon containing an entry for each terminal in the grammar is part of the formalism, and there is a translation associated with every entry in the lexicon. Associated with each grammar rule is a rule that specifies how to combine translations associated with its constituents.

Parser

1.3.4 The parser keeps track of selections and determines the next valid choices. The parser used in the NLMenu system is enhanced to enable parsing of one word at a time and to predict the set of next possible words in a sentence, given the input that has come before. This increases the perceived speed of the parser because the parser works as the interface user is entering input. The NLMenu parser also features a rubout facility, which eliminates previous choices in reverse order of selection.

Grammar Compiler

1.3.5 The grammar compiler converts the external form of a grammar into an internal form that is convenient for the parser. The component of a grammar rule that determines valid word order is converted to list representation, and the rule is compiled into an internal format. The grammar compiler also creates rules that indicate how to combine the meaning contributed by each rule component to produce the meaning of the sentence or sentence fragment that the rule defines. Compiled grammars are automatically saved and reused provided you have made no changes to the source grammar and lexicon since compilation took place.

NLMenu Tools

1.3.6 The NLMenu tools are a complete set of interface development tools. These tools help you, the application designer, to produce and package a high-level interface that accepts simple, natural language words and phrases as input. In creating a natural language interface, you must supply information regarding the structure and content of sentences, and descriptions of how a screen will look. The NLMenu tools help you write a grammar and build the lexicon and screen descriptions that govern interface-user input.

Grammar Tests

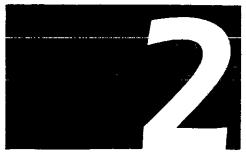
1.3.6.1 The NLMenu grammar tests provide you with an automated means of debugging and testing a handwritten grammar. Section 5 describes in detail the various grammar tests available to you. The tests are automatically loaded when the NLMenu development system is loaded.

Lexicographer/ Screen-Builder

1.3.6.2 The lexicographer offers several options that aid in the development of the lexicon and the screen description:

- The Specify New Lexical Items option prompts you for each lexical entry required by that grammar and then automatically saves the lexicon.
- The Modify Existing Lexical Items option lets you change various entries in the lexicon, and the updated lexicon is automatically saved.
- The View Selection Window option provides you with a means to examine the size and contents of any selection menu based on the current state of the lexicon.
- The Build Screen option automatically constructs a screen description based on the current state of the lexicon. This description usually is optimal, so you need to be concerned only with grammar and lexicon development. In situations where you want to provide your own screen description, the description produced by the screen-builder is extremely useful in suggesting a final layout; you can obtain specific window geometries directly from the View Selection Window option.

USING NATURAL LANGUAGE MENU



Highlights of This Section

- Functional overview of the NLMenu system
  - Invoking an NLMenu interface
    - Selecting an NLMenu System menu
    - NLMenu System menu without database interface
    - NLMenu System menu with database interface
  - Invoking NLMenu from an application
  - Screen appearance and sentence building
    - Interface screen and windows
    - Moving around the interface screen
    - NLMenu help facilities
  - Description of commands
    - Restart command
    - Refresh command
    - Rubout command
    - Exit System command
    - Save Input command
    - Retrieve Input command
    - Delete Inputs command
    - Play Input command
    - Show Translation command
    - Show Parse Tree command
    - Execute command
    - Save Output command

- Exiting an NLMenu interface

- Interface development

- Starter kits
  - Core NLMenu starter kit
  - NLMenu-RTMS-interface starter kit

Introduction

2.1 This section outlines the Natural Language Menu (NLMenu) interface development procedure and provides starter-kit descriptions, instructions for invoking an interface, and a functional overview of the system. Details concerning the development of particular interface components are contained in subsequent sections.

Functional Overview of the NLMenu System

2.2 The following paragraphs describe:

- Instructions for invoking an NLMenu interface—either the starter kit interface (see paragraph 2.4 for information about the starter kits) or an interface with user-supplied parameters
- Components of the NLMenu screen that appear after invoking the interface
- Operations that you can carry out on natural language sentences built using the interface

Invoking an NLMenu Interface

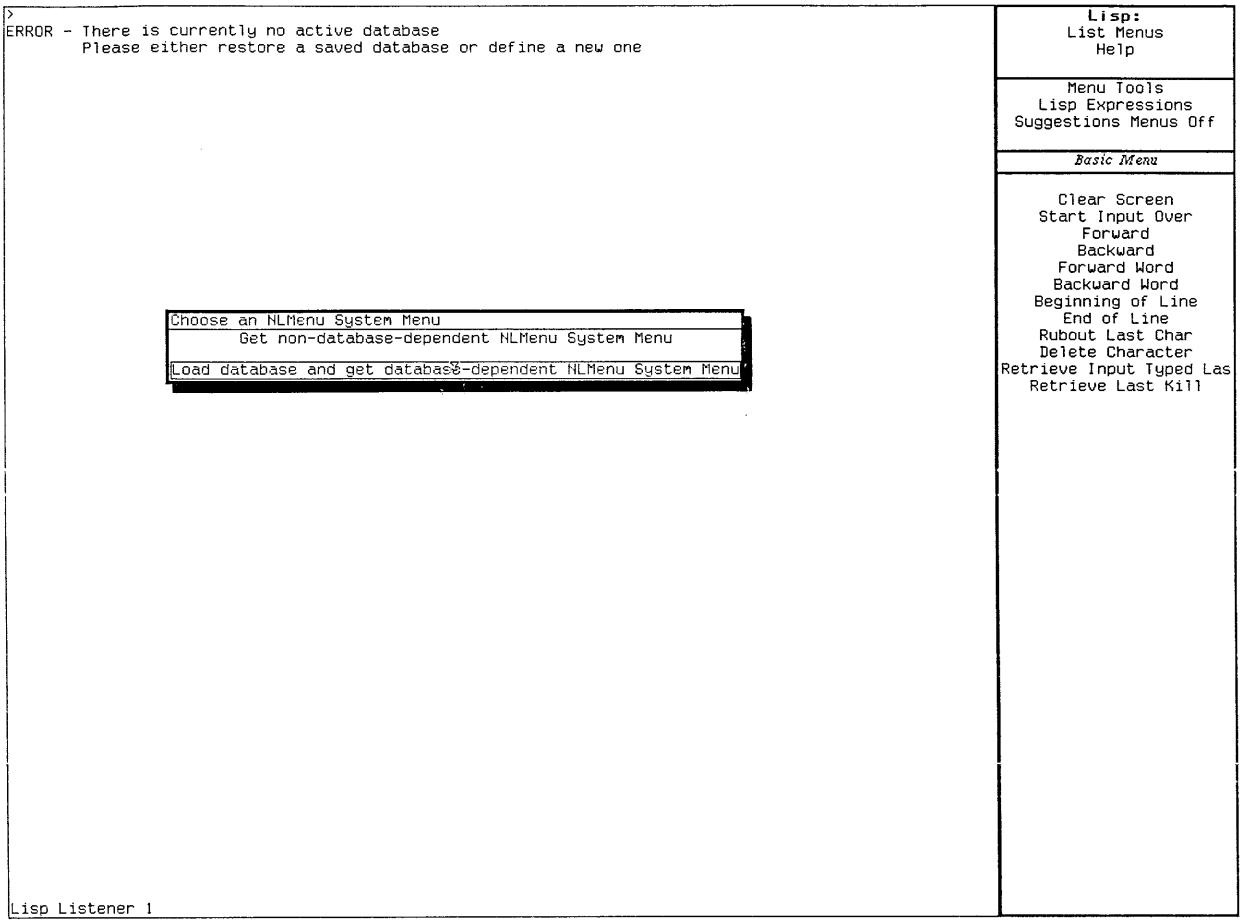
2.2.1 To invoke an NLMenu interface, you either select the NLMenu option from the System menu or press SYSTEM N. Invoking NLMenu typically brings up another menu that you use to select an NLMenu System menu. The NLMenu System menu lists available interfaces and various system commands. The precise form of the NLMenu System menu depends on whether the Relational Table Management System (RTMS) toolkit is available.

NOTE: If the RTMS toolkit is available and the NLMenu System menu is being managed by the interface management tools described in Section 7, an active database must reside in memory before the NLMenu System menu can be created.

Selecting an NLMenu System Menu

2.2.1.1 After you invoke the NLMenu system by either selecting the NLMenu option from the System menu or pressing SYSTEM N, the menu shown in Figure 2-1 appears if the RTMS toolkit is available. If the RTMS toolkit is not installed, the nondatabase-dependent NLMenu System menu described in paragraph 2.2.1.2 appears immediately.

Figure 2-1 Menu for Choosing an NLMenu System Menu

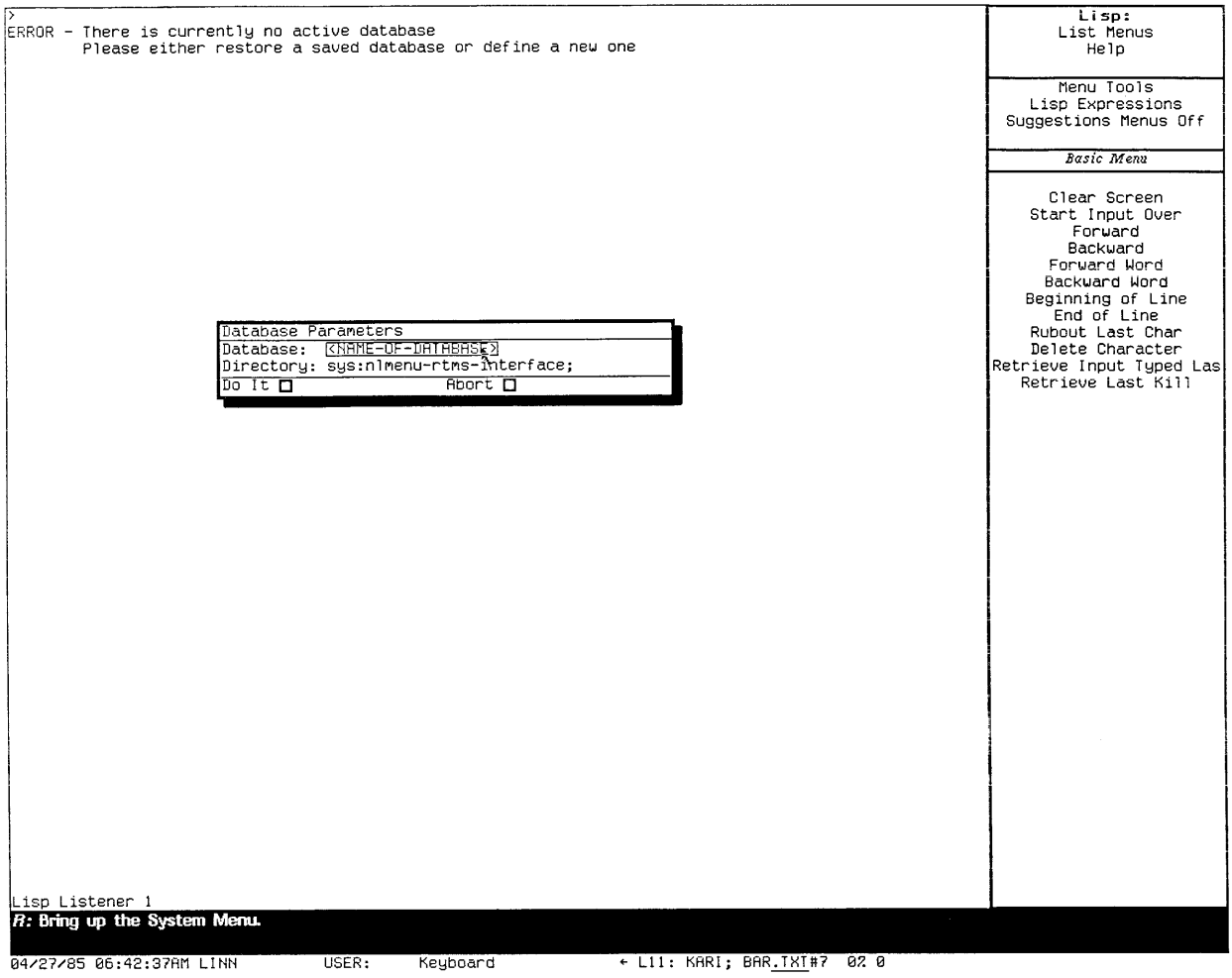


The options available in this menu allow you to select the NLMenu System menu appropriate for your needs.

If you select the first option, NLMenu displays the nondatabase-dependent NLMenu System menu that is described in paragraph 2.2.1.2. You are not able to access the automatic interface generator described in Section 7 if you select this option.

If you select the second option, you are asked to load a database from a menu. That menu is shown in Figure 2-2.

Figure 2-2 Database Parameters Choose-Variable-Values Menu



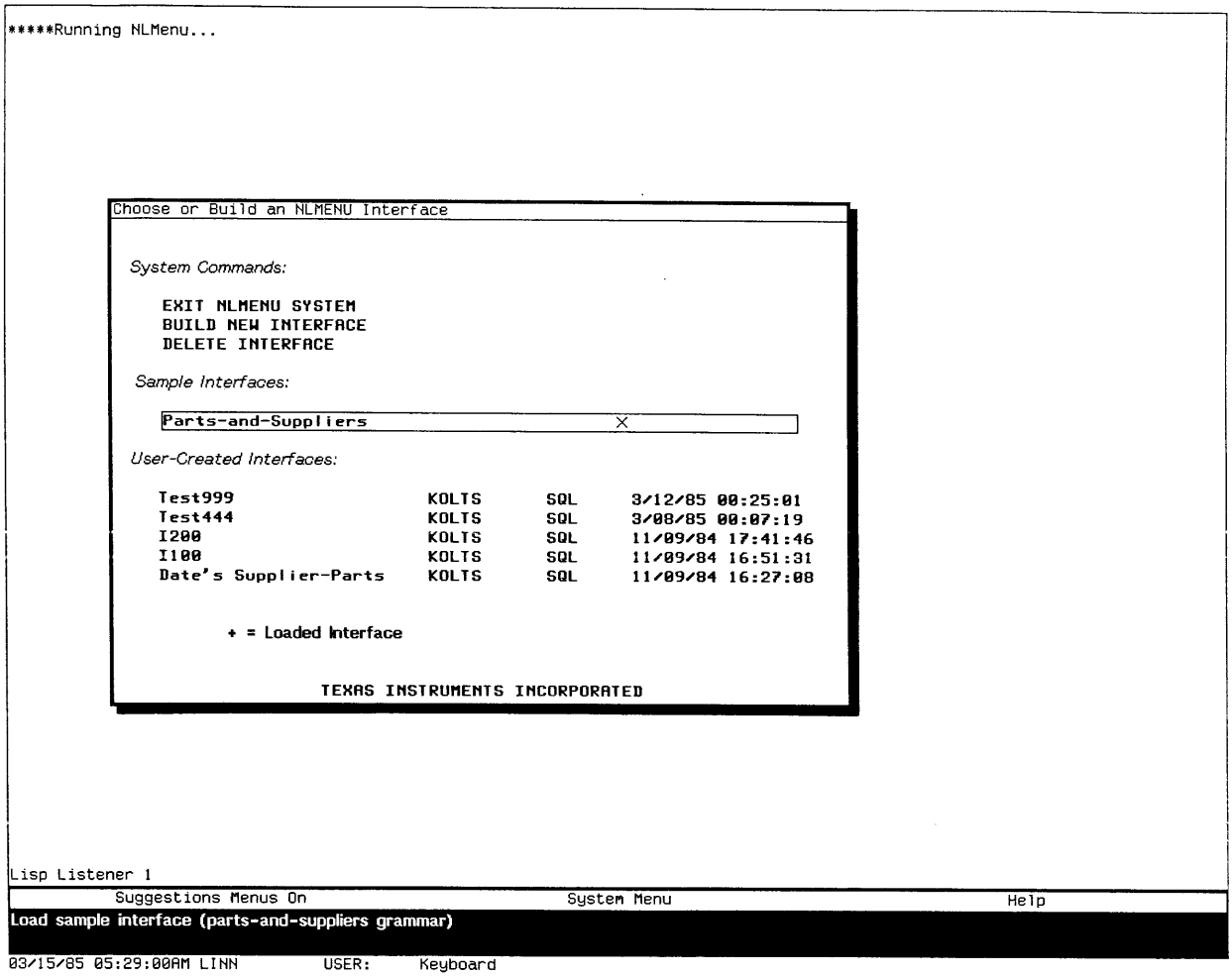
Your response to the database prompt is the name of the database to be loaded. This database should contain the NLMenu-interfaces and the interface-grants relations. In response to the directory prompt, enter the name of the directory where the database resides.

When you have entered the required information in the Database Parameters choose-variable-values menu, select either the Do It or the Abort option. If you select the Abort option, you are returned to the Lisp Listener and the Explorer system displays a message that you aborted entry to NLMenu. If you select the Do It option, the database is loaded and an NLMenu System menu similar to that described in paragraph 2.2.1.3 appears. Note that access to the automatic interface generator described in Section 7 is available from this NLMenu System menu.

NLMenu System Menu Without Database Interface

2.2.1.2 If RTMS is not installed or if you select the nondatabase-dependent NLMenu System menu, you are presented with an NLMenu System menu that is similar to the one shown in Figure 2-3.

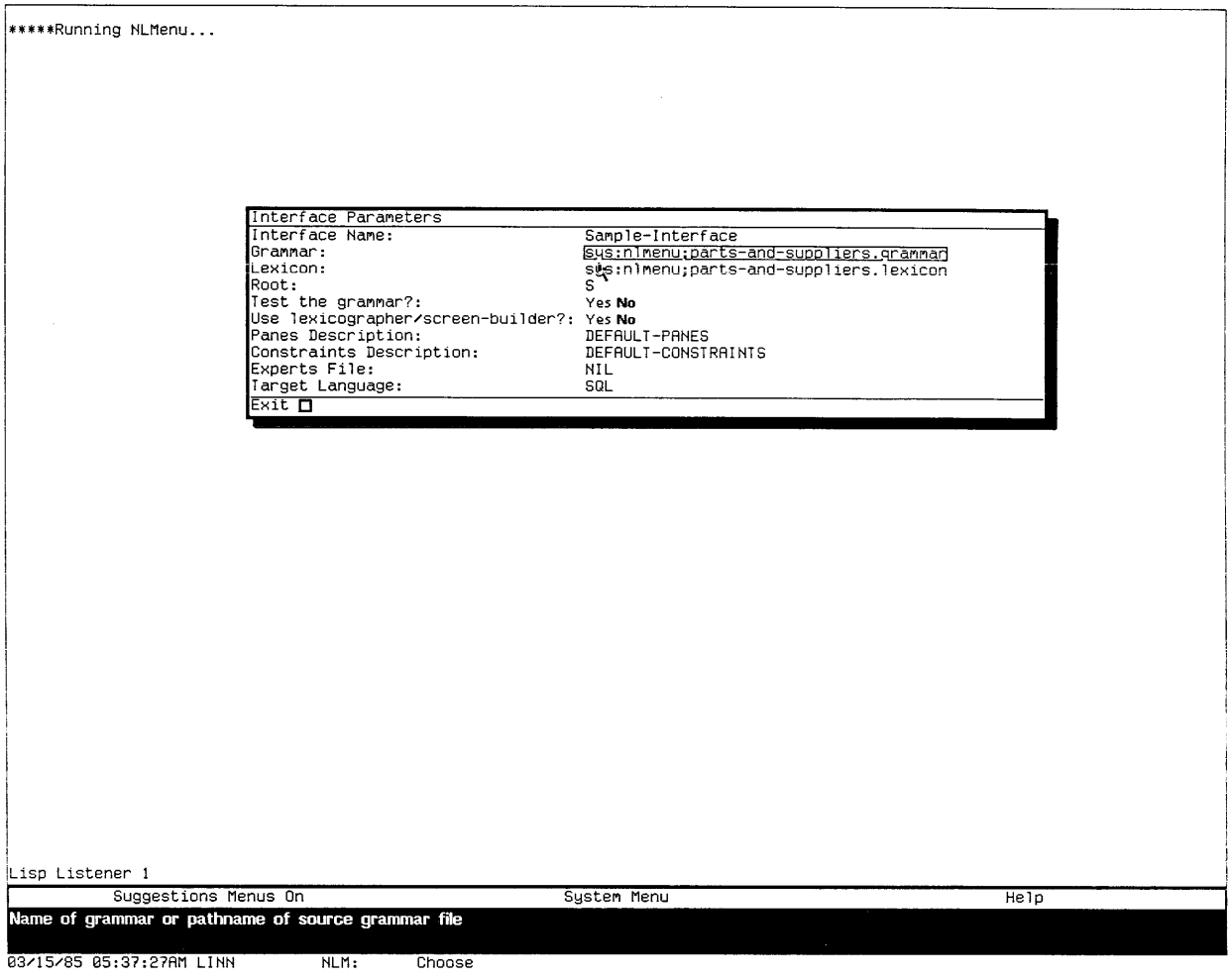
Figure 2-3 Typical NLMenu System Menu — Without RTMS



To build an interface with starter-kit values for all components, select the Parts-and-Suppliers sample interface. This loads the compiled parts-and-suppliers grammar and a corresponding NLMenu constraint window is created and displayed.

Initially, there are no user-created interfaces available. The Build New Interface option adds such interfaces. Selecting this option displays a choose-variable-values menu from which you can select various interface inputs. This window is shown in Figure 2-4. Subsequent invocations of NLMenu include the new interface on the NLMenu System menu, preceded by a plus sign (+) if the interface is loaded. There is no limit to the number of interfaces that can be loaded.

Figure 2-4 Interface Parameters Choose-Variable-Values Menu



The input parameters shown in the Interface Parameters choose-variable-values menu are as follows:

- Interface Name — This name appears in the logo window and under the user-created interfaces column on the NLMenu System menu. Type the name of an interface, and press the RETURN key. There are no restrictions on the length of the interface name; although, for purposes of saving and/or retrieving user-created sentences or queries for a particular interface, only the first four characters of the interface name are significant. The NLMenu System menu is arranged to handle interface names of up to 25 characters without truncation.

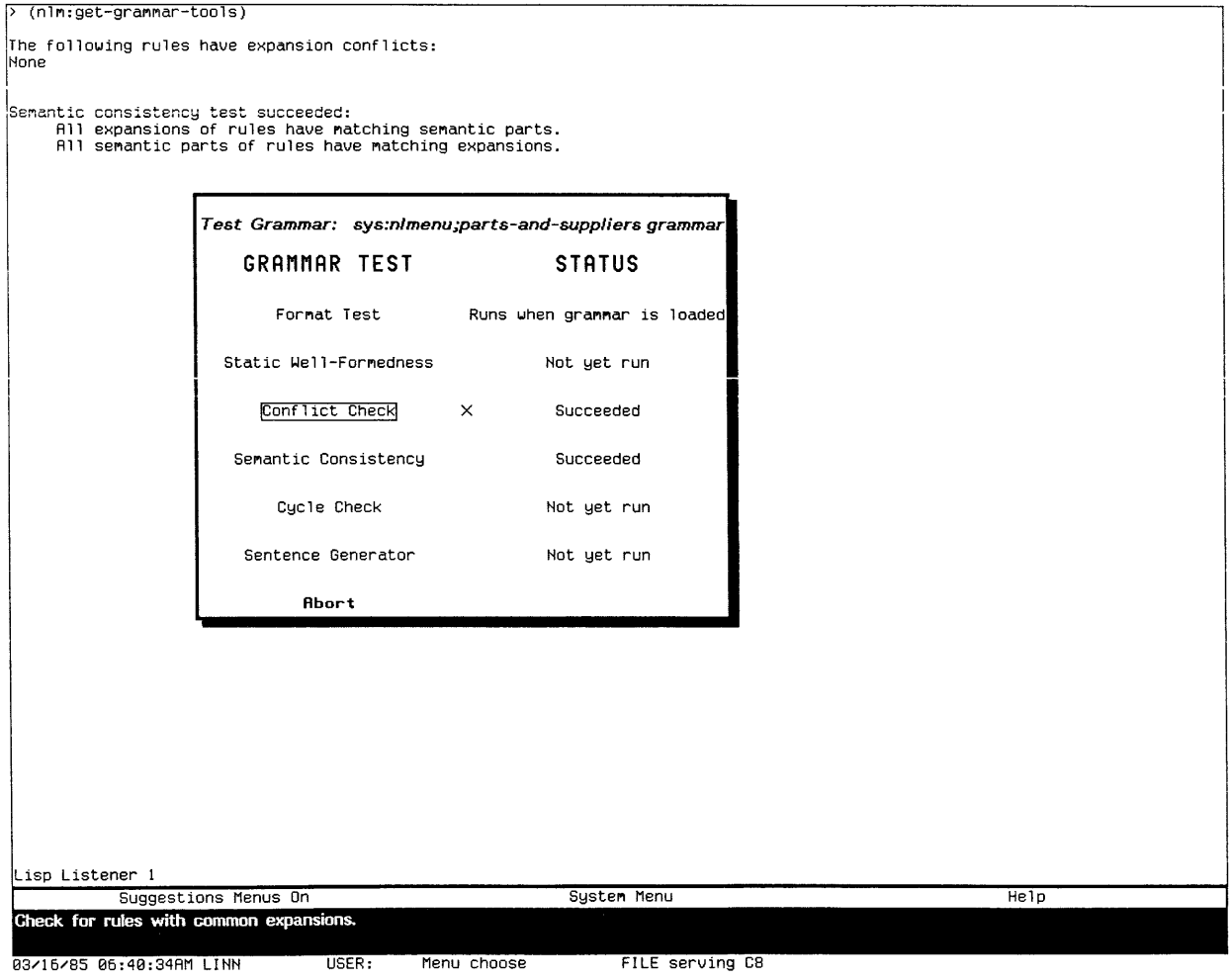
- Grammar — Specify the pathname for the source grammar. To specify the starter-kit grammar name, type `SYS:NLMENU;PARTS-AND-SUPPLIERS.GRAMMAR`.

- Lexicon — Specify the pathname for the source lexicon. To specify the starter-kit lexicon name, type `SYS:NLMENU;PARTS-AND-SUPPLIERS.LEXICON`.

- Root — Respond with the start symbol of the source grammar. In many instances, this is the symbol `S`, which is the default value.

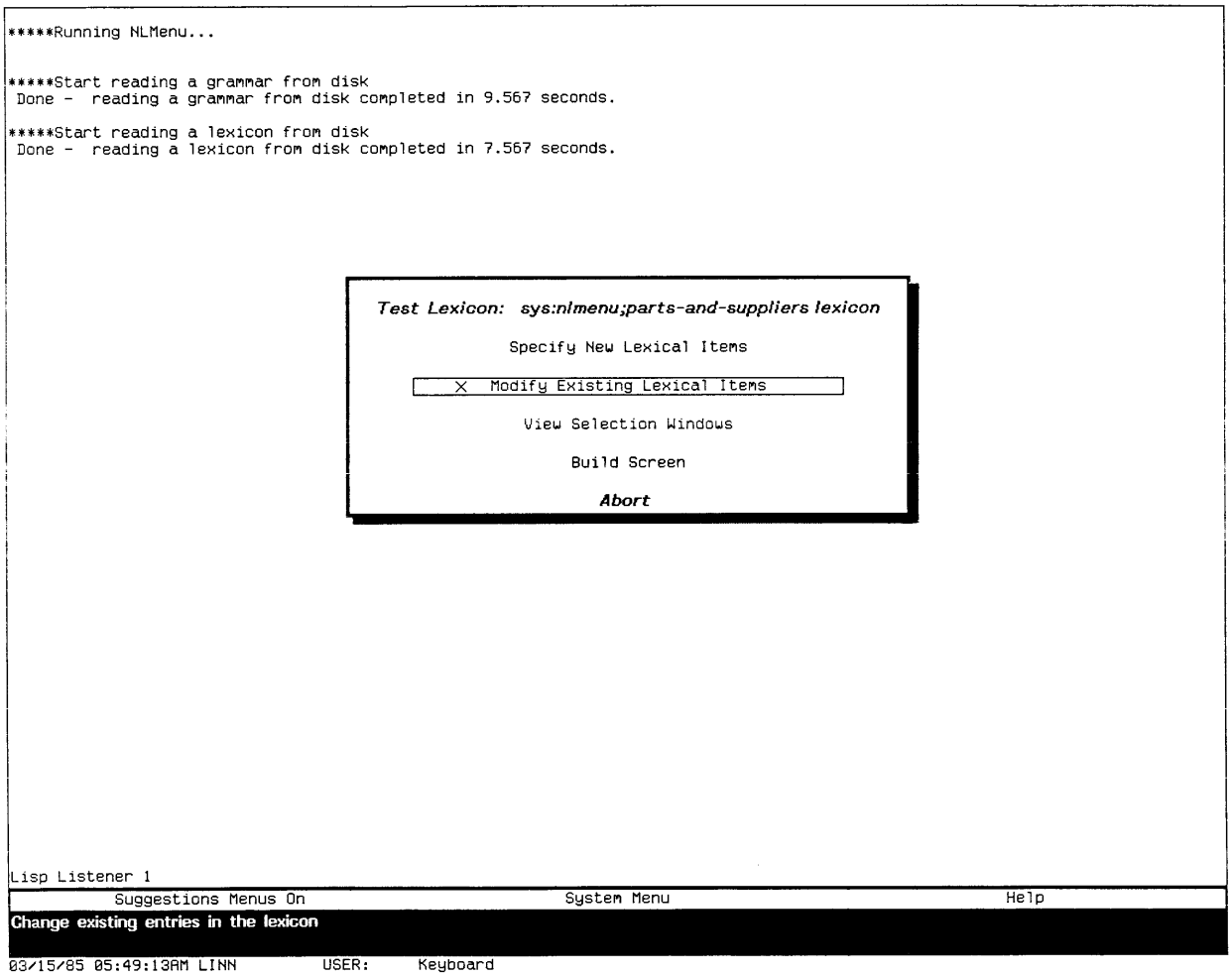
- Test the grammar? — Selecting `yes` causes the Grammar Test menu (see Figure 2-5) to appear after all input parameters have been collected. The menu is centered on the current position of the mouse. The default value is `no`, which is initially highlighted. You can also invoke the Grammar Tool menu directly with the function **nlm:get-grammar-tools**. In this case, first call the function **nlm:reset-grammar-tool-interface** to specify the name of the grammar to be tested. Otherwise, **nlm:get-grammar-tools** builds a Grammar Tool interface to the sample grammar. Section 5 discusses the various grammar tests in detail.

Figure 2-5 Grammar Test Menu



- Use lexicographer/screenbuilder? — Selecting `yes` causes the Lexicographer/Screen-builder menu (see Figure 2-6) to appear after all input parameters have been collected. The menu is centered on the current position of the mouse. The default value is `no`, which is initially highlighted. You can also invoke the lexicographer/screen-builder directly with the function **nlm:get-lexicographer**. However, first call the function **nlm:reset-grammar-tool-interface** to specify the name of the lexicon. Section 6 describes the various options available in the lexicographer.

Figure 2-6 Lexicographer/Screen-Builder Menu



- Panes Description — Use this parameter to control the arrangement of the selection menus in the NLMenu constraint window. There are four choices, documented in the who-line.

  - If you select **nil**, the system builds one row of panes with *n* columns, where *n* is the number of unique menus referenced in the lexicon (see paragraph 4.5). Long entries are truncated. This option can be sufficient for an interface still under development that will eventually require a nondefault pane arrangement.

  - The Default-Panes option causes the system to load a predefined set of window panes (see paragraph 7.2.4). Note that the lexicographer/screen-builder includes an option to assist in creating pane descriptions based on the current state of the lexicon.

  - You can select the third option, Screen-Builder-Panes, if you have run the Build Screen option of this utility (see paragraph 6.3.4).

  - You use the fourth option to designate a pathname for a file in which a panes description is stored. For information on creating a panes description, see paragraph 6.3.4.

- Constraints Description — With this parameter, you control the arrangement of the entire constraint window. The same four options described for the Panes Description parameter are available. For information on creating a constraint-window description, see paragraph 6.3.4.

- Experts File — Specify a pathname for a file in which any special-purpose experts are stored, or **nil** if there are none. For information concerning the creation of experts, see paragraph 4.9.

- Target Language — Specify the language into which natural language inputs are to be translated.

Items in the Interface Parameters choose-variable-values menu are mouse-sensitive, and documentation on each appears in the who-line. A default choice is always given. If both a default and a nondefault choice are displayed, the default is highlighted in boldface. If you select the nondefault, the highlighting changes to reflect your selection. If only a default value is displayed, changing the default involves the following general steps:

1. Click on the item. The default value disappears.

2. Type the desired value.

3. Press the RETURN key to record the change.

When you have specified all parameters, click on the Do It option. If you instead wish to abort, selecting the Abort option returns you to the NLMenu System menu. If you select Do It, the system checks whether the grammar you specified is already compiled. If it is, the compiled grammar is loaded. If the grammar is not compiled, the system calls the grammar compiler, compiles the grammar and lexicon, and saves the compiled version to disk. This saved version is automatically used on the next invocation of any interface using the same grammar, unless either the source grammar or lexicon files have changed since the grammar was last compiled.

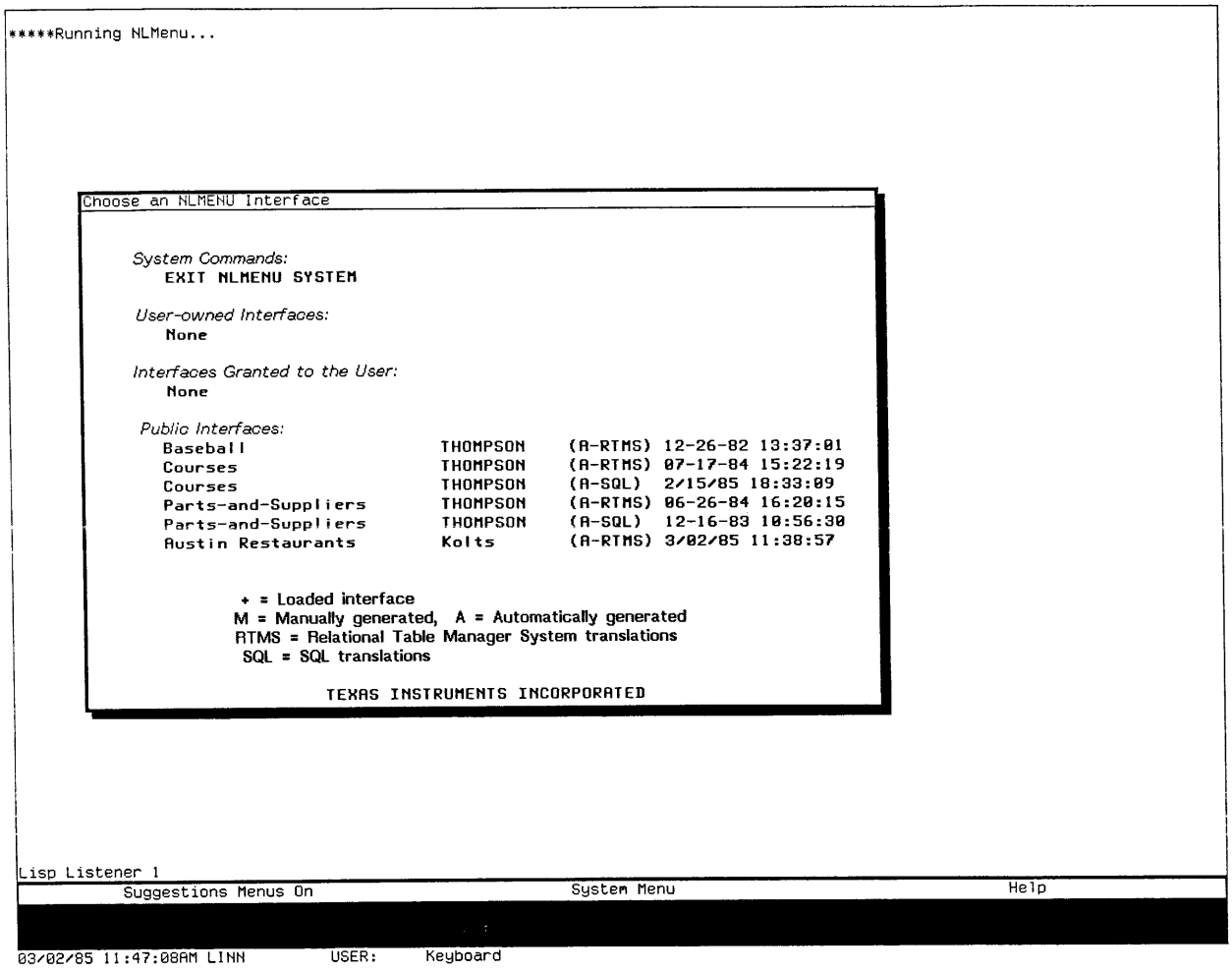
Once grammar compilation and loading are complete, the system makes the NLMenu constraint window. Then it creates window items, based on the lexicon, for the selection menus. When these processes are complete, a record containing the parameters that specify the new interface is appended to the file SYS:NLMENU;NLMENU-INTERFACES (the file is automatically created if it does not exist). Subsequent invocations of NLMenu include the new interface on the NLMenu System menu, preceded by a plus sign (+) if the interface is loaded. Finally, the NLMenu constraint window appears, and you can use the interface.

To remove an interface from the NLMenu System menu, select the Delete Interface option from the NLMenu System menu (see Figure 2-3), or use the editor to delete its defining entry in the file SYS:NLMENU;NLMENU-INTERFACES.

NLMenu System Menu With Database Interface

2.2.1.3 When you select the database-dependent NLMenu System menu and designate the database to be loaded, the system displays the NLMenu System menu which has the appearance shown in Figure 2-7.

Figure 2-7 Typical NLMenu System Menu — With RTMS



This menu gives you access to NLMenu interfaces that you have previously created or have been granted access to by other users. The interface management scheme that controls the presentation of NLMenu interfaces on the NLMenu System menu is discussed in detail in paragraph 7.2.2.

When you select an interface, the system first determines whether you have specified a portable spec file (specifying a set of categories containing all the domain-dependent information necessary for a complete user-interface grammar and lexicon) in the tuple entry (row of an interface database table) for the selected interface. This tuple is in the NLMENU-INTERFACES relation (database table) of the currently active RTMS database (see paragraph 7.2.2.2). If you have specified a portable spec file, and if the grammar-file and lexicon-file entries are **nil** in the NLMENU-INTERFACES relation, the system does the following:

- Reads the portable spec file from disk
- Examines the VERSION entry in the NLMENU-INTERFACES relation
- Generates a grammar and lexicon of the appropriate type

Currently supported types of database interfaces are Structured Query Language (SQL) and RTMS. In the latter case, natural language input is translated to the RTMS query language that is executable on the Explorer. The automatically generated source grammar and lexicon are then saved to disk. The tuple entry for the interface is updated to reflect that a grammar and lexicon are now available. The pathnames of the source grammar and lexicon are stored in the grammar-file and lexicon-file entries. The string *grammar generated* is stored in the portable-spec-file entry. The modified NLMENU-INTERFACES relation is not automatically saved. If grammar regeneration is not required, you can save the modified relation with the call `(rtms:save-relation* nlmenu-interfaces)`. From this point, the process is similar to what occurs when the RTMS toolkit is not available:

- The grammar is compiled if necessary.
- The compiled grammar is written to disk.
- The NLMenu constraint window is created.
- Menu items are filled in.
- The constraint window is exposed.

Appropriate messages appear on the video display informing you of the progression of these processes.

Invoking NLMenu From an Application

2.2.2 To invoke the NLMenu system from an application program, you must call the **nlm:nlmenu-driver** function. A description of the syntax and the parameters of this function follows.

nlm:nlmenu-driver *interface-name grammar-file lexicon-file root* &optional *panes constraints experts owner version* — Function

Arguments:

interface-name — This parameter specifies the name of the interface to be invoked by the NLMenu system.

grammar-file — This parameter specifies the pathname for the source grammar. Supply the value as a string in double quotes.

lexicon-file — This parameter specifies the pathname for the source lexicon. Supply the value as a string in double quotes.

root — This parameter specifies the start symbol of the source grammar.

panes — This parameter specifies the panes description for the interface. Its value is one of the following:

- A string in double quotes representing the pathname of the file containing the panes description
- A symbol bound to the panes description
- **nil**

Specifying a value for this parameter is optional. If no value is provided, the NLMenu system uses defaults that are appropriate for many interfaces.

constraints — This parameter specifies the constraints description for the interface. Its value is one of the following:

- A string in double quotes representing the pathname of the file containing the constraints description
- A symbol bound to the constraints description
- **nil**

Specifying a value for this parameter is optional. If no value is provided, the NLMenu system uses defaults that are appropriate for many interfaces.

experts — The value of this parameter is a double-quoted string that designates the pathname of a file containing special-purpose experts. This is an optional parameter.

owner — The value of this parameter is a double-quoted string that appears on the NLMenu System menu to indicate who created the interface. This is an optional parameter.

version — The value of this parameter is a symbol designating the type of interface; for example, SQL or RTMS are possible values. This is an optional parameter.

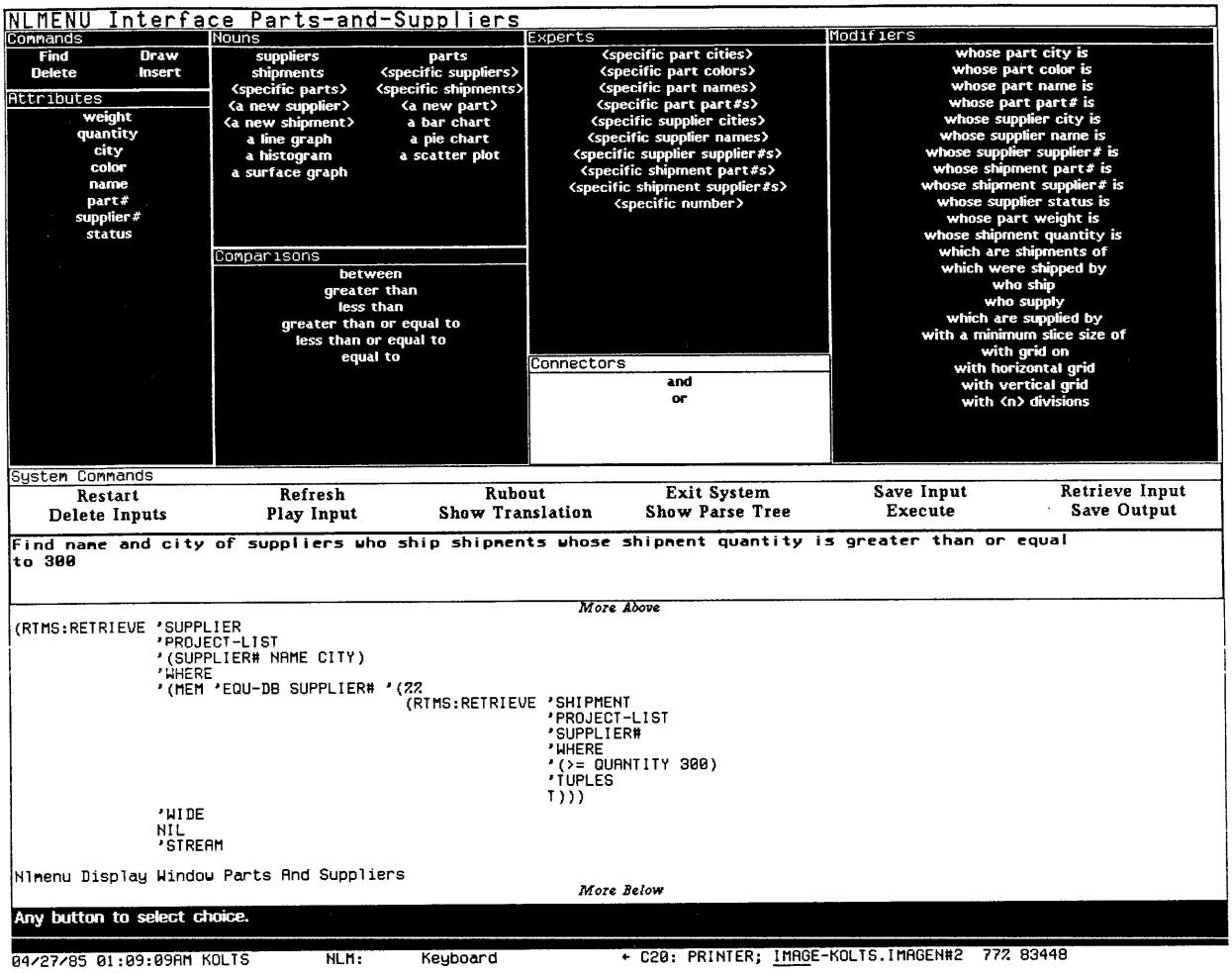
Screen Appearance and Sentence Building

2.2.3 The following paragraphs describe how NLMenu guides you (or the user of your interface) in composing sentences that are guaranteed to be understood by the system. After you examine the NLMenu System menu and select or build an interface, the NLMenu screen for that interface appears. This screen consists of the following five windows (unless you have built your own special-purpose constraint window, see paragraph 6.3.4):

- Logo window
- Selection window
- System Commands window
- Input window
- Display window

Figure 2-8 illustrates an NLMenu screen for the Parts-and-Suppliers interface.

Figure 2-8 Parts-and-Suppliers Interface Screen



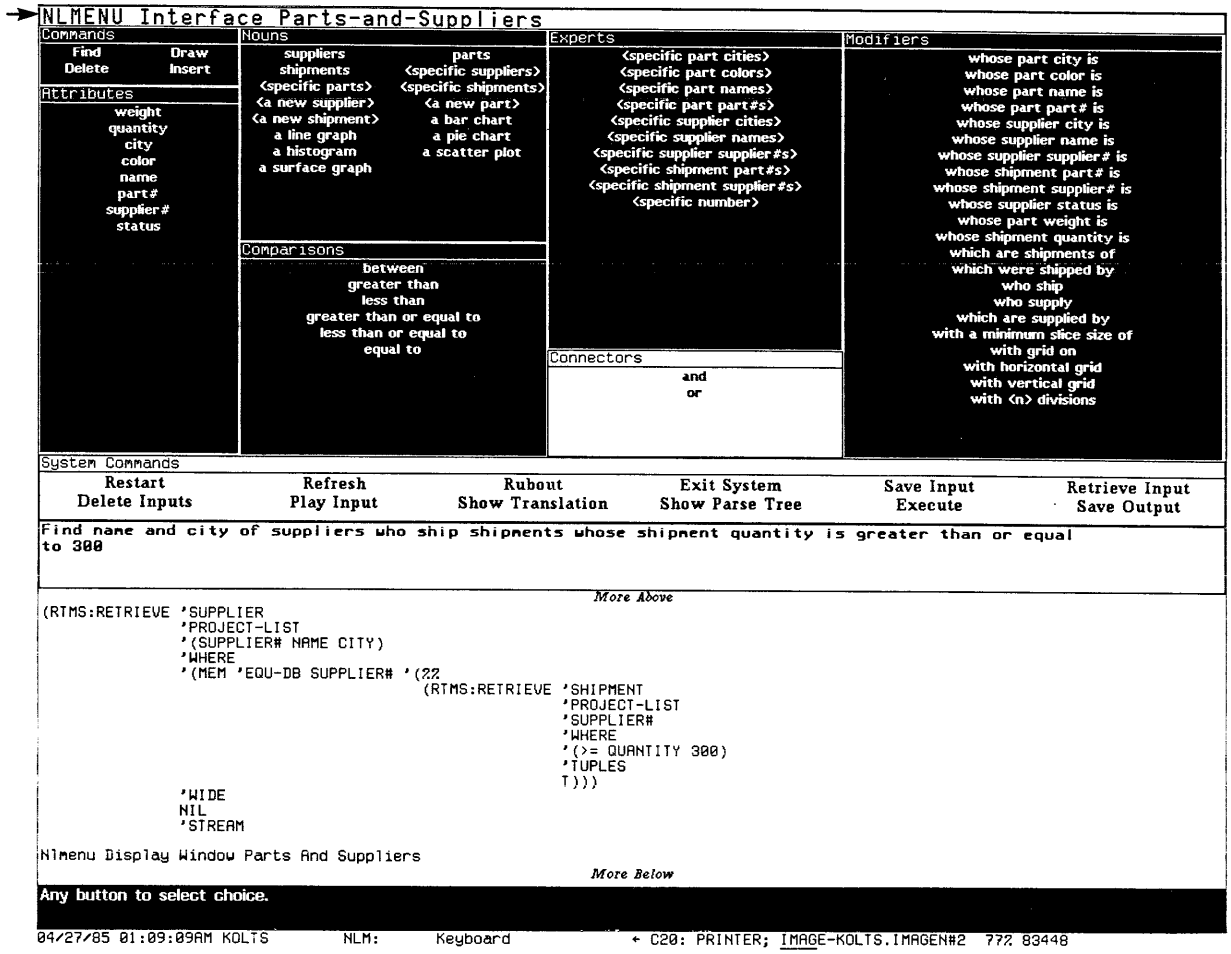
The following discussion assumes that this is the selected interface, and that the translations generated by the system are in the RTMS query language.

Interface Screen and Windows

2.2.3.1 The interface screen is a constraint frame made up of several windows, some of which contain lists of items, or menus, for selection. The following paragraphs describe these windows in more detail.

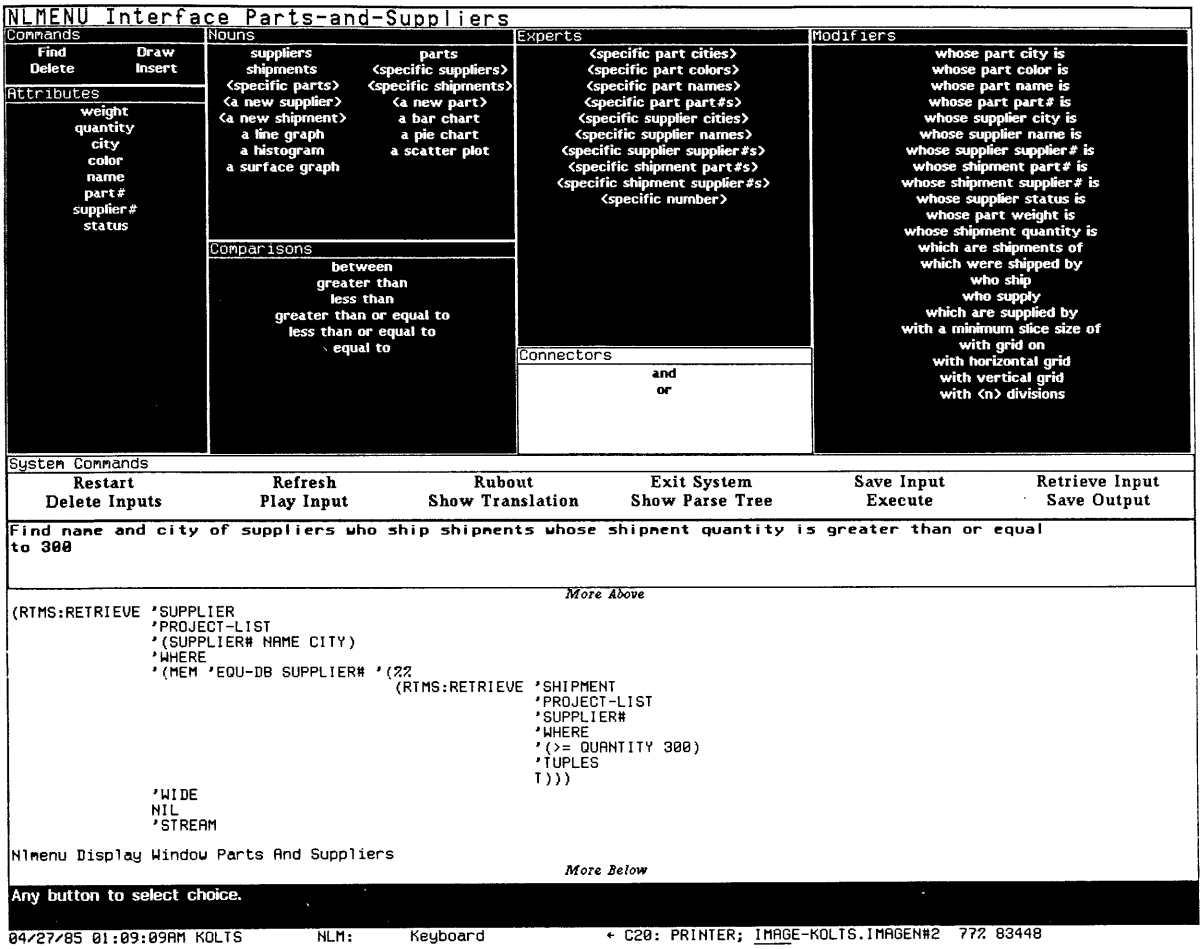
Logo Window The logo window (see Figure 2-9) is where the interface name appears.

Figure 2-9 Parts-and-Suppliers Logo Window



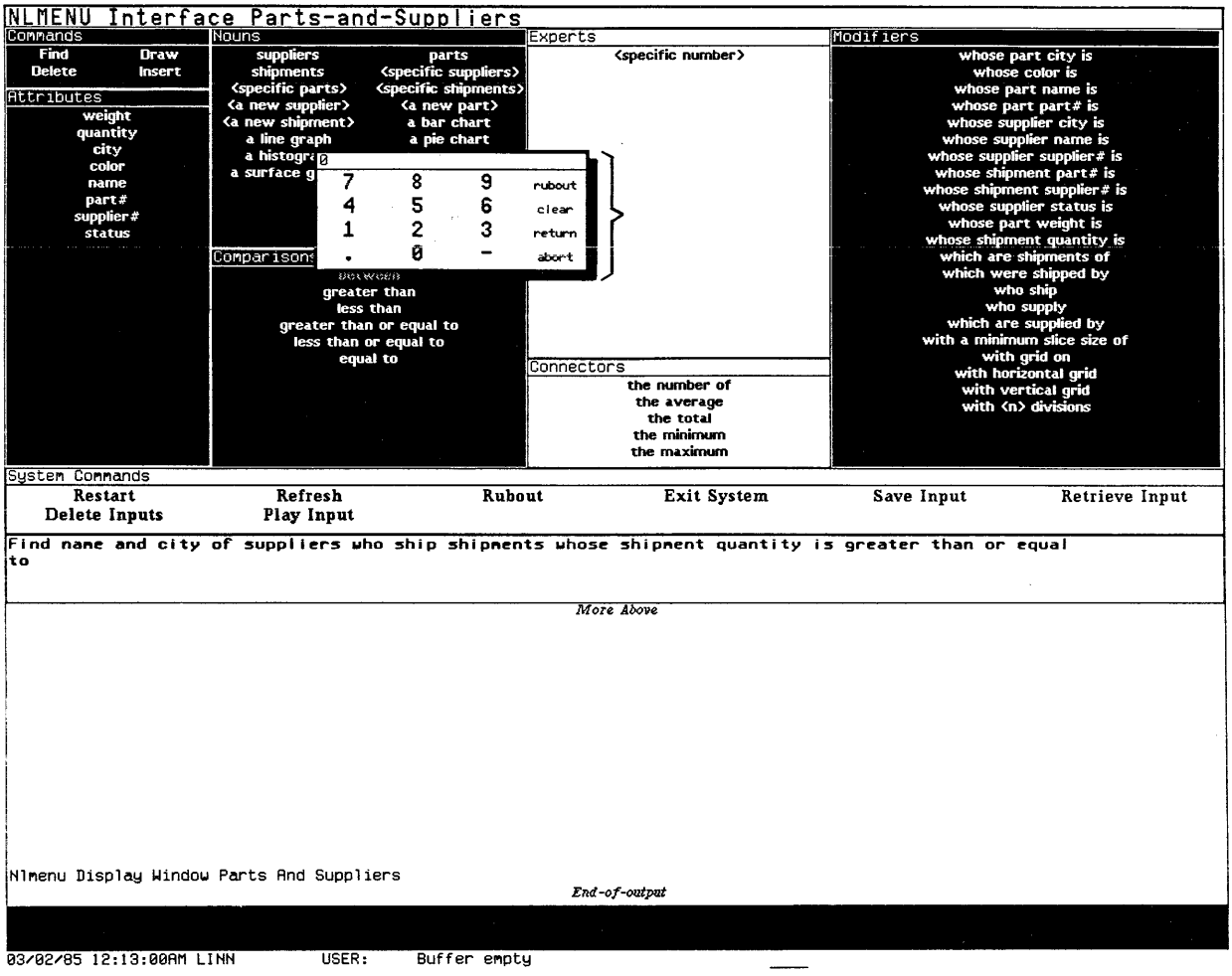
Selection Window The selection window (see Figure 2-10) consists of several panes. The total number of panes depends on the panes description, which in turn depends on the underlying grammar and lexicon. Each pane has a label corresponding to a menu referenced in one or more lexical entries (see paragraph 4.5). Typically, this label names a grammatical category. Within each pane is a list of items, each of which corresponds to a possible input word or phrase. You can scroll the selection panes by bumping the mouse cursor against the left border of the window until the mouse cursor is transformed into a thick double arrow and a scroll bar appears. Instructions in the who-line documentation specify procedures for scrolling in various directions.

Figure 2-10 Parts-and-Suppliers Selection Window



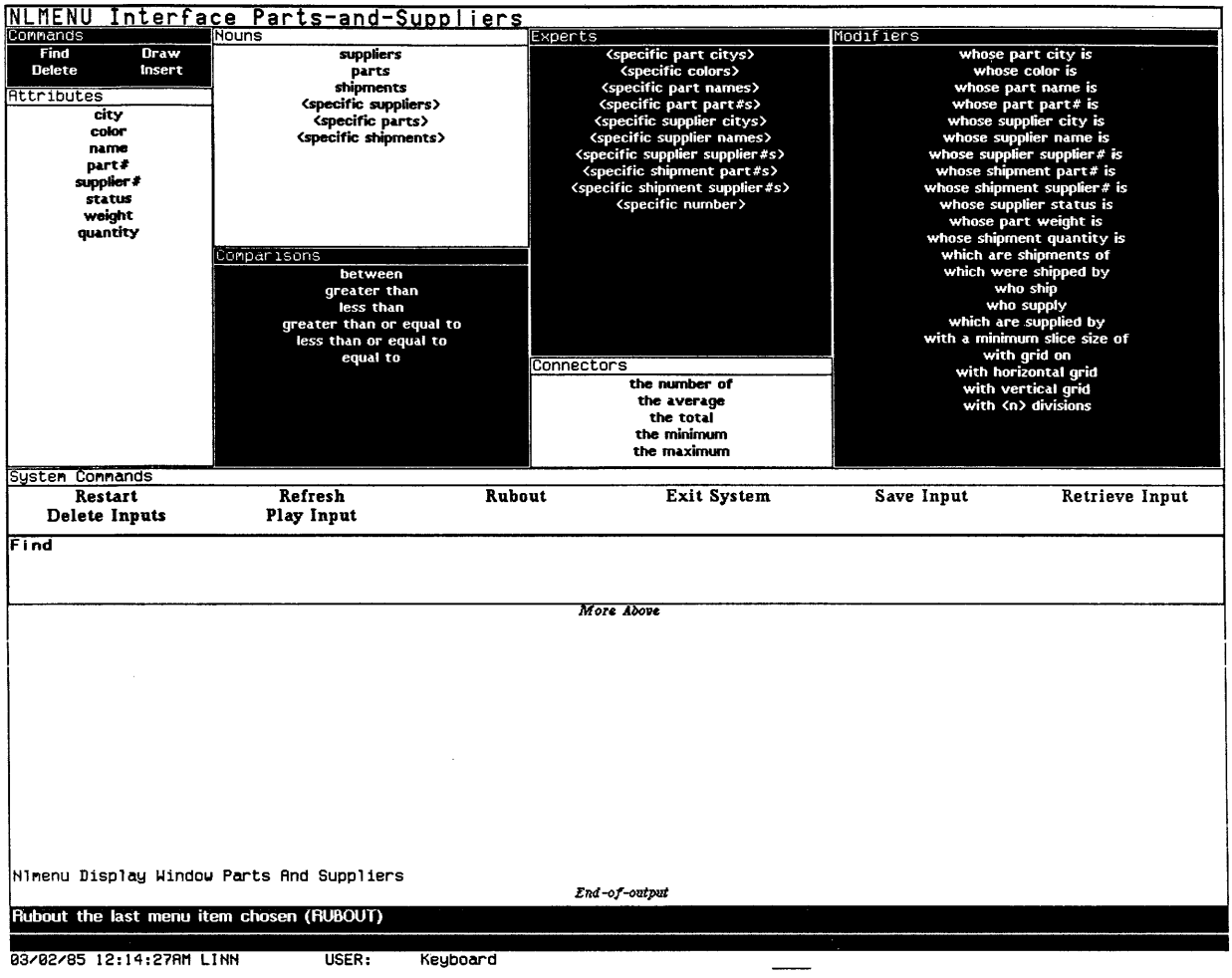
Some items that you select invoke interaction *experts*. These items are typically enclosed in angle brackets. Experts allow variable input to be entered by means of a special-purpose interactive display. When an expert pane item is selected, a pop-up window or menu (see Figure 2-11) appears where you can enter or select a desired value. Code to invoke experts is incorporated in the underlying lexicon. Experts are individually programmable in the NLMenu system; each type of expert is a specialized piece of code. For example, one type of expert is the calculator expert. It works like a hand-held calculator. Each item in the calculator expert pop-up window is mouse-sensitive. For more details about experts, see paragraph 4.9.

Figure 2-11 Parts-and-Suppliers Calculator Expert



Selection panes change as you make selections (see Figure 2-12), allowing you to make only meaningful choices. Active panes, from which you can make selections, appear with menu items listed against a white background. Inactive panes appear in reverse video. As you make selections, the active/inactive status of windows changes. The window items themselves, and whether they are selectable, also change dynamically. The parser parses each meaning unit as it encounters it. The parser pursues each possible parse path and, on the basis of this search, determines which selections are valid as the next element. Based on this determination, the NLMenu system activates only those window panes and only those window pane items that are valid next selections.

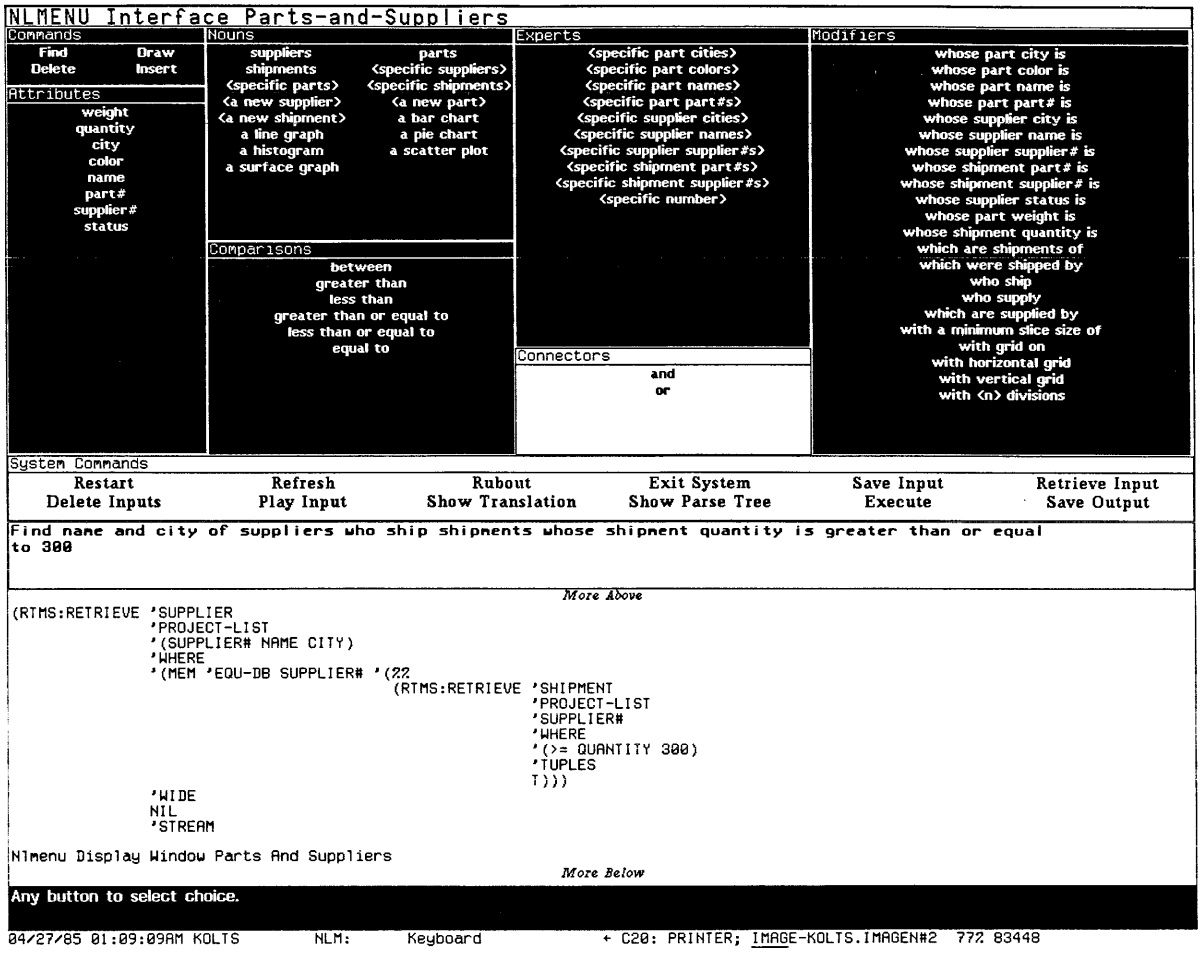
Figure 2-12 Parts-and-Suppliers Active/Inactive Panes



There are no restrictions on the grammar with respect to ambiguity. If the system detects multiple parses (interpretations) of your input sentence, a pop-up window appears with a listing of the various interpretations. These parses are arranged according to probability of intent. From this pop-up window, you select the parse that corresponds to your intent.

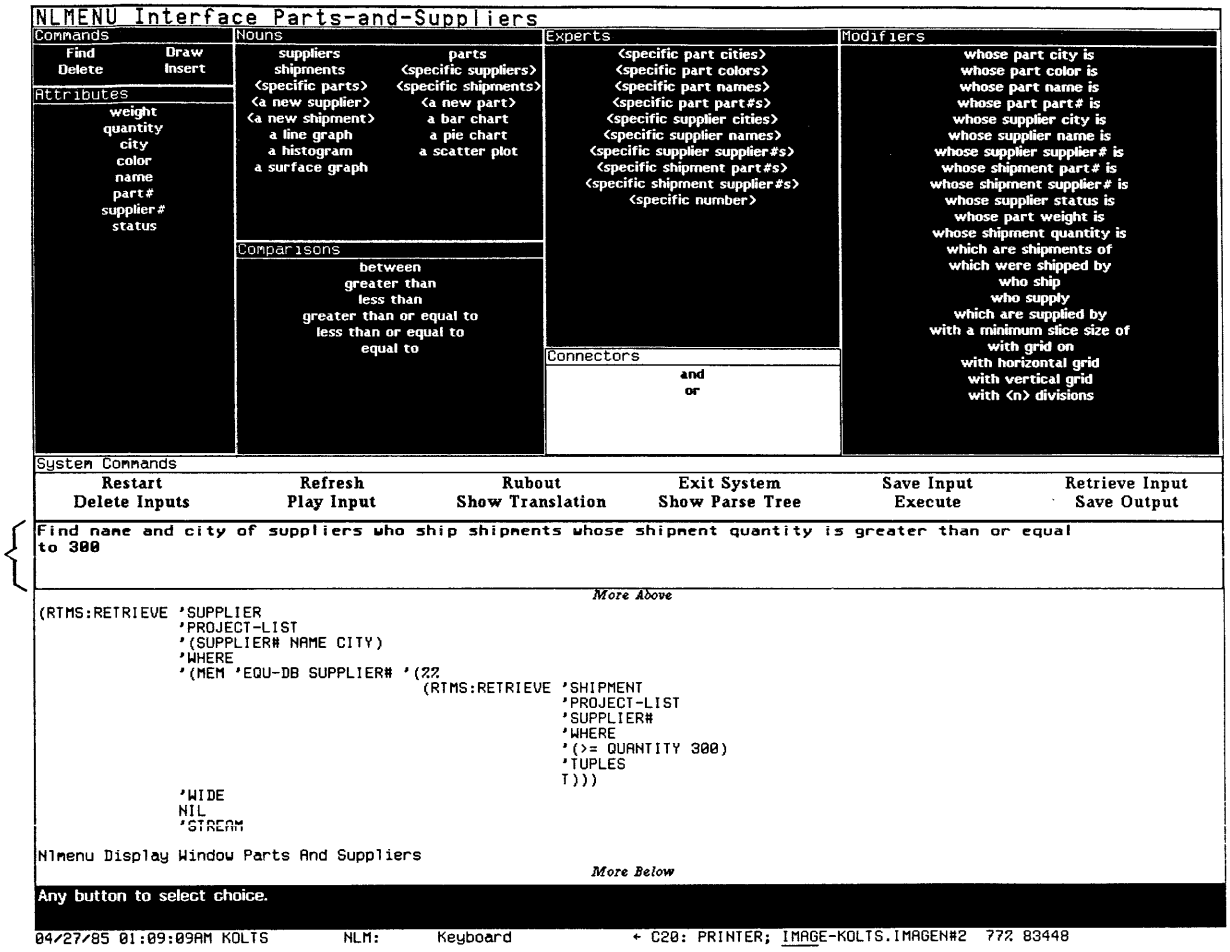
System Commands Window The System Commands window (see Figure 2-13) contains commands for building new inputs or performing operations on existing inputs. Refer to paragraph 2.2.4 for more detailed information on these commands. Documentation on these commands is also available on the who-line. The list of available commands depends on the state of the input sentence you are constructing. For instance, the Execute command only becomes available when you have constructed a complete input sentence.

Figure 2-13 Parts-and-Suppliers System Commands Window



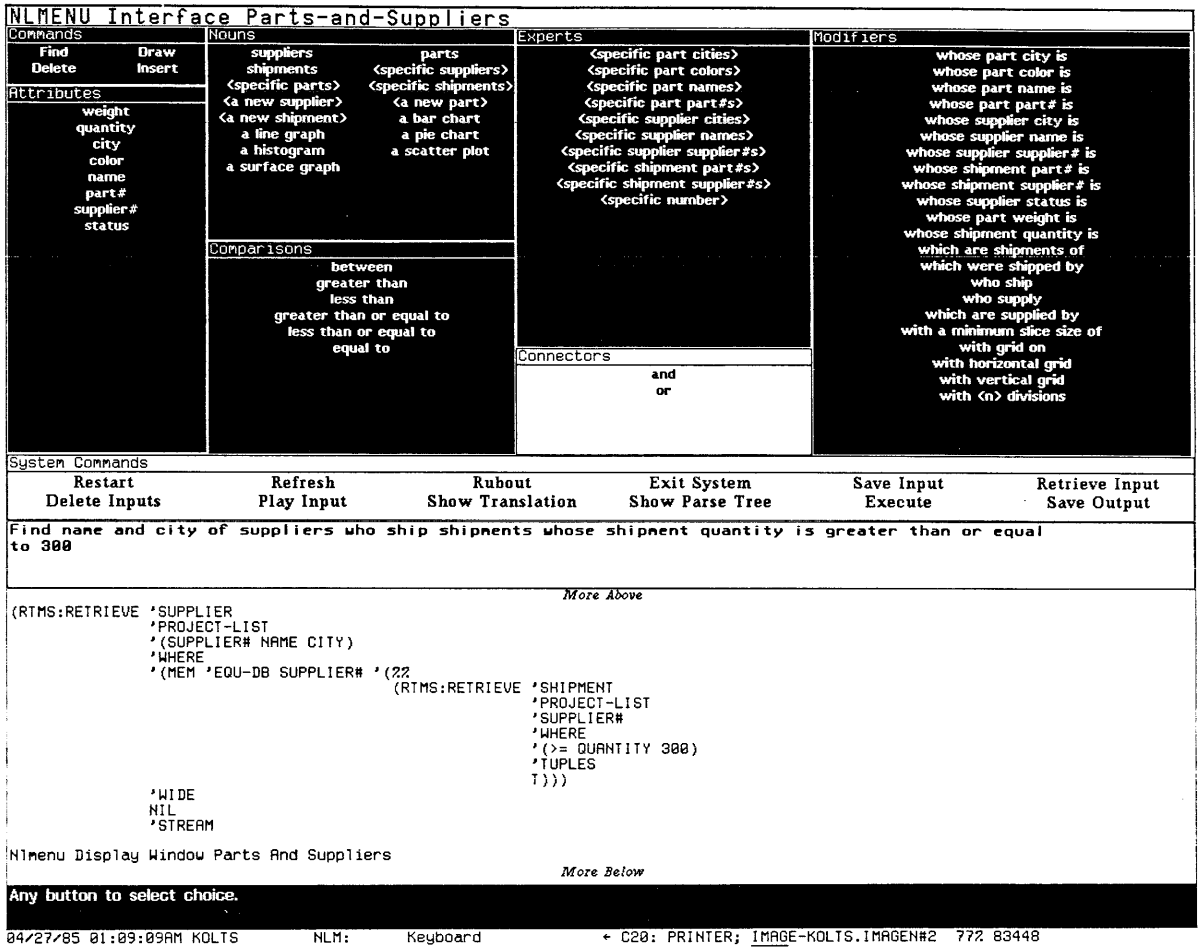
Input Window The input window (see Figure 2-14) is where the natural language sentence appears as you build it. This window expands dynamically, allowing you to construct lengthy, involved input sentences.

Figure 2-14 Parts-and-Suppliers Input Window



Display Window The display window (see Figure 2-15) presents the results of executing the various system commands.

Figure 2-15 Parts-and-Suppliers Display Window



For example, any output that your application produces in response to your input sentence is displayed in this window. The display window can be scrolled both up and down. The labels Beginning-of-Output and End-of-Output appear as appropriate in the output display. An output display can contain more information than will fit into the physical dimensions of the window. You can move unseen output into the display window either a page at a time or on a line-by-line basis. To scroll a new page of information into the display window, bump the mouse cursor against the left side of the window until it becomes boldface, with arrows pointing both up and down. The information in the mouse documentation line tells you what options are then available to you with the various mouse clicks (left, right, or middle). To move new information into the window on a line-by-line basis, bump the mouse cursor against the appropriate end of the window. Depending on the current window position, there can be more information above, below, or both above and below the present location. The messages More Above and More Below appear as appropriate to indicate your position.

Moving Around the Interface Screen

2.2.3.2 Usually, you use the mouse to move around the interface screen (from item to item and from menu to menu) as you make your selections. You can also use keystrokes to move around the screen. Use the arrow keys to move from item to item within a selection menu, and use the CTRL key with the arrow keys to move from menu to menu.

NLMenu Help Facilities

2.2.3.3 The NLMenu system provides a help facility for either individual items in a selection menu or for selection menus as a whole.

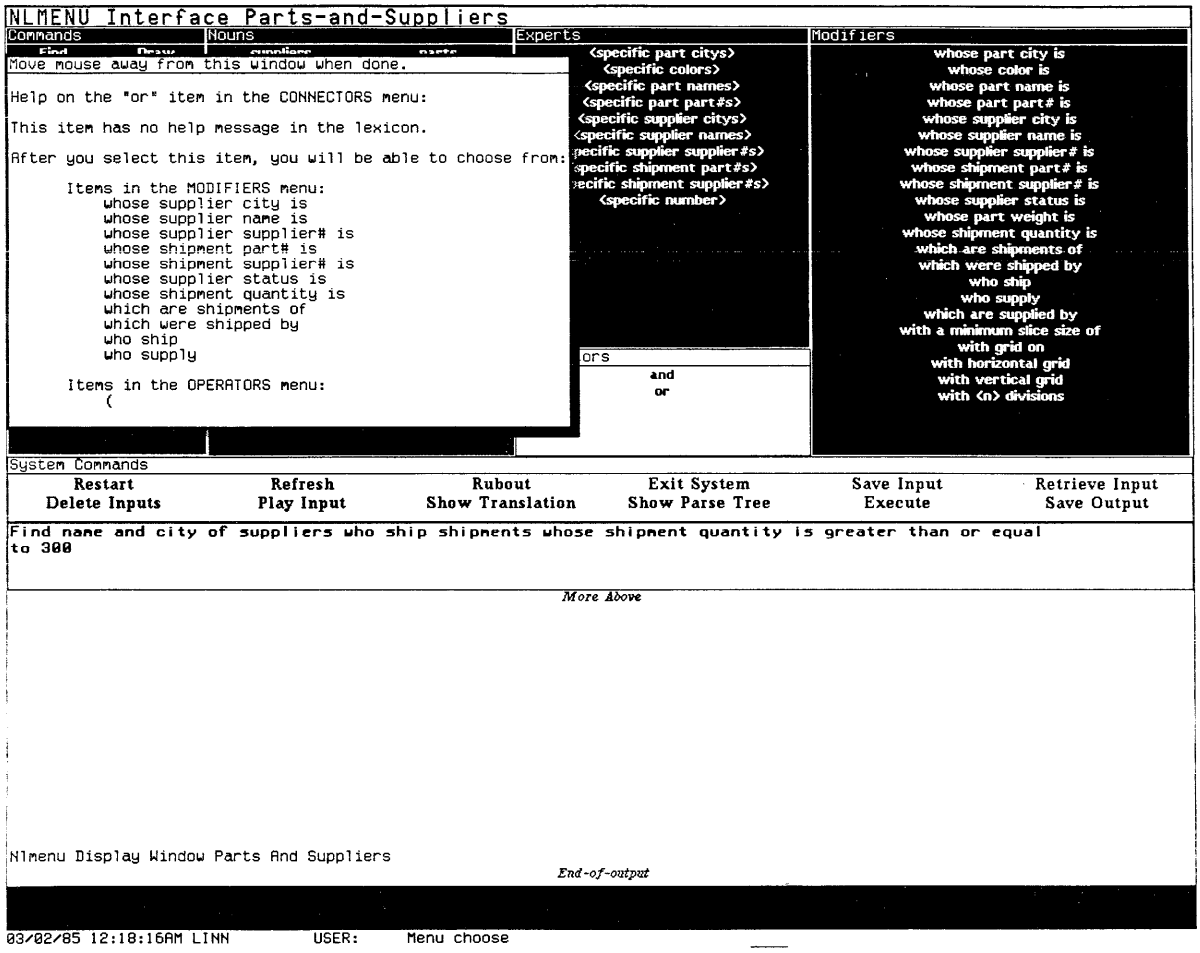
To obtain item help, move the mouse cursor so that it boxes the item for which you desire help. Then either press the HELP key or click once on the middle mouse button. A pop-up window appears containing the following information:

- A description of the item and its menu
- The help message for the item
- The consequences of selecting the item (that is, what menu(s) and item(s) would become active next)

The help message is taken from the NLMenu lexicon for the interface. The process of designating help messages is described in Section 4.

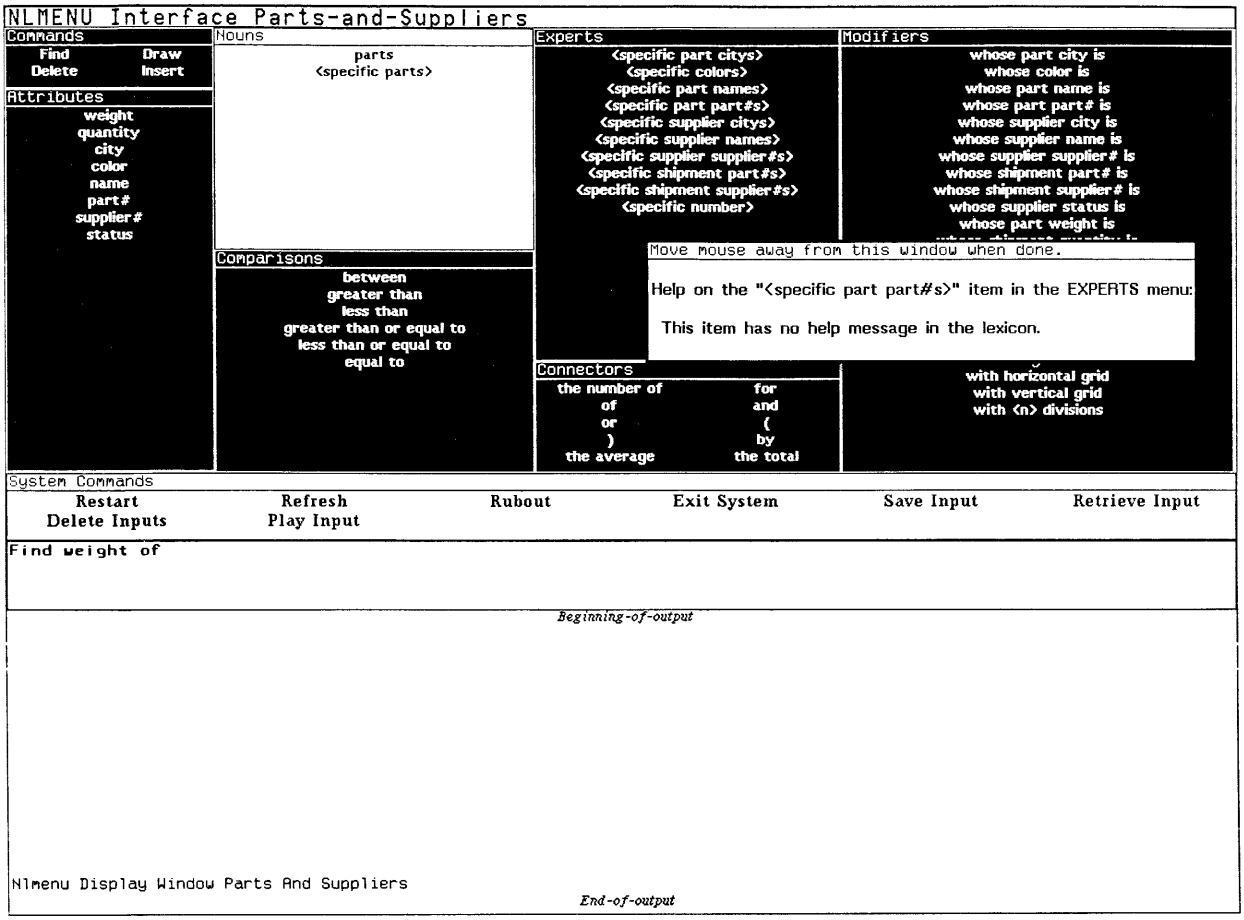
Figure 2-16 presents an example of help information for an item in an active menu.

Figure 2-16 Active Menu Item Help Information



You can obtain help on any item within a selection menu, regardless of whether that menu is currently active. However, if you seek help on an item in an inactive menu, you receive only a description of the item and its menu and the help message for the item. Since the item is inactive, it is impossible to select and, therefore, there are no selection consequences to report. Figure 2-17 is an example of help information for an item in an inactive menu.

Figure 2-17 Inactive Menu Item Help Information



When finished with the help message, move the mouse cursor out of the pop-up window, and the window disappears.

If you seek help on some item in a window other than a selection menu (for example, the logo window or the display window), the video display flashes, indicating that such an operation is invalid.

If you seek menu help, do the following:

1. Move the mouse cursor so that it is in the menu for which you desire help.

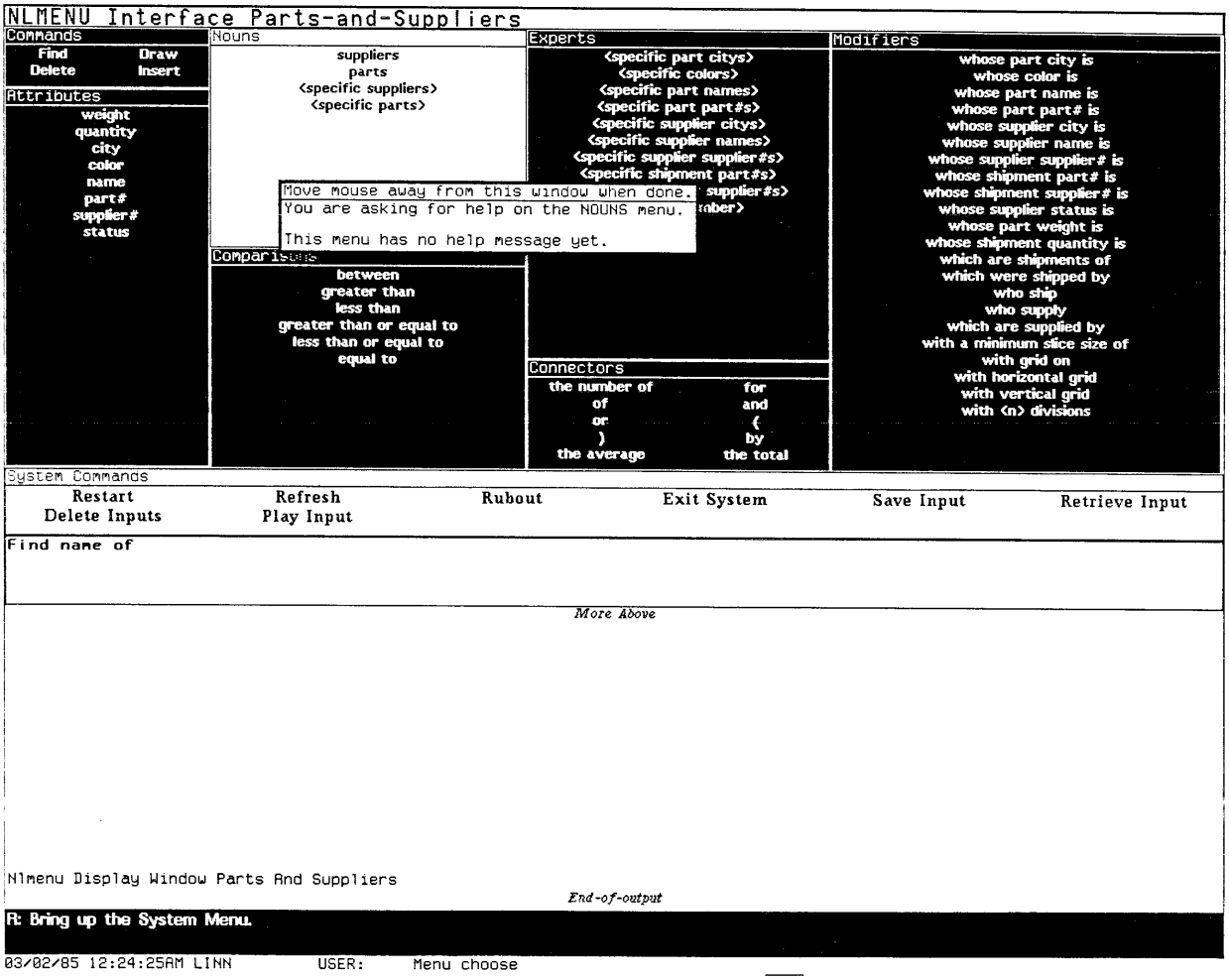
2. Either press CTRL-HELP, META-HELP, or click right once on the mouse.

In response, the NLMenu system displays a pop-up window with the following information:

- The name of the menu
- The help message associated with the menu (If you are working with the default screen layout, you get the description of the menu as the second item.)

Figure 2-18 is an approximate example of help information for a menu.

Figure 2-18 Menu Help Information



When you are finished with the help message, move the mouse cursor out of the pop-up window.

If you attempt to seek help in some window other than a selection menu, the video display flashes.

Description of Commands

2.2.4 The following paragraphs present brief descriptions of the options that appear in the System Commands window of an NLMenu interface. You can select these options by clicking left once on the mouse. If you prefer, you can use keystrokes to enter your selection. The keystroke equivalents for these options appear in the who-line documentation on the screen.

Restart Command

2.2.4.1 The Restart command clears the current parse without saving it and restarts the parsing process. Any contents of the input window are erased when this command option is exercised.

Refresh Command

2.2.4.2 The Refresh command clears the display window but maintains the current state of the parse. You can restore the display by scrolling backward.

Rubout Command

2.2.4.3 The Rubout command backs up the parse one element at a time, erasing as many elements as you select from the input window.

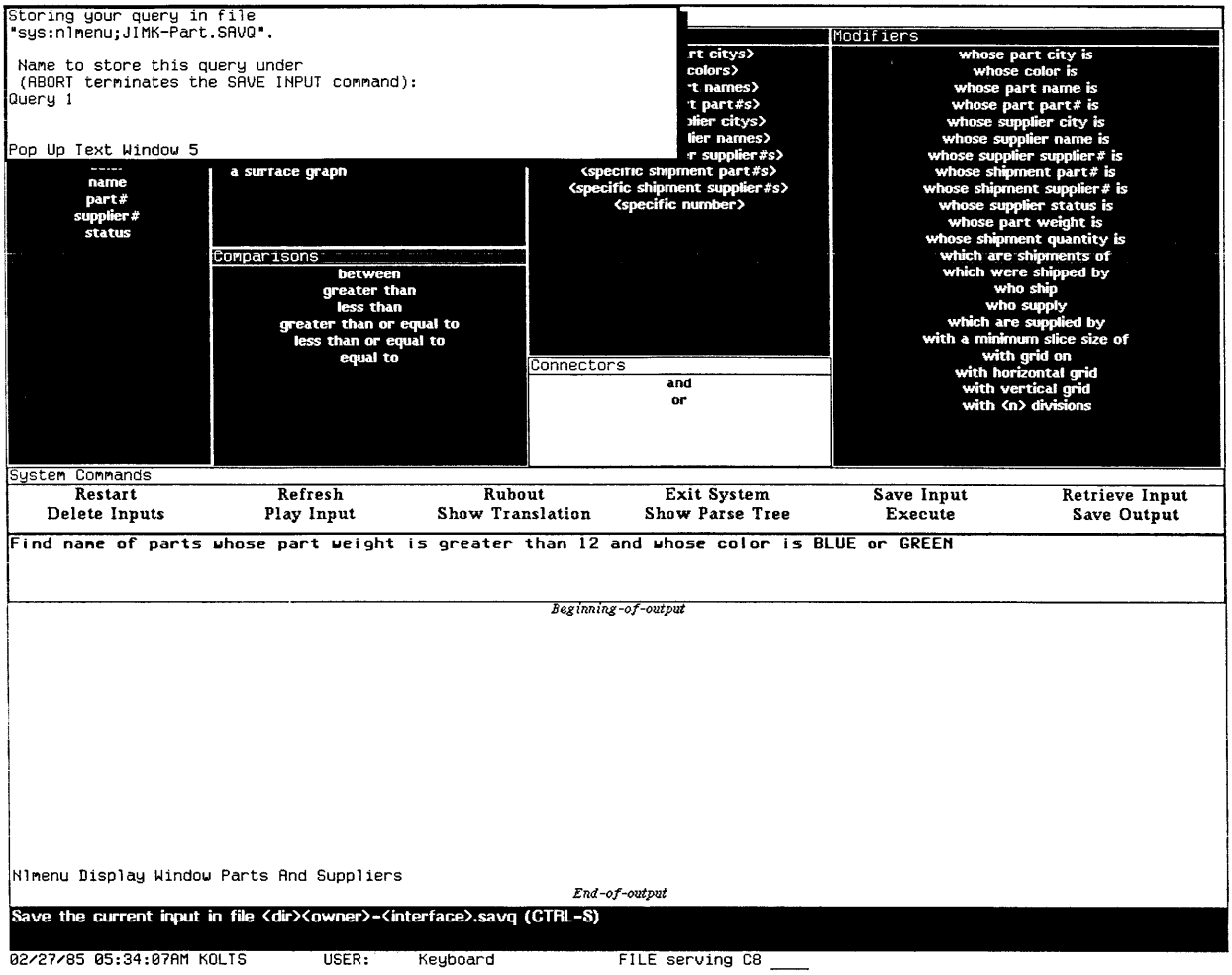
Exit System Command

2.2.4.4 Select the Exit System command to leave the interface and return to the NLMenu System menu. The system maintains the current state of the interface window. To return to the Lisp Listener, select the Exit NLMenu System option in the NLMenu System menu.

Save Input Command

2.2.4.5 The Save Input command saves the current sentence to a file. A pop-up text window (see Figure 2-19) appears with a descriptive message at the top and tells you where it is saving the input sentence.

Figure 2-19 Save Input Command Pop-Up Text Window



The name of the file has the following format:

SYS: *dir; owner-interface*.SAVQ

where:

SYS is the logical name for some machine.

dir is the directory name.

owner is the login name truncated to four characters.

interface is the interface name truncated to four characters.

Example: SYS:NLMENU;USER-SAMP.SAVQ

In response to the prompt, designate the name under which the system is to save the sentence. If the file does not currently exist, the system automatically creates the file and gives it the appropriate extension. Otherwise, the saved input sentence is appended to an existing file.

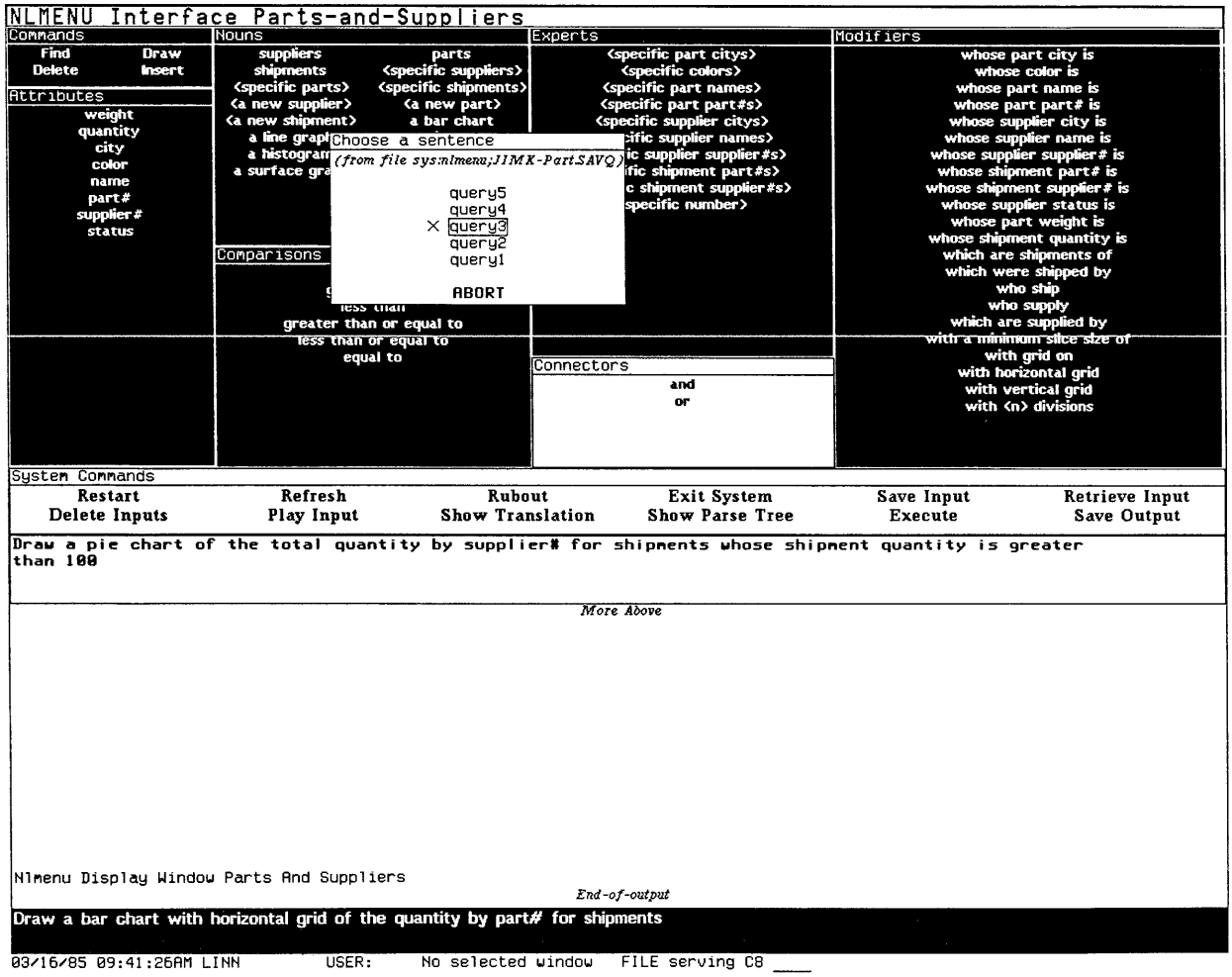
To exit the window without saving the input sentence, press the ABORT key. Also, if you do not give the input sentence a name, the NLMenu system does not save the sentence (that is, it does not write it to the designated file).

Retrieve Input Command

2.2.4.6 The Retrieve Input command allows you to recall a previously saved sentence. Once the sentence is retrieved, you can apply any of the options from the System Commands window to it. The input is taken from a file that the system automatically created previously when you selected the Save Input command.

When you select the Retrieve Input command, a pop-up window (see Figure 2-20) appears containing the names of saved input sentences.

Figure 2-20 Retrieve Input Command Pop-Up Window

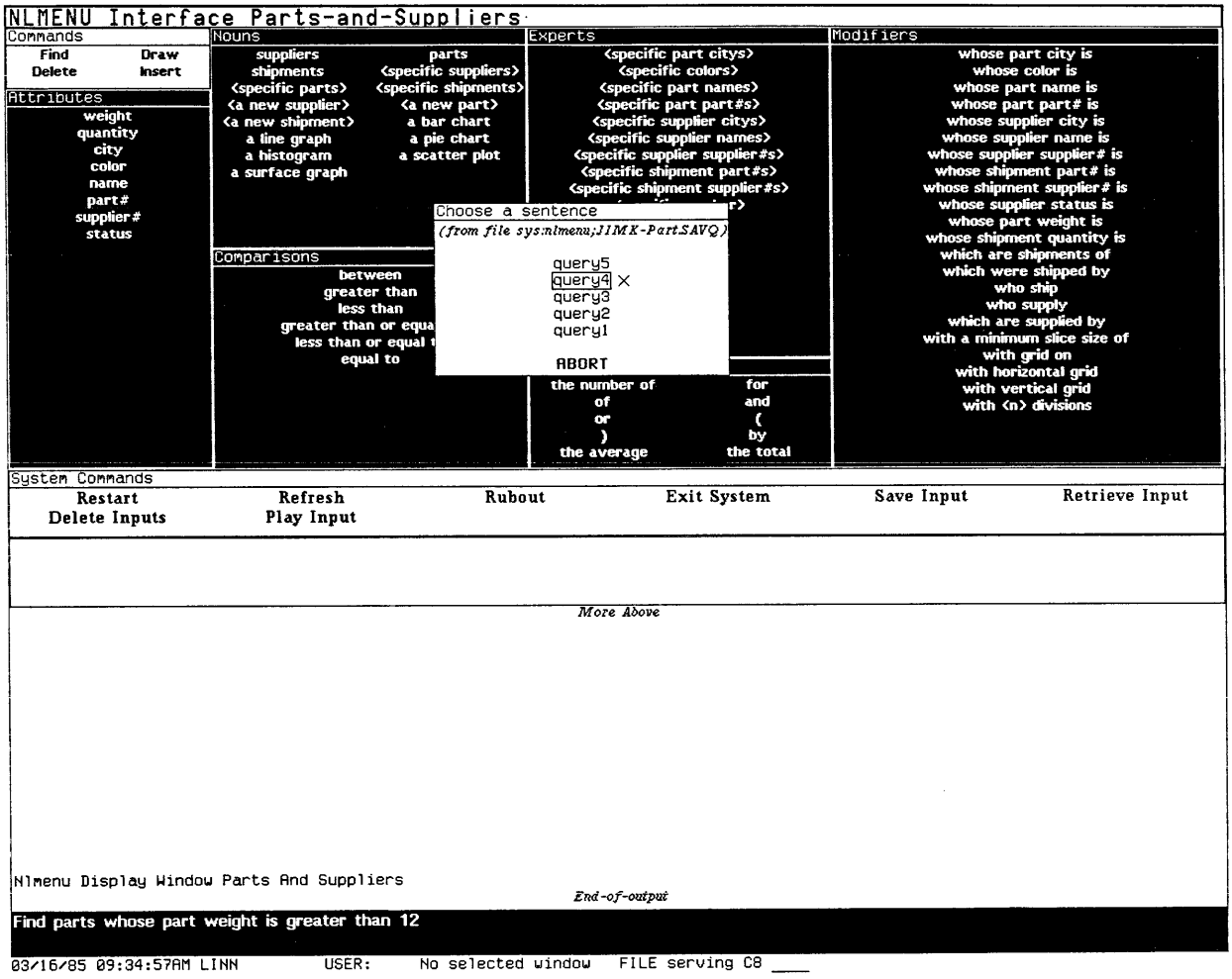


The items in this menu are mouse-sensitive. Position the mouse cursor over the name of a saved sentence to display that sentence on the who-line. Position the mouse cursor and click left once on the mouse to make a selection. The Abort option is also present, allowing you to exit the pop-up window and terminate this option.

Delete Inputs Command

2.2.4.7 The Delete Inputs command enables you to delete previously saved input. A pop-up window (see Figure 2-21) appears containing the names of all saved input sentences.

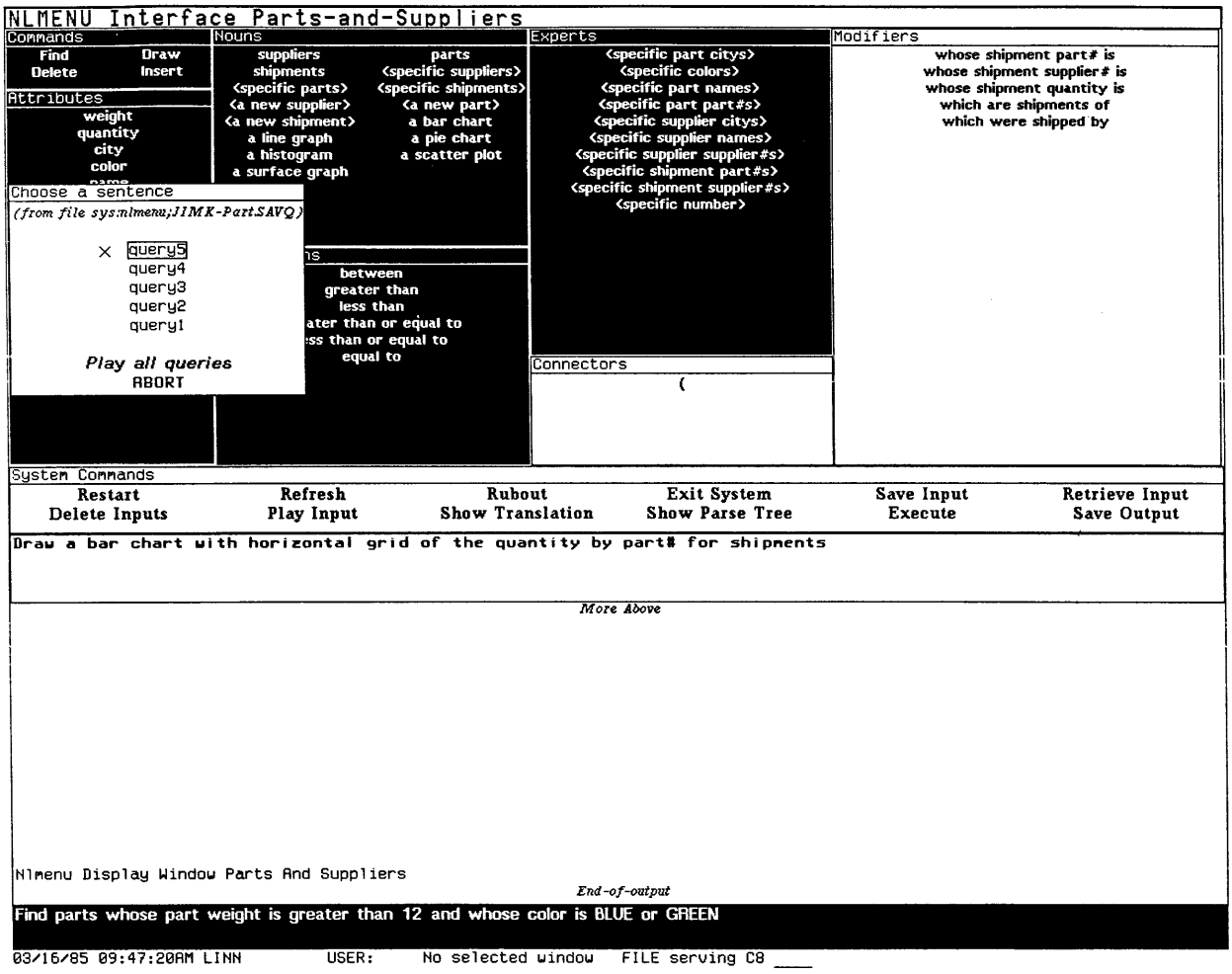
Figure 2-21 Delete Inputs Command Pop-Up Window



The items in this menu are mouse-sensitive. Position the mouse cursor over the name of a saved sentence to display that sentence on the who-line. Position the mouse cursor and click left once on the mouse to select an input for deletion. To terminate this option without making a deletion, select the Abort option from the menu.

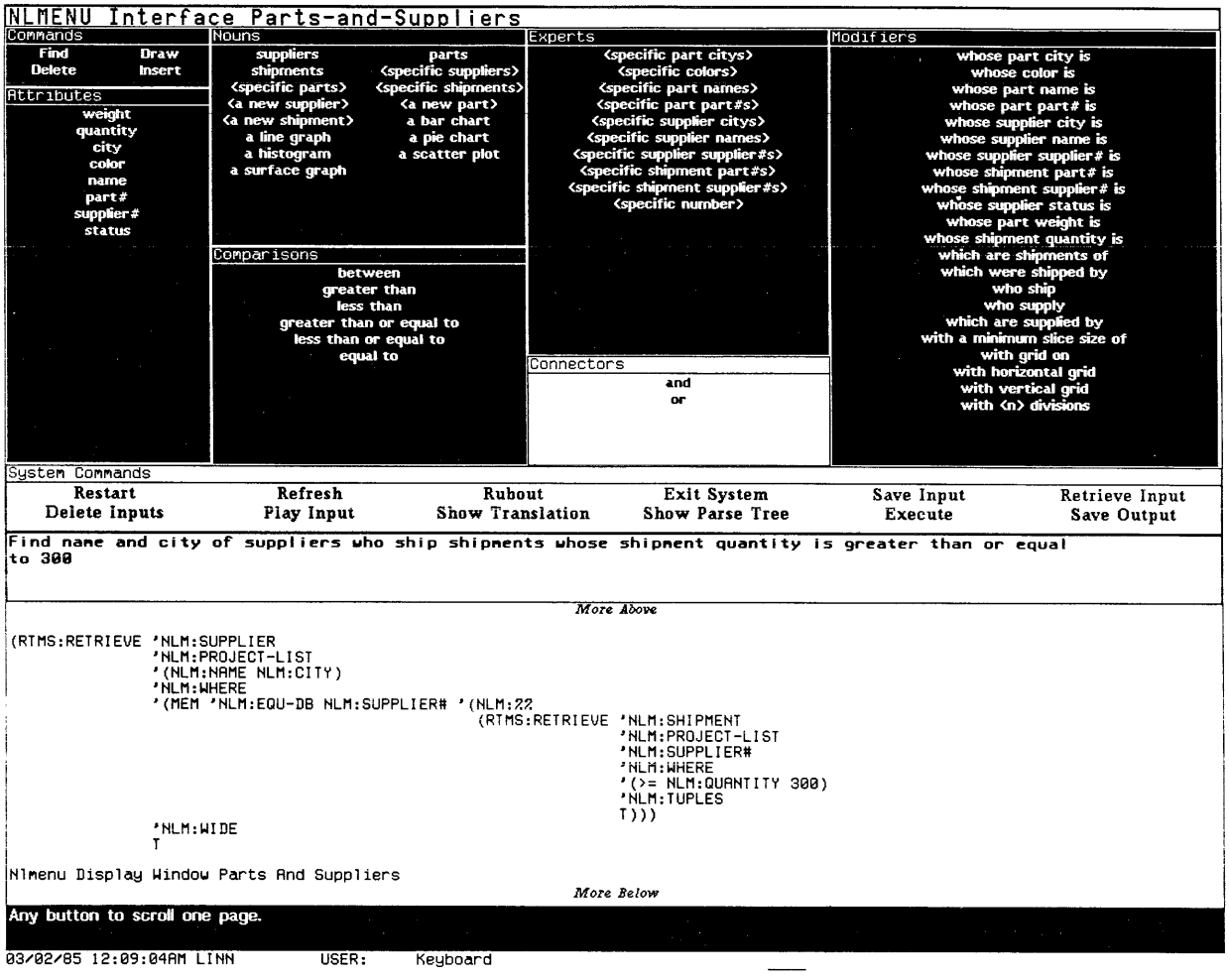
Play Input Command **2.2.4.8** The Play Input command enables you to retrieve an input and play it (that is, watch the NLMenu system automatically go through the selection process) in slow motion. Position the mouse over the name of a saved sentence to display that sentence on the who-line. In a pop-up window (represented in Figure 2-22) containing a mouse-sensitive list of saved inputs plus the Abort option, you position the mouse cursor and click left once on the mouse to make a selection. Or, if you want to play all saved queries, select the Play All Queries option.

Figure 2-22 Play Input Command Pop-Up Window



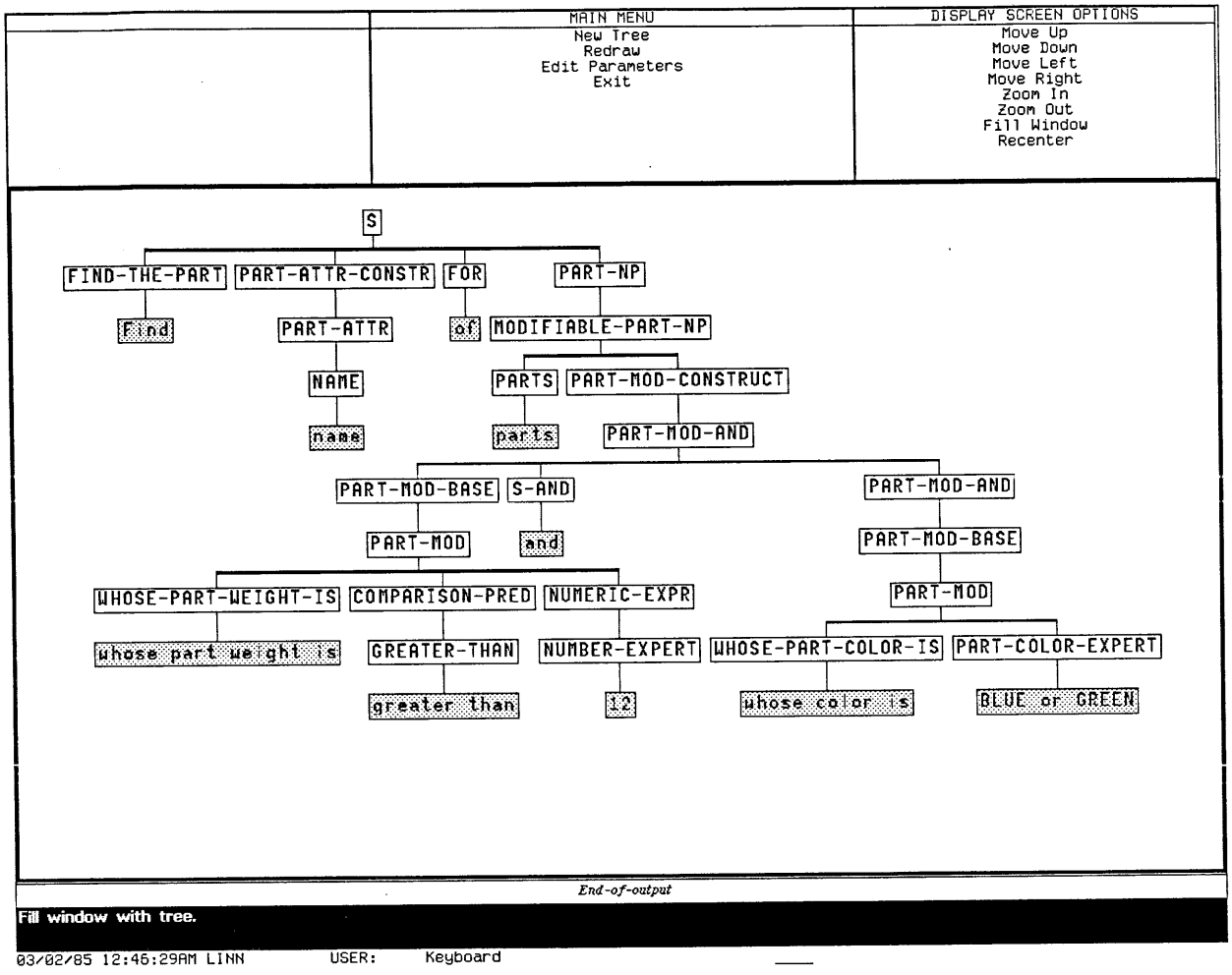
Show Translation Command **2.2.4.9** The Show Translation command displays the target language translation of an input sentence. When you select this command, the translation appears in the display window. The Show Translation command also indicates the number of parses. This command asks you to choose a particular parse when the parser encounters ambiguity. The screen appears approximately as follows (see Figure 2-23) when you select this command.

Figure 2-23 Show Translation Command Output



Show Parse Tree Command **2.2.4.10** Choosing the Show Parse Tree command displays the parse of an input sentence in tree form in a tree editor window. The screen appears approximately as follows (see Figure 2-24) when you select the Fill Window command.

Figure 2-24 Show Parse Tree Command Output



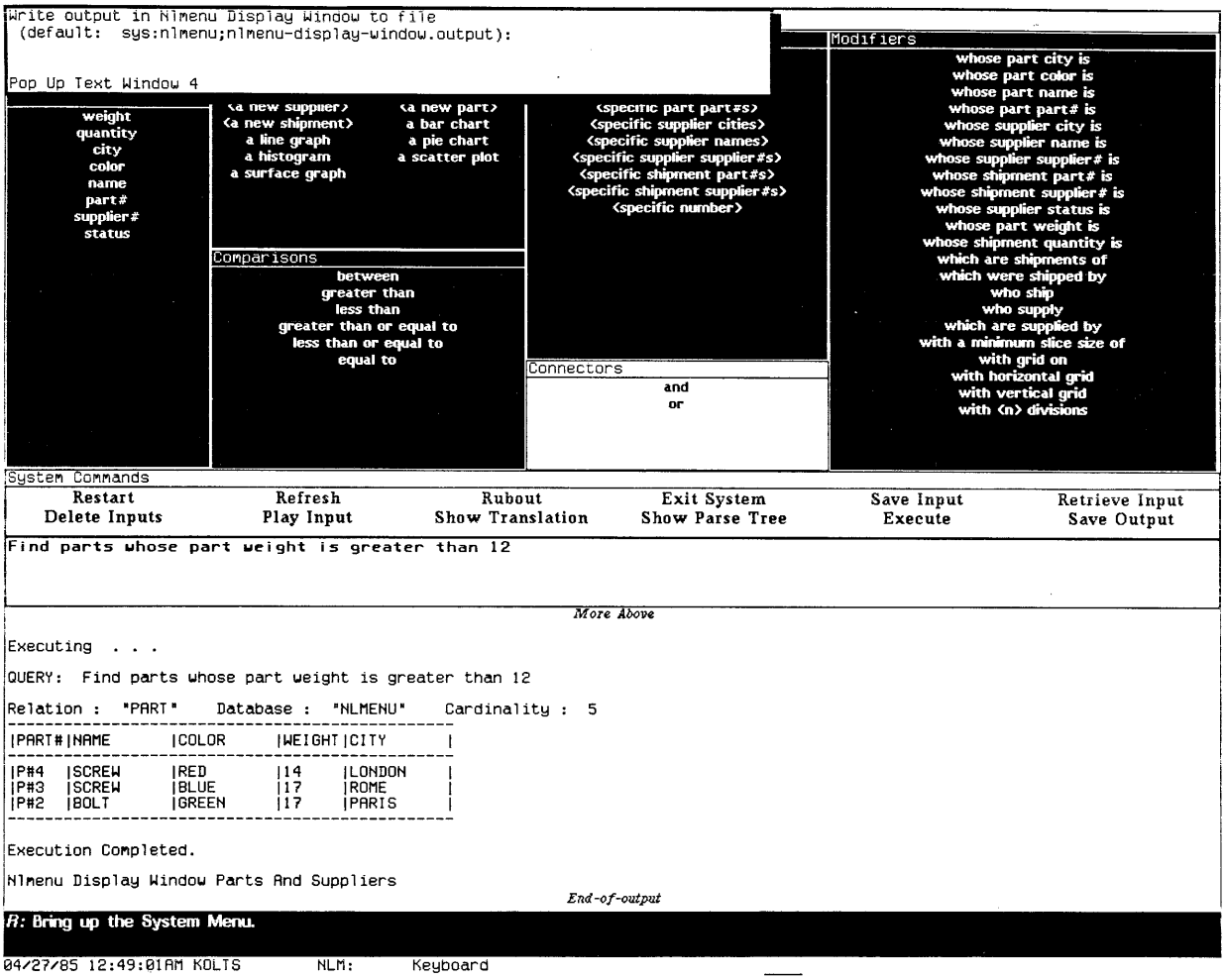
You can execute standard tree display commands (other than the New Tree command). The who-line provides some documentation to guide you through the selection process but not enough for the novice. For more detailed information on tree display commands and how to execute them, see Part 3 of the *Explorer Graphics Toolkit User's Guide*.

Clicking on Exit returns you to the NLMenu window. The Show Parse Tree command also lists ambiguous parses and asks you to choose a particular parse.

Execute Command **2.2.4.11** Selecting the Execute command causes evaluation of the currently completed input. Any output is displayed in the output window.

Save Output Command **2.2.4.12** The Save Output command saves the contents of the display window to a file. Everything in the display window, not just the results of query execution is saved; this includes items not currently visible. A pop-up window (see Figure 2-25) appears in which you designate the name under which the system is to save the output. The system automatically creates the file and gives it the appropriate extension. A message appears in the Display window when this operation is completed.

Figure 2-25 Save Output Command Pop-Up Window



Exiting an NLMenu Interface

2.2.5 There are two ways to exit an interface and return to the Lisp Listener:

- Use the mouse to select the Exit System option from the System Commands window of the interface.
- Press the END key, which is equivalent to selecting the Exit System option with the mouse.

To return to an NLMenu window from the Lisp Listener, select the NLMenu option from the System menu.

Interface Development

2.3 Before examining the various NLMenu utilities that are used in developing a natural language interface and the detailed steps involved in developing each interface component, it is appropriate to provide a framework for these later discussions by briefly listing the steps involved in the development of an interface. A typical interface development procedure is as follows (assuming the development does not merely involve the automatic generation of an SQL or RTMS database interface as described in Section 7):

1. Create a set of English sentences. Study the application and determine the sentences that best represent the capabilities of the application. Refer to Section 3 for guidelines.

2. Write a context-free grammar that defines the set of English sentences created in step 1. Refer to paragraph 3.2 for detailed examples of writing a grammar for the NLMenu software.

3. Develop grammar translation information. Such translation information becomes part of the lexicon. Using the formats in Section 4, develop the translation information for the grammar created in step 2. Keep a list of the terminals that occur in your grammar. With each element in this list, include the appropriate translation. You will use this list during the lexicon building procedure. Refer to the sample lexicon in Section 4 for examples of an SQL translation scheme.

4. Use the Zmacs editor to create and save a grammar file. Refer to the sample grammar in Section 4 for grammar file format and to Section 3 for grammar-writing guidance.

5. Call the grammar tests that examine the grammar for format errors and test it for cycles. See Section 5 for more information about these tests.

6. Build the lexicon. Enter your own lexicon file using Zmacs, or call the lexicographer. The lexicographer assists you by identifying the terminals of the grammar and prompting you for appropriate translation information. Refer to Section 6 for details.

7. Develop and test the screen configuration definitions. For some applications, the default screen description (see paragraph 7.2.4) may be appropriate. You can also delay final screen configuration definition until after you prototype an interface by specifying **nil** for the panes and constraints description when an interface is invoked (see paragraph 2.2.1.2). If you do not accept the default screen description, you can use the appropriate options of the lexicographer or develop the screen configuration definition from scratch and include it in a file. Refer to Section 6 for guidance in using the lexicographer or to paragraph 4.8 for information concerning developing the screen from scratch.

8. Test the grammar. You can access the grammar tool interface either during interface invocation by specifying `yes` for the `Test the grammar?` option (see paragraph 2.2.1.2) or by calling the function **nlm:get-grammar-tools**. Refer to Section 5 for details concerning the various grammar tests.

9. The NLMenu system builds the interface automatically from parameters you specify (see paragraph 2.2.1.2). Use the Build New Interface option on the NLMenu System menu to add and load the new interface.

10. Test the translations. Attempt to construct queries that the interface should handle. Construct various random queries to determine if the interface is accepting some queries that were not intended to be handled.

11. Revise the grammar and lexicon based on the results obtained from step 10.

Starter Kits

2.4 A starter kit containing a grammar, lexicon, screen description, and set of experts is included in the core NLMenu software to permit easy familiarization with the system. You can easily construct a sample interface using these default components (see paragraph 2.2.1.2). This sample interface is an application that allows you to formulate queries in English to run against a database. The NLMenu system then translates these inputs into a database retrieval language, in this case, Structured Query Language. (Software to execute SQL queries is not included.) If you have the Relational Table Management System, a second starter kit is also available. In addition to appropriate grammars, lexicons, screen descriptions, and experts, this kit contains provisions for generating a sample Explorer RTMS database. The RTMS starter kit also contains NLMenu interfaces to the sample database that were automatically generated with tools in the NLMenu-RTMS-Interface toolkit. The RTMS query language is the target language of these interfaces. The RTMS starter kit also includes methods for query execution. The two starter kits are discussed more fully in the following paragraphs.

Core NLMenu Starter Kit

2.4.1 The core NLMenu starter kit includes an interface to the sample Parts-and-Suppliers database. The sample database is reproduced in Figure 2-26.

Figure 2-26 Parts-and-Suppliers Database*

S

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	Pname	Color	Weight	City
P1	nut	red	12	London
P2	bolt	green	17	Paris
P3	screw	blue	17	Rome
P4	screw	red	14	London
P5	cam	blue	12	Paris
P6	cog	red	19	London

SP

S#	P#	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Figure 2-26 shows three tables or relations. The supplier relation, S, lists a unique supplier number (S#) for each supplier and also the supplier name, status, and city. The part relation, P, lists a unique part number (P#) for each part and also the part name, color, weight, and the city where the part is located. The shipment relation, SP, describes the quantity of parts shipped by suppliers.

* C.J. Date, *INTRODUCTION TO DATABASE SYSTEMS*, © 1981, Addison-Wesley, Reading, Massachusetts. Pg 92, Fig. 4.7. Reprinted with permission.

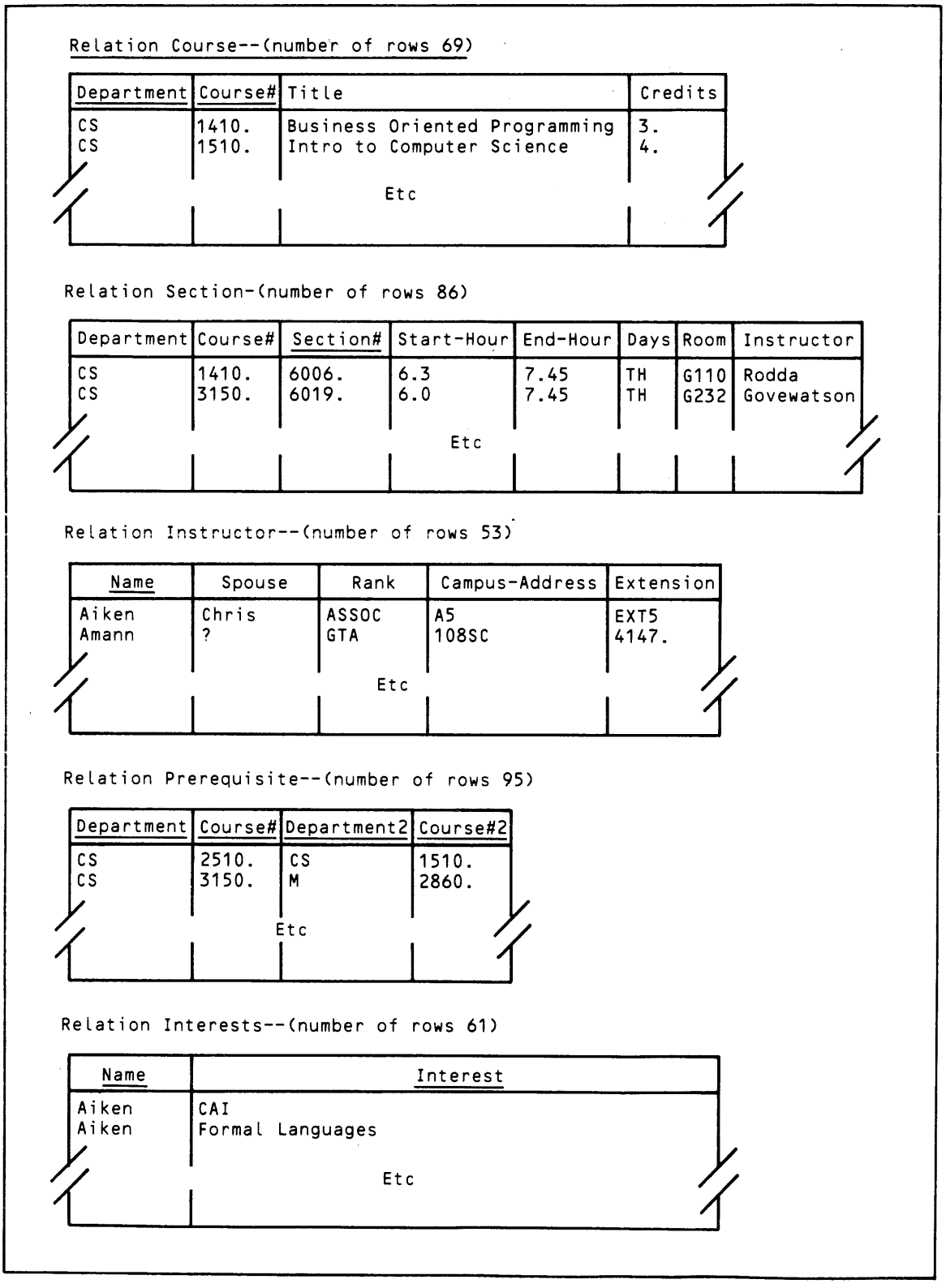
After loading this interface, all that you do to query the database is formulate a query in English and find the proper paraphrase by making appropriate selections from the selection menus (for additional guidance, refer to paragraph 2.2.1.2). For instance, to get a list of all the parts and the location of the warehouse where they are stored, you can quickly determine that the proper paraphrase is *Find city and name of parts.* If you cannot find an appropriate paraphrase, you can generally conclude that the query is outside the conceptual range of the database.

You can find the grammar used in the implementation of this interface in the file SYS:NLMENU;PARTS-AND-SUPPLIERS.GRAMMAR, while the corresponding lexicon is in the file SYS:NLMENU;PARTS-AND-SUPPLIERS.LEXICON. To fully describe the screen, the NLMenu system binds the appropriate descriptions to the variables **nlm:*nl-default-panes*** and **nlm:*nl-default-constraints***. These bindings appear in the file SYS:NLMENU;NLMENU.LISP.

NLMenu-RTMS-Interface Starter Kit

2.4.2 The NLMenu-RTMS-Interface starter kit includes a university-courses database, a baseball-statistics database, and the Parts-and-Suppliers database, as well as appropriate grammars, lexicons, screen descriptions, and experts to allow the generation of interfaces to these databases. A portion of the courses database appears in Figure 2-27, and Figure 2-28 shows a portion of the baseball database.

Figure 2-27 Courses Database



Relation Course--(number of rows 69)

Department	Course#	Title	Credits
CS	1410.	Business Oriented Programming	3.
CS	1510.	Intro to Computer Science	4.
		Etc	

Relation Section-(number of rows 86)

Department	Course#	Section#	Start-Hour	End-Hour	Days	Room	Instructor
CS	1410.	6006.	6.3	7.45	TH	G110	Rodda
CS	3150.	6019.	6.0	7.45	TH	G232	Govewatson
			Etc				

Relation Instructor--(number of rows 53)

Name	Spouse	Rank	Campus-Address	Extension
Aiken	Chris	ASSOC	A5	EXT5
Amann	?	GTA	108SC	4147.
		Etc		

Relation Prerequisite--(number of rows 95)

Department	Course#	Department2	Course#2
CS	2510.	CS	1510.
CS	3150.	M	2860.
		Etc	

Relation Interests--(number of rows 61)

Name	Interest
Aiken	CAI
Aiken	Formal Languages
	Etc

Figure 2-28 **Baseball Database**

```
SAMPLE DATA FOR THE BASEBALL STATISTICS DATABASE

Relation Team--(number of rows 26)
|Name                : Atlanta
|League              : N (N = National, A = American)
 Average             : 0.256
 Games_Played        : 162.
 At_Bats             : 5507.
 Runs                : 739.
 Hits                : 1411.
 Doubles             : 215.
 Triples             : 22.
 Homeruns            : 146.
 Steals              : 151.
 Shut_Out_By_Others: 10.
 ERA                 : 3.82
 Complete_Games      : 15.
 Innings_Pitched     : 1463.
 Hits_Given_Up       : 1484.
 Runs_Given_Up       : 702.
 Walks               : 502.
 Strikeouts          : 813.
 Shutouts            : 11.
 Saves               : 51.
 .............etc.....................
                .
                .
                .

Relation Pitcher--(number of rows 389)
|Name                : Boitano
|League              : A
|Team                : Texas
 Wins                : 0.
 Losses              : 0.
 ERA                 : 5.34
 Games_Pitched       : 19.
 Complete_Games      : 0.
 Innings_Pitched     : 30.
 Hits_Given_Up       : 33.
 Walks               : 13.
 Strikeouts          : 28.
 Shutouts            : 0.
 Saves               : 0.
 .............etc.....................
                .
                .
                .

Relation Batter--(number of rows 40)
|Name                : Ashby
|League              : N
|Team                : Houston
 Average             : 0.257
 Games_Played        : 100.
 At_Bats             : 339.
 Runs                : 40.
 Hits                : 87.
 Doubles             : 14.
 Triples             : 2.
 Homeruns            : 12.
 RBIS                : 49.
 Steals              : 2.
 .............etc.....................
                .
                .
                .
```

The courses database contains the following relations:

- A course relation lists the department, title, credit hours, and unique course number for each course.
- A section relation lists the department, course number, starting hour, ending hour, days, room, instructor, and unique section number for each section.
- An instructor relation lists for each instructor a unique name, that instructor's spouse, rank, campus address, and telephone extension.
- A prerequisite relation lists for various department-course number pairs other department-course number pairs that are prerequisites for the first pair.
- An interests relation lists instructor-special interest pairs.

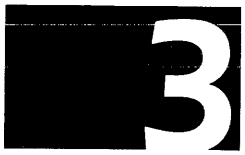
The baseball database includes the following relations:

- A team relation lists for each team its league, unique name, and also various team statistics, such as total hits.
- A batter relation lists for each hitter his unique name, team, and league, and also various individual statistics, such as home runs.
- A pitcher relation lists for each pitcher his unique name, team, league, and also various individual statistics, such as games won.

To use this starter kit, you first generate an RTMS database from the sample data. This procedure is fully described in paragraph 7.2.3. You then need to insert entries for each NLMenu-RTMS interface into the database. The necessary interface management techniques are discussed in paragraph 7.2.2. The grammars, lexicons, and spec files used to generate the grammars and lexicons (see paragraph 7.2.1), and sample data items are contained in the following files:

- SYS:NLMENU-RTMS-INTERFACE;COURSES.SPEC
- SYS:NLMENU-RTMS-INTERFACE;COURSES-RTMS.GRAMMAR
- SYS:NLMENU-RTMS-INTERFACE;COURSES-RTMS.LEXICON
- SYS:NLMENU-RTMS-INTERFACE;BASEBALL.SPEC
- SYS:NLMENU-RTMS-INTERFACE;BASEBALL-RTMS.GRAMMAR
- SYS:NLMENU-RTMS-INTERFACE;BASEBALL-RTMS.LEXICON
- SYS:NLMENU-RTMS-INTERFACE;S-P.SPEC
- SYS:NLMENU-RTMS-INTERFACE;S-P-RTMS.GRAMMAR
- SYS:NLMENU-RTMS-INTERFACE;S-P-RTMS.LEXICON
- SYS:NLMENU-RTMS-INTERFACE;SAMPLE-DATABASE.LISP

CREATING NATURAL LANGUAGE MENU SENTENCES



Highlights of This Section

- Grammar Writing
  - Introduction (terminology and underlying principles)
  - Formalism of NLMenu grammar
  - Sentence generation
  - Abbreviatory conventions in grammar writing
    - Parentheses convention
    - Braces convention
    - Square brackets convention
    - Improper use of abbreviatory elements
  - How to write a grammar
    - Simple sentences
    - Sentences with complex noun phrases
    - Sentences with recursion
    - Ambiguous sentences
    - Sentences with relative clauses
- Summary of grammar writing hints

Introduction

3.1 This section shows you how to create a set of Natural Language Menu (NLMenu) sentences. You base your NLMenu grammar on the sentences the interface user can create. Before you begin to create sentences, you must become familiar with the concept of grammar writing. This section begins with a review of linguistic terminology (see the Glossary as well), an explanation of context-free grammars, and several examples of grammar writing for a number of different applications.

Grammar Writing

3.2 The following paragraphs attempt to demystify the process of writing grammars for your application interface. The topics covered include:

- Terminology
- NLMenu grammar formalism (what grammar rules look like)
- Sentence generation (how grammar rules are used)
- Abbreviatory conventions (how to optimize grammar rules by efficiently combining them)
- Example grammars (progressing from the simple to the complex)

Terminology

3.2.1 A *language* is defined as a (possibly infinite) set of strings (sentences) formed from a vocabulary of symbols. In a natural language such as English, the symbols are *words*. A *sentence* is defined as a sequence of one or more words in a language, but not all sequences of words are permissible in a language. For example, the sequence *the why house come or* is not an English sentence because English speakers do not understand the meaning of this sequence of words. Therefore, you need a *grammar* that states explicitly, by means of rules, how words are put together to form the legal sentences of a language.

Grammars have a *syntactic* and a *semantic* component.

- The syntactic component, or *syntax*, defines the appearance of legal sentences in the language. The syntax is a set of construction rules, with words or categories of words serving as the building blocks.
- The semantic component of the grammar determines the meaning of legal sentences. At the center of this second component is the *lexicon*, a collection of words in the language.

A grammar must account for the language structure by means of some finite set of syntactic and semantic (lexical) rules. From the grammar, it should be possible to generate any sentence from the set of legal sentences. In practice, writing a grammar for the complete set of legal sentences in any natural language is a monumental undertaking. The task is

much more manageable, however, when the goal is to generate some subset of the set of legal sentences. It then becomes possible to limit the syntactic and semantic structures for which the grammar must account. When you use the NLMenu system to design application interfaces, you will need to describe some such subset. The complexity of your grammars depends on the total number of sentences for which you must account and the extent to which you allow variation in the way such sentences are expressed. Note here that a grammar describing one language cannot describe other languages because each language has its own way of combining words to form sentences.

Grammar writing is better taught through the presentation and description of examples than by means of theoretical discussion. You also need to do some experimenting on your own. Before studying the starter-kit examples (in Section 4), familiarize yourself with the formalism (symbolic language) in which grammar rules are written.

Formalism of NLMenu Grammar

3.2.2 The following set of grammar rules describe a particular language, where the symbols *a*, *b*, and *c* represent words of the language as follows:

1 a. S $\rightarrow$ A b
  b. A $\rightarrow$ a c

NOTE: You observe here the convention that is followed throughout this section when discussing some arbitrary set of grammar rules or sentences. The sets are numbered consecutively throughout the section, and the individual rules or sentences within each set are referenced by an alphabetic character.

This grammar is in *context-free grammar* notation, which you will use throughout the following paragraphs. Context-free or Backus-Normal-Form (BNF) grammars describe the structure of sentences by means of a finite set of rules called *productions*. The format shown in the preceding example is a common grammar writing convention. Each rule in the grammar has a single symbol on the left-hand side connected by an arrow to one or more symbols on the right-hand side. The symbol *S* stands for *sentence*, and the arrow, also known as the *production symbol*, is equivalent to the English phrase *can consist of* or *can be*. The preceding rules read as follows:

A sentence *S* can consist of *A* followed by *b*, and *A* can consist of *a* followed by *c*.

Each symbol in a rule is called a *rule element*. Thus, the symbols *S*, *A*, *b*, *a*, and *c* are rule elements. There are two categories of symbols:

- Terminal symbols — Symbols of the language alphabet (words of the language). These cannot produce other symbols and appear in BNF productions only on the right-hand side. In this manual, they appear in lowercase.

- Nonterminal symbols — Intermediate symbols that can be expanded into something else. In this manual, they are in uppercase. They can appear on the right-hand side of productions or, singly, on the left-hand side.

According to the definition, symbols *a*, *b*, and *c* are terminal symbols since they do not appear on the left-hand side of any rule, while the symbols *A* and *S* are nonterminal symbols. In addition, there is a special kind of nonterminal symbol called the *start* symbol, which is the *root* (origin) of all sentences in the language. In this example, *S* is the start symbol.

To relate this discussion to your experience with natural languages, consider the following subset grammar of English:

SENTENCE → NOUN-PHRASE VERB-PHRASE
NOUN-PHRASE → the NOUN
NOUN → system
VERB-PHRASE → crashes

In this subset the start symbol is *SENTENCE*; the nonterminal symbols are *SENTENCE*, *NOUN-PHRASE*, *VERB-PHRASE*, and *NOUN*; and the terminal symbols are *the*, *system*, and *crashes*.

The NLMenu lexicon is a sort of dictionary that contains entries in one language, often referred to as the *source language*, and translations for those entries in a second language, commonly called the *target language*. Each word or phrase in the lexicon has precisely one translation, and that translation is used regardless of the context in which the word or phrase might actually appear in some sentence.

Rules for combining translations to form sentences must also be provided. These rules, part of the semantic component of the NLMenu grammar, indicate the order in which the translations of the symbols on the right-hand side of the arrow of a context-free rule are to be combined. The NLMenu system takes the approach that the meaning of a sentence is a function of the meaning of its parts and the manner in which they are combined.

Sentence Generation

3.2.3 The following discussion concerns how to generate sentences from the following set of grammar rules for a language:

2 a. S → A B C
  b. A → a C
  c. B → b
  d. C → c

The following explanation of Derivation 1 (illustrated below) demonstrates the generation of sentences from this set of rules:

1. Write the symbol *S*.

2. Below that, write the sequence of symbols that appears on the right-hand side of the *S* rule 2a. If the grammar had more than one rule with *S* on the left-hand side, you could pick any *S* rule.

3. Form a third line by replacing one of the nonterminal symbols in the second line with the sequence of symbols that appears on the right-hand side of the replaced symbol in one of the rules. To form the third line *a C B C* in the following derivation, replace the symbol *A* with the sequence of symbols *a C* that appears on the right-hand side of the *A* rule 2b.

4. Repeat this procedure until there are no more rules to apply, or in other words, until all of the symbols are terminal symbols.

Derivation 1

S	
A B C	if you apply rule 2a
a C B C	if you apply rule 2b
a C b C	if you apply rule 2c
a c b c	if you apply rule 2d twice

When all symbols in a sentence are terminal, that sentence has been *generated* by the grammar. Frequently, you can rewrite more than one symbol in a given line to create the next line. In the preceding example, there are three nonterminal symbols *A*, *B*, and *C* on the second line. Which symbol you choose to expand in such situations has no effect on the ultimate outcome. Compare the following derivation with Derivation 1.

Derivation 2

S	
A B C	if you apply rule 2a
A b C	if you apply rule 2c
A b c	if you apply rule 2d
a C b c	if you apply rule 2b
a c b c	if you apply rule 2d

If there is more than one rule in the grammar that can apply to a particular symbol, the ultimate outcome depends upon which rule you choose. For example, if you add a rule to the grammar, it is possible to generate either sentence shown in the following two derivations.

3 a. S → A B C
  b. A → a C
  c. B → b
  d. C → c
  e. A → a B

Derivation 1

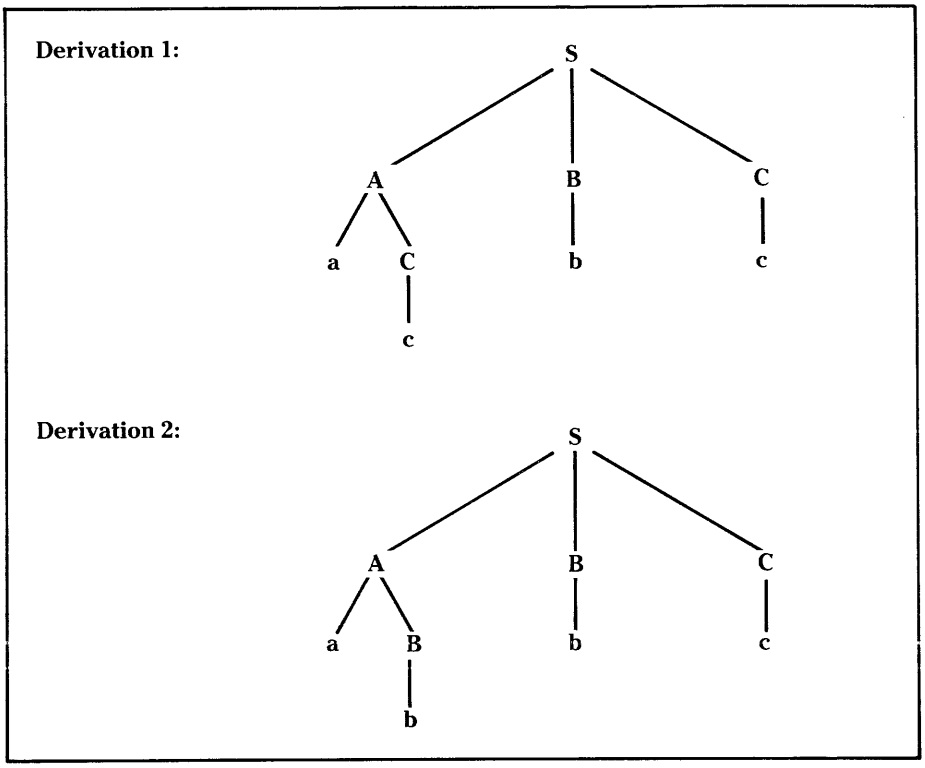
S	
A B C	if you apply rule 3a
a C B C	if you apply rule 3b ←
a C b C	if you apply rule 3c
a c b c	if you apply rule 3d twice

Derivation 2

S	
A B C	if you apply rule 3a
a B B C	if you apply rule 3e ←
a b b C	if you apply rule 3c
a b b c	if you apply rule 3d twice

A tree structure best represents sentence generation. This structure symbolizes the *parse* of the sentence. Figure 3-1 illustrates the generation of the sentences from the preceding derivations. In the natural language example developed in Section 4, you will see how a parse tree captures both the underlying syntactic and semantic structure of a sentence.

Figure 3-1 **Sentence Derivation Trees**



The tree develops by starting with the start symbol *S* as the root. Below that write the sequence of symbols that appears on the right-hand side of some rule that has *S* as its root and draw branches back to the root. Continue in this way to draw branches from every symbol that can expand. A tree is complete when each branch ends in a terminal symbol. The string formed by the terminal symbols is the generated sentence, for example, the generated sentence for Derivation 2 in Figure 3-1 is *a b b c*.

Abbreviatory Conventions in Grammar Writing

3.2.4 These paragraphs contain three abbreviatory conventions that grammar writers use for collapsing (combining) partially similar rules and a brief theoretical justification for those conventions. The three conventions are:

- Parentheses convention
- Braces convention
- Square brackets convention

The NLMenu system supports all three abbreviatory conventions.

Parentheses Convention

3.2.4.1 If two grammar rules are identical except that one contains a symbol or a sequence of symbols that the other does not, you can collapse the two rules into one by writing out the longer rule and enclosing in parentheses the symbol or the sequence of symbols not contained in the shorter rule. For example, consider the following pair of rules:

```
4 a.  A →C D B
  b.  A →C B
```

The first rule differs from the second in that it contains one additional symbol. Hence, the condition for applying the parentheses convention is met. Collapse these two rules into one by putting the additional element in parentheses:

```
4 c.  A → C (D) B
```

This rule reads as follows:

The nonterminal symbol *A* can consist of *C* followed by an optional element *D* followed by *B*.

Braces Convention

3.2.4.2 The braces convention uses braces to indicate either/or options in the selection of elements when there is a rule that can expand in more than one way. The following examples illustrate this:

```
5 a.  S →A B
  b.  S →A C
  c.  S →A D
```

The preceding rules say that a sentence can consist of *A* followed by *B* or *C* or *D*. You can collapse these three rules into one by using the braces convention:

```
5 d.  S → A { B C D }
```

Square Brackets Convention

3.2.4.3 The square brackets convention defines groups and can be used efficiently inside braces. Suppose a grammar has the following two rules:

```
6 a.  A →B C D
  b.  A →B E F
```

Collapse these two rules into one using braces:

```
6 c.  A → B { C D  E F }
```

To show that rule elements *C* and *D* constitute one group and that *E* and *F* constitute another group, you must use square brackets:

```
6 d.  A → B {  [C D]  [E F]  }
```

As a further example, you can rewrite a rule such as

S → A { (B) C E}

as 6e below using the square brackets because the sequence *(B) C* constitutes one group:

6 e. S → A { [(B) C] E }

These abbreviatory conventions are more than just a convenient shorthand for stating rules. They express an important generalization that would otherwise be overlooked. However, you must be very careful when applying them. Suppose the grammar has the following two rules:

6 f. S →A B C D
g. S →A E C F

The preceding rules say that either *B* or *E* follows *A*, and either *D* or *F* follows *C*. At first glance, it appears that braces can be used here to form the following rule:

6 h. S → A { B E } C { D F }

If you do this, however, when you expand *S* again, you not only generate the two original rules, but also rules 6i and 6j which are not in the grammar:

6 f. S →A B C D
g. S →A E C F
i. S →A B C F
j. S →A E C D

This is an undesirable result because the grammar should generate all and only those sentences in the set of valid sentences in the language. The correct way to combine rules 6f and 6g is:

6 k. S → A { [B C D] [E C F] }

Improper Use of Abbreviatory Elements

3.2.4.4 You can nest abbreviatory elements, but be careful when doing so. Some constructs may result in meaningless rule expansions or inefficient use of elements. Some examples of this are:

- Inefficient use of elements — You can simplify the rule

  A → B ((C) (D) (E))

  with no loss of meaning to

  A → B (C) (D) (E)

- Meaningless expansions — The braces mean that you must choose one of the enclosed items. If any choices are in parentheses, such as in a rule of the form

  A → B {(C) ... X}

  then one of the possible rule expansions has no elements within the braces chosen. This will not pass the syntax check. Any time that the grammar writer wishes to include the possibility of choosing no element, parentheses should be the outermost symbol. In this case, rewriting the rule as follows

  A → B ({C ... X})

  or as the following group of rules

  A → B (Z)
  Z → C
  ...
  Z → X

  would be appropriate.

The grammar syntax checking utility flags these incorrect constructs as errors.

How to Write a Grammar

3.2.5 These paragraphs show you how to write a grammar that generates all the grammatical sentences and only the grammatical sentences for a particular subset of a language. In this manual this set of sentences is referred to as all and only the grammatical sentences.

Simple Sentences

3.2.5.1 Assume that you have the following set of sentences defined for a database interface language and that you are writing a grammar for them:

7 a. Find workers.
  b. Find jobcards.
  c. Find orders.
  d. Find operations.

The informal description of these sentences is:

A sentence has as its first element the word *find*. This can be followed by the words *workers*, *jobcards*, *orders*, or *operations*.

The following context-free grammar (Grammar 1) suggests itself from this informal description:

Grammar 1

S → find workers
S → find jobcards
S → find orders
S → find operations

The sentence symbol *S* is the only nonterminal symbol in Grammar 1. The rest of the words are terminal symbols. Grammar 1 works fine if you have only four sentences to describe. But consider having thousands of different sentences defined for the interface language. You do not want thousands of *S* rules in the grammar that simply indicate a particular sentence consists of a specific sequence of individual words. Although Grammar 1 looks like a grammar, it is nothing more than a list of sentences. A grammar should reveal the patterns or regularities of sentences in the language. Look at sentence set 7 again and see whether you can find some regularities. All four words, *workers*, *jobcards*, *orders*, and *operations*, appear after the word *find*. You can write the following intermediate grammar rule to represent this fact:

S → find X

At appropriate points during this presentation on grammar writing, useful concepts will be highlighted as follows as a *Grammar Writing Hint*. A collection of these hints is provided at the conclusion of this section.

Grammar Writing Hint: Do not merely list valid sentences; use meaningful categories to capture regularities.

The words *workers*, *jobcards*, *orders*, and *operations* belong to a category of words called nouns. For the present, use *NOUN* as the name of the category to which the words *workers*, *jobcards*, *orders*, and *operations* belong. When you select more meaningful category names, such as *NOUN*, your grammar better reflects the syntactic and/or semantic structure of the language. Linguists refer to such grammars as being more *revealing*. The concept is similar to that of using more meaningful identifier names when writing computer programs. As you follow the process of developing this example grammar, you will observe that revealing grammars have highly abstract *S* rules that are expanded to an ever greater degree of specificity as each element of the *S* rule(s) is defined. Using this category, *NOUN*, you can rewrite Grammar 1 as Grammar 2:

Grammar 2

S → find NOUN
NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations

This new grammar generates all and only those sentences in set 7. Unlike Grammar 1, Grammar 2 reveals a pattern in the sentences, namely, that all sentences in this language consist of a terminal element *find* followed by one of the words from the category *NOUN*. Note that categories are more than just convenient groupings of words. The real significance of categories lies in the fact that by defining certain categories, you can begin to construct a precise and revealing description of a language.

You can also assign a category name *VERB* to the word *find* and write a grammar (Grammar 3) in the following way:

Grammar 3

S → VERB NOUN
VERB → find
NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations

Although this grammar is more revealing than Grammar 2 (by indicating the underlying structure more abstractly), it is not necessary to introduce a new category name since *find* is the only verb known in the language so far. There is one more reason for not choosing Grammar 3. One of the principles in grammar writing is that if two grammars generate exactly the same sentences, then prefer the grammar with fewer rules. Grammar 2 is also better from a computational point of view. Having fewer rules means that the parser has fewer rules to consider at any given time, thus saving the parser time and space.

Grammar Writing Hint: Do not introduce unnecessary categories.

Sentences With Complex Noun Phrases

3.2.5.2 Grammar writing involves the following:

- Constrain the power of the grammar.
- Assign the proper lexical categories to groups of individual words to generate all and only the grammatical sentences.
- Use the notational conventions with the grammar rules to show generalizations that exist in the language.
- Assign the proper grammatical categories to the sequences of words to show the basic structure of the language.
- Evaluate the grammar in terms of power and efficiency.

Grammar Power The natural language interface to an application might contain the following two sets of sentences:

7 a. Find workers.
  b. Find jobcards.
  c. Find orders.
  d. Find operations.

8 a. Find the name of workers.
  b. Find the hours of jobcards.
  c. Find the price of orders.
  d. Find the descriptions of operations.

All the sentences in set 8 have as their first element the word *find*, followed by the word *the*, followed by the word *name*, *hours*, *price*, or *descriptions*, followed by *of*, followed by a word that belongs to the category *NOUN* as defined in Grammar 2 in the preceding paragraphs. An intermediate form of the grammar that represents this is:

S → find the X of NOUN

Since the words *name*, *hours*, *price*, and *descriptions* are also nouns, you can use the same category name for them. You can now write the following grammar (Grammar 4) to generate the sentences in set 8:

Grammar 4

S → find the NOUN of NOUN
NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations
NOUN → name
NOUN → hours
NOUN → price
NOUN → descriptions

You can now generate all of the following sentences:

Find the name of workers.
*Find the name of jobcards.
*Find the name of orders.
*Find the name of operations.

Find the hours of jobcards.
*Find the hours of workers.
*Find the hours of orders.
*Find the hours of operations.

Find the price of orders.
*Find the price of workers.
*Find the price of jobcards.
*Find the price of operations.

Find the descriptions of operations.
*Find the descriptions of workers.
*Find the descriptions of jobcards.
*Find the descriptions of orders.

*Find the workers of name.
*Find the workers of hours.
*Find the workers of price.
*Find the workers of descriptions.

*Find the orders of name.
*Find the orders of hours.
*Find the orders of price.
*Find the orders of descriptions.

*Find the jobcards of name.
*Find the jobcards of hours.
*Find the jobcards of price.
*Find the jobcards of descriptions.

*Find the operations of name.
*Find the operations of hours.
*Find the operations of price.
*Find the operations of descriptions.

Not only does Grammar 4 generate all the sentences in set 8, it also generates a long list of sentences (marked with an asterisk) that are not grammatical because they are not in the original list of sentences contained in set 8. A grammar should be powerful enough to generate all the well-formed sentences of a language but should be sufficiently restricted so that it does not generate any sentences that are not allowed in that language. Grammar 4 is too powerful. The problem is that the same category name is used for two groups that, even though they are both made up of nouns, function differently in these sentences. Therefore, you should assign different category names to these two groups of words.

Grammar Writing Hint: Do not make the grammar too powerful. Suggestions on limiting the power of grammars follow.

Assigning Lexical Categories Before you decide on category names, determine the characteristics of each group of words. The words *name*, *hours*, *price*, and *descriptions* in the phrases *the name of workers*, *the hours of jobcards*, *the price of orders*, and *the descriptions of operations* are the attributes of the nouns that follow them. So, you can use *ATTRIBUTE* as a new category name and rewrite Grammar 4 as Grammar 5:

Grammar 5

S → find the ATTRIBUTE of NOUN
NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations

ATTRIBUTE → name
ATTRIBUTE → hours
ATTRIBUTE → price
ATTRIBUTE → descriptions

Grammar 5 does not generate sentences like:

*Find the workers of (name, hours, price, descriptions).

*Find the orders of (name, hours, price, descriptions).

*Find the jobcards of (name, hours, price, descriptions).

*Find the operations of (name, hours, price, descriptions).

However, Grammar 5 still generates the following ungrammatical sentences:

*Find the name of (jobcards, orders, operations).

*Find the hours of (workers, orders, operations).

*Find the price of (workers, jobcards, operations).

*Find the descriptions of (workers, jobcards, orders).

Grammar 5 ignores the fact that each word has its own attributes. This becomes clearer when you look at the following additional sentences that are allowed in the interface language:

8 e. Find the employee number of workers.
  f. Find the birthdate of workers.
  g. Find the wage rate of workers.

  h. Find the date filled of orders.
  i. Find the date received of orders.
  j. Find the quantity of orders.

  k. Find the order number of jobcards.
  l. Find the job date of jobcards.
  m. Find the piece done of jobcards.

  n. Find the rejects of operations.
  o. Find the piece number of operations.
  p. Find the operation number of operations.

You can solve this problem by assigning different category names to the different attributes. This gives the following grammar (Grammar 6):

Grammar 6

S → find the WORKER-ATTRIBUTE of NOUN
S → find the JOBCARD-ATTRIBUTE of NOUN
S → find the ORDER-ATTRIBUTE of NOUN
S → find the OPERATION-ATTRIBUTE of NOUN

WORKER-ATTRIBUTE → name
JOBCARD-ATTRIBUTE → hours
ORDER-ATTRIBUTE → price
OPERATION-ATTRIBUTE → descriptions

NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations

Grammar 6 is still not restrictive enough and generates the following ungrammatical sentences:

*Find the name of (jobcards, orders, operations).

*Find the hours of (workers, orders, operations).

*Find the price of (workers, jobcards, operations).

*Find the descriptions of (workers, jobcards, orders).

To make Grammar 6 more restrictive, replace the category *NOUN* with a set of new categories, each containing an element from the old *NOUN* category. By using a different category for each noun, you finally get Grammar 7, which generates all and only the grammatical sentences shown in set 8.

Grammar Writing Hint: Incorporate semantic detail in the grammar if necessary to constrain its power and disambiguate it.

Grammar 7

S → find the WORKER-ATTRIBUTE of WORKER-NOUN
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN
S → find the ORDER-ATTRIBUTE of ORDER-NOUN
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

WORKER-ATTRIBUTE → name
JOBCARD-ATTRIBUTE → hours
ORDER-ATTRIBUTE → price
OPERATION-ATTRIBUTE → descriptions

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

If you want Grammar 7 to generate the sentences 8e through 8p, add the following rules to Grammar 7:

WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

NOTE: The hyphenated terminal elements in the preceding set of rules are compound elements. The hyphen indicates that, although composed of multiple elements, these terminals are functionally single units.

Now Grammar 2 generates the sentences in set 7, and Grammar 7 generates the sentences in set 8. But you do not want two different grammars generating two basically similar sets of sentences in the same language. You need to combine Grammar 2 and Grammar 7 so that the resulting grammar generates all and only those sentences in both sets—the simple ones in set 7 and the slightly more complex ones in set 8. Here are Grammars 2 and 7 again for you to compare:

Grammar 2

S → find NOUN
NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations

Grammar 7

S → find the WORKER-ATTRIBUTE of WORKER-NOUN
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN
S → find the ORDER-ATTRIBUTE of ORDER-NOUN
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

Grammar Writing Hint: Do not write one grammar for simple sentences and another for more complex sentences. Devise a grammar that generates both.

To make one grammar, you cannot just add Grammar 7 to Grammar 2 because a grammar should not have two different nonterminal symbols that expand to the same terminal symbols since such a situation often leads to ambiguity:

Grammar 2

S → find NOUN

NOUN → workers
NOUN → jobcards
NOUN → orders
NOUN → operations

Grammar 7

S → find the WORKER-ATTRIBUTE of WORKER-NOUN
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN
S → find the ORDER-ATTRIBUTE of ORDER-NOUN
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

The goal in writing a grammar is to exhibit the most revealing structure of the language as possible. Doing so greatly simplifies modifications and additions to the grammar. Also, the practice of allowing several non-terminals to expand to the same terminal increases the complexity of the lexicon, thus increasing the difficulty of specifying translations. Each non-terminal mapping to terminal *t* introduces a new constraint on *t*, and the set of constraints is not guaranteed to be consistent.

You can combine these two grammars in two ways:

- Replace the single *S* rule in Grammar 2 with four new *S* rules and add those four new rules to Grammar 7. In each of these new rules, you replace the category *NOUN* with one of the four categories, *WORKER-NOUN*, *JOBCARD-NOUN*, *ORDER-NOUN*, and *OPERATION-NOUN*. The four *NOUN* rules of Grammar 2 are not added to Grammar 7 to form Grammar 8.

- Keep the single *S* rule of Grammar 2 but change the *NOUN* rules as shown in Grammar 9.

Grammar 8

S → find WORKER-NOUN
S → find JOBCARD-NOUN
S → find ORDER-NOUN
S → find OPERATION-NOUN

S → find the WORKER-ATTRIBUTE of WORKER-NOUN
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN
S → find the ORDER-ATTRIBUTE of ORDER-NOUN
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

Grammar 9

S → find NOUN
S → find the WORKER-ATTRIBUTE of WORKER-NOUN
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN
S → find the ORDER-ATTRIBUTE of ORDER-NOUN
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

NOUN → WORKER-NOUN
NOUN → JOBCARD-NOUN
NOUN → ORDER-NOUN
NOUN → OPERATION-NOUN

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

Although Grammar 9 has one more rule than Grammar 8, it is still preferable because Grammar 9 has only five *S* rules while Grammar 8 has eight.

Grammar Writing Hint: Make the *S* rule(s) as simple as possible.

Capturing Generalizations Now determine whether one grammar is better than the other in expressing generalizations of the language. Grammar 8 has as many *S* rules as there are sentences. All that you have done so far by restricting the grammar is assign the proper categories to the words in the sentence:

7 a. Find workers.
S → find WORKER-NOUN

b. Find jobcards.
S → find JOBCARD-NOUN

c. Find orders.
S → find ORDER-NOUN

d. Find operations.
S → find OPERATION-NOUN

8 a. Find the name of workers.
S → find the WORKER-ATTRIBUTE of WORKER-NOUN

b. Find the hours of jobcards.
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN

c. Find the price of orders.
S → find the ORDER-ATTRIBUTE of ORDER-NOUN

d. Find the descriptions of operations.
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

.
.
.

Grammar Writing Hint: When evaluating alternative grammars, consider which one best captures the regularities of the language.

Assigning Grammatical Categories Grammar 8 is not a good grammar because it does not show regularities or generalizations that exist in the language. Use the abbreviatory conventions discussed in previous paragraphs to collapse some of the rules in Grammar 8. The following sets of *S* rules can be collapsed into one rule by using the parentheses convention:

(A) S → find WORKER-NOUN
S → find the WORKER-ATTRIBUTE of WORKER-NOUN

(B) S → find JOBCARD-NOUN
S → find the JOBCARD-ATTRIBUTE of JOBCARD-NOUN

(C) S → find ORDER-NOUN
S → find the ORDER-ATTRIBUTE of ORDER-NOUN

(D) S → find OPERATION-NOUN
S → find the OPERATION-ATTRIBUTE of OPERATION-NOUN

↓

(A)'	S →	find (the WORKER-ATTRIBUTE of)	WORKER-NOUN
(B)'	S →	find (the JOBCARD-ATTRIBUTE of)	JOBCARD-NOUN
(C)'	S →	find (the ORDER-ATTRIBUTE of)	ORDER-NOUN
(D)'	S →	find (the OPERATION-ATTRIBUTE of)	OPERATION-NOUN

The new *S* rules (A)' through (D)' show you that a sequence like (*the WORKER-ATTRIBUTE of*) *WORKER-NOUN* behaves as a unit. You can assign category names to these new units. Grammatically, these new units are noun phrases, abbreviated as NP:

WORKER-NP	→	(the WORKER-ATTRIBUTE of)	WORKER-NOUN
JOBCARD-NP	→	(the JOBCARD-ATTRIBUTE of)	JOBCARD-NOUN
ORDER-NP	→	(the ORDER-ATTRIBUTE of)	ORDER-NOUN
OPERATION-NP	→	(the OPERATION-ATTRIBUTE of)	OPERATION-NOUN

You can now rewrite Grammar 8 by rewriting the *S* rules to form Grammar 10:

Grammar 10

S → find WORKER-NP
S → find JOBCARD-NP
S → find ORDER-NP
S → find OPERATION-NP

```
WORKER-NP →
   (the WORKER-ATTRIBUTE of) WORKER-NOUN
JOBCARD-NP →
   (the JOBCARD-ATTRIBUTE of) JOBCARD-NOUN
ORDER-NP →
   (the ORDER-ATTRIBUTE of) ORDER-NOUN
OPERATION-NP →
   (the OPERATION-ATTRIBUTE of) OPERATION-NOUN

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → name
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN    → workers
JOBCARD-NOUN   → jobcards
ORDER-NOUN     → orders
OPERATION-NOUN → operations
```

Grammar Writing Hint: Simplify the grammar by using abbreviatory conventions when possible.

Collapse the four *S* rules in Grammar 10 into one rule by using the brace convention:

```
S → find {WORKER-NP   JOBCARD-NP   ORDER-NP   OPERATION-NP}
```

Then introduce another category to make a more revealing grammar (Grammar 11):

Grammar 11

S → find NP

NP → {WORKER-NP JOBCARD-NP ORDER-NP OPERATION-NP}

WORKER-NP →
  (the WORKER-ATTRIBUTE of) WORKER-NOUN
JOBCARD-NP →
  (the JOBCARD-ATTRIBUTE of) JOBCARD-NOUN
ORDER-NP →
  (the ORDER-ATTRIBUTE of) ORDER-NOUN
OPERATION-NP →
  (the OPERATION-ATTRIBUTE of) OPERATION-NOUN

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

Grammar 11 reads as follows:

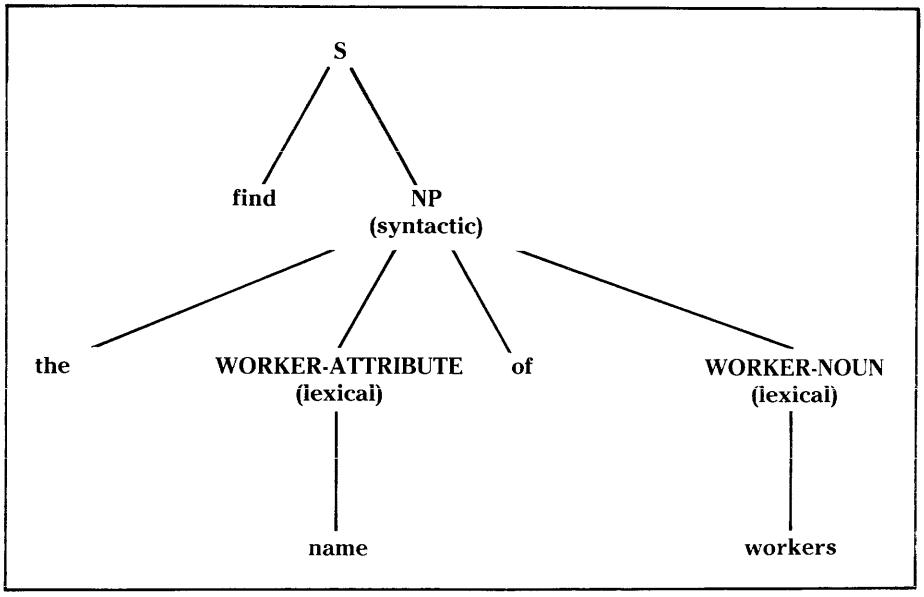
A sentence can consist of the terminal symbol *find* followed by NP (noun phrase). NP can consist of *WORKER-NP*, *JOBCARD-NP*, *ORDER-NP*, or *OPERATION-NP*. *WORKER-NP* can consist of an optional element of *the WORKER-ATTRIBUTE of* followed by *WORKER-NOUN* and so on.

Grammar 11 shows the basic structure of the language by using abbreviatory conventions and by introducing new broader categories (such as *NP*, *WORKER-NP*, *JOBCARD-NP*, *ORDER-NP*, and *OPERATION-NP*) into the grammar. These new categories, called *syntactic categories*, differ from the previous categories, called *lexical categories*. Syntactic categories define a sequence of words. Lexical categories define only groups of individual words (for example, *WORKER-ATTRIBUTE*, *JOBCARD-ATTRIBUTE*, *WORKER-NOUN*, and *JOBCARD-NOUN*).

Grammar Writing Hint: Introduce higher-level categories when necessary to make the grammar more revealing.

Figure 3-2 illustrates the distinction between lexical and syntactic categories for the sentence *find the name of workers*.

Figure 3-2 **Lexical and Syntactic Category Distinction**



Evaluating Grammars You can now revise Grammar 9 using grammatical categories and compare the resulting Grammar 12 with Grammar 11:

Grammar 12

S → find {NOUN NP}

NP → {WORKER-NP JOBCARD-NP ORDER-NP OPERATION-NP}

WORKER-NP → the WORKER-ATTRIBUTE of WORKER-NOUN
JOBCARD-NP → the JOBCARD-ATTRIBUTE of JOBCARD-NOUN
ORDER-NP → the ORDER-ATTRIBUTE of ORDER-NOUN
OPERATION-NP → the OPERATION-ATTRIBUTE of OPERATION-NOUN

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

NOUN → WORKER-NOUN
NOUN → JOBCARD-NOUN
NOUN → ORDER-NOUN
NOUN → OPERATION-NOUN

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
ORDER-NOUN → orders
OPERATION-NOUN → operations

Since Grammar 11 has 14 rules and Grammar 12 has 18 rules, Grammar 11 is preferable to Grammar 12.

Grammar Writing Hint: When two grammars are equally expressive, select the one that contains fewer rules.

Sentences With Recursion

3.2.5.3 Grammar writing also involves handling sentences in which repeated occurrences of some category are present. Consider the following sets of sentences and note that Grammar 11 can generate every sentence up to and including 9a, but not sentences 9b through 9d.

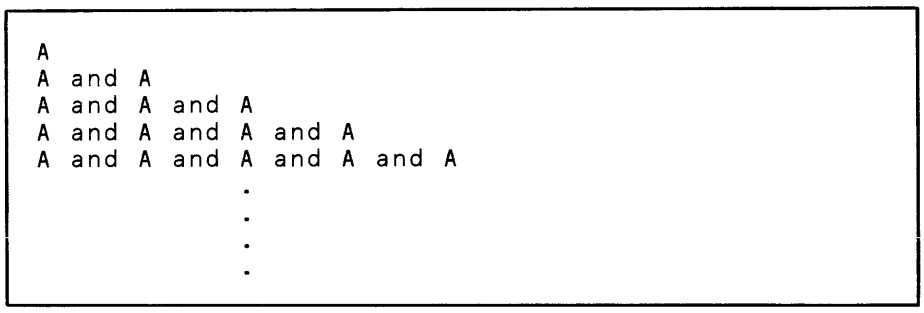
7 a. Find workers.
  b. Find jobcards.
  c. Find orders.
  d. Find operations.

8 a. Find the name of workers.
  b. Find the hours of jobcards.
  c. Find the price of orders.
  d. Find the descriptions of operations.

9 a. Find the name of workers.
  b. Find the name and birthdate of workers.
  c. Find the name and birthdate and wage rate of workers.
  d. Find the name and birthdate and wage rate and employee number of workers.

Reread sentences 9a through 9d, noting the emerging pattern. Figure 3-3 illustrates this pattern, where the symbol *A* stands for the category *WORKER-ATTRIBUTE*.

Figure 3-3 **Repeated Elements Pattern**



This is an example of recursion, and the repeated sequence of symbols is called the *recursive element*. The best way to write a grammar rule to handle recursion is as follows:

1. Find the recursive element, which in sentence set 9 is *and WORKER-ATTRIBUTE*.

2. Introduce a grammatical category that is one level higher than the basic element. Since *WORKER-ATTRIBUTE* is the basic element in the above instance, use *WORKER-ATTRIBUTE-NP* as the new category.

3. Write a rule where this new category appears on the left-hand side.

The first element on the right-hand side of this rule must be the *basic element*. The recursive element with the higher category comes next and is enclosed in parentheses to indicate that it is optional. This rule calls itself as many times as required:

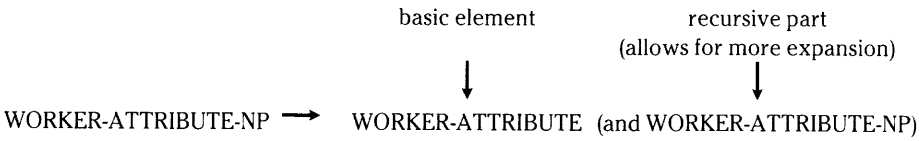
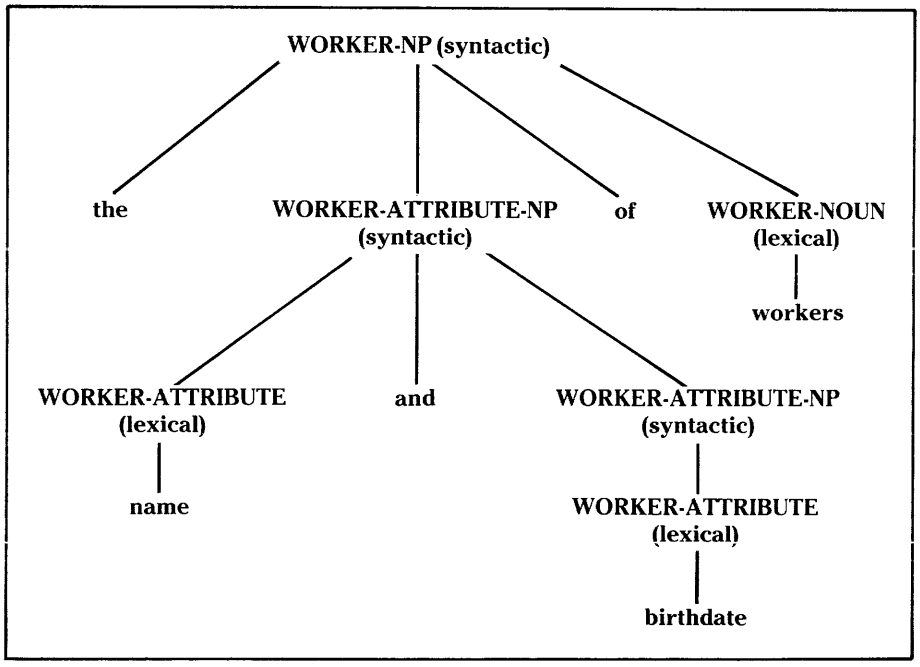


Figure 3-4 shows the partial tree diagram for the phrase *the name and birthdate of workers.*

Figure 3-4 **Tree Diagram With Recursion**



Replace the nonterminal symbol *WORKER-ATTRIBUTE* in the *WORKER-NP* rule with the new nonterminal symbol *WORKER-ATTRIBUTE-NP*:

WORKER-NP → (the WORKER-ATTRIBUTE-NP of) WORKER-NOUN

Since sentences of this type can also occur with the words *orders*, *jobcards*, and *operations*, you need to write the same type of rules for them, too. Now, the grammar (Grammar 13) is as follows:

Grammar 13

S → find NP

NP → {WORKER-NP JOBCARD-NP ORDER-NP OPERATION-NP}

```
WORKER-NP →
  (the WORKER-ATTRIBUTE-NP of) WORKER-NOUN
JOBCARD-NP →
  (the JOBCARD-ATTRIBUTE-NP of) JOBCARD-NOUN
ORDER-NP →
  (the ORDER-ATTRIBUTE-NP of) ORDER-NOUN
OPERATION-NP →
  (the OPERATION-ATTRIBUTE-NP of) OPERATION-NOUN

WORKER-ATTRIBUTE-NP →
  WORKER-ATTRIBUTE (and WORKER-ATTRIBUTE-NP)
JOBCARD-ATTRIBUTE-NP →
  JOBCARD-ATTRIBUTE (and JOBCARD-ATTRIBUTE-NP)
ORDER-ATTRIBUTE-NP →
  ORDER-ATTRIBUTE (and ORDER-ATTRIBUTE-NP)
OPERATION-ATTRIBUTE-NP →
  OPERATION-ATTRIBUTE (and OPERATION-ATTRIBUTE-NP)

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN → workers
JOBCARD-NOUN → jobcards
JOBCARD-NOUN → orders
JOBCARD-NOUN → operations
```

Ambiguous Sentences

3.2.5.4 If a sentence is subject to more than one interpretation—if it can be generated in more than one way—then that sentence and its grammar are *ambiguous*. The following hypothetical statement is ambiguous:

```
IF A=B THEN IF C=D THEN CALL F ELSE CALL G.
```

According to the definition of conditional statements, the ELSE branch could be associated with either the inner or the outer IF statement.

In natural languages, there are two types of ambiguities:

- Lexical ambiguity
- Surface structure ambiguity

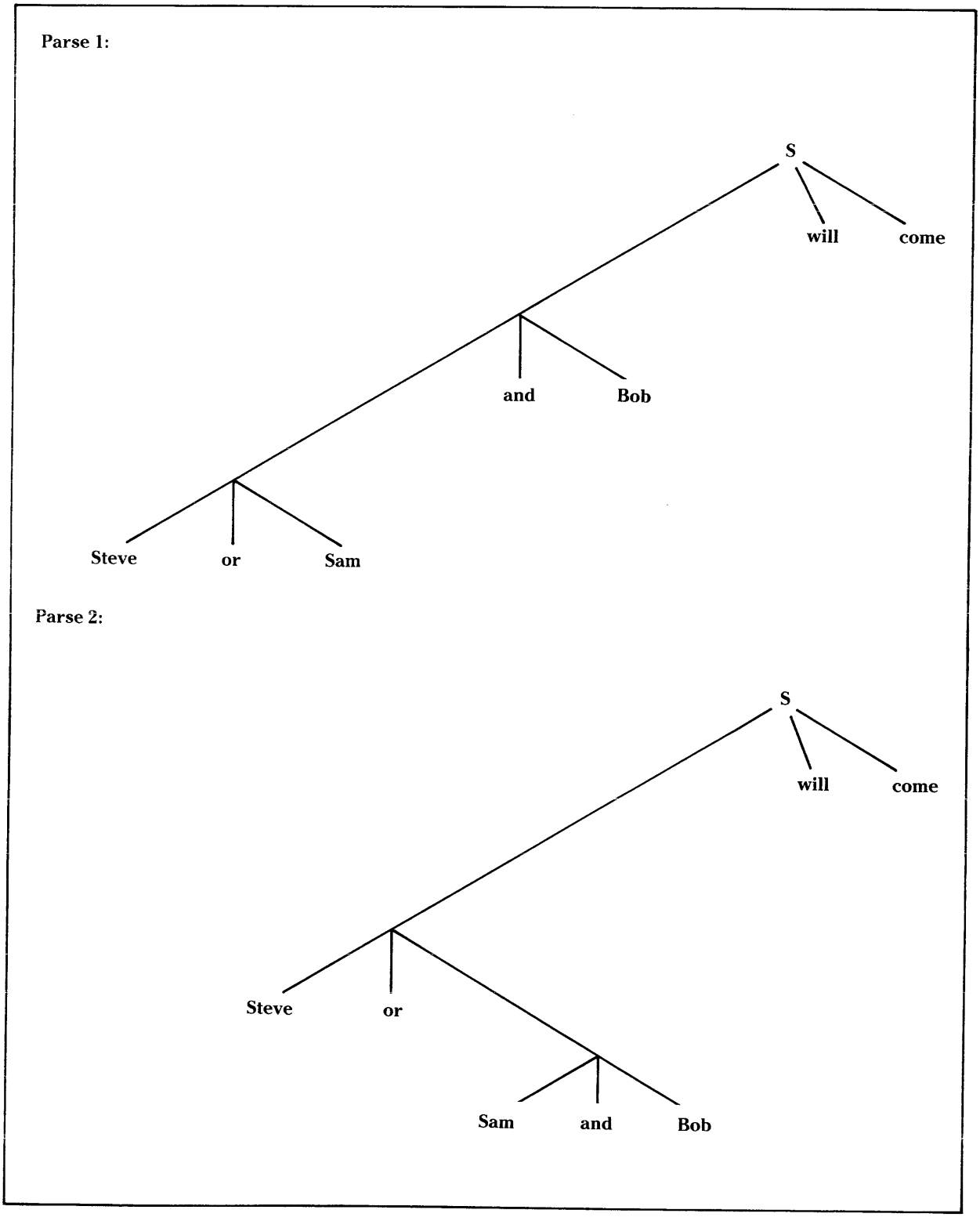
Consider the following two sentences:

10 a. The man is standing near the bank.
   b. Steve or Sam and Bob will come.

Sentence 10a is ambiguous because the word *bank* can mean either *river bank* or *financial institution*. Lexical ambiguity can be resolved by assigning two different lexical categories to the word *bank*. In natural languages, the context of a sentence tends to resolve lexical ambiguity. A clue to the lexical category of *bank* in sentence 10a would probably be found in some preceding or subsequent sentence. Because they deal with limited, context-free subsets of a language, NLMenu users can avoid this kind of ambiguity. For instance, the designer can require that *river bank*, not *bank* alone, be specified when that is the intention.

Sentence 10b is ambiguous because you cannot be sure who will come. Perhaps, Bob will come and either Steve or Sam will come. Or perhaps Steve will come or Sam and Bob will come. This type of ambiguity is called *surface structure ambiguity* because the surface appearance of the sentence itself causes ambiguity. Figure 3-5 shows trees that illustrate the underlying structure of each interpretation and that illustrate the ambiguity more clearly.

Figure 3-5 Parse Trees of Superficially Ambiguous Sentences



To resolve this surface structure ambiguity, you can use parentheses to indicate different interpretations (10c or 10d):

10 c. Steve or (Sam and Bob) will come.
d. (Steve or Sam) and Bob will come.

e. Find the jobcards that list operations whose operation number is equal to 50 and whose order number is greater than 100.

Sentence 10e also illustrates surface structure ambiguity. The sentence could be the natural language equivalent of a query to a database. The query indicates that the database contains a *JOBCARD* table and an *OPERATION* table, but the structure is ambiguous. If both the *JOBCARD* table and the *OPERATION* table have a field *ORDER NUMBER*, then the referent of the phrase *whose order number is greater than 100* is unclear. The referent can be either *jobcard order number* or *operation order number*. If you rewrite sentence 10e to read like either 10f or 10g, this ambiguity disappears:

10 f. Find jobcards that list operations whose operation number is equal to 50 and whose jobcard order number is greater than 100.

g. Find jobcards that list operations whose operation number is equal to 50 and whose operation order number is greater than 100.

Further examples of this type of sentence are as follows:

10 h. Find workers who have jobcards whose jobcard order number is 20 and whose worker employee number is 150.

i. Find workers who have jobcards whose jobcard order number is 20 and whose jobcard employee number is 150.

j. Find operations that are listed on jobcards whose jobcard hours are 10 and whose operation piece number is 200.

k. Find operations that are listed on jobcards whose jobcard hours are 10 and whose jobcard piece number is 200.

Although some of the preceding sentences may seem awkward, they are actually just very explicit. In normal spoken or written usage, the context of a sentence helps to disambiguate the underlying meaning. When taken out of context, this seemingly awkward explicitness is needed to avoid potential ambiguity.

A grammar should be able to resolve these ambiguities. However, if the grammar you write allows multiple interpretations of some input sentence, the NLMenu parser indicates that there is more than one parse for the input sentence. The parser lists the various parses according to its determination of the probability of intent. The interface user can then indicate which parse corresponds to his intent. If such ambiguities are found, you can use the NLMenu grammar tools to refine the grammar and/or lexicon.

Sentences With Relative Clauses

3.2.5.5 The following discussion concerns how to revise the evolving grammar of previous paragraphs to generate more complicated sentences, such as the following:

- Find workers whose name is equal to <specific name>.
- Find jobcards whose order number is not equal to <specific order number>.
- Find orders whose price is greater than <specific price>.
- Find operations whose operation number is less than <specific operation number>.
- Find workers whose wage rate is greater than or equal to <specific wage rate>.
- Find orders whose quantity is less than or equal to <specific quantity>.

.
.
.

These sentences have additional clauses introduced by *whose*. This clause describes and limits the *NOUN* element preceding it. The output of such a query is then modified to include *only* those elements which meet the description or restriction given in the *whose* clause. *MODIFIER* is, then, a good identifying label for such constructions. *MODIFIER*s occur with elements of the *NOUN* category.

In the development of the grammar, you saw the need to incorporate a semantic or meaning factor into the grammar by splitting a general category into the more specific categories. If this is not done, the grammar can be too powerful and can generate sentences beyond the scope of the language or language subset. For similar reasons, it is necessary to split the *MODIFIER* category and, from it, create the new categories: *WORKER-MOD*, *JOBCARD-MOD*, *ORDER-MOD*, and *OPERATION-MOD*.

Within the *whose* clause, you see two addtional new constructions:

- *Comparison predicates* — Natural language expressions that correspond to the set of comparison operators (=, ≠, <, >, ≤, and ≥) used in the field of logic
- *Open slots* — Constructions that enable the grammar to handle specific numeric or other constant values

Since there is an infinite number of possible numeric or constant values, it is unreasonable to attempt to write a grammar to generate all possibilities. Therefore, the NLMenu system provides the expert window facility that allows you to designate such open slots, or *experts*. The types of experts and their uses are as follows:

- Type-in experts — For entering a specific alphanumeric value
- Range experts — For entering a value within some range
- Calculator experts — For entering numeric values
- Default experts — For choosing a value from a menu or, by moving the mouse away from this menu, automatically setting some predetermined value (choosing the default value)

Refer to paragraph 4.9 for more information on experts.

The following revision of Grammar 13 (Grammar 14) generates sentences with constructions involving comparison operators. A new category *COMPARISON-PRED* is introduced to represent such constructions.

Grammar 14

S → find NP

NP → {WORKER-NP JOBCARD-NP ORDER-NP OPERATION-NP}

WORKER-NP →
  (the WORKER-ATTRIBUTE-NP of) WORKER-NOUN (WORKER-MOD)
JOBCARD-NP →
  (the JOBCARD-ATTRIBUTE-NP of) JOBCARD-NOUN (JOBCARD-MOD)
ORDER-NP →
  (the ORDER-ATTRIBUTE-NP of) ORDER-NOUN (ORDER-MOD)
OPERATION-NP →
  (the OPERATION-ATTRIBUTE-NP of) OPERATION-NOUN
  (OPERATION-MOD)

WORKER-ATTRIBUTE-NP →
  WORKER-ATTRIBUTE (and WORKER-ATTRIBUTE-NP)
JOBCARD-ATTRIBUTE-NP →
  JOBCARD-ATTRIBUTE (and JOBCARD-ATTRIBUTE-NP)
ORDER-ATTRIBUTE-NP →
  ORDER-ATTRIBUTE (and ORDER-ATTRIBUTE-NP)
OPERATION-ATTRIBUTE-NP →
  OPERATION-ATTRIBUTE (and OPERATION-ATTRIBUTE-NP)

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

```
JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → operation-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN    → workers
JOBCARD-NOUN   → jobcards
ORDER-NOUN     → orders
OPERATION-NOUN → operations

COMPARISON-PRED → equal-to
COMPARISON-PRED → not-equal-to
COMPARISON-PRED → greater-than
COMPARISON-PRED → less-than
COMPARISON-PRED → greater-than-or-equal-to
COMPARISON-PRED → less-than-or-equal-to

WORKER-MOD → whose-worker-name-is COMPARISON-PRED
               worker-name-expert
WORKER-MOD → whose-worker-employee-number-is
               COMPARISON-PRED worker-employee-number-expert
WORKER-MOD → whose-worker-birthdate-is COMPARISON-PRED
               worker-birthdate-expert
WORKER-MOD → whose-worker-wage-rate-is COMPARISON-PRED
               worker-wage-rate-expert

JOBCARD-MOD → whose-jobcard-hours-are COMPARISON-PRED
               jobcard-hours-expert
JOBCARD-MOD → whose-jobcard-piece-done-is COMPARISON-PRED
               jobcard-piece-done-expert
JOBCARD-MOD → whose-jobcard-order-number-is COMPARISON-PRED
               jobcard-order-number-expert
JOBCARD-MOD → whose-jobcard-job-date-is COMPARISON-PRED
               jobcard-job-date-expert

ORDER-MOD → whose-order-price-is COMPARISON-PRED
              order-price-expert
ORDER-MOD → whose-order-date-filled-is COMPARISON-PRED
              order-date-filled-expert
ORDER-MOD → whose-order-date-received-is COMPARISON-PRED
              order-date-received-expert
```

ORDER-MOD → whose-order-quantity-is COMPARISON-PRED order-quantity-expert

OPERATION-MOD → whose-operation-descriptions-are COMPARISON-PRED operation-description-expert
OPERATION-MOD → whose-operation-rejects-are COMPARISON-PRED operation-rejects-expert
OPERATION-MOD → whose-operation-operation-number-is COMPARISON-PRED operation-operation-number-expert
OPERATION-MOD → whose-operation-piece-number-is COMPARISON-PRED operation-piece-number-expert

The various *MODIFIER* categories treat each entire whose-clause as a terminal and as a unit, rather than further subdividing them. There is a practical reason for foregoing such subdivision. The number of selection window panes that the interface must present to the end user increases as the number of grammatical categories increases. In this instance, the underlying function and meaning of the whose-clause is better captured by treating it as a unit. And, from a practical standpoint, the interface is not unduly complicated because only one new window pane is added.

To make this grammar more complete, add the capability of recursively generating *MODIFIER*s. Introduce a new syntactic category one level above the *MODIFIER* type presented in Grammar 14. Label this *MOD-CONSTR*. Define this new category in terms of the *MOD* construction, using *MOD* as the basic element and *and MOD-CONSTR* as the recursive element. A generic version of the rule is:

MOD-CONSTR → MOD (and MOD-CONSTR)

Grammar 15 incorporates this latest feature into the grammar design:

Grammar 15

S → find NP

NP → {WORKER-NP JOBCARD-NP ORDER-NP OPERATION-NP}

WORKER-NP →
(the WORKER-ATTRIBUTE-NP of) WORKER-NOUN (WORKER-MOD-CONSTR)
JOBCARD-NP →
(the JOBCARD-ATTRIBUTE-NP of) JOBCARD-NOUN (JOBCARD-MOD-CONSTR)
ORDER-NP →
(the ORDER-ATTRIBUTE-NP of) ORDER-NOUN (ORDER-MOD-CONSTR)

```
OPERATION-NP →
    (the OPERATION-ATTRIBUTE-NP of) OPERATION-NOUN
    (OPERATION-MOD-CONSTR)

WORKER-ATTRIBUTE-NP →
    WORKER-ATTRIBUTE (and WORKER-ATTRIBUTE-NP)
JOBCARD-ATTRIBUTE-NP →
    JOBCARD-ATTRIBUTE (and JOBCARD-ATTRIBUTE-NP)
ORDER-ATTRIBUTE-NP →
    ORDER-ATTRIBUTE (and ORDER-ATTRIBUTE-NP)
OPERATION-ATTRIBUTE-NP →
    OPERATION-ATTRIBUTE (and OPERATION-ATTRIBUTE-NP)

WORKER-MOD-CONSTR →
    WORKER-MOD (and WORKER-MOD-CONSTR)
JOBCARD-MOD-CONSTR →
    JOBCARD-MOD (and JOBCARD-MOD-CONSTR)
ORDER-MOD-CONSTR →
    ORDER-MOD (and ORDER-MOD-CONSTR)
OPERATION-MOD-CONSTR →
    OPERATION-MOD (and OPERATION-MOD-CONSTR)

WORKER-ATTRIBUTE → name
WORKER-ATTRIBUTE → employee-number
WORKER-ATTRIBUTE → birthdate
WORKER-ATTRIBUTE → wage-rate

JOBCARD-ATTRIBUTE → hours
JOBCARD-ATTRIBUTE → piece-done
JOBCARD-ATTRIBUTE → order-number
JOBCARD-ATTRIBUTE → job-date

ORDER-ATTRIBUTE → price
ORDER-ATTRIBUTE → date-filled
ORDER-ATTRIBUTE → date-received
ORDER-ATTRIBUTE → quantity

OPERATION-ATTRIBUTE → descriptions
OPERATION-ATTRIBUTE → rejects
OPERATION-ATTRIBUTE → opertion-number
OPERATION-ATTRIBUTE → piece-number

WORKER-NOUN    → workers
JOBCARD-NOUN   → jobcards
ORDER-NOUN     → orders
OPERATION-NOUN → operations

COMPARISON-PRED → equal-to
COMPARISON-PRED → not-equal-to
COMPARISON-PRED → greater-than
COMPARISON-PRED → less-than
```

```
COMPARISON-PRED → greater-than-or-equal-to
COMPARISON-PRED → less-than-or-equal-to

WORKER-MOD → whose-worker-name-is
                COMPARISON-PRED worker-name-expert
WORKER-MOD → whose-worker-employee-number-is
                COMPARISON-PRED worker-employee-number-
                expert
WORKER-MOD → whose-worker-birthdate-is
                COMPARISON-PRED worker-birthdate-expert
WORKER-MOD → whose-worker-wage-rate-is
                COMPARISON-PRED worker-wage-rate-expert

JOBCARD-MOD → whose-jobcard-hours-are
                COMPARISON-PRED jobcard-hours-expert
JOBCARD-MOD → whose-jobcard-piece-done-is
                COMPARISON-PRED jobcard-piece-done-expert
JOBCARD-MOD → whose-jobcard-order-number-is
                COMPARISON-PRED jobcard-order-number-expert
JOBCARD-MOD → whose-jobcard-job-date-is
                COMPARISON-PRED jobcard-job-date-expert

ORDER-MOD → whose-order-price-is
                COMPARISON-PRED order-price-expert
ORDER-MOD → whose-order-date-filled-is
                COMPARISON-PRED order-date-filled-expert
ORDER-MOD → whose-order-date-received-is
                COMPARISON-PRED order-date-received-expert
ORDER-MOD → whose-order-quantity-is
                COMPARISON-PRED order-quantity-expert

OPERATION-MOD → whose-operation-descriptions-are
                COMPARISON-PRED operation-description-
                expert
OPERATION-MOD → whose-operation-rejects-are
                COMPARISON-PRED operation-rejects-expert
OPERATION-MOD → whose-operation-operation-number-is
                COMPARISON-PRED operation-operation-
                number-expert
OPERATION-MOD → whose-operation-piece-number-is
                COMPARISON-PRED operation-piece-number-
                expert
```

This limited sketch of grammar development demonstrates the steps and considerations involved in grammar writing. Section 4 describes the format in which you enter the grammar and lexicon files.

Summary

3.3 As a way to review this section, consider again the grammar writing hints that were discussed during the development of the example grammar.

- Do not merely list valid sentences; use meaningful categories to capture regularities.
- Do not introduce unnecessary categories.
- Do not make the grammar too powerful.
- Incorporate semantic information in the grammar to constrain its power by further disambiguating the grammar.
- Do not write one grammar for simple sentences and another for more complex sentences. Devise a grammar that generates both.
- Make the *S* rule as simple as possible.
- When evaluating alternative grammars, consider which one best captures the regularities of the language.
- Simplify the grammar by using abbreviatory conventions when possible.
- Introduce higher-level categories when necessary to make the grammar more revealing.
- When two grammars are equally expressive, select the one that contains fewer rules.

Following this presentation of the fundamentals of grammar writing, it is appropriate to point out that there will be no single correct grammar for a particular interface. Nor should you expect to write a completely unambiguous grammar on your first attempt. However, do not be intimidated by the process. The NLMenu system contains a number of tools to assist in grammar writing and interface development. The subsequent sections treat these tools in detail, in roughly the same order in which the interface designer would employ them.

APPLICATION DESIGNER INPUT TO NLMENU 4

Highlights of This Section:

- How Natural Language Menu uses the grammar and lexicon
- Grammar format
- Translation lists
- Grammar file
  - Grammar file example
  - Grammar file discussion
- Lexicon file
  - Lexicon file example
  - Lexicon file discussion
- Translation process
  - Illustrations of the translation process
  - Summary of translator algorithm
- Developing translations
  - Dow Jones™ example
  - Choosing an appropriate translation scheme
- Developing screen configuration descriptions
- Implementation of experts

Dow Jones is a trademark of Dow Jones & Company, Inc.

How Natural Language Menu Uses the Grammar and Lexicon

4.1 You provide the Natural Language Menu (NLMenu) system with three types of input to create a natural language interface:

- A grammar that defines the valid combinations of words and phrases in a selected natural language, the order of such words and phrases, and the appropriate mapping of these words and phrases into the target language of the application
- A lexicon that provides translations of the words or phrases used in the mapping and establishes the relationship between the grammar's terminals and the target language
- A screen description that tells the system where to place windows and what words or phrases to insert into them

The NLMenu tools help you create an NLMenu interface by automating much of the grammar and lexicon testing and by providing an interactive lexicon building utility.

The parser and translator use the grammar and lexicon to control the screen display and the translation into the target language. The parser is responsible for not only constructing the parse tree, but also for determining the set of next legal choices given the current item selection. The parser must use information in both the grammar and the lexicon to perform this task. The translator takes the parse tree built by the parser and, by referencing the grammar and lexicon, develops the target string. Understanding how the translator builds the target string is important in developing NLMenu grammar and lexicon files. The translation process is presented after a discussion of the grammar format and lexicon content.

Grammar Format

4.2 You need to modify the basic format of your grammar before the NLMenu system can process it. These modifications entail recasting each grammar rule in a Lisp format that is described in detail below.

The NLMenu system does not support a number of grammar components. It takes as input a context-free semantic grammar, with the following restrictions:

- No null terminals — No grammatical category can have **nil** as its expansion; for example, the rule

  WORKER-NP → nil

  is not allowed.

- No cycles — Derivations that can produce an infinitely looping path are not allowed; for example, the following sequence of rules would violate the restriction since the final rule introduces a cycle back to the first:

  S →A B
  A →a
  B →b S

NOTE: Most standard abbreviatory conventions are permitted, although Kleene star (unlimited repetition of an element) is not supported. An example of Kleene star is the following: `A --> B* C.`

An example of an NLMenu grammar rule follows:

```
("S --> change change-suppliers (SUPPLIER-MOD-CONSTRUCT)
 so-that SUPPLIER-UPDATES"
 ((1 2 5) (1 2 4 5))((4 2 5 3) (1 2 3 4 5)))
```

You write each individual grammar rule as a Lisp list. The entire grammar is one long list, comprising lists of rules written in the following format:

(*"mother → left-corner daughter2...daughterN"*
(*translation-function1 expansion-list1*)...
(*translation-functionM expansion-listM*))

where:

mother (S) is the single element on the left-hand side of the rule.

left-corner (change) is the first element (that is, daughter) on the right-hand side of the rule. This must be a grammatical category.

daughterN (change-suppliers, (SUPPLIER-MOD-CONSTRUCT), so-that, and SUPPLIER-UPDATES) is a remaining element on the right-hand side (if any) of the rule. It must be a grammatical category and can include the use of any of the abbreviatory symbols.

translation-functionM ((1 2 5) and (4 2 5 3)) is a Lisp expression where numbers correspond to the position of the left corner or daughters in the grammar rule. Begin numbering by assigning the value 1 to the left corner and assign sequentially increasing values to any other elements on the right-hand side. Analysis of a translation function proceeds from left to right unless overridden by parentheses. For example, if a translation function reads (1 2 5), the translation portion of the lexical entry for *element1* (right-hand side) is applied with the translation portion of the lexical entry for *element2* (right-hand side) as its argument. That result then becomes the function that is applied with the translation portion of the lexical entry for *element5* (right-hand side) as its argument.

NOTE: The translation portion of the lexical entry is always either a lambda expression with one argument or an atomic element. Application of a lambda expression to an argument can produce another lambda expression (as in points 1 and 2 directly above) or an atomic element. In general, to combine n atomic elements together, a lambda expression with $n - 1$ occurrences of *lambda* in its body is required.

If, however, the translation function reads (4 (1 2)), the translation portion of the lexical entry for *element1* is applied to that of *element2*, and that result then becomes the argument of the function given in the translation portion of the lexical entry for *element4*.

expansion-listM (`(1 2 4 5)` and `(1 2 3 4 5)`) is a listing of the numbers corresponding to the position of all the left corner or daughter elements of the rule associated with translation-functionM. The expansion list resolves ambiguity regarding which translation list corresponds to which expansion of a rule when abbreviatory conventions occur in the rule. If no ambiguity is possible, you need not include the expansion list. Rules without abbreviatory elements never need to contain expansion lists. For example, the preceding *S* rule has the following two expansions:

```
(1) S --> change change-suppliers so-that supplier-updates
              1          2            4           5

(2) S --> change change-suppliers supplier-mod-construct
              1          2                      3

            so-that supplier-updates
              4           5
```

Expansion 1 has `(1 2 4 5)` as its expansion list, while Expansion 2 has an expansion list of `(1 2 3 4 5)`.

Thus, the representation of the syntactic component of an NLMenu grammar corresponds to the standard notations used by linguists for context-free grammars. The translation functions and expansion lists represent the semantic component of the grammar. For example, a rule with syntactic component

```
X --> A B C
```

is written as

```
("X --> A B C" ((translation-function) (expansion-list)))
```

Translation functions and expansion lists are discussed in more detail later in this section.

Rule elements must be grammatical categories, and you delimit them by one or more spaces or carriage returns. A grammatical category is either a terminal (appears only on the right-hand side of the rule) or nonterminal (appears on either side of the rule) in the grammar.

Abbreviatory elements allow you to collapse rules that share the same mother and left corner into a single rule. Refer to Section 3 for detailed information on the various abbreviatory elements and development of the syntactic portion of an NLMenu grammar.

Translation Lists

4.3 A translation list consists of a series of translation functions and their corresponding expansion lists. The expansion list lets the translator know which expansion of the rule the parser used, while the translation function indicates how to go about building the target string. To establish which expansion was used, the system first assigns each grammatical category in the rule an integer, starting with the value 1 at the left corner and incrementing by one for each succeeding category. For example, the rule

```
X --> A {[B C] [D E]} F
```

is assigned the values

```
X --> A {[B C] [D E]} F
      1   2 3   4 5   6
```

The relevant expansion lists are `(1 2 3 6)` for the expansion

```
X --> A B C F
```

and `(1 4 5 6)` for the expansion

```
X --> A D E F
```

The resulting translation list will thus have the form:

`((x1 ... xM)`	`(1 2 3 6))`	`((y1 ... yN)`	`(1 4 5 6))`
translation function1	expansion list1	translation function2	expansion list2

If the first expansion of the rule was used by the parser, translation function1 is used to indicate how to combine the translations associated with each grammar rule component. This means that the translation of x1 is taken as a lambda expression or function and applied to the translation of x2 as its argument. The resulting new translation is then taken as a function and applied to the translation of x3 as its argument, and so forth. In general, the translation of the leftmost element of the translation function applies to the translation of the element to its right as the argument. This process continues until there are no more arguments. Examples of the translation process are included in paragraph 4.6. An example of a grammar rule and its associated translation functions and expansion lists follows:

```
("S --> change change-suppliers (SUPPLIER-MOD-CONSTRUCT)
 so-that SUPPLIER-UPDATES"
 ((1 2 5) (1 2 4 5))((4 2 5 3) (1 2 3 4 5)))
```

It is not necessary for all grammatical categories used in a given rule expansion to appear in the translation function. The only integers that need appear are those the translator uses to build the target string.

Nesting is allowed in the translation function. To do this, simply use an additional set of parentheses. For example, the translation list

```
((2 (3 1) 6) (1 2 3 6))
```

tells the translator to first apply the lambda expression representing the translation of *daughter3* to the expression representing the translation of *daughter1*. The lambda expression representing the translation of *daughter2* is then applied to this result, producing another lambda expression that is subsequently applied to the expression representing the translation of *daughter6*.

Grammar File

4.4 The following is a sample NLMenu grammar for a natural language interface to the sample Parts-and-Suppliers database. (This is the default grammar mentioned in paragraph 2.2. The grammar is similar to but more complex than an extended version of the one developed in Section 3 and is written in the format described in paragraph 4.2.) This grammar allows you to build sentences to change, insert, or delete records in the Parts-and-Suppliers database. It contains built-in mechanisms that allow more computational/numerical constructs and instance-specific, or expert, values as input. This grammar contains many examples of recursive rules and of the introduction of higher-level categories to make the grammar more revealing (see paragraph 3.2.5.2). Notice also that the second set of integers in many of these rules, the expansion list, is missing because the translation function itself is sufficient to determine which rule expansion corresponds to it.

```
(

("S --> change change-suppliers (SUPPLIER-MOD-CONSTRUCT)
 so-that SUPPLIER-UPDATES"
 ((1 2 5) (1 2 4 5))((4 2 5 3) (1 2 3 4 5)))

("S --> change change-parts (PART-MOD-CONSTRUCT)
 so-that PART-UPDATES"
 ((1 2 5) (1 2 4 5))((4 2 5 3) (1 2 3 4 5)))

("S --> change change-shipments (SHIPMENT-MOD-CONSTRUCT)
 so-that SHIPMENT-UPDATES"
 ((1 2 5) (1 2 4 5))((4 2 5 3) (1 2 3 4 5)))

("S --> insert TUPLES"
 ((1 2)))

("S --> delete-rel SUPPLIER-NP"
 ((2 1)))

("S --> delete-rel PART-NP"
 ((2 1)))

("S --> delete-rel SHIPMENT-NP"
 ((2 1)))

("S --> find-rel SUPPLIER-NP"
 ((2 1)))

("S --> find-rel PART-NP"
 ((2 1)))

("S --> find-rel SHIPMENT-NP"
 ((2 1)))
```

```
("S --> find-attr-rel SUPPLIER-ATTR-CONSTR for SUPPLIER-NP"
 ((4 (1 2)) (1 2 3 4)))

("S --> find-attr-rel PART-ATTR-CONSTR for PART-NP"
 ((4 (1 2)) (1 2 3 4)))

("S --> find-attr-rel SHIPMENT-ATTR-CONSTR for SHIPMENT-NP"
 ((4 (1 2)) (1 2 3 4)))

("S --> find-comp {NUMBER-NP COMPUTABLE-NP}"
 ((2) (1 2)) ((3) (1 3)))

("SUPPLIER-MOD-CONSTRUCT --> SUPPLIER-MOD-AND
 (s-or SUPPLIER-MOD-CONSTRUCT)"
 ((1))((2 1 3)))

("PART-MOD-CONSTRUCT --> PART-MOD-AND
 (s-or PART-MOD-CONSTRUCT)"
 ((1))((2 1 3)))

("SHIPMENT-MOD-CONSTRUCT --> SHIPMENT-MOD-AND
 (s-or SHIPMENT-MOD-CONSTRUCT)"
 ((1))((2 1 3)))

("SUPPLIER-MOD-AND --> SUPPLIER-MOD-BASE
 (s-and SUPPLIER-MOD-AND)"
 ((1))((2 1 3)))

("PART-MOD-AND --> PART-MOD-BASE (s-and PART-MOD-AND)"
 ((1))((2 1 3)))

("SHIPMENT-MOD-AND --> SHIPMENT-MOD-BASE
 (s-and SHIPMENT-MOD-AND)"
 ((1))((2 1 3)))

("SUPPLIER-MOD-BASE --> left-bracket
 SUPPLIER-MOD-CONSTRUCT right-bracket"
 ((2) (1 2 3)))

("PART-MOD-BASE --> left-bracket PART-MOD-CONSTRUCT
 right-bracket"
 ((2) (1 2 3)))

("SHIPMENT-MOD-BASE --> left-bracket
 SHIPMENT-MOD-CONSTRUCT right-bracket"
 ((2) (1 2 3)))

("SUPPLIER-MOD-BASE --> SUPPLIER-MOD"
 ((1)))

("PART-MOD-BASE --> PART-MOD"
 ((1)))
```

```
("SHIPMENT-MOD-BASE --> SHIPMENT-MOD"
 ((1)))

("PART-MOD --> whose-part-city-is part-city-expert"
 ((1 2)))

("PART-MOD --> whose-part-color-is part-color-expert"
 ((1 2)))

("PART-MOD --> whose-part-name-is part-name-expert"
 ((1 2)))

("PART-MOD --> whose-part-part#-is part-part#-expert"
 ((1 2)))

("SUPPLIER-MOD --> whose-supplier-city-is
 supplier-city-expert"
 ((1 2)))

("SUPPLIER-MOD --> whose-supplier-name-is
 supplier-name-expert"
 ((1 2)))

("SUPPLIER-MOD --> whose-supplier-supplier#-is
 supplier-supplier#-expert"
 ((1 2)))

("SHIPMENT-MOD --> whose-shipment-part#-is
 shipment-part#-expert"
 ((1 2)))

("SHIPMENT-MOD --> whose-shipment-supplier#-is
 shipment-supplier#-expert"
 ((1 2)))

("SUPPLIER-MOD --> whose-supplier-status-is
 {[COMPARISON-PRED
 SUPPLIER-STATUS-NUMERIC-EXPR]
 [between SUPPLIER-STATUS-NUMERIC-EXPR between-and
 SUPPLIER-STATUS-NUMERIC-EXPR]}"
 ((1 2 3))((1 4 (6 5 7))))

("PART-MOD --> whose-supplier-status-is
 {[COMPARISON-PRED PART-WEIGHT-NUMERIC-EXPR]
 [between PART-WEIGHT-NUMERIC-EXPR
 between-and PART-WEIGHT-NUMERIC-EXPR]}"
 ((1 2 3))((1 4 (6 5 7))))

("SHIPMENT-MOD --> whose-shipment-status-is
 {[COMPARISON-PRED SHIPMENT-QUANTITY-NUMERIC-EXPR]
 [between SHIPMENT-QUANTITY-NUMERIC-EXPR
 between-and SHIPMENT-QUANTITY-NUMERIC-EXPR]}"
 ((1 2 3))((1 4 (6 5 7))))
```

```
("SHIPMENT-MOD --> shipment-which are shipments of-part
 PART-NP"
 ((2 1)))

("SHIPMENT-MOD --> shipment-which were shipped
 by-supplier SUPPLIER-NP"
 ((2 1)))

("SUPPLIER-MOD --> supplier-who ship-shipment SHIPMENT-NP"
 ((2 1)))

("SUPPLIER-MOD --> who supply-shipment-supplier-part
 PART-EMBEDDED-NP"
 ((2 1)))

("PART-MOD --> which are supplied by-shipment-part-supplier
 SUPPLIER-EMBEDDED-NP"
 ((2 1)))

("COMPARISON-PRED --> equal-to"
 ((1)))

("COMPARISON-PRED --> greater-than"
 ((1)))

("COMPARISON-PRED --> less-than"
 ((1)))

("COMPARISON-PRED --> greater-than-or-equal-to"
 ((1)))

("COMPARISON-PRED --> less-than-or-equal-to"
 ((1)))

("COMPARISON-PRED --> not-equal-to"
 ((1)))

("SUPPLIER-STATUS-NUMERIC-EXPR --> supplier-status-expert"
 ((1)))

("SUPPLIER-STATUS-NUMERIC-EXPR --> NUMBER-NP"
 ((1)))

("SUPPLIER-STATUS-NUMERIC-EXPR --> COMPUTABLE-NP"
 ((1)))

("PART-WEIGHT-NUMERIC-EXPR --> part-weight-expert"
 ((1)))

("PART-WEIGHT-NUMERIC-EXPR --> NUMBER-NP"
 ((1)))

("PART-WEIGHT-NUMERIC-EXPR --> COMPUTABLE-NP"
 ((1)))
```

```
("SHIPMENT-QUANTITY-NUMERIC-EXPR -->
 shipment-quantity-expert"
 ((1)))

("SHIPMENT-QUANTITY-NUMERIC-EXPR --> NUMBER-NP"
 ((1)))

("SHIPMENT-QUANTITY-NUMERIC-EXPR --> COMPUTABLE-NP"
 ((1)))

("SUPPLIER-NP --> MODIFIABLE-SUPPLIER-NP"
 ((1)))

("PART-NP --> MODIFIABLE-PART-NP"
 ((1)))

("SHIPMENT-NP --> MODIFIABLE-SHIPMENT-NP"
 ((1)))

("MODIFIABLE-SUPPLIER-NP --> suppliers"
 ((1)))

("MODIFIABLE-PART-NP --> parts"
 ((1)))

("MODIFIABLE-SHIPMENT-NP --> shipments"
 ((1)))

("MODIFIABLE-SUPPLIER-NP --> where-suppliers
 SUPPLIER-MOD-CONSTRUCT"
 ((1 2)))

("MODIFIABLE-PART-NP --> where-parts PART-MOD-CONSTRUCT"
 ((1 2)))

("MODIFIABLE-SHIPMENT-NP --> where-shipments
 SHIPMENT-MOD-CONSTRUCT"
 ((1 2)))

("SUPPLIER-EMBEDDED-NP --> MODIFIABLE-SUPPLIER-EMBEDDED-NP"
 ((1)))

("PART-EMBEDDED-NP --> MODIFIABLE-PART-EMBEDDED-NP"
 ((1)))

("MODIFIABLE-SUPPLIER-EMBEDDED-NP --> embedded-suppliers"
 ((1)))

("MODIFIABLE-PART-EMBEDDED-NP --> embedded-parts"
((1)))

("MODIFIABLE-SUPPLIER-EMBEDDED-NP -->
 embedded-where-suppliers
 SUPPLIER-MOD-CONSTRUCT"
 ((1 2)))
```

```
("MODIFIABLE-PART-EMBEDDED-NP --> embedded-where-parts
 PART-MOD-CONSTRUCT"
 ((1 2)))

("PART-UPDATES --> the-new-city-is part-city-expert
 (update-and PART-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("PART-UPDATES --> the-new-color-is
 part-color-expert (update-and PARTS-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("PART-UPDATES --> the-new-name-is part-name-expert
 (update-and PART-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("PART-UPDATES --> the-new-part#-is
 part-part#-expert (update-and PART-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SUPPLIER-UPDATES --> the-new-city-is
 supplier-city-expert (update-and SUPPLIER-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SUPPLIER-UPDATES --> the-new-name-is
 supplier-name-expert (update-and SUPPLIER-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SUPPLIER-UPDATES --> the-new-supplier#-is
 supplier-supplier#-expert (update-and SUPPLIER-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SHIPMENT-UPDATES --> the-new-part#-is
 shipment-part#-expert (update-and SHIPMENT-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SHIPMENT-UPDATES --> the-new-supplier#-is
 shipment-supplier#-expert (update-and SHIPMENT-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SUPPLIER-UPDATES --> the-new-status-is
 supplier-status-expert (update-and SUPPLIER-UPDATES)"
((1 2))((3 (1 2) 4)))

("PART-UPDATES --> the-new-weight-is
 part-weight-expert (update-and PART-UPDATES)"
 ((1 2))((3 (1 2) 4)))

("SHIPMENT-UPDATES --> the-new-quantity-is
 shipment-quantity-expert (update-and SHIPMENT-UPDATES)"
 ((1 2))((3 (1 2) 4)))
```

```
("TUPLES --> new-supplier-tuple (tuple-and TUPLES)"
 ((1))((2 1 3)))

("TUPLES --> new-part-tuple (tuple-and TUPLES)"
 ((1))((2 1 3)))

("TUPLES --> new-shipment-tuple (tuple-and TUPLES)"
 ((1))((2 1 3)))

("SUPPLIER-ATTR-CONSTR --> SUPPLIER-ATTR
 (attr-and SUPPLIER-ATTR-CONSTR)"
 ((1))((2 1 3)))

("PART-ATTR-CONSTR --> PART-ATTR (attr-and
 PART-ATTR-CONSTR)"
 ((1))((2 1 3)))

("SHIPMENT-ATTR-CONSTR --> SHIPMENT-ATTR
 (attr-and SHIPMENT-ATTR-CONSTR)"
 ((1))((2 1 3)))

("PART-ATTR --> city"
 ((1)))

("PART-ATTR --> color"
 ((1)))

("PART-ATTR --> name"
 ((1)))

("PART-ATTR --> part#"
 ((1)))

("SUPPLIER-ATTR --> city"
 ((1)))

("SUPPLIER-ATTR --> name"
 ((1)))

("SUPPLIER-ATTR --> supplier#"
 ((1)))

("SHIPMENT-ATTR --> part#"
 ((1)))

("SHIPMENT-ATTR --> supplier#"
 ((1)))

("SUPPLIER-ATTR --> status"
 ((1)))
```

```
("PART-ATTR --> weight"
 ((1)))

("SHIPMENT-ATTR --> quantity"
 ((1)))

("NUMBER-NP --> the-number-of SUPPLIER-NP"
 ((2 1)))

("NUMBER-NP --> the-number-of PART-NP"
 ((2 1)))

("NUMBER-NP --> the-number-of SHIPMENT-NP"
 ((2 1)))

("COMPUTABLE-NP --> COMPUTATIONS part-weight-computable
 num-of PART-NP"
 ((4 (1 2)) (1 2 3 4)))

("COMPUTABLE-NP --> COMPUTATIONS
 shipment-quantity-computable num-of SHIPMENT-NP"
 ((4 (1 2)) (1 2 3 4)))

("COMPUTATIONS --> the-average"
 ((1)))

("COMPUTATIONS --> the-sum"
 ((1)))

("COMPUTATIONS --> the-minimum"
 ((1)))

("COMPUTATIONS --> the-maximum"
 ((1)))

)
```

Lexicon File

4.5 You can build the lexicon file using either the NLMenu lexicographer or the Zmacs editor. The lexicon defines the items listed on the screen and relates them to items in the grammar. It also contains information on the target language meaning of the screen items, thus allowing the translator to transform the screen sentence to a string in the target language.

The lexicon should contain one entry for each terminal in the grammar. The specific information required for each lexical entry is:

- Category in the grammar — Terminals generated automatically if the Specify New Lexical Items option of the lexicographer is selected
- Item — The natural language form of input as it appears in the selection menu
- Source menu — The window pane in the selection screen to which the lexical item is mapped
- Translation — A lambda or atomic expression that the translator uses in building the target string. (See paragraph 4.6 for information on how to determine the contents of this field.)
- Help message — An optional description of the item

For more information on interactive construction of the lexicon, refer to Section 6.

The following is a sample lexicon file for a natural language interface to the Parts-and-Suppliers database. Each lexical entry is a list, and the entire lexicon is thus a list of lists. Notice that the items slot for each lexical entry must be a string. In this example, the translations also happen to be strings, although the precise format of the translations depends on the target language. Although the prompt for string experts is **quote**d while that for number experts is not, this is not an inconsistency because a string evaluates to itself in Lisp, while quoting a string also evaluates to the string. Finally, note that the translations are of two forms:

- Atomic translations have values that are passed up the parse tree and usually combined with other elements by means of a lambda expression.
- Lambda expressions serve to combine atomic expressions and other lambda expressions to produce the target string.

```
(

(SUPPLIERS  "suppliers"  NOUNS
 "lambda m m SUPPLIER)")

(PARTS  "parts"  NOUNS
 "lambda m m PART)")

(SHIPMENTS  "shipments"  NOUNS
 "lambda m m SHIPMENT)")

(WHERE-SUPPLIERS  "suppliers"  NOUNS
 "lambda n lambda o o SUPPLIER where n)")

(WHERE-PARTS  "parts"  NOUNS
 "lambda n lambda o o PART where n)")

(WHERE-SHIPMENTS  "shipments"  NOUNS
 "lambda n lambda o o SHIPMENT where n)")

(CHANGE-SUPPLIERS  "suppliers"  NOUNS
 "SUPPLIER")

(CHANGE-PARTS  "parts"  NOUNS
 "PART")

(CHANGE-SHIPMENTS  "shipments"  NOUNS
 "SHIPMENT")

(WHOSE-PART-CITY-IS  "whose part city is"  MODIFIERS
 "lambda a CITY in (a)")

(WHOSE-PART-COLOR-IS "whose color is"  MODIFIERS
 "lambda a COLOR in (a)")

(WHOSE-PART-NAME-IS "whose part name is"  MODIFIERS
 "lambda a NAME in (a)")
```

```
(WHOSE-PART-PART#-IS "whose part part# is"  MODIFIERS
 "lambda a PART# in (a)")

(WHOSE-SUPPLIER-CITY-IS "whose supplier city is"  MODIFIERS
 "lambda a CITY in (a)")

(WHOSE-SUPPLIER-NAME-IS "whose supplier name is"  MODIFIERS
 "lambda a NAME in (a)")

(WHOSE-SUPPLIER-SUPPLIER#-IS
 "whose supplier supplier# is"  MODIFIERS
 "lambda a SUPPLIER# in (a)")

(WHOSE-SHIPMENT-PART#-IS "whose shipment part# is"  MODIFIERS
 "lambda a PART# in (a)")

(WHOSE-SHIPMENT-SUPPLIER#-IS
 "whose shipment supplier# is"  MODIFIERS
 "lambda a SUPPLIER# in (a)")

(PART-CITY-EXPERT  "<specific part cities>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input CITY of PART"))))

(PART-COLOR-EXPERT  "<specific colors>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input COLOR of PART"))))

(PART-NAME-EXPERT  "<specific part names>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input NAME of PART"))))

(PART-PART#-EXPERT  "<specific part part#s>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input PART# of PART"))))

(SUPPLIER-CITY-EXPERT
 "<specific supplier cities>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input CITY of SUPPLIER"))))

(SUPPLIER-NAME-EXPERT
 "<specific supplier names>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input NAME of SUPPLIER"))))

(SUPPLIER-SUPPLIER#-EXPERT
 "<specific supplier supplier#s>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input SUPPLIER# of SUPPLIER"))))
```

```
(SHIPMENT-PART#-EXPERT
 "<specific shipment part#s>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input PART# of SHIPMENT"))))

(SHIPMENT-SUPPLIER#-EXPERT
 "<specific shipment supplier#s>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-STRING-EXPERT
 (QUOTE "Input SUPPLIER# of SHIPMENT"))))

(SUPPLIER-STATUS-EXPERT
 "<specific supplier status>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-NUMBER-EXPERT
 "Input STATUS of SUPPLIER")))

(WHOSE-SUPPLIER-STATUS-IS
 "whose supplier status is"  MODIFIERS
 "lambda b lambda c STATUS b c")

(PART-WEIGHT-EXPERT  "<specific part weight>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-NUMBER-EXPERT "Input WEIGHT of PART")))

(WHOSE-PART-WEIGHT-IS  "whose part weight is"  MODIFIERS
 "lambda b lambda c WEIGHT b c")

(SHIPMENT-QUANTITY-EXPERT
 "<specific shipment quantity>"  ATTRIBUTES
 (EXPERT (TYPE-ANY-NUMBER-EXPERT
 "Input QUANTITY of SHIPMENT")))

(WHOSE-SHIPMENT-QUANTITY-IS
 "whose shipment quantity is"  MODIFIERS
 "lambda b lambda c QUANTITY b c")

(PART-WEIGHT-COMPUTABLE  "weight"  FEATURES
 "WEIGHT")

(SHIPMENT-QUANTITY-COMPUTABLE "quantity"  FEATURES
 "QUANTITY")

(CITY  "city"  FEATURES
 "CITY")

(COLOR  "color"  FEATURES
 "COLOR")

(NAME  "name"  FEATURES
 "NAME")

(PART#  "part#  FEATURES
 "PART#")

(SUPPLIER#  "supplier#  FEATURES
 "SUPPLIER#")
```

```
(STATUS  "status"  FEATURES
 "STATUS")

(WEIGHT  "weight"  FEATURES
 "WEIGHT")

(QUANTITY  "quantity"  FEATURES
 "QUANTITY")

(THE-NEW-CITY-IS  "the new city is"  MODIFIERS
 "lambda z CITY = z")

(THE-NEW-COLOR-IS  "the new color is"  MODIFIERS
 "lambda z COLOR = z")

(THE-NEW-NAME-IS  "the new name is"  MODIFIERS
 "lambda z NAME = z")

(THE-NEW-PART#-IS  "the new part# is"  MODIFIERS
 "lambda z PART# = z")

(THE-NEW-SUPPLIER#-IS  "the new supplier# is" MODIFIERS
 "lambda z SUPPLIER# = z")

(THE-NEW-STATUS-IS  "the new status is"  MODIFIERS
 "lambda z STATUS = z")

(THE-NEW-WEIGHT-IS  "the new weight is"  MODIFIERS
 "lambda z WEIGHT = z")

(THE-NEW-QUANTITY-IS  "the new quantity is"  MODIFIERS
 "lambda z QUANTITY = z")

(SHIPMENT-which are shipments
 of-PART  "which are shipments of"  MODIFIERS
 "PART# in (select PART# from")

(SHIPMENT-which were shipped
 by-SUPPLIER  "which were shipped by"  MODIFIERS
 "SUPPLIER# in (select SUPPLIER# from")

(SUPPLIER-who ship-SHIPMENT  "who ship"  MODIFIERS
 "SUPPLIER# in (select SUPPLIER# from")

(who supply-SHIPMENT-SUPPLIER-PART  "who supply"  MODIFIERS
 "SUPPLIER# in (select SUPPLIER# from SHIPMENT where PART# in
 (select PART# from")

(which are supplied by-SHIPMENT-PART-SUPPLIER
 "which are supplied by" MODIFIERS
 "PART# in
 (select PART# from SHIPMENT where SUPPLIER# in
 (select SUPPLIER# from")
```

```
(EMBEDDED-SUPPLIERS  "suppliers"  NOUNS
 "lambda x x SUPPLIER))")

(EMBEDDED-PARTS  "parts"  NOUNS
 "lambda x x PART))")

(EMBEDDED-WHERE-SUPPLIERS  "suppliers"  NOUNS
 "lambda y lambda z z SUPPLIER where y))")

(EMBEDDED-WHERE-PARTS  "parts"  NOUNS
 "lambda y lambda z z PARTS where y))")

(TUPLE-AND  "and"  OPERATORS
 "lambda c lambda d c;d")

(ATTR-AND  "and"  OPERATORS
 "lambda f lambda g f,g")

(S-AND  "and"  OPERATORS
 "lambda h lambda i (h and i)")

(UPDATE-AND  "and"  OPERATORS
 "lambda x lambda y x, y")

(S-OR  "or"  OPERATORS
 "lambda j lambda k (j or k)")

(LEFT-BRACKET  "("  OPERATORS
 NIL)

(RIGHT BRACKET ")"  OPERATORS
 NIL)

(BETWEEN-AND  "and"  OPERATORS
 "lambda z lambda y z and y")

(BETWEEN  "between"  COMPARISONS
 "between")

(GREATER-THAN  "greater than"  COMPARISONS
 ">")

(LESS-THAN  "less than"  COMPARISONS
 "<")

(GREATER-OR-EQUAL-TO  "greater than or equal to"  COMPARISONS
 ">=")

(LESS-THAN-OR-EQUAL-TO  "less than or equal to"  COMPARISONS
 "<=")

(EQUAL-TO  "equal to"  COMPARISONS
 "=")
```

```
(NOT-EQUAL-TO  "not equal to"  COMPARISONS
 "≠")

(FOR  "of"  OPERATORS
 NIL)

(THE-AVERAGE  "the average"  OPERATORS
 "lambda p (select avg(p) from")

(THE-SUM  "the total"  OPERATORS
 "lambda q (select sum(q) from")

(THE-MAXIMUM  "the maximum"  OPERATORS
 "lambda r (select max(r) from")

(THE-MINIMUM  "the minimum"  OPERATORS
 "lambda s (select min(s) from")

(NUM-OF  "of"  OPERATORS
 NIL)

(FIND-REL  "Find"  ACTIONS
 "(Select * from")

(FIND-ATTR-REL  "Find"  ACTIONS
 "lambda l (Select l from")

(FIND-COMP  "Find"  ACTIONS
 NIL)

(THE-NUMBER-OF  "the number of"  OPERATORS
 "(select count(*) from")

(INSERT  "Insert"  ACTIONS
 "lambda l (l)")

(NEW-SUPPLIER-TUPLE  "<a new supplier>"  NOUNS
 (EXPERT (SQL-INSERTION-EXPERT (QUOTE SUPPLIER))))

(NEW-PART-TUPLE  "<a new part>"  NOUNS
 (EXPERT (SQL-INSERTION-EXPERT (QUOTE PART))))

(NEW-SHIPMENT-TUPLE  "<a new shipment>"  NOUNS
 (EXPERT (SQL-INSERTION-EXPERT (QUOTE SHIPMENT))))

(DELETE-REL  "Delete"  ACTIONS
 "(Delete from")

(CHANGE  "Change"  ACTIONS
 "lambda x lambda y (Update x set y)")

(SO-THAT  "so that"  OPERATORS
 "lambda x lambda y lambda z (Update x set y where z)")

)
```

Translation Process

4.6 Once you build a sentence and ask the system to begin executing, the translator takes the parse tree built by the parser and constructs the target string using the information in the grammar and lexicon.

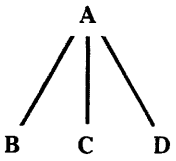
The translation process works as follows:

1. The translator traverses the parse tree, finding the innermost subtree. This subtree consists of a mother and one or more terminal daughters.

2. The translator then gets the translation list for the rule expansion represented by this subtree. For example, if the rule

   ```
   ("A --> B (C D)" ((1)) ((2 1 3)))
   ```

   produces the subtree

   the translator determines that the proper translation list for this expansion is `((2 1 3))`.

3. The translator gets the translations for *B*, *C*, and *D* from their corresponding lexical entries, applies the lambda expression representing *C*'s translation to *B*'s translation to produce a new lambda expression, and then applies that new expression to *D*'s translation. The resulting expression represents *A*'s translation and is passed up the tree as this process is repeated. For example, possible translations are:

Category	***Category No.***	***Translation***
B	1	"wheels"
C	2	"lambda x lambda y x,y"
D	3	"axles"

4. The lambda expressions used in NLMenu translations are actually lambda calculus functions in which the first variable following the occurrence of a lambda is a formal parameter, and the rest of the expression represents the body of the function. Applying a function to an argument produces the body of the function with the argument or actual parameter substituted for all occurrences of the formal parameter. Thus the translation function `(2 1 3)` produces

   ("lambda x lambda y x,y" "wheels" "axles")

   or (applying 2 to 1)

   ("lambda y wheels,y" "axles")

   or (applying the result to 3)

   "wheels,axles"

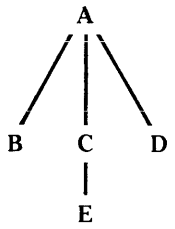
   which serves as *A*'s translation.

5. The translator moves up the tree to find the mother of the root of the current subtree.

6. The translator repeats steps 2 through 4 until it processes the root of the parse tree.

Some special cases that can appear in an application need explanation:

- If at any time the mother of a subtree has subtrees under it that the translator has not yet processed, they are worked on before that mother is processed. For example, if the subtree with *A* for a mother is



  then the subtree with *C* for a mother is processed before the whole.

- A translation list of length 1 always has an atomic translation that is simply passed up the tree unchanged. The translation list in such an instance is `((1))`.

Examples of the Translation Process

4.6.1 The following examples show the translation process for the natural language query:

Change shipments so that the new quantity is 300.

The applicable grammar rules are:

1. ("S → change change-shipments
   (SHIPMENT-MOD-CONSTRUCT) so-that SHIPMENT-UPDATES"
   ((1 2 5) (1 2 4 5)) ((4 2 5 3) (1 2 3 4 5)))

2. ("SHIPMENT-UPDATES → the-new-quantity-is
   shipment-quantity-expert (update-and SHIPMENT-UPDATES)"
   ((1 2)) ((3 (1 2) 4)))

The applicable lexicon entries are:

1. (CHANGE "change" ACTIONS
   "lambda x lambda y (Update x set y)")

2. (CHANGE-SHIPMENTS "shipments" NOUNS "shipment")

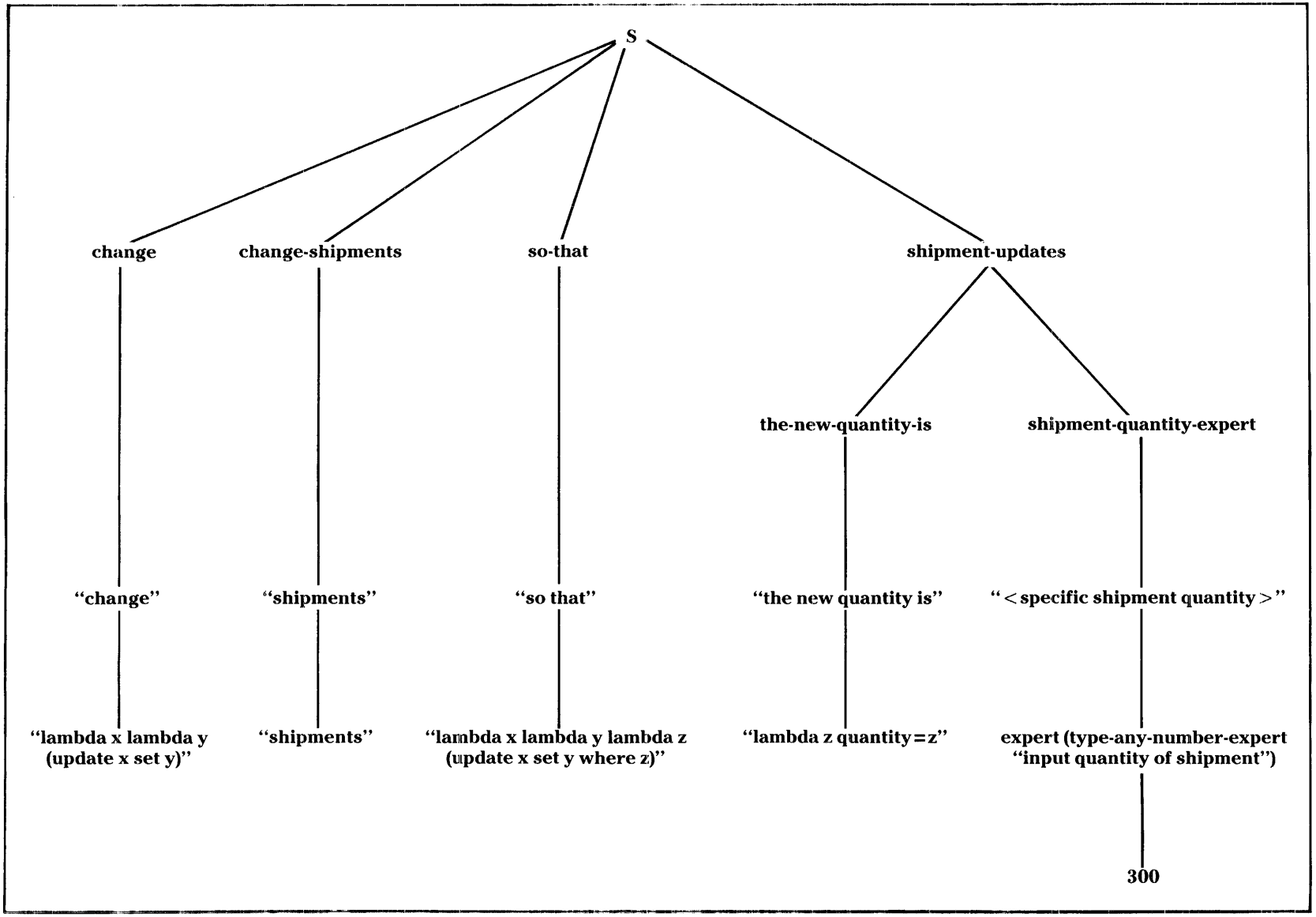
3. (SO-THAT "so that" OPERATORS
   "lambda x lambda y lambda z (Update x set y where z)")

4. (THE-NEW-QUANTITY-IS "the new quantity is" MODIFIERS
   "lambda z QUANTITY = z")

5. (SHIPMENT-QUANTITY-EXPERT
   "< specific shipment quantity> "
   ATTRIBUTES (EXPERT (TYPE-ANY-NUMBER-EXPERT
   "Input QUANTITY of SHIPMENT")))

Figure 4-1 shows the semantic parse tree for the query. The translation portion of the lexical entries for the terminals is included as well as the appropriate translation functions.

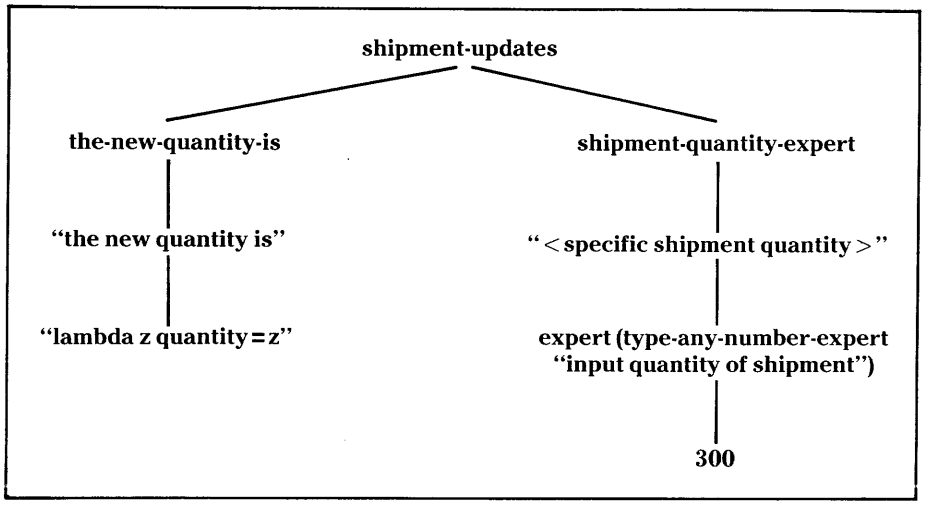
Figure 4-1 Semantic Parse Tree1



The translation proceeds as follows:

1. The innermost subtree is determined to be the subtree with the *shipment-updates* node as root. This subtree is shown in Figure 4-2.

Figure 4-2 **Partial Semantic Parse Tree1**



2. This subtree represents an application of rule 2, so the appropriate translation function is identified as `(1 2)` since the optional element in rule 2 is not used in producing the parse.

3. The following translations are then extracted from the lexicon:

Category	*Category Number*	*Translation*
THE-NEW-QUANTITY-IS	1	"lambda z QUANTITY = z"
SHIPMENT-QUANTITY-EXPERT	2	"300"

Observe that the translation for the *SHIPMENT-QUANTITY-EXPERT* is the value resulting from invocation of the expert when you formulate the input. Thus, the translation function `(1 2)` produces

("lambda z QUANTITY = z" "300")

or (applying 1 to 2)

"QUANTITY = 300"

4. The translator now moves up the tree to find the mother of the root of this subtree. This yields a new subtree, in this case the original parse tree of Figure 4-1. The result obtained in step 3 is used as the translation for the rightmost daughter of this new subtree.

5. Steps 2 through 4 are now repeated for the new subtree. As the new subtree represents an application of rule 1 without the optional element, the appropriate translation function is `(1 2 5)`.

6. The following translations are now used:

Category	***Category Number***	***Translation***
CHANGE	1	"lambda x lambda y (Update x set y)"
CHANGE-SHIPMENTS	2	"shipment"
SHIPMENT-UPDATES	5	"QUANTITY = 300"

Observe that the last entry in this table represents the result obtained in step 3. The translation function `(1 2 5)` produces

("lambda x lambda y (Update x set y)"
"shipment" "QUANTITY = 300")

or (applying 1 to 2)

("lambda y (Update shipment set y)" "QUANTITY = 300")

or (applying the result to 5)

"(Update shipment set QUANTITY = 300)".

When the final output of the translator is a string, the system does some postprocessing by removing the double quotes and outer parentheses and appending a semicolon to the end of the translation. This results in the following SQL query that is the translation of the original natural language query:

Update shipment set QUANTITY = 300;

Consider the more complex process to produce the translation for the following natural language query:

Find city and name of parts.

The applicable grammar rules are:

1. ("S → find-attr-rel PART-ATTR-CONSTR for PART-NP"
   ((4 (1 2)) (1 2 3 4)))

2. ("PART-ATTR-CONSTR → PART-ATTR (attr-and
   PART-ATTR-CONSTR)"
   ((1)) (2 1 3)))

3. ("PART-NP → MODIFIABLE-PART-NP"
   ((1)))

4. ("MODIFIABLE-PART-NP → parts"
   ((1)))

5. ("PART-ATTR → city"
   ((1)))

6. ("PART-ATTR → name"
   ((1)))

The applicable lexicon entries are:

1. (PARTS "parts" NOUNS "lambda m m PART)")

2. (CITY "city" FEATURES "city")

3. (NAME "name" FEATURES "name")

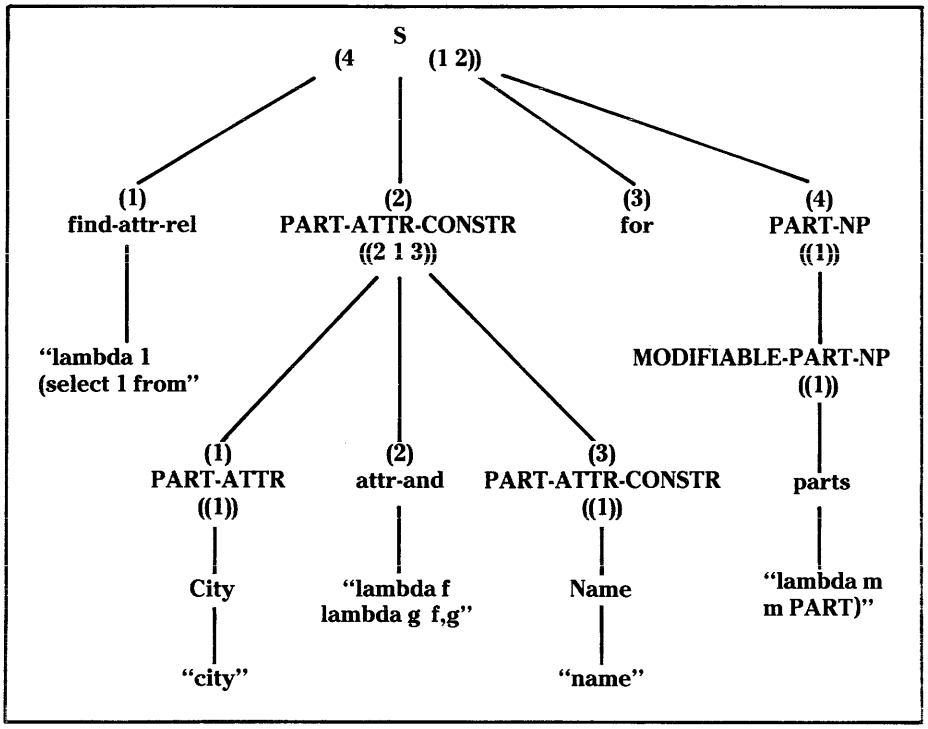
4. (ATTR-AND "and" OPERATORS "lambda f lambda g f,g")

5. (FOR "of" OPERATORS nil)

6. (FIND-ATTR-REL "find" ACTIONS "lambda l (Select l from")

Figure 4-3 shows the semantic parse tree for the query. The translation portion of lexical entries for the terminals are included as well as the appropriate translation functions.

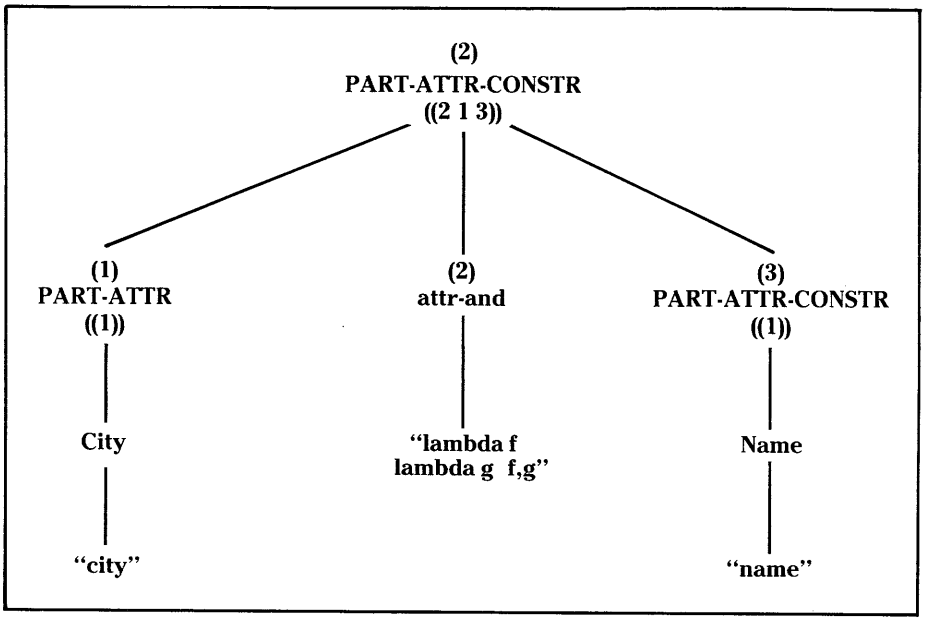
Figure 4-3 **Semantic Parse Tree2**



The translation process proceeds as follows:

1. The innermost subtree is determined to be the subtree with the *PART-ATTR-CONSTR* node as root and `((2 1 3))` as its translation function. Figure 4-4 shows this subtree.

Figure 4-4 **Partial Semantic Parse Tree2**



2. This subtree represents an application of rule 2, so the appropriate translation function is identified as `(2 1 3)` since the optional elements in rule 2 are used in producing the parse.

3. The following translations are then extracted from the lexicon:

Category	*Category Number*	*Translation*
City	1	"city"
attr-and	2	"lambda f lambda g f,g"
Name	3	"name"

The translation function `(2 1 3)` produces

("lambda f lambda g f,g" "city" "name")

or (applying 2 to 1)

("lambda g city,g" "name")

or (applying the result to 3)

"city,name"

4. The translator now moves up the tree to find the mother of the root of this subtree. This yields a new subtree, in this case the original parse tree of Figure 4-3. The result obtained in step 3 is used as the translation for the second daughter of this new subtree.

5. Steps 2 through 4 are now repeated for the new subtree. Because the new subtree represents an application of rule 1, the translation function is `(4 (1 2))`.

6. The following translations are then used:

Category	*Category Number*	*Translation*
find-attr-rel	1	"lambda l (Select l from"
PART-ATTR-CONSTR	2	"city,name"
for	3	nil
PART-NP	4	"lambda m m PART)"

   Observe that the second entry in the preceding table represents the result obtained in step 3. The translation function `(1 2)` produces:

   ("lambda l (Select l from" "city,name")

   or (applying 1 to 2)

   "(Select city,name from"

7. That output then becomes input for application of the translation function `(4 (1 2))` and produces

   ("lambda m m PART)" "Select city,name from")

   or (applying 4 to the result)

   "(Select city,name from PART)".

When the final output of the translator is a string, the system does some postprocessing by removing the double quotes and outer parentheses and appending a semicolon to the end of the translation. This results in the following SQL query that is the translation of the original natural language query:

Select city, name from PART;

Summary of Translator Algorithm

4.6.2 The translation process follows these steps:

1. Find the lowest, rightmost subtree in the parse tree.
2. The rule expansion that this subtree represents has a corresponding translation list. Perform the text substitution as prescribed in the semantic substitution list. (If it contains an unprocessed subtree, perform text substitution on the unprocessed subtree before the whole.)
3. Move up the tree one rule. The root of this subtree is the mother of the previous root.
4. Repeat steps 2-4 until the entire subtree is processed.

Examples of Developing Translations

4.7 These paragraphs contain examples of how you use a set of natural language sentences and their target language translations. The examples focus on developing the translation functions for an already established grammar and lexicon. In each example, there are several ways to develop the translations, and these are examined in detail.

In the following discussion, the grammar and lexicon are repeated as the translation information evolves. Entries that change in each are flagged by the symbol >>>.

Example 1 **4.7.1** In the first example, the source language is English; the target language is the Dow Jones News/Retrieval® query language.

During this example, you want to ask the following questions:

Sentence	*Target String*
1. What is the current quote for Company A?	,STA
2. What is the current quote for Company B?	,STB
3. What is the current quote for Company A on the New York exchange?	,1STA
4. What is the current quote for Company A and Company B?	,STA STB
5. What are the headlines concerning Company A?	.STA 01
6. What are the headlines in the Wall Street Journal®?	//WSJ

STA and *STB*, which appear in the target strings, are the stock symbols for the companies *Company A* and *Company B*. Assume the grammar to be as follows:

S → current-quote STOCK
S → headlines HEAD-LIST
STOCK → STOCK and STOCK
STOCK → NY-COMPANY
STOCK → NY-COMPANY ny-exchange
NY-COMPANY → company-a
NY-COMPANY → company-b
HEAD-LIST → in-wsj
HEAD-LIST → concerning-the-company NY-COMPANY

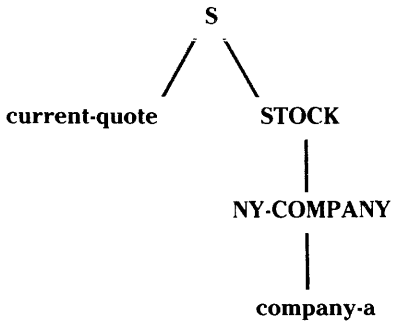
Dow Jones News/Retrieval and the Wall Street Journal are registered trademarks of Dow Jones & Company, Inc.

Although this grammar produces sentences in addition to those listed above, all sentences are within the scope of the application. This example develops the translation information based on the six sentences. The terminals in the grammar, and thus the list of entries required in the lexicon, are in lowercase; the nonterminals are all in uppercase. This is merely for clarity and not required by the system. A first pass at construction of the lexicon might be:

Terminal	***Item***	***Menu***	***Translation***
(			
(current-quote	"What is the current quote for"	menu1	nil)
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	nil)
(company-a	"Company A"	menu2	nil)
(company-b	"Company B"	menu2	nil)
(in-wsj	"in the Wall Street Journal"	menu4	nil)
(concerning-the-company	"concerning the company"	menu4	nil)
(and	"and"	menu5	nil)
)			

The translation field for all lexical entries is now **nil**, but you fill in translations as the example proceeds.

Now you must begin work on the translation lists for the grammar and the translations for the lexicon. It is helpful at this point to draw a parse tree of the query on which you are working. Start with sentence 1: *What is the current quote for Company A?* The parse tree is



while the target string is

,STA

Whenever you ask about current quotes, the query is preceded by a comma followed by the company's stock symbol. To start construction of translation lists and lexicon translations, assign , to be the translation in the lexicon for *current-quote*, and the company's stock symbol *STA* to *company-a*.

You can see where all the pieces of the target string come from; now you must put them together to form the whole. It is obvious that what you need to do is pass *STA* up the tree to be combined finally with the comma. The way to pass an item up the tree with no modification is to have a translation function consisting of the single integer `((1))`. Now you can add translation functions to a few of the grammar rules (they have been reformatted to conform with input grammar format), leaving empty the translation lists not yet treated. Observe that expansion lists are now appended to each rule, using the simple ordering implied.

```
        (
        ("S → current-quote STOCK" ( ) (1 2))
        ("S → headlines HEAD-LIST" ( ) (1 2))
        ("STOCK → STOCK and STOCK" ( ) (1 2 3))
> > > ("STOCK → NY-COMPANY" ((1)))
        ("STOCK → NY-COMPANY ny-exchange" ( ) (1 2))
> > > ("NY-COMPANY → company-a" ((1)))
        ("NY-COMPANY → company-b" ( ) (1))
        ("HEAD-LIST → in-wsj" ( ) (1))
        ("HEAD-LIST → concerning-the-company NY-COMPANY" ( ) (1 2))
        )
```

Now that you have passed the values for *current-quote* and *STOCK* up the parse tree, you must next concatenate the two strings that you have to work with: , and *STA*. Accomplish this by transforming one string into a lambda calculus function that can be applied to the other string. The best choice would be to convert the comma to a lambda expression.

Define the translation as follows:

1. The value added to/inserted into another is always atomic.

2. The value to which you add something else has a lambda expression as its translation. The body of the lambda expression depends on the number of items to be added to the element. In this instance, there is only one. The lambda expression should be as follows:

   lambda x ,x

3. The translation function in the grammar must take the lambda expression and apply it to the atomic expression as its argument. Since evaluation takes place left to right unless overridden by parentheses, the translation function in this example is `(1 2)`.

4. When applied, the result of this translation function is as follows:

   ("lambda x ,x" "STA") → ,STA

Since the translation function for the root rule should be `(1 2)`, the lexicon now looks like:

Terminal	*Item*	*Menu*	*Translation*
(			
(current-quote	"What is the current quote for"	menu1	"lambda x ,x")
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	nil)
(company-a	"Company A"	menu2	"STA")
(company-b	"Company B"	menu2	nil)
(in-wsj	"in the Wall Street Journal"	menu4	nil)
(concerning-the-company	"concerning the company"	menu4	nil)
(and	"and"	menu5	nil)
)			

Now move on to the second sentence: *What is the current quote for Company B?* Since it is nearly identical to the first, add the same translation function to the grammar rule

NY-COMPANY → company-b

as you used for

NY-COMPANY → company-a

and insert *Company B*'s stock symbol in the translation field for its lexical entry. You now have the following grammar:

```
        (
>>> ("S → current-quote STOCK" ((1 2)))
        ("S → headlines HEAD-LIST" ( ) (1 2))
        ("STOCK → STOCK and STOCK" ( ) (1 2 3))
        ("STOCK → NY-COMPANY" ((1)))
        ("STOCK → NY-COMPANY ny-exchange" ( ) (1 2))
        ("NY-COMPANY → company-a" ((1)))
>>> ("NY-COMPANY → company-b" ((1)))
        ("HEAD-LIST → in-wsj" ( ) (1))
        ("HEAD-LIST → concerning-the-company NY-COMPANY" ( ) (1 2))
        )
```

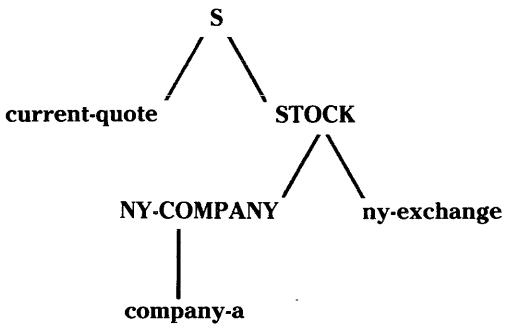
and the following lexicon:

Terminal	*Item*	*Menu*	*Translation*
(			
(current-quote	"What is the current quote for"	menu1	"lambda x ,x")
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	nil)
(company-a	"Company A"	menu2	"STA")
(company-b	"Company B"	menu2	"STB")
(in-wsj	"in the Wall Street Journal"	menu4	nil)
(concerning-the-company	"concerning the company"	menu4	nil)
(and	"and"	menu5	nil)
)			

Now on to the third sentence: *What is the current quote for Company A on the New York Exchange?* The target string is

,1STA

and here is the parse tree:



Focusing on the rightmost subtree, you need to be able to concatenate *1* and *STA* at the mother node *STOCK*. Since it would not be good to modify the translation for *company-a*, transform *1* to a lambda expression. For the translation function of the grammar rule, the lambda expression is daughter2, its argument is daughter1, and so the rule is: `(2 1)`. The lexical translation for daughter1 is atomic, while that for daughter2 is the lambda expression:

"lambda x 1x".

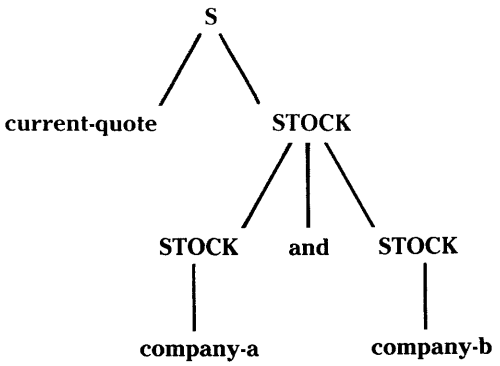
Updating the grammar, you now have:

```
        (
        ("S → current-quote STOCK" ((1 2)))
        ("S → headlines HEAD-LIST" ( ) (1 2))
        ("STOCK → STOCK and STOCK" ( ) (1 2 3))
        ("STOCK → NY-COMPANY" ((1)))
>>>     ("STOCK → NY-COMPANY ny-exchange" ((2 1))
        ("NY-COMPANY → company-a" ((1)))
        ("NY-COMPANY → company-b" ((1)))
        ("HEAD-LIST → in-wsj" ( ) (1))
        ("HEAD-LIST → concerning-the-company NY-COMPANY" ( ) (1 2))
        )
```

and the following lexicon:

Terminal	***Item***	***Menu***	***Translation***
(			
(current-quote	"What is the current quote for"	menu1	"lambda x ,x")
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	"lambda x 1x")
(company-a	"Company A"	menu2	"STA")
(company-b	"Company B"	menu2	"STB")
(in-wsj	"in the Wall Street Journal"	menu4	nil)
(concerning-the-company	"concerning the company"	menu4	nil)
(and	"and"	menu5	nil)
)			

Here is the parse tree for sentence 4, *What is the current quote for Company A and Company B?*:



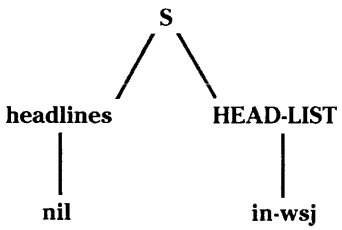
One way to implement it is:

```
        (
        ("S → current-quote STOCK" ((1 2)))
        ("S → headlines HEAD-LIST" ( ) (1 2))
> > > ("STOCK → STOCK and STOCK" ((2 1 3))
        ("STOCK → NY-COMPANY" ((1)))
        ("STOCK → NY-COMPANY ny-exchange" ((2 1))
        ("NY-COMPANY → company-a" ((1)))
        ("NY-COMPANY → company-b" ((1)))
        ("HEAD-LIST → in-wsj" ( ) (1))
        ("HEAD-LIST → concerning-the-company NY-COMPANY" ( ) (1 2))
        )
```

with the following lexicon:

Terminal	***Item***	***Menu***	***Translation***
(			
(current-quote	"What is the current quote for"	menu1	"lambda x ,x")
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	"lambda x 1x")
(company-a	"Company A"	menu2	"STA")
(company-b	"Company B"	menu2	"STB")
(in-wsj	"in the Wall Street Journal"	menu4	nil)
(concerning-the-company	"concerning the company"	menu4	nil)
(and	"and"	menu5	"lambda x lambda y ,x y")
)			

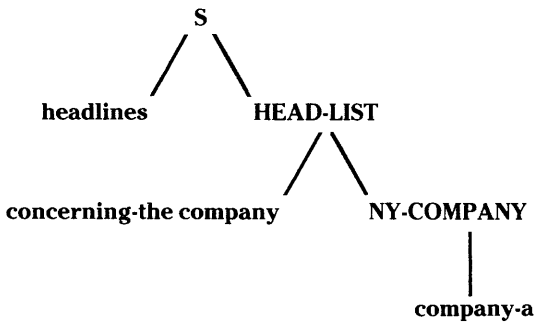
In sentences 5 and 6, building translations is a little different. When asking about headline information, there is no common prefix such as the comma for current quotes. This signifies that the lexicon translation field for headlines should be empty, that is, **nil**. Therefore, all construction of the target string should be done at a lower level of the parse trees and merely passed up at the root node. If implemented this way, sentence 6 becomes quite easy to work with: assign the atom *//WSJ* to the lexical translation of the terminal in *in-wsj* and pass it up via single integer translation lists. The parse tree for sentence 6, *What are the headlines in the Wall Street Journal?*, is:



The target string for sentence 6 is:

//WSJ

Now work with sentence 5: *What are the headlines concerning Company A?*. The parse tree is



and the target string is

.STA 01

You already have worked with the lower rightmost subtree; assume that it stands as is. You have also already assigned a translation list for the uppermost subtree, which is simply to pass up the string from *HEAD-LIST*'s subtree. So, you must create a function for the terminal *concerning-the-company* which will allow inserting the stock symbol into the appropriate place. That lexical entry for *concerning the company* is:

"lambda x (.x 01)"

The translation function for the grammar for the second *HEAD-LIST* rule is `(1 2)`.

The final grammar for the set of example sentences is:

```
        (
        ("S → current-quote STOCK" ((1 2)))
> > > ("S → headlines HEAD-LIST" ((2) (1 2)))
        ("STOCK → STOCK and STOCK" ((2 1 3)))
        ("STOCK → NY-COMPANY" ((1)))
        ("STOCK → NY-COMPANY ny-exchange" ((2 1)))
        ("NY-COMPANY → company-a" ((1)))
        ("NY-COMPANY → company-b" ((1)))
> > > ("HEAD-LIST → in-wsj" ((1)))
> > > ("HEAD-LIST → concerning-the-company NY-COMPANY" ((1 2)))
        )
```

with the following lexicon:

Terminal	*Item*	*Menu*	*Translation*
(			
(current-quote	"What is the current quote for"	menu1	"lambda x ,x")
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	"lambda x 1x")
(company-a	"Company A"	menu2	"STA")
(company-b	"Company B"	menu2	"STB")
(in-wsj	"in the Wall Street Journal"	menu4	"//WSJ")
(concerning-the-company	"concerning the company"	menu4	"lambda x .x 01")
(and	"and"	menu5	"lambda x lambda y ,x y")
)			

Now consider how alternate translation schemes affect the translation process. You could have made alternate decisions with the rules defining *NY-COMPANY*. Since those rules are used in the parse of several sentences, the choice could either resolve problems or cause them, depending on its interaction with other items. For example, when working on the first sentence, you might choose to build a lambda expression around the stock symbol (*STA*, *STB*, and so on). Such a function would read:

lambda x x STA

However, similar but separate functions would be necessary for *STB* and any other stock symbol. Therefore, your choice would complicate the translation process. Consider further how such a choice would impact another type of sentence that refers to a stock symbol. Queries about stock company headlines could not use the stock symbol rule as redefined here. There would be no way to construct the translations for headline queries using the expressions

"lambda x x STA"

or

"lambda x x STB"

previously defined for stock symbols *A* and *B*, respectively. Ambiguity arises if you allow any item to have more than one entry in the lexicon.

To resolve this dilemma, you would have to rewrite the grammar in an ad hoc manner to differentiate a stock symbol in one context from a stock symbol in another context. Even then, you would not eliminate the need for separate versions of the ad hoc function for each possible stock symbol in the target language. Note then the following guideline:

Lexicon Writing Hint: When faced with a choice of translation schemes, you usually convert the terminal's translation that is involved in the fewest parses to the lambda expression.

Deciding which translations should be lambda expressions can be crucial. For example, if you want to modify the sentences to allow the interface user to type in a given stock symbol, change the grammar to read as follows:

```
        (
>>> ("S → current-quote STOCK" ((2 1)))
        ("S → headlines HEAD-LIST" ((2) (1 2)))
        ("STOCK → STOCK and STOCK" ((2 1 3)))
        ("STOCK → NY-COMPANY" ((1)))
        ("STOCK → NY-COMPANY ny-exchange" ((2 1)))
        ("NY-COMPANY → company-a" ((1)))
        ("NY-COMPANY → company-b" ((1)))
>>> ("NY-COMPANY → <stock expert>" ((1)))
        ("HEAD-LIST → in-wsj" ((1)))
        ("HEAD-LIST → concerning-the-company NY-COMPANY" ((1 2)))
        )
```

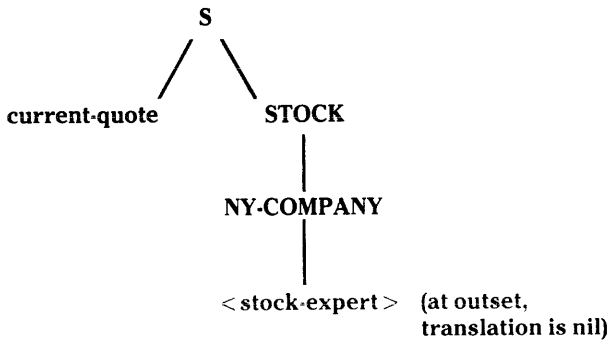
with the following lexicon:

Terminal	*Item*	*Menu*	*Translation*
(			
(current-quote	"What is the current quote for"	menu1	",")
(headlines	"What are the headlines"	menu1	nil)
(ny-exchange	"on the New York exchange"	menu3	"lambda x 1x")
(company-a	"Company A"	menu2	"lambda x xSTA")
(company-b	"Company B"	menu2	"lambda x xSTB")
(<stock expert>	"<stock expert>"	menu2	**********
(in-wsj	"in the Wall Street Journal"	menu4	"//WSJ")
(concerning-the-company	"concerning the company"	menu4	"lambda x .x 01")
(and	"and"	menu5	"lambda x lambda y ,x y")
)			

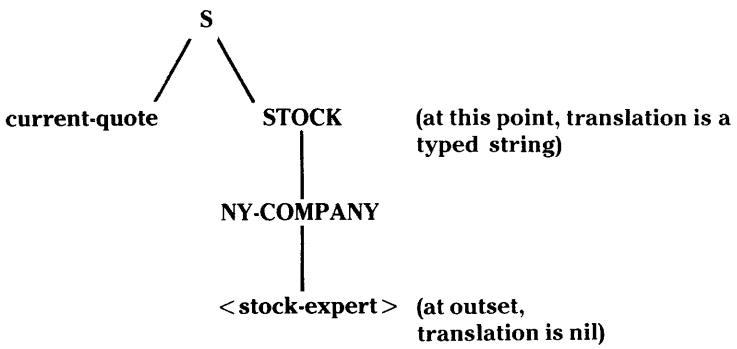
This version of the grammar and lexicon allows the interface user to ask questions such as:

What is the current quote for < stock-expert> ?
What are the headlines concerning the company < stock-expert> ?

You would, however, run into translation difficulties by converting the strings for *Company A* and *Company B* into lambda expressions. Such a conversion also forces you to change the translation for *current-quote* to an atomic symbol. Because the translations produced by invoking experts are also atomic, the translations produced by *current-quote* and the *< stock-expert>* cannot be combined, and, thus, the preceding query cannot be expressed. For example, the parse tree for the sentence *What is the current quote for < stock-expert>* follows:



Starting at the rightmost subtree, the translation for the lexical entry <*stock-expert*> is **nil**. The translator will take the string you have typed and use that for the translation. The typed string is passed up the tree to the nonterminal *STOCK*, giving



Remember that the translation in the lexicon for *current-quote* is now , because you made the translations for *Company A* and *Company B* lambda expressions. You then change the translation list for the root rule accordingly because the second daughter would contain a lambda expression translation rather than the first. When the translator comes to the root rule, it expects the translation of *daughter2* to be a lambda expression and fails.

Choosing an Appropriate Translation Scheme

4.7.2 As discussed earlier, the translation process allows for different constructions of the target string. You can experiment with the example and produce a different scheme than above. Which one is better? Here are points to consider when evaluating competing schemes:

*Which scheme is more general?

When you expanded the list of sentences to include an expert, one implementation caused you to recreate the translation information for a portion of the grammar and lexicon, and one implementation did not; the one that did not is more general. As in any application program, the maintainability of the grammar and lexicon in an NLMenu system is an important consideration.

*Which scheme is simpler?

One way to implement the translation information for sentence 6, *What are the headlines in the Wall Street Journal?*, is to use the following fragments of grammar and lexicon:

("S → headlines HEAD-LIST ((1 2)))
('HEAD-LIST → in-wsj ((1)))

Terminal	*Item*	*Menu*	*Translation*
(			
.			
.			
.			
(headlines	"What are the headlines"	menu1	"lambda x x")
(in-wsj	"in the Wall Street Journal"	menu4	"//WSJ")
.			
.			
.			
)			

While this translation scheme would produce a valid translation, it is not as simple as the scheme used in the example in paragraph 4.7.1. Another processing step is involved for the translator, and more storage is used in recording the necessary information.

Meeting both criteria may be impossible in some instances. You must weigh the options and use the translation scheme that best meets the needs of the application.

Developing Screen Configuration Descriptions

4.8 The NLMenu window is implemented as a *constraint frame*, a window divided into subwindows using the hierarchical structure of the window system. The subwindows are called *panes*. Constraint frames adjust the shape of their panes automatically as their own shape changes.

To make a constraint frame, specify the configuration of panes within the frame by means of a list structure to represent the layout. The format of this list structure is the constraint language described in detail in the *Explorer Window System Reference*. Note that you can specify panes to occupy a certain percentage of the available space within a row or column in the constraint frame. This approach is used in the default screen configuration as shown in Figure 4-5.

Figure 4-5 Default Screen Configuration

```
(defvar *nl-default-panes*
        ((actions tv:kbd-command-menu-pane
                  :label "Commands" :item-list (""))
         (nouns tv:kbd-command-menu-pane
                  :label "Nouns" :item-list (""))
         (modifiers tv:kbd-command-menu-pane
                  :label "Modifiers" :item-list (""))
         (attributes tv:kbd-command-menu-pane
                  :label "Experts" :item-list (""))
         (comparisons tv:kbd-command-menu-pane
                  :label "Comparisons" :item-list (""))
         (features tv:kbd-command-menu-pane
                  :label "Attributes" :item-list (""))
         (operators tv:kbd-command-menu-pane
                  :label "Connectors" :item-list (""))
         (commands tv:kbd-command-menu-pane
                  :label "System Commands" :item-list (""))
         (input sensitive-window-pane :label nil)
         (logo tv:window-pane :label nil)
         (display scrollable-output-window
                  :label "Nlmenu Display Window")))

(defvar *nl-default-constraints*
        ((main . ((logo menu-strip commands input display)
                  ((logo .025)
                   (menu-strip :horizontal (.49)
                               (first-strip second-strip third-strip modifiers)
                               ((first-strip :vertical (.17)
                                             (actions features)
                                             ((actions .14)
                                              (features .86)))
                                (second-strip :vertical (.26)
                                              (nouns comparisons)
                                              ((nouns .5)
                                               (comparisons .5)))
                                (third-strip :vertical (.25)
                                             (attributes operators)
                                             ((attributes .75)
                                              (operators .25)))
                                (modifiers .32)))
                   (commands .075)
                   (input .08)
                   (display .33))))))
```

Two lists are provided—one indicating which panes are to be part of the constraint frame and a second indicating what constraints are to be imposed on the panes. In the default screen configuration, each pane entry is of the form:

(*panename pane-flavor* **:label** *pane-label* **:item-list** (""))

The pane flavor is the flavor of the particular pane. For instance, the flavor of the actions pane in Figure 4-5 is `tv: kbd-command-menu-pane`, which is a system-defined flavor. The pane label is the user-specified string that appears at the top of each pane. The optional item list for selection panes is added later by the system, which accesses the appropriate menu association lists to obtain the necessary information.

The complete description of how to construct constraint frames (see Figure 4-5) using the full capabilities of the constraint frame language is rather complicated. Refer to the *Explorer Window System Reference* for details. As an example, however, consider the default screen configuration. It contains just one window configuration, called *MAIN*. In this configuration, five panes appear from top to bottom:

- *LOGO*
- *MENU-STRIP*
- *COMMANDS*
- *INPUT*
- *DISPLAY*

The *LOGO* pane consumes 2.5 percent of the available space, the *MENU-STRIP* pane consumes 49 percent of the remaining space, and so on. The *MENU-STRIP* pane is horizontally subdivided into four columns:

- *FIRST-STRIP*
- *SECOND-STRIP*
- *THIRD-STRIP*
- *MODIFIERS*.

Notice the use of dummy names (for example, *FIRST-STRIP*) to name a pane that is to be further subdivided into panes mentioned in the panes description. Finally, three of these columns are vertically subdivided into rows. For instance, the *FIRST-STRIP* pane consists of an *ACTIONS* pane, consuming 14 percent of the column, on top of a *FEATURES* pane that consumes the remainder of the column.

Implementation of Experts

4.9 The NLMenu system uses coded *interaction experts* to avoid lexical ambiguity when specific input parameters are to appear in a sentence or query. This eliminates the need for storing all values of parameters in the lexicon. Moreover, you can use experts as support in specifying legal ranges of values for specific parameters.

In the NLMenu system, you specify experts in the lexicon as segments of code that the system executes as you are constructing the input sentence or query. Lexical entries that invoke experts have translations of a special form `(expert ( code ))`, that is, a list beginning with a special tag and followed by some code, where *code* consists of an expert-type code and the expert window title. A simple type-in expert might appear in the lexical entry for the term `<specific supplier city>` as follows:

```
(supplier-city-expert "<specific supplier city>" NOUNS
(EXPERT (typein-expert "Type a supplier city:"))
optional-help-text)
```

Note that lexical entries have the form

(category item source-menu translation optional-help-text)

and are discussed in detail in paragraph 4.5.

During parsing, when you select a non-expert lexical item from some menu, the item and its translation are added to the parse verbatim. For experts, however, the system executes the code specified in the translation, often popping up an interaction window or menu designed to allow you to specify one or more values. In the example above, when you select the item `<specific supplier city>`, the code `(typein-expert "Type a supplier city:")` pops up a type-in window to allow you to specify a city, for instance New York. *New York* is then used as the next phrase in the input sentence and is also treated as the translation of `<specific supplier city>`.

In general, experts obey certain conventions. First, since it is possible to rub out expert values, expert code should not have any irreversible side-effects. Second, experts typically support an Abort option as well as value selection so that you can escape from experts without successful termination. Most importantly, experts return either **nil**, indicating that the expert was aborted, or a two-element list of the form (*phrase translation*), where *phrase* is the value obtained from interaction with you and *translation* is the value used by the parser as the translation for the expert. This allows transformations of your inputs (for example, you may want the system to convert your input into a string). For instance, the supplier-city-expert lexical item might return the list `(New York "New York")`. The NLMenu system relies on experts returning two-element lists with the semantics just described.

Typically, experts have a standard structure consisting of two parts, a target-language independent *interaction* specification and an outer shell specific to a target language. The interaction component pops up the interaction window or menu, returning **nil** or a value of the form just described. Outer shells are of the form:

```
(defun expertname-expert (parameters)
  (setq value (expertname-expert-interaction parameters))
  (if (abortedp value)
      nil
    (list (expertname-make-lexical-item value)
          (expertname-make-target-language-specific-translation
                                        value))))
```

The outer shell handles aborted interactions and also translates values returned by the interaction function into lexical items for display in the input window and for addition to the parse. This structure for experts allows separation of the interaction from the value returned, as suggested above, so that different outer shells corresponding to different target languages can use the same interaction. For instance, a multi-item interaction expert might pop up a menu of items and return a list as its *phrase*. An SQL translation of the list might be (*v, v, ...*); an RTMS translation might be (*v v ...*); an English translation might be *v, v, ... or v*; a French phrasing might be *v, v, ... ou v*.

Several generic experts are included in the core NLMenu system, and some additional such experts ideal for interfacing NLMenu to databases are included in the NLMenu-RTMS-Interface portion of the system. The former are described in paragraph 6.3 and also appear in the file SYS:NLMENU;EXPERTS-BASIC-ONES.LISP, while the latter appear in the file SYS:NLMENU-RTMS-INTERFACE;GENERIC-EXPERTS.LISP and are referenced in the lexicons that the database interface can automatically generate. To include additional special-purpose experts in a particular NLMenu interface, develop the experts in accordance with the description above and include them in an experts file. You then specify the experts file to the interface builder when you add a new interface to the system as described in paragraph 2.2.1.2.

NATURAL LANGUAGE MENU GRAMMER-TESTING TOOLS 5

Highlights of This Section

- Grammar-tool interface
- Format test
- Static well-formedness test
- Conflict checker
- Semantic consistency test
- Cycle checker
- Sentence generator
- Changing test grammars

Grammar-Tool Interface

5.1 The Natural Language Menu (NLMenu) grammar-testing tools provide you with an automated means of debugging and testing your grammar. These tools can be useful during grammar development. The grammar-tool interface provides access to six grammar tests:

- Format test
- Static well-formedness test
- Conflict checker
- Semantic consistency test
- Cycle checker
- Sentence generator

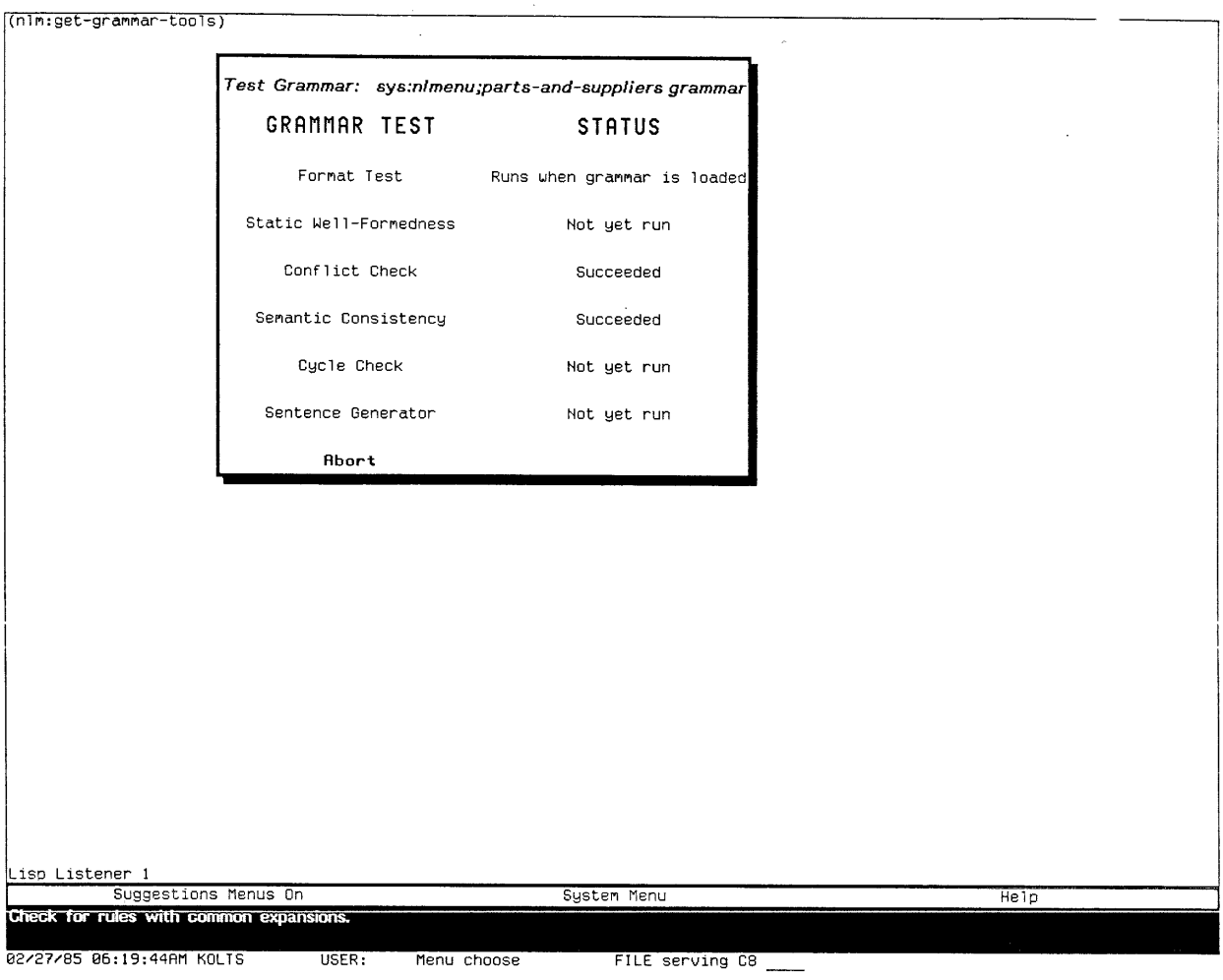
You can invoke the interface in two different ways:

1. During NLMenu interface construction (see paragraph 2.2.1.2), select `yes` for the value of `Test the grammar?`.
2. Invoke the function **nlm:get-grammar-tools**.

In either instance, the current source grammar is read in from disk, and a *structured grammar* is constructed. This structured grammar is an aggregate consisting of various representations of each source grammar rule. These representations can be used by one or more of the grammar tests. The representations are also accessed by the grammar compiler if the structure is created prior to grammar compilation. Once the structure is created, the individual grammar tests proceed quite rapidly.

After grammar loading and the creation of the structured grammar (appropriate messages appear on the video display to indicate the progression of these events), a multicolumn menu appears on the screen. Figure 5-1 shows this menu, the grammar-tool interface screen.

Figure 5-1 Grammar Test Menu



Column 1 of the menu contains a list of the various tests that are available. Each item in this column is mouse-sensitive, and clicking left on an item runs the test, processes the results, and redisplays the menu. To exit the grammar-tool interface, select the Abort option or move the mouse cursor out of the menu. Column 2 lists a brief indication of the status of each test for the grammar under consideration. The status entry has one of the following values:

- Not yet run — The grammar test in question has not yet been applied.
- Succeeded — The grammar test in question has successfully completed.
- Failed — The grammar test in question is complete, and at least one error was found.
- N sentences generated — This applies only to the sentence generator and indicates how many sample sentences were generated during the most recent application of this test.
- Runs when grammar is loaded — This applies only to the format test, which is automatically applied each time a source grammar is compiled.

The status entries are dynamically updated as various tests are performed.

The various tests are independent of each other but, in some instances, require that a lexicon be present. When a particular test is run, you can choose (by means of a choose-variable-values menu) where to route the results. By default, all results go to the video display, but if a pathname is selected, the output is appended to the current file contents, if any. Tests that require other input parameters (for example, the sentence generator) also get their input from a choose-variable-values menu.

The following paragraphs provide detailed descriptions of each test.

Format Test

5.2 The format test checks the grammar for unbalanced delimiters and is automatically run when a grammar is compiled. Recognized delimiters include braces, square brackets, and parentheses. The error handler traps unbalanced parentheses if you attempt to load a source grammar in such a state. However, other unbalanced delimiters are trapped at grammar-compile time. The format test is applied at this time because it requires a modified form of the rule generated by the grammar compiler. All other grammar tests can proceed on a grammar that would fail the format test. If an error is detected, an explanatory message containing the failed rule is printed on the video display.

A trap is then generated to the error handler. This allows the grammar writer to return to the editor, make the necessary correction, and continue. For example, if the bad grammar rule is $S \longrightarrow A \{B\ C\ D$, the error message generated includes `(({B C D ))` in its *failed rule* slot (that is, a list form of the portion of the offending rule beginning with the unmatched delimiter).

Static Well-Formedness Test

5.3 The static well-formedness test checks:

- Whether all left-hand sides of the grammar rules are reachable (other than the root category)
- Whether all terminals in the grammar have corresponding lexical entries
- Whether the lexicon has extraneous entries (that is, entries that do not appear as terminals in the grammar)
- Whether the lexicon has redundant entries (that is, entries that appear as nonterminals in the grammar)

In general, errors of one type do not affect other parts of the well-formedness test. The last three of these conditions involve the lexicon, which is loaded if necessary when the test is run. The static well-formedness test reports the following types of results:

- Unreachable left-hand sides of grammar rules (no right-hand side refers to them, and they are not the root)
- Undefined lexical items (items appearing as terminals in the grammar without corresponding entries in the lexicon)
- Unused lexical items (items appearing as entries in the lexicon that are not terminals in the grammar)
- Redundant lexical items (items appearing in the lexicon that appear as nonterminals in the grammar)

NOTE: Because the Explorer system is not case-sensitive, the standard practice of designating nonterminal elements with uppercase letters and terminal elements with lowercase letters must be amended. In the following example, uppercase letters from the beginning of the alphabet designate nonterminal, while uppercase letters from the end of the alphabet represent terminal elements.

For a simple grammar containing only the following syntactic components

```
S --> A B
S --> D Y
B --> C (A C)
A --> U
C --> W
D --> X
G --> A C
```

and a corresponding lexicon of

```
(U      "string 1"      MENU1      translation1)
(W      "string 2"      MENU2      translation2)
(X      "string 3"      MENU1      translation3)
(Z      "string 4"      MENU1      translation4)
(B      "string 5"      MENU2      translation5)
```

the static well-formedness test would generate the following messages:

```
Unreachable left-hand sides of grammar rules (no right-hand
side refers to them, and they are not the root):
(G)

Undefined lexical items (items appearing as terminals in the
grammar without corresponding entries in the lexicon):
(Y)

Unused lexical items (items appearing as entries in the
lexicon that are not terminals in the grammar):
(Z B)

Redundant lexical items (items appearing in the lexicon that
appear as non-terminals in the grammar):
(B)
```

Conflict Checker

5.4 The conflict checker checks for rules with common expansions, that is, rules that, when all abbreviatory symbols are simplified, generate exactly the same path. Redundant rules generate such paths and are undesirable because they unnecessarily increase the size of the grammar. For instance, the following rules, when expanded, have an unabbreviated rule in common:

Rule A: S → A {B C} {D E}

which expands to:

1. S → A B D
2. S → A B E
3. S → A C D
4. S → A C E

Rule B: S → A {Q C} {D F}

which expands to:

1. S → A Q D
2. S → A Q F
3. S → A C D Same as Rule A, Expansion 3
4. S → A C F

The results reported by the conflict checker indicate the rules that have expansion conflicts.

For example, for a grammar containing rules with syntactic components

```
S-->A B (C) D E
```

and

```
S-->A B C D E
```

the following message would be generated by the conflict checker:

```
The following rules have expansion conflicts:
S --> A B (C) D E and S --> A B C D E conflict at:
A B C D E
```

Semantic Consistency Test

5.5 In NLMenu grammars, a rule is considered to be semantically consistent if the number of its expansions and the number of its semantic components are equivalent. That is, each rule expansion (path through the grammar rule) must have a matching semantic component (translation list). If there are too many or too few translation lists, the grammar is in error. To illustrate:

Element numbers

```
      1     2 3       4 5
S ⟶ A    {B C}     {D E}     ((1 2 4)  (1 2 4))  ((1 2 5)  (1 2 5))  )
```

The expansions $S \longrightarrow A\ B\ D$ and $S \longrightarrow A\ B\ E$ have translation lists; the expansions $S \longrightarrow A\ C\ D$ and $S \longrightarrow A\ C\ E$ do not. You must correct missing or extra translation list errors before you can build an NLMenu interface.

The semantic consistency test reports the following messages:

If successful:

```
Semantic consistency test succeeded.
All expansions of rules have matching semantic parts.
All semantic parts of rules have matching expansions.
```

If unsuccessful, the problem is due to one of the following possible causes:

- The expansions of a rule have no matching semantic parts.
- The expansions of a rule have more than one matching semantic part.
- The semantic parts of a rule have no matching expansions.

For example, for a grammar containing the following rules

```
("A --> B (C A)"
((1)))

("A --> D (C A)"
((1))((2 1 3))((1 2 3)))

("A --> E (C A)"
((1))((2 1 3))((2 1 4)))
```

the semantic consistency test would generate the following messages:

```
These expansions of rule A --> B (C A) have no matching
semantic parts:
B C A

These expansions of rule A --> D (C A) have more than one
matching semantic part:
D C A

These semantic parts of rule A --> E (C A) have no matching
expansions:
2 1 4
```

The problem with the second rule in the example could be interpreted in two ways:

- There is an extra semantic part.
- There are two semantic parts that match the expansion $A \longrightarrow D\,C\,A$.

As the example illustrates, the semantic consistency test uses the second interpretation since the *extra* semantic part is a potentially valid one.

Cycle Checker

5.6 The cycle checker reports an error if any set of rules in the grammar produces an infinitely looping path. Consider the following example:

```
S --> A B C
A --> B
B --> C
B --> W
C --> W
C --> A
```

The path from *A* to *B* to *C* and back to *A* is an infinite loop. The test also detects problems of the following sort:

```
S --> A B C
A --> B U
B --> V A
C --> W
```

This grammar cannot generate sentences because of the rules that have *A* and *B* as mothers. There is no termination of that path (from *A* to *B* back to *A*, and so forth). If the test completes with a successful result, the cycle checker reports the following message:

```
There are no cycles in the grammar.
```

If the test completes without a successful result, a message containing the cycle is displayed.

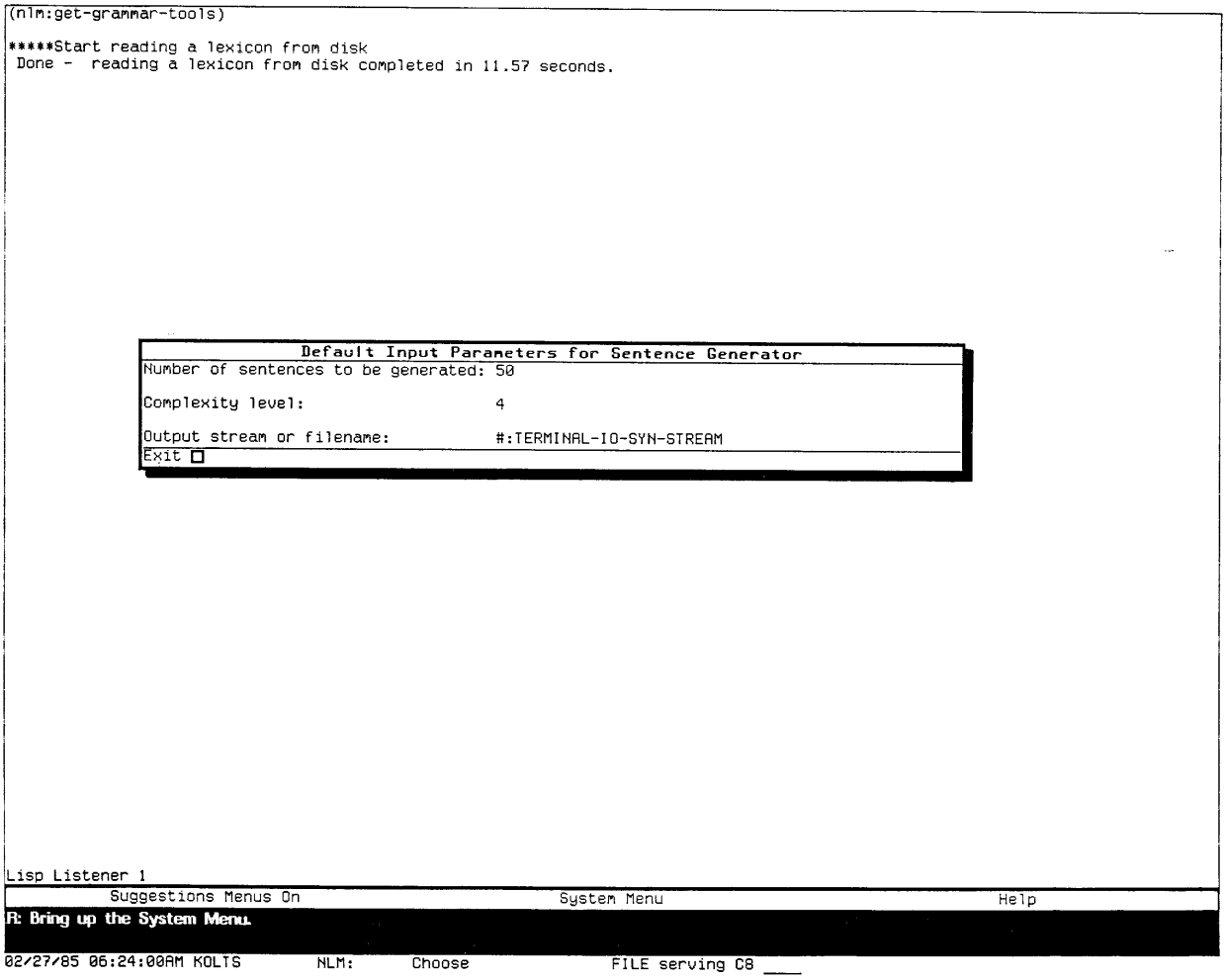
NOTE: NLMenu permits *left-recursive rules* (rules where the recursive, or expanding, element appears on the left-hand side of the grammar rule) and does not consider them to be cyclic.

Because this test does not require a lexicon file, you should run your grammar through the cycle checker before building the lexicon.

Sentence Generator

5.7 The sentence generator provides you with a means of generating a specified number of representative sentences from the grammar. Examination of the test sentences often reveals whether the grammar is too weak (certain types of desired sentences are not covered) or too strong (certain types of sentences not intended to be covered can be generated). You can also specify the *complexity level* of the sentences generated. The complexity level is a parameter representing the number of grammar rules that are applied before convergence begins (that is, before no more recursive rules are selected). A level in the range of 3 to 6 gives a set of sentences of sufficient complexity. You select values for the number of sentences to be generated, the complexity level, and the destination of the output from a choose-variable-values menu (see Figure 5-2) that pops up when you select the sentence generator test from the Grammar-Tool menu. Default values are 50, 4, and the video display.

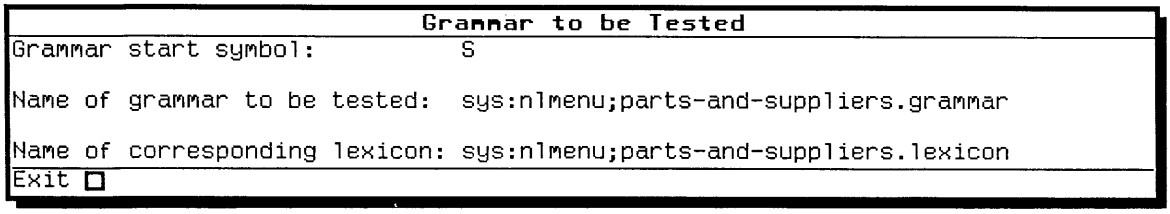
Figure 5-2 Sentence Generator Choose-Variable-Values Menu



Changing Test Grammars

5.8 You can test more than one grammar during a machine session and switch from one grammar to another at any time since a list of grammar-tool interfaces is maintained. This list keeps track of the structured grammar for each test grammar as well as the current state of each grammar test for that test grammar. To change grammars, exit from the current Grammar-Tool menu and call the function **nlm:reset-grammar-tool-interface**. A choose-variable-values menu (see Figure 5-3) pops up, allowing you to specify a new test grammar and corresponding lexicon. You can then invoke the new grammar-tool interface with a call to the function **nlm:get-grammar-tools**.

Figure 5-3 Grammar Tool Interface Reset Menu



NATURAL LANGUAGE MENU LEXICOGRAPHER 6

Highlights of This Section

- Lexicographer functionality
- Lexicographer interface
- Lexicographer options
  - Specify new lexical items
  - Modify existing lexical items
  - View selection windows
  - Build screen
  - Abort

Introduction

6.1 The Natural Language Menu (NLMenu) lexicographer assists you in the completion or modification of your lexicon file, in laying out the individual windows that contain the selection items, and in building the selection screens that are composed of the various windows.

To use the NLMenu lexicographer, you must first have a grammar and a lexicon file (the lexicon file can be empty). The NLMenu lexicographer first reads the lexicon file and then presents a menu listing the various lexicographer options. For most options, an uncompiled grammar is loaded upon selection. The grammar is then saved and reused if other options are later selected so that it is never loaded more than once during a session. Options that change the lexicon (Specify New Lexical Items or Modify Existing Lexical Items) first find all terminal categories in the grammar and then construct templates of the following form for each terminal category:

(*category word-or-phrase menu translation*)

The NLMenu system uses information already present in the lexicon for the remaining three slots in each template, if possible, or else leaves the slots empty (that is, setting them to **nil**). Templates of the form

(*category nil1 nil2 nil3*)

represent items that you can insert into the lexicon, while all other templates represent items that you can modify. Template building can occur only once per session and provides a specification of what items are required in the lexicon based on the current state of the grammar.

The first two options of the NLMenu lexicographer (Specify New Lexical Items and Modify Existing Lexical Items) form an automated tool to complete the lexicon file. You can, however, create your own lexicon file using the Zmacs editor. For specifics on lexicon file format, refer to Section 4. The remaining options are useful in laying out the final NLMenu screen.

Lexicographer Interface

6.2 There are two ways to invoke the lexicographer. First, when the NLMenu system is initially loaded without the starter kit values, the NLMenu system asks if you would like to use the lexicographer. If the NLMenu system is already loaded, you can access the lexicographer from a Lisp Listener by entering:

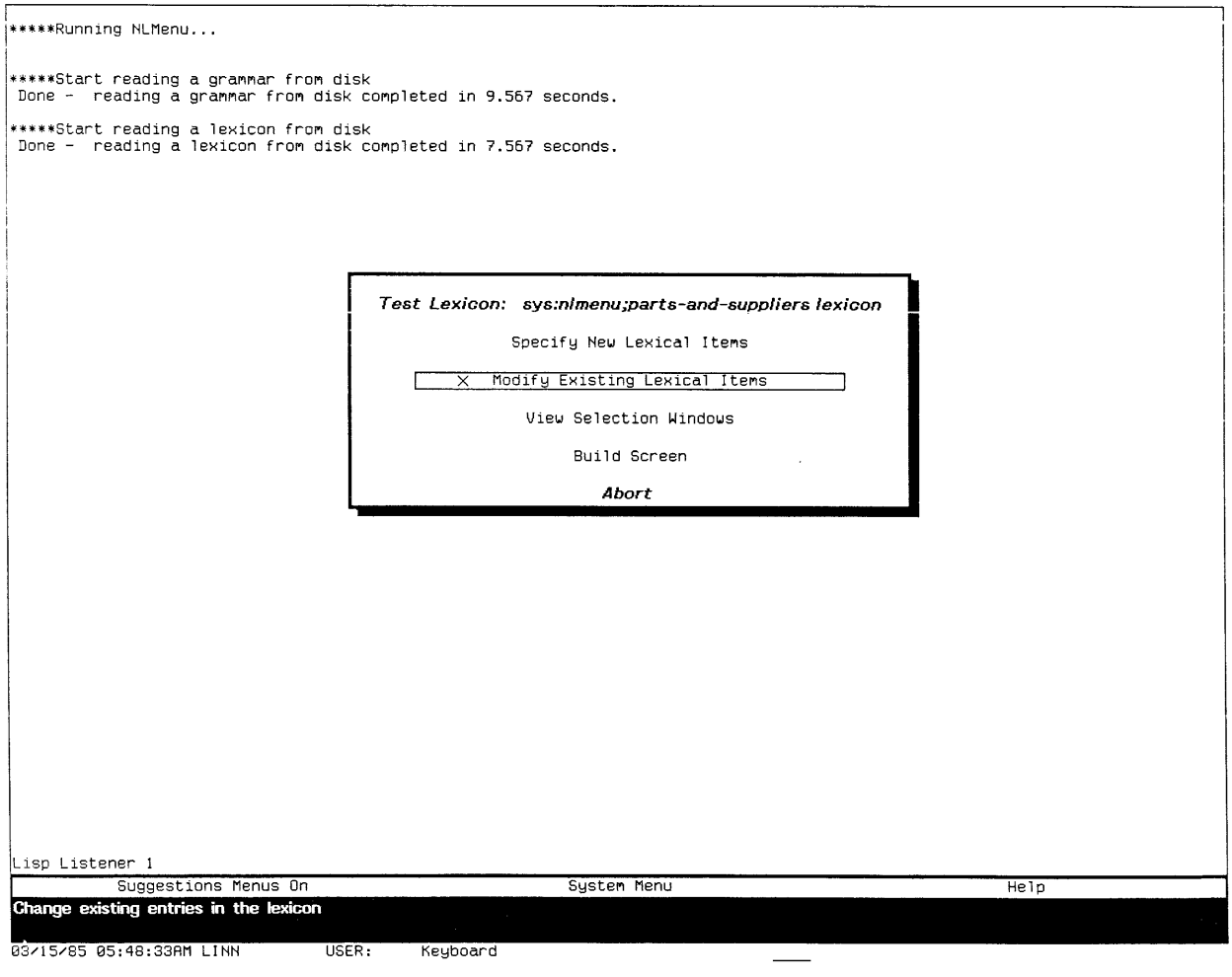
```
(nlm:get-lexicographer)
```

The following prompt appears on the video display:

```
Reset the lexicon (y/n)?
```

If you respond *n*, the system uses the latest version of the lexicon file in memory. If you respond *y*, the system loads a new source lexicon. Depending on the state of the lexicon file, a menu appears containing either one choice (`Specify New Lexical Items`) or a list of four choices (see Figure 6-1).

Figure 6-1 Lexicographer/Screen-Builder Menu



Lexicographer Options

6.3 A detailed discussion of each Lexicographer/Screen-Builder menu option follows.

Specify New Lexical Items

6.3.1 To add new entries for lexical items to the lexicon, select the Specify New Lexical Items option. The lexicographer loops through the list of lexical templates. If it does not encounter a template with three **nil** values, it prints a message that the lexicon is complete. However, if the lexicographer encounters such a template, it displays the template in the following way:

```
(a nil1 nil2 nil3)
```

where:

`a` is the lexical category.

`nil1` is a slot for the word or phrase to appear in the selection menus.

`nil2` is a slot for the menu in which the word or phrase is to occur.

`nil3` is a slot for the translation for the word or phrase.

NOTE: The second item (word or phrase) must be a string. You must enclose string values in double quotes.

The lexicographer then prompts you for the pieces of information it needs to complete the lexical entry:

```
Enter word or phrase to appear in selection menus:
```

Your response must be a string. After your response is entered, another prompt appears:

```
Enter selection menu in which this phrase should appear:
```

Respond with the name of the menu in the selection screen to which the lexical item maps.

The next prompt is:

```
Will translation be an Expert (yes/no)?:
```

You can enter a single letter or the entire word in either uppercase or lowercase. If you indicate the item is not an expert, the system prompts you for the desired translation:

```
Enter translation for this phrase:
```

In response, enter either a lambda expression or an atomic expression. The system then displays the added item.

If you move the cursor out of this menu without making a selection, the system sets the translation to **nil**.

On the other hand, if you indicate the translation is an expert item, another menu pops up. Appearing at the top is the prompt:

```
Select an Expert type:
```

In the menu beneath the prompt is a list of all current generic expert types. These are:

- Type-in — This is the expert for entering any character string without restrictions.
- Calculator — This is a pop-up calculator for entering integer or real numbers.
- Range — This expert requires numeric input within a specified range (for example, 0 through 100).
- Default — This expert pops up a menu of nondefault options so that you can select one of them or select nothing (by moving the mouse out of the menu). If you select nothing, a prespecified default value is selected. The default value is displayed at the top of the menu.

If you designate the lexical item status as a calculator expert, the lexicographer automatically constructs a suitable expert translation. If, instead, you designate the lexical item as a type-in expert, the lexicographer requires a prompt before it can process the translation of the lexical item. The NLMenu lexicographer queries you as follows:

```
Indicate prompt for type-in window
(prompt will be converted into a string):
```

The NLMenu lexicographer converts your response into a string, so it is not necessary to enter it in double quotes. You must enter a response to this prompt. You are not able to leave the window until you do so, although you can type `ABORT`, in which case **nil** is returned as the prompt.

As with the type-in expert, designation of the lexical item as a range or default expert requires you to provide additional information before the expert translation can be automatically generated. For the range expert, you are queried as follows:

```
Indicate lower bound of range (e.g., 0):

Indicate upper bound of range (e.g., 100):
```

For the default expert, the query is:

```
Input list of values with default value first
(e.g., (def-val val2 val3 ....)):
```

After you provide the translation for a lexical item, the lexicographer issues the following prompt:

```
Do you wish to add more items (yes/no)?:
```

If you do add more items, the process described above is repeated. Eventually, you either respond that you no longer wish to continue adding items, or the lexicographer finds no more templates that require completion.

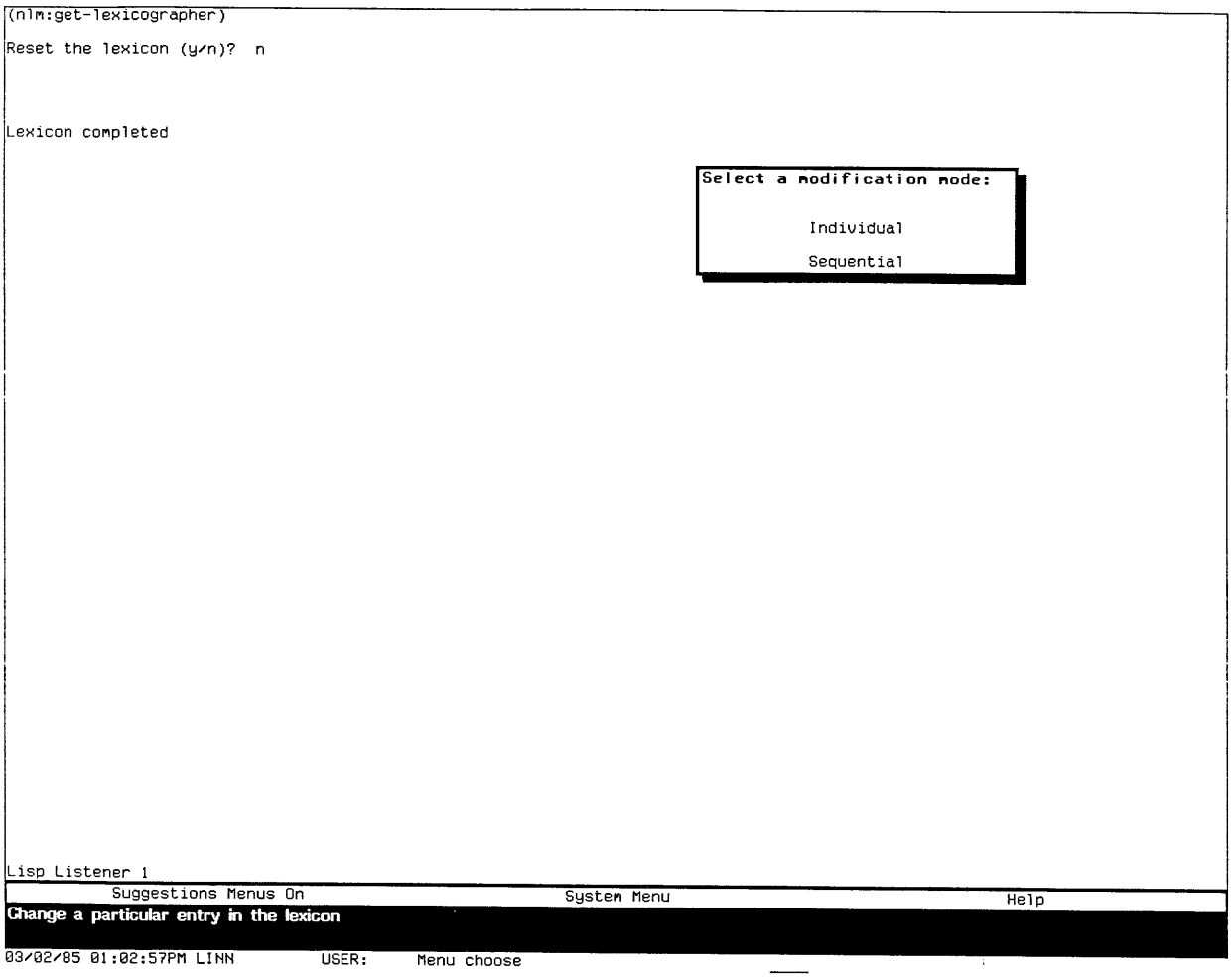
While you are using the lexicographer to insert items into the lexicon, the system maintains and updates a copy of the lexicon in the Lisp environment. When you leave the lexicographer, whether by choice or when all templates are completed, the system writes the latest version of the lexicon to disk, issuing a message to that effect.

Modify Existing Lexical Items

6.3.2 To change the content of an existing lexical entry, use the Modify Existing Lexical Items option of the lexicographer interface. The lexicographer skips over entries of the form `(a nil1 nil2 nil3)`. You cannot modify these entries because they contain no previous information.

When you select this option, a menu (see Figure 6-2) appears.

Figure 6-2 Modify Existing Lexical Items Menu



You are prompted to indicate a modification mode. Your response options are two: sequential or individual.

If you move the cursor out of this window rather than clicking on a particular mode, the system defaults to individual mode (see details below).

Sequential Modification Mode

6.3.2.1 If you choose sequential mode, the lexicographer loops through the lexicon and sequentially displays entries eligible for modification. The NLMenu system prints out the prompt:

```
Item to be modified is:  lexical entry
```

The existing template then appears with the following prompts in succession:

```
Enter word or phrase to appear in selection menus:
```

Your response must be a string. After your response is entered, another prompt appears:

```
Enter selection menu in which this phrase should appear:
```

Respond with the name of the menu in the selection screen to which the lexical item maps.

The next prompt is:

```
Will translation be an Expert (yes/no)?:
```

The response to this prompt and the system reaction are described for the corresponding prompt in the preceding paragraph on the addition of lexical items. Note that if you change only one component of the lexical entry, you still need to respond to all other prompts by reentering the values that you want to retain.

When you have responded to all the prompts, the system displays the template of the changed item and then inquires whether you want to see more entries for modification or terminate the looping process:

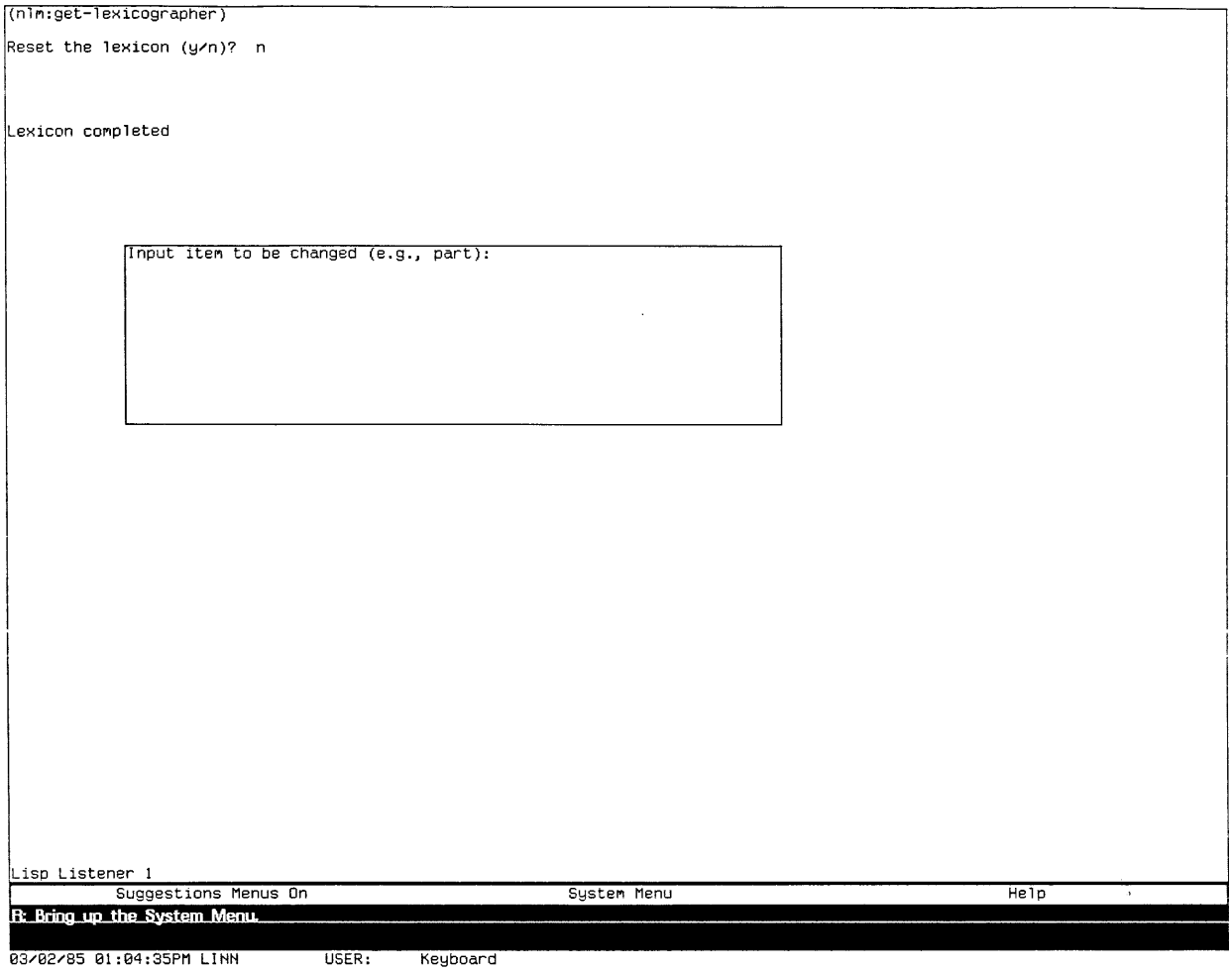
```
Do you wish to change more entries (yes/no)?:
```

If you decline to see more, the system writes the new lexicon to disk and issues a message that the process is in progress.

Individual Modification Mode

6.3.2.2 To change a single lexical item, select individual mode, and a menu (see Figure 6-3) appears:

Figure 6-3 Individual Mode Menu



In response, enter the lexical category of the entry you want to change. You need not enter this value as a string. The same four prompts

```
Item to be modified is:  lexical entry

Enter word or phrase to appear in selection menus:

Enter selection menu in which this phrase should appear:

Will translation be an Expert (yes/no)?:
```

are displayed, and you respond to them in turn. Once again, you must enter either a new value or reenter an existing value that you want to retain. To confirm the change(s), the NLMenu system displays the modified template. At this point you are asked whether you want to modify another entry in individual mode. When you ultimately decide not to exercise this option, the lexicon is written to disk. The difference between individual mode and sequential mode is that you control the items called up for modification.

View Selection Windows

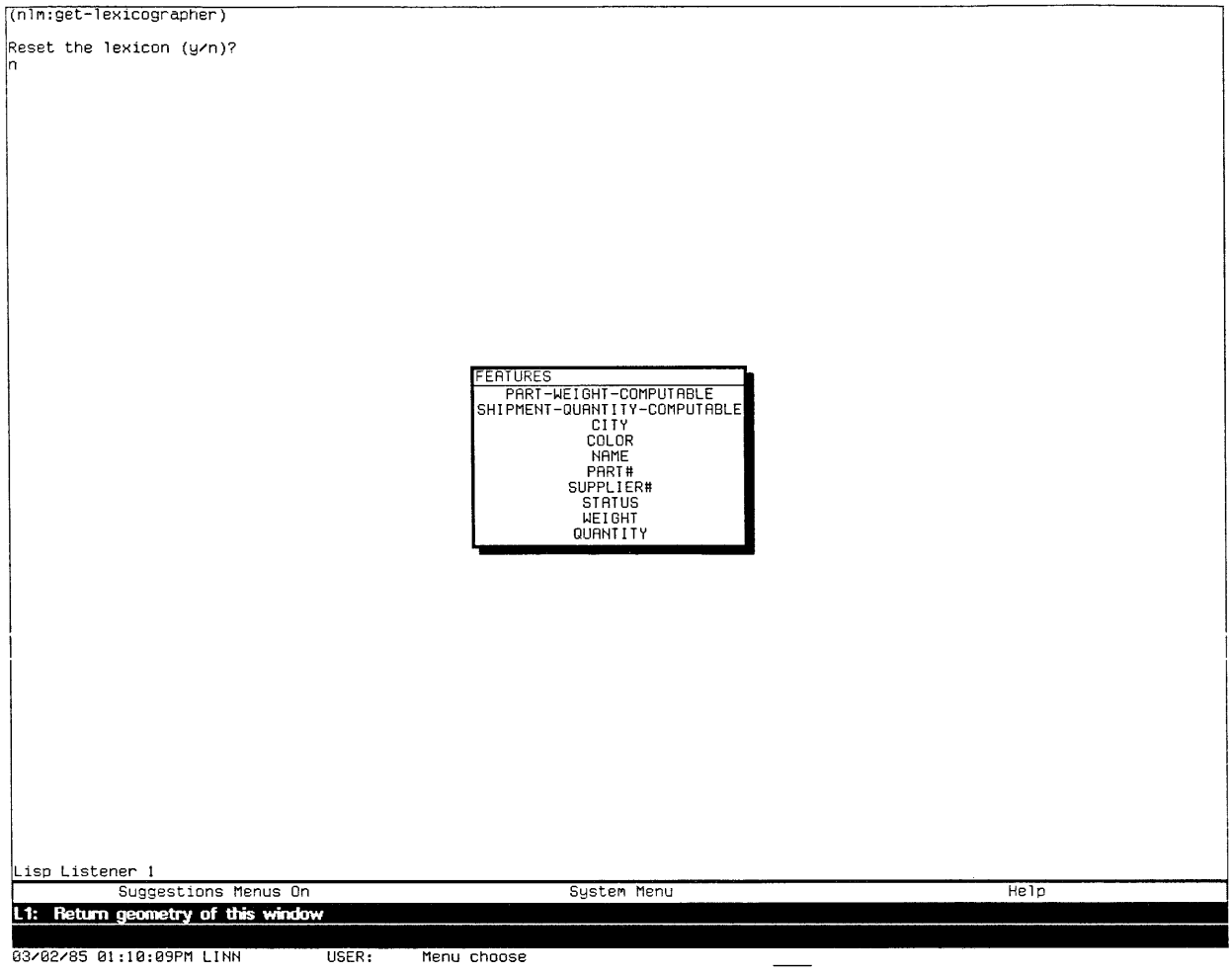
6.3.3 To see all the items that appear in a particular selection window, select the View Selection Windows option. The resulting menu has all the items in the lexicon that currently map to that menu. Consequently, if the lexicon changes, the content of the window also changes.

To begin this process, click once on the View Selection Windows option of the lexicographer interface, and a menu (see Figure 6-4) appears.

Figure 6-4 View Selection Windows Menu

In it are the names of all menus (without repetition) specified in the lexicon. The currently loaded lexicon in this instance is the Parts-and-Suppliers lexicon. You can view any one of these menus. To do so, position the mouse cursor over a menu name and click left once. For instance, if you click left once on the item FEATURES, a menu (see Figure 6-5) appears.

Figure 6-5 Menu Items Display



All lexical items that map to the menu appear in the window when you select the View Selection Windows option of the lexicographer interface. The information supplied by this option is useful for screen layout. The data items assist you by indicating the size of the window and by providing a quick way to determine all the items that map to a particular window.

In addition, all the items that the NLMenu system displays in a selected window, (for example, FEATURES), are mouse-sensitive. If you click on any one of them, the NLMenu system provides the current geometry, that is, the precise dimensions, of the window. The geometry displayed for the FEATURES window is `(1 10 224 130 nil nil)`. These dimensions are displayed to the screen as a list of six features:

- Number of columns in the window
- Number of rows in the window
- Inside width of the window in pixels or **nil**
- Inside height of the window in pixels or **nil**
- Maximum width in pixels
- Maximum height in pixels

The first four items determine how the window is constructed and are most relevant as you lay out your own windows. The last two items are constraints whose likely values are **nil**. They are essentially meaningless to you unless you want to specify the geometry of the window you are constructing. For more information on the construction of windows, refer to the operations **:geometry** and **:current-geometry** in the *Explorer Window System Reference.*

After presenting this list of window dimensions, the NLMenu system redisplays the Lexicographer Interface menu. If you do not want to examine the window geometry, move the mouse cursor out of the menu items display. The display disappears, and the Lexicographer Interface menu reappears.

Build Screen

6.3.4 To view the layout of your interface screen, select the Build Screen option of the lexicographer. The NLMenu system can construct reasonably optimal screen descriptions. This automates the task of providing descriptions of the panes and constraints (see paragraph 4.8) or, at least, suggests possible layouts if you want to specify your own description. Clicking left on the Build Screen option of the lexicographer interface thus constructs descriptions of the panes and constraints in the constraint-frame language as defined in the *Explorer Window System Reference*. The panes description is bound to the global variable **nlm:*screen-builder-panes***, and the constraints description is bound to **nlm:*screen-builder-constraints***. This permits easy viewing of the descriptions produced (for example, try: `(grind-top-level nlm:*screen-builder-constraints*)`). To view the screen specified by the description produced by the screen-builder, select the Build New Interface command (see paragraph 2.2.1.2) from the NLMenu System menu.

The screen-builder uses various heuristics to determine screen layout. The first window, placed in the upper left-hand corner of the screen, is generally active initially, and contains items from the grammatical category from which interface users are most likely to make their first selections. From this configuration, an optimal number of rows and columns is determined, and additional menus are positioned by various techniques, such as matching their sizes with the amount of space remaining. Backtracking occurs as needed until a valid description is produced.

A quick, nonoptimal screen description is also available. This screen description simply produces one row of menus with each menu getting an equal amount of space. Overly long items are truncated. However, this description is still useful during grammar development as a quick way to see which lexical items map to which menus. Although it is not a separate lexicographer option, you can invoke it by responding with **nil** to both the panes and constraints description options when you select the Build New Interface command (see paragraph 2.2.1.2) from the NLMenu System menu.

Abort

6.3.5 To return to the Lisp Listener, select the Abort option from the Lexicographer Interface menu.

DATABASE INTERFACE 7

Highlights of This Section

- Comparing interface generators
- NLMenu interface generation
  - Portable spec
  - Managing NLMenu interfaces
    - NLMenu System menu
    - NLMenu-Interfaces relation
    - Interface-Grants relation
    - Creating, modifying, and dropping an interface
    - Granting and revoking interfaces
  - Database creation
  - Default constraint window
  - Formal ideas underlying database interface generation
    - Semantic grammar generation
    - Semantic lexicon generation

Introduction

7.1 The database interface allows you or the user of your application to generate Natural Language Menu (NLMenu) interfaces to database applications automatically. In contrast to more conventional natural language interface generators, automatically-generated NLMenu interfaces are immediately usable without a long, empirical lexicon acquisition phase. In particular, while certain other systems support automatic interface generation to new applications, such interfaces require a relatively lengthy, empirical tuning phase. During this tuning phase, you (or an experienced user of your interface) must develop a large lexicon that is application-specific and not portable to new domains. However, when you (or your interface user) generate an NLMenu interface, the lengthy tuning phase is unnecessary because the selection menus provide direct guidance to use of the language of the interface. Thus, the NLMenu database interface supports a greater degree of portability across database applications than typical interface generators.

NLMenu Interface Generation

7.2 Automatic NLMenu interface generation requires some means of identifying appropriate application and domain-specific information. The interface generator combines such information with a core grammar and lexicon in such a way as to allow the usual NLMenu look-ahead strategy. With this strategy, individual words or phrases are parsed as they are input. The required information is elicited in the form of a *portable spec*. Portable-spec creation requires no knowledge of grammars, lexicons or the target query language—only an elementary knowledge of tables, keys, and joins. Thus, a large class of users can build their own interfaces. Portable-spec creation is described in paragraph 7.2.1. Additionally, the ability to build new natural language interfaces creates a need to manage those interfaces. NLMenu interfaces are treated as a new type of database object, a sort of composite application *view* of a user's data. In particular, two dedicated relations in a Relational Table Management System (RTMS) database contain information required to manage NLMenu interfaces. Appropriate updating of these relations is described in paragraph 7.2.2. Database creation is described in paragraph 7.2.3.

Portable Spec

7.2.1 A *portable spec* is a specification of a set of categories containing all the domain-dependent information necessary to specify a complete user-interface grammar and lexicon. A representative portable spec, illustrating the sort of information required by the interface generator, is shown in Figure 7-1.

Figure 7-1 Portable Spec Used to Generate Parts-and-Suppliers Interface

```
((SUPPLIER PART SHIPMENT)                          ; covered tables
 (SUPPLIER PART SHIPMENT)          ; access rights-retrieval relations
 (SUPPLIER PART SHIPMENT)          ; access rights-insertion relations
 (SUPPLIER PART SHIPMENT)          ; access rights-deletion relations
 (SUPPLIER PART SHIPMENT)          ; access rights-modification relations
 ((PART CITY COLOR NAME PART#)                     ; non-numeric attributes
  (SUPPLIER CITY NAME SUPPLIER#)
  (SHIPMENT PART# SUPPLIER#))
 ((SUPPLIER STATUS))                               ; numeric attributes
 ((PART WEIGHT)                                    ; computable attributes
  (SHIPMENT QUANTITY))
 ((SUPPLIER SUPPLIER# NAME)                        ; identifying attributes
  (PART PART# NAME)
  (SHIPMENT SUPPLIER# PART#))
 ((USE "which are shipments of" AND                ; two-table joins
       NIL
       TO JOIN (SHIPMENT PART#) AND (PART PART#))
  (USE "which were shipped by" AND
       "who ship"
       TO JOIN (SHIPMENT SUPPLIER#) AND (SUPPLIER SUPPLIER#)))
 ((USE "who supply" AND                            ; three-table joins
       "which are supplied by"
       TO JOIN (SUPPLIER SUPPLIER#) AND (SHIPMENT SUPPLIER#)
       AND TO JOIN (SHIPMENT PART#) AND (PART PART#)))
 NIL                                               ; user-supplied relation experts
 NIL                                               ; user-supplied relation-attribute
                                                   ; experts
 ((ATTRIBUTES "<specific part colors>"             ; edited items
        "<specific colors>"
  (MODIFIERS "whose part color is"
       "whose color is")))
```

The example corresponds to the sample Parts-and-Suppliers interface, as discussed in paragraph 2.2. The spec describes a database containing the following three relations:

SUPPLIER (supplier# name city status)
PART (part# name city color weight)
SHIPMENT (supplier# part# quantity)

Portable Spec Categories

7.2.1.1 The following discussion explains portable spec categories, such as those shown in Figure 7-1. Note that each category is written as a list.

Covered Tables All the relations that the interface covers are specified in this slot of the spec.

Access Rights The retrieval, insertion, deletion, and modification relations all specify access rights on selected covered tables. For instance, listing the supplier relation in the insertion relation slot indicates to the interface generator that it should create appropriate grammar rules and lexicon entries to allow the addition of new suppliers to the database.

Attribute Classifications Nonnumeric attributes, numeric attributes, and computable attributes classify database attributes according to type. These categories are disjoint, and you must specify each database attribute in at most one category. Thus, you need not specify all attributes in this part of the spec. This permits hiding surrogate attributes that can result from normalizing relations and whose main purpose is to serve as internal tuple (row) identifiers. Computable attributes are numeric attributes upon which various mathematical operations can be performed. For instance, in the spec shown in Figure 7-1, the status attribute of the supplier relation is specified as a numeric attribute because performing algebraic operations on status numbers is not meaningful. However, such operations are meaningful when applied to the weight attribute of the part relation and the quantity attribute of the shipment relation. These attributes are specified as computable. Each attribute classification is written as a list of lists, with sublists that contain the relation name followed by the appropriate attributes of that relation. This allows attributes of the same name to appear in various relations.

Identifying Attributes These are typically relation keys—sets of attributes within a relation that the system uses to identify tuples uniquely. The identifying attributes slot can include nonkey attributes if they better identify tuples. These can even include less than a full key if you seek to identify sets of rows together.

Two-Table Joins This category specifies supported join paths between relations. A bridging relative clause that appears in the selection menus is included. For instance, the first two-table join specification shown in Figure 7-1 permits construction of the following sort of sentence:

Find shipments WHICH ARE SHIPMENTS OF parts ...

but blocks construction of the sort:

Find parts *relative clause* shipments ...

In other words, the first bridging relative clause connects shipments and parts, while the second such phrase, which is **nil** in this example, connects parts and shipments. The connections are not, in general, symmetric but are tied to word order in the sentences that you construct.

The second two-table join specification in Figure 7-1 permits construction of sentences of the following sorts:

Find shipments WHICH WERE SHIPPED BY suppliers ...

Find suppliers WHO SHIP shipments ...

You also specify which two relations to join by providing a list that includes the relation name followed by one or more attributes common to each relation being joined. Thus, returning to the first entry in the two-table join slot, the entries `(SHIPMENT PART#)` and `(PART PART#)` indicate

that you can join the shipment and part relations over the common attribute part number (part#).

Three-Table Joins This category specifies supported binary relationships (in the entity-relationship data model sense) where one relation relates to two others. As with two-table joins, you specify which bridging relative clauses appear in the selection menus. For instance, the shipment relation relates to both the supplier relation through the common attribute supplier number (supplier#) and the part relation through the common attribute part number (part#). This provides a path from a supplier entity to a part entity to another (perhaps different) supplier entity (there is also a part-supplier-part route). In other words, a supplier can supply many parts, each of which can have many suppliers. The three-table join specification shown in Figure 7-1 permits the type of expression and enables you to construct the following types of sentences:

Find suppliers WHO SUPPLY parts WHICH ARE SUPPLIED BY suppliers . . .

Find parts WHICH ARE SUPPLIED BY suppliers WHO SUPPLY parts . . .

User-Supplied Experts The next two slots define special-purpose, user-supplied experts to replace system-generated defaults:

- *New-rel* (new-relation) experts are referenced in the translation slot of lexicon entries. These entries are reached by means of certain rules that allow construction of queries that refer to specific tuples. For instance, the following contain new-relation experts:

  Find *specific tuples in specific relation* . . .

  Delete *specific tuples in specific relation* . . .

  Find average weight of *specific tuples in PART relation* . . .

- *New-rel-attr* (new-relation-attribute) experts support references to specific attribute qualifications. For instance, the following contain new-relation-attribute experts:

  Find *tuples in specific relation* with *attribute qualification* . . .

  Find parts whose color is red.

  Find suppliers whose city is Paris.

The system-generated default experts for RTMS interfaces are functions that pop up multiple menus when you select an expert item. The items appearing on the menu are determined by means of an appropriate retrieve operation embedded within the expert code. You can select any number of items from the pop-up menu. In building up the query, these items are **or**ed together. For interfaces to non-RTMS databases, these experts behave somewhat differently. For instance, the default experts for interfaces to SQL databases present you with a type-in menu for specifying the value of a particular attribute. In contrast, RTMS interface experts examine the database in constructing the expert menu.

In the sample spec of Figure 7-1, there are no special-purpose, user-supplied experts. However, to replace a default expert, you would provide a list of the following form in the appropriate slot of the spec:

(*relation means-of-invoking-expert*)

or

(*relation attribute means-of-invoking-expert*)

where the means of invoking the expert is of the form:

(*new-expert-function parameter-1...parameter-n*)

Your definition of the new expert function is saved in a file of your choosing. You must store the pathname in a tuple in the NLMenu-interfaces relation (see paragraph 7.2.2.2). The file is then loaded at interface-construction time. For example, in a restaurant database that included a credit-card attribute indicating which credit cards a restaurant accepted, the various credit card types might be stored as a string (for example, *AE MC Visa* . . .). However, you might want the interface user to see a menu that contains entries for each possible type of credit card. This would require an expert function that does more than retrieve entries from the credit-card attribute in the database. The following entry in the New-Rel-Attr slot of the spec would be suitable:

```
(RESTAURANT CREDIT-CARD (credit-card-exp 'restaurant
'credit-cards))
```

NOTE: Every type of database interface includes lexicon entries that call certain invariant experts, such as the calculator expert (see paragraph 3.2.5.5). You do not need to take special action to ensure that such experts are included in the interface.

Edited Items This category includes old and new values for menu phrases so that you can replace automatically generated menu items with phrasings specifically customized for an application. The required format for an entry in the edited item slot is:

(*menu* "*old-phrase*" "*new-phrase*")

The *old-phrase* is the automatically generated phrase. Automatically generated phrases that reference database attributes typically include the relation name. This avoids ambiguity when several identical attributes coexist in more than one relation in the database. However, this sometimes leads to awkward phrasings (for example, *whose restaurant telephone number*) that you can eliminate if there is no ambiguity in a particular case. In general, any sort of customization is permitted. For instance, in the sample spec of Figure 7-1, the first edited item involves removing a reference to the part relation since the color attribute occurs nowhere else in the database. Menu, in this case the Attributes menu, refers to the menu in the default constraint window to which the phrase maps. Paragraph 7.2.4 includes indications of what sorts of phrases map to which menus.

Integrity Constraints

7.2.1.2 Certain relationships must hold among the portable-spec categories. These integrity constraints are summarized below:

- The union of the retrieval, insertion, deletion, and modification relations must be a subset of the covered-tables category. The union is a proper subset if some binary relationship table, which is otherwise hidden, is needed in a three-way join.

- The union of the relations mentioned in the attribute classification category (nonnumeric, numeric, and computable attribute lists) must be a subset of the covered-tables category. Not all attributes of all relations must appear in these lists; you can hide attributes so that they do not occur in the natural language interface.

- The nonnumeric, numeric, and computable attribute lists must be disjoint.

- The relations and attributes mentioned in the spec must conform precisely with actual database objects.

Violation of any of these integrity constraints results in error messages during interface generation. You must correct such errors before interface generation can proceed.

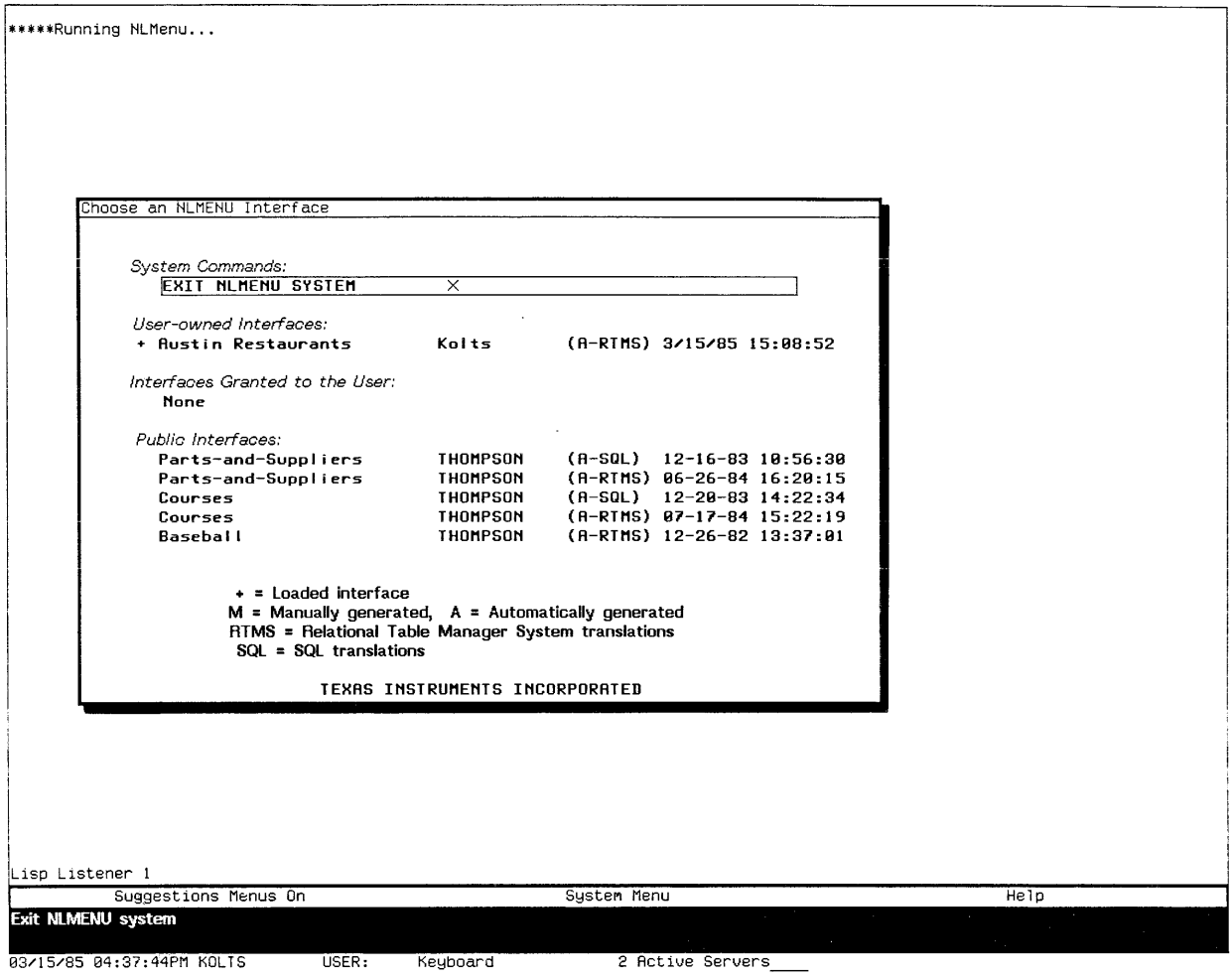
Managing NLMenu Interfaces

7.2.2 Interface generation requires a scheme for managing newly created interfaces. This scheme is discussed in the following paragraphs.

NLMenu System Menu

7.2.2.1 When you select NLMenu from the System menu with a suitable RTMS database loaded into memory, the system presents you with a menu of available interfaces. This menu is called the NLMenu System menu. An approximate NLMenu System menu appears in Figure 7-2.

Figure 7-2 Sample NLMenu System Menu



NLMenu interfaces are partitioned not only by application but by ownership as well. Architecturally, having several natural language interfaces owned by or granted to different users highlights the notion that natural language interfaces can be shared like other database objects (views and tables). Selective grant and revoke operations, described in paragraph 7.2.2.3, support interface sharing.

An alternative architecture for natural language interfaces might try to cover all of the applications of a user community simultaneously. That architecture would have the advantage of masking transitions into and out of different natural language interfaces. However, partitioning interfaces has the important advantage of keeping grammars relatively small, localizing application maintenance, and easily supporting customization of particular interfaces.

As can be seen from Figure 7-2, the NLMenu System menu presents you with choices among system-owned interfaces, user-owned interfaces

(those that the interface user created), interfaces granted to the user, and interfaces granted to the public (all users). Different interface users see different menus according to their access rights to various NLMenu interfaces. Two system-owned relations govern which interfaces users own and which interfaces they have access rights to. These relations also contain information for fully specifying a particular NLMenu interface.

NLMenu-Interfaces Relation

7.2.2.2 The NLMenu-interfaces relation contains 11 attributes. Each tuple in the relation describes one NLMenu interface. The tuple describing the parts-and-suppliers database discussed above is shown in Figure 7-3.

Figure 7-3 **NLMenu-Interfaces Relation for Parts-and-Suppliers Database**

```
① ("THOMPSON"
②   "parts-and-suppliers"
③    RTMS
④    nil
⑤   "grammar generated"
⑥   "sys:nlmenu-rtms-interface;s-p-rtms.grammar"
⑦   "sys:nlmenu-rtms-interface;s-p-rtms.lexicon"
⑧    s
⑨    default-panes
⑩    default-constraints
⑪   "06-26-84 16:20:15")
```

The attributes of this relation are:

① Owner — The owner or creator of the interface.

② Interface-Name — A string indicating the name of the interface to appear in the logo window.

③ Version — The target database. Currently-valid target databases are SQL and RTMS. This file controls what types of grammar and lexicon are generated.

④ Experts-File — The pathname of the file containing any special-purpose experts specified in the portable spec (see paragraph 7.2.1). If there are no special-purpose experts, a **nil** should appear in this field.

⑤ Portable-Spec-File — The pathname of the portable-spec file you created, or a special token string. The pathnames of the generated

grammar and lexicon are derived from this name. By convention, an extension of SPEC is typically assigned to portable-spec files. The token string *grammar generated* in this slot indicates to the system that the grammar and lexicon are already generated. If an actual spec pathname is stored in this slot and the grammar-file and lexicon-file slots contain **nil**, the system generates a new grammar and lexicon when an interface is loaded. Each occurrence of grammar generation includes a call that modifies this attribute and the grammar-file and lexicon-file attributes accordingly. Subsequent invocations of the interface can then load a compiled version of the previously generated grammar. To regenerate a grammar and lexicon, use the **modify-interface** function described in paragraph 7.2.2.4 to update the tuple describing the interface in the NLMenu-Interfaces relation. Then reselect the interface from the NLMenu System menu.

⑥ Grammar-File — The pathname of the source grammar or **nil** if no grammar has yet been generated.

⑦ Lexicon-File — The pathname of the source lexicon or **nil** if no lexicon has yet been generated.

⑧ Root — The root or grammar start symbol. SQL and RTMS interfaces both use *S* (sentence).

⑨ Panes — A variable bound to a pane description or a pathname for a file containing a pane description. SQL and RTMS interfaces use the default-pane description provided with the system. References to menus in generated lexicons refer to panes in the default-pane description.

⑩ Constraints — A variable bound to a constraint description or a pathname for a file containing a constraint description. Database interfaces typically use the default-constraint description provided with the system, although this is not necessary. For instance, customization of the Command menu might necessitate use of an alternate constraint-description.

⑪ Creation-Date — The time and date of interface creation, as formatted by the function **time:print-current-time**. This field is displayed on the NLMenu System menu but is not otherwise used.

Interface-Grants Relation

7.2.2.3 The interface-grants relation contains tuples indicating that an owner of an interface has granted interface access rights to another user. The attributes of this relation are:

- Owner — The owner who is granting access rights to the interface. The system can also own an interface. This is designated by placing `SYSTEM` in this field. In this case the interface becomes a System command on the NLMenu System menu.

- Interface-Name — A string specifying the name of the interface to which access is granted.

- User — The name of the user to whom access is granted. One valid user is PUBLIC, in which case all system users have access rights, and the interface appears under Public Items on the NLMenu System menu.

Creating, Modifying, and Dropping an Interface

7.2.2.4 When you complete a portable spec and are ready to build a new interface, you should call the function **nlm:create-interface** to insert suitable tuples into the NLMenu-interfaces relation (it may also be desirable to call the **nlm:grant-interface** function, see paragraph 7.2.2.5). Call this function with seven arguments:

(**nlm:create-interface** *interface-name version experts-file portable-spec-file root panes constraints*)

For instance, the tuple for the sample Parts-and-Suppliers interface is inserted with the call:

```
(nlm:create-interface "Parts-and-Suppliers"
                      'RTMS
                      nil
                      "sys:nlmenu-rtms-interface;s-p.spec"
                      's
                      'nlm:default-panes
                      'nlm:default-constraints)
```

The currently logged-in user name is inserted into the owner field of the tuple, **nil**s are inserted into the grammar-file and lexicon-file fields (indicating to the system that the grammar and lexicon need to be generated), and a date and time (in the format mentioned in paragraph 7.2.2) are automatically added to the creation-date field.

You can use the function **nlm:modify-interface** to change fields in a tuple in the NLMenu-interfaces relation (you can also do this by using the **rtmsmodify** or **modify*** functions directly). The function **nlm:modify-interface** takes keyword arguments and modifies only those fields of the tuple for which you provide arguments. Tuples are uniquely identified by a combination of their owner, interface-name, and version fields, although you can change any field by calling this function. The calling syntax is:

(**nlm:modify-interface** *owner interface-name version keywords*)

All attribute names except creation-date are recognized keywords (the creation-date field is automatically updated each time **nlm:modify-interface** is called). For instance, to regenerate an RTMS grammar and lexicon for the sample Parts-and-Suppliers interface, the appropriate call is:

```
(nlm:modify-interface "Thompson"
                      "Parts-and-Suppliers"
                      'RTMS
                      :portable-spec-file
                          "sys:nlmenu-rtms-interface;s-p.spec"
                      :grammar-file nil
                      :lexicon-file nil)
```

where "Thompson" is the interface owner.

The function **nlm:drop-interface** allows you to remove owned and granted interfaces from the database. Suitable tuples in the NLMenu-interfaces and interface-grants relations are deleted. (Again, interfaces are uniquely identified by a combination of their owner, interface-name, and version fields.) For instance, to drop the Parts-and-Suppliers interface from the NLMenu System menu, the appropriate calling syntax is:

```
(nlm:drop-interface "Thompson" "Parts-and-Suppliers"  'RTMS)
```

or

```
(nlm:drop-interface  'Thompson "Parts-and-Suppliers"  'RTMS)
```

NOTE: Commands for creating, modifying, and dropping an interface will appear on the NLMenu System menu in a future release.

Granting and Revoking Interfaces

7.2.2.5 The function **nlm:grant-interface** inserts a tuple into the interface-grants relation. You could also do this by using the **rtmsinsert** or **insert*** function directly. However, in these cases, you must also destroy the temprel relation, a temporary relation created by joining the NLMenu-interfaces and the interface-grants relations when the NLMenu System menu is built. The calling syntax is:

(**nlm:grant-interface** *owner interface-name user*)

The function **nlm:revoke-interface** deletes a tuple from the interface-grants relation (you could also do this by using the **rtmsdelete-tuples** or **delete*** functions). The calling syntax is:

(**nlm:revoke-interface** *owner interface-name user*)

At present, these interface granting and revoking procedures support one-level sharing. That is, they do not allow you to give grant-privileges to

other users so that they can share interfaces that they have access to but do not own with still other users.

NOTE: Commands for granting and revoking interfaces will appear on the NLMenu System menu in a future release.

Database Creation

7.2.3 The following discussion steps through the generation of a sample RTMS database. Note that an already-generated version of the database described below is included as part of the NLMenu RTMS Interface starter kit. All portable specs, as well as already generated source grammars and lexicons used in the database creation, are included with the system software. In addition, sample data items, as described in paragraph 2.4.2, are included in the file SYS:NLMENU-RTMS-INTERFACE;SAMPLE-DATA.LISP. For a more complete description of the various RTMS functions mentioned below, see the *Explorer Relational Table Management System User's Guide*.

1. Transfer to the NLM package. Do this by typing either of the following:

```
(pkg-goto "nlm")

(pkg-goto 'nlm)
```

2. Create a database environment where the database system storage structure is a heap, system implementation is a list form, relation implementation is a list-heap form, all RTMS-generated messages are turned off, and a directory is specified where RTMS saves the environment and database. The function call is as follows:

```
(rtms:define-environment 'nlmenu-env
                         'sys-sto "heap"
                         'sys-imp "list"
                         'rel-imp "list"
                         'rel-sto "heap"
                         'status nil
                         'errors nil
                         'warnings nil
                         'validity nil
                         'dir "sys:nlmenu-rtms-interface;")
```

3. Save the environment:

```
(rtms:save-environment  'nlmenu-env)
```

4. Define the database, named *NLMenu*, so that the value of `env` matches the environment name chosen when you defined the environment:

```
(rtms:defdb* nlmenu (dir "sys:nlmenu-rtms-interface;"
                     doc "NLMenu database directory"
                     env nlmenu-env))
```

5. Define the two relations necessary for interface management (definitions of these relations are included in the file SYS:NLMENU-RTMS-INTERFACE;SAMPLE-DATABASE.LISP):

```
(rtms:defrel* nlmenu-interfaces (owner
                                 interface-name
                                 version
                                 experts-file
                                 portable-spec-file
                                 grammar-file
                                 lexicon-file
                                 root
                                 panes
                                 constraints
                                 creation-date)
        (key (owner interface-name version)
             tuple-format (14 26 10 30 30 30 30 8 15 15 17)))

(rtms:defrel* interface-grants (owner
                                interface-name
                                user)
         (key (owner interface-name user)
              tuple-format (14 26 14)))
```

6. Define other desired relations.

The file SYS:NLMENU-RTMS-INTERFACE;SAMPLE-DATABASE.LISP includes relations to build the parts-and-suppliers database, a baseball statistics database, and a university-courses database. The relations constituting these three, logically distinct, databases are all defined in one actual database, the NLMenu database. This facilitates rapid switching of NLMenu interfaces, since you need not delete an old database from memory and load a new one into memory with each context switch. Once a relation is defined with **defrel***, you can insert tuples as follows:

(**rtms:insert*** *relation-name* **tuples**
(*tuple-1 ... tuple-n*))

Loading the SYS:NLMENU-RTMS-INTERFACE;SAMPLE-DATABASE.LISP file defines all the relations for the sample database and also inserts sample data. Next, save the database on disk:

```
(rtms:save-database 'nlmenu
                    '(dir "sys:nlmenu-rtms-interface;"))
```

If you change some tuples in a particular relation, you can save that relation with the call:

(**rtms:save-relation** *relation*)

You need not resave the entire database.

Once you have created and saved the database, you can restore it to memory in future Explorer sessions as follows:

```
(rtms:load-database* 'nlmenu
                     '(dir "sys:nlmenu-rtms-interface;"))
```

Because individual relations (other than the system relations) are demand-loaded, you usually request preloading of the relations in the NLMenu interface(s) with which you are currently working. A good way to do this is:

```
(mapcar #'rtms:load-relation  '(nlmenu-interfaces
                                interface-grants
                                relation-3
                                ...
                                relation-n)
        (circular-list '(dir "sys:nlmenu-rtms-interface;")))
```

This technique speeds up the initial access to a particular relation.

Default Constraint Window

7.2.4 A default screen configuration suitable for creating NLMenu constraint windows for typical database interfaces is included with the system and is shown in Figure 7-4.

Figure 7-4 Default Screen Configuration

```
(defvar *nl-default-panes*
        ((actions tv:kbd-command-menu-pane
                  :label "Commands" :item-list (""))
         (nouns tv:kbd-command-menu-pane
                  :label "Nouns" :item-list (""))
         (modifiers tv:kbd-command-menu-pane
                  :label "Modifiers" :item-list (""))
         (attributes tv:kbd-command-menu-pane
                  :label "Experts" :item-list (""))
         (comparisons tv:kbd-command-menu-pane
                  :label "Comparisons" :item-list (""))
         (features tv:kbd-command-menu-pane
                  :label "Attributes" :item-list (""))
         (operators tv:kbd-command-menu-pane
                  :label "Connectors" :item-list (""))
         (commands tv:kbd-command-menu-pane
                  :label "System Commands" :item-list (""))
         (input sensitive-window-pane :label nil)
         (logo tv:window-pane :label nil)
         (display scrollable-output-window
                  :label "Nlmenu Display Window")))


(defvar *nl-default-constraints*
        ((main . ((logo menu-strip commands input display)
                  ((logo .025)
                   (menu-strip :horizontal (.49)
                               (first-strip second-strip third-strip modifiers)
                               ((first-strip :vertical (.17)
                                             (actions features)
                                             ((actions .14)
                                              (features .86)))
                                (second-strip :vertical (.26)
                                              (nouns comparisons)
                                              ((nouns .5)
                                               (comparisons .5)))
                                (third-strip :vertical (.25)
                                             (attributes operators)
                                             ((attributes .75)
                                              (operators .25)))
                                (modifiers .32)))
                   (commands .075)
                   (input .08)
                   (display .33))))))
```

The pane description is bound to the variable **nlm:*nl-default-panes***, and the constraint description is bound to **nlm:*nl-default-constraints***. The specific menus established in the default screen configuration are as follows:

Menu-Name	*Label*	*Function*
Actions	Commands	Imperatives, such as *find*, *change*, *insert*, *modify*, and *draw*, are typically mapped to this menu.
Nouns	Nouns	Relation names and experts that involve an entire relation tuple are typically mapped to this menu. For instance, the phrases *parts* and *<a specific part>* might appear as items on this menu.
Modifiers	Modifiers	This menu generally contains modifying phrases based on database attributes. For instance, the phrases *whose color is*, *which are shipments of*, and *the new city is* might appear as items on this menu.
Attributes	Experts	Experts that involve specific database attributes typically appear on this menu. For instance, the phrases *<specific colors>* and *<specific part#>* might appear.
Comparisons	Comparisons	This menu is reserved for comparison predicates such as *less than* and *between*.
Features	Attributes	Database attributes such as *weight*, *supplier#*, and *status* typically appear on this menu.
Operators	Connectors	Database operators or English connecting phrases are typically mapped to this menu. For instance, the phrases *and*, *or*, *the total*, *(*, and *the number of* might appear.

An introductory discussion of screen configuration development appears in paragraph 4.8. For a more detailed account of the full constraint frame language, refer to the *Explorer Window System Reference*.

Formal Ideas Underlying Database Interface Generation

7.2.5 These paragraphs describe the formal ideas behind the **make-semantic-grammar-***version* and **make-semantic-lexicon-***version* methods.

Semantic Grammar Generation

7.2.5.1 Figure 7-5 illustrates a fragment of an RTMS semantic grammar (minus translation lists) for the parts-and-suppliers example.

Figure 7-5 Partial RTMS Semantic Grammar

```
s -->                      Find-PART PART-np
s -->                      Find-the-PART PART-attr-constr for PART-np
s -->                      Delete-PART PART-np
s -->                      Insert-PART PART-tuples
PART-tuples -->            new-PART-tuple (tuple-and PART-tuples)
PART-attr-constr -->       PART-attr (attr-and PART-attr-constr)
PART-attr -->              {PART# NAME COLOR CITY WEIGHT}
PART-np -->                {modifiable-PART-np PART-expert}
modifiable-PART-np -->     PARTs (PART-mod-constr)
PART-mod-construct -->     PART-mod-and (s-or PART-mod-constr)
PART-mod-and -->           PART-mod-base (s-and PART-mod-and)
PART-mod-base -->          {[left-bracket PART-mod-constr right-bracket]
                                PART-mod}
PART-mod -->               {[whose-PART-PART#-is PART-PART#-expert]
                            [whose-PART-NAME-is PART-NAME-expert]
                            [whose-PART-CITY-is PART-CITY-expert]
                            [whose-PART-COLOR-is PART-COLOR-expert]}
PART-mod -->               whose-PART-WEIGHT-is comparison-pred
                             numeric-expr
PART-mod -->               with-PART-WEIGHT between numeric-expr
                             between-and numeric-expr
PART-mod -->               |which-are-supplied-by-SHIPMENT-PART-SUPPLIER|
                             SUPPLIER-np
```

A formal description of the technique for structurally embedding the application-specific semantics into grammar rules can be given in terms of a *W-grammar*. A W-grammar uses two levels of rules, *hyperrules* and *metarules*. The hyperrules are templates that can be expanded into an infinite set of context-free rules. The metarules provide the slot fillers that instantiate these templates a finite number of times. In the NLMenu interface generator, a *core grammar* is a set of hyperrules, together with metarule constraints on how the substitutions occur. The portable-spec categories provide actual values for the metarules. For example, one of the core grammar hyperrules can be stated as:

rel-mod ⟶ whose-*rel-attr*-is *rel-attr*-expert

rel-attr is an element of the nonnumeric-attributes category.

Assuming a corresponding metarule, taken from the portable spec of Figure 7-1,

```
Non-numeric-attributes =
     ((part city) (part color) (part name) (part part#)
      (supplier city) (supplier name) (supplier supplier#)
      (shipment part#) (shipment supplier#))
```

nine instantiated grammar rules result from the substitutions—one for each (*rel attr*) pair in the nonnumeric-attributes category. For example, when *rel* equals *part* and *attr* equals *city*, the following instantiated grammar rule results from the substitution:

part-mod ⟶ whose-part-city-is part-city-expert

Thus, for the NLMenu-RTMS-interface system, semantic grammars are just W-grammars where the binding time of semantic information to the application-independent core grammar is at interface-generation time, before any parsing begins.

Semantic Lexicon Generation

7.2.5.2 Semantic lexicon generation proceeds in a manner analogous to grammar generation. Each lexicon entry schema, when instantiated with slot fillers from the portable spec, results in a lexical entry consisting of a 4-tuple with fields (*category menu-item menu translation*). An example of an instantiated RTMS lexical entry for the parts-and-suppliers database is:

```
(whose-PART-CITY-is "whose part city is" MODIFIERS
   lambda '(mem 'equ-db) city 'y))
```

In this example, the nonnumeric-attributes pair `PART CITY` was substituted into a lexical entry schema with slots for *rel* and *attr*. Using the value of the nonnumeric-attributes category mentioned in paragraph 7.2.5.1, nine lexical entries would result from the substitution.

In addition to acting as substitutions, portable-spec categories can act as existence constraints. Thus, if the portable-spec category *insertion relations* is empty, no lexical entry for *Insert* is present in the generated lexicon, and you see no *Insert* choice in the NLMenu selection windows, in accordance with the intent of the spec.

While the core grammar and lexicon are relatively small (on the order of 30 grammar rules and 45 lexical entries), the size of the resulting semantic grammars and lexicons is typically much larger, depending on the portable spec, which reflects the number of database relations and attributes covered in the interface. Instantiating the core grammar and lexicon with the portable-spec categories that describe the three relations in the parts-and-suppliers database results in 118 semantic grammar rules and 111 lexical entries.

GLOSSARY

A

abbreviatory conventions — The rules concerning the use of abbreviatory symbols (see abbreviatory symbol) within grammar rules. The abbreviatory conventions enable grammar writers to collapse partially similar grammar rules into one rule and thereby increase the efficiency of the grammar.

abbreviatory symbol — One of the three types of symbols used to collapse a number of rules into a single rule. The abbreviatory symbols used by Natural Language Menu (NLMenu) grammars are parentheses (for optional elements), braces (for alternative expansions), and square brackets (for groupings within braces).

C

category — See grammatical category.

context-free grammar — A grammar where every rule has the form:

$$A \longrightarrow X$$

where

A is a single nonterminal symbol

$\longrightarrow$ corresponds to English *can be* or *becomes*

X is a non-null string of elements that can include terminal and/or nonterminal symbols.

Neither null terminals nor cyclical rules are allowed in the context-free grammars handled by the NLMenu system.

context-free semantic grammar — A context-free grammar where each rule is augmented by a semantic component, which is the translation function.

D

daughter — All grammatical categories that appear on the right-hand side of a rule.

depth-first backtracking parser — A parser whose control structure applies rules to produce new states until either a parse is found or no rules apply. In either case, such a parser then backs through all choice points until all parses are found.

depth-first backtracking parser with !MORE feature — A parser with a normal depth-first backtracking control structure, except that backtracking begins when no rules apply and the special token *!MORE* is encountered. This feature enables the parser to leave a parse path before it is finished and pursue another. The parser takes up the original path later from the point at which it was left. This technique allows parsing to begin given only a subset of the input string. The NLMenu parser utilizes a control structure of this sort.

E

expansion list — A grammar rule component that lists numbers corresponding to the position of all left-corner or daughter elements of a rule associated with a given translation function. The expansion list resolves ambiguity regarding which translation list corresponds to which expansion of a rule when abbreviatory conventions occur in the rule. Unambiguous rules and rules without abbreviatory elements never need to contain expansion lists.

expert — A routine that handles a specific task, such as input and validation of literal values. Experts can be technically regarded as metalinguistic categories.

G

grammar — A set of rules that defines how words can be put together to form sentences in a language.

grammar compiler — The NLMenu system component that converts the external form of a grammar into an internal form that is convenient for the parser.

grammatical category — Any symbol in a grammar rule besides the abbreviatory symbols. Each terminal category also appears as the first part of some entry in the lexicon.

I

item — The second part of a lexical entry. It is the natural language form of the input as it appears in a selection menu. The interface user constructs an input sentence by selecting such items from a series of menus.

L

left corner — The first element on the right-hand side of a grammar rule.

left-corner parsing algorithm — See NBT parsing algorithm.

lexicon — The dictionary of the NLMenu system. It relates each terminal category of the grammar to a particular natural language phrase, its location on the screen, and some translation in the target language. The four parts of a lexical entry are the category, the item, the source menu, and the translation.

M

mother — The category on the left-hand side of a grammar rule.

N

nonselective bottom-to-top (NBT) algorithm — A parsing algorithm that uses several push-down stacks. When some reserved token, such as *END*, appears on the top of each stack, the string is recognized. In its simplest form, the NBT algorithm begins with the lexical categories of the words to be parsed on one stack and the start symbol on another stack. It then applies rules in a bottom-up manner, with the ultimate goal of constructing on the first stack a tree that has at its top the symbol *END* followed by whatever start symbol was on the second stack. This algorithm does not eliminate bad parse paths before trying them. (See SBT algorithm.)

nonterminal — A category in a grammar that appears only singly on the left-hand, and also without restriction on the right-hand, sides of grammar rules.

P

parse (n) The representation of the structure and meaning of a sentence that results from the act of parsing (see parse (v)). This representation often is a tree structure (see parse tree).

parse (v) To resolve a sentence into its syntactic and lexical components; that is, to analyze the structure and meaning of a sentence.

parser The NLMenu lexical analyzer. By using the grammar and lexical rules of the language, it can examine input sentences for meaning and validity.

parse tree A graphic representation of a parse of a sentence. Each rule used in the parse is depicted as a subtree that has the mother as the root node and the daughters as the child nodes. The root of the parse tree is the start symbol of the grammar, and the leaf nodes (those nodes with no children) are the terminals in the sentences.

portable spec A specification of a set of categories containing all the domain-dependent information necessary to specify a complete user-interface grammar and lexicon.

predictive parsing A parsing technique to predict the set of possible *n*th words of a sentence, given the first $(n-1)$ words. The parser accomplishes this by looking at the immediate goal of the parse state. To determine the words that can come next, the parser determines the set of all nodes reachable from that node as a left daughter. Then the parser finds the subset of such nodes that can dominate (be the origin of) lexical material.

R

reachability matrix A matrix that indicates whether each nonterminal node in the grammar can dominate each terminal or nonterminal category in the grammar where that terminal or nonterminal is on the leftmost branch. By means of a reachability matrix, bad parse paths can be selectively eliminated without being pursued.

root rule Any grammar rule that has the start symbol of the grammar as its left-hand side.

rule element Any grammatical category occurring in a grammar rule.

S

screen — A collection of windows.

screen description — A form containing all the information necessary for the NLMenu system to draw and manipulate a collection of windows.

selective bottom-to-top (SBT) algorithm — A parsing algorithm that is similar to the NBT algorithm. However, the SBT algorithm employs a reachability matrix to eliminate bad parse paths before trying them. The algorithm used by the NLMenu parser is of this sort.

semantics — The rules that govern what constitutes a meaningful statement in a language. Semantic rules determine whether sentences, while syntactically correct, are nonsensical in meaning.

source menu — The window pane to which a lexical item maps.

syntax — The set of rules that govern the legal order of words within a language.

T

target language — The language that is used by an application that has a natural language front end (interface) created by means of the NLMenu system. A front end is a package that acts as a buffer between the interface user and an application program. The data manipulation language of a database is an example of a target language.

terminal — A category that appears only on the right-hand side of some rule(s) in the grammar.

translation — An expression used by the translator in building a target string. A translation is part of every NLMenu lexical entry.

translation function — A rule associated with each NLMenu context-free rule indicating the order in which the translations of the symbols on the right-hand side of the arrow symbol of a context-free rule are to be combined. When a translation function appears in a rule containing abbreviatory elements, it can optionally include a number list indicating which expansion of the rule is to be associated with it.

W

well-formed grammar — A grammar that is free of syntactic and semantic errors.

window — A region of the screen. Typically, distinct programs can coexist by being run in separate windows.

word-at-a-time parsing — A parsing technique that enables the parser, given a word, to go as far as it can in analyzing the current input and also in predicting the set of next possible inputs. This technique increases the perceived speed of the parser because parsing continues while the user is entering input.

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z

USER RESPONSE SHEET

Our manuals are written for you. Please help us improve them by completing and mailing this form. List any discrepancy or other problem that you found in this manual by page, paragraph, figure, or table number in the space provided, and add any suggestions you have. Thank you.

Manual Title: _____

Manual Date: _____ **Today's Date:** _____

Your Name: _____ **Telephone:** _____

Company: _____ **Department:** _____

Company Address: _____

Product Purchased From: _____

Company **City** **State**

Location in Manual	Comment/Suggestion
_____	_____
	_____
	_____
_____	_____
	_____
	_____
_____	_____
	_____
	_____

The manual's good points: _____

What needs to be improved: _____

No postage necessary if mailed in U.S.A.
Fold on two lines (located on reverse side), tape and mail

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP

ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769

FOLD