8-1982

Recipe Design and Management in a Nutritional Data Base System

Robert G. Trenary

Western Michigan University

Follow this and additional works at: http://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons, and the Food Science Commons

Recommended Citation
http://scholarworks.wmich.edu/masters_theses/1750

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
RECIPE DESIGN AND MANAGEMENT IN A
NUTRITIONAL DATA BASE SYSTEM

by

Robert G. Trenary

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
August 1982
RECIPE DESIGN AND MANAGEMENT IN A NUTRITIONAL DATA BASE SYSTEM

Robert G. Trenary, M.S.
Western Michigan University, 1982

The Nutritional Data Base System (NDBS) is an interactive information system which provides the user the ability to create and process menus, recipes and food items. Three primary program modules are thus indicated along with modules to translate user requests and process file and directory information. A discussion of standard data base concepts and an application of these concepts to the processing of recipes in the NDBS system is presented. Future possible expansions are suggested and other similar information systems are discussed.
ACKNOWLEDGEMENTS

I wish to acknowledge the Computer Science Department for understanding; Kathleen Kostrzewa for her industry and diligence; my advisor Dr. Kountanis for his steady trust and friendship; and Kathy for enduring it all.

Robert G. Trenary
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
TRENNARY, ROBERT GLENN

RECIPE DESIGN AND MANAGEMENT IN A NUTRITIONAL DATA BASE SYSTEM

WESTERN MICHIGAN UNIVERSITY

M.S. 1982

University
Microfilms
International

300 N. Zeeb Road, Ann Arbor, MI 48106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ............................................................. ii
LIST OF TABLES ................................................................. vi
LIST OF FIGURES ............................................................... vii

CHAPTER

I. INTRODUCTION ................................................................. 1

II. DATABASE TERMINOLOGY .................................................. 5
    File Organization .......................................................... 7
    Record Structure .......................................................... 7
    Sequential and Random File Organization .............................. 8
    Query Language and Attribute Values ................................ 10
    Directory Structure ....................................................... 11
    Summary ........................................................................ 14

III. THE LOGICAL DESIGN OF NDBS ......................................... 16
    Food Queries ................................................................. 16
    Recipe Commands ........................................................... 19
    Primary Commands ......................................................... 20
    Secondary Commands ....................................................... 21
    Modular Design ............................................................. 22

IV. THE DATABASES OF NDBS .................................................. 25
    The Food Database ......................................................... 25
    Main Directory ............................................................. 25
    Food Description File ...................................................... 27
    The Pointer File ............................................................ 29
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS (CONTINUED)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nutrient File</td>
<td>31</td>
</tr>
<tr>
<td>Nutrient Pointer File</td>
<td>31</td>
</tr>
<tr>
<td>The Recipe Database</td>
<td>34</td>
</tr>
<tr>
<td>Main Directory</td>
<td>34</td>
</tr>
<tr>
<td>Header File</td>
<td>35</td>
</tr>
<tr>
<td>Ingredient File</td>
<td>35</td>
</tr>
<tr>
<td>Preparation Text File</td>
<td>37</td>
</tr>
<tr>
<td>Inverted Recipe File</td>
<td>37</td>
</tr>
<tr>
<td>Summary</td>
<td>38</td>
</tr>
<tr>
<td>V. RECIPE IMPLEMENTATION AND ANALYSIS</td>
<td>40</td>
</tr>
<tr>
<td>FASBOL</td>
<td>40</td>
</tr>
<tr>
<td>Recipe Implementation</td>
<td>43</td>
</tr>
<tr>
<td>The Create Command</td>
<td>43</td>
</tr>
<tr>
<td>The Find Command</td>
<td>48</td>
</tr>
<tr>
<td>The Print Command</td>
<td>50</td>
</tr>
<tr>
<td>The Reduce and Increase Commands</td>
<td>51</td>
</tr>
<tr>
<td>VI. DATABASE ADMINISTRATION</td>
<td>56</td>
</tr>
<tr>
<td>READ.FOR</td>
<td>57</td>
</tr>
<tr>
<td>RANFOR.FOR</td>
<td>57</td>
</tr>
<tr>
<td>PTR.PAS</td>
<td>57</td>
</tr>
<tr>
<td>RANFOD.FOR</td>
<td>58</td>
</tr>
<tr>
<td>SETUP.PAS</td>
<td>59</td>
</tr>
<tr>
<td>BLKDES.PAS</td>
<td>60</td>
</tr>
<tr>
<td>BLKNOD.PAS</td>
<td>60</td>
</tr>
<tr>
<td>COMPAC.FOR</td>
<td>61</td>
</tr>
<tr>
<td>TABLE OF CONTENTS (CONTINUED)</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>VII.  FUTURE IMPROVEMENTS</td>
<td>62</td>
</tr>
<tr>
<td>Partial String Matches</td>
<td>62</td>
</tr>
<tr>
<td>Expanded Applications</td>
<td>66</td>
</tr>
<tr>
<td>APPENDIX A - USER'S MANUAL</td>
<td>67</td>
</tr>
<tr>
<td>APPENDIX B - MENU PLANNING CODES</td>
<td>102</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>104</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE NO.</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>27</td>
</tr>
<tr>
<td>2.</td>
<td>30</td>
</tr>
<tr>
<td>3.</td>
<td>33</td>
</tr>
<tr>
<td>4.</td>
<td>36</td>
</tr>
</tbody>
</table>

Main Directory Entry
Node Model
Nutrient Substring of a Food Group Inverted Directory Record
A Recipe Header

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE NO.</th>
<th>Description</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>USDA Handbook #8 Description of Beets</td>
<td>17</td>
</tr>
<tr>
<td>2.</td>
<td>Tree Structure of Beet Food Group</td>
<td>18</td>
</tr>
<tr>
<td>3.</td>
<td>Top-Down System Design</td>
<td>23</td>
</tr>
<tr>
<td>4.</td>
<td>Record for Beets</td>
<td>28</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

The computer is an information processor. Information about food concerns every human being. The computer therefore offers an opportunity to simplify everyday tasks and improve the quality of many decisions based on nutritional information.

The Nutritional Data Base System (NDBS) is an information processing system using the Handbook #8 from the United States Department of Agriculture (USDA, 1975). This handbook contains nutrient values for approximately 2,500 foods. NDBS was designed and programmed as part of the master's thesis at Western Michigan University by Kathleen Kostrzewa (Kostrzewa, 1981) and the author.

A nutritional information system could serve many different users. Workers in food services such as those in school and businesses could automate much of their activity. In fact, such commercial systems exist. The Sentry system, a product of the CBORD group, is used at The Upjohn Company in Kalamazoo for managing the cafeteria system used by employees. Hospitals could analyze patient menus with computer tools and indeed a commercial food analysis service is used for kidney patients in the dialysis unit at Bronson Hospital in Kalamazoo.

There are two previous efforts at automated food information systems that deserve note. Eckstein (1966) was interested in various algorithms which would plan optimum menus based on various meal characteristics including color, frequency of serving, and the usual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
nutrient values based on USDA Handbook #8 (USDA, 1975). The program was designed for institutional uses. Several measures of meal characteristics were established on the basis of surveys of students who were representative of the population which the program serves. So foods and meals were given "desirability" values and the program took these values into account when designing a menu. Eckstein summarizes various strategies which can be used to plan menus (such as an optimization goal in a linear programming context) and then she demonstrates a strategy of her own based on ranked attributes of meals.

Balintfy (1967) discusses an institutional system which includes inventory and price considerations as well as program routines which issue stock orders and inventory reports. This system is a complete data processing application appropriate for a large institution where production preparation and planning are necessary.

There have been many different attempts at menu planning systems since these two. Pfeiffer (1970) uses a simulation method of menu selection arguing that linear programming models are too restricted to accommodate the complexities of menu selection. A number of applications in hospital environments have been considered in attempts to limit cost (Andrews, 1969), enhance patient care (Ford, 1979), and reduce workloads (McLaren, 1972).

Both Eckstein and Balintfy designed batch systems for institutional use. NDBS was designed with a less specialized use in mind—the home user. It is believed that even the daily tasks of menu design and recipe analysis could be accomplished more efficiently with computer aid.
This choice of user narrows but does not severely restrict the application of NDBS. System design allows for expansion without loss of the present NDBS functions. But the choice of the home as the primary focus of NDBS does force some immediate design decisions. One such choice reflects the need for an interactive system. Certainly a home user expects (and is best served by) online service. The notion of batch processing and delayed service is antithetical to the kind of application originally conceived when NDBS was proposed. For this reason the programming of the system was done on a Digital Equipment Corporation PDP-10 computer using the FASBOL programming language. Both the computer and the language are appropriate for an interactive application.

Another design choice that home use predicates is revealed in the query language of NDBS. The user is assumed to be a computer novice. The command language of the system must therefore be simple, both in syntax and scope. The simple syntax does not necessarily imply simple programs. Instead, the programming language used helps to provide substantial processing which might otherwise be expected of the user. Kostrzewa (1981) discusses the menu design and management of NDBS.

This thesis is concerned primarily with the design and management of recipes in NDBS. Chapter 2 is a discussion of database terminology which is used in the remainder of the thesis. Chapter 3 explains the query language and the resultant software design of NDBS—the various modules and their relationships. Chapter 4 discusses the various databases and file structures in the system.
Chapter 5 explains in detail the implementation of the recipe module of NDBS with emphasis on advantages of FASBOL. Chapter 6 describes programs which are available in NDBS for database administration. Chapter 7 is a discussion of possible future expansions of NDBS.
CHAPTER II

DATABASE TERMINOLOGY

A computer is often used to store and process information. A more or less fixed set of information is a database and the software used to manage the information is a database management system (Uhlmann, 1980). The management of the information includes such functions as storage, retrieval and modification. Invariably the database is stored on a secondary storage device external to the main memory of the computer. Either the amount of information or the need to store the current information while the database system is not executing require external storage. Usually a high-speed secondary device such as magnetic disc or drum is used so that online access of the database is possible.

There are always two (usually distinct) views of a database. One reflects the viewpoint of the user of the information. This "logical view" of the information is abstract and does not require knowledge of the actual arrangement of the information as it is stored in either main or secondary storage. It is the "physical view" of the database which describes the way in which the abstract units of information (such as the name of an employee and all dependents) are represented on the physical devices. The database system allows the user to work with a particular logical view of the database independent of the physical organization of the data.
The logical units of information in a database are the datum, field, record, and file (Freeman, 1975). The smallest unit is some particular logical fact—a datum. Names and pay rates are examples of data. A user normally assumes that movement of information between main memory and secondary storage occurs in larger units: Records. A record consists of several data items. Each data item of a particular record is contained in a field of that record. Often a record and its fields are given logical names. A PERSON record might have fields known as NAME, PAY, and DEPENDENT. Formally, these logical field names are known as the attributes of a record. A particular record has specific values for its attributes. So a PERSON record could be described as having NAME = "JOHN DOE", PAY = $4.53, and DEPENDENT = "LOUISE" as the values of its attributes. A user knows one record from another by the data in particular fields. Put another way, a user knows a record by the values of its attributes. A file is a collection of records (usually related in some logical way). The assumption from the logical point of view is that all the records in a file have a uniform structure although this does not have to be true at the physical level.

The physical view of a database assumes that a block is the smallest unit of information that can move between main memory and secondary storage. The block size is a function of the physical device used to store the data. A single read from a magnetic disc transmits the information on one "sector" of a disc track. A logical record may be larger or smaller than the size of a block. Again, the database system must facilitate the logical view of the data so that the
movement of a record can be accomplished without the user needing to know that several blocks or a part of a block actually comprise the record.

File Organization

Record Structure

Although the records on a file are seen as having a uniform pattern, they are not all necessarily of the same (physical) size. One can describe a record and its fields as fixed or variable size.

For example, a file containing personnel information might contain records, each of which contains only a name field and a salary field. This would be a file of fixed size records. But if information including the names of all dependents of each person was also part of the record, then the records would be of variable length—a person might have no dependents or several dependents.

When variable length records are maintained, internal information about each record must be stored in order to identify a particular record. This can be done by storing information indicating the number of fields of a certain type—a count field might be included to indicate the number of dependent fields. Another method would be to use a trailing value as an indicator of the terminating field—a dependent named "NONE" might always follow the fields storing names of actual dependents. Both of these strategies can be used to advantage in a particular application.

Just as there can be variable length records in a file, there can be variable length fields in a record. Again, some method of indicating
the size of a field is necessary and a count indicator preceding the field or a special flag trailing the field can be used.

A name field in a personnel record might be fixed at 15 characters. Or the name field might be variable length with a preceding count field indicating the number of characters in the following name field. Alternately, an asterisk might be included as a trailing character in every name.

Record organizations can range from the simple—with fixed size records of fixed size fields—to the complex file of variable size records including variable size fields. The choice of record organization can depend on many factors. Simple organizations can be programmed and managed easily, demanding less programmer and computer time. More complex organizations require less storage and can reflect more accurately the nature of the information at hand (reality is rarely in a "fixed size" format).

**Sequential and Random File Organization**

It was stated earlier that database systems invariably store information on secondary storage and then bring information to main memory as needed. The mode of access of a record in secondary storage can be categorized as sequential or random.

The sequential accessing of file records is a function of the order of the records on the file. There is a first, second, ..., and last record. At any time a particular record is the next one which will be accessed. The records can be accessed only in a particular order. If we are ready to access record 3 and we wish to
access record 10, then records 3, 4, 5, 6, 7, 8, and 9 must be accessed first. This physical ordering of the records can have a logical meaning. The records might be organized in alphabetical order. But updating the physical ordering of a sequential file can be difficult, particularly if insertions and deletions are required, because the physical records must be moved appropriately.

A fact about file accessing should be noted here. The time to access a particular block on a file (i.e. the transmission time between memory and secondary storage) is at least three orders of magnitude greater than the time required to move data between memory and the processing unit. This relative slowness of secondary storage reflects the mechanical aspects of secondary devices. There is a rotating disc or drum and usually mechanical movement of a read/write head. No such mechanisms exist to move data between the CPU and main memory. The effect is that perhaps 1,000 machine instructions can be executed while a single record is being retrieved from a file!

The performance of a database system (measured in terms of the response time to the user) is therefore practically a measure of how well the number of secondary storage accesses have been minimized.

Sequential access can thus be seen as too slow for many applications. Accessing records uselessly in order to retrieve a particular record can slow an information system. Random access is the alternative accessing method. Random access allows a particular record to be read or written independently of other records in the file.

An important consideration when using random files is that, unlike sequential files, the "next" record to be accessed is not known
by the operating system. Instead, the program accessing the file must supply information to the operating system which identifies the record to be accessed.

A program which uses random access must provide two processes in order to retrieve information for a user:

1. The user must be able to request a desired record and a vocabulary must be provided with which to make such a request.

2. The program must be able to translate a user's request and identify a particular record on a file.

Query Language and Attribute Values

The first process allows a user to request information using the logical view and the associated logical terms. The user does not (and would rather not) have to request information by specifying physical information such as a particular disc sector and track. In fact, the user would prefer not having to know about the actual file organization. Instead, the user cares about the information on the file and makes requests according to currently needed information. The process of "requesting" can be very complex. A personnel record might be needed by the value of the name field (NAME = "LOUIE"), by the absence of certain values (DEPENDENTS = "NONE"), or by ranges or combinations of data values ("PAY IS LESS THAN 4.20 AND GREATER THAN 3.40").

Implicit in these examples of requests are two concepts:

1. Attributes and values of records,
2. Query language.

A file contains records, each of which has attributes. These are just logical names for specific fields in the record. Each record has a value for each of its attributes and by virtue of attribute values a record can be requested. Usually a record is requested in predictable ways. A personnel record might be requested by the value of a name field but it would probably not be wanted because of the value of the ones digit of the pay field. The likelihood of a particular request reflects the nature of the information in the record and not the ease or difficulty of either mode of access. The allowable requests for a record in a file are realized in the vocabulary provided to the user. For example, a user probably should be allowed to type

```
FIND NAME = "JOHN SMITH"
```

if the system is providing useful information. But

```
FIND ONES DIGIT = 4
```

probably does not provide useful information and will likely not be allowed as a request. The vocabulary and syntax which is provided to the user to request information is termed the query language. A query language determines what information can be retrieved for the user. As we shall see, the query language influences all of the structures of an information system.

Directory Structure

The second process which must be provided when using a random file organization allows the program to take a user's request and identify the physical blocks which contain the requested information.
This is accomplished by storing identifying information associated with each block (such as a record number) and associating that information with particular attribute values.

The storage of the identifying data for the records in a file is known as the file directory. The identifying data is possibly a record number or a block and record number. A general term for the identifying data is "pointer."

An information retrieval can thus be seen as an event initiated by a request in the form of an allowable query. The request must ultimately be translated to identify the records on a file where the desired information exists. It is the directory which provides the interface between the query and the record identifier.

A directory can formally be seen as a set of (attribute value, record identifier) ordered pairs. It should be noted that the relation implied is not necessarily a mathematical function but rather a relation—many records might have the same attribute value, just as many employees might have the same pay.

The variety of information needs and the resulting variety of possible queries suggest that directories can be designed in many ways. The choice of design attempts to minimize the number of accesses of secondary storage. The nature of the actual stored data (the frequency of particular data values) can also influence the choice of directory design.

Any directory can be seen to fall between two extremes. The directory may contain every record identifier associated with a particular data value. For example, for the attribute value, PAY = 4.25,
every record number of a record with a pay field value of 4.25 may be
stored in the directory. A directory in which all records with a
particular attribute value are specifically identified in the directory
is termed an inverted directory.

At the other extreme is a directory which contains the identifier
of a list of records, each pointing to another record with the common
attribute value. The directory contains the information pointing to
the head of a list of records and the program moves from record to
record by means of information stored in the records themselves.
This is a list organization of a directory.

Any directory may be described on the basis of how much of the
information identifying records with a particular value is contained
in the directory itself. The information is either in the directory
or in the records of the file. The extremes of this spectrum are the
inverted and list directory structures.

There are also directories which are partially inverted or not
completely list (multilist). The choice of directory design is deter-
mined by the space which can be allotted to the directory (pointers
to 1,000,000,000,000 records could not be practically stored in a
directory that is kept in main memory) and the time required to traverse
a list of records, each requiring another disk access. It is fair to
say that only in some unrealized ideal situation is there a "correct"
solution. Generally, tradeoffs of directory space versus accessing
time are made and a situation is optimized as well as possible.

All directories can be seen as specific examples of a general
directory format. Hsiao and Harary (1970) describe a "generalized

directory" as a set of entries \((K, h, n; a_1, a_2, \ldots, a_h)\) where

- \(K\) is the attribute value or "key" requested,
- \(h\) is the number of lists for this key value,
- \(n\) is the total number of records in the file that contain \(K\),
- \(a_1 \ldots a_h\) are the addresses of the heads of the \(h\) lists.

This directory structure is generalized because both inverted and list directories can be seen as specific examples of the generalized structure and any arrangement that falls between these two can also be represented. For example, an inverted directory is one in which \(h = n\). That is, there are \(n\) records with the key value \(K\) and each is seen as a list of length 1. The values \(a_1\) through \(a_h\) are merely the record identifiers stored in the directory itself. At the other extreme, a list structure could be represented if \(h = 1\). Then there would be only one list of records and only one head address. This is precisely a list structured directory.

Summary

This chapter has introduced terms which will be used in the remainder of this thesis. Any database system can be described in terms of the requests for information that are implemented via the query language with a particular logical view of the information. Such requests must be translated into requests for specific records which are made up of (possibly several) physical blocks of secondary storage. The directory is the interface which allows the requested attribute value (the key) to be associated with information identifying the
physical location of the requested information.

The variety of record and file structures is unlimited. A particular choice of structure attempts to minimize the physical storage required for the database and also attempts to minimize the time required to access information by limiting the number of accesses of secondary storage, since these accesses are so much slower than the time required for processing information in main memory.
CHAPTER III

THE LOGICAL DESIGN OF NDBS

Uhlmann (1980) considers a database from the viewpoint of the database administrator, the creator of different subschemes with which to manipulate the database. Each subscheme corresponds to the logical view of a particular user community. NDBS as originally conceived assumes only one such user view. This subscheme is supported by the query language supplied to the user. The query language consists of three major subdivisions, each of which corresponds to a major module of the NDBS software.

The requests from the user of NDBS can generally be said to be requests about food. Upon closer inspection possible requests can be seen as of three types: Food, recipes, and menus. There are, therefore, three independent subsets of commands available to the NDBS user. The first choice a user makes, via the query language, decides which set of commands will be invoked.

Food Queries

The first category of requests concerns specific foods. Inquiries concerning nutritional values of a particular food are food requests (e.g. "How many calories are there in carrots?). The nature of the food database does not allow as simple a search command as might be desired. This is a reflection of the contrast between what might be
considered "normal" food names and the food names in the Department of Agriculture Handbook #8 (USDA, 1975). A user who desires to know about something as simple as beets is faced with eight different choices for beets.

Beets, common, red:
1. raw
2. cooked, boiled, drained

Canned:

Regular pack:
3. solids and liquid
4. drained solids
5. drained liquid

Special dietary pack:
6. solids and liquid
7. drained solid
8. drained liquid

Figure 1. USDA Handbook #8 Description of Beets

The food query language provides a vocabulary which allows a user to identify and investigate a particular food group. Each food class displays a "tree structure" and the query language assumes this model of the data.

Harary (1969) defines a tree as a graph in which no cycle exists. That is, no path can be traced that visits a node twice. In addition, a particular node is called the root of the tree and all paths that begin at the root terminate at nodes called leaves. In a food tree the leaf nodes correspond to the food in the handbook. The name of the
food can be constructed by combining the names of the nodes visited beginning at the root and ending at the leaf.

![Tree Structure of Beet Food Group](image)

**Figure 2. Tree Structure of Beet Food Group**

The FOOD module commands which allow a user to select and traverse a food tree are:

- **SELECT** — allows a user to choose a particular "tree."
  
  If there is some ambiguity in the food requested, a dialog allows the user to specify a particular food from possible food classes.

- **UP** — allows a user to move up the tree. So if the food currently being investigated is Beets, canned, regular pack, an UP command will put the user at the Beets, canned node from which both regular and dietary pack nodes of the tree can be reached via DOWN commands.

- **DOWN** — moves the node pointer down by using the convention that all nodes below are ordered and allowing the user
to request DOWN 2 and move to the second of the nodes below the current node.

**LIST** — allows nutritional information about any of the nodes at the current level or below to be displayed for any of 21 nutritional values stored for each food. So LIST CALORIES is a command to print the caloric value per 100 grams of all foods in the current food class represented by the node pointer.

**POINTER** — prints the name of the current node and the nodes accessible via DOWN commands.

Recipe Commands

Another class of NDBS requests concerns groups of food items and associated quantities. Such groups of foods form the NDBS model of a recipe. Associated with the foods comprising a recipe is a set of distinguishing attribute values such as name, menu type (e.g. LUNCH), and a set of text which describes the preparation of the meal.

The subset of NDBS requests concerning recipes constitute the **RECIPE commands**. Implicit in the dynamic commands is the notion that a recipe is currently being dealt with—created, modified, or investigated. The commands of the RECIPE module may therefore be seen as primary or secondary. Primary commands are those which may change the current recipe. Secondary commands always assume a current recipe exists. Primary commands are CREATE, FIND, and PRINT.
Primary Commands

CREATE causes a dialog to occur which allows the user to specify the distinguishing attribute values such as name and menu type. The ingredients of a recipe and the associated units are then inserted into the recipe until the DONE command is issued (no food named DONE exists so the command may be distinguished from the inserted food). The preparation text is then inserted into the recipe and the user is free to use any of the secondary commands.

FIND is a primary command which can be used to retrieve a recipe or to search the recipe database for recipes of a given type. The command has two forms:

1. FIND "name"—assumes that a recipe with the given name has been stored. The recipe is fetched for investigation. A PRINT command will then allow the user to display the recipe. Any secondary command can now be used.

2. FIND key AND key AND ... AND key—is a form of the command which will allow the database to be searched for the recipes with a conjunction of attribute values. The current recipe is not replaced with this form of a FIND command. The names of all recipes which satisfy the request are returned when the FIND command is used.

PRINT lets the user display the information about a recipe. If the form PRINT "name" is used, the requested recipe becomes current.
Secondary Commands

The secondary commands are ANALYZE, CHANGE, INSERT, DELETE, REDUCE, INCREASE and REPLACE.

ANALYZE displays all nutritional attribute values of the current recipe on a per serving basis. If only one attribute value is needed, ANALYZE attribute value may be used as a command.

The commands which will modify the current recipe can be thought of in two classes: Those that modify either fixed attributes or preparation text, and those that modify ingredients.

CHANGE is used to modify any fixed attribute. The form of the command is

CHANGE attribute name

For example, typing the command

CHANGE MEAL TYPE

will allow the user to replace the current meal type value.

PREP invokes a module which is essentially a very small text editor. Text is considered in terms of lines and a user can replace a given line of text. An escape character ends the text editing.

Whereas fixed attributes are modified in a straightforward manner, the ingredient attributes change more often and the functions provided by the query language reflect that dynamic quality.

INSERT, DELETE, and REPLACE are used to change a particular ingredient. The forms of the commands are

INSERT food name
DELETE food name

REPLACE food name

and the appropriate food and/or quantity is then requested of the user.

The REDUCE/INCREASE command is used when the particular ingredient to be altered is not chosen by the user. Instead, the attribute to be changed is known (e.g. calories) and a suggested choice is made to be accepted or rejected by the user. The form of the command is

REDUCE attribute AND attribute ... AND attribute

INCREASE attribute AND attribute ... AND attribute

There is no guarantee that a REDUCE/INCREASE command can be satisfied. The command is provided as an exploratory tool that relieves the user of the task of searching the food items exhaustively. The commands of the FOOD module are also available to the RECIPE module. A user might wish to know about a specific food before inserting it in the current recipe. So the SELECT command of the food module and the subsequent pointer movements through the tree structure are available.

The third set of food requests available to the NDBS user treats groups of recipes as a unit—a menu. The MENU module provides extensive menu analysis and planning functions. A complete discussion of the strategies and abilities of the MENU module is provided in Kostrzewa (1981).

Modular Design

Some comments about design strategy of NDBS are in order.

Figure 3 shows the overall design of NDBS. The design can be seen to
Figure 3. Top-Down System Design
be "top down" in the following sense: The query language was chosen first, indicating the desired NDBS functions. Then a modular organization of software was designed. Then the actual databases (files and directories) were designed and finally software was written.

Clearly the modules correspond to the query language provided to the NDBS user. Associated with each of the major divisions of NDBS (FOOD, RECIPE, and MENU) are databases and accessing modules: File and directory structures which attempt to implement the query language in the most efficient manner (in terms of speed of access and amount of storage). Each module can be seen logically in terms of the commands and associated functions provided to the user. Alternately, the modules can be seen in terms of the associated software: The data, as it is stored externally in files and the program which exists internally.

The modular design thus provides the interface between logical user view of the system and the underlying software. Further, modularity and the associated advantages of programming (small isolated problems and the resulting easier debugging) can be maintained. Functions can be added to the system one at a time. Also, the advantages of top down design allow implementation decisions to be postponed until necessary. The choice of programming language was not made until well into the project—modules had then been considered and an advantageous choice of language and file structures was made.
CHAPTER IV

THE DATABASES OF NDBS

There are three main NDBS databases corresponding to the three modules of the system: FOOD, RECIPE, and MENU. This chapter describes the FOOD and RECIPE databases. The MENU database is described in Kostrzewa (1981).

The FOOD Database

The FOOD database is designed to support the tree structure of the logical model of the food data— the tree structure implicit in the query commands UP, DOWN, and POINTER. There are therefore specific directories and files to accomplish this goal.

Main Directory

This set of data supports the SELECT food command. Each record consists of a 15-character food group name (the name of a tree) and the associated pointers which allow information about a given tree to be found. Every SELECT command must use the main directory. Other system functions also need the information about food groups. Therefore, the main directory is kept in main memory whenever the NDBS system is executing instead of reading the directory into memory when information from it is needed.
The size of the main directory should therefore be minimized. This can be done by limiting the number of entries in the main directory and by limiting the size of the directory entry.

Since the food information is based on the relatively static Handbook #8 (USDA, 1975), the number of food trees is known—approximately 370. The number of food groups can be modified by changing the stored version of the handbook and reconfiguring the system. And new food groups have been created to change the size of records and files other than the main directory. But since the most commonly investigated foods are not easily known until usage statistics have been gathered, it is not easy to reduce the size of the directory by limiting the number of entries (leaving some records on disk). Thus the number of main directory entries remains at about 370.

The size of the directory entry can be reduced, however. Associated with each 15-character name in the directory are three records: A description file record consisting of character strings used in the food names, a tree file record which represents the internal structure of the food group, and a pointer to an inverted directory of the foods in the particular food group. The main directory allows a user to find these records by using the name of the food group as a key. The main directory entry thus consists of a name and three pointers, positive integers which indicate block and record numbers. In order to conserve space these integers have been compacted so that the main directory entry takes only two words of memory (Table 1). When the NDBS system is initialized, the main directory is read into memory from a sequential file where the records are organized.
### Table I

**Main Directory Entry**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Number of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>108</td>
</tr>
<tr>
<td>Initial Item Number</td>
<td>12</td>
</tr>
<tr>
<td>Number of Items</td>
<td>12</td>
</tr>
<tr>
<td>Nutrient Block Pointer</td>
<td>8</td>
</tr>
<tr>
<td>Nutrient Record Pointer</td>
<td>4</td>
</tr>
<tr>
<td>Description Block Pointer</td>
<td>8</td>
</tr>
<tr>
<td>Description Record Pointer</td>
<td>6</td>
</tr>
<tr>
<td>Tree Block Pointer</td>
<td>8</td>
</tr>
<tr>
<td>Tree Record Pointer</td>
<td>6</td>
</tr>
<tr>
<td>Future Expansion</td>
<td>8</td>
</tr>
</tbody>
</table>

TOTAL 180 BITS = 5 words of Memory

alphabetically. When a particular food tree is investigated, the appropriate main directory record is unpacked so that the pointers to the description, tree, and nutrient pointer files can be used.

**Food Description File**

Associated with any particular food group is a tree consisting of nodes, each with a name. Many of these names contain repeated character strings. For example, the phrases "drained solids," "drained liquid," and "solids and liquid" are each repeated in descriptions of
different beet nodes. The food description file consists of records, each of which corresponds to a food group. Any unique character string used to describe a node of a food group is stored only once in a description file record.

The description file record consists of a single string of characters comprised of the unique description strings separated by asterisks. Figure 4 represents the record associated with Beets.

*BEETS, COMMON, RED*RAW*COOKED, BOILED, DRAINED*CANNED*
REGULAR PACK*SOLIDS AND LIQUID*DRAINED SOLIDS*DRAINED LIQUID*SPECIAL DIETARY PACK (LOW SODIUM)*

Figure 4. Record for Beets

A particular phrase is now identified by pointing to the record and character in that record where the phrase starts. The trailing asterisk acts as a delimiter to end the phrase. The pointers to the record and to the first character in the phrase are kept, respectively, in the main directory and the tree file record.

The description file is a random access file with each block of the file consisting of 640 characters. Since a description record for a food group is usually not 640 characters and may not be even close to 640 characters, a significant problem of efficient storage exists. It would be wasteful to use a block of disc storage to store only one description record of 200 characters. The allocation of records to blocks would be done to minimize wasted disc storage without increasing the program complexity and retrieval time considerably.
The strategy used in NDBS to allocate records to blocks assumes that the records are ordered according to length, from smallest to largest. It is further assumed that records that are put in the same block will have the same length in order to extract a record from a block. It is also assumed that each record will be stored completely in one block.

The ordered list of records is then allocated to blocks by packing as many records of identical length as possible into the same block. When it is necessary to allocate blocks for records of different sizes, the smaller records are padded with blanks to the largest record size. This policy was applied beginning with the smallest records.

An allocation method that minimizes wasted disc space is not known except an exhaustive consideration of all possibilities. The problem is similar to the "packing problem," suggesting that a trial and error method is not practical (Baase, 1978).

The Pointer File

Records on the pointer file are used to follow the logical structure of food groups presented to the NDBS user. Each node in a tree falls into one of three categories:

1) it is a root node and has no parent or sibling,
2) it is a leaf node and has no child,
3) it is both a parent and a child.

A uniform model for a node is used to store nodes of all types, with field values indicating the type of node.
Table 2

Node Model

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child Pointer</td>
<td>4</td>
</tr>
<tr>
<td>Sibling Pointer</td>
<td>4</td>
</tr>
<tr>
<td>Point</td>
<td>4</td>
</tr>
<tr>
<td>FLAG1</td>
<td>1</td>
</tr>
<tr>
<td>FLAG2</td>
<td>1</td>
</tr>
</tbody>
</table>

*Child field* is a pointer to the node that is the oldest child of the current node. Note that an ordering of children is implicit in this language, an ordering which is not necessarily in a tree structure.

*Sibling field* is a pointer to the next youngest sibling of this node.

*Point* is a pointer into the description file record indicating where the description string corresponding to this node begins.

Since each node need not have siblings and a leaf node has no children, these pointers are used in special ways for such nodes. FLAG1 and FLAG2 are bits that are used to indicate these special cases.

FLAG1—If 0 then normal pointer, if 1 then the sibling pointer indicates the parent node.

FLAG2—If 0 then normal pointer, if 1 then the child pointer is a pointer to the record in the nutrient file
(Handbook #8, USDA, 1975) which contains nutrient values. Note that a root can be identified as a node with FLAG1 set to 1 and the parent pointer equal to 0 (i.e. no parent exists).

This arrangement of pointers is actually an implementation of a general tree as a binary tree (Stone, 1975). To trace from a parent through all the children, the oldest child must first be found and then the sibling pointers followed until a node with FLAG1 set to 1 is found. Note that the addition of two flags (1 bit per flag) causes no pointer to be wasted. This arrangement allows all nodes to be represented uniformly and efficiently.

The tree file is a random access file. Since the number of foods in a food group (hence number of nodes in such a group) is variable, the size of a tree file record is variable and the same allocation problem that exists for the description file exists with the tree file. The same allocation strategy was used for both files.

**Nutrient File**

The records in this file contain the nutrient values in Handbook #8 (USDA, 1975) associated with particular foods. These records are all of the same size and are allocated two per disc block on a random access file.

**Nutrient Pointer File**

The REDUCE/INCREASE commands of the RECIPE module require that foods in a food group be searched on the basis of particular attribute values. A cheaper asparagus or lower calorie beet might be needed. This function can be accomplished with the previously described files.
But the search would require a disc access of the nutrient file to investigate the attribute values. Instead, a directory of foods that have attribute values in a certain range exists. This file is the nutrient pointer file.

For a particular attribute the records in a food group are divided into ranges of attribute value. The program can then attempt to find a lower cost asparagus by looking in the directory for a record number that is in the next lowest cost range. Note that while this method does not allow the exact cost to be known (since ranges of cost, not exact costs, are stored in the directory), no access of disc is required to find a record with a lower attribute value and a significantly different value is more likely to be found rather than the next lowest cost (which could differ by a small amount).

A nutrient pointer record is represented as a string of characters. A substring exists for each attribute for which the directory is inverted as well as a substring which is a header that contains internal information about the record.

The size of attribute value ranges are decided by taking the difference between the largest and smallest attribute values in a food group and then dividing by the number of attribute value ranges that are desired. The choice of the ranges of attribute values assumes that the attribute values within a particular food group approximate an arithmetic sequence. If one food has a value that is very much different than the other foods, then all foods might be bunched into one range of values. This does not occur often and the phenomenon can be discouraged by increasing the number of ranges that exist in
### Table 3

#### Nutrient Substring of a Food Group Inverted Directory Record

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food Name</td>
<td>20</td>
</tr>
<tr>
<td>Number of Items</td>
<td>5</td>
</tr>
<tr>
<td>Number of Attributes Inverted</td>
<td>5</td>
</tr>
<tr>
<td>Number of Divisions of Attribute Values</td>
<td>5</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>35 characters</strong></td>
</tr>
</tbody>
</table>

#### Header of an Inverted Food Directory Record

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute Name</td>
<td>10</td>
</tr>
<tr>
<td>Lowest Attribute Value</td>
<td>8</td>
</tr>
<tr>
<td>Increment Size (Size of Range)</td>
<td>8</td>
</tr>
<tr>
<td>Sequence Numbers of Lowest Record in a Range</td>
<td>5 characters per sequence number</td>
</tr>
<tr>
<td>Sequence of Records Ordered Attribute Value</td>
<td>5 characters per record number</td>
</tr>
</tbody>
</table>

The nutrient pointer records are created for food groups with at least three foods. The items in any smaller food group can be investigated by looking at the actual attribute values in the record from the nutrient file. The nutrient pointer records vary in size depending on the number of foods in a food group. The records are allocated to the blocks using the same method as used with the description and tree files.
The RECIPE Database

The RECIPE module, like the FOOD module has an underlying database to support functions supplied in the query language. There is a main directory of stored recipes and associated files containing recipe information. Of particular interest is the implementation of all storage as a string of characters, an implementation suggested by the use of FASBOL. Since the RECIPE database is a dynamic set of data, the database design as well as the implementation strategy provides a means of efficiently modifying the recipe data.

Storage of recipe consists of storage of three parts:

1) a set of fixed information, known as the recipe header, which contains various categories that the recipe falls into as well as an analysis of the recipe.

2) the set of ingredients and the amounts of those ingredients that are used in the recipe.

3) the text that describes the method of preparing the recipe.

Each of these recipe parts is stored in a different file. In addition, there is an inverted directory which allows retrieval of specified recipes without searching the recipe information itself.

Main Directory

The main recipe directory is a set of records with a key of name attribute. Associated with each name is a pointer to that recipe's header, ingredients, and text.
This directory is kept in main memory at all times. It is main­
tained as a sequential set of records. The order of the records is
random, changing as recipes are deleted and inserted. Associated
with this sequential file is a string which indicates the records in
the sequential file which are unused. This string contains only 1's
and 0's, with 1 indicating that a particular record is used to store
a current recipe. If a recipe (say the fifth) is deleted, the fifth
character is this string is changed to a 0 and the record associated
with this character can be used again. This method prevents the
sequential file from growing large due to many insertions and deletions.

**Header File**

The header file consists of records which contain fixed infor­
mation about a recipe. The attribute values for many of these cate­
gories are coded so that instead of, for example, storing the string
"italian" for the ethnic attribute, the code 3 can be stored and a
table of attribute values consulted. The table of attribute codes is
supplied in Appendix B. A recipe header requires 303 characters.
These records are stored two per disc block on a random access file.

**Ingredient File**

An ingredient in a recipe is logically seen as one of approxi­
mately 2,400 foods in Handbook 8 (USDA, 1975) and the associated num­
ber of units of that food. The information that is stored to maintain
that logical model consists of substantially more information.
An ingredient record consists of the name given to the food by the user (a name which may differ from the name in the Handbook, USDA, 1975), the record of the nutrient values for that food, and the "multiplier," a positive decimal which is the conversion factor from the 100 grams to the number of units requested by the user. (Handbook #8, USDA, 1975, contains analyses per 100 gram quantities.)
Since any recipe contains several ingredients, the pointer to the ingredients is actually a pointer to the first of the ingredients. The remaining ingredients in a block of disc storage are assumed to be part of the recipe until a "last" ingredient is found. If no last food is found, the next disc block is assumed to contain more ingredients. Since ten ingredients can be represented and stored on one disc block, and the majority of recipes require no more than ten ingredients, the majority of recipe requests require no more than one disc access to obtain the ingredient information.

Preparation Text File

The preparation text record consists of a string of 640 characters. One disc block is required to store this string. When the preparation text is read into main memory, a line structure is presented to the user and it is via this model that the user can modify the preparation text. These records are stored in a random access file.

Inverted Recipe File

The query language of the RECIPE module provides a FIND command. This command allows the NDBS user to find a recipe with certain attribute values. It is not practical, particularly when combinations of attribute values are requested, to investigate each recipe record to find recipes that satisfy a request. Therefore an inverted directory is maintained so that the names of recipes (these names are already in memory in the main directory) can be found without requiring more than one disc access.
This inverted recipe directory is kept on a disc file with (logically) one record per attribute. If a request is made for a certain caloric value, the calorie record is brought to memory and the record scanned for recipes with that value and then names of satisfactory recipes are printed for the user.

This directory is dynamic, with recipes inserted and deleted. This means that there is a lot of directory modification, since a recipe modification means that every attribute entry in the directory must be modified.

Summary

Once the query language of a database system is specified, the processing requests a user can possibly, or even probably, make are known. The system's design can then attempt to create files and directories which will service those requests in the least time. Since the response time is governed primarily by the number of disk accesses required to satisfy a request, the file and directory designs are critical to system's performance. However, the design choices are also constrained by the amount of memory space that can be used to store the directory. The directory itself may need to be stored on disc and then the accesses of directory information also effect the response of the system.

NDBS is typical of database systems in that a several file and directory arrangements are used: The food main directory is always kept in main memory while the inverted recipe file is brought to memory one record at a time; fixed-size nutrient records are...
efficiently allocated to disk files while variable size description records are stored using a simple strategy to solve a complex problem in disc utilization. Since such complex interplay exists between separate decisions, the database designer is hard put to demonstrate a "best design." Rather, a demonstration of reasonable design principles and the production of software that is easily modified as system performance demands are practically attainable goals.
CHAPTER V
RECIPE IMPLEMENTATION AND ANALYSIS

This chapter describes the overall strategy and specific problems involving the implementation of the RECIPE queries which store and analyze information about recipes. Also discussed is the implementation language which was used to program NDBS.

FASBOL

The NDBS system was programmed primarily in FASBOL, a dialect of the SNOBOL programming language. This language was first designed by Griswold (1971) to manipulate character data (often referred to as "string" data). Santos (1971) has written a compiled version of SNOBOL which is FORTRAN callable and allows convenient file handling. FASBOL is written for use on PDP-10 computers. The NDBS runtime system was written using FASBOL. The only routines which are exceptions are some routines which handle random access files, written in MACRO-10, and some separate programs used to configure NDBS, written in PASCAL and FORTRAN.

The choice of programming language was made because of the kind of processing NDBS typically requires. This processing can be described in terms of the kind of data processed and the amounts of input-output and computing processing required. Consider a typical NDBS request: It is a retrieval of stored information. A number of disc accesses might be required to accomplish this task. The data
processing involves manipulation of directory information, deciphering user query, and some numeric computation.

The choice of FASBOL was made because the majority of data items processed consisted of text—user queries or food and attribute names. The FASBOL language is designed for precisely this kind of data. The language supports elaborate string pattern and substitution functions. Numeric data is always represented as a string with conversion as needed. This can be both an advantage and a disadvantage. If a great deal of recomputation of values is necessary, then type conversion is a costly process. But if input and output of already known numeric values is required, there is no type conversion at all. When compared with FORTRAN processing, with the elaborate FORMAT interpretation that is required for every input and output, FASBOL can be very advantageous because of the default assumption that all data is ASCII text (which is the usual case when data is stored externally).

Another advantage of FASBOL is the ease with which the user requests can be processed. Character processing in some languages can be very tedious and the programmer sometimes responds by forcing a rigid syntax on the query language.

The programming needed to allow, for example, the request for "4 POUNDS GROUND BEEF" to be equivalent to "GROUND BEEF 4 POUNDS" is trivial in FASBOL. The discovery and handling of query errors is thus greatly enhanced by pattern matching functions in FASBOL. The user therefore is presented with a friendly system.
Dynamic storage, the ability to request more core from the operating system during execution of a program, is another advantage of FASBOL. Directory entries can therefore grow as needed, without having to allocate maximum space at initialization. Some care must be taken, however, that unused variables be returned to the run time system, for otherwise the size of the program becomes large. The amount of time that the program is swapped out of memory in a time-sharing environment is thus increased since large programs tend to wait longer than small programs in a typical multiprogramming system.

Lest it appear that FASBOL is being claimed as the best of programming languages, it should be noted that the size of the intermediate and object program files generated by the FASBOL compiler can become very large and system quotas for disk storage can be exceeded. This can be a problem for the FASBOL user. The reason for this phenomenon is that at runtime FASBOL requires substantially larger programs. The method of compilation causes the large intermediate files because the compiler produces only an assembly language version of the program. This assembly program then is assembled using the MACRO-10 assembler. Since the intermediate file tends to be large, a system such as NDBS which has many routines cannot be compiled at one time given the normal quota of memory. The program must therefore be broken into different modules which are compiled separately and then loaded together. This requires some amount of programming overhead to manage the global variables necessary to tie the modules together. The resultant object code still tends to be substantially larger than, for example, an equivalent FORTRAN program.
Even with the aforementioned drawbacks, the FASBOL language with its straightforward view of data, allowed NDBS to supply functionality at a low cost in programming time.

**RECIPE Implementation**

The logic flow of the RECIPE module is determined by the view of recipes predicated by the query language. This query language assumes either that there is a current recipe being referenced or that there is a recipe being created. The commands FIND and CREATE fetch current directory entries or start the creation of new directory entries. The other commands REDUCE, CHANGE, ANALYZE, and PRINT all assume that a recipe is already chosen.

The top level of the NDBS system allows the user to choose the particular top level module to be used (FOOD, RECIPE, or MENU) and further decides which function in the particular module will be used (see Figure 3). For example, if the user chooses the CREATE command of the RECIPE module, then the CREATE entry point of the RECIPE subroutine is jumped to from the main, top level of NDBS. The exit from the CREATE routine returns the user to the top level of NDBS to continue RECIPE queries.

**The CREATE Command**

The CREATE command begins a dialog which builds directory entries to point to the various parts of a recipe: Preparation text, ingredients, and attribute categories.
**Text Storage.** The storage of preparation text is accomplished by assuming a particularly simple model for text storage. The text is assumed to be 640 characters long (at most) including carriage return—line feed pairs. These line delimiters allow the user to view the text as though it consisted of several lines although internally the text is merely a string of characters.

**Ingredient Storage.** An ingredient is stored in two parts—the number of units of that ingredient and the name and pointer to the location where the actual ingredient is stored. The pointer is the record number of the nutrient values of that food. The name that is stored with the pointer is the name that is input by the user and not necessarily the name in the main food directory. This is accomplished by allowing "partial matches" to be used to identify a food. For example, the request for the food, "raw, fresh carrots" can be used to identify a particular node of the carrot tree which might not be named "raw, fresh carrots" but the user is not obligated to know this fact.

The units that a user chooses are usually not grams and yet the nutritional data is stored in terms of 100 gram units. Each ingredient is therefore stored with a factor termed a "multiplier," the conversion factor from the units at hand to 100 gram units. For example, the request for "5 ounces RAW CARROTS" is stored with a multiplier of .287. This value is equal to the conversion factor of ounces to grams (1 ounce = 28.7 grams) divided by 100 because the nutrient values are stored per 100 gram units. Thus the ingredient, "RAW CARROTS" has a multiplier of .287, a unit count of 5 and a
pointed to the record with the nutrient values for "RAW CARROTS."

The calculation of any attribute value for CARROTS, which might be
necessary for the analysis of the recipe attributes, can be accomplished
by the retrieval of the 100 gram value, multiplication by the unit
count and then by the "multiplier."

For example, $5 \times 42 \times .287 = 60.27$ is the number of calories
in 5 ounces of raw carrots, where 42 is the number of calories in 100
grams of carrots.

The calculation of the multiplier is accomplished by first matching
the string of characters that describes the units (e.g. "ounces,"
"lb") against predefined unit names. If no such match is found, the
user is not allowed to use those units. If a match if found, then a
vector of multipliers is inspected to retrieve and store the appropriate
multiplier. The legal unit names are stored in a string divided by
asterisks. For example, the unit name string might be

```
*ounces*oz*pounds*lb*
```

The corresponding set of multipliers would be

```
*.287*.287*4.592*4.592*
```

So the string "lb" is recognized as the fourth substring in the name
string. The fourth value in the multiplier string is 4.592 and this
is the multiplier actually stored with the ingredients. The strategy
of storing the multiplier with the ingredient saves recalculation of
conversion factors when a recipe analysis is necessary after recipe
changes have been made.

The name and multiplier string are read when a specific recipe
has been referenced. They can be modified as needed and these
modifications are current when another recipe is referenced or when a CREATE dialog is ended.

**Attribute Storage and Directories.** Each recipe must be categorized on a variety of attributes such as food type or ethnic type. This information is stored in a string which is termed the "header" of the recipe. What is actually stored in the header, however, are numeric codes for the attribute values. For example, the MEAL types are BREAKFAST, LUNCH, DINNER, LUNCH OR DINNER, and ANY MEAL. That is, any recipe is put into one of these categories. These categories are ordered so that numeric code 3 means a DINNER value for the MEAL attribute. The attributes themselves are ordered (MEAL, PREPARATION TYPE, COURSE CODE, ETHNIC TYPE, FLAVOR, COLOR, TEXTURE, SHAPE, and FREQUENCY). The choice of the codes is accomplished by means of a dialog. The user is presented with a request to choose each attribute. The prompt

```
ETHNIC TYPE?
```

might receive a response, GREEK. This response is matched against a string which contains, as substrings, each possible ethnic value, separated by asterisks:

```
*CHINESE*FRENCH*GERMAN*GREEK*ITALIAN*JEWISH*POLISH*
```

The match with the fourth substring then establishes the ethnic code as 4. There is a similar string for each coded attribute. These strings can be extended if other codes are to be added. It should be noted that directories which are stored as strings separated by artificial delimiters can be trivially searched with the FASBOL abilities.
The storage of directory and recipe information is maintained on several files:

**CURRNT.ATT**—contains the strings that hold the attribute values as substrings.

**CURRNT.RCP**—contains the names of recipes and corresponding pointers to text, ingredient, and header file. The first record is a string consisting of 1's and 0's which indicate whether a particular record in the **HEADER.FIL** has been used. This information is necessary since insertions and deletions are allowed and previously used records might later become available. Thus if the string is 11001 and a new recipe is created, the third record will be overwritten instead of appending a new record to the file.

**HEADER.FIL**—is the file of recipe headers, each containing recipe names, recipe analysis, and coded attribute values.

**CURRNT.RCP** and **CURRNT.ATT** are sequential files which contain recipe directories which are stored in memory in arrays during system execution. These arrays are known as **ATTDIR** and **RCPDIR**. The files are rewritten when the system execution is stopped. **HEADER.FIL** is a random access file which is accessed using the **RCPDIR** as a directory.
The FIND Command

The FIND command allows the NDBS user to request information about currently stored recipes. The requests for recipes can be made on the basis of any attribute value: Name, coded categories, or nutritional values. The requests for recipes by name are managed by investigating the storage of the main directory RCPDIR. The names of the recipes are stored along with pointers to the constituents of a recipe: Text, ingredients, and header. When a match on a request is made, this data is retrieved and the found recipe is the current recipe, subject to PRINT, REDUCE, and other NDBS commands.

Searches are not always accomplished by name. Therefore, two directories are maintained, one for searches via nutritional values (NUTDIR) and one for searches on the basis of coded attributes (CODDIR). The user request for nutritional attributes is made by requesting a recipe that has a particular value (e.g. CALORIES = 150) or a range of values (CHOLESTEROL = 120). Note that the requests are on a per serving basis.

Searches for the recipes which satisfy a request could be accomplished by reading the headers for every recipe and storing the names of any which satisfy the request. But, of course, such a sequential search is extremely expensive. A request which cannot be satisfied still requires a disk I/O for every recipe.

Therefore, a directory of recipes which fall into certain ranges of certain attribute values is maintained. It is the NUTDIR array which contains such entries. A sample entry in this directory is

/CHOLESTEROL/0.0;30;10;/18;27;/;;/;31;/

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The semicolons and slashes are artificial delimiters which allow the program to easily find particular substrings. This particular entry means that the directory entries for CHOLESTEROL are stored in ranges of values such that the lowest cholesterol value is 0.0, 30 is the largest value and the size of ranges is 10. Two slashes delimit any pointers that fall into a certain range and the pointers to any recipes in that range are separated by semicolons. Thus recipes 18 and 27 have cholesterol values between 0.0 and 10, there are no recipes in the range from 10 to 20, and recipe 31 is in the range from 20 to 30. (Any recipe which falls on a boundary value such as 10 will be assigned into the higher of the two ranges.)

This directory scheme is an inverted one—each recipe is represented once in the directory for cholesterol. Disk accesses are minimized but not eliminated because if a request is made for a CHOLESTEROL value such as 25, the information in the directory entry only gives pointers to values from 20 to 30 and not pointers to specific values. This is a problem typical of values that are on a (more or less) continuous range—it would be impossible to maintain a directory entry for every possible cholesterol value (101.1, 101.2, 101.3 etc.). Therefore, a request for a recipe with CHOLESTEROL = 25 requires that each recipe in the 20 to 30 range be inspected and compared to 25.

Entries in NUTDIR, the nutrient directory, are made each time CREATE or any of the modifying commands are used. Such modifying commands can require the deletion of one directory entry and insertion of a pointer into another range of values. This processing is necessary for every attribute which is managed in the directory.
The actual attributes which are represented in the inverted directory are read from a sequential file NUT.DIR at system initialization and this file is rewritten when the system is terminated. This file can be modified if the inverted directory is to contain entries for additional attributes. Such an addition requires all current recipes to be inspected and insertions to be made in the appropriate substrings which represent range values. This process is necessary because an inverted directory requires each recipe to be represented once for every attribute that is inverted.

It is precisely this fact which might require directory modification. If each of the 19 nutritional attributes is represented in a directory of 300 recipes, 5,700 pointers are in the directory. But some of the attributes are rarely the basis of requests and the maintenance of a directory for ranges of, for example, POTASSIUM values is an extravagant use of storage. Therefore, only frequently used attributes have an inverted directory maintained for ranges of values. Requests for recipes with values of less frequently requested attributes are satisfied by searching the headers sequentially. Although this is a more time consuming method, the user is not penalized very often because of the infrequency of the requests. The attributes that are inverted in NDBS are COST, CHOLESTEROL, CALORIES, CARBOHYDRATES, FAT, and PROTEIN.

The PRINT Command

The PRINT command can take three forms. If no parameter is supplied, then the current recipe is printed in its entirety. If an
attribute name is supplied as a parameter, then that attribute value is printed for the current recipe. Multiple attributes can be requested. If a recipe name is supplied as a parameter (in quotes to distinguish from attribute names), that recipe becomes the current recipe, and all attributes are printed. Specific attributes may be requested by appending the attribute name (e.g. PRINT "LASAGNA" / CALORIES).

The REDUCE and INCREASE Commands

The commands to REDUCE or INCREASE a particular attribute in the current recipe is a request for the system to provide processing which will search food items and possibly substitute a new item for a current one. This processing could be accomplished "manually" by the user who has access to the FOOD module and therefore information about particular items, as well as the RECIPE commands which modify the recipe. But the RECIPE/INCREASE command is a great convenience to a user since the searching of food values is done automatically.

The REDUCE/INCREASE function does not search with the intelligence that a user might. Substitutions are made only within a given food tree. For example, a carrot item can be substituted for by only another carrot item. The REDUCE/INCREASE logic must, therefore, decide first which food item will potentially be replaced, and hence which trees of food items will be searched, and second which of the foods in the tree will be appropriate for the substitution.

A special directory is managed to allow searches for substitute items without having to execute a disk read for every candidate to be
inspected. A small inverted directory is managed for each food tree of more than two items. Small food trees are as easy (and cheap in terms of response time) to inspect directly, item by item. But for food trees of many items (such as BEEF), an item by item investigation for a substitution that might not even exist is expensive, and a more judicious investigation is in order.

Every entry in the main food directory contains a pointer to the Nutrient Pointer file. This file contains the directories used to accommodate searches for food substitution. Such a directory can be retrieved when it is known that a substitution in a particular food tree is needed. This directory contains the pointers to all the food items in the desired food tree. These pointers are sorted in ascending numeric order on a particular attribute. These pointers are further organized into ranges of values. A search for a food item in a specific range of values can be aided by means of this directory by merely finding record numbers of recipes known to be in that specific range.

Consider an example: The carrot food records sorted according to calorie value are

626, 623, 624, 625, 621, 622, 620, 619, 627

The lowest calorie value is 16 (record 626). The largest value is 325 (record 627). If we wish to organize these records into five categories, a reasonable range for these five categories is 325/5 = 65 calories. There may not be a uniform distribution of carrot items in each category, but the categorizing method can be easily used on all food groups. Since the records are already sorted by calorie, we need to know only which record is the first (smallest) in each category.
to know which range each record falls in.

The following directory arrangement takes advantage of this fact.

Record pointers 626, 623, 624, 625, 621, 622, 620, 619, 627

- lowest value 16
- number of ranges 5
- range size 65
- sequence numbers 3, 6, 0, 9

Figure 5. Inverted Directory Entry for FIND Command

The sequence numbers should be explained. It is clear that the first record lies in the lowest range. The directory needs to provide the information to find the records in the second (third, fourth, etc.) range. The sequence numbers provide that information. The third record in the ordered record numbers is the place in that sequence where the second range begins. By similar application, the sixth record in the sequence (record 622) begins the fourth range. There is no record in the fourth range (and therefore the sequence number is 0), and the first record in the fifth range is record 9. We can deduce that records 620 and 619 both fall into the third range. The actual values for these ranges can be calculated by noting what the lowest value for calories is (16) and what the size of a range is (65 calories). Therefore, records 620 and 619 have a calorie value greater than 16 + 2 * 65 = 146 calories and have values less than 16 + 3 * 65 = 201 calories. The exact value of one of these calorie values can be found by retrieving the record from the file. This
directory scheme allows a compact method of creating directories for all food trees on all possible nutrient values. The end goal, of course, is the minimization of disk accesses. If we wish to find a carrot item with a calorie value of exactly 175, we might still need to access more than one record, but we do not need to search every carrot record.

This directory is what allows INCREASE/REDUCE commands to be implemented. As noted before, the choice must first be which food to substitute and second which substitution to make. The choice of food to substitute for is made on the relative weight of each food ingredient in the recipe. So the ingredient with the largest multiplier is investigated first for possible substitution. If no satisfactory substitution is found, the ingredient with the next largest multiplier is investigated.

The question of whether a satisfactory substitution exists can be answered either by the user or the system. The user is allowed to approve any substitution suggested by the system with a straightforward dialog. The suggested food items must be chosen by the system with some care. Meaningless or relatively useless modifications must be avoided. For example, a request by the user to reduce calorie values that is satisfied with a reduction of two calories is not a large enough reduction to be helpful. The system therefore looks for a "large enough" modification to present to a user. This notion is defined for the system by declaring a uniform constant that is a minimum percentage change which any possible ingredient substitution must cause in the recipe. If the current calorie count of a recipe is 400, then a substitution of a 45-calorie item for a 50-calorie item is (5/400)
1.25 percent change. If the system constant is 5 percent, then such a substitution is not presented to the user for approval.

This processing can be accomplished with a minimum number of disk accesses because much of the information required to choose candidate ingredients is in directories. The current analysis of a recipe is in the header of the file and the ranges of particular ingredient attributes can be found in the nutrient pointer file (and hence no disk access is required to find an unsatisfactory candidate) and the multiplier size is stored with the ingredient records.
CHAPTER VI

DATABASE ADMINISTRATION

Uhlmann (1980) defines the database administrator as the person responsible for the database as a whole with responsibilities that include:

1) modification of implementation,
2) modification of database views,
3) making backup copies and repairing damage to the database.

These functions can be included as a part of the user query language which is restricted to the administrator. NDBS provides these functions as a set of programs which are executed independently (although they could be included as part of the NDBS system proper by including them with the system commands as a subset of executive commands implemented using the Monitor Interpreted Commands (MIC) of the Decsystem-10). This chapter will explain these administrative programs which are used to configure the NDBS system for the user's view.

The original data for any configuration of NDBS is the disk file version of the United States Department of Agriculture Handbook #8, (USDA, 1975). The disk file equivalent is called FOOD.FIL. The name field on this file is not as complete as the printed handbook and so a file of names identical to the printed handbook has been typed into a disk file (groan) called FDATA1.ALL. It is from these two files that subsequent
directories are created for the run time NDBS system. These files and directories are created with a variety of PASCAL, MACRO-10 and FORTRAN programs.

READ.FOR

Input: FOOD.FIL
Output: FOOD.OK

The original tape differs from the handbook not only in name field but in the number of records. The data fields are also packed, a circumstance which makes processing by PASCAL programs more difficult. The program READ.FOR creates a more compatible version of the FOOD.FIL, discarding records which do not occur in the printed handbook and creating an "unpacked" version of the data. This file, FOOD.OK, is then used by other programs.

RANFOR.FOR

Input: FOOD.OK
Output: RANDOM.FOR

The handbook nutrient values must be retrieved at NDBS runtime and there must be a random access file containing these values. RANDOM.FOR is a FORTRAN program with FOOD.OK as its input. The output file is then accessed randomly by NDBS.

PTR.PAS

Input: FOOD.OK, APARAM.FIL
Output: INPUT.RAN, PARRAL.LEL, DIRECT.ORY

A PASCAL program PTR.PAS is used to create the inverted directory of nutrition values. This directory is necessary to efficiently
implement CHANGE commands. The parameters of the directory creation, the nutrients which are inverted and the number of ranges of each inverted attribute are included in the directory. These parameters are specified in APARAM.FIL which can be modified to reconfigure the system. The output records which contain the directory entries are packed as closely to 640 characters as possible. These records are output to the file INPUT.RAN (so called because it must be processed through a FORTRAN program to create a random access file because PASCAL does not support random access files on the DEC-10). A directory of these records is output in PARRAL.LEL. This file is used by the program which creates the main NDBS directory. Another file, DIRECT.NAR, keeps the association between block number and record size and this information is also included in the main directory. This information is necessary because the variable length record of INPUT.RAN are packed into 640 character blocks and a program which inputs blocks of information from this file needs to separate the blocks into records.

RANFOD.FOR

Input: INPUT.RAN
Output: NUTVAL.RAN

This FORTRAN program creates the random access file consisting of inverted directories of nutrient values, one per food group. Each of these records is pointed to by an entry in the main directory.
SETUP.PAS

Input: FDATA1.ALL
Output: FCHARS.REC, FFNODE.REC, STATSS, BRKDWN.DES, BRKDWN.NOD, FDMAIN.DIR

The more complex food names which originally suggested the tree structure implemented in the FOOD module are stored originally in FDATA1.ALL. These names must be separated by food group and hierarchy within food group. This is done by grouping strings of characters (descriptions) and pointers to these descriptions (nodes). Descriptions and nodes are grouped according to size and packed as closely as possible into blocks of 640 characters. SETUP.PAS creates records of similar size descriptions and nodes in two files, FCHARS.REC and FFNODE.REC. Statistics for use in other programs are also stored in two files:

1) BRKDWN.DES—This file contains a table consisting of entries pairing each description size with the number of description records of that size.

2) BRKDWN.NOD—This file contains entries pairing each node size with the number of node records of that size.

The beginning of the main directory is output from SETUP.PAS also. FDMAIN.DIR contains entries which associate the food name used as a main directory key with pointers to the associated description record and length and to the associated node and node record length.
**BLKDES.PAS**

**Input:** FDMAIN.DIR, BRKDWN.DES, SORT01.DES  
**Output:** BLOCKS.DES, NEWDIR.DIR, DESBLK.LEN

This program creates the random access file of descriptions which are used by NDBS at runtime. This file is BLOCKS.DES and it contains description records grouped so that as much of a 640-character block is used. The file DESBLK.LEN contains each block number of BLOCKS.DES and the associated size of records in that block along with the number of records in the block. An updated main directory with pointers to the BLOCK.DES file is output from BLKDES.PAS.

The input to BLKDES.PAS consists of output files from SETUP.PAS. SORT01.DES is the file FCHARS.REC sorted by length using the DEC-10 system sort.

**BLKNOD.PAS**

**Input:** FDMAIN.DIR, SORT01.NOD  
**Output:** NODBLK.LEN, DIRECT.ORY

BLKNOD.PAS is a program similar to BLKDES.PAS. It creates a random access file, BLOCKS.NOD, which contains node records packed into 640-character blocks as efficiently as possible. The block numbers and record lengths associated with each block along with the number of records are stored on NODBLK.LEN. The updated directory is called DIRECT.ORY and contains each food name key with associated node and description pointers.

Input to BLKNOD.PAS consists of files output from SETUP.PAS (although SORT01.NOD is a sorted-by-length version of FFNODE.REC which is output from SETUP.PAS).
COMPAC.FOR

Input: DIRECT.ORY, PARAL.LEL
Output: MAIN.DIR

A compact version of the NDBS main directory with pointers to description, node, and nutrient values is output from the FORTRAN program COMPAC.FOR. The input files are the updated directory from BLKNOD.PAS and the nutrient pointer information from PTR.PAS. The output is the main file MAIN.DIR which has been packed to minimize the amount of memory required to store this directory since the complete directory is kept in memory. Bit packing routines are used to minimize storage.

A new configuration of NDBS can be created using the programs described in this chapter. New releases of Handbook #8 or the desire to have directories inverted on different attributes might require a new configuration to be created.
CHAPTER VII

FUTURE IMPROVEMENTS

The NDBS system is a modest attempt at a nutritional information system. But it is an interesting direction for a system by virtue of the potential user and the quality of the processing provided.

Most previous systems (Balintfy, 1967, and Eckstein, 1966) have institutional applications. This emphasis is natural, since institutions wish to automate repetitive information processing. Although home users do not usually process the quantity of information that is processed in an institution, the convenience of ready information, a convenience that the computer can provide, can encourage more creative and careful analysis of recipe and menu design.

Another emphasis of NDBS is reflected in the attempt to make the system convenient for the user. This desire was the motive to have an interactive system, a simple query language, and as much decision making made by the system as is possible within given constraints. There are interesting future problems and directions which are suggested by the attempts of NDBS in the domain of nutritional systems.

Partial String Matches

Any information system must translate a user input (in the form of a query) into the retrieval of appropriate information or the denial of any available information. Usually an exact request must be made—
request for "CAROTS" will not match "CARROTS" and the food will not be found. Mistypings by the user are very difficult for the system to interpret to recover information. By contrast, some perfectly reasonable request, accurately typed, might not be recognized by the system because the query syntax does not provide a general enough syntax to recognize the request. The net result of too narrow a syntax is possibly an uncomfortable user experience and hence a system which will not be used. One strategy which might be used to allow a more general method to recognize query requests is the use of an evaluation function which matches the requested attribute value against directory entries and retrieves information for the directory entry which has the highest calculated value.

The number of food requests, with the various ways that foods might be requested, provide situations where an evaluating function can be quite useful. A general way to view this partial string matching task is to see the directory as sets of ordered pairs of attribute values and pointers to records in a file.

\[ D = (K_i, p), i = 1, \ldots, v \] where \( v \) is the number of directory entries. Further, assume that each string \( K_i \) is composed of a finite sequence (or vector) of \( n_v \) substrings, separated by delimiters. So \[ K_i = (k_{i1}, k_{i2}, \ldots, k_{in_v}) \]. For example, a directory entry "BOILED BLUE FISH" has three substrings. A user request can also be seen as a string, \( U \), which can be broken into a vector of substrings \[ U = (u_1, u_2, \ldots, u_1) \]. The usual sort of directory matching scheme requires to match some \( K_i \) exactly. In terms of the vectors we require
u_j to match k_{ij} for all j. In effect, we have an evaluation function
which has two arguments, the two vectors of substrings, and outputs a
value of 1 or 0 depending on exact or inexact match.

\[
F((u_1, u_2, \ldots, u_1), (v_1, v_2, \ldots, v_m)) =
\begin{cases}
1 & \text{if } l=m \text{ and } u_i = v_i, i = 1, \ldots, m \\
0 & \text{otherwise}
\end{cases}
\]

That is, if the two vectors have the same number of elements and those
elements are identical and in the same order, then we claim a match,
otherwise no match exists. This scheme can be loosened substantially
to allow partial matches. For example, the component "BURGER" of a
request is very much like the component "HAMBURGER." But with the
above scheme this similarity goes unnoticed. A similarity function
\(s(c_1, c_2)\) is suggested which will have a value indicating similarity.
A reasonable attempt would be based on the fact that one of strings
is found in the other. This match is tempered against the fact that
there are characters that do not match. In the example of "BURGER"
versus "HAMBURGER" we can see this as a percentage match, a sequence
of six characters matching out of a possible nine matching characters.
This strategy requires matching the smaller string into a larger
string. Assume then two strings, \(c_1\) and \(c_2\), such that \(\text{LEN}(c_1) \leq \text{LEN}(c_2)\) where \(\text{LEN}\) is the number of characters in the string. A
similarity of two strings can then be defined.

\[
s(c_1, c_2) =
\begin{cases}
\frac{\text{LEN}(c_1)}{\text{LEN}(c_2)} & \text{if } c_1 \text{ is a substring of } c_2 \\
0 & \text{if } c_1 \text{ is not a substring of } c_2
\end{cases}
\]

The function, \(F\), which describes the matching of two strings, can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
designed in terms of this similarity function applied to the elements of each string when the strings are seen as vectors of substrings.

Note that if the two vectors of substrings are

\[ V = (v_1, v_2, \ldots, v_l) \]

and

\[ U = (u_1, u_2, \ldots, u_m) \]

then a similarity of components might occur in components \( v_i \) and \( u_j \) where \( i \) does not equal \( j \). This requires each component of one vector to be matched against all components of the other. Assuming \( U \) and \( V \) as above and a similarity function, \( s \), we can define

\[
F(U, V) = \sum_{i=1}^{l} \sum_{j=1}^{m} s(v_i, u_j)
\]

This function provides the usual 1 and 0 result that the exact match strategy provides, but it is also sensitive to partial matches and hence inexact descriptions.

A request for a particular food, \( U \), might be matched against each \( K_k \) in a directory. There might be several entries which return a non-zero value and the system can determine which possible match is appropriate to retrieve information for the request. The greatest value is the best match and values above a certain threshold could constitute the better matches each of which might be retrieved for the user. A partial matching ability is what accommodates a great deal of human to human communication and the lack of this mode of communication often sets computer applications apart as "too logical." An information system can provide more comfortable interaction if exact matches of
string data (in the form of queries) are not required. Note that a string processing language such as FASBOL has many functions available in the language to accommodate partial string matches.

Expanded Applications

NDBS retrieves information about recipes and provides some analytical processing which otherwise must be done by the user. This hypothetical user might be another program that takes advantage of NDBS processing. Medical applications are an obvious possibility. NDBS could be a part of a larger system which requests NDBS to design a menu or analyze a recipe within specified constraints.

Diabetics and other patients with diet sensitive maladies could use the NDBS processing as a tool to design menus according to personal tastes. Such a tool could encourage diabetics to attend to the chore of diet design. It is a known phenomenon that diabetics tend to aggravate their condition simply because they cannot find the self discipline to maintain their diet. Diabetics are lax about their diet partly because the search for variety in a diet is hindered by the restrictions that must be maintained. If a program did some of the necessary tedious work, perhaps diabetics would be more inclined to develop the self discipline necessary to eat a satisfactory and healthful diet.
APPENDIX A

USER'S MANUAL

THE NUTRITIONAL DATABASE SYSTEM (NDBS) QUERY LANGUAGE CAN BE ENTERED FROM THREE MAIN LEVELS. WHEN THE USER IS AT THE TOP LEVEL THE SYSTEM Responds WITH

WHICH COMMAND?

THE RESPONSES TO THIS MAY BE ONE OF THE FOLLOWING:

FOOD(F) TAKES THE USER TO THE FOOD AREA
RECIPE(R) TAKES THE USER TO THE RECIPE AREA
MENU(M) TAKES THE USER TO THE MENU AREA
FINISH(FIN) EXITS FROM NDBS

AFTER ENTERING ONE OF THE FOUR AREAS THE SYSTEM Responds WITH:

-F- FOR THE FOOD AREA
-R- FOR THE RECIPE AREA
-M- FOR THE MENU AREA

THIS SIGNIFIES THAT A SECOND LEVEL COMMAND MAY NOW BE ISSUED. TYPING ESCAPE(E) TAKES THE USER BACK TO THE TOP LEVEL OF THE SYSTEM SO THAT A FIRST LEVEL COMMAND CAN AGAIN BE GIVEN.

EX. 1
-----

WHICH COMMAND? R
-R-

EX. 2
-----

WHICH COMMAND? M
-M-

EX. 3
-----

WHICH COMMAND? FIN
EXIT FROM NDBS
THE SECOND LEVEL FOOD COMMANDS

A user enters the food module by typing "food" while at the top level of commands. The system will respond with the prompt -F- which tells the user that any second level food command may now be given. The food commands may be used in any of the other three modules. These are the only commands which can be used in this way.

The nutritional data base system will contain approximately 2500 different foods. This is the foundation of the food module. The food items and their nutritional value are listed in the Department of Agriculture Handbook No. 8. Each food item also has an item number which is listed in the handbook. Each food has many different attributes which are listed in the glossary.

The food items are grouped together to form 658 major food classes. For instance, the major food class carrots consists of nine food items 619-627. The following demonstrates the carrot food class.

CARROT FOOD TREE

```
CARROTS
  ____________
  RAW  COOKED, BOILED, DRAINED   CANNED   DEHYDRATED
     (619)    (620)                   (627)
         /   \
        /     \                       \
       /       \                      \
      /         \                   S&D PACK  S&D PACK
     /           \               (621)    (622)    (623)    (624)    (625)    (626)
```

One major food class may be considered as one food tree. The query language allows the user to select a major food class which allows access to the entire food class tree. Then traversal commands are available which allow the user to travel up and down the tree listing any information desired.
SELECT(S)

PURPOSE: THIS COMMAND BRINGS INTO MEMORY THE RECORD CONTAINING THE SPECIFIED FOOD CLASS. IT ALSO LISTS THE NEXT LEVEL OF FOOD CLASSES. IT POSITIONS THE POINTER TO THE TOP OF THE FOOD CLASS SPECIFIED.

SYNTAX: SELECT F
BRINGS INTO MEMORY THE RECORD WHICH CONTAINS ALL THE INFORMATION FOR THE FOOD CLASS F. THEN THE SYSTEM WILL LIST ALL THE SUBCLASSES OF F.

EX.1

-F-SELECT ASPARAGUS

ASPARAGUS

1. RAW SPEARS
2. COOKED SPEARS, BOILED, DRAINED
3. CANNED SPEARS
4. FROZEN

-F-

EX.2

-F-SELECT PEANUT

THE FOLLOWING FOOD CLASSES ARE POSSIBLE CHOICES FOR SELECTION:
1. PEANUTS
2. PEANUT BUTTERS
3. PEANUT SPREAD
4. PEANUT FLOUR

PLEASE ENTER NUMBER OF FOOD TO BE SELECTED: 1

PEANUTS

1. RAW, WITH SKINS
2. RAW, WITHOUT SKINS
3. BOILED
4. ROASTED, WITH SKINS
5. ROASTED AND SALTED

-F-
DOWN(D)

PURPOSE: TO MOVE ONE LEVEL DOWN THE FOOD TREE. REPOSITIONS THE POINTER.

SYNTAX: DOWN N

REPOSITIONS THE POINTER TO POINT AT THE NODE N. THE SUBCLASSES OF THIS FOOD ARE NOW PRINTED. N IS A POSITIVE NON-ZERO INTEGER.

FOR THE FOLLOWING EXAMPLES, ASSUME THE FOLLOWING COMMAND WAS GIVEN:

-F-SELECT ASPARAGUS
ASPARAGUS
  1. RAW SPEARS
  2. COOKED SPEARS, BOILED, DRAINED
  3. CANNED SPEARS
  4. FROZEN
-F-

EX.1
-----

-F-DOWN 1
ASPARAGUS, RAW SPEARS
-F-

EX.2
-----

-F-DOWN 3
ASPARAGUS, CANNED SPEARS
  1. GREEN
  2. WHITE (BLEACHED)
-F-DOWN 1
ASPARAGUS, CANNED SPEARS, GREEN
  1. REGULAR PACK
  2. SPECIAL DIETARY PACK (LOW-SODIUM)
-F-
UP(UP)

PURPOSE: TO MOVE ONE LEVEL UP THE FOOD TREE. REPOSITIONS THE POINTER.

SYNTAX: UP
REPOSITIONS THE POINTER TO POINT TO ONE LEVEL HIGHER THAN THE CURRENT POSITION.

FOR THE FOLLOWING EXAMPLE ASSUME THE FOLLOWING COMMAND WAS GIVEN:

-F-SELECT ASPARAGUS
ASPARAGUS
  1. RAW SPEARS
  2. COOKED SPEARS, BOILED, DRAINED
  3. CANNED SPEARS
  4. FROZEN
-F-DOWN 3
ASPARAGUS, CANNED SPEARS
  1. GREEN
  2. WHITE (BLEACHED)
-F-DOWN 1
ASPARAGUS, CANNED SPEARS, GREEN
  1. REGULAR PACK
  2. SPECIAL DIETARY PACK (LOW-SODIUM)
-F-

EX.1
-----

-F-UP
ASPARAGUS, CANNED SPEARS
  1. GREEN
  2. WHITE (BLEACHED)
-F-
POINTER(P)

PURPOSE: PRINTS OUT CURRENT POSITION OF POINTER IN FOOD TREE.

EX.1
-----

-F-P
ASPARAGUS, CANNED SPEARS, GREEN
  1. REGULAR PACK
  2. SPECIAL DIETARY PACK (LOW-SODIUM)
-F-
LIST(L)

PURPOSE: TO PRINT ALL LEAF NODES OF THE FOOD TREE BASED ON POSITION OF POINTER IN TREE. WILL LIST ATTRIBUTES SPECIFIED.

SYNTAX: LIST

PRINTS NAME OF ALL LEAF NODES FROM POINTER POSITION.

LIST N

PRINTS NAME OF ALL LEAF NODES FROM POSITION N WHERE N IS THE NUMBER OF A SUBCLASS OF THE PRESENT POINTER POSITION. THE POINTER IS NOT REPOSITIONED.

LIST ALL

PRINTS NAME AND ALL ATTRIBUTES OF ALL LEAF NODES FROM POINTER POSITION.

LIST N ALL

PRINTS NAME AND ALL ATTRIBUTES OF ALL LEAF NODES FROM POSITION N WHERE N IS THE NUMBER OF A SUBCLASS OF THE PRESENT POINTER POSITION. THE POINTER IS NOT REPOSITIONED.

LIST ATTR1,ATTR2,...

PRINTS NAME AND SPECIFIED ATTRIBUTES OF ALL LEAF NODES FROM POINTER POSITION.

LIST N ATTR1,ATTR2,...

PRINTS NAME AND SPECIFIED ATTRIBUTES OF ALL LEAF NODES FROM POSITION N WHERE N IS THE NUMBER OF A SUBCLASS OF THE PRESENT POINTER POSITION. THE POINTER IS NOT REPOSITIONED.

FOR THE FOLLOWING EXAMPLES, ASSUME THE FOLLOWING COMMAND IS GIVEN:

-F-SELECT ASPARAGUS

ASPARAGUS
  1. RAW SPEARS
  2. COOKED SPEARS, BOILED, DRAINED
  3. CANNED SPEARS
  4. FROZEN

-F-DOWN 3

ASPARAGUS, CANNED SPEARS
  1. GREEN
  2. WHITE (BLEACHED)

-F-DOWN 1

ASPARAGUS, CANNED SPEARS, GREEN
  1. REGULAR PACK
  2. SPECIAL DIETARY PACK (LOW-SODIUM)
**EX. 1**

---

- **F-List**
  - ASPARAGUS, CANNED SPEARS, GREEN, REGULAR PACK
  - ASPARAGUS, CANNED SPEARS, GREEN, SPECIAL DIETARY PACK

---

**EX. 2**

---

- **F-List 1 All**

<table>
<thead>
<tr>
<th>ASPARAGUS, CANNED, GREEN, REGULAR PACK, SOLIDS AND LIQUIDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>WATER(%)</td>
</tr>
<tr>
<td>FOOD ENERGY(CAL)</td>
</tr>
<tr>
<td>PROTEIN(GM)</td>
</tr>
<tr>
<td>FAT(GM)</td>
</tr>
</tbody>
</table>

**COST** | .59 |

<table>
<thead>
<tr>
<th>ASPARAGUS, CANNED, GREEN, REGULAR PACK, DRAINED SOLIDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>WATER(%)</td>
</tr>
<tr>
<td>FOOD ENERGY(CAL)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASPARAGUS, CANNED, GREEN, REGULAR PACK, DRAINED LIQUIDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>
HELP(H)

PURPOSE: PRINTS OUT ALL COMMANDS WHICH ARE AVAILABLE TO USER AT FOOD LEVEL

ESCAPE
HELP
LIST
POINTER
SELECT
DOWN
UP

EX.1
----

-F-HELP

THE FOLLOWING COMMANDS ARE AVAILABLE IN THE FOOD FILE:

ESCAPE(ESC) EXITS FROM FOOD MODULE AND RETURNS TO TOP LEVEL COMMANDS
HELP(H) PRINTS HELP FILE OF COMMANDS AT THE FOOD LEVEL
LIST(L) LISTS ALL FOOD ITEMS OF SELECTED FOOD CLASS
POINTER(P) LISTS FOOD CLASS CURRENTLY IN MEMORY
SELECT(SEL) SELECTS A FOOD OR FOOD CLASS
DOWN(D) LISTS SUB-CLASSES OF FOOD CURRENTLY IN MEMORY
UP(UP) MOVES UP ONE LEVEL OF FOOD CLASSES
-F-
ESCAPE (ESC)

PURPOSE: PUTS USER AT TOP LEVEL COMMANDS.

EX. 1

-F-ESC

WHICH COMMAND?
SECOND LEVEL RECIPE COMMANDS

THE RECIPE MODULE PROVIDES THE FACILITIES TO CREATE, STORE, MODIFY, AND ANALYZE GROUPS OF ITEMS IN THE FOOD FILE AS RECIPES. TWO DYNAMIC COMMANDS, INCREASE AND REDUCE, ALLOW THE USER TO OPTIMIZE ATTRIBUTES OF A RECIPE SUCH AS COST WITHOUT THE USER HAVING TO SEARCH THE FOOD FILE ITEM BY ITEM.
CREATE
---

PURPOSE: CREATE AND STORE A RECIPE

SYNTAX: CREATE "RECIPE NAME"

A DIALOG NOW OCCURS IN WHICH THE USER ANSWERS THE FOLLOWING QUESTIONS.

<table>
<thead>
<tr>
<th>QUESTION #</th>
<th>QUESTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RECIPE NAME?</td>
</tr>
<tr>
<td>2</td>
<td>MINUTES OF PREPARATION?</td>
</tr>
<tr>
<td>3</td>
<td>COOKING TEMPERATURE? (IF IT APPLIES)</td>
</tr>
<tr>
<td>4</td>
<td>NUMBER OF SERVINGS?</td>
</tr>
<tr>
<td>5</td>
<td>MEAL TYPE?</td>
</tr>
<tr>
<td>6</td>
<td>PREPARATION TYPE?</td>
</tr>
<tr>
<td>7</td>
<td>COURSE CODE?</td>
</tr>
<tr>
<td>8</td>
<td>ETHNIC TYPE?</td>
</tr>
<tr>
<td>9</td>
<td>FLAVOR?</td>
</tr>
<tr>
<td>10</td>
<td>COLOR?</td>
</tr>
<tr>
<td>11</td>
<td>TEXTURE?</td>
</tr>
<tr>
<td>12</td>
<td>SHAPE?</td>
</tr>
<tr>
<td>13</td>
<td>FREQUENCY?</td>
</tr>
<tr>
<td>14</td>
<td>MENU TYPE?</td>
</tr>
</tbody>
</table>

THE FOLLOWING TABLE GIVES ACCEPTABLE AND DEFAULT RESPONSES TO QUESTIONS. A CARRIAGE RETURN SIGNIFIES A DEFAULT RESPONSE. N.A. MEANS NOT ACCEPTABLE.

<table>
<thead>
<tr>
<th>QUESTION #</th>
<th>DEFAULT</th>
<th>POSSIBLE RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N.A.</td>
<td>ANY NAME WITH FEWER THAN 35 CHARACTERS</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>ANY POSITIVE INTEGER</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>ANY POSITIVE INTEGER</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>ANY POSITIVE INTEGER</td>
</tr>
<tr>
<td>5</td>
<td>N.A.</td>
<td>ANY OF DEFINED MEAL TYPES</td>
</tr>
<tr>
<td>6</td>
<td>N.A.</td>
<td>ANY OF DEFINED PREPARATION TYPES</td>
</tr>
<tr>
<td>7</td>
<td>N.A.</td>
<td>ANY DEFINED COURSE CODE</td>
</tr>
<tr>
<td>8</td>
<td>NO TYPE</td>
<td>IT (ITALIAN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FR (FRENCH)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GE (GERMAN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CH (CHINESE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PO (POLISH)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GR (GREEK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JE (JEWISH)</td>
</tr>
<tr>
<td>9</td>
<td>N.A.</td>
<td>ANY DEFINED FLAVOR</td>
</tr>
<tr>
<td>10</td>
<td>N.A.</td>
<td>ANY DEFINED COLOR</td>
</tr>
<tr>
<td>11</td>
<td>N.A.</td>
<td>ANY DEFINED TEXTURE</td>
</tr>
</tbody>
</table>
A ? AFTER ANY PROMPT WILL PRINT THE OPTIONS THAT ARE AVAILABLE FOR THE USER.

THE USER IS NOW ASKED TO:

ENTER RECIPE ITEMS - END WITH "DONE"

<table>
<thead>
<tr>
<th>QUANTITY</th>
<th>UNITS</th>
<th>NAME OF FOOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;DONE&quot;</td>
</tr>
</tbody>
</table>

AFTER THE CONTROL-Z INSERT, DELETE, REPLACE, REDUCE, AND ANALYZE COMMANDS ARE AVAILABLE FOR THE PRESENT RECIPE. FOOD FILE SEARCH COMMANDS ARE ALSO AVAILABLE AT THIS TIME.

EXAMPLE OF CREATE COMMAND

```
-R- CREATE
WHAT IS THE RECIPE NAME? "LASAGNA SAUCE"
MINUTES OF PREPARATION? 45
COOKING TEMPERATURE? 375
NUMBER OF SERVINGS? 5
MEAL TYPE? SAUCE
PREPARATION TYPE? BAKE
COURSE CODE? ENTREE
ETHNIC TYPE? ITALIAN
FLAVOR? STRONG
COLOR? ORANGE
TEXTURE? SOFT
SHAPE? AMORPHOUS
FREQUENCY? EVERY OTHER WEEK
```
2.5 POUNDS HAMBURGER
4 10 OZ CANS OF TOMATO SAUCE
2 5 OZ CANS OF MUSHROOM SLICES
4 TBSP OLIVE OIL
2 TSP OREGANO
DONE

BROWN HAMBURGER, SAUTEE MUSHROOMS IN OLIVE OIL, ADD OREGANO AND TOMATO SAUCE AND MUSHROOMS TO GROUND BEEF. SERVE OVER LASAGNA NOODLES.

OK TO ENTER LASAGNA SAUCE? OK

LASAGNA SAUCE RECIPE IS STORED
REDUCE(INCREASE)

PURPOSE: TO MODIFY RECIPE INGREDIENTS TO SATISFY ATTRIBUTE REQUIREMENTS. THIS COMMAND CAN BE USED AFTER ANY RECIPE IS SPECIFIED VIA A FIND, CREATE, OR CHANGE COMMAND.

SYNTAX: 1) REDUCE(INCREASE) ATTRIBUTE

TO REDUCE OR INCREASE THE ATTRIBUTE VALUE IN THE CURRENT RECIPE.

2) REDUCE(INCREASE) ATTRIBUTE TO SPECIFICATION

WILL REDUCE OR INCREASE THE ATTRIBUTE VALUE TO THE SPECIFIED VALUE IN THE CURRENT RECIPE.

AN ATTRIBUTE IS ANY OF THE DEFINED CHARACTERISTICS OF A FOOD (SEE GLOSSARY) SUCH AS COST OR CALORIES.

A SPECIFICATION IS OF THE FORM

OPERATOR QUANTITY

WHERE AN OPERATOR IS <, >, OR =
AND A QUANTITY IS ANY NON-NEGATIVE DECIMAL.
IF NO OPERATOR IS SPECIFIED = IS ASSUMED.

THE RESULT IS A NEW RECIPE WITH THE ATTRIBUTE SPECIFIED HAVING A REDUCED(INCREASED) VALUE.

EXAMPLE OF REDUCE COMMAND

-------------

-R- FIND "LASAGNA SAUCE"
LASAGNA SAUCE FOUND
-R- REDUCE SALT TO 0.0
NEW RECIPE: CANNED FISH REMOVED, FRESH FISH INSERTED.
NEW ANALYSIS: SALT = 0.0
CALORIES = 450 / SERVING (INCREASE)
COST = $2.85 / SERVING (INCREASE)
FIND
-----

PURPOSE: FIND ALL OCCURRENCES OF A RECIPE WITH GIVEN ATTRIBUTES

SYNTAX: 1) FIND "RECIPE NAME"

TO FIND A RECIPE WITH THE GIVEN NAME. AFTER THE RECIPE IS FOUND REDUCE, INCREASE, ANALYZE, AND CHANGE COMMANDS ARE AVAILABLE.

2) FIND TERM WITH SPEC

A "TERM" CAN BE
A) PREPARATION TYPE
B) RECIPE TYPE
C) MENU TYPE
D) AN INGREDIENT NAME

A LIST OF VALID PREPARATION, RECIPE, OR MENU TYPES IS GIVEN IN GLOSSARY.

NOT CAN PRECEDE A TERM

A "SPEC" (IFICATION) IS OF THE FORM

ATTRIBUTE OPERATOR QUANTITY

ATTRIBUTE - IS ONE OF THE DEFINED FOOD CHARACTERISTICS IN GLOSSARY

OPERATOR - IS <, >, OR =

QUANTITY - IS A NON-NEGATIVE DECIMAL

EXAMPLES OF FIND COMMAND USAGE
----------------------------------

1) FIND "LASAGNA"
WILL RETRIEVE A RECIPE WITH THE NAME LASAGNA.
2) FIND ZUCCHINNI OR CASSEROLES
WILL RETRIEVE ALL RECIPES WITH ZUCCHINNI OR CASSEROLES.
4) FIND SAUCE WITH CALORIES < 150
WILL RETRIEVE ALL SAUCE RECIPES THAT HAVE FEWER THAN 150 CALORIES PER SERVING.
PRINT

PURPOSE: TO LIST THE INGREDIENTS IN A RECIPE

SYNTAX: PRINT "RECIPE NAME"

OPTIONS WHICH MAY BE APPENDED WITH A / TO THE PRINT COMMAND ARE:
/FULL TO SEE EXACT ITEM DESCRIPTIONS AND TABLE 8
ITEM NUMBERS
/ANAL TO SEE AN ANALYSIS ON ALL ATTRIBUTES OF THE RECIPE
/PREP TO SEE THE PREPARATION TEXT OF THE RECIPE
/FOODS TO SEE EACH FOOD IN THE RECIPE ANALYZED

EXAMPLE OF PRINT COMMAND
-----------------------------------
-R- PRINT "LASAGNA SAUCE"/FULL/ANAL

INGREDIENTS:

<table>
<thead>
<tr>
<th>QUANTITY</th>
<th>UNITS</th>
<th>NAME OF FOOD</th>
<th>ITEM#</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>POUNDS</td>
<td>HAMBURGER</td>
<td>(ITEM 348)</td>
</tr>
<tr>
<td>4</td>
<td>10 OZ CANS</td>
<td>TOMATO SAUCE</td>
<td>(ITEM 576)</td>
</tr>
<tr>
<td>2</td>
<td>5 OZ CANS</td>
<td>MUSHROOM SLICES</td>
<td>(ITEM 405)</td>
</tr>
<tr>
<td>4</td>
<td>TBSP</td>
<td>OLIVE OIL</td>
<td>(ITEM 456)</td>
</tr>
<tr>
<td>2</td>
<td>TSP</td>
<td>OREGANO</td>
<td>(ITEM 472)</td>
</tr>
</tbody>
</table>

ANALYSIS PER SERVING:

CARBOHYDRATES 450 GRAMS
FATS 320 GRAMS
SALT 12 MG
CHOLESTEROL 125 MG

COST $2.50 / SERVING
CHANGE

PURPOSE: TO MODIFY AN EXISTING RECIPE

SYNTAX: CHANGE "RECIPE NAME"

THE FOLLOWING COMMANDS ARE NOW IN ORDER:

INSERT

PURPOSE: PUT A NEW ITEM INTO A RECIPE

SYNTAX: INSERT QUANTITY UNITS FOOD

EX. INSERT 10 LB ALMONDS

REPLACE

PURPOSE: TO SUBSTITUTE A FOOD IN A RECIPE WITH ANOTHER FOOD.

SYNTAX: REPLACE "FOOD NAME" WITH QUANTITY UNITS FOOD

DELETE

PURPOSE: REMOVE AN ITEM FROM A RECIPE

SYNTAX: DELETE FOOD NAME

PREPARATION

PURPOSE: TO CHANGE THE PREPARATION TEXT OF A RECIPE

SYNTAX: PREPARATION

THE PREPARATION TEXT CAN NOW BE CHANGED UNTIL CONTROL-Z

DONE

PURPOSE: TO SIGNAL THAT NO MORE CHANGES ARE DESIRED

SYNTAX: DONE

THE USER THEN HAS AN OPTION TO INCLUDE THE CHANGES OR NOT.
EXAMPLE OF CHANGE COMMAND

--------------------------

-R- CHANGE "LASAGNA SAUCE"
-R- HELP

THE FOLLOWING COMMANDS ARE AVAILABLE:

INSERT
REPLACE
DELETE
PREPARATION
DONE

TYPE HELP FOLLOWED BY THE COMMAND FOR DIRECTIONS
FOR ANY PARTICULAR COMMAND

-R- INSERT 3 TSP THYME
-R- DELETE OREGANO
-R- REPLACE TOMATO SAUCE WITH 1 10 OZ CAN TOMATO PASTE
-R- DONE

YOUR RECIPE: LASAGNA SAUCE

<table>
<thead>
<tr>
<th>QUANTITY</th>
<th>UNITS</th>
<th>NAME OF FOOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>POUNDS</td>
<td>HAMBURGER</td>
</tr>
<tr>
<td>1</td>
<td>10 OZ CAN</td>
<td>TOMATO PASTE</td>
</tr>
<tr>
<td>3</td>
<td>TSP</td>
<td>THYME</td>
</tr>
<tr>
<td>2</td>
<td>5 OZ CAN</td>
<td>MUSHROOM SLICES</td>
</tr>
<tr>
<td>4</td>
<td>TBSP</td>
<td>OLIVE OIL</td>
</tr>
</tbody>
</table>

DO YOU WISH TO ENTER THE CURRENT CHANGES? Y
-R- CHANGES ENTERED
-R-
ANALYZE

PURPOSE: TO HAVE REQUESTED ATTRIBUTE VALUES PRINTED ABOUT A GIVEN RECIPE.

SYNTAX: 
1) ANALYZE "RECIPE NAME"
2) ANALYZE
AFTER A SPECIFIC RECIPE HAS BEEN IDENTIFIED VIA A FIND, CHANGE, OR CREATE COMMAND

SUFFIXES MAY OPTIONALLY APPENDED TO ANALYZE SPECIFIC ATTRIBUTES AND PRINT THEIR VALUES BY TYPING FOR AND A LIST OF ATTRIBUTES.

EXAMPLE OF ANALYZE COMMAND

-R- ANALYZE "LASAGNA SAUCE" FOR SALT

ANALYSIS:
850 CALORIES / SERVING
$2.50 / SERVING
2 MG SALT / SERVING

-R- CHANGE "LASAGNA SAUCE"

-R- INSERT 2 TSP CHIVES

-R- ANALYZE
ANALYSIS:
850 CALORIES / SERVING
$2.51 / SERVING
THE SECOND LEVEL MENU COMMANDS

THE MENU MODULE CONSISTS OF GROUPS OF RECIPES CALLED "MENUS." EACH MENU, BASED ON USER SPECIFICATIONS, IS CREATED DYNAMICALLY BY THE SYSTEM. A MENU MAY BE Created, CHANGED, ANALYZED, SAVED, DELETED, OR PRINTED.

THE FOLLOWING PAGES DESCRIBE THE MENU MODULE COMMANDS AND THEIR USES.
CREATE(CR)

PURPOSE: TO CREATE A MENU BASED ON USER SPECIFICATION.

SYNTAX: CREATE

THIS IS AN INTERACTIVE COMMAND. THE SYSTEM RESPONDS WITH A SERIES OF QUESTIONS TO WHICH THE USER CAN SPECIFY MANY OPTIONS. THE FOLLOWING LISTS THE QUESTIONS ALONG WITH ALL POSSIBLE ANSWERS AND DEFAULT CASES IF APPLICABLE.

<table>
<thead>
<tr>
<th>QUES. #</th>
<th>QUESTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MENU NAME:</td>
</tr>
<tr>
<td>2</td>
<td>NO. OF DAYS IN MENU:</td>
</tr>
<tr>
<td>3</td>
<td>TYPE OF MEANS(BR,LU,DI):</td>
</tr>
<tr>
<td>4</td>
<td>APPROX. COST OF MENU:</td>
</tr>
<tr>
<td>5</td>
<td>NO. OF PEOPLE MENU SERVES:</td>
</tr>
<tr>
<td>6</td>
<td>SPECIFY ETHNIC FOOD, CR FOR NONE:</td>
</tr>
<tr>
<td>7</td>
<td>SPECIAL RECIPES TO CONSIDER:</td>
</tr>
<tr>
<td>8</td>
<td>THE MENU SELECTED IS:</td>
</tr>
<tr>
<td>9</td>
<td>ACCEPTABLE(YES OR NO):</td>
</tr>
<tr>
<td>10</td>
<td>DAY TO BE CHANGED:</td>
</tr>
<tr>
<td>11</td>
<td>MEAL TO BE CHANGED:</td>
</tr>
<tr>
<td>12</td>
<td>CHANGE:</td>
</tr>
<tr>
<td>13</td>
<td>MENU IS CREATED AS ABOVE. IF YOU WANT MENU SAVED TYPE &quot;SAVE&quot;.</td>
</tr>
</tbody>
</table>

THE FOLLOWING TABLE GIVES ACCEPTABLE AND DEFAULT RESPONSES TO QUESTIONS. A CARRIAGE RETURN SIGNifies A DEFAULT RESPONSE. "N.A." MEANS NOT ACCEPTABLE.

<table>
<thead>
<tr>
<th>QUES. #</th>
<th>DEFAULT</th>
<th>POSSIBLE RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N.A.</td>
<td>UP TO 20 CHARACTERS</td>
</tr>
<tr>
<td>2</td>
<td>N.A.</td>
<td>ANY POSITIVE NON-ZERO INTEGER 29</td>
</tr>
<tr>
<td>3</td>
<td>ALL MEALS</td>
<td>ANY COMBINATION OF THE THREE MEALS:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BR(BREAKFAST)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LU(LUNCH)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DI(DINNER)</td>
</tr>
<tr>
<td>4</td>
<td>N.A.</td>
<td>ANY DOLLAR AND CENT AMOUNT 1000</td>
</tr>
<tr>
<td>5</td>
<td>N.A.</td>
<td>N IS A POSITIVE NON-ZERO INTEGER.</td>
</tr>
<tr>
<td>6</td>
<td>NONE</td>
<td>H WILL TYPE THIS LIST OF LEGAL ETHNIC GROUPS:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IT(ITALIAN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FR(FRENCH)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GE(GERMAN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CH(CHINESE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PO(POSLIKE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GR(GREEK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JE(JEWISH)</td>
</tr>
</tbody>
</table>
INPUT SPECIAL RECIPES TO CONSIDER ENDING EACH RECIPE WITH A CR. END THE STEP BY INPUTTING ANOTHER CR. UP TO 10 RECIPES MAY BE INPUT.

THE SYSTEM WILL PRINT OUT THE MENU SELECTED.

YES OR NO

ANY POSITIVE NON-ZERO INTEGER.

ONE OF THE FOLLOWING:
  BR(EAKFAST)
  LU(NCHEON)
  DI(NNER)

ONE OF THE FOLLOWING OR A COMBINATION OF THE FOLLOWING:
  INSERT RECIPE1
  DELETE RECIPE1
  REPLACE RECIPE1


EX. 1

-M-CR

MENU NAME: WEEK1
NO. OF DAYS IN MENU: 1
TYPE OF MEALS(BR, LU, DI): DI
APPROX. COST OF MENU: 10.00
NO. OF PEOPLE MENU SERVES: 4
SPECIFY ETHNIC FOOD, CR FOR NONE: IT
SPECIAL RECIPES TO CONSIDER:
THE MENU SELECTED IS:
WEEK 1
DAY 1

DINNER
3 IN. SQ. LASAGNA
1/2 C. PARMESAN BRUSSEL SPROUTS
1 C. TOSSED SALAD/ITALIAN DRESSING
1 SLICE GARLIC TOAST
1/2 C. SPUMONI ICE CREAM
COFFEE

COST OF MEAL IS 9.80 FOR 4 SERVINGS.

ACCEPTABLE (YES OR NO): YES

MENU IS CREATED AS ABOVE. IF YOU WANT MENU SAVED TYPE "SAVE"
CHANGE(CH)

PURPOSE: TO CHANGE PART OF ANY MENU. THAT IS, TO INSERT, DELETE, OR REPLACE A MENU ITEM.

SYNTAX: CHANGE

THIS IS AN INTERACTIVE COMMAND WHICH WILL RESPOND WITH THE FOLLOWING QUERIES:

<table>
<thead>
<tr>
<th>QUESTION#</th>
<th>QUESTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MENU NAME:</td>
</tr>
<tr>
<td>2</td>
<td>DAY TO BE CHANGED:</td>
</tr>
<tr>
<td>3</td>
<td>MEAL TO BE CHANGED:</td>
</tr>
<tr>
<td>4</td>
<td>CHANGE:</td>
</tr>
</tbody>
</table>

THE FOLLOWING TABLE GIVES ACCEPTABLE AND DEFAULT RESPONSES TO QUESTIONS. A CARRIAGE RETURN SIGNIFIES A DEFAULT RESPONSE. "N.A." MEANS NOT ACCEPTABLE.

<table>
<thead>
<tr>
<th>#</th>
<th>DEFAULT</th>
<th>POSSIBLE RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CURRENT MENU</td>
<td>ANY NAME OF A STORED MENU</td>
</tr>
<tr>
<td>2</td>
<td>EXIT FROM COMMAND</td>
<td>POSITIVE NON-ZERO INTEGER</td>
</tr>
<tr>
<td>3</td>
<td>N.A.</td>
<td>ONE OF THE FOLLOWING:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BR(EAKFAST)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LU(NCHEON)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DI(NNER)</td>
</tr>
<tr>
<td>4</td>
<td>N.A.</td>
<td>ONE OR A COMBINATION OF THE FOLLOWING:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INSERT RECIPE1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DELETE RECIPE1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REPLACE RECIPE1</td>
</tr>
</tbody>
</table>

EX. 1
-----

-M-CH
MENU NAME: MENU1
DAY TO BE CHANGED: 2
MEAL TO BE CHANGED: BR
CHANGE: DELETE ORANGE
CHANGE: INSERT FRUIT CUP
CHANGE:
CHANGE COMPLETED.
DAY TO BE CHANGED:
-M-

EX. 2
-----

-M-CH
MENU NAME: MENU2
DAY TO BE CHANGED: 6
MEAL TO BE CHANGED: DI
CHANGE: DELETE CORN
CHANGE: INSERT FRENCH FRIES
CHANGE:
MEAL CONTAINS MASHED POTATOES, DO YOU WANT CHANGE
2 COMPLETED (YES OR NO): NO
CHANGE: INSERT PEAS
CHANGE:
CHANGE COMPLETED.
DAY TO BE CHANGED:
-M-
PRINT(PR)

PURPOSE: ALLOWS THE USER TO PRINT OUT THE DIRECTORY OF MENU NAMES. ALSO ALLOWS THE USER TO PRINT OUT ALL OR PORTIONS OF A MENU.

SYNTAX: PRINT

PRINT CURRENT MENU CREATED.

PRINT NAME
PRINT THE MENU WHICH IS SAVED UNDER NAME
THE SYSTEM RESPONDS WITH:

SPECIFY OPTIONS:

TO WHICH THE USER MAY INPUT ONE OF THE FOLLOWING:

DAY X
BR
LU
DI

PRINT DIR
PRINT DIRECTORY OF MENU NAMES.

EX.1

---

-M-PRINT MENU1
SPECIFY OPTIONS: DAY 1

MENU1

DAY 1
BREAKFAST
1/2 C. ORANGE JUICE
2 SCRAMBLED EGGS
2 SLICES BACON
1 SLICE TOAST/BUTTER/JAM
COFFEE
MILK

-M-
<table>
<thead>
<tr>
<th>Menu</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>MENU1</td>
<td>1-15-80</td>
</tr>
<tr>
<td>MENU2</td>
<td>3-13-80</td>
</tr>
<tr>
<td>MENUN</td>
<td>7-28-80</td>
</tr>
</tbody>
</table>
ANALYZE(AN)

PURPOSE: ALLOWS ANALYZING OF A MENU FOR VARIOUS ATTRIBUTES.
THE ATTRIBUTES ON WHICH THE MENU CAN BE ANALYZED ARE
THE FOLLOWING:

<table>
<thead>
<tr>
<th>CODE NAME</th>
<th>ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAT</td>
<td>WATER</td>
</tr>
<tr>
<td>CAL</td>
<td>CALORIES</td>
</tr>
<tr>
<td>PRO</td>
<td>PROTEIN</td>
</tr>
<tr>
<td>FAT</td>
<td>FAT</td>
</tr>
<tr>
<td>TOT</td>
<td>TOTAL CARBOHYDRATE</td>
</tr>
<tr>
<td>FIB</td>
<td>FIBER CARBOHYDRATE</td>
</tr>
<tr>
<td>ASH</td>
<td>ASH</td>
</tr>
<tr>
<td>CAL</td>
<td>CALCIUM</td>
</tr>
<tr>
<td>PHO</td>
<td>PHOSPHOROUS</td>
</tr>
<tr>
<td>IRO</td>
<td>IRON</td>
</tr>
<tr>
<td>SOD</td>
<td>SODIUM</td>
</tr>
<tr>
<td>POT</td>
<td>POTASSIUM</td>
</tr>
<tr>
<td>VIT</td>
<td>VITAMIN A</td>
</tr>
<tr>
<td>THI</td>
<td>THIAMINE</td>
</tr>
<tr>
<td>RIB</td>
<td>RIBOFLAVIN</td>
</tr>
<tr>
<td>NIA</td>
<td>Niacin</td>
</tr>
<tr>
<td>ASC</td>
<td>ASCORBIC</td>
</tr>
<tr>
<td>LIN</td>
<td>LINEOLIC ACID</td>
</tr>
<tr>
<td>CHO</td>
<td>CHOLESTEROL</td>
</tr>
</tbody>
</table>

SYNTAX: ANALYZE
ANALYZE CURRENT MENU IN MEMORY FOR ALL ATTRIBUTES.
WILL GIVE ANALYSIS ON A MEAL BASIS.

ANALYZE NAME
ANALYZE THE ENTIRE MENU NAME FOR THE ATTRIBUTES SPECIFIED.

THE SYSTEM RESPONDS WITH:

SPECIFY OPTIONS:

TO WHICH THE USER MAY INPUT ONE OF THE FOLLOWING:

DAY N
BR
LU
DI

WHERE N IS THE NUMBER FROM 1 TO THE MAXIMUM NUMBER OF DAYS IN
THE MENU.
THE SYSTEM THEN Responds WITH

SPECIFY ATTRIBUTES:

TO WHICH THE USER MAY INPUT ONE OF THE ABOVE ATTRIBUTE CODES. IF ONLY A CARRIAGE RETURN IS INPUT AT THIS POINT ALL ATTRIBUTES WILL BE ANALYZED.

EX.1

- -

SPECIFY OPTIONS: BR

SPECIFY ATTRIBUTES:

CURRENT MENU ANALYSIS
(Per serving)

DAY 1

BREAKFAST

<table>
<thead>
<tr>
<th>WATER(%)</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOOD ENERGY(CAL)</td>
<td>700</td>
</tr>
<tr>
<td>PROTEIN(GM)</td>
<td>50</td>
</tr>
</tbody>
</table>

DAY 2

BREAKFAST

<table>
<thead>
<tr>
<th>WATER(%)</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOOD ENERGY(CAL)</td>
<td>667</td>
</tr>
<tr>
<td>PROTEIN(GM)</td>
<td>48</td>
</tr>
</tbody>
</table>

DAY 3

BREAKFAST

<table>
<thead>
<tr>
<th>WATER(%)</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- -
EX. 2

- M-AN MENU1
SPECIFY OPTIONS: DAY1
SPECIFY ATTRIBUTES: CAL

MENU1 ANALYSIS
(PER SERVING)

DAY 1
BREAKFAST

FOOD ENERGY (CAL) 500
DELETE(DE)

PURPOSE: ALLOWS DELETION OF MENUS FROM THE SYSTEM

SYNTAX: DELETE NAME
DELETE THE MENU NAME FROM THE SYSTEM.

EX.1

--M-DE MENU1

MENU1

DAY 1

DINNER
3 IN. SQ. LASAGNA
1/2 C. PARMESAN BRUSSEL SPROUTS
1 C. TOSSED SALAD/ITALIAN DRESSING
1 SLICE GARLIC TOAST
1/2 C. SPUMONI ICE CREAM
COFFEE

MENU DELETED (YES OR NO): YES
MENU MENU1 DELETED.

--M--
SAVE

PURPOSE: ALLOWS THE USER TO SAVE A MENU ON FILE SO THAT THE SYSTEM MAY RECALL IT AT A LATER DATE. A MENU CANNOT BE SAVED UNDER THE NAME "DIR".

SYNTAX: SAVE NAME
SAVE THE MENU WHICH HAS CURRENTLY BEEN CREATED. "NAME" IS THE NAME GIVEN TO THE MENU BY WHICH IT CAN BE REFERENCED LATER. "NAME" MAY BE OF LENGTH 20 CHARACTERS OR LESS. IF THE COMMAND HAS BEEN COMPLETED THE SYSTEM WILL ISSUE THE FOLLOWING MESSAGE:

MENU SAVED UNDER NAME NAME

EX.1

--M--SAVE MENU1
MENU SAVED UNDER NAME MENU1
--M--
HELP (H)

PURPOSE: PRINTS OUT THE COMMANDS AVAILABLE TO THE USER AT THE SECOND LEVEL MENU COMMANDS.

- ANALYZE
- CHANGE
- CREATE
- DELETE
- HELP
- ESCAPE
- PRINT
- SAVE

THE FOLLOWING ARE USER COMMANDS IN THE MENU MODULE:

ANALYZE (AN) ANALYZES MENU ON DAILY BASIS ON ANY OR ALL ATTRIBUTES
CHANGE (CH) CHANGES A MENU ITEM
CREATE (CR) CREATES A MENU
DELETE (DE) DELETES A STORED MENU
SAVE (S) STORES A CREATED MENU
PRINT (PR) PRINTS ALL OR PARTS OF MENU OR MENU DIRECTORY
ESCAPE (ESC) RETURNS TO TOP LEVEL COMMANDS
ESCAPE(E)

PURPOSE: ALLOWS USER TO GET TO THE TOP LEVEL COMMAND

-M-ESC

WHICH COMMAND?
# APPENDIX B

## MENU PLANNING CODES

<table>
<thead>
<tr>
<th>Meal Type</th>
<th>Ethnic Type</th>
<th>Course Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakfast</td>
<td>Chinese</td>
<td>Appetizer</td>
</tr>
<tr>
<td>Lunch</td>
<td>French</td>
<td>Juice</td>
</tr>
<tr>
<td>Dinner</td>
<td>German</td>
<td>Fruit</td>
</tr>
<tr>
<td>Lunch or Dinner</td>
<td>Greek</td>
<td>Soup</td>
</tr>
<tr>
<td>Any Meal</td>
<td>Italian</td>
<td>Meat</td>
</tr>
<tr>
<td></td>
<td>Jewish</td>
<td>Poultry</td>
</tr>
<tr>
<td></td>
<td>Polish</td>
<td>Fish</td>
</tr>
<tr>
<td>Flavor</td>
<td>Prep. Type</td>
<td>Cold Plate</td>
</tr>
<tr>
<td>Bland</td>
<td>Bake</td>
<td>Hot Sandwich</td>
</tr>
<tr>
<td>Mild</td>
<td>Boil</td>
<td>Cold Sandwich</td>
</tr>
<tr>
<td>Strong</td>
<td>Broil</td>
<td>Entree</td>
</tr>
<tr>
<td></td>
<td>Braize</td>
<td>Starch</td>
</tr>
<tr>
<td>Color</td>
<td>Cool</td>
<td>Vegetable</td>
</tr>
<tr>
<td>Red</td>
<td>Freeze</td>
<td>Tossed Salad</td>
</tr>
<tr>
<td>Orange</td>
<td>Fry</td>
<td>Vegetable Salad</td>
</tr>
<tr>
<td>White</td>
<td>Heat</td>
<td>Fruit Salad</td>
</tr>
<tr>
<td>Yellow</td>
<td>Puree</td>
<td>Gelatin Salad</td>
</tr>
<tr>
<td>Green</td>
<td>Sauté</td>
<td>Cottage Cheese</td>
</tr>
<tr>
<td>Brown</td>
<td></td>
<td>Cake</td>
</tr>
<tr>
<td>Purple</td>
<td></td>
<td>Pie</td>
</tr>
<tr>
<td>Multi-color</td>
<td></td>
<td>Pudding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cookies</td>
</tr>
<tr>
<td>Texture</td>
<td>Shape</td>
<td>Other Dessert</td>
</tr>
<tr>
<td>Liquid</td>
<td>Diced</td>
<td>Bread</td>
</tr>
<tr>
<td>Soft</td>
<td>Sliced</td>
<td>Cereal</td>
</tr>
<tr>
<td>Medium</td>
<td>Chunk</td>
<td>Beverage</td>
</tr>
<tr>
<td>Crisp</td>
<td>Julienne</td>
<td>Jams</td>
</tr>
<tr>
<td>Chewy</td>
<td>Balls</td>
<td>Sauce</td>
</tr>
<tr>
<td>Frequency</td>
<td>Shred</td>
<td>Casserole</td>
</tr>
<tr>
<td>Twice a day</td>
<td>Squares</td>
<td></td>
</tr>
<tr>
<td>Once a day</td>
<td>Amorphous</td>
<td></td>
</tr>
<tr>
<td>Every other day</td>
<td>Circles</td>
<td></td>
</tr>
<tr>
<td>Twice a week</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Once a week</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Every other week</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Once a month</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nutrient Attributes</td>
<td>Nutrient Codes</td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>Water (%)</td>
<td>Wat</td>
<td></td>
</tr>
<tr>
<td>Calories</td>
<td>Cal</td>
<td></td>
</tr>
<tr>
<td>Protein (gm)</td>
<td>Pro</td>
<td></td>
</tr>
<tr>
<td>Fat (gm)</td>
<td>Fat</td>
<td></td>
</tr>
<tr>
<td>Carbohydrate (gm)</td>
<td>Car</td>
<td></td>
</tr>
<tr>
<td>Fiber (gm)</td>
<td>Fib</td>
<td></td>
</tr>
<tr>
<td>Ash (gm)</td>
<td>Ash</td>
<td></td>
</tr>
<tr>
<td>Calcium (gm)</td>
<td>Cal</td>
<td></td>
</tr>
<tr>
<td>Phosphorous (mg)</td>
<td>Pho</td>
<td></td>
</tr>
<tr>
<td>Iron (mg)</td>
<td>Iro</td>
<td></td>
</tr>
<tr>
<td>Sodium (mg)</td>
<td>Sod</td>
<td></td>
</tr>
<tr>
<td>Potassium (mg)</td>
<td>Pot</td>
<td></td>
</tr>
<tr>
<td>Vitamin A (IU)</td>
<td>Vit</td>
<td></td>
</tr>
<tr>
<td>Thiamin (mg)</td>
<td>Thi</td>
<td></td>
</tr>
<tr>
<td>Riboflavin (mg)</td>
<td>Rib</td>
<td></td>
</tr>
<tr>
<td>Niacin (mg)</td>
<td>Nia</td>
<td></td>
</tr>
<tr>
<td>Ascorbic Acid (mg)</td>
<td>Asc</td>
<td></td>
</tr>
<tr>
<td>Sat. Fatty Acid (gm)</td>
<td>Sat</td>
<td></td>
</tr>
<tr>
<td>Olein Acid (gm)</td>
<td>Ole</td>
<td></td>
</tr>
<tr>
<td>Lineolic Acid (gm)</td>
<td>Lin</td>
<td></td>
</tr>
<tr>
<td>Cholesterol (gm)</td>
<td>Cho</td>
<td></td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


