A NATURAL LANGUAGE INTERFACE TO DATABASES

Prepared by:

D.R. Ford
Johnson Research Center
The University of Alabama in Huntsville
Huntsville, AL 35899

Prepared for:

Tim Crumbley
System Software Branch
Information and Electronic Systems Lab
George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

February 1990
TABLE OF CONTENTS

ABSTRACT...................................................................................................................... 1
1.0 Natural Language Interface to Databases.............................................................. 2
1.1 Task Statement........................................................................................................ 3
1.2 Task Conditions...................................................................................................... 3
1.3 Task Approach........................................................................................................ 4
1.4 Task Results........................................................................................................... 17

Appendix A............................................................................................................... 19
Appendix B................................................................................................................ 25
ABSTRACT

This paper presents the development of a Natural Language Interface (NLI) which is semantic-based and uses Conceptual Dependency representation. The system was developed using Lisp and currently runs on a Symbolics Lisp machine.
1.0 Natural Language Interface to Databases

Natural languages are the languages used by people in the course of their daily affairs, for example, English, French, Japanese, etc. Natural languages are used to express a broad range of ideas to others. Given enough attention, nearly any concept that comes to mind can be conveyed to another person through a common natural language. Some concepts are easy to express, such as, "I am hungry," whereas others may require lengthy explanations. The prime characteristic of natural languages is that they can be used to express nearly all the concepts that occur to the people who speak and understand them.

The word natural emphasizes a contrast with artificial languages. Artificial languages are those that have been designed to be highly expressive over a limited range of ideas. Musical notation is an artificial language. Another set of artificial languages is programming languages. These are interesting because, like natural languages, they can be used to express a broad range of concepts. LISP, for instance, is an extendable language, that is, if an idea is difficult to express in its current form, it can be improved at will. But programming languages have been designed with their application to computers in mind, and this has affected their form. Programming languages have been written so as to be analyzed easily by computers.

Research in natural language understanding is concerned with making computers capable of using natural languages. There are two reasons for this. First, computers that can use natural languages would undeniably be a useful tool. It would mean that a person in need of information retrieval or information processing on a computer could obtain it without having to learn a computer language or go through an intermediary. They would not have to worry about becoming fluent in a "foreign" language and maintaining that fluency just to
accomplish their jobs. A computer that could use natural languages could read normal text, providing users with access to computer-generated summaries or reports synthesized from reading several text sources.

The second motivation for natural language research is that it will increase our understanding of how human languages and minds work. To develop the technology for a computer to use language, we must first be able to say specifically what language is. We must be able to say precisely how the concepts we wish to express can be represented in the computer. Building computer programs requires this precision and attention to detail. A programming implementation of a theory of language can be used to identify flaws, inconsistencies, and areas of incompleteness that may go unnoticed.

1.1 Task Statement

The purpose of this task was to develop an interface to a database in order to determine the feasibility of such an interface. Also, the desirability of the interface was to be addressed. In addition, a secondary task was to gain a better understanding of what the capabilities of such an interface should be, and to determine some of the limits of this type of interface.

1.2 Task Conditions

The natural language interface was to be developed on a Symbolics 3600 series Lisp machine using Symbolics Lisp. The interface should understand a limited subset of English in order to allow a user to query a database. The database should be located on another computer other than the Symbolic's machine that contained the interface. In addition, it should be on a completely different computer from the Symbolic's. There were no assumptions
or conditions placed on the means of communication between the Symbolic's machine and the computer with the database.

1.3 Task Approach

A NL Query generator prototype is being developed at the Johnson Research Center at the University of Alabama in Huntsville wherein queries, in natural language are generated for dbaseIII+. The main program resides on a symbolics 3620 machine while the database resides on an IBM personal computer. The database is manipulated through commands from the Symbolics. The communication protocol is established via RS 232. The process of database query and results are shown by the following figure:

A Graphic representation of the Natural Language Query Generator
The user types in a query in natural language on the symbolics. The parser translates it into Conceptual Dependency representation and generates a dbase query which is communicated to the PC via the RS 232. The RS 232 was chosen over others as the main idea was to set up communications between the Symbolics and the P.C. The manipulation is performed on the database and the results are communicated back to the user on the symbolics. The significant point of the exercise is that the user is not restricted to using specific dbase commands for manipulating the database. He can do so in the manner and language he prefers (provided it is in English).

The database is a simple one designed to represent the student records. It has been designed more to test the execution of the program and the generated queries. The following is a section of the database:

<table>
<thead>
<tr>
<th>RECORD #</th>
<th>NAME</th>
<th>ST. NUM</th>
<th>S. S. NUM</th>
<th>SEX</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ash</td>
<td>12345</td>
<td></td>
<td>m</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>Dion</td>
<td>67897</td>
<td></td>
<td>m</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>John</td>
<td>70601</td>
<td></td>
<td>m</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Mike</td>
<td>46893</td>
<td></td>
<td>m</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Lisa</td>
<td>25789</td>
<td></td>
<td>f</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Eddie</td>
<td>71214</td>
<td></td>
<td>m</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Linda</td>
<td>45109</td>
<td></td>
<td>f</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>Cynthia</td>
<td>28633</td>
<td></td>
<td>f</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Ben</td>
<td>65432</td>
<td></td>
<td>m</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>Gedro</td>
<td>64646</td>
<td></td>
<td>m</td>
<td>33</td>
</tr>
<tr>
<td>11</td>
<td>Paige</td>
<td>29099</td>
<td></td>
<td>m</td>
<td>38</td>
</tr>
<tr>
<td>12</td>
<td>Bernie</td>
<td>10001</td>
<td></td>
<td>m</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>Donnie</td>
<td>53200</td>
<td></td>
<td>f</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>Darlene</td>
<td>66677</td>
<td></td>
<td>f</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>Rica</td>
<td>99999</td>
<td></td>
<td>f</td>
<td>20</td>
</tr>
</tbody>
</table>
Execution of Database Manager Program on the P.C

The DB_MGR.PRG program first runs the basic program GETQUERY.BAS which receives the query from the Symbolics and writes a Dbase III program called QUERY.PRG. DB_MGR.PRG then executes QUERY.PRG storing the results in RESULTS.TXT. DB_MGR.PRG then runs the basic program SENDRES.BAS which sends the contents of RESULTS.TXT back to the Symbolics. Finally, DB_MGR.PRG loops back to GETQUERY.BAS, waiting for the next query from the Symbolics.

The words and the expressions are all defined in the dictionary. The database can handle all display and retrieve DBase commands from the Natural Language Query generator.

Working Examples of the Interface

1. The first example demonstrates the use of the verb "List". The word "List" is a Dbase III command which performs, as the name suggests, the function of Listing the field names, required by the user. In this case, the user wants the Natural Language Query Generator (NLQG) to generate a query for listing all the males in the database. The user presses the Select key and the Natural Language Query Generator is displayed on the screen, with the prompt-Query. The user then types in the command: "List all males". The NLQG generates the query in Conceptual Dependency, the communication protocol is established with the P.C. and the command chain is established in the manner described above. The command is then executed in dBASE III and the results flash for a second on the P.C. before they are communicated to the Symbolics and displayed there. The user types in his/her query in the the top half of the screen and the results are displayed on the bottom half of the screen.
The response to the query "List all males" is the Record number and the Names of all the males in the database. A printout of the screen is displayed below:

![Figure 1](image-url)
2. In the manner stated above, if the user wants to retrieve the names of all the females in the database, he/she types in the command "List all females." The names of all the females in the database will be displayed in the bottom half of the screen.

The text above the output, i.e.

(DISPLAY DB-FIELD-VALUE (DB-FIELD-VALUE VALUE ("f") OPERATOR ("=") FIELD (SEX)))

is the Conceptual Dependency representation of the typed in text. It tells the computer to look in the database records which have a value of "f" (meaning female) in the field "Sex" and display the contents of all the fields in the records matching the search. The query and the output are shown in the following figure:

---

**Figure 2**

---
3. In this example, the user wants to retrieve the name of women, who are more than 30 years of age. However, he is not restricted to the "List" or "Display" commands, which are DBase commands. He/she can just type in "show" and whatever records he/she wants to get and the Query Generator will retrieve it for him/her. Only one record matches the query and the same is displayed at the bottom of the screen. The output is shown in the following figure:

<table>
<thead>
<tr>
<th>Natural Language Query Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Dictionary</td>
</tr>
<tr>
<td>Set Up Help</td>
</tr>
<tr>
<td>Issue DBase Commands</td>
</tr>
</tbody>
</table>

- Query:
  - Query: list all males
  - Query: list all females
  - Query: show all women over 30 years of age
- Query: show all women over 30 years of age

<table>
<thead>
<tr>
<th>Dialog</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 Rice</td>
</tr>
<tr>
<td>(DISPLAY DB-FIELD-VALUE &quot;(f&quot;) OPERATOR (&quot;=&quot;) FIELD (SEX))</td>
</tr>
<tr>
<td>DB-FIELD-VALUE DB-FIELD-VALUE FIELD (AGE) OPERATOR (&quot;&lt;=&quot;) VALUE (NUMBER VALUE (30))</td>
</tr>
<tr>
<td>DB-FIELD (DB-FIELD NAME (AGE))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Records</th>
<th>NAME</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linda</td>
<td>35</td>
</tr>
</tbody>
</table>

![Figure 3]

4. To demonstrate the capability of the NLQG, we can use the following example. The user can use any word synonymous with "List" or "Display" in the manner and the
NLOQ will retrieve the records required. The output and the query are shown in the following figure:

![Figure 4]

Figure 4
5. In this example, the user wants to retrieve the name and Social Security Number of all the males in the data base. In stead of using the word "Retrieve," he/she uses the word "Get." The query and the result are shown in the following figure:

![Natural Language Query Generator](image)

**Figure 5**
6. In this example, the user uses the word "Retrieve" to display the name and student number of all the males in the database. The query and the output are shown below:

```
Figure 6
```
7. To demonstrate the capability of the NLQG to handle different words of the English Language, the following example is used. Instead of "Females" the user uses the word "Women"—The NLQG recognizes that women and females mean the same thing and retrieves the name and student number of all women. It is displayed in the following figure:
8. In the following example the user uses the word find instead of retrieve or get to display the name and sex of all members of the database. Instead of saying male or female, he/she just types in all and the NLOQG retrieves the name and sex of all the members. The following figure illustrates the query and the output.

![Figure 8](image-url)
9. This example the user uses the word find to retrieve the name and sex of all males, but uses the word men instead. In addition to this, he also types in the word "please." The NLQG ignores the "please" in that it adds no pertinent new information to the query. The query and output are shown in the following figure.

![Figure 9](image-url)
10. In this example the user asks the query in the form of a question. The NLOG retrieves and displays the names of all females. The query and its output are shown in Figure 10.
1.4 Task Results

The specifics stated in the Task Statement were successfully completed. The interface was developed and operates on the Symbolic's Lisp machine. The database can be queried from the Symbolic's and the data is returned to the Symbolic's. The interface allows users to query the database in their natural language, if it's English. The interface understands a limited subset of English.

However, novice users can use the interface to query the database, but they still must know some things about the database. For example, they must know the field names used in the construction of the database. Using the above example, the user would have to know that the database contained information about males and females. They would not have to know the exact field name. This is provided for in the dictionary; i.e., other words used to describe the same concept are identified and linked to the appropriate definition. An example of this is using men for male or women for female.

The solution to this type of problem is to develop a generic interface system. However, it is difficult to develop a such a system; i.e., one that will allow the user to simply ask what databases the system knows about and to use any terms to query the system. In order to develop a system like this more time and money needs to be allocated. Another problem associated with the generic system is that the interface has to know about each database and the terms it uses to describe the data. These terms have to be defined in the dictionary. This makes the interface database dependent. In order to make the interface work with another database, these termsshave to be redone. Also, if the database changes a similar process must be accomplished.

This approach holds much promise of making database use by novice users simpler. The generic system is not an absurdity. If it is approached
correctly, parts of this concept could prove beneficial to users. The ability to explain what databases it knows about is feasible at present, as well as, being able to develop an interface that will allow a user to teach it about new databases so that its capabilities can increase. These extensions to the present research would simply require time for development.
Appendix A

Listing of Words Used by the NLI in Lisp Format
(learn-words

'; ; Syntax: Common-Lisp; Package: COMMON-LISP-USER; Base: 10; Mode: LISP -x-

(defun learn-words ()
  (def (human name (john)
    gender (male))
  demons (save-character)))

(defun pick ()
  demons ((pick-up?) (decide?) (determine-voice))
  n1 (grasp actor h <= (exp-verb-human 'human before)
    object x <= (exp-verb-phys-obj 'after)
    instr (move actor h
      object (fingers)
      to x))
  n2 (build actor i <= (exp-verb-human 'human before)
    object (pass actor i <= (exp-verb-human 'human before)
      object i <= (exp-verb-human 'human phys-obj 'after))))

(defun up ()
  demons (ignore))

(defun the ()
  demons (ignore))

(defun ball ()
  def (phys-obj class (gane-obj)
    name (ball))
  demons (save-object))

(defun and ()
  def (conjunction conjunct1 <= (exp-verb-human 'db-field phys-obj human 'before)
    conjunct2 <= (exp-verb-human 'db-field phys-obj human 'after))
  demons (determine-voice))

(defun dropped ()
  def (prem actor i <= (exp-verb-human 'human before)
    object thing <= (exp-verb-human 'phys-obj 'after)
    instr (propel actor (gravity)
      object thing)
  demons (ignore))

(defun it ()
  def (pronoun)
  demons (ignore))

(defun in ()
  def (prep is (in))
  demons (ins-aft 'phys-obj setting 'prepobj))

(defun box ()
  def (phys-obj class (container)
    name (box)))

(defun pcb ()
  def (process-object name (printed-circuit-board)
    demons (save-object))
  demons (save-object))

(defun enter ()
  def (prem actor nil
    object thing <= (exp-verb-human 'process-object pronoun 'before)
    to thing <= (exp-verb-human 'complex process-actor pronoun 'after))
  demons (set-sentence-number (determine-voice)))

(defun exit ()
  def (prem actor nil
    object thing <= (exp-verb-human 'process-object pronoun 'before)
    from thing <= (exp-verb-human 'complex process-actor pronoun 'after))
  demons (set-sentence-number (determine-voice)))

(defun process ()
  def (do actor i <= (exp-verb-human 'process-object complex pronoun 'before)
    object i <= (exp-verb-human 'process-object pronoun 'after))
  demons (set-sentence-number (determine-voice)))

(defun proceed ()
  def (prem actor nil
    object i <= (exp-verb-human 'process-object pronoun 'before)
    to thing <= (exp-verb-human 'complex process-actor pronoun 'after))
  demons (set-sentence-number (determine-voice)))

(defun go ()
  def (prem actor nil
    object i <= (exp-verb-human 'process-object pronoun 'before)
    to thing <= (exp-verb-human 'complex process-actor pronoun 'after))
  demons (set-sentence-number (determine-voice)))
ANDY:=>ash>nl>words.lisp.73

(arrive def (ptrn~ctor nil)
 object + get (exp = vocal "process object pronoun") 'before)
 denom ((get sentence-number) (determine voice))

(to def (prep is (to))
 denom (ins-act 'process-actor pronoun) 'preobj))

(at def (prep is (at))
 denom (ins-aft 'complex process-actor pronoun) 'preobj))

(second def (time name (second)
 base-units (1))
 denom ((attach-time 'dist-type) 'before) (how-many 'quantity 'suffix=0's)))

(minute def (time name (minute)
 base-units (60))
 denom ((attach-time 'dist-type) 'before) (how-many 'quantity 'suffix=0's)))

(hour def (time name (hour)
 base-units (3600))
 denom ((attach-time 'dist-type) 'before) (how-many 'quantity 'suffix=0's)))

(for def (prep is (for))
 denom (ignor))

(a def (ignor))

(ne def (ignor))

(then def (ignor))

(where def (ignor))

(next def (ignor))

(is def (be-verb name (is))
 denom (ignor))

(are def (be-verb name (are))
 denom (ignor))

(by def (prep is (by))
 denom (ignor))

(of def (prep is (of))
 denom (ignor))

(around def (prep is (around))
 denom (ignor))

(near def (prep is (near))
 denom (ignor))

(with def (prep is (with))
 denom (ignor))

(sic def (complex name (automatic-insertion))
 denom (save-complex))

(mi def (complex name (manual-insertion))
 denom (save-complex))

(mic def (complex name (manual-insertion))
 denom (save-complex))

(testing def (complex name (test-and-assembly))
 denom (save-complex))

(dip def (process-actor class (station)
 name (dip-mach))
 denom ((save-actor) (how-many 'quantity 'suffix=0's)))

20
ANDY:~ash>nl>words.lisp.73 2/20/90 11:31:26  Page 3

(vcd def (process-actor class (station)
        name (vcd-machine))
demons ((save-actor)(how many 'quantity 'suffix 0 's)))

(tdk def (process-actor class (station)
        name (tdk-machine))
demons ((save-actor)(how many 'quantity 'suffix 0 's)))

(rli def (process-actor class (station)
        name (tdk-machine))
demons ((save-actor)(how many 'quantity 'suffix 0 's)))

(bpi def (process-actor class (station)
        name (berg-pin-machine))
demons ((save-actor)(how many 'quantity 'suffix 0 's)))

(swedge def (process-actor class (station)
           name (swedge-nut-machine))
demons ((save-actor)(how many 'quantity 'suffix 0 's)))

(qc def (process-actor class (station)
          name (quality-control))
demons (save-actor))

(assembly def (process-actor class (station)
              name (mechanical-assembly))
demons (save-actor))

(shipping def (process-actor class (station)
               name (shipping))
demons (save-actor))

(storage def (process-actor class (station)
             name (shipping))
demons (save-actor))

(mean def (statistic name (mean)
           measure * <== (find-stat-value))

(sd def (statistic name (standard-deviation)
         measure * <== (find-stat-value))

(swedge def (process-actor class (station)
             name (swedge-nut-machine))
demons ((save-actor)(how many 'quantity 'suffix 0 's)))

(poisson def (dist-type name (poisson)
               nlt * <== (exp-statistic 'mean-interarrival-time' after))
demons (ins-bef 'ptrans do 'dist))

(normal def (dist-type name (normal)
            mean * <== (exp-statistic 'mean' after)
            sd * <== (exp-statistic 'standard-deviation' after))
demons (ins-bef 'ptrans do 'dist))

(uniform def (dist-type name (uniform)
               nln * <== (exp-statistic 'min' after)
               nax * <== (exp-statistic 'max' after))
demons (ins-bef 'ptrans do 'dist))

(min def (statistic name (min)
          measure * <== (find-stat-value))

(minmax def (statistic name (min)
              measure * <== (find-stat-value))

(max def (statistic name (max)
           measure * <== (find-stat-value))}
(maximum def (statistic name (max)
  measure #: (find-stat-value)))

(nit def (statistic name (mean-interval-time)
  measure #: (find-stat-value)))

(mike def (human name (mike)
  gender (male))
  demons (save-character))

(ate def (trans actor nil)
  object #: (exp-wrt-voice "process-object noun" 'before)
  to #: (find (prep '(a the) "(complex process-actor noun)" 'after)
  demons ((get-sentence-number) (determine-voice)))

(an def (prep is (an))
  demons (ignore))

(apple def (food type (apple)))

(a def (prep is (a))
  demons (ignore))

(years def (trans actor nil)
  object #: (exp-wrt-voice "human" 'before)
  to #: (find (prep '(a the) "(garment)" 'after)
  demons ((get-sentence-number) (determine-voice)))

(shirt def (garment type (shirt)))

(retrieve def (trans actor nil)
  object #: (exp-wrt-voice "process-object noun" 'before)
  to #: (find (prep '(from) "(process actor class)" 'after)
  demons ((get-sentence-number) (determine-voice)))

(get def (trans actor nil)
  object #: (exp-wrt-voice "process-object pronoun" 'before)
  to #: (find (prep '(from) "(complex process-actor pronoun)" 'after)
  demons ((get-sentence-number) (determine-voice)))

(shot n1 (propel object (bullets))
  def (trans actor nil)
  object #: (exp-wrt-voice "process-object pronoun" 'before)
  to #: (find (prep '(the) "(process actor class)" 'after)
  demons ((get-sentence-number) (determine-voice))
  n2 (# take-picture))

(from demons (ignore))

(insert def (trans actor nil)
  object #: (exp-wrt-voice "process-object pronoun" 'before)
  to #: (find (prep '(in) "(process actor class)" 'after)
  demons ((get-sentence-number) (determine-voice)))

(delete def (trans actor nil)
  object #: (exp-wrt-voice "process-object pronoun" 'before)
  to #: (find (prep '(from) "(complex process-actor pronoun)" 'after)
  demons ((get-sentence-number) (determine-voice)))

(modify def (trans actor nil)
  object #: (exp-wrt-voice "process-object pronoun" 'before)
(setq 'actor (+ 1 actor))
to 1 (cond (from) ((complex process actor pronoun) 'after)))

(print def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(fetch def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(retrieve def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(get def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(show def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(select def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(list def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(enumerate def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(display def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(give def (ptrans actor (display))
  object (exp-wrt-voice '(db-field) 'after)
  objects (exp-wrt-voice '(conjunction) 'after)
  denoms ((get-sentence-number) (determine-voice)))

(all denoms (ignore)))

(age def (db-field name 'age)
  denoms (save-object))

(sex def (db-field name 'sex)
  denoms (save-object))

(name def (db-field name 'name)
  denoms (save-object))

(people def (db-field name 'people)
  denoms (save-object))

(person def (db-field name 'person)
  denoms (save-object))

(male def (db-field-value value 'm)
  field (sex)
  denoms (ins-bef '(ptrans) 'db-field-value))

(female def (db-field-value value 'f)
  field (sex)
  denoms (ins-bef '(ptrans) 'db-field-value))
(man def (db-field-value value "m")
  field (sex))
  (woman def (db-field-value value "f")
    field (sex))
  (table def (process-object name (row))))
Appendix B

Listing of Expressions Used by the NLI in Lisp Format
(learn-expressions

'(((automatic-insertion-center)
  def (complex-name (automatic-insertion))
  def (save-complex))

(((a1-work-center)
  def (complex-name (automatic-insertion))
  def (save-complex))

(((numeric-insertion)
  def (complex-name (automatic-insertion))
  def (save-complex))

(((automatic-insertion-work-center)
  def (complex-name (automatic-insertion))
  def (save-complex))

(((a1-center)
  def (complex-name (automatic-insertion))
  def (save-complex))

(((manual-insertion-center)
  def (complex-name (manual-insertion))
  def (save-complex))

(((manual-insertion)
  def (complex-name (manual-insertion))
  def (save-complex))

(((manual-insertion-work-center)
  def (complex-name (manual-insertion))
  def (save-complex))

(((manual-load)
  def (complex-name (manual-insertion))
  def (save-complex))

(((manual-load-center)
  def (complex-name (manual-insertion))
  def (save-complex))

(((manual-load-work-center)
  def (complex-name (manual-insertion))
  def (save-complex))

(((test-and-assembly)
  def (complex-name (test-and-assembly))
  def (save-complex))

(((test-and-assembly-center)
  def (complex-name (test-and-assembly))
  def (save-complex))

(((test-and-assembly-work-center)
  def (complex-name (test-and-assembly))
  def (save-complex))

(((testing-work-center)
  def (complex-name (test-and-assembly))
  def (save-complex))

(((testing-center)
  def (complex-name (test-and-assembly))
  def (save-complex))

(((testing-and-assembly-work-center)
  def (complex-name (test-and-assembly))
  def (save-complex))

)
(def (complex name (test-and-assembly))
  defons (save-complex))

(def (complex name (test-and-assembly))
  defons (save-complex))

((t & a) center)
  def (complex name (test-and-assembly))
  defons (save-complex))

((t & a) center)
  def (complex name (test-and-assembly))
  defons (save-complex))

((finished goods) center)
  def (complex name (finished-goods))
  defons (save-complex))

((dip machine) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix10 's)))

((dual inline package insertion machine) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix40 's)))

((dual inline package insertion machine) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix30 's)))

((dual inline package insertion) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix30 's)))

((dip insertion) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix10 's)))

((dip insertion machine) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix20 's)))

((masking machine) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix20 's)))

((masking) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix20 's)))

((vcd machine) center)
  def (process-actor class (station))
  defons ((save-actor)(how-many 'quantity 'suffix10 's)))
((ucd insertion machine)
def (process-actor class (station)
  name (ucd-machine))
demons ((save-actor)(how-many 'quantity 'suffix20 's)))

((ucd insertion)
def (process-actor class (station)
  name (ucd-machine))
demons ((save-actor)(how-many 'quantity 'suffix10 's)))

((variable center distance insertion)
def (process-actor class (station)
  name (vcd-machine))
demons ((save-actor)(how-many 'quantity 'suffix10 's)))

((variable center distance insertion machine)
def (process-actor class (station)
  name (vcd-machine))
demons ((save-actor)(how-many 'quantity 'suffix40 's)))

((tdk machine)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix10 's)))

((radial lead insertion machine)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix30 's)))

((radial lead insertion)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix20 's)))

((rii machine)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix10 's)))

((rii insertion machine)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix20 's)))

((radial lead machine)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix20 's)))

((radial lead)
def (process-actor class (station)
  name (tdk-machine))
demons ((save-actor)(how-many 'quantity 'suffix20 's)))

((berg pin machine)
def (process-actor class (station)
  name (berg-pin-machine))
demons ((save-actor)(how-many 'quantity 'suffix20 's)))

((berg pin)
def (process-actor class (station)
  name (berg-pin-machine))
demons ((save-actor)(how-many 'quantity 'suffix10 's)))

((berg machine)
def (process-actor class (station)
  name (berg-pin-machine))
demons ((save-actor)(how-many 'quantity 'suffix10 's)))

((berg pin insertion machine)
def (process-actor class (station)
  name (berg-pin-machine))
{{{save-actor}(how-many 'quantity 'suffix30 's))}}

(defun (berg-pin-insertion)
def (process-actor class (station)
  name (berg-pin-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix20 's))}}

(defun (bpi-machine)
def (process-actor class (station)
  name (berg-pin-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix10 's))}}

(defun (swedge-nut-machine)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix20 's))}}

(defun (swedge-nut)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix10 's))}}

(defun (swedge-machine)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix10 's))}}

(defun (spanish-terminal-insertion-machine)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix30 's))}}

(defun (spanish-terminal-insertion)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix20 's))}}

(defun (spanish-terminal)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix10 's))}}

(defun (spanish-terminal-machine)
def (process-actor class (station)
  name (swedge-nut-machine))
demos ({{{save-actor}(how-many 'quantity 'suffix20 's))}}

(defun (component-preparation)
def (process-actor class (station)
  name (component-preparation))
demos (save-actor))

(defun (component-prep)
def (process-actor class (station)
  name (component-preparation))
demos (save-actor))

(defun (hand-load)
def (process-actor class (station)
  name (manual-load))
demos (save-actor))

(defun (manual-load)
def (process-actor class (station)
  name (manual-load))
demos (save-actor))

(defun (wave-solder-machine)
(def (process-actor class (station)
    name (wave-solder-machine))
  (save-actor)

(isave (wave-solder)
  def (process-actor class (station)
    name (wave-solder-machine))
  (save-actor))

((aqua clean machine)
  def (process-actor class (station)
    name (aqua-clean-machine))
  (save-actor)(how-many 'quantity 'suffix28 's)))

((aqua clean)
  def (process-actor class (station)
    name (aqua-clean-machine))
  (save-actor))

((aqua bath)
  def (process-actor class (station)
    name (aqua-clean-machine))
  (save-actor)(how-many 'quantity 'suffix29 's)))

((secondary operations)
  def (process-actor class (station)
    name (secondary-operations))
  (save-actor))

((secondary ops)
  def (process-actor class (station)
    name (secondary-operations))
  (save-actor))

((quality control station)
  def (process-actor class (station)
    name (quality-control))
  (save-actor))

((qc station)
  def (process-actor class (station)
    name (quality-control))
  (save-actor))

((qc point)
  def (process-actor class (station)
    name (quality-control))
  (save-actor))

((repair station)
  def (process-actor class (station)
    name (repair-station))
  (save-actor))

((fault finder)
  def (process-actor class (station)
    name (fault-finder))
  (save-actor)(how-many 'quantity 'suffix10 's)))

((burn in)
  def (process-actor class (station)
    name (burn-in))
  (save-actor))

((automatic test)
  def (process-actor class (station)
    name (automatic-test))
  (save-actor))

((first functional test)
  def (process-actor class (station)
    name (automatic-test))
  (save-actor))
ANDY:>ash>nl(expressions.lisp.22

((mechanical assembly)
def (process-actor class (station)
  name (mechanical-assembly))
def (save-actor))

((final functional test)
def (process-actor class (station)
  name (final-functional-test))
def (save-actor))

((final inspection)
def (process-actor class (station)
  name (final-inspection))
def (save-actor))

((button up)
def (process-actor class (station)
  name (button-up))
def (save-actor))

((according to)
def (save-actor))

((poisson process)
def (dist-type name (poisson)
  mean \approx (exp-statistic '(mean-interarrival-time) 'after))
def (ins-bef '(ptrans do) 'dist))

((poisson distribution)
def (dist-type name (poisson)
  mean \approx (exp-statistic '(mean-interarrival-time) 'after))
def (ins-bef '(ptrans do) 'dist))

((poisson model)
def (dist-type name (poisson)
  mean \approx (exp-statistic '(mean-interarrival-time) 'after))
def (ins-bef '(ptrans do) 'dist))

((normal process)
def (dist-type name (normal)
  mean \approx (exp-statistic '(mean) 'after)
  sd \approx (exp-statistic '(standard-deviation) 'after))
def (ins-bef '(ptrans do) 'dist))

((normal distribution)
def (dist-type name (normal)
  mean \approx (exp-statistic '(mean) 'after)
  sd \approx (exp-statistic '(standard-deviation) 'after))
def (ins-bef '(ptrans do) 'dist))

((normal model)
def (dist-type name (normal)
  mean \approx (exp-statistic '(mean) 'after)
  sd \approx (exp-statistic '(standard-deviation) 'after))
def (ins-bef '(ptrans do) 'dist))

((uniform distribution)
def (dist-type name (uniform)
  min \approx (exp-statistic '(min) 'after)
  max \approx (exp-statistic '(max) 'after))
def (ins-bef '(ptrans do) 'dist))

((uniform process)
def (dist-type name (uniform)
  min \approx (exp-statistic '(min) 'after)
  max \approx (exp-statistic '(max) 'after))
def (ins-bef '(ptrans do) 'dist))

((uniform model)
def (dist-type name (uniform)
  min \approx (exp-statistic '(min) 'after)
  max \approx (exp-statistic '(max) 'after))
def (ins-bef '(ptrans do) 'dist))

30
((minimum value)
def (statistic name (min)
  measure (find-stat-value)))

((maximum value)
def (statistic name (max)
  measure (find-stat-value)))

((print circuit board)
def (process-object name (printed-circuit-board)
  demons ((save-object):(how-many 'quantity 'suffix 's))))

((standard deviation)
def (statistic name (standard-deviation)
  measure (find-stat-value)
  demons (ignor)))

((std dev)
def (statistic name (standard-deviation)
  measure (find-stat-value)
  demons (ignor)))

((mean interarrival time)
def (statistic name (mean-interarrival-time)
  measure (find-stat-value)))

((memory)
def (process-actor class (station)
  name (memory)
  demons ((save-actor):(how-many 'quantity 'suffix 's))))

((data-base)
def (process-actor class (station)
  name (data-base)
  demons ((save-actor):(how-many 'quantity 'suffix 's))))

((student number) def (db-field name (stnum)
  demons (save-object)))

((social security number) def (db-field name (ssnum)
  demons (save-object))))