Experiments with PLUM

W. Lehnert, K. Narasimhan, B. Draper
B. Stucky and M. Sullivan

COINS Technical Report 87-115

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Acknowledgement: This research was supported by NSF Presidential Young Investigators Award NSF IST-8351863 and DARPA contract N00014-85-K-0017.

1. Forward

This report consists of four projects undertaken by graduate students in COINS during the 1984-5 school year. At this time a number of us were involved in the design of a natural language sentence analyzer, PLUM (The Predictive Language Understanding Mechanism). PLUM was in a reasonably stable state, but we needed to reassure ourselves that the primary objectives of the PLUM Project were being met. These objectives are twofold in nature:

(1) We want a powerful natural language analyzer which can be used by students specializing in natural language who want to experiment with various theoretical problems in natural language processing, and

(2) We want a practical sentence analyzer that can be used by research assistants who need to build natural language interfaces for various AI systems whose primary goals may not be in natural language processing.

Users of the first variety require a general framework that can be modified and extended as needed without serious revision of PLUM itself, while users of the second type require a programming tool that is easy to learn and use with a minimal background in natural language processing theory. The four projects included in this report were chosen as representative problems spanning both types of user groups. In each case, the programming for these projects required only a few weeks of time on the part of a full-time graduate student for whom this was one of many course-related responsibilities. And in all but one case, the student involved was a first-year student whose experience with natural language processing systems was relatively minimal.

K. Narasimhan implemented a natural language interface for an intelligent operating system. A successful interface of this type has to exhibit robust language comprehension, and Narasimhan was careful to design his interface so that ungrammatical sentence fragments could be understood as well as complete sentences. An extensive list of sample inputs included in his report shows that PLUM lends itself to the design of flexible and robust language comprehension, at least in limited domains of disourse.

Bruce Draper constructed another natural language interface, but for a less general user community. Draper was interested in providing a useful interface for researchers working with a complex system for scene analysis. Programmers working with the VISIONS system at UMass. are frequently engaged in time consuming debugging sessions whenever a scene is interpreted incorrectly. VISPLUM was designed to test the feasibility of a natural language interface to expedite certain debugging activities. Draper's project appears to be a very promising step toward an effective interface, and we hope to see this work continued at some future time.

Brian Stucky undertook a more theoretically motivated project. He was interested in testing the generality of PLUM's control structure as a tool for experimentation in sentence analysis. Since a number of natural language processing systems employ augmented transition networks (ATN's), Draper chose to design an ATN interpreter that would receive an ATN specification as input and translate that specification into executable PLUM definitions. This system makes it possible to implement ATN's wihtout any additional knowledge of PLUM on the part of the user.

Mike Sullivan's experiment with PLUM was also more theoretical than practical. Sullivan wanted to investigate the sort of problems encountered when designing a system that must disambiguate pronominal references. His system, ANA, works by producing Conceptual Dependency representations for the input sentences, and using these representations to aid in the disambiguation process wherever possible. This project involved making some additions to PLUM's original implementation in order to preserve information relevant to pronoun disambiguation.

These four projects have served to reassure us that PLUM is meeting its general goals with reasonable success. The work reported here is not meant to represent significant progress in theoretical natural language processing. In each case, it would be more appropriate to view these accomplishments as first steps toward more serious undertakings. These steps have been made tenuously, and with a sense of playful experimentation. I doubt that any of the authors would want to publicize their efforts here as anything more than exploratory experimentation.

Each chapter that follows contains both a high-level description of the experiment taking place, as well as actual dictionary definitions for PLUM's use. We hope that the inclusion of these technical details will aid future PLUM users who are learning to use PLUM for the first time, without distracting more casual readers who merely want to see what sorts of things can be accomplished in the space of a few weeks. For a more detailed introduction to PLUM, see "The PLUM User's Manual" by W. Lehnert and S. Rosenberg (CPTM #1).

TABLE OF CONTENTS

A NATURAL LANGUAGE INTERFACE TO THE VAX/VMS EXAMPLE HELP SYSTEM

K. NARASIMHAN

1.0 INTRODUCTION

On-line help systems provide command assistance and assistance about certain concepts and standard tasks in operating systems. Evolution of operating systems has seen more of the assistance that has been traditionally offerred by help systems *move into the command language* itself. e.g. In VMS as one types **COPY**, the operating system prompts for the source file and the destination file in that order. There are operating system in which this order has been reversed. Persons having experience in such operating systems would have difficulty in keying in the parameters in the required order. In VMS this problem is avoided. However, it is a fact that most help systems are terse and unhelpful. This situation is remedied if two salient features are implemented in on-line help systems.

a. *Ability to understand a wide variety of natural language input.* This ability is all the more important from the perspective of a naive user or a user who has had experience on a different operating system. In such a case, he should be able to get on-line help by summoning the help for the required command by keying in the equivalent command in the operating system with which he is familiar. Failing this, the semantic import of the natural language input keyed in must be unambiguously identified with one of the commands.

Another aspect that is normally noticed in the use of on-line help systems in current systems is *the lack of a coherant discourse-like interaction* with the on-line help system. Previous work in natural language interfaces have been in request understanding. This allows the user to ask for assistance in natural language. This has been implemented in the UC system at Berkeley by Wilensky [WILE82]. Quite often it is desired by the user to have hypothetical interactions with the system rather then have the actual natural language input translated into command language constructs and executed. In such cases it is natural for the user to use at least a limited conversational mode with the help system.

The above implies that contextual infomation be carried on from each interaction to the next. This kind of requirement carries along with it, the accompanying problems of resolution of pronouns and elliptical uttrerances which are so natural in a conversation. However, such problems are not too pronounced if the domain is constrained and the expected input is fairly free from ambiguity. Summarizing, the variety of natural language input expected in a typical help system varies from single word, phrasal utterances, equivalent commands from other operating systems to a dialogue model with entailing problems of resolution of elliptical and anaphorical references.

b. Another aspect is *Improving on-line help with regard to quality of help provided*. There have been several approaches and they are summarized in [RISS84]. Some of them have been towards supplementing the existing help system with a help key, making the content of the help and error messages more concrete, responding to command synonyms etc.. Another approach is to improve the help system by providing examples of various categories, heuristics and pragmatic knowledge in the information provided to the user and embed user-tailored examples in the explanations.

2.0 OBJECTIVES

A help system which provides examples of various categories of command usage, heuristics and pragmatic knowledge in the information provided to the user and which provides user-tailored examples in the explanations already exists. This system at present takes in input in the form of single words (equivalent to commands) and produces as output, help messages augmented with examples. More of this would follow in the following pages. The main objective is to build a natural language interface to this help system. For this, the PLUM parser (Predictive Language Understanding Mechanism) [LEHN85] is used. More on this is covered in the a coming chapter. When one looks at issues of natural language in restricted domains, one repeatedly comes across these questions

-*What level of understanding must be achieved of the natural language input* - This is clearly a function of the domain. e.g **PRINT** in VMS command language or when given as natural language input could mean just printing on the printer and at worst, **PRINT ON THE SCREEN** would mean **TYPE**.

-*Handling of ill-formed input* - This could be ungrammatical input or "unnatural language input" as equivalent commands from another operating system. e. g. **LS** is a valid command in the UNIX operating system. Someone who has been trained in the UNIX system would be able to summon up the appropriate help for **DIRECTORY** in the VMS help system if the natural language interface does this.

-*Ability to maintain a minimum context focus* and to understand natural language input in the context of the interactions that have gone on before.

Another objective of this exercise is to achieve the above goals or answer the above questions. It is interesting to note, that as a result of this exercise, a lot is learned about the way the basic parser mechanism itself is to be used. Keepinf efficiency in mind, the frames and their associated control structures have to be constantly "trimmed", all the while keeping them as general as possible. This is

because this exercise serves not only the purpose of building a natural language interface, but also building a natural language interface but also learning about the requirements of the domain and how the parsing approach should vary with them.

3.0 EXAMPLE HELP SYSTEM

As set out earlier the existing example help system provides more ingredients of expert knowledge like examples of various categories, heuristics and pragmatic knowledge in the information that is given to the user. User-tailored examples are also embedded in the explanations. Taxonomic knowledge of examples is used to help select and order the presentation of examples. The absolute neophyte user is provided with "start-up" examples and the more experienced user with "references". Where a sequence of examples is called for, the taxonomic knowledge can be used to order the examples. For instance, references are presented before models which are presented before counter-examples and anomalies. Techniques of *Retrieval and Modification* and *Instantiation* are used by linking the explanation program with an example generator, which uses an Examples-Knowledge Base (EKB) of already existing examples together with procedures for modification and instantiation. The EKB consists of examples, represented as frames, harvested and organized by an expert. Procedurally attached to examples are instantiation and modification procedures like those to generate extreme variations or to personalize an example.

In the help system, the control of the help session is separated from its content. A Script-like control structure of text and examples organized in a template, a "TEXPLATE" which is the used to generate help's response. The explanations are assembled by retrieving the needed text and examples which are pointed to in the relevant texplate (texplates are indexed by commands, jargon and task keywords). A **Texplate Interpreter** controls the flow through the texplates, including which user options to present and what to do in response (**MORE** to go on with the explanation with further examples, **EXAMPLE** for an example, **QUIT** etc.) as well as directing the generation of sequences of examples, should the user repeatedly ask to see an example [RISS84].

4.0 PREDICTIVE LANGUAGE UNDERSTANDING MECHANISM

PLUM (the Predictive Language Understanding Mechanism) provides a control structure for integrated language analysis. PLUM does not make specific commitments to particular linguistic theories or strategies for conceptual information processing. Rather, the intent of PLUM is to provide a powerful framework that facilitates both theoretical experimentation and practical applications.

However, no general framework is totally devoid of some commitments to specific processing strategies and PLUM is no exception. PLUM is committed to predictive sentence analysis, where pre-defined frame structures are instantiated by slot-filling demons. PLUM also assumes that the memory representation created in response to each sentence (or group of sentences) will be multi-layered, utilizing vdistinct levels of linear memory organization. PLUM does not impose constraints on what sorts of frame structures are to be used, or what the individual levels of memory representation contain. PLUM can support levels of lexical analysis and syntactic analysis, as well as any variety of conceptual or semantic analysis that the system designer finds useful. PLUM is most accurately described as a control structure for language analysis. It does not embody a specific theory of natural language comprehension per se.

PLUM is a predictive sentence analyzer that seeks to build memory structures as it moves through each sentence. PLUM also allows for intra-sentential predictions so that general text analysis can be achieved with the same mechanisms used for sentence analysis. PLUM makes no specific commitments to styles of memory representations per se, beyond the use of frames, but there is an assumption about the organization of structures produced by PLUM. PLUM assumes that the memory representation being constructed will be *content-addressable* – that is, memory structures will be accessed by addressing their "type".

The *activate-instantiate* cycle is the driving mechanism behind all of PLUM's processing. Predictions about future information are made on the basis of existing structures and partially-confirmed structures. This is accomplished through the use of predictive *demons*. A predictive demon is associated with each slot that needs to be filled in a frame. These search for a particular type of memory structure, and they are told where to look in memory -- what level and what direction. Each search is restricted to a specific level of memory. A demon must also know if it is to look at past memory structures or structures waiting to be instantiated in the future, and it must know when to give up the search. Some searches will terminate at clause boundaries or sentence boundaries, while others may terminate only when memory is exhausted.

Every possible memory structure that PLUM constructs must be defined beforehand by the system designer. This is done by the use of conceptual definitions (static frame structures called **concept-frames**). In PLUM, a **concept-frame** is part of a function-call which contains additional information, about the frame structure, that is useful during run-time. The system knows which slots of the **concept-frame** must be filled before the structure can be instantiated and added to memory. It also contains information about how to find slot-fillers for the **concept-frame**, and which other predictions should be activated if the current prediction succeeds in instantiating its **concept-frame**.

Prediction prototypes are declarative data structures that create slot-filling demons when activated. As soon as a prediction is activated, its prototype is passed to an interpreter which creates slot-filling demons which are either run immediately, or added to the appropriate waiting list (if they are forward searching). The interpreter also creates an copy of the **concept-frame** which is altered by associated slot-filling demons when they locate appropriate structures to fill the frame.

PLUM, therefore, insulates the system designer from the lowest levels of its machinery. All demons are created automatically by the interpreter, so no one should ever write code for a demon directly. All the designer needs to learn are the conventions for specifying a search procedure declaratively inside the

function-call.

As more structures are added to memory, more and more predictions will be activated. Some of these will die off without success and some will retire when a structure is successfully located, but at any given time, a number of predictions may be pending. It is therefore necessary to impose some sort of control on these pending demons.

When a prediction is activated, the demons from that prediction are also activated. If they are backward-searching, they can conduct their search immediately; looking back through memory for the appropriate memory structure, either succeeding or failing. However, if they are forward-searching, they are stored on the waiting list of the type of structure for which they are looking. Then, when a memory structure is created, all demons on its waiting list are tested.

The basic execution cycle is to read a word, create a word structure in memory, and then test any demons that are waiting for that word structure. These demons can fill slots in frames, which will cause new structures to be created, thereby allowing other demons to be tested, etc. The result is a "depth-first" activation of the demons where one successful demon can set off a chain of activity that propagates through the set of waiting demons in the system. After those demons test, any predictions from the current word are activated. Since the waiting demons are tested before the new predictions are activated, timing problems are minimized.

5.0 INTERFACE TO THE EXAMPLE HELP SYSTEM

5.1 Types of Input

The types of input that can be envisioned in a natural language interface to an help system could vary in a continuum of very simple words and command language constructs to a sophisticated dialogue with the help system. In the current system, due to the separation of the natural language parser mechanism from the Example help system, it is not possible to handle very sophisticated dialogues since they involve a lot of interactions between the parser and the example help system. In addition, we need supervisors who handle keeping track of subtle shifts of focus, maintenance of user models etc.. Since here the natural language parser is only passing the parse to the example help system and there is no information passed back from the help system, the sophistication of the input that can be handled is limited. So the types of input that can be handled can be of two types:

a. Single words and phrases

b. Short-dialogues with a limited context mechanism. This context would just keep track of objects like commands, files etc.. that have been talked about in the previous interactions.

At present, the various types of natural language input that the parser can handle are of the first kind although the other is being worked on. The subtleties in parsing different kinds of natural language input are discussed in the next chapter. The kinds of input that the parser is capable of handling now are indexed by the appropriate VMS command. The various sentences parsed now are:

PRINT

1. Print
2. hardcopy
3. printout

4. print-out
5. to print a file
6. to make a hardcopy of a file
7. to make hardcopies of a file
8. i want five copies of a file printed out
9. to print something after eight o´clock
10. print six copies after five
11. print on the laser printer after five copies six
12. print on the usual printer
13. print on the normal printer
14. print on the smaller printer
15. print on the small printer
16. after nine print five copies on the normal printer
17. after nine ,five copies, print on the narrow printer
18. print some files excluding others
19. print certain files except others
20. to get a file printed and a message when it is done
21. print a file confirming it

DIRECTORY

1. directory
2. ls
3. catalog
4. files
5. area
6. sub-directory
7. directories
8. to see the files
9. can you show me my files
10. can you show me my area
11. list my files
12. files in my area
13. show me files created since yesterday
14. the files modified today
15. what are the files altered since yesterday
16. the list of fiiles changed yesterday
17. what files were created yesterday

18. files with date created
19. when were the files created
20. files created since yesterday
21. files with their protection
22. files with their owners
23. who owns what files
24. please show me files with their access rights and owner information
25. catalog with access rights
26. I want to see some files excluding some others
27. to see some files except others

COPY

1. copy
2. spip
3. mpip
4. pip
5. cp
6. transfer
7. to copy a file
8. to duplicate a file
9. make a copy
10. i want to make a copy
11. how can i make a copy
12. how can i transfer this file
13. duplicate a set of files
14. copy some files and confirm it
15. copy files modified since yesterday
16. copy a set of files and confirm it

DELETE

1. delete
2. to delete a file
3. erase
4. to eliminate
5. remove
6. re

7. how do i eliminate a job from the print queue
8. how to remove the job from the narrow queue
9. to delete a job already batched
10. delete an entry
11. how do i delete a batched job
12. how do i delete a job from the usual printer queue
13. eliminate a job sent to the batch queue

CREATE

1. create
2. make file
3. generate a file
4. create a directory
5. creating a file
6. how do i create a file
7. how is generating a file done
8. how is a directory split
9. to make a file exist
10. how to make sub directories
11. i want to generate a catalog
12. generating a catalog

PURGE

1. to purge
2. purge
3. purge some files excluding others
4. purge versions before a certain date
5. purge certain versions excluding others
6. to purge some files and confirm it
7. to get a message after certain files are purged
8. after purging is done to get a message

SHOW QUOTA

1. to see space available
2. show space

3. display space
4. display unused space
5. show empty space available for use
6. how do i see how much empty space is available
7. how big is the unused space
8. how large is the space available
9. display what space is available for use

SHOW PROCESS

1. to display processes
2. display proc
3. to see what processes are running
4. what processes are running on the system now
5. to show processes
6. show proc

SHOW CPU

1. to display cpu usage
2. show cpu
3. display cpu
4. what is the cpu usage
5. how is the cpu being used
6. what is happening in the cpu
7. how is the cpu occupied now

TYPE

1. type
2. print on the screen
3. How do i see a file on the screen

5.2 Output Interface

PLUM parser, because of the user-defined frames with slots and slot-fillers it uses and a dictionary which defines words in terms of these frames, allows *one to build in as much of syntax and semantics as needed for the particular natural language understanding task at hand*. Here, the level of understanding of the natural language input is brought out by the following parse of a query:

WELCOME TO THE VMS CONSULTANT

Type bye to exit the VMS CONSULTANT

- -

VMS CONSULTANT>
I want five copies of a file printed
on the laser printer after five

The user wants help about the command PRINT

<Actual word used- printed>

Command Qualifiers

Qualifier- COPIES
  Value specified by user- <5>

- -

Qualifier- QUEUE
  Value specified by user- <SYS$LASER>

- -

Qualifier- AFTER

Value specified by user- <5>

- -

VMS CONSULTANT>

The level of understanding achieved is to breaking down the query into a hierarchy of *commands and command qualifiers*. The structure:

Command -

Command Qualifier-1 —

Command-qualifier-value-1 —

Command Qualifier-2 —

Command-qualifier-value-2 —

etc...

With the above structure it is possible to translate the natural language input into a command language construct call itself. Thus the level of understanding achieved is enough to serve as *a natural language interface to the VMS operating system itself*. However, the present mechanism of the Example Help system dictates that the parse passed on to it is of a much simpler nature. Hence only the command is passed on. From then on, the Example help system does the needed processing of the query.

6.0 NATURAL LANGUAGE ISSUES

6.1 What has been achieved

The approach taken is that of exploiting the semantics of the command language structure. This structure is translated directly into corresponding frames. This technique always seems to work in *restricted domains and a lower level of sophistication in input understanding*. This has been proven in more than one domain in which PLUM operates now. To elaborate further, consider the parse tree structure that PLUM produces in this interface. It is of the form

Command -

Command qualifier1 -

Command value1 -

Command qualifier2 -

Command value2 -

The advantages of this approach transcend the utility of the interface to the Example Help system. The structure of the command language found in the parse can be used to make this interface a *veritable interface for the VMS command language itself*. Recalling the output trace from the previous section we see that

I want five copies of a file printed
on the laser printer after five

gets translated to something similar to

PRINT ?/COPIES-5/QUEUE-SYS$LASER/AFTER-5

This means that if the facility for recognising file names is available with the parser, it could be used as a natural language interface to the operating system itself.

Another domain in which the above observation was found to be extremely valid was the interface built to a purchase order processing database in the POISE office automation system. There also, the semantics of the relational database structure was exploited in the interface. The database consisted of a number of attribute-value pairs and the objective of any query was to find the value of an attribute given a number of matching attribute value pairs.

e.g.**Who bought the steel writing tables**

gets translated to

Wanted-Attribute - Purchaser

Matching attribute-value pairs

Material - steel

item - writing-table

Equivalent commands from other operating systems are recognized and adding new ones is just a matter of finding equivalent ones in VMS and adding entries to the dictionary.

e.g. For the command **DIRECTORY** in VMS we can recognise **LS** from UNIX and **CATALOG** from the CYBER NOS/VS operating systems.

The phrasal definition facility allows us to equate commands with naturally occurring terms such as "Sub-directory".

e.g. The user wants to see a sub-directory and could type in

sub-directory or

I want to see one of the sub-directories

Here these are equivalent to **DIRECTORY** in VMS.

Incomplete utterances that occur in a pseudo-discourse-like setting are also handled. The following is such a piece:

I want to print a file on the laser printer

To print it on the usual printer?

To print it on the narrow printer?

The above is parsed without recourse to any discourse processing, because of the current nature of the interface. Now it just homes in on the key pointers to command language constructs.

Semantic variations are very well handled by the existing synonym facility.

e.g. **What are the files altered since yesterday**

The list of files changed yesterday

the files modified yesterday

are all handled alike, since "altered", "changed" are all synonyms of the VMS command qualifier "Modified".

The biggest advantage of the current approach is the variety of single utterances that the interface can handle. Because of the unsophisticated nature of the understanding that is done of the input, single words as well as complete sentences could be understood. Here is a classic case where syntax has been minimally used, while a translation of the semantics of the task on hand into appropriate frames has paid off handsomely.

6.2 Future directions

This exercise is just a start in the kinds of understanding that can be accomplished of more sophisticated natural language input. Currently morphological variants of the keywords that key in to the commands have been put in the dictionary explicitly. *A morphology package would aid in extending the vocabulary very much.*

The prototypes or the frames could be rationalized by forming a proper taxonomy of VMS commands. Drawing up a proper structure for all the various commands and command qualifiers would enable identify some inconsistencies in the VMS command structure itself.

e.g. **PRINT TEST.LGP/QUEUE-SYS$LASER**
DELETE/ENTRY-2030 SYS$LASER

In the above example getting a file printed on a particular printer queue involves invoking the command with the "QUEUE=" qualifier but deleting the entry does not. This has implications for the natural language input parsing because of the reliance of the frames on the command language structure. Identifying such inconsistencies and making exceptions only in such cases improves the generality of the parser.

There could be ambiguity in mapping commands from another operating system into VMS commands.

e.g. **MPIP** in the CP/M operating system does the job
of many commands in VMS like COPY, TYPE etc..

What does one do in such cases?. One has to have much more sophistication and discrimination in the parsing of such cases.

Simple dialogues are in the process of implementation at the time of this report. This involves a totally different approach with a different set of prototypes and dictionary definitions. This suggests the use of two modes of the parser operation.

a. Single utterance mode
b. Conversational mode

In the single input mode PLUM requires only a single utterance and does not "remember" anything when it starts parsing the next input utterance. This is because

the parsing function cleans up the internal memory before it starts on a new utterance. However, for the kind of processing that is needed for the short dialogues, the mechanism would work like this:

a. PLUM parses the first utterance and instead of cleaning the *EVENT-MEM* removes it to the *SHORT-TERM-MEM*.

b. The second utterance is parsed. At the end of the parse the demons failing on the current input search selectively in the *SHORT-TERM-MEM*.

c. The *SHORT-TERM-MEM* is a queue with a limited number of place-holders. For example, it could be just three. The fourth utterance once parsed cleans out the first utterance from the *SHORT-TERM-MEM*. Thus reference could be made anaphorically or elliptically only to a certain "depth" in the dialogue. This would ensure that unnecessary conclusions would not be made about references to objects that have been featured very long ago. This provides a very efficient context mechanism. More of the things found in [GROS77] could be helpful.

Another feature that could really be worth pursuing, is the shareability of the knowledge that is built up with the other component of the total Example Help system. Currently there is one way communication between PLUM and any other system to which it is an interface. If the other system is powerful enough, it could augment the confidence with which PLUM instantiates frames. There could be a field in the prediction prototype which is filled with a confirmatory "yes" or "no" by the other system. This could come in handy in deciding between competing interpretations.

7.0 APPENDICES

7.1 Implementation Notes

Currently the interface runs on the version of PLUM that is available on VAX4::nlp$disk:[narasim.master]. This is in CLisp. The basic PLUM mechanism is loaded by executing (eval-file 'loadplum). The domain dependent files are available by executing (eval-file 'domain). The interface is made easier with typing in (plum). This puts the user in a mode which prompts for the input thus:

CLisp:
(plum)

WELCOME TO THE VMS CONSULTANT

Type bye to exit the VMS CONSULTANT

- -

VMS CONSULTANT>
I want five copies of a file printed on the laser printer after five

---

The user wants help about the command PRINT

<Actual word used- printed>

Command Qualifiers

Qualifier- COPIES
  Value specified by user- <5>

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Qualifier-  QUEUE
  Value specified by user- <SYS$LASER>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Qualifier-  AFTER
  Value specified by user- <5>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

VMS CONSULTANT>

7.2. Prototypes

```
;*****************************************************************
; PROTOTYPE PREDICTIONS
;*****************************************************************

;*****************************************************************
; This is the top-most-level frame attached to *EVENT-MEM*. Accepts in its *
; available slot any frame instantiated for a command                      *
;*****************************************************************
(create-pred type (help-frame)
             comment (This is triggered by the pseudo-word *start*)
             concept-frame ( frame-name      = HELP-FRAME
                             help-frame-slot = (nil))
             control-structure ((expect help-frame-slot in future print-frame
                                        return print-frame)
                                (expect help-frame-slot in future
                                  directory-frame return directory-frame)
                                (expect help-frame-slot in future
                                  copy-frame return copy-frame)
                                (expect help-frame-slot in future
                                  delete-frame return delete-frame)
                                (expect help-frame-slot in future
                                  create-frame return create-frame)
                                (expect help-frame-slot in future
                                  purge-frame return purge-frame)
                                (expect help-frame-slot in future
                                  show-frame return show-frame)
                                (expect help-frame-slot in future
                                  type1-frame return type1-frame)
                                (expect help-frame-slot in future
                                  type2-frame return type2-frame))
             required-slots (help-frame-slot)
             memory-level (*EVENT-MEM*))

;*****************************************************************
; This frame is for the command PRINT. PRINT could have qualifier-values as
; qualifiers. e.g COPIES=5 or just qualifiers alone e.g. PRINT/CONFIRM
;*****************************************************************
(create-pred type (print-frame)
             comment (This is triggered by a print word)
             concept-frame ( frame-name     = PRINT
                             main-word      = (print-verb or print-noun)
                             qualifiers     = (nil))
             control-structure ((expect main-word in last word
                                        part-of-speech return word)
                                (expect qualifiers in past qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in future qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in past qualifier-alone1
                                        return qualifier-alone1)
                                (expect qualifiers in future qualifier-alone1
                                        return qualifier-alone1))
             required-slots (main-word)
             memory-level (*CONCEPT-MEM*))

;*****************************************************************
; The frame for the command Directory                              *
;*****************************************************************
(create-pred type (directory-frame)
             comment (This is triggered by a directory word)
             concept-frame ( frame-name     = DIRECTORY
                             main-word = (directory-noun)
                             qualifiers     = (nil))
             control-structure ((expect main-word in last word
                                        part-of-speech return word)
                                (expect qualifiers in past qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in future qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in past qualifier-alone1
                                        return qualifier-alone1)
                                (expect qualifiers in future qualifier-alone1
                                        return qualifier-alone1))
             required-slots (main-word)
             memory-level (*CONCEPT-MEM*))

;*****************************************************************
; The frame for the command DELETE                                 *
;*****************************************************************
(create-pred type (delete-frame)
```

```
            comment (This is triggered by a delete word)
            concept-frame ( frame-name    = DELETE
                            main-word = (delete-verb or delete-noun)
                            qualifiers    = (nil))
            control-structure ((expect main-word in last word
                                       part-of-speech return word)
                               (expect qualifiers in past qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in future qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in past qualifier-alone1
                                       return qualifier-alone1)
                               (expect qualifiers in future qualifier-alone1
                                       return qualifier-alone1))
            required-slots (main-word)
            memory-level (*CONCEPT-MEM*))

;*****************************************************************************
; The frame for the command COPY                                             *
;*****************************************************************************
(create-pred type (copy-frame)
            comment (This is triggered by a copy word)
            concept-frame ( frame-name    = COPY
                            main-word = (copy-word)
                            qualifiers    = (nil))
            control-structure ((expect main-word in last word
                                       part-of-speech return word)
                               (expect qualifiers in past qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in future qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in past qualifier-alone1
                                       return qualifier-alone1)
                               (expect qualifiers in future qualifier-alone1
                                       return qualifier-alone1))
            required-slots (main-word)
            memory-level (*CONCEPT-MEM*))

;*****************************************************************************
; The frame for the command  CREATE                                          *
;*****************************************************************************
(create-pred type (create-frame)
            comment (This is triggered by a create word)
            concept-frame ( frame-name    = CREATE
                            main-word = (create-verb or create-noun)
                            qualifiers    = (nil))
            control-structure ((expect main-word in last word
                                       part-of-speech return word)
                               (expect qualifiers in past qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in future qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in past qualifier-alone1
                                       return qualifier-alone1)
                               (expect qualifiers in future qualifier-alone1
                                       return qualifier-alone1))
            required-slots (main-word)
            memory-level (*CONCEPT-MEM*))

;*****************************************************************************
; The frame for the command PURGE                                            *
;*****************************************************************************
(create-pred type (purge-frame)
            comment (This is triggered by a directory word)
            concept-frame ( frame-name    = PURGE
                            main-word = (purge-verb)
                            qualifiers    = (nil))
            control-structure ((expect main-word in last word
                                       part-of-speech return word)
                               (expect qualifiers in past qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in future qual1-val-pair
                                       return qual1-val-pair)
                               (expect qualifiers in past qualifier-alone1
                                       return qualifier-alone1)
                               (expect qualifiers in future qualifier-alone1
                                       return qualifier-alone1))
            required-slots (main-word)
            memory-level (*CONCEPT-MEM*))

;*****************************************************************************
; The frame for the command SHOW                                             *
;*****************************************************************************
```

```
(create-pred type (show-frame)
             comment (This is triggered by a show word)
             concept-frame ( frame-name    = SHOW
                             main-word = (show-verb)
                             qualifiers    = (nil))
             control-structure ((expect main-word in last word
                                        part-of-speech return word)
                                (expect qualifiers in past qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in future qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in past qualifier-alone1
                                        return qualifier-alone1)
                                (expect qualifiers in future qualifier-alone1
                                        return qualifier-alone1))
             required-slots (main-word qualifiers)
             memory-level (*CONCEPT-MEM*))

;*************************************************************************
; The frame for the command TYPE. This is instantiated only when the user types *
; in TYPE in the input                                                    *
;*************************************************************************
(create-pred type (type1-frame)
             comment (This is triggered by a type word)
             concept-frame ( frame-name    = TYPE
                             main-word = (type-verb)
                             qualifiers    = (nil))
             control-structure ((expect main-word in last word part-of-speech
                                        return word)
                                (expect qualifiers in past qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in future qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in past qualifier-alone1
                                        return qualifier-alone1)
                                (expect qualifiers in future qualifier-alone1
                                        return qualifier-alone1))
             required-slots (main-word)
             memory-level (*CONCEPT-MEM*))

 *************************************************************************
  Alternative Frame for the command TYPE. When the user types in 'Show on the  *
; screen' or 'Print on the screen' etc.., this is instantiated             *
;*************************************************************************
(create-pred type (type2-frame)
             comment (This is triggered by a type word)
             concept-frame ( frame-name    = TYPE
                             main-word = (print-verb or show-verb)
                             qualifiers    = (nil))
             control-structure ((expect main-word in past type-compound
                                        main-word part-of-speech return
                                        type-compound main-word)
                                (expect main-word in future type-compound
                                        main-word part-of-speech return
                                        type-compound main-word)
                                (expect qualifiers in past qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in future qual1-val-pair
                                        return qual1-val-pair)
                                (expect qualifiers in past qualifier-alone1
                                        return qualifier-alone1)
                                (expect qualifiers in future qualifier-alone1
                                        return qualifier-alone1))
             required-slots (main-word)
             memory-level (*CONCEPT-MEM*))

;*************************************************************************
; This is the frame necessary for separating out 'print on the screen' from    *
; 'print'                                                                 *
;*************************************************************************
(create-pred type (type-compound)
             comment (This is triggered by the psuedo-word *START*)
             concept-frame ( main-word         = (show-verb or print-verb)
                             display-medium    = (display-medium))
             control-structure ((expect main-word in future word
                                        part-of-speech return word)
                                (expect display-medium in future word
                                        part-of-speech return word))
             required-slots (main-word display-medium)
             memory-level (*CONCEPT-MEM*))

; *************************************************************************
; This is an inelegant solution to the problem of looking for number of       *
```

```
; attribute-value pairs. This looks like this because the multiple slot-filler   *
; facility is not working properly currently                                      *
; ********************************************************************************
'create-pred type (qual1-val-pair)
             comment (This is triggered by the pseudo-word *start*)
             concept-frame ( frame-name        = QUAL1-VAL-PAIR
                             command-qualifier = (command-qualifier-word)
                             com-qual-value    = (com-qual-value-word)
                             next-qual-val-pai = (nil))
             control-structure ((expect command-qualifier in future word
                                        part-of-speech return word)
                                (expect com-qual-value in future word
                                        part-of-speech return word)
                                (expect next-qual-val-pair in future
                                        qual2-val-pair return qual2-val-pair))
             required-slots (command-qualifier com-qual-value)
             predicts (qual2-val-pair-prediction)
             memory-level (*CONCEPT-MEM*))

;*******************************************************************************
; This is predicted by qual1-val-pair.This predicts a qual1-val-pair and so the  *
; chain is continued                                                              *
;*******************************************************************************
(create-pred type (qual2-val-pair)
             comment (This is triggered by a qual1-val-pair-prediction)
             concept-frame ( frame-name        = QUAL2-VAL-PAIR
                             command-qualifier = (command-qualifier-word)
                             com-qual-value    = (com-qual-value-word)
                             next-qual-val-pair = (nil))
             control-structure ((expect command-qualifier in future word
                                        part-of-speech return word)
                                (expect com-qual-value in future word
                                        part-of-speech return word)
                                (expect next-qual-val-pair in future
                                        qual1-val-pair return qual1-val-pair))
             required-slots (command-qualifier com-qual-value)
             predicts (qual1-val-pair-prediction)
             memory-level (*CONCEPT-MEM*))

;*******************************************************************************
  This looks for just one qualifier without a value                              *
;*******************************************************************************
(create-pred type (qualifier-alone1)
             comment (This is triggered by the pseudo-word *start*)
             concept-frame ( frame-name        = QUALIFIER-ALONE1
                             command-qualifier = (command-qualifier-lone-word)
                             com-qual-value    = (com-qual-value-word)
                             next-qual-val-pair = (nil))
             control-structure ((expect command-qualifier in future word
                                        part-of-speech return word)
                                (expect com-qual-value in future word
                                        part-of-speech return word)
                                (expect next-qual-val-pair in future
                                    qualifier-alone2 return qualifier-alone2))
             required-slots (command-qualifier)
             predicts (qualifier-alone2-prediction)
             memory-level (*CONCEPT-MEM*))

;*******************************************************************************
; This is part 2 of the chain for the qualifier alone frames                     *
;*******************************************************************************
(create-pred type (qualifier-alone2)
             comment (This is triggered by the pseudo-word *start*)
             concept-frame ( frame-name        = QUALIFIER-ALONE2
                             command-qualifier = (command-qualifier-lone-word)
                             com-qual-value    = (com-qual-value-word)
                             next-qual-val-pair = (nil))
             control-structure ((expect command-qualifier in future word
                                        part-of-speech return word)
                                (expect com-qual-value in future word
                                        part-of-speech return word)
                                (expect next-qual-val-pair in future
                                    qualifier-alone1 return qualifier-alone1))
             required-slots (command-qualifier)
             predicts (qualifier-alone1-prediction)
             memory-level (*CONCEPT-MEM*))
```

7.3. Dictionary

```
;*****************************************************************************
  Dictionary definitions
;*****************************************************************************

; ********** Pseudo- word **********************************
(defword *start* (pseudo-word) (help-frame-prediction
                                qual1-val-pair-prediction
                                type-compound-prediction
                                qualifier-alone1-prediction) nil)


; ********** Commands      **********************************

; - - - - - - - - - PRINT - - - - - - - - - - - - - - - - - -
(defword print (print-verb) (print-frame-prediction type2-frame-prediction) nil)
(syn print (printed prints))

(defword hardcopy (print-noun) (print-frame-prediction) nil)
(syn hardcopy (print-out printout hardcopies))

; - - - - - - - - - DIRECTORY - - - - - - - - - - - - - - - - -
(defword directory (directory-noun) (directory-frame-prediction) (/DIRECTORY))
(syn directory (files dir catalog  area sub-directory list-of-files my-stuff
                ls sub-directories directories))

(phrase ((sub directory) (list of files) (my stuff)(sub directories)))

; - - - - - - - - - COPY - - - - - - - - - - - - - - - - - - -
(defword copy (copy-word) (copy-frame-prediction) nil)
(syn copy (cp transfer duplicate pip eplp mpip))

; - - - - - - - - - DELETE - - - - - - - - - - - - - - - - - -
(defword delete (delete-verb) (delete-frame-prediction) nil)
(syn delete (erase remove eliminate get-rid-of))
(phrase ((get rid of)))

; - - - - - - - - - CREATE - - - - - - - - - - - - - - - - - -
 defword create (create-verb) (create-frame-prediction) nil)
.syn create (creating created generate generating generated make making
             split splitting makes))

; - - - - - - - - - PURGE - - - - - - - - - - - - - - - - - - -
(defword purge (purge-verb) (purge-frame-prediction) nil)
(phrase ((clean up)))
(syn purge (clean-up))

; - - - - - - - - - SHOW - - - - - - - - - - - - - - - - - - -
(defword show (show-verb) (show-frame-prediction type2-frame-prediction) nil)
(syn show (display what how))

; - - - - - - - - - TYPE - - - - - - - - - - - - - - - - - - -
(defword type (type-verb) (type1-frame-prediction) nil)


; ********** Display-medium **********************************
(defword screen (display-medium) nil nil)
(syn screen (vdu monitor visual-display-unit))

; ********** Enhancer Words **********************************
(defword file (file-noun) nil nil)

; ********** Command qualifier words **********************************
(defword after (command-qualifier-word) nil (/AFTER))

(defword copies (command-qualifier-word) nil (/COPIES))
(syn copies (print-outs))

(defword queue (command-qualifier-word) nil (/QUEUE))
(syn queue (printer))

(defword since (command-qualifier-lone-word) nil (/SINCE))

(defword before (command-qualifier-lone-word) nil (/BEFORE))

 defword confirm (command-qualifier-lone-word) nil (/CONFIRM))
(syn confirm (message))

(defword owner (command-qualifier-lone-word) nil (/OWNER))
(syn owner (owns owned owners))
```

```
(defword created (command-qualifier-lone-word) nil (/CREATED))
(syn created (date-of-creation date-created))
(phrase ((date of creation) (date created)))

 defword protection (command-qualifier-lone-word) nil (/PROTECTION))
(syn protection (security access-rights access-info access-information))
(phrase ((access rights) (access info) (access information)))

(defword modified (command-qualifier-lone-word) nil (/MODIFIED))
(syn modified (changed altered))

(defword exclude (command-qualifier-lone-word) nil (/EXCLUDE))
(syn exclude (excluding excepting except))

(defword entry (command-qualifier-lone-word) nil (/ENTRY))
(syn entry (job))

(defword quota (command-qualifier-lone-word) nil (/QUOTA))
(phrase ((file space) (empty space) (unused space)))
(syn quota (file-space space empty-space unused-space))

(defword process (command-qualifier-lone-word) nil (/PROCESS))
(syn process (processes proc))

(defword cpu (command-qualifier-lone-word) nil (/CPU))
(phrase ((central processing unit)))
(syn cpu (central-processing-unit))


; ********** Command qualifier value words **********
(defword laser (com-qual-value-word) nil (SYS$LASER))

(defword batch (com-qual-value-word) nil (SYS$BATCH))
(syn batch (batched))

(defword normal (com-qual-value-word) nil (SYS$PRINT))
(syn normal (usual ordinary))

(defword narrow (com-qual-value-word) nil (SYS$NARROW))
(syn narrow (small smaller))

 defword one (com-qual-value-word) nil (1))

(defword two (com-qual-value-word) nil (2))

(defword three (com-qual-value-word) nil (3))

(defword four (com-qual-value-word) nil (4))

(defword five (com-qual-value-word) nil (5))

(defword six (com-qual-value-word) nil (6))

(defword seven (com-qual-value-word) nil (7))

(defword eight (com-qual-value-word) nil (8))

(defword nine (com-qual-value-word) nil (9))

(defword ten (com-qual-value-word) nil (10))

(defword tomorrow (com-qual-value-word) nil (TOMORROW))

(defword yesterday (com-qual-value-word) nil (YESTERDAY))

(defword today (com-qual-value-word) nil (TODAY))
```

8.0 REFERENCES

[GROS77] The representation and use of Focus in Dialogue Understanding
Barbara Grosz, SRI Technical Report 151. 1977

[LEHN85] The PLUM User's Manual
Wendy G. Lehnert
University Of Massachusetts at Amherst. 1985

[RISS84] Explaining and arguing with examples
Edwina L. Rissland. et al
University of Massachusetts at Amherst. 1984

[WILE82] Talking to UNIX in English :An Overviw of UC
Robert Wilensky
In Proceedings of AAAI-82, Pittsburgh, PA 1982.

VISPLUM: A PARSER FOR QUERYING HIGH-LEVEL VISION SYSTEM

Bruce Draper

1.0 INTRODUCTION

(VISPLUM:) A parser for querying a high-level vision system VISPLUM is intended to serve as the front end for a natural language interface to a high-level image interpretation system. This interface is to provide the vision system designer a tool to aid his/her debugging. In particular, it will allow him/her to inquire about the system's beliefs and assertions at any point in the computation. Moreover, it will allow this to be done without the programmer having to know the details of the data structures and code of the system. This ability grows more important as schema-based vision systems expand, requiring a large number of complex schemas to be written. It will allow the authors of these schemas to be involved not with the mundane details of the system, but with the problems of image interpretation.

Any natural language front end is designed around the particulars of its domain. This made the initial planning for VISPLUM difficult, since the schema-based vision system that it was meant to serve in a state of flux, and it was not at all clear how real users would choose to actually use the system. We were therefore forced to make assumptions about both the user queries to and the optimal output from his system. These assumptions underlie the basic structure of visplum. Briefly, they are:
(1) the users will want to make their communications as telegraphic as possible. The objects of the queries will generally by very simple, and very specific. The user will not want to type much, and this system's response is more likely to be a number, a graph, or a highlighted piece of an image then an english sentence. Thus, the system should be designed to handle "what house" or "what color is region 12", and should

be less worried about "how many bushes are next to the large red chimney with a[1] of 12 (although it should be able to handle this case, too).

2) The vision system using VISPLUM is a schema based system that attempts to give semantic labels to objects in the image. As such, its data base consists of schemas and schema instances, and connections between schema instances (the connections from schema to schema instance have no value other then speeding up access, and are therefore not considered). Furthermore, schema instances have features that correspond to characteristics of the pixel groups (regions) to which they are bound, such as color, texture, vepua, etc. The features of a schema are those characteristics the would constitute an ideal match in the image; Features are also possessed by the connections between schema instances (in the current system: goals, hypothesis´), as they transmit data from requested to requesting instance.

These assumptions guide VISPLUM throughout. The first assumption motivates how sentences are broken up and initially analyzed; the second suggests what concept frames must be built and what the final representation should be. Therefore our discussion of VISPLUM will be similarly divided. The first section describes the algorithm, syntax, and semantics used to parse the sentence. The second explains the knowledge representation scheme and justifies the selection of defined words.

---

[1] vepua [vertical-edges-per-unit-area"]

2.0 THE PARSING APPROACH

The working assumption of VISPLUM, as stated above, is that the queries to the system are likely to be fragmentary. "What color?" and "Next to what?" should be accepted (and parsed) as valid input, even though they are not, grammatically speaking, sentences. Although this does not exempt them from the rules of english grammar - (I am not claiming that standard grammars are unable to handle interrogative phrases) - it does suggest that these sentences are better handled by a simpler, less cumbersome mechanism. This view is furthered when one considers the domain of these queries. The words "bush", "goal", and "color" are all syntactic nouns, yet they represent the three fundamentally different types of objects that the system knows about. Going the other way, verbs are one of the most important of all syntactic categories, but in this domain there are only two verbs - to be and to have - and these appear only in the present tense. They more closely resemble special cases then a unique syntactic category. The result of this line of thinking was a new, abbreviated syntax based on the semantic category of words relative to the visual data base. The primitives of this grammar are object, feature type, feature value, interrogative, relationship, schema-connector, and the (minor) holdovers from standard grammars, article and verb. While the new system has roughly the same number of basic elements as standard grammars, it is simpler for two reasons. First, there is a clear conceptual mapping from category to objects in the universe (i.e. the data base). This simplifies the job of expanding and/or altering the parser as the vision system changes (a constant concern). Second, it reduces the number of

intermediate structures that need to built between the initial, lexical stage and the final representation. This is a by-product of the simpler mapping; categories that do not directly correspond to types of objects in the data base will require one more layer of processing then those that do. As a result of the new set of primitives, much (but not all) of the syntactic layer of phrases has been done away with.

It could be argued that what has been done is that syntax has been done away with entirely, and that what is happening is parsing by semantic category. This was not my original intent. Moreover, I don't believe this view would be completely accurate. There is still a good deal of embedded syntax, most of it in the form of mandatory word orderings. "What color" is a coherent question; "Color what" will confuse VISPLUM and cause it to pass an empty structure to whoever called it. A more apt description would be that the syntactic and semantic processing has been fused (although not completely).

Until the knowledge representation system has been explained, I cannot give a precise explanation of the parsing algorithm. This information is better culled from the (well documented) code anyway. However, I will provide some examples of how it runs here to show how the new categories are used. VISPLUM assumes that any line typed by the user is, in fact, a query (a statement will be interpreted as a predicate to be corroborated). Furthermore, it assumes that the user is always referring to some "object", where an object is a schema, schema instance, goal or hypothesis. Given the query "What color?", it assumes the user is asking about some object. It recognizes "What" as an interrogative asking about this object that

may or may not take a feature value. When a feature value is given (color), it assumes the user wants to know the value of the feature color for the object. Since no more information is given, it passes this unfinished structure up to whoever called it, presumably some kind of dialogue interpreter, who is left with the task of finding out to what object this query corresponds. When given a more complicated query - "What color is the large bush next to the chimney" - it must do more work. It again recognizes the request for the value of the color feature. This time, it continues, but gets just a little information from "is" and "the". When it hits "large", (a feature value) it realizes this is a description of the value of the size feature of the (still mysterious) object. "Bush" it recognizes as being the semantic label of the object. This is where things get a little tricky. VISPLUM recognizes that there are certain points in a query where a "break" might occur. That is, at these points, a relationship is being described between objects, and everything after that refers to a new entity (note: relationships do not always signify break points. Consider "what region is next to region 24", while remembering that region is a feature of an object). It therefore establishes the geometric relationship between the two objects (in this case "next to"), and interprets the rest of the query with regard to the new object. What is left to the rest of the system is for it to identify (in the data base) objects matching the "objects" thus created, and for the color value of these objects to be ascertained.

3.0 THE KNOWLEDGE REPRESENTATION SCHEME

The objects defined in the dictionary are those that appear in the IM image data base, and those which were considered desirable by (schema based) vision system creator Terry Weymouth "for academic perversity or general interest". The feature types defined are a subset of those used in Terry's system, plus certain other features which do not correlate very well to measured values but which naive users would want to use (i.e. color). In general, the values of any mesaurement on the pixels should be accessible through a defined feature value. This dictionary contains only a subset due to the evolving nature of the set of such measurements. Non numeric feature values are given for use with the naive feature types defined (e.g. blue, large). Finally, there are only two interrogatives completely defined, "what" and "where". This is clearly an inadequacy; "how", "why" and possibly "who" should also be defined. The trouble, not yet overcome, with these words is that they can be used in a large number of queries which have no corresponding meaning in the data base. "Why is the house blue" and "how is the sky blue" cannot be sensibly reprented by our data structure. VISPLUM needs to be extended so that such queries are rejected (parsed into empty structures), while other, sensible queries using the same wh-words are accepted. VISPLUM's domain gives it some interesting representational properties. The most striking of these is the total absence of actions. For the moment, the system only considers inquiries into a static data base. In the future it may be expanded to also entail the construction and destruction of nodes within the data base, and/or the execution of steps in the schema system. Even then,

however, there will still be a total absence of sentences involving typical actions. The conceptual analysis of a question in this system involves only the representation of the objects involved, and their static relations to each other. The point of inquiry is an unknown or unverified (e.g. predicate) relation between the objects. The top level representation of a question, therefore, will be a list of "relationship" structures. The first relations are those that conceptually must hold in order for the question to be sensical; the last is the relation the user asked about. Thus the question "Is that a shutter below the chimney?" would be represented as the following list of relationships:

```
Is there a perceived object corresponding to the semantic
   label chimney?
Is there something (i.e. a segmented region) below the
   chimney-object?
```

Does that region correspond to the semantic label shutter? Now comes the confusing jump. Objects actually are relations between regions (and their associated objects) and semantic labelings (with their associated modifiers, et. al.). Hence the relations previously mentioned are just the representations of objects. Let me describe what this representation looks like by example: The object "shutter" in the previous example would be described as:

```
(Object-2 (class instance)
          (type shutter)
          (token ?)
          (number single)
          (features nil)
          (relationships (below Object-1)))
```

In this representation, every semantic object has six fields: class, type, token, number, features, and relationships. The class of an object is prototype, instance, hypothesis or goal. The type is the label for the specific linguistic object (e.g. shutter, door), while the token is the label for the specific observed object (e.g. a collection of regions and all the semantic labelings and subsequent knowledge associated therewith). The features are, in effect, constraints upon the relation between the linguistic and observed objects (i.e. No observed object can be associated with a given semantic label unless it has the aforesaid features, which can be viewed as a predicate function on the observed object). In reality, they correspond to measured quantities on the pixels of the observed object. The relationships are also a predicate function, this time one that must be true for pairs (or triplets, or whatever) of observed objects. A token (observed object) can't be equated with the linguistic object shutter unless it has the right relationship with the token (observed object) that we equate with chimney.

As a result, the full representation for the question "Is that a shutter below the chimney?" is:

```
(Object-1 (class instance)
          (type chimney)
          (token ?)
          (number single)
          (features nil)
          (relationships nil))

(Object-2 (class instance)
          (type shutter)
          (token ?)
          (number single)
          (features nil)
```

```
(relationships (below Object-1))))
```

What makes a statement a question is the presence of unknown slots; A predicate, therefore, is a relation with just the token slot unknown (as opposed to unfilled). The question "What is below the chimney" becomes:

```
(Object-3 (class instance)
          (type chimney)
          (token ?)
          (number single)
          (features nil)
          (relationships nil))

(Object-4 (class instance)
          (type ?)
          (token ?)
          (number single)
          (features nil)
          (relationships (below Object-3)))
```

The intent of this knowledge representation is to keep the interface between the natural language and visual systems as simple as possible. This seemed the most important criterion (other then being able to represent all queries that are meaningfull to the sytem), since if the experienced user has to second guess what the visual analogue of his words will be, VISPLUM would quickly grow to be more of a hindrance than an aid.

What follows are the prediction prototypes and the dictionary of VISPLUM, along with the test transcripts on which it (successfully) runs. First, however, some concluding comments are in order. VISPLUM is centered around two basic ideas, each of which I believe is sound. First is the idea that most queries will be very simple, and should be treated simply. Above the lexical level the only structures that

should be built are those that correspond to properties of the object (class, features, etc.) and the "Object" itself. This is not only computationally efficient, it will allow the user to quickly get a feel for how the system works, and thus he/she will quickly be able to make his/her communication more telegraphic. More importantly, the user will not be confused about how to word a question to get the piece of information desired, thus making VISPLUM a tool instead of a burden. Second is the idea that the interface between the natural language and vision systems must be kept simple. This is more then just a consideration of style. The schema based systems under developement here changes daily. An interface that was difficult to change would quickly cause the system to be thrown away. Even were this not a consideration, a complex interface would not work. Vision systems typically encode a large amount of semantic information in a manner that is not explicit in the representation. Natural Language systems often encode their semantic information in much the same way. The interface between two such systems must bring information across from one to the other in a way that is either obvious or transparent to the user; otherwise he/she will spend more time worrying about VISPLUM than the problem he/she is trying to debug. Given the domain, transparency is impossible, and I can't imagine a more complicated system not confusing the user. If this is to be a debugging tool, simplicity is the only way.

Unfortunately, these ideas constrain VISPLUM a great deal, especially the first one. A lot of work has gone into making it parse complex sentencee even though it lacks the standard syntactic structures. This has been partially successfull, but there are still sentences it should handle that it doesn't. It can be further expanded, but it already parses complicated sentences in an ugly manner. Still, it is functional, and simply processes most of its input. However, if the domain were to be expanded to allow the user to alter the data base or engage in hypothetical reasoning, VISPLUM would crumble. At this point, something more closely resembling a conventional natural language analyzer would be needed. My intuition is that this is the limit as to how far VISPLUM could be stretched. To handle hypothetical sentences, you must start over. And this is the primary disadvantage to VISPLUM.

```
;*****************************************************************
;
; Type
;*****************************************************************
;
;
; Type is central prediction prototype. It preserves the semantic label
; (object:) and number from the object-noun for the Object structure.
; It this type of label (bush, tree, etc.) that most structures are built
; around
;
;*****************************************************************
;

(create-pred type: (type)
             comment: (triggered by any object-noun creates type field)
             concept-frame: (object = (nil)
                             number = (nil)
                             word-used = (nil))
             control-structure: ((expect object in last word features:
                                  return word features: )
                                 (expect number in last word part-of-speech:
                                  return word part-of-speech: )
                                 (expect word-used in last word item:
                                  return word item: ))
             required-slots: (object word-used)
```

```
                predicts: (instantiator-prediction)
                memory-level: ("CONCEPT-MEM")
                priority: 1)

;*************************************************************************
; Object Number prototypes
;*************************************************************************
;
; Most of the time, the number of the object is derived from the type
; structure (who gets it from whether the object word is singular or plural).
; There are a few instances, however, where the number must be picked
; up from another structure. In particular, the question how many puts
; a question mark into this field by predicting an object-number proto-
; type. Also, plural verbs and the word goal/goals make structures to
; state the numbr of the Object.
;
;*************************************************************************

(create-pred type: (object-number)
                comment: (triggered by the phrase how many)
                concept-frame: (number = (nil))
                control-structure: ((expect number in last word features:))
                required-slots: (number)
                predicts: nil
                memory-level: ("CONCEPT-MEM")
                priority: 1)

(create-pred type: (object-number-single)
                comment: (triggered by goal and singular verbs).
                concept-frame: (number = single
                                instantiator = (nil))
                control-structure: ((expect instantiator in last word item:))
                required-slots: (instantiator)
                predicts: nil
                memory-level: ("SYNTAX-MEM")
                priority: 1)

(create-pred type: (object-number-plural)
                comment: (triggered by goal and plural verbs).
                concept-frame: (number = plural
                                instantiator = (nil))
                control-structure: ((expect instantiator in last word item:))
                required-slots: (instantiator)
                predicts: nil
                memory-level: ("SYNTAX-MEM")
                priority: 1)

;*************************************************************************
```

```
; Object Class prototypes
;*****************************************************************************
;
; The class of an object determines what type of object the system should go
; search for (i.e. a schema instance, a schema prototype, or a goal).
; Unfortunately, no obvious way to always algorithmically determine the
; class of an object from a natural language query exists. However, the
; rather simple heuristics used here are provided correct answer most of
; the time. It should be noted, too, that where the parser does not specify
; the class, the dialogue interpreter has a chance to do so. Moreover, if
; no class gets specified, the system may still proceed; it may be just that
; its search takes longer.
;
; The goal prototype is predicted by the word goal.
;
; The class instance structure is predicted and instantiated by a mention of
; or query about the location of an object. Semanticly, this feature
; is nonsensical if applied to a schema prototype or a goal. It is also
; predicted if the number of an object is queried, since semantically this
; eliminates the "prototype" class, and if a goal is mentioned explicitly
; that structure will be picked up first by Object. [note: if connected to
; a vision system that allows multiple prototypes for one Object, change this]
;
; The default-prototype is predicted by the plural verb "are". In sentences
; using this word, the class "instance" is usually implied, since schema
; prototypes are rarely mentioned in the plural in this domain. Since this
; structure is the class structure searched by Object, it is overridden by
; an explicit mentions of goals or a location.
;
;*****************************************************************************

(create-pred type: (goal)
             comment: (triggered by the word goal)
             concept-frame: (class = goal
                             of-word = (of or for))
             control-structure: ((expect of-word in next word features:))
             required-slots: (of-word)
             predicts: nil
             memory-level: ("CONCEPT-MEM")
             priority: 1)

(create-pred type: (hypothesis)
             comment: (triggered by the word hypothesis)
             concept-frame: (class = hypothesis
                             of-word = (of or for))
             control-structure: ((expect of-word in next word features:))
             required-slots: (of-word)
             predicts: nil
```

```
                memory-level: ('CONCEPT-MEM')
                priority: 1)

(create-pred type: (default-prototype)
                comment: (predicted by a plural "to be")
                concept-frame: (class = prototype
                                instantiator = (nil))
                control-structure: ((expect instantiator in future instantiator
                                        return instantiator word-used))
                required-slots: (instantiator)
                predicts: nil
                memory-level: ('SYNTAX-MEM')
                priority: 1)

(create-pred type: (class-instance-structure)
                comment: (triggered the following ways
                                specifying (or asking about) a number of
                                specifying (or asking about) a specific location)
                concept-frame: (class = instance
                                grbge = (nil))
                control-structure: ((expect grbge in last word item:))
                predicts: nil
                required-slots: (grbge)
                memory-level: ('SYNTAX-MEM')
                priority: 1)

;*************************************************************************
; Relationship Prototypes
;*************************************************************************
;
; These prototypes create relationshis between entities. In particular,
; relationships may exist between objects (e.g.the bush next to the house),
; and between feature values (e.g. a vepua between 9 and 11).
;
; Relationships between objects are typically geometric in nature, and are
; refered to as geometric relationships for that reason. Relationships between
; feature values must be between values to the same feature ("a color near
; large" is nonsensical), and there must be some ordering defined on those
; values. These relationships are referred to by this ordering. The only
; implemented case here is the numeric ordering.
;
; As a special case, conjunctions (and disjunctions) can also exist between
; feature types (e.g. the size and color of the house). They therefore have
; their own set of prototypes to account for this possibility.
;
; Feature value relationship prototypes are predicted by their words (i.e
; "next to", "between"). Object relationships can (by their nature) exist
; only between object descriptions. They are therefore predicted by the
```

```
; instantiator structure.
;
;*****************************************************************************
;

(create-pred type: (geometric-binary-relationship)
             comment: (next to, etc.)
             concept-frame: (type = (binary-relationship)
                                first-object = (nil)
                                second-object = (nil))
             control-structure: ((expect type in last word part-of-speech:
                                      return word features:)
                                 (expect first-object in last object)
                                 (expect second-object in next object
                                      after first-object))
             required-slots: (type)
             predicts: nil
             memory-level: ('EVENT-MEM')
             priority: 1)

(create-pred type: (geometric-tertiary-relationship)
             comment: (between, etc.)
             concept-frame: (type = (tertiary-relationship)
                                first-object = (nil)
                                second-object = (nil)
                                third-object = (nil))
             control-structure: ((expect type in last word part-of-speech:
                                      return word features:)
                                 (expect first-object in last object)
                                 (expect second-object in next object)
                                 (expect third-object in next object
                                      after second-object))
             required-slots: (type)
             predicts: nil
             memory-level: ('EVENT-MEM')
             priority: 1)

(create-pred type: (numeric-binary-relationship)
             comment: (next to, etc.)
             concept-frame: (type = (binary-relationship)
                                first-number = (number))
             control-structure: ((expect type in last word part-of-speech:
                                      return word features:)
                                 (expect first-number
                                      in next word features:
                                      return word item:
                                      until geometric-binary-relationship))
             required-slots: (type first-number)
             predicts: (instantiator-prediction)
```

```
                memory-level: ('CONCEPT-MEM')
                priority: 1)

(create-pred type: (numeric-tertiary-relationship)
                comment: (between, etc.)
                concept-frame: (type = (tertiary-relationship)
                                first-number = (number)
                                second-number = (number))
                control-structure: ((expect type in last word part-of-speech:
                                        return word features:)
                                    (expect first-number
                                        in next word features:
                                        return word item:)
                                    (expect second-number
                                        in future word features:
                                        return word item:
                                        after first-number
                                        until geometric-tertiary-relationship
                                            or instantiator))
                required-slots: (type first-number second-number)
                predicts: (instantiator-prediction)
                memory-level: ('CONCEPT-MEM')
                priority: 1)


;*****************************************************************************
; Feature Element Prototypes
;*****************************************************************************
;
;
; These prototype define optional features of an object. In general, there are
; two types of features; statements that are made by the user about the object,
; and facts the user would like to know about the object. The former may be
; specified by the user in two ways. He/she may state the feature value. If
; the value implies the feature (i.e. blue => color), then that is all the
; user need specify. If not (e.g. the value is numeric), the user must first
; specify the value type [note: if cardinal numbers are to be implemented,
; the implied order of this must be changed for that case]. The other way
; that a feature may be specified is with a relative clause.
;
; Unknown features (i.e. feature inquiries) are introduced by an interrogative,
; and are represented as (feature-name question-mark). If what is asked is not
; the value of a specific feature but is instead a general question (e.g. what
; is a house) then a universal feature type (everything question-mark) is
; added to the features: field of the object.
;
;*****************************************************************************

(create-pred type: (value-based-feature-element)
```

```
                comment: (triggered by a feature value)
                concept-frame: (feature-type = (feature-value or feature-type)
                                feature-value = (nil))
                control-structure: ((expect feature-type
                                        in last word part-of-speech:
                                        return word features:)
                                    (expect feature-type
                                        in next word part-of-speech:
                                        return word features:)
                                    (expect feature-value in last word item:))
                required-slots: (feature-type feature-value)
                predicts: (instantiator-prediction)
                memory-level: ("CONCEPT-MEM")
                priority: 1)


(create-pred type: (specific-feature-question)
                comment: (triggered by what competes with
                            general-feature-question)
                concept-frame: (feature-type = (nil))
                control-structure: ((expect feature-type
                                        in future non-numeric-feature-type
                                        until general-feature-question
                                            or specific-bounded-feature-question
                                            or instantiator)
                                    (expect feature-type
                                        in future bounded-feature
                                        until general-feature-question
                                            or instantiator)
                                    (expect feature-type
                                        in future unspecific-numeric-feature
                                        until general-feature-question
                                            or instantiator))
                required-slots: (feature-type)
                predicts: nil ;;; (instantiator-prediction)
                memory-level: ("CONCEPT-MEM")
                priority: 1)


(create-pred type: (general-feature-question)
                comment: (triggered by what; competes with
                            specific-bounded-feature-question and
                            specific-feature-question)
                concept-frame: (feature-type = (nil)
                                trigger  = (nil))
                control-structure: ((expect trigger in future type
                                        until specific-feature-question
                                            or specific-bounded-feature-question)
```

```
                                        (expect trigger in future instantiator
                                          until specific-feature-question
                                             or specific-bounded-feature-question)
                                        (expect trigger in future to-have
                                          until specific-feature-question)
                                        (expect feature-type in last word features:))
                    required-slots: (feature-type trigger)
                    predicts: (instantiator-prediction)
                    memory-level: ("CONCEPT-MEM")
                    priority: 1)

(create-pred type: (non-numeric-feature-type)
                    comment: (triggered by a non-numeric feature type)
                    concept-frame: (feature = (feature-type))
                    control-structure: ((expect feature in last word part-of-speech:
                                          return word features:))
                    required-slots: (feature)
                    predicts: nil
                    memory-level: ("SYNTAX-MEM")
                    priority: 1)

(create-pred type: (numeric-feature)
                    comment: (triggered by any feature with a numerical range
                                competes with bounded-feature)
                    concept-frame: (feature = (feature-type)
                                      value = (number))
                    control-structure: ((expect feature in last word part-of-speech:
                                          return word item:)
                                        (expect value in future word features:
                                          until bounded-feature or instantiator
                                             or unspecific-numeric-feature
                                          return word item:))
                    required-slots: (feature value)
                    predicts: (instantiator-prediction)
                    memory-level: ("CONCEPT-MEM")
                    priority: 1)

(create-pred type: (plural-numeric-specific-feature)
                    comment: (triggered by any feature with a numerical range
                                competes with bounded-feature)
                    concept-frame: (feature = (feature-type)
                                      value =& (number))
                    control-structure: ((expect feature in last word part-of-speech:
                                          return word item:)
                                        (expect value in future word features:
                                          until bounded-feature or instantiator
                                             or plural-numeric-feature
                                          return word item:))
```

```
                required-slots: (feature value)
                predicts: (instantiator-prediction)
                memory-level: (*CONCEPT-MEM*)
                priority: 1)

(create-pred type: (unspecific-numeric-feature)
                comment: (triggered by any feature with a numerical range
                              competes with bounded-feature
                                        and plural-numeric-specific-feature
                                        and numeric-feature)
                concept-frame: (feature = (feature-type)
                                   instantiator =& (verb or single or plural
                                                       article or interrogative
                                                       or other or feature-type))
                control-structure: ((expect feature in last word part-of-speech:
                                         return word item:)
                                    (expect instantiator
                                         in future word part-of-speech:
                                         until bounded-feature or
                                             plural-numeric-specific-feature
                                             or numeric-feature))
                required-slots: (feature instantiator)
                predicts: nil
                memory-level: (*CONCEPT-MEM*)
                priority: 1)

(create-pred type: (bounded-feature)
                comment: (triggered by any feature with a numerical range)
                concept-frame: (feature = (feature-type)
                                   instantiator = (binary-relationship or
                                                       tertiary-relationship)
                                   value = (nil))
                control-structure: ((expect feature in last word part-of-speech:
                                         return word item:)
                                    (expect instantiator
                                        in next word part-of-speech:
                                        return word item:)
                                    (expect value
                                         in next numeric-binary-relationship
                                         until instantiator)
                                    (expect value
                                         in next numeric-tertiary-relationship
                                         until instantiator)
                                    (expect value in next numeric-feature
                                         until instantiator))
                required-slots: (feature instantiator)
                predicts: nil
                memory-level: (*CONCEPT-MEM*)
```

```
                priority: 1)

;*****************************************************************
; Object and Object intermediate structures
;*****************************************************************
;
; Collectively, these structures create an object. The feature-list is
; automatically predicted at the beginning of a sentence. It collects all
; the features stated or asked about the object. The instantiator is predicted
; at every point in the parse at which a break could occur (i.e. a relation-
; ship could shift the sentence from refering to one object to refering to
; another, as in X near Y). If it doesn't see a break in the next word it
; dies. If it does, it becomes instantiated, and in so doing instantiates
; the feature-list structure (hence the name). At this point, the feature-list
; a) stops collecting features and b) predicts the Object structure, which
; fills as many of its slots as possible and is immediately instantiated.
; The instantiator then predicts a new feature-list, and the process starts
; over for the next object.
;
; The eol-instantiator is predicted and invoked at the end of every sentence.
; It is just like instantiator, except it doesn't predict a new feature-list.
;
; The seperation of the feature-list and object structures is partly to divide
; an otherwise huge prediction prototype, partly to reduce the number of out-
; standing demons (for conceptual and computational simplicity), and largely
; an historical accident, stemming from the days before the UNTIL feature of
; PLUM.
;
;*****************************************************************

(create-pred type: (instantiator)
             comment: (signal the end of an object description)
             concept-frame: (word-used = (binary-relationship or
                                          tertiary-relationship))
             control-structure: ((expect word-used in
                                  next word part-of-speech:))
             required-slots: (word-used)
             predicts: (geometric-binary-relationship-prediction
                        geometric-tertiary-relationship-prediction
                        feature-list-prediction)
             memory-level: ('SYNTAX-MEM')
             priority: 1)

(create-pred type: (eol-instantiator)
             comments: (triggered at end of sentence to force an object)
             concept-frame: (end = (nil))
             control-structure: ((expect end in last word item:))
             required-slots: (end)
```

```
                predicts: nil
                memory-level: ("SYNTAX-MEM")
                priority: 1)


(create-pred type: (feature-list)
                comment: (Collects all feature pairs. inst. by end or something)
                concept-frame: (list =& (nil)
                                    instantiator = (nil))
                control-structure: ((expect list in future
                                        specific-feature-question
                                        until instantiator)
                                    (expect list in future
                                        general-feature-question
                                        until instantiator)
                                    (expect list in future
                                        value-based-feature-element
                                        until instantiator)
                                    (expect list in future
                                        relative-based-feature-element)
                                    (expect list in future location
                                        return location until instantiator)
                                    (expect list in future bounded-feature
                                        return bounded-feature until instantiator)
                                    (expect list in future numeric-feature
                                        return numeric-feature until instantiator)
                                    (expect list
                                        in future plural-numeric-specific-feature
                                        return plural-numeric-specific-feature
                                        until instantiator)
                                    (expect instantiator in future instantiator
                                        return instantiator word-used)
                                    (expect instantiator in future
                                        eol-instantiator))
                predicts: (object-prediction)
                required-slots: (instantiator)
                memory-level: ("CONCEPT-MEM")
                priority: 1)


(create-pred type: (object)
                comment: (Top level structure : one is always predicted)
                concept-frame: (class = (nil)
                                    object-type = (nil)
                                    token = QUESTION-MARK
                                    number = (nil)
                                    features = (nil)
                                   relationships = (nil))
```

```
                control-structure: ((expect class in last goal class)
                                    (expect class in last hypothesis class)
                                    (expect class in last class-instance-structure
                                       return class-instance-structure class)
                                    (expect class in last default-prototype
                                       return default-prototype class)
                                    (expect object-type in last type object
                                       return type object)
                                    (expect object-type in last object-type
                                       return object-type type)
                                    (expect number in last object-number number
                                       return object-number number)
                                     (expect number in last type number
                                        return type number)
                                     (expect features in last feature-list
                                        return feature-list)
                                    (expect relationships in next
                                       geometric-binary-relationship)
                                    (expect relationships in next
                                       geometric-tertiary-relationship))
                required-slots: (features)
                predicts: nil
                memory-level: ("EVENT-MEM")
                priority: 1)

;*****************************************************************
; To-Have
;*****************************************************************
;
; This is what has generally been avoided - an intermediate syntactic
; structure. It is used to differentiate some "general what" questions
; from "specific what" questions.
;
;*****************************************************************

(create-pred type: (to-have)
                comment: (triggered by to have)
                concept-frame: (to-have = (nil))
                control-structure: ((expect to-have in last word))
                required-slots: (to-have)
                predicts: nil
                memory-level: ("SYNTAX-MEM")
                priority: 1)

;*****************************************************************
; PLUM's system prediction prototype
;*****************************************************************
;
;
```

```
; This prototype is hard-wired into PLUM. It is given here for the sake
; of completeness.
;
;*****************************************************************************

(create-pred type: (word)
             concept-frame: (item = (nil)
                             part-of-speech = (nil))
             control-structure: ((expect item in current input-stream))
             predicts: nil
             required-slots: (item part-of-speech)
             memory-level: ('LEXICAL-MEM')
             priority: 1)


;*****************************************************************
; Structural Lisp Functions
;*****************************************************************
;
; This adds the word START to the beginning of the string and END to
; the back. This is to establish initial expectations, and to force
; instantiation of certain structures at the end.
;
;*****************************************************************

(defun p (lst) (parse (append (cons 'START lst)(list 'END))))


(msg t "==> VIS$PLUM loaded")

;*********************************************************************
; Word Definitions
;*********************************************************************


;*************************************************************************
; relationships
;*************************************************************************

(defword above (binary-relationship) nil (above))

(defword and (binary-relationship) nil (and))

(defword below (binary-relationship) nil (below))

(defword less-then (binary-relationship)
                   (numeric-binary-relationship-prediction)
                   (less then))
```

```
(defword between (tertiary-relationship)
                                (numeric-tertiary-relationship-prediction)
                                (between))

(defword next-to (binary-relationship) nil (next-to))

(defword or (binary-relationship) nil (or))

;*****************************************************************************
;
; Articles
;*****************************************************************************
;


(defword a (article)  nil (ref indef))
   (defword an (article) nil  (ref indef))

(defword the (article) nil (ref def))

;*****************************************************************************
;
; Semantic Objects
;*****************************************************************************
;

(defword bush (single) (type-prediction) (bush))
   (defword bushes (plural) (type-prediction) (bush))

(defword barn (single) (type-prediction) (bush))
   (defword barns (plural) (type-prediction) (barn))

(defword car (single) (type-prediction) (car))
   (defword cars (plural) (type-prediction) (car))

(defword chair (single) (type-prediction) (chair))
   (defword chairs (plural) (type-prediction) (chair))

(defword chimney (single) (type-prediction) (chimney))
   (defword chimneys (plural) (type-prediction) (chimney))

(defword curtain (single) (type-prediction) (curtain))
   (defword curtains (plural) (type-prediction) (curtain))

(defword drive-way (single) (type-prediction) (drive-way))
   (defword drive-ways (plural) (type-prediction) (drive-way))

(defword door (single) (type-prediction) (door))
   (defword doors (plural) (type-prediction) (doors))

(defword floor (single) (type-prediction) (floor))
   (defword floors (plural) (type-prediction) (floor))
```

```
(defword foliage (single) (type-prediction)  (foliage))

(defword garage (single) (type-prediction) (garage))
   (defword garages (plural) (type-prediction) (garage))

(defword grass (single) (type-prediction) (grass))

(defword ground (single) (type-prediction) (ground))

(defword house (single) (type-prediction) (house))
   (defword houses (plural) (type-prediction) (house))

(defword house-wall (single) (type-prediction) (house wall))
   (defword house-walls (plural) (type-prediction) (house wall))

(defword lamp (single) (type-prediction) (lamp))
   (defword lamps (plural) (type-prediction) (lamp))

(defword lawn (single) (type-prediction) (grass))
   (defword lawns (plural) (type-prediction) (grass))

(defword light-pole (single) (type-prediction) (light pole))
   (defword light-poles (plural) (type-prediction) (light pole))

(defword mail-box (single) (type-prediction) (mail box))
   (defword mail-boxes (plural) (type-prediction) (mail box))

(defword porch (single) (type-prediction) (porch))
   (defword porches (plural) (type-prediction) (porch))

(defword rain-gutter (single) (type-prediction) (rain gutter))
   (defword rain-gutters (plural) (type-prediction) (rain-gutter))

(defword road (single) (type-prediction) (road))
   (defword roads (plural) (type-prediction) (road))

(defword roof (single) (type-prediction) (roof))
   (defword roofs (plural) (type-prediction) (roof))

(defword shadow (single) (type-prediction) (shadow))
   (defword shadows (plural) (type-prediction) (shadow))

(defword shed (single) (type-prediction) (shed))
   (defword sheds (plural) (type-prediction) (shed))

(defword shutter (single) (type-prediction) (shutter))
   (defword shutters (plural) (type-prediction) (shutter))
```

```
(defword shrub (single) (type-prediction) (bush))
   (defword shrubs (plural) (type-prediction) (bush))

(defword sidewalk (single) (type-prediction) (sidewalk))
   (defword sidewalks (plural) (type-prediction) (sidewalk))

(defword sky (single) (type-prediction) (sky))

(defword sofa (single) (type-prediction) (sofa))
   (defword sofas (plural) (type-prediction) (sofa))

(defword table (single) (type-prediction) (table))
   (defword tables (plural) (type-prediction) (table))

(defword telephone-pole (single) (type-prediction) (telephone pole))
   (defword telephone-poles (plural) (type-prediction) (telephone poles))

(defword tree (single) (type-prediction) (tree))
   (defword trees (plural) (type-prediction) (tree))

(defword truck (single) (type-prediction) (truck))
   (defword trucks (plural) (type-prediction) (trucks))

(defword wall (single) (type-prediction) (wall))
   (defword walls (plural) (type-prediction) (wall))

(defword window (single) (type-prediction) (window))
   (defword windows (plural) (type-prediction) (window))

(defword wire (single) (type-prediction) (wire))
   (defword wires (plural) (type-prediction) (wire))

(defword vehicle (single) (type-prediction) (vehicle))
   (defword vehicles (plural) (type-prediction) (vehicles))

;*****************************************************************
;
; Feature Types
;*****************************************************************
;

(defword color (feature-type) (non-numeric-feature-type-prediction) (color))

(defword region (feature-type) (bounded-feature-prediction
                                numeric-feature-prediction
                                unspecific-numeric-feature-prediction
                                class-instance-structure-prediction)
                                (location))
   (syn region (area pseudo-region))
   (defword regions (feature-type) (bounded-feature-prediction
```

```
                                        plural-numeric-specific-feature-prediction
                                        unspecific-numeric-feature-prediction
                                        class-instance-structure-prediction)
                                        (locations))
    (syn regions (areas pseudo-regions))

(defword size (feature-type) (non-numeric-feature-type-prediction)  (size))

(defword texture (feature-type) (non-numeric-feature-type-prediction)
                                    (texture))

(defword vepua (feature-type) (bounded-feature-prediction
                                    unspecific-numeric-feature-prediction
                                    numeric-feature-prediction) (vepua))
    (syn vepua (vertical-edges-per-unit-area))

(defword wthr (feature-type)  (bounded-feature-prediction
                                    unspecific-numeric-feature-prediction
                                    numeric-feature-prediction) (wthr))
    (syn wthr (width-to-height-ratio))

;*****************************************************************************
;
; Feature Values
;*****************************************************************************
;

(defword blue (feature-value) (value-based-feature-element-prediction)
                                    (color))

(defword bright (feature-value) (value-based-feature-element-prediction)
                                    (illumination))

(defword dark (feature-value) (value-based-feature-element-prediction)
                                    (illumination))

(defword green (feature-value) (value-based-feature-element-prediction)
                                    (color))

(defword large (feature-value) (value-based-feature-element-prediction)
                                    (size))

(defword red (feature-value) (value-based-feature-element-prediction)
                                    (red))

(defword small (feature-value) (value-based-feature-element-prediction)
                                    (size))
;*****************************************************************************
;
; Interrogatives
;*****************************************************************************
;
```

```
(defword how-many (interrogative) (object-number-prediction
                                    class-instance-structure-prediction)
                                   (number QUESTION-MARK))

(defword what (interrogative) (specific-feature-question-prediction
                                general-feature-question-prediction)
                                (EVERYTHING))

(defword where (interrogative) (general-feature-question-prediction
                                 class-instance-structure-prediction)
                                (location))

;*****************************************************************
;
; Verbs
;*****************************************************************
;

(defword are (verb) (default-prototype-prediction
                     object-number-plural-prediction
                     instantiator-prediction) nil)

(defword is (verb) (instantiator-prediction
                     object-number-single-prediction) nil)

(defword has (verb) (instantiator-prediction
                     to-have-prediction
                     object-number-single-prediction) (to-have))

(defword have (verb) (instantiator-prediction
                     to-have-prediction
                     object-number-plural-prediction) (to-have))

;*****************************************************************
;
; Special Definitions
;*****************************************************************
;

(defword $number (feature-value) nil (number))

(defword START nil (feature-list-prediction) nil)

(defword END (other) (eol-instantiator-prediction) nil)

(phrase ((how many)(next to)(house wall)(house walls)(light pole)(mail box)
         (light poles)(mail boxes)(less then)(rain gutter)(rain gutters)
         (telephone pole)(telephone poles)))

(defword of (other) nil (of))
(defword for (other) nil (for))
(defword goal (other)  (goal-prediction
```

```
                                object-number-single-prediction) nil)
(defword goals (other) (goal-prediction
                                object-number-plural-prediction) nil)
(defword hypothesis (other) (hypothesis-prediction
                                        object-number-single-prediction) nil)
(defword it (single) nil nil)
(defword there (other) (class-instance-structure-prediction) nil)

(msg t "==> VIS$DICT loaded")
```

4.0 APPENDICES

4.1 Appendix A

Dialogues between a user and a vision system

[Scenario: The system just failed on a new house scene. The user is trying to find out why]

User: "Is there a house wall"
System: "Yes (confidence = .832...)"

U: "Is there a shutter"
S: "No"

U: "Is there a goal for a shutter"
S: "Yes, but I haven't found one"

U: "What is a shutter"
S: "A shutter is a dark rectangle with width to height ratio 1:2 next to a house wall"

U: "What regions are next to the house wall"
S: "195, 196, 233"

U: "Is region 195 dark"
S: "Yes (mean illumination = 21.03)

U: "What is its width-to-heigth-ratio"
S: "1.3:1.4"

U: "What is the width-to-heigth-ratio of region 196"
S: "1.1:2"

U: "Is 196 dark"
S: "No (mean illumination = 23.09...)"

U: "What is a dark shutter"
S: "A shutter with mean illumination under 23"

4.2 Appendix B

Some harder sentences which VISPLUM is perfectly happy with:

1) What color is the bush next to the large red chimney with a vepua between 9 and 11?

Yields:

```
(Objectxxx1
    (class nil)
    (type bush)
    (token ?)
    (number single)
    (features (color ?))
    (relationships (next-to Objectxxx2)))

(Objectxxx2
    (class nil)
    (type chimney)
    (token ?)
    (number single)
    (features (size large)
              (color red)
              (vepua (bounded 9 11)))
    (relationships nil))
```

2) What tree with wthr between 2 and 3 is between a house and a light pole?

```
(Objectxxx3
    (class nil)
    (type tree)
    (token ?)
    (number single)
    (features (EVERYTHING ?)
              (wthr (bounded 2 3)))
    (relationships (between Objectxxx4 Objectxxx5)))

(Objectxxx4
    (class nil)
    (type house)
    (token ?)
    (number single)
    (features nil)
    (relationships (and Objectxxx5)))

(Objectxxx5
    (class nil)
    (type (light pole))
```

(token ?)
(number single)
(features nil)
(relationships nil))

3) What texture is the blue house with vepua 10 in region 11?

(Objectxxx6
(class instance)
(type house)
(token ?)
(number single)
(features (texture ?)
(color blue)
(vepua 10)
(region 11))
(relationships nil))

4.3 Appendix C

Twenty more example sentences

1) Is there a goal for a bush?

2) Is there a hypothesis for a barn?

3) Are there any cars?

4) what is a chair?

5) what are chairs?

6) where are there chimneys?

7) Is there a green curtain?

8) Where is a green curtain?

9) How many doors are there?

10) How many doors?

11) what is next to the garage?

12) What trees are between the house and the light pole?

13) What has a vepua between 10 and 20?

14) What large blue car has vepua 13?

15) What has a vepua less then 10?

16) What is the vepua of the telephone pole?

17) What color lamp?

18) What has the color blue?

19) How many green sofas?

20) What table?

ATN PARSING WITH PLUM

Brian Stucky

1.0 INTRODUCTION

This project serves as a study of the implementation of an ATN parser in PLUM. The initial goals of the project were twofold: first, to demonstrate the flexibility of PLUM, a semantic-based parsing system, by simulating a syntactic-based parser such as ATN's, and to provide a system by which a user familiar with the ATN formalization could use PLUM as a driver for the system with virtually no knowledge or expertise in the machanics of PLUM. At this point, although more work is needed, I believe both goals have been satisfactorily accomplished. The project, and this report, consist of a general discussion of the implementation of ATN's in PLUM, a program that interprets or translates an ATN specification into PLUM-executable code, and a brief discussion of possible future directions of study which involves the combination of the syntactic ATN parser with the samantic PLUM parser to provide a powerful tool for natural language understanding.

2.0 IMPLEMENTING ATNS IN PLUM

At the simplest level, an ATN is a finite state automaton that imposes additional conditions and actions on the transition arcs. We will use the prediction prototype in PLUM to represent an arc in the network and LISP code to handle the conditions and actions as necessary. An example will facilitate this discussion.

```
(Q1
  (CAT V T
  (SETR AUX NIL)
 .BR
  (SETR V *)
  (TO QR))
```

We will need one prediction prototype to "duplicate" this arc between states Q1 and Q4. The prototype necessary for this arc will be:

```
(CREATE-PRED TYPE:              (Q1-Q4)
  CONCEPT-FRAME:                (value = (v))
  CONTROL-STRUCTURE:            ((do-first FQ1Q4 on current Q1-Q4)
                                 (expect value in current word))
  PREDICTS:                     (Q4-Q5-prediction)
  REQUIRED-SLOTS:               (value)
  MEMORY-LEVEL:                 ('SYNTAX-MEM')
  PRIORITY:                     1)
```

The implementation also requires the following LISP code:

```
(DEFUN FQ1Q4 (Q1-Q4)
  (SETR AUX NIL)
  (SETR V *))
```

The name of the prototype is created to indicate the states connected by the arc. Its only purpose is for clarity. The "CAT" declaration in the ATN indicates the arc is to be traversed if a verb is encountered in the input-stream. Hence, the concept-frame requires a word of type 'verb' to be instantiated. This is also recognized as a required slot in the prototype and we expect this in the current word. We must utilize the DO-FIRST facility to handle the specified actions. This involves calling the function specified and computing the appropriate updates to the register. It is not necessary to require any additional information from the function since the instantiation of the frame depends only on the word being parsed from the input stream. Finally, the PREDICTS slot indicates the next arcs to be traversed if the prototype is successful (in this example, assume state Q4 has some arc to state Q5). With this specific example in mind, I will generalize the constructions necessary in the prediction prototypes.

2.1 Registers

The registers are easily handled by global variables. They will only be accessed from the LISP functions we create. Since the entire PLUM system is based in the CLisp environment, we simply create and initialize all registers needed when the system is loaded. They will be of the form *AUX*, *NP*, etc...

2.2 Prediction of ARCS (prototypes)

This is not handled as easily as the previous example may have indicated. In fact, we are faced with somewhat of a dilemma. Note that the control-structure of the example expects the input in the current word. This implies that the activation of all prototypes (arcs) comes from the words parsed as defined in the dictionary. Therefore, the current word being parsed will specify which prototypes should be triggered and the PREDICTS slot should be empty. If this slot actually predicts the next prototype, it will be activated and will search to fill its slots on the current word which has not changed and will presumably prematurely fail. The implementation will work if we activate predictions from the syntactic items (and leave the predicts slot nil) but this is not quite the idea of ATN's. In ATN processing, we would like the current state of the automaton to dictate the processing - not the dictionary items. To simulate this in PLUM, we remove all predictions from the dictionary items, 'pad' the prototypes with an additional START arc, utilize the PREDICTS slot of the prototype to activate the next arc, and search for the required slot in the NEXT word. In this manner, any prototype activated will not be instantiated until the next word is processed. Only then will the PREDICTS slot activate further prototypes which can't be instantiated until the next word is processed. This requires some extra manipulation of the actions for an arc but it preserves the sanctity of the ATN motivation.

2.3 CAT & JMP ARCS

These arcs are trivially handled. As we can see from the example, the CAT arc only requires an indication of the type of word required in the input-stream in the concept-frame and the listing of this category as a required slot. JMP arcs only require a prediction of the state we jump to and an empty concept-frame (unless the arc has conditions specific on it). The necessary category can either be required in the current or next word as we have previously discussed.

2.4 Conditions & Actions

As we have seen from the previous example, the existence of conditions and/or actions on an arc necessitate the utilization of the do-first control structure. In the case of actions to be performed on the arc, we simply call a LISP function that performs the necessary operations. No further work is necessary. For conditions, we must provide a little more. Since only one do-first clause is recognized in the current PLUM implementation, conditions and actions will be lumped in one function. Conditions on an arc impose additional constraints that must be fulfilled before the arc is successfully traversed. Therefore, we must create an additional required slot that ensures the condition is met. We can use the BLACKBOARD in PLUM to handle this. As an example, assume the previous ATN arc also had a condition imposed on it. Then the resulting prototype would take this form:

```
CONCEPT-FRAME:        (value = (v)
                       RQ1QR = (OK))
```

CONTROL-STRUCTURE: ((do-first FQ1Q4 on current Q1-Q4)
(expect value in current word)
(expect RQ1Q4 in current blackboard))
REQUIRED-SLOTS: (value RQ1Q4)

The function FQ1Q4 would compute the given condition and place the result on the blackboard where its value could be checked by the prototype. We need not pass any specific values, just an indication of success or failure (in my implementation, OK → t and NOPE → f). After the assignment to the blackboard, the specified actions are carried out. There is a potential for problems in that the actions will be carried out before the prototype is actually instantiated, however, all cases I have viewed will have no conflicts. It would be beneficial to change the function of the do-first clause so two functions could be specified; one to take place on activation (conditions) and one to take place after instantiation (actions).

2.5 PUSH & POP - Recursive Nets

The function of recursive networks in ATN's is to provide a lower level of computation on a portion of the input-stream independent of previous actions. It basically involves the saving of current registers on the upper level, initializing and computing new values on a lower level, and passing the result to the calling level when the POP is encountered to be used with the prior computations. We have two options available to us to implement this in PLUM. The first is a redefinition of the PARSE function in PLUM to allow recursive calls to handle the push and pop. This could cause some problems as it may require a morass of prototypes to be

"floating" about various levels of computation. I have instead opted for an external controlling device that will basically allow us to keep everything on one level. As I have mentioned above, the only thing we really have to be concerned with is the passing and reinitializing of the various registers. To handle this, we create a stack for each register and work through a controlling structure that is called as an action from the do-first clause when a pop or push is encountered. When a push is encountered, the current value of each register is put on a stack and they are resent to nil (unless we are told otherwise by a SENDR instruction). On a pop, we simply reset the registers to their previous value from the stack (unless a LEFTR instruction is encountered). We also define an additional register, *OUT* which will contain the structure computed at a lower level - in essence the result of a BUILDQ function typically associated with the pop. The structure is very crude at this point and further research will probably yield more efficient implementations. However, I am convinced that the problem of recursive nets should be handled by some controller external to PLUM rather than in PLUM through the redefinition of parse.

3.0 THE ATN-PLUM INTERPRETER

A system written in CLisp that performs the above translation has been completed. Please refer to APPENDIX B which contains the interpreter code and a sample run of the system with the input and output. You will note that minor surgery has been done on the ATN syntax to facilitate the translation but the changes are both trivial and self-explanatory. APPENDIX A contains a sample run of an ATN constructed in PLUM by the interpreter with the output. It includes the prototypes and auxiliary functions constructed along with the dictionary. The following is a delineation of the procedure used in the interpreter.

The system requires as input two arguments: the ATN specification which takes the form of a set of arcs and a list of the states connected in the ATN. For example, if state Q1 has arcs to both states Q2 and Q3, it is represented as (Q1 (Q2 Q3)). The latter is used in the construction of the PREDICTS slot of the prototype. The output of the system consists of a set of prototypes and a set of LISP functions for the corresponding actions and conditions.

The interpreter begins by constructing a prototype template and then it seeks to fill each slot. CAT and JMP arcs are handled by one component and PUSH and POP arcs by another. Each slot is constructed as specified below:

Name: The system simply constructs a name based on the initial and final states of the arc. This is done only for the sake of clarity. It also enable us to maintain consistency between the naming of arcs and the listing of arcs in the PREDICTS

slot.

Concept-frame: In the case of CAT arcs, we construct a slot for the category of word that must be parsed for traversal. If there is a condition, we add a second slot that seeks to fill the value with the result of the computation. PUSH, POP, and JMP arcs create a slot to be filled by the do-first function. No category type slots are necessary.

Predicts: This component utilizes the second input argument; the lists of arc-links. It searches in the arc-list with the final state of the arc and fetches all states that can be reached from the final state to construct the predictions.

Required-Slots: This component takes the result of the CONCEPT-FRAME construction and lists all of its slots. In this system, anything in the concept-frame (CAT or conditions) will be required.

Control-Structure: This component is the real workhorse as it both fills the control-structure argument of the prototype and dynamically constructs the LISP function for the do-first clause if necessary. The control-structure argument is filled in three steps. If there is an action and/or condition or we have a PUSH or POP arc we construct a form:

(do-first FUNC on current ARC)

where FUNC is a function name (based on the states) and arc is a dummy argument to pass the current prototype although we don't really need the frame. Next, if we are dealing with a CAT arc, we construct an EXPECT clause to pull the type from the current word. Finally, if there is a condition specified, we add an EXPECT

clause to search for the value of the conditional argument on the blackboard. This completes the construction of the control-structure. Now the LISP function must be created in the event conditions, actions, or PUSH-POP arcs.

1. Conditions require the creation of a template to pass the value to the blackboard:

```
(cond
(COND (okward))
(t     (nopeword)))
```

COND is the condition specified in the ATN specification of the arc. *BLACKBOARD* is assigned the value of the cond. OKWORD and NOPEWORD are dictionary items of type OK and NOPE respectively which indicate the success or failure of the condition.

2. Actions append the computations specified to the function creation. In ATN's, the value * refers to the current value of a specific register - in most cases, the current word. We change this to a reference to the input stream to get the word. In the case of a POP, * refers to a buildq structure specified in the corresponding PUSH and this value is appropriately represented. Finally, we substitute SETQ for SETR since the PLUM implementation uses global variables for the registers.

3. In the case of a PUSH or POP arc, we add a call to the recursive net controller, ST-CTRL which takes as the argument push or pop. If we have a POP, the special register *out* is assigned the value of the lower level computation, usually in the form of a buildq.

Controller-Functions: The interpreter also requires the use of several functions during the actual computation. These include the aforementioned recursive controller, a simple BUILDQ function which creates a syntactic form filled with the current values of the registers, initializing functions to set up the registers as global variables, and a simple definition of GETR which just returns the value of a specified register. All of these functions are present and active while the system executes but they are not used in the actual translation of ATN specifications to PLUM.

4.0 FUTURE DIRECTIONS

Future research and work on the system should take two forms. The first is a continuation of the work mentioned in this report. This work should include:

1. Increasing the capability of the interpreter to handle
   -WRD arcs
   -VIR arcs
   -HOLD arcs
   and other ATN features as deemed necessary.

2. Updating the recursive net controller so it is both more sophisticated and flexible.

3. Modifying the role of the do-first clause in PLUM to handle constructs that specify functions to be executed upon activation (conditions) and upon instantiation (actions).

   (do first F1 after F2 on search-direction memory-structure)

4. Modification to the ATN to decrease non-determinism for the use as described below.

The second line of research is the aforementioned creation of a powerful natural language understanding tool that would use the syntactic-based ATN parser in PLUM with PLUM's own semantic-based representation tools. The function of this combined system would be similar to the RUSParser. In general terms, the ATN built in plan will drive the parser on a syntactic level (*SYNTAX-MEM*) with additional "requests" on arcs that will activate and seek to instantiate additional semantic prototypes (*CONCEPT-MEM*). Initially, it would seem beneficial to have the dictionary items predict the semantic prototypes as we currently operate. These prototypes, operating on a different level, would seek to fill the slots of their concept-frame with syntactic units parsed by the ATN. Consider the following

examples:

1. John gave Mary to the Sheik.
2. John gave Mary a book.
3. John gave the sheik Mary.

Without explicit use of syntax, PLUM has difficulty parsing these sentences into ATRANS conceptual dependencies. Indeed, to actually handle problems such as this required the construction of additional PLUM capabilities and a heirarchy of prototypes. This is all because the recipient slot of the ATRANS can assume various syntactic roles. With the inclusion of the ATN, we parse the sentence on a syntactic basis until the ATRANS verb "gave" is read. This verb will activate the ATRANS prototype and tnen it is a simple manner of waiting for the ATN to parse the necessary syntax to fill all the ATRANS slot. The "John gave Mary" examples can be determined on a strictly syntactic basis (if a recipient exists it either follows the "gave" or is in a prepositional to-phrase) and it shouldn't be too difficult to relate this information from syntax (ATN) to semantics (PLUM CD's). We have the unique advantage of both the ATN and CD's operating in <u>one</u> system - the PLUM environment. This should make any message-passing (transmissions and requests in RUSParser) trivially handled.

These ideas are very rough but I believe the potential exists for the creation of a powerful tool for language analysis. At the very least, we have demonstrated the flexibility of PLUM and this is an advantage in itself.

```
(defword # (eos) nil nil)
(defword okword (OK) nil nil)
(defword nopeword (NOPE) nil nil)
(defword a (aux) nil nil)
(syn a (the an))
(defword big (adj) nil nil)
(syn big (small blue green yellow red little large empty))
(defword on (prep) nil nil)
(syn on (by near under beside))
(defword block (n) nil nil)
(syn block (boy girl dog cube sphere pyramid table floor box))
(defword moved (v) nil nil)
(syn moved (ran walked))

(defword *start* nil (S1-Q2-prediction) nil)


(msg t ">atndict.evl loaded.")
```

```
(defun getr (regt)
  regt)

(defun buildq (form1 reg-list)
  (prog (f1)
      (setq f1 form1)
  l1    (cond ((null reg-list) (go done)))
        (setq f1 (subst (caar reg-list) (cadar reg-list) f1))
        (setq reg-list (cdr reg-list))
        (go l1)
  done  (setq *out* f1)
        (return t)))

(defun st-ctrl (optr)
  (cond
    ((eq optr 'push)
      (setq *subj-st* (append (list *subj*) *subj-st*))
      (setq *subj* nil)
      (setq *type-st* (append (list *type*) *type-st*))
      (setq *type* nil)
      (setq *aux-st* (append (list *aux*) *aux-st*))
      (setq *aux* nil)
      (setq *v-st* (append (list *v*) *v-st*))
      (setq *v* nil)
      (setq *vp-st* (append (list *vp*) *vp-st*))
      (setq *vp* nil)
      (setq *n-st* (append (list *n*) *n-st*))
      (setq *n* nil))
    ((eq optr 'pop)
      (setq *subj* (car *subj-st*))
      (setq *subj-st* (cdr *subj-st*))
      (setq *type* (car *type-st*))
      (setq *type-st* (cdr *type-st*))
      (setq *aux* (car *aux-st*))
      (setq *aux-st* (cdr *aux-st*))
      (setq *v* (car *v-st*))
      (setq *v-st* (cdr *v-st*))
      (setq *vp* (car *vp-st*))
      (setq *vp-st* (cdr *vp-st*))
      (setq *n* (car *n-st*))
      (setq *n-st* (cdr *n-st*)))))

(setq *subj* nil)  (setq *subj-st* (list nil))
(setq *type* nil)  (setq *type-st* (list nil))
(setq *aux* nil)   (setq *aux-st* (list nil))
(setq *v* nil)     (setq *v-st* (list nil))
(setq *vp* nil)    (setq *vp-st* (list nil))
(setq *n* nil)     (setq *n-st* (list nil))
(setq *out* nil)

(msg t ">atnf [auxiliary functions] loaded.")
```

```
(setq a '(
  (S1 Q2
    (cat aux t)
    ((setr *aux* *) (setr *type* (quote dcl))))

  (Q2 NP1
    (push NP1 t))

  (NP1 NP2
    (cat n t)
    ((setr *n* *)))

  (NP2 Q3
    (pop (buildq '(NP e) (list (list *n* 'e))) t)
    ((setr *subj* *out*)))

  (Q3 Q4
    (cat v t)
    ((setr *v* *)))

  (Q4 NP3
    (push NP3 t))

  (Q4 QF
    (cat eos t)
    ((buildq '(S e # % (VP &))
             (list (list *type* 'e)
                   (list *subj* '#)
                   (list *aux* '%)
                   (list *v* '&)))))

  (NP3 NP4
   (cat n t)
   ((setr *n* *)))

  (NP4 Q5
    (pop (buildq '(NP e) (list (list *n* 'e))) t)
    ((setr *vp* (buildq '(VP (V e) *out*)
                        (list (list *v* 'e))))))

  (Q5 QF
    (cat eos t)
    ((buildq '(S e # % &)
             (list (list *type* 'e)
                   (list *subj* '#)
                   (list *aux* '%)
                   (list *vp* '&)))))))

(setq b
'((S1 (Q2))
 (Q2 (NP1))
 (NP1 (NP2))
 (NP2 (NP3))
 (Q3 (Q4))
 (Q4 (NP3 QF))
 (NP3 (NP4))
 (NP4 (Q5))
 (Q5 (QF))
))


;*****************************************************************************

(setq c '(

 (create-pred type: (S1-Q2)
   concept-frame: (cat = (aux))
   control-structure: ((do-first FS1-Q2 on last word)
                       (expect cat in next word))
   predicts: (Q2-NP1-prediction)
   required-slots: (cat)
   memory-level:  (*SYNTAX-MEM*)
   priority: 1)

(defun FS1-Q2 (word)
   (setq *aux* (cadr stream))
   (setq *type* (quote dcl)))

; *****

(create-pred type: (Q2-NP1)
   concept-frame: (RQ2NP1 = (OK))
   control-structure: ((do-first FQ2-NP1 on last word)
```

```
                        (expect RQ2NP1 in current blackboard))
   predicts: (NP1-NP2-prediction)
   required-slots: (RQ2NP1)
   memory-level: (*SYNTAX-MEM*)
   priority: 1)

 (dex FQ2-NP1 (word)
   (:= *blackboard*
     (cond
       (t (quote okword))
       (t (quote nopeword))))
     (st-ctrl push))

 ;*****

(create-pred type: (NP1-NP2)
   concept-frame: (cat = (n))
   control-structure: ((do-first FNP1-NP2 on last word)
                       (expect cat in next word))
   predicts: (NP2-Q3-prediction)
   required-slots: (cat)
   memory-level: (*SYNTAX-MEM*)
   priority: 1)

(defun FNP1-NP2 (word)
   (setq *n* (cadr stream)))

; *****

(create-pred type: (NP2-Q3)
   concept-frame: (RNP2Q3 = (OK))
   control-structure: ((do-first FNP2-Q3 on last word)
                       (expect RNP2Q3 in current blackboard))
   predicts: (Q3-Q4-prediction)
   required-slots: (RNP2Q3)
   memory-level: (*SYNTAX-MEM*)
   priority: 1)

(dex FNP2-Q3 (word)
 (:= *blackboard*
   (cond (t (quote okword))
   (t (quote nopeword))))
   (buildq '(NP +1) (list (list *n* '+1)))
   (st-ctrl pop)
   (setq *subj* *out*))

; *****

(create-pred type: (Q3-Q4)
   concept-frame: (cat = (v))
   control-structure: ((do-first FQ3-Q4 on last word)
                       (expect cat in next word))
   predicts: (Q4-NP3-prediction Q4-QF-prediction)
   required-slots: (cat)
   memory-level: (*SYNTAX-MEM*)
   priority: 1)

(defun FQ3-Q4 (word)
  (setq *v* (cadr stream)))

; *****

(create-pred type: (Q4-NP3)
   concept-frame: (RQ4NP3 = (OK))
   control-structure: ((do-first FQ4-NP3 on last word)
                       (expect RQ4NP3 in current blackboard))
   predicts: (NP3-NP4-prediction)
   required-slots: (RQ4NP3)
   memory-level: (*SYNTAX-MEM*)
   priority: 1)

 (dex FQ4-NP3 (word)
   (:= *blackboard*
     (cond
       (t (quote okword))
       (t (quote nopeword))))
       (st-ctrl push))

; *****

(create-pred type: (Q4-QF)
  concept-frame: (cat = (eos))
  control-structure: ((do-first FQ4-QF on last word)
```

```
                        (expect cat in next word))
  predicts: nil
  required-slots: (cat)
  memory-level: (*SYNTAX-MEM*)
  priority: 1)

(defun FQ4-QF (word)
  (buildq '(S +1 +2 +3 (VP +4))
          (list (list *type* '+1)
                (list *subj* '+2)
                (list *aux* '+3)
                (list *v* '+4))))

; *****

(create-pred type: (NP3-NP4)
  concept-frame: (cat = (n))
  control-structure: ((do-first FNP3-NP4 on last word)
                      (expect cat in next word))
  predicts: (NP4-Q5-prediction)
  required-slots: (cat)
  memory-level: (*SYNTAX-MEM*)
  priority: 1)

(defun FNP3-NP4 (word)
  (setq *n* (cadr stream)))

; *****

(create-pred type: (NP4-Q5)
  concept-frame: (RNP4Q5 = (OK))
  control-structure: ((do-first FNP4-Q5 on last word)
                      (expect RNP4Q5 in current blackboard))
  predicts: (Q5-QF-prediction)
  required-slots: (RNP4Q5)
  memory-level: (*SYNTAX-MEM*)
  priority: 1)

(dex FNP4-Q5 (word)
  (:= *blackboard*
    (cond
      (t (quote okword))
      (t (quote nopeword))))
  (buildq '(NP +1) (list (list *n* '+1)))
  (st-ctrl pop)
  (setq *vp* (buildq '(VP (V +1) *out*) (list (list *v* '+1)))))

; *****

(create-pred type: (Q5-QF)
  concept-frame: (cat = (eos))
  control-structure: ((do-first FQ5-QF on last word)
                      (expect cat in next word))
  predicts: nil
  required-slots: (cat)
  memory-level: (*SYNTAX-MEM*)
  priority: 1)

(defun FQ5-QF (word)
  (buildq '(S +1 +2 +3 +4)
          (list (list *type* '+1)
                (list *subj* '+2)
                (list *aux* '+3)
                (list *vp* '+4))))
)

(msg t ">atno.evl loaded.")
```

```
(defun atn (a-specs arcl)
  (terpri)
  (print a-specs)
  (terpri) (terpri)
  (print arcl)
  (terpri) (terpri)
  (setq plum-atn (atn-1 a-specs arcl))
  (terpri))

(defun atn-1 (a-specs arcl)
  (prog (a-temp a-plum)
      (setq a-plum (list nil))
      (setq a-temp a-specs)
  who  (cond ((null a-temp) (go bks)))
       (cond
         ((eq (car (caddar a-temp)) 'cat)
           (setq a-plum (append a-plum (make-arc-1 (car a-temp) arcl)
                                (list '*****))))
         (t
           (setq a-plum (append a-plum (make-arc-2 (car a-temp) arcl)
                                (list '*****)))))
       (setq a-temp (cdr a-temp))
       (go who)
  bks  (return a-plum)))


(defun make-arc-1 (arc-spec arcl)
  (setq name (c-name (car arc-spec) (cadr arc-spec)))
  (setq plist (c-preds (cadr arc-spec) arcl))
  (cond
    ((null plist) nil)
    (t (setq plist (cdr plist))))
  (setq aconds (c-conds arc-spec))
  (setq csl (c-ctrl aconds (caddr arc-spec) (cadddr arc-spec) (car name)))
  (setq rslots (c-slots aconds))
  (disp name aconds (cadr csl) plist rslots (car csl))
  (list
    (list
      'create-pred 'type: name
      'concept-frame:  aconds
      'control-structure: (cadr csl)
      'predicts: plist
      'required-slots: rslots
      'memory-level: '(*SYNTAX-MEM*)
      'priority: '1)
    (car csl)))

(defun make-arc-2 (arc-spec arcl)
  (setq name (c-name (car arc-spec) (cadr arc-spec)))
  (setq plist (c-preds (cadr arc-spec) arcl))
  (cond
    ((null plist) nil)
    (t (setq plist (cdr plist))))
  (setq aconds (c1-conds arc-spec))
  (setq csl (c1-ctrl aconds (caddr arc-spec) (cadddr arc-spec) (car name)))
  (setq rslots (c1-slots aconds))
  (disp name aconds (cadr csl) plist rslots (car csl))
  (list
    (list
      'create-pred 'type: name
      'concept-frame: aconds
      'control-structure: (cadr csl)
      'predicts: plist
      'required-slots: rslots
      'memory-level: '(*SYNTAX-MEM*)
      'priority: '1)
    (car csl)))

(defun c-name (init finl)
  (list (pack (append (unpack init) (list '-) (unpack finl)))))

(defun c-preds (finl arc-l)
  (cond
    ((null arc-l) nil)
    ((eq finl (caar arc-l)) (pred1 finl (cadar arc-l)))
    (t (c-preds finl (cdr arc-l)))))

(defun pred1 (finl next-l)
  (prog (f1 p-lst)
     (setq p-lst (list nil))
     (setq f1 next-l)
  l2   (cond ((null next-l) (go fini)))
```

```
        (setq p-lst (append p-lst
          (list (pack (append (unpack finl) (list '-) (unpack (car next-l))
                              (list '-) (unpack 'prediction))))))
        (setq next-l (cdr next-l))
        (go l2)
  finl (return p-lst)))

(defun c-conds (arc-spec)
  (setq init (car arc-spec))
  (setq finl (cadr arc-spec))
  (setq arc-spec (caddr arc-spec))
  (setq cf (list 'cat '= (list (cadr arc-spec))))
  (cond
    ((eq (caddr arc-spec) 't) cf)
    (t
      (setq nl (pack (append (list 'R) (unpack init) (unpack finl))))
      (append cf (list nl '= (list 'OK))))))

(defun cl-conds (arc-spec)
  (setq init (car arc-spec))
  (setq finl (cadr arc-spec))
  (setq nl (pack (append (list 'R) (unpack init) (unpack finl))))
  (list nl '= (list 'OK)))

(defun c-slots (cf)
  (setq rs (list 'cat))
  (cond
    ((null (cdddr cf)) rs)
    (t
      (append rs (list (cadddr cf))))))

(defun cl-slots (cf)
  (list (car cf)))

(defun c-ctrl (cf arc-spec actns nml)
  (setq nm2 (pack (append (list 'F) (unpack nml))))
  (setq call (list (list 'do-first nm2 'on 'last nml)))
  (setq call (append call (list (list 'expect 'cat 'in 'next 'word))))
  (cond
    ((eq (caddr arc-spec) 't)
      (setq fnl (list 'defun nm2 (list nml)))
      (setq fnl (append fnl actns))
      (list (conv fnl) call))
    (t
      (setq call (append call (list
                 (list 'expect (cadddr cf) 'in 'current 'blackboard))))
      (setq fnl (list 'dex nm2 (list nml)))
      (setq fnl (append fnl (list (list ':= '*blackboard* (list 'cond
        (list (caddr arc-spec) '(quote okword))
        (list 't '(quote nopeword)))))))
      (setq fnl (append fnl actns))
      (list (conv fnl) call))))

(defun cl-ctrl (cf arc-spec actns nml)
  (setq nm2 (pack (append (list 'F) (unpack nml))))
  (setq call (list (list 'do-first nm2 'on 'last nml)))
  (setq call (append call (list
    (list 'expect (car cf) 'in 'current 'blackboard))))
  (setq fnl (list 'dex nm2 (list nml)))
  (setq fnl (append fnl (list (list ':= '*blackboard*
    (list 'cond (list (caddr arc-spec) '(quote okword))
    (list 't '(quote nopeword)))))))
  (cond
    ((eq (car arc-spec) 'push)
      (setq fnl (append fnl actns))
      (setq fnl (append fnl (list (list 'st-ctrl 'push)))))
    (t
      (setq fnl (append fnl (list (cadr arc-spec) (list 'st-ctrl 'pop))))
      (setq fnl (append fnl actns))))
  (list fnl call))


(defun conv (fnn)
  (setq tpl (subst 'setq 'setr fnn))
  (subst (list 'car 'stream) '* tpl))


(defun disp (v w x y z q)
  (terpri)
  (princ "(") (prinl 'create-pred) (princ " ") (prinl 'type:) (princ " ")
  (prinl v) (terpri) (princ "  ")
  (prinl 'concept-frame:) (princ " ") (prinl w)
  (terpri) (princ "  ")
```

```
  (prinl 'control-structure:) (princ " ") (princ "(") (disp-cs x) (princ ")")
  (terpri) (princ "  ")
  (prinl 'predicts:) (princ " ") (prinl y)
  (terpri) (princ "  ")
  (prinl 'required-slots:) (princ " ") (prinl z)
  (terpri) (princ "  ")
  (prinl 'memory-level:) (princ " ") (prinl '(*SYNTAX-MEM*))
  (terpri) (princ "  ")
  (prinl 'priority:) (princ " ") (prinl '1) (princ ")")
  (terpri)
  (terpri)
  (print q)
  (terpri) (terpri)
  (terpri) (terpri))


(defun disp-cs (x)
  (cond
    ((null x) nil)
    (t
      (prinl (car x))
      (disp-csl (cdr x)))))

(defun disp-csl (x)
  (cond
    ((null x) nil)
    (t
      (terpri)
      (princ "                          ")
      (prinl (car x))
      (disp-csl (cdr x)))))
```

PRONOUN RESOLUTION WITH PLUM

Michael Sullivan

1.0 INTRODUCTION

This PLUM experiment addresses the topic of pronoun resolution (anaphora) within a discourse setting, where pronouns in later sentences refer to things were originally mentioned in previous sentences. There is much that can be done in this area, and this work just scratches the surface. However, we sought to come up with as general a resolution algorithm as possible, so that code to cover new situations not encountered by our test sentences could be easily added. The program we have inplemented is called ANA.

In order to restrict this effort, we have made some assumptions about the sentences we wish to tackle. These assumptions can be violated in everyday use of the English language, but for the vast majority of sentences the assumptions should hold. The assumptions we made are:

1. The sentences in a discourse set must be related. The topic introduced in the first sentence of a discourse set relates to the second and further sentences of the discourse set; we are not going to change the subject or introduce a totally new topic in the later sentences.

2. All pronouns refer to objects or people who have already been mentioned in the discourse set. We will not try to handle sentences like "He is a fine man', said John of Harry." ANA keeps track of all possible referents as it encounters them. When it sees a pronoun, it considers only those referents already encountered as possible resolutions for the pronoun.

3. Each pronoun can only refer to a noun. This was done in part because only noun phrases can become referents, and our resolution algorithm assumes (except in a couple of special cases) that it is dealing with referents. A discourse set like "John tried to eat while handcuffed. It was difficult.", where "it" refers to the eating, can't be handled by my ANA.

2.0 THE TEST DISCOURSE SETS HANDLED BY ANA

ANA handles twelve sets of sentences as an illustration of its capabilities. They are each listed below, with a brief explanation of the reasoning required to determine the proper referent for each of the pronouns.

The first set of sentences illustrates the simplest kind of pronoun resolution - simple feature matching between pronouns and possible referents, with only one possible match.

Discourse Set 1 - John bought a honda.

He likes it.

The features of "he" are matched only by "John", and the features of "it" are met only by "honda". This is the simplest strategy for resolving pronomial referents.

The next seven sets of sentences illustrate cases where feature matching is not always sufficient to resulve pronouns. In these sentences additional reasoning is required to decide among potential candidates.

Discourse Set 2 - John gave Bill the book.

He wanted to.

The first sentence is represented by an ATRANS event. The information in this conceptual dependency frame is used to resolve ambiguous pronouns in the next sentence, along with time frame information. If the word "then" or a similar time indicator appears in the next sentence, ANA assumes the events in this sentence took place following the ATRANS. If there is no such indicator, ANS assumes the subsequent descriptions took place in the same time frame, and are either

restatements of elaborations of the prior events.

Feature matching leaves us with two possibilities: "He" could refer to either John or Bill. The gapped verb in the second sentence allows us to infer that the subject of the sentence must be the actor of some previous event. Since John was the actor in the previous ATRANS, "he" must refer to John.

Discourse Set 3 - John gave Bill the book.

Then he read it.

Here we resolve "it" by feature matching to understand that the object read was the book. We must then make an inference about the enabling conditoins for reading in order to deduce that the person reading the book must first possess the book. Using inference rules of the sort first proposed by (Rieger, 1976), we infer that "he" must refer to Bill.

Discourse Set 4 - John gave Bill the book.

But he read it first.

This set is similar to discourse set 3, except that the word "first" clues us to the fact the reading took place before the giving, so the original possessor of the book, "John", must be the referent for "he" in this case.

Discourse Set 5 - John gave Bill the book.

Then he took it from him.

To resolve "he", first we decide that the features of "it" match only those of "book", so "it" refers to the book. We then have another ATRANS representation with two indeterminent role bindings:

act = ATRANS

actor = UNKNOWN1 referent for

recipient = UNKNOWN1 "he"

source = UNKNOWN2 referent for "him"

object = book

By inference, we know that the source of this ATRANs must have possessed the book before it could be taken from him. And again by inference, we know that Bill possessed the book after the initial ATRANS of the first sentence. So we conclude that the referent for "him" must be Bill. For syntactic and semantic reasons we can say that "he" and "him" must not refer to the same person. Having eliminated Bill (because Bill is the referent for "him") we can conclude that "he" refers to John.

Discourse Set 6 - John gave Bill the book.

Then he returned it.

By influences similar to those used in discourse set 5, we see that the only person who could return the book, is the current possessor of the book. So "he" must refer to Bill.

Discourse Set 7 - John gave Bill the book .

He returned it from last week.[1]

In this case we stay within the original time frame, so the second sentence is just an explanation of the first. When we try to find out who possesses the book, we therefore, use the past possessor, John.

Discourse Set 8 - John gave Bill the book.

He gave it to him as a gift.

Again, the absence of a word such as "then" leads one to believe that this is a restatement of the first sentence, so we resolve "he" with the original possessor of the book, John. "Him" refers to Bill because it can't refer to the same person as "he".

The final four discourse sets resolve the ambiguity of pronouns by using heuristic rules based on Grice's maxims of conversation (Levinson, 1983) to determine pronominal referents.

Discourse Set 9 - John asked Bill to leave.

He did.

Using feature matching alone, he could refer to either John or Bill. But Grice's maxim of Manner says to be as brief as possible. If "he" referred to John the sentence would only restate that John asked Bill to leave.[2] We therefore

[1] The point of this sentence would be clearer if we stated it as "He was returning it from last week", but we have not tackled the problems of representing past progressive tenses differently from simple past tenses.

[2] It is interesting to note how this interpretation is dependent on punctuation. If the sentence had been "He did?" one would conclude that the question was in fact restating the first question and "he" would then refer to John.

conclude that he refers to Bill since Bill's leaving is not explicitly stated in the first sentence.

Discourse Set 10 - John asked Bill to leave.

He didn't.

Another point of the Gricean maxim of Manner mentioned above is to not be ambiguous. If "he" referred to John, we would be contradicting what was stated in the first sentence. This rule appears to be even stronger than the rule employed in the previous discourse set, so we can fairly confidently conclude that "he" refers to "Bill".

Discourse Set 11 - John asked Bill to leave.

But he didnt want to.

The word "didn't" in the second sentence negates a specific event, unlike its use in the previous discourse set. So to resolve "he" we must first determine who wanted to do what. We can make cases for John not wanting to ask Bill to leave and for Bill not wanting to leave, so we are unable to resolve this pronoun, and the program will come to no conclusion. The resolution of referents in a case like this is dependent on additional knowledge about John and Bill which would presumably be available in episodic memory if these sentences appeared in a larger context. In the absence of such information, it is reasonable to hold an ambiguous interpretation pending further explanation.

Discourse Set 12 - John asked Bill to leave.

But he likes fish.

Grice's maxim of relevance states that a sentence should be relevant to the topic at hand. Since liking fish seems to have nothing to do with asking Bill to leave, the second sentence gives no clues as to how to resolve the ambiguity of the pronoun "he", so again the program will come to no conclusion.

3.0 ANAS ALGORITHM

ANA's pronoun resolution algorithm conceptual dependency representations generated by PLUM, goes through each event, and resolves any pronouns it finds filling the slots of those events. Pronouns are treated by PLUM as noun phrases which eventually become referents, so referents can either be nouns or pronouns. Pronouns carry the semantic features just like nouns, and semantic feature matching between pronouns and nouns is the first and simplest strategy employed by ANA for resolving pronouns. We did not try to build the pronoun resolution facility directly into PLUM because in almost all the cases where simple feature matching was not enough to determine an antecedent, the correct resolution of a pronoun required examination of the conceptual dependency events themselves to direct the resolution. It was therefore easier to wait until all events were known before trying to do anything with the pronouns.

ANA also required some global data structures which were not used by PLUM, and it was easier to manipulate these data structures outside of PLUM. These data structures often stored semantic information which is used when a pronoun has an ambiguous antecedent. The major global data structures required by ANA were the following:

> *referent-bag* - This is a list of noun referents encountered by the program when checking the slots of a conceptual dependency event. When ANA finds a pronoun, this list contains the possible values to which it can be resolved.

event-bag - A list of each of the conceptual dependency events created by a PLUM parse. Slot values in some of these events can be used to resolve ambiguous pronouns in some cases. For example, in the sentences "John gave Bill the book," and "He wanted to", the ATRANS for John giving Bill the book was placed in the event bag. When we tried to resolve the gapped referent using events in the event bag, this ATRANS event was our only possibility.

matched-referents - A list of pairs, each pair being a pronoun and the noun to which it refers.

unmatched-referents - A list of pronouns which could not be successfully resolved. This list and the previous list serve mainly to keep the program from trying to resolve the same pronouns over and over again.

possession-indicators - This is a list of triples indicating possession of objects by certain people at certain times. A triple is an object, a possessor, and a time frame. Knowledge of who possesses an object can aid in pronoun resolution for some sentences. For example, for the sentence "John gave Bill the book," we would generate the following possession indicators: (book_0 john_0 past) (book_0 bill_0 current). When ANA encounters the sentence "Then he returned it," and determines that "it" refers to the book, ANA then uses the current possession indicator to determine that "he" refers to Bill.

ANA's algorithm operates as follows:

First all the global data structures are initialized to nil. Then ANA loops through each sentence in the discourse set. A PLUM parse for each sentence is generated. Then ANA goes through each event in the *EVENT-MEM*, in the order in which they were created by PLUM. For each event it checks all slot values in the concept-frame. If a slot value is a referent, the program determines whether the referent is a noun or a pronoun. If the referent is a noun, it is added to the *referent-bag* (if it is not already there). If the referent is a pronoun and the pronoun does not appear in either the *matched-referents* or the

unmatched-referents, then the function called "resolve-pronoun" is invoked on that referent.

Resolve-pronoun first finds all the nouns in the *referent-bag* that share all the features of the pronoun. If there is only one noun with the same features as the pronoun, then there is no need to go on: the pronoun refers to that noun. If there is more than one noun with the same features, then resolve-pronoun determines which slot the pronoun is filling, and then branches to another function designed to resolve pronouns for that slot. For example, in the test discourse sets used, all the pronouns which reached this point were either in the actor or the source slots, so there are functions called "resolve-actor" and "resolve-source" for these respective slots. ANA calls different strategies for resolving pronouns in each of these slots, so separate functions are needed for each slot.

To resolve a pronoun in the actor slot, the algorithm proceeds according to the kind of conceptual dependency event which contains the actor slot with the pronoun. If the event is an ATRANS, the algorithm must know the object involved, recursively invoking the resolve-pronoun function if the object slot is filled with a pronoun. It must also know if this event took place after the events in the previous sentence. It determines this by looking in the *CONCEPT-MEM* for a "time-advance" prototype. Then the algorithm has the following options:

1. If the source slot is filled with the same value as the actor, and there has been a time advance, then the pronoun refers to the current possessor of the object. Otherwise the pronoun refers to the past possessor of the object.

2. If the source slot is filled with a value other than that of the actor, and if the value of the source slot is one of the nouns whose features match those of the pronoun, then we can eliminate that noun from consideration. If there is only one noun left to be considered, then the pronoun refers to that noun.

3. If there has been no time advance then the pronoun refers to the past owner of the object.

4. If there has been a time advance, then the pronoun refers to the current owner of the object.

If the event is a CONTROL type event, then the object slot refers to the event being controlled. If this slot has a value, then it is another conceptual dependency event, and the actor in that conceptual dependency event is to whom the pronoun refers. If the object slot is empty, then it is a gapped control verb and can refer to any of the events in the *event-bag*. The program finds the actors in all the events in the *event-bag*, and if there is only one, then the pronoun refers to that actor.

If the event is a CONFIRM or a NEGATE type event, where the event confirms or negates some other conceptual dependency event, and the object slot is empty, then we find all the actors in the events in the *event-bag*. If there is one, the pronoun refers to that actor. If there are more than one, then we find the actors in "controlled" events. If there is only one actor in the controlled events then the pronoun refers to that actor. This is the implementation of Grice's maxim of Manner, discussed earlier, which states that conversation should not be redundant or contradictory. If the object slot is not empty then the actor in the event in the object slot is to whom the pronoun refers.

To resolve an ambiguous pronoun in the source slot, the program must find the current possessor of the object in the object slot. The pronoun then refers to the object's owner.

When ANA has finished going through the slots of an event, it checks to see if the event is an ATRANs. If it is, then it calls a function called "process-possession-changes" to determine the transfer of possession of objects between the source and the recipient. Then the event is thrown in the *event-bag* and the next event in the *EVENT-MEM* is processed, until all the events have been taken care of.

4.0 DISCUSSION OF RESULTS

We were able to develop algorithms to handle many of the common problems which arise in resolving pronouns. Conceptual Dependency representations for sentences prove to be quite helpful in designing ANA since we could rely on certain slots always being in an event in the *EVENT-MEM*, regardless of the type of event. For example, ANA could use the value in the "act" slot of an event to direct the program when it encountered an ambiguous pronoun. Furthermore, ANA was able to classify the types of strategies needed for pronoun resolution by the various types of conceptual dependency events.

Probably the most difficult problem we encountered in developing this resolution algorithm was the problem of handling time. In our early passes at the algorithm we made the simple assumption that the events in one sentence would be followed by the events in the next sentence, so we could reason on the basis of this clear temporal progression. However, it soon became apparent that there are many exceptions to this rule. Often a sentence simply restates, or is a further explanation of an earlier sentence. As we showed with discourse sets 2-8, knowing whether or not the time frame has advanced can determine who or what a pronoun is referring to.

We handled this problem by creating a time advance prototype in PLUM. It was triggered by words like "then" and would simply fire into the *CONCEPT-MEM*. ANA then included a function to determine whether or not this prototype was in *CONCEPT-MEM*, and it reasoned accordingly. The execution

results show that this worked for the test discourse sets, but it still wasn't completely satisfactory, because the time advance has to be triggered explicitly in a sentence. There will be many times when we will need to advance the time implicitly.

For example, the discourse set containing "John gave Bill the book" and "Then he returned it." shows the word "Then" in the second sentence explicitly advancing the time frame for the events in the second sentence. From this information we reason that "he" refers to the current owner of the book, who is Bill and we resolve the pronoun. However, if we had omitted the word "Then" from the second sentence ANA would have assumed the same time frame as the first sentence and it would have reasoned that John, not Bill, was the possessor of the book. Thus the pronoun would have resolved to John. This is a mistake, since the time frame should advance even without being explicitly specified by a word such as "then". The problem is, there are some cases which implicitly entail a time frame advancement, and others which do not. A general solution for implicit time frame advancement was beyond the scope of this project. Although the importance of this type of informance is apparent, and points to the necessity for sophisticated inference mechanisms if we expect to handle problems in anaphora in any comprehensive sense.

5.0 REFERENCES

Levinson, S.C. (1983) Pragmatics, by Steven C. Levinson, Cambridge University Press. See pp 97 - 118 for a discussion of Grice's theory of implicature.

Rieger, C.J. (1975). "Conceptual Memory and Inference", in Conceptual Information Processing. (ed. R. Schank), North Holland Press.

```
; This file contains the access functions needed for my Plum project.

(:= *constraint-access*
 '( (word)
        (lambda (cand) (locate-filler cand '(word part-of-speech:)))
    (word item)
        (lambda (cand) (locate-filler cand '(word item: features)))
    (blackboard)
       (lambda (cand) (eval *blackboard*))
    (referent)
       (lambda (cand) (locate-filler cand '(referent token features)))
    (referent token features)
       (lambda (cand) (locate-filler cand '(referent token features)))
    (determiner)
       (lambda (cand) (locate-filler cand
                                     '(determiner value part-of-speech:)))
    (direction-from value)
       (lambda (cand) (locate-filler cand
                                     '(direction-from value token features)))
    (direction-to value)
       (lambda (cand) (locate-filler cand
                                     '(direction-to value token features)))
    (describe-as value)
       (lambda (cand) (locate-filler cand
                                     '(describe-as value token features)))
    (pn value)
       (lambda (cand) (locate-filler cand
                                     '(pn value)))
    (cd value)
       (lambda (cand) (locate-filler cand
                                     '(cd value)))
    (control-cd value)
       (lambda (cand) (locate-filler cand
                                     '(control-cd value)))
    (np)
       (lambda (cand) (locate-filler cand
                                    '(np)))
    (adjective-group modifying-adjectives)
       (lambda (cand) (locate-filler cand
                                    '(adjective-group modifying-adjectives)))
    (adjective-group last-entry)
       (lambda (cand) (locate-filler cand  '(adjective-group last-entry))) ))

(msg t "> ppacc.evl loaded.")
```

```
(:= *constraint-access*
 '( (word)
        (lambda (cand) (locate-filler cand '(word part-of-speech:)))
    (word item)
        (lambda (cand) (locate-filler cand '(word item: features)))
    (blackboard)
       (lambda (cand) (eval *blackboard*))
    (referent)
       (lambda (cand) (locate-filler cand '(referent token features)))
    (referent token features)
       (lambda (cand) (locate-filler cand '(referent token features)))
    (determiner)
       (lambda (cand) (locate-filler cand
                                     '(determiner value part-of-speech:)))
    (direction-from value)
       (lambda (cand) (locate-filler cand
                                     '(direction-from value token features)))
    (direction-to value)
       (lambda (cand) (locate-filler cand
                                     '(direction-to value token features)))
    (describe-as value)
       (lambda (cand) (locate-filler cand
                                     '(describe-as value token features)))
    (pn value)
       (lambda (cand) (locate-filler cand
                                     '(pn value)))
    (cd value)
       (lambda (cand) (locate-filler cand
                                     '(cd value)))
    (control-cd value)
       (lambda (cand) (locate-filler cand
                                     '(control-cd value)))
    (np)
       (lambda (cand) (locate-filler cand
                                     '(np)))
    (adjective-group modifying-adjectives)
       (lambda (cand) (locate-filler cand
                                     '(adjective-group modifying-adjectives)))
    (adjective-group last-entry)
       (lambda (cand) (locate-filler cand  '(adjective-group last-entry))) ))

(msg t "> ppacc.evl loaded.")
```

```
; This file contains the dictionary definitions needed for my Plum project.

(defword bought (verb) (bought1-ATRANS-prediction bought2-ATRANS-prediction)
        nil)

(defword likes (verb) (likes-MBUILD-prediction) nil)

(defword gave (verb) (gave-ATRANS-prediction) nil)
(syn gave (give))

(defword was (verb) (describe-object-prediction) nil)

(defword wanted (verb) (want-MBUILD-prediction) nil)
(syn wanted (want))

(defword read (verb) (read-ATRANS-prediction) nil)

(defword took (verb) (took-ATRANS-prediction) nil)

(defword returned (verb) (returned-ATRANS-prediction) nil)

(defword asked (verb) (asked-MBUILD-prediction) nil)

(defword leave (verb) (leave-PTRANS-prediction) nil)

(defword did (verb) (did-CONFIRM-prediction) nil)

(defword didnt (verb) (didnt-NEGATE-prediction) nil)

(defword john (noun) (np-terminator-prediction adjective-group-prediction)
        (animate male proper-noun))
(syn john (bill))

(defword honda (noun) (np-terminator-prediction adjective-group-prediction)
        (physobj))

(defword book (noun) (np-terminator-prediction adjective-group-prediction)
        (physobj))

(defword fish (noun) (np-terminator-prediction adjective-group-prediction)
        (animate))

(defword gift (noun) (np-terminator-prediction adjective-group-prediction)
        (physobj))

(defword week (noun) (np-terminator-prediction adjective-group-prediction)
        (time))

(defword he (pronoun) (pn-prediction) (animate male))
(syn he (him))

(defword it (pronoun) (pn-prediction) (physobj))

(defword a (article) (determiner-prediction) nil)
(syn a (the))

(defword big (adjective) nil nil)

(defword last (adjective) nil nil)

(defword first (adverb) nil (past-indicator))

(defword to (preposition) (direction-to-prediction) nil)

(defword from (preposition) (direction-from-prediction time-from-prediction)
        nil)

(defword as (preposition) (describe-as-prediction) nil)

(defword and (conjunction) nil nil)
(syn and (but))

(defword then (conjunction) (time-advance-prediction) nil)

(defword period (period) nil nil)

(msg t "> ppdict.evl loaded.")
```

```
(defword bought (verb) (bought1-ATRANS-prediction bought2-ATRANS-prediction)
        nil)

(defword likes (verb) (likes-MBUILD-prediction) nil)

(defword gave (verb) (gave-ATRANS-prediction) nil)
(syn gave (give))

(defword was (verb) (describe-object-prediction) nil)

(defword wanted (verb) (want-MBUILD-prediction) nil)
(syn wanted (want))

(defword read (verb) (read-ATRANS-prediction) nil)

(defword took (verb) (took-ATRANS-prediction) nil)

(defword returned (verb) (returned-ATRANS-prediction) nil)

(defword asked (verb) (asked-MBUILD-prediction) nil)

(defword leave (verb) (leave-PTRANS-prediction) nil)

(defword did (verb) (did-CONFIRM-prediction) nil)

(defword didnt (verb) (didnt-NEGATE-prediction) nil)

(defword john (noun) (np-terminator-prediction adjective-group-prediction)
        (animate male proper-noun))
(syn john (bill))

(defword honda (noun) (np-terminator-prediction adjective-group-prediction)
        (physobj))

(defword book (noun) (np-terminator-prediction adjective-group-prediction)
        (physobj))

(defword fish (noun) (np-terminator-prediction adjective-group-prediction)
        (animate))

(defword gift (noun) (np-terminator-prediction adjective-group-prediction)
        (physobj))

(defword week (noun) (np-terminator-prediction adjective-group-prediction)
        (time))

(defword he (pronoun) (pn-prediction) (animate male))
(syn he (him))

(defword it (pronoun) (pn-prediction) (physobj))

(defword a (article) (determiner-prediction) nil)
(syn a (the))

(defword big (adjective) nil nil)

(defword last (adjective) nil nil)

(defword first (adverb) nil (past-indicator))

(defword to (preposition) (direction-to-prediction) nil)

(defword from (preposition) (direction-from-prediction time-from-prediction)
        nil)

(defword as (preposition) (describe-as-prediction) nil)

(defword and (conjunction) nil nil)
(syn and (but))

(defword then (conjunction) (time-advance-prediction) nil)

(defword period (period) nil nil)

(msg t "> ppdict.evl loaded.")
```

```
; This file contains the prediction prototypes for my Plum project.

(create-pred type: (gave-ATRANS)
             comment: (this is triggered by an gave ATRANS verb)
             concept-frame: (actor -& (animate)
                              act = ATRANS
                              object = (physobj)
                              source = (same-as actor)
                              recipient = (animate)
                              description = (nil))
             control-structure: ((expect actor in past referent)
                                 (expect object in future referent)
                                 (expect recipient in future referent)
                                 (expect recipient in future direction-to
                                         value)
                                 (expect description in future describe-as
                                         value))
             predicts: (cd-prediction)
             required-slots: (actor object recipient)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (returned-ATRANS)
             comment: (this is triggered by the word returned)
             concept-frame: (actor = (animate)
                              act = ATRANS
                              object = (physobj)
                              source = (same-as actor)
                              recipient = (animate)
                              description = (nil))
             control-structure: ((expect actor in past referent)
                                 (expect object in future referent)
                                 (expect recipient in future referent)
                                 (expect description in future time-from
                                         value))
             predicts: (cd-prediction)
             required-slots: (actor object)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (took-ATRANS)
             comment: (this is triggered by a variant of the word take)
             concept-frame: (actor = (animate)
                              act = ATRANS
                              object = (physobj)
                              source = (animate)
                              recipient = (same-as actor))
             control-structure: ((expect actor in past referent)
                                 (expect object in future referent)
                                 (expect source in future
                                         direction-from value))
             predicts: (cd-prediction)
             required-slots: (actor object source)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (bought1-ATRANS)
              comment: (this is triggered by the word bought and represents
                       the object going from the seller to the buyer.)
              concept-frame: (actor -& (animate)
                               act = ATRANS
                               object -& (physobj)
                               source -& (animate)
                               recipient -& (same-as actor))
              control-structure: ((expect actor in past referent)
                                  (expect object in future referent))
              predicts: (cd-prediction)
              required-slots: (actor object)
              memory-level: (*EVENT-MEM*)
              priority: 3)

(create-pred type: (bought2-ATRANS)
              comment: (This is triggered by the word bought, and represents
                       the money going from the buyer to the seller.)
              concept-frame: (actor -& (animate)
                               act = ATRANS
                               object = MONEY
                               source -& (same-as actor)
                               recipient -& (animate))
              control-structure: ((expect actor in past referent))
              predicts: (cd-prediction)
              required-slots: (actor)
```

```
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (likes-MBUILD)
             comment: (this indicates someone likes something or someone)
             concept-frame: (actor =& (animate)
                              act = MBUILD
                              object =& (nil))
             control-structure: ((expect actor in past referent)
                                 (expect object in future referent))
             predicts: (cd-prediction)
             required-slots: (actor object)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (describe-object)
             comment: (this is triggered by some form of the verb "to be" and
                       will expect an adjective following.)
             concept-frame: (object =& (nil)
                              property =& (adjective))
             control-structure: ((expect object in past referent)
                                 (expect property in future word))
             required-slots: (object property)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (want-MBUILD)
             comment: (this is triggered by some form of the verb "to want")
             concept-frame: (actor =& (animate)
                              act = CONTROL
                              to-value = (preposition)
                              object = (nil))
             control-structure: ((expect actor in past referent)
                                 (expect to-value in next word)
                                 (expect object in future cd value))
             predicts: (control-cd-prediction)
             required-slots: (actor to-value)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (read-ATRANS)
             comment: (this is triggered by the verb read)
             concept-frame: (actor =& (animate)
                              act = ATRANS
                              object = (physobj)
                              time-period = (adverb))
             control-structure: ((expect actor in past referent)
                                 (expect object in future referent)
                                 (expect time-period in future word))
             predicts: (cd-prediction)
             required-slots: (actor object)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (asked-MBUILD)
             comment: (this is triggered by some form of the verb ask)
             concept-frame: (actor =& (animate)
                              act = CONTROL
                              to-value = (preposition)
                              object = (nil))
             control-structure: ((expect actor in past referent)
                                 (expect to-value in future word)
                                 (expect object in future cd value))
             predicts: (control-cd-prediction)
             required-slots: (actor to-value)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (leave-PTRANS)
             comment: (this is triggered by the word leave)
             concept-frame: (actor = (animate)
                              act = PTRANS
                              source = (location)
                              destination = (location))
             control-structure: ((expect actor in last referent)
                                 (expect source in future referent)
                                 (expect destination in future referent))
             predicts: (cd-prediction)
             required-slots: (actor)
             memory-level: (*EVENT-MEM*)
             priority: 3)

(create-pred type: (did-CONFIRM)
```

```
                comment: (this is triggered by the word did)
                concept-frame: (actor = (animate)
                                act = CONFIRM
                                object = (nil))
                control-structure: ((expect actor in past referent)
                                    (expect object in future cd value)
                                    (expect object in future control-cd value))
                required-slots: (actor)
                memory-level: (*EVENT-MEM*)
                priority: 3)

(create-pred type: (didnt-NEGATE)
                comment: (this is triggered by the word didnt)
                concept-frame: (actor = (animate)
                                act = NEGATE
                                object = (nil))
                control-structure: ((expect actor in past referent)
                                    (expect object in future cd value)
                                    (expect object in future control-cd value))
                required-slots: (actor)
                memory-level: (*EVENT-MEM*)
                priority: 3)

(create-pred type: (direction-from)
                comment: (this is triggered by the word from)
                concept-frame: (value = (animate or physobj)
                                orientation = FROM)
                control-structure: ((expect value in next referent))
                required-slots: (value)
                memory-level: (*CONCEPT-MEM*)
                priority: 3)

(create-pred type: (direction-to)
                comment: (this is triggered by the word to)
                concept-frame: (value = (animate or physobj)
                                orientation = TO)
                control-structure: ((expect value in next referent))
                required-slots: (value)
                memory-level: (*CONCEPT-MEM*)
                priority: 3)

(create-pred type: (time-from)
                comment: (this is triggered by the word from)
                concept-frame: (value = (time)
                                orientation = FROM)
                control-structure: ((expect value in next referent))
                required-slots: (value)
                memory-level: (*CONCEPT-MEM*)
                priority: 3)

(create-pred type: (describe-as)
                comment: (this is triggered by the word as)
                concept-frame: (value = (physobj)
                                orientation = AS)
                control-structure: ((expect value in next referent))
                required-slots: (value)
                memory-level: (*CONCEPT-MEM*)
                priority: 3)

(create-pred type: (time-advance)
                comment: (this is triggered by a word such as 'then)
                concept-frame: (value = (nil))
                control-structure: ((expect value in current word))
                memory-level: (*CONCEPT-MEM*)
                priority: 1)

(create-pred type: (pn)
                comment: (this is triggered by any pronoun)
                concept-frame: (value = (pronoun))
                control-structure: ((expect value in current word))
                predicts: (np-terminator-prediction)
                required-slots: (value)
                memory-level: (*SYNTAX-MEM*)
                priority: 1)

(create-pred type: (np)
                comment: (this is triggered by any noun-phrase terminator)
                concept-frame: (determiner = (article)
                                modifiers = (nil)
                                headnoun = (nil))
                control-structure: ((expect determiner in last determiner)
                                    (expect modifiers in last
                                          adjective-group modifying-adjectives)
```

```
                                (expect headnoun in last
                                        adjective-group last-entry)
                                (expect headnoun in last pn value))
                predicts: (referent-prediction)
                required-slots: (headnoun)
                memory-level: (*SYNTAX-MEM*)
                priority: 1)

(create-pred type: (np-terminator)
             comment: (this is triggered by any determiner or noun)
             concept-frame: (value = (conjunction or period or verb or
                                      adverb or article or preposition))
             control-structure: ((expect value in future word))
             predicts: (np-prediction)
             required-slots: (value)
              memory-level: (*SYNTAX-MEM*)
             priority: 1)

(create-pred type: (adjective-group)
             comment: (this is triggered by any adjective or noun)
             concept-frame: (modifying-adjectives = (adjective or noun)
                             last-entry = (noun))
             control-structure: ((do-first gather-adjectives on
                                           current adjective-group)
                                 (expect modifying-adjectives in
                                         current blackboard)
                                 (expect last-entry in current word))
             required-slots: (last-entry)
              memory-level: (*SYNTAX-MEM*)
             priority: 1)

(create-pred type: (determiner)
             comment; (this is triggered by any article)
             concept-frame: (value = (article))
             control-structure: ((expect value in current word))
             required-slots: (value)
              memory-level: (*SYNTAX-MEM*)
             priority: 1)

(create-pred type: (referent)
             comment: (this is triggered by any noun phrase)
             concept-frame: (token = (nil))
             control-structure: ((do-first memtok on last np)
                                 (expect token in current blackboard))
             required-slots: (token)
              memory-level: (*CONCEPT-MEM*)
             priority: 1)

(create-pred type: (to-trigger)
             comment: (this is triggered by the word to)
             concept-frame: (token = (nil))
             control-structure: ((expect token in current word))
             required-slots: (token)
             memory-level: (*CONCEPT-MEM*)
             priority: 2)

(create-pred type: (word)
             concept-frame: (item = nil
                             part-of-speech = nil)
             control-structure: ((expect item in current input-stream))
             predicts: nil
             required-slots: (item part-of-speech)
              memory-level: (*LEXICAL-MEM*)
             priority: 1)

(create-pred type: (cd)
             comment: (this is triggered by the activation of the standard
                       conceptual dependency prototypes)
             concept-frame: (value = (nil))
             control-structure: ((do-first find-event on current word)
                                 (expect value in current blackboard))
             required-slots: (value)
             memory-level: (*CONCEPT-MEM*)
             priority: 3)

(create-pred type: (control-cd)
             comment: (this is triggered by the activation of any control type
                       conceptual dependency representation)
             concept-frame: (value = (nil))
             control-structure: ((do-first find-event on current word)
                                 (expect value in current blackboard))
             required-slots: (value)
             memory-level: (*CONCEPT-MEM*)
```

```
            priority: 3)

(dex find-event (word)
     (:= *blackboard* (nth (eval *EVENT-MEM*) 3)))

(msg t "> ppprot.evl loaded.")
```

```
; This file calls the files with the definitions, prototypes and access
; functions for my plum project.

(msg t ">Loading Plum project files:")

(setq *counted-variables* '(c p status group))

(ndefun newsym (sym)
        (cond ((or (listp sym) (numberp sym))
                   (error '"NEWSYM only takes literal atoms"))
              (t (let (count (add1 (or (get sym 'sym-count) -1)))
                   (putprop sym count 'sym-count)
                   (make-atom (string-append sym "_" count))))))

(dex degenerate-sym (symbol)
     (let (underscore-position
            (locate-character (make-string symbol) (cvtn '_)))
          (cond
            ((greaterp underscore-position 0)
             (make-symbol
               (extract-substring
                 (make-string symbol)
                 1
                 (difference
                   (locate-character (make-string symbol) (cvtn '_)) 1))))
            (t
              symbol))))

(dex create-pred1 (type comment c-frame control predicts req priority)
     (cond (**startup-trace**
            (msg t "** function create-pred1 called **" t
                 "   creating prediction prototype for " type t t)))
     (let (stripped-frame (revise-frame c-frame))
          (find-slots stripped-frame req type)
          (add-counted-variable type)
          (set (build-atom type '-prediction)
               (prediction type comment (initial-frame stripped-frame)
               (create-demons c-frame stripped-frame control) predicts
               (execute-first control) req priority))))

(dfx defword (item index pred features)
     (putprop item (worddef index pred features) 'word-definition)
     (add-counted-variable item))

(dfx syn (item clones)
     (for (x in clones) (do
                          (add-counted-variable x)
                          (putprop x
                                   (get item 'word-definition)
                                   'word-definition))))

(dex add-counted-variable (temp-variable)
     (cond
       ((null (element? temp-variable *counted-variables*))
        (setq *counted-variables*
              (append *counted-variables* (list temp-variable))))))

(dex clear-counters ()
     (map-nil
       (lambda (temp-variable)
               (remprop temp-variable 'sym-count))
       *counted-variables*))

(mapcar '(ppacc ppdict ppprot ppfuncs pputils)
        '(nlambda (x) (eval-file x)))

(ptrace brief)

(setq discourse-set-1 '((john bought a honda period)
                        (he likes it period))
      discourse-set-2 '((john gave bill the book period)
                        (it was big period))
      discourse-set-3 '((john gave bill the book period)
                        (he wanted to period))
      discourse-set-4 '((john gave bill the book period)
                        (then he read it period))
      discourse-set-5 '((john gave bill the book period)
                        (but he read it first period))
      discourse-set-6 '((john gave bill the book period)
                        (then he took it from him period))
      discourse-set-7 '((john gave bill the book period)
                        (then he returned it period))
```

```
discourse-set-8 '((john gave bill the book period)
                  (he returned it from last week period))
discourse-set-9 '((john gave bill the book period)
                  (he gave it to him as a gift period))
discourse-set-10 '((john asked bill to leave period)
                   (he did period))
discourse-set-11 '((john asked bill to leave period)
                   (he didnt period))
discourse-set-12 '((john asked bill to leave period)
                   (but he didnt want to period))
discourse-set-13 '((john asked bill to leave period)
                   (but he likes fish period)))
```

```
; This file contains the functions needed to create [illegible]
; structures created by PLUM and resolve any pronouns which are referents in
; those structures.


; This is the main function for parsing and resolving pronouns in a set of
; sentences.
(dex parse-and-resolve-pronouns (sentence-list)
     (msg t "Parsing the following sentences and resolving the pronouns"
          t "in those sentences:" t)
     (mapcar sentence-list
             '(nlambda (x) (msg 2 x t)))
     (clear-counters)
     (setq *referent-bag* ()
           *event-bag* ()
           *matched-pronouns* ()
           *unmatched-pronouns* ()
           *possession-indicators* ())
     (loop (while sentence-list)
           (do
             (let (this-sentence (car sentence-list))
                  (msg t "Parsing this sentence: " this-sentence t)
                  (msg t (parse this-sentence) t)
                  (process-sentence-events)))
           (next sentence-list (cdr sentence-list))
           (result nil))))


; This function goes through each conceptual event generated in the parse of
; the sentence, and processes each referent. If the referent is a noun, it
; adds the noun to the *referent-bag*. If it is a pronoun, it calls functions
; to help resolve that pronoun.
(dex process-sentence-events ()
     (loop (initial this-event-cell (get-first-event-cell))
           (while this-event-cell)
           (do
             (setq *this-event* (get-event this-event-cell))
             (process-event-referents (cddr *this-event*))
             (process-possession-changes (cddr *this-event*))
             (add-element (get-event-name this-event-cell)
                          *event-bag*))
           (next this-event-cell (get-next-cell this-event-cell))
           (result nil))))


; This function processes the individual referents in a conceptual dependency
; event representation.
(dex process-event-referents (event-referents)
     (loop (while event-referents)
           (do
             (let (slot-name (car event-referents)
                             slot-value (cadr event-referents))
                  (process-individual-referent slot-name slot-value)))
           (next event-referents (cddr event-referents))
           (result nil))))


; This function processes a single slot value. First it determines if the slot
; value is a referent. If the referent is a noun, it adds that noun to the
; *referent-bag*. If it is a pronoun, the function calls another function to
; try and resolve that pronoun.
(dex process-individual-referent (slot-name slot-value)
     (cond
       ((greaterp
          (locate-substring (make-string slot-value) "referent") 0)
        (let (object-referred-to (get-referent-object slot-value))
             (cond
               ((noun? object-referred-to)
                (cond
                  ((null (element? object-referred-to *referent-bag*))
                   (msg t "Adding " object-referred-to " to *referent-bag*." t)
                   (add-element object-referred-to *referent-bag*))))
               ((and
                  (pronoun? object-referred-to)
                  (unresolved-pronoun? object-referred-to)
                  (not (unresolvable-pronoun? object-referred-to)))
                (resolve-pronoun slot-name object-referred-to)))))))


; This function attempts to resolve the pronoun. The first try is feature
; matching; if there is only one referent whose features match that of the
; pronoun, then the match is done. Otherwise, more involved methods are tried.
```

```
(dex resolve-pronoun (slot-name pronoun-name)
     (msg t "Resolving the pronoun of " pronoun-name " in the " slot-name
          " slot." t)
     (setq *feature-matches* ())
     (match-pronoun-noun-features pronoun-name)
     (cond
       ((equal (length *feature-matches*) 1)
        (msg t "The pronoun " pronoun-name " refers to "
             (car *feature-matches*) t 2 "because it is the only noun whose "
             "features match those of " pronoun-name "." t)
        (add-element
          (list pronoun-name (car *feature-matches*))
          *matched-pronouns*))
       (t
         (msg t "These are the nouns to which " pronoun-name " could refer: "
              *feature-matches* t)
         (cond
           ((equal slot-name 'actor)
            (resolve-actor pronoun-name))
           ((equal slot-name 'object)
            nil)
           ((equal slot-name 'source)
            (resolve-source pronoun-name))
           ((equal slot-name 'recipient)
            (resolve-recipient pronoun-name))))))


; This function checks each noun in the *referent-bag* and compares its
; features against those of the pronoun passed in as an argument. If there is a
; match the noun is added to the *feature-matches*.
(dex match-pronoun-noun-features (this-pronoun)
     (loop (initial temp-referents *referent-bag*
                    pronoun-features (get-word-features this-pronoun))
           (while temp-referents)
           (do
             (cond
               ((equal pronoun-features
                       (intersection
                         pronoun-features
                         (get-word-features (car temp-referents))))
                (add-element (car temp-referents) *feature-matches*))))
           (next temp-referents (cdr temp-referents))
           (result nil)))


; This function resolves a pronoun in the actor slot which has more than one
; possible resolution. The exact resolution strategy is determined by the
; filler of the ACT slot, which indicates the type of cd representation for
; this event.
(dex resolve-actor (pronoun-name)
     (let (temp-act
           (get-slot-filler 'act)
           actor-matches
           *feature-matches*
           resolved-actor
           nil)
          (cond
            ((or
               (equal temp-act 'ATRANS)
               (equal temp-act 'MBUILD))
             (setq resolved-actor
                   (resolve-ATRANS-MBUILD-actor actor-matches pronoun-name)))
            ((equal temp-act 'CONTROL)
             (setq resolved-actor
                   (resolve-CONTROL-actor actor-matches pronoun-name)))
            ((equal temp-act 'CONFIRM)
             (setq resolved-actor
                   (resolve-CONFIRM-actor actor-matches pronoun-name)))
            ((equal temp-act 'NEGATE)
             (setq resolved-actor
                   (resolve-NEGATE-actor actor-matches pronoun-name))))
          (cond
            (resolved-actor
              (add-element (list pronoun-name resolved-actor)
                           *matched-pronouns*))
            (t
              (add-element pronoun-name *unmatched-pronouns*))))))


; This function uses the object and what the program knows about who possessed
; that object (past and current) to resolve a pronoun for ATRANS and MBUILD
; events
(dex resolve-ATRANS-MBUILD-actor (actor-matches pronoun-name)
```

```
(let (temp-object
      (get-slot-filler 'object)
      temp-actor
      (get-slot-filler 'actor)
      temp-source
      (get-slot-filler 'source)
      time-indicator
      (cond
        ((advance-time?) 'current)
        (t 'past))
      resolved-actor
      nil)
     (cond
       (temp-object
         (let (actual-object (get-slot-value temp-object 'object))
              (cond
                (temp-source
                  (cond
                    ((and
                       (equal temp-actor temp-source)
                       (setq resolved-actor
                             (find-possessor actual-object
                                             time-indicator)))
                     (msg t "Since the actor and the source are the"
                          " same, the " time-indicator " possessor" t 2
                          "of " actual-object ", " resolved-actor
                          ", is who the pronoun " pronoun-name
                          " refers to." t))
                    ((not (equal temp-actor temp-source))
                     (let* (actual-source
                             (get-slot-value temp-source 'source)
                             actor-match-position
                             (length
                               (element? actual-source
                                         actor-matches)))
                           (cond
                             ((greaterp actor-match-position 0)
                              (setq actor-matches
                                    (delete-nth
                                      actor-matches
                                      (add1
                                        (difference
                                          (length actor-matches)
                                          actor-match-position)))))))
                     (cond
                       ((equal (length actor-matches) 1)
                        (setq resolved-actor (car actor-matches))
                        (msg t "By eliminating the " time-indicator 1
                             "owner of " actual-object " from contention,"
                             t 2 pronoun-name " must stand for "
                             resolved-actor t))
                       (t
                         (msg t "We could not eliminate the "
                              time-indicator " owner of " actual-object
                              " from contention," t 2 "so the pronoun "
                              pronoun-name " could refer to: "
                              actor-matches t))))
                    ((equal temp-actor temp-source)
                     (msg t "Since we do not know the " time-indicator
                          " possessor of " actual-object "," t 2
                          "we cannot resolve the pronoun " pronoun-name
                          "." t))))
                ((null temp-source)
                 (cond
                   ((setq resolved-actor
                          (find-possessor actual-object
                                          time-indicator))
                    (msg t "Since the sentence refers to the "
                         time-indicator " owner of "
                         actual-object "," t 2 "the pronoun "
                         pronoun-name " refers to "
                         resolved-actor "." t))
                   (t
                     (msg t "Since we cannot use the object "
                          actual-object " or the action in this "
                          "case," t 2 "we cannot resolve the "
                          "pronoun " pronoun-name t)))))))
       (t
         (msg t "Because this is an ATRANS, and there is no object,"
              t 2 "we cannot resolve this pronoun in this event."
              t)))
     resolved-actor))
```

```
; This function resolves the pronoun for a controlled event, by finding the
; actors in events which could be controlled.
(dex resolve-CONTROL-actor (actor-matches pronoun-name)
     (let (eligible-actors
           (find-actors-in-earlier-events *this-event* actor-matches)
           resolved-actor
           ())
          (cond
            ((equal (length eligible-actors) 1)
             (setq resolved-actor (car eligible-actors))
             (msg t "Since the actor must have been an actor in an earlier "
                  "event," t 2 "and the only actor in an earlier event is "
                  (car eligible-actors) ", the pronoun " pronoun-name
                  " refers to " (car eligible-actors) "." t))
            ((not (equal (length eligible-actors) 1))
             (msg t "The possible actors in earlier events who could be the"
                  t 2 "actor in this event are: " eligible-actors t)))
          resolved-actor))

; This function resolves the pronoun in the actor slot when the type of act is
; confirming a prior event.
(dex resolve-CONFIRM-actor (actor-matches pronoun-name)
     (let (eligible-actors
           (find-actors-in-earlier-events *this-event* actor-matches)
           resolved-actor
           nil)
          (cond
            ((greaterp (length eligible-actors) 1)
             (let (controlled-actors
                   (find-actors-in-controlled-events eligible-actors))
                  (cond
                    ((equal (length controlled-actors) 1)
                     (setq resolved-actor (car controlled-actors))
                     (msg t "Since we generally don't confirm what is already "
                          "explicitly stated," t 2 "and the only actor in an "
                          "implicitly stated event is " (car controlled-actors)
                          "."  t 2 "the pronoun " pronoun-name " refers to "
                          (car controlled-actors) "." t))
                    (t
                      (msg t "The pronoun " pronoun-name " could still refer "
                           "to the following nouns:" t 2
                           controlled-actors))))))
          resolved-actor))

; This function resolves the pronoun in the actor slot for an event which
; serves to negate an earlier event, either explicitly or implicitly stated.
(dex resolve-NEGATE-actor (actor-matches pronoun-name)
     (let (temp-object
           (get-slot-filler 'object)
           resolved-actor
           nil)
          (cond
            (temp-object
              (let (negated-event (eval temp-object))
                   (cond
                     ((and
                        (equal 'CONTROL
                               (cadr (element? 'act negated-event)))
                        (setq resolved-actor
                              (resolve-CONTROL-actor
                                actor-matches pronoun-name)))
                      (msg t "The actor in the event being negated is "
                           resolved-actor t 2 ", so the pronoun " pronoun-name
                           " refers to " resolved-actor "." t))
                     ((equal 'CONTROL
                             (cadr (element? 'act negated-event)))
                      (msg t "Since the negated event is a control event,"
                           t 2 "and we cannot determine the actor in the "
                           "controlled event," t 2 "we cannot resolve the "
                           "pronoun " pronoun-name "." t)))))
            ((null temp-object)
             (let (eligible-actors
                   (find-actors-in-earlier-events *this-event* actor-matches))
                  (cond
                    ((greaterp (length eligible-actors) 1)
                     (let (controlled-actors
                           (find-actors-in-controlled-events eligible-actors))
                          (cond
                            ((equal (length controlled-actors) 1)
                             (setq resolved-actor (car controlled-actors))
```

```
                                (msg t "Since we generally don't contradict what "
                                     "is already explicitly stated," t 2 "and "
                                     "the only actor in an implicitly stated "
                                     "event is " (car controlled-actors) "," t 2
                                     "the pronoun " pronoun-name " refers to "
                                     (car controlled-actors) "." t))
                               (t
                                 (msg t "The pronoun " pronoun-name " could "
                                      "still refer to one of the following nouns:"
                                      t 2 controlled-actors t)))))))))
          resolved-actor))


; This function resolves a pronoun in the source slot which could match more
; than one noun. It does this by showing the source to be the current or past
; possessor of the object in the cd representation from which the source arose.
(dex resolve-source (pronoun-name)
     (let* (temp-object
            (get-slot-filler 'object)
            actual-object
            (get-slot-value temp-object 'object)
            time-indicator
            (cond
              ((advance-time?) 'current)
              (t 'past))
            object-possessor
            (find-possessor actual-object time-indicator))
           (msg t "Since " object-possessor " is the " time-indicator
                " possessor of " actual-object "," t 2 pronoun-name
                " refers to " object-possessor t)
           (add-element
             (list pronoun-name object-possessor)
             *matched-pronouns*)))


; This function resolves a pronoun in the recipient slot which could match more
; than one noun. It does this by showing the source to be the past or current
; possessor of the object in the cd representation from which the recipient
; arose.
(dex resolve-recipient (pronoun-name)
     (let* (temp-object
            (get-slot-filler 'object)
            actual-object
            (get-slot-value temp-object 'object)
            time-indicator
            (cond
              ((advance-time?) 'past)
              (t 'current))
            object-possessor
            (find-possessor actual-object time-indicator))
           (msg t "Since " object-possessor " is the " time-indicator
                " possessor of " actual-object "," t 2 pronoun-name
                " refers to " object-possessor t)
           (add-element
             (list pronoun-name object-possessor)
             *matched-pronouns*)))


; This function returns a list of all the possible actors in any events which
; could be controlled by the event passed in.
(dex find-actors-in-earlier-events (temp-event possible-actors)
     (loop (initial eligible-actors ()
                    event-list *event-bag*)
           (while event-list)
           (do
             (let* (temp-actor
                    (cadr (element? 'actor (eval (car event-list))))
                    actor-name
                    (get-slot-value temp-actor 'actor))
                   (cond
                     ((element? actor-name possible-actors)
                      (add-element
                        (get-slot-value temp-actor 'actor)
                        eligible-actors)))))
           (next event-list (cdr event-list))
           (result eligible-actors)))


; This function finds all the possible actors which can be in controlled
; events, meaning those conceptual dependency events which are controlled,
; or dependent upon, some outside conceptual dependency event.
(dex find-actors-in-controlled-events (possible-actors)
     (loop (initial event-list *event-bag*
```

```
            controlled-actors ())
        (while event-list)
        (do
          (let (temp-event (eval (car event-list)))
               (cond
                 ((equal (cadr (element? 'act temp-event)) 'CONTROL)
                  (let* (controlled-event
                         (eval (cadr (element? 'object temp-event)))
                         controlled-event-actor
                         (cadr (element? 'actor controlled-event))
                         controlled-event-actor-name
                         (get-slot-value controlled-event-actor
                                         'actor))
                        (cond
                          ((element? controlled-event-actor-name
                                     possible-actors)
                           (add-element controlled-event-actor-name
                                        controlled-actors)))))))
        (next event-list (cdr event-list))
        (result controlled-actors)))


; This function records a change of possession of an object if the event is an
; ATRANS.
(dex process-possession-changes (slots-and-values)
     (let (act-list (element? 'act slots-and-values))
          (cond
            ((and act-list
                  (equal 'ATRANS (cadr act-list)))
             (let (temp-object
                    (cadr (element? 'object slots-and-values))
                    temp-source
                    (cadr (element? 'source slots-and-values))
                    temp-recipient
                    (cadr (element? 'recipient slots-and-values)))
                  (cond
                    ((and temp-object temp-source)
                     (add-possession-indicator
                       (get-slot-value temp-object 'object)
                       (get-slot-value temp-source 'source)
                       'past)))
                  (cond
                    ((and temp-object temp-recipient)
                     (add-possession-indicator
                       (get-slot-value temp-object 'object)
                       (get-slot-value temp-recipient 'recipient)
                       'current)))))))


; This function adds an entry to the *possession-indicators*, unless the entry
; already exists. If there is an entry with the same object and referent and a
; a different time value, that entry is deleted.
(dex add-possession-indicator (temp-object temp-referent time-frame)
     (cond
       ((null (element? (list temp-object temp-referent time-frame)
                        *possession-indicators*))
        (loop (initial possession-count 1)
              (while (not (greaterp possession-count
                                    (length *possession-indicators*))))
              (do
                (let (comp-time-frame
                       (opposite-time-frame time-frame)
                       this-possession-indicator
                       (nth *possession-indicators* possession-count))
                     (cond
                       ((equal
                          (list temp-object temp-referent comp-time-frame)
                          this-possession-indicator)
                        (setq *possession-indicators*
                              (delete-nth *possession-indicators*
                                          possession-count))))))
              (next possession-count (add1 possession-count))
              (result nil))
        (add-element
          (list temp-object temp-referent time-frame)
          *possession-indicators*))))


; This function returns 'past if passed the argument of 'current and vice
; versa.
(dex opposite-time-frame (time-frame)
     (cond
       ((equal time-frame 'current) 'past)
```

```
((equal time-frame 'past) 'current)))
```

```
; This file contains utilities and access functions for the PLUM project.


;This macro allows the user to destructively add an element to some list.
(macro add-element (body)
       `(setq ,(caddr body)
              (append ,(caddr body) (list ,(cadr body)))))


; This function gets the first conceptual dependency event produced in the
; parse of the sentence by PLUM.
(dex get-first-event-cell ()
     (loop (initial temp-event-cell (eval *EVENT-MEM*))
           (do
             (cond
               ((null (get-event temp-event-cell))
                (return (get-next-cell temp-event-cell))))))
           (next temp-event-cell (get-previous-cell temp-event-cell))))


;This function returns the event being stored in the memory cell.
(dex get-event (event-cell)
     (eval (nth event-cell 3)))


; This function returns the next event pointed to by a memory cell.
(dex get-next-cell (event-cell)
     (eval (nth event-cell 4)))


; This function returns the previous event pointed to by a memory cell.
(dex get-previous-cell (event-cell)
     (eval (nth event-cell 5)))


; This function returns the name of an event.
(dex get-event-name (event-cell)
     (nth event-cell 3))


; This function is a predicate indicating whether or not the word passed in is
; a pronoun.
(dex pronoun? (this-object)
     (element? 'pronoun
               (get-grammatical-type this-object)))


; This function is a predicate indicating whether or not the word passed in is
; a noun.
(dex noun? (this-object)
     (element? 'noun
               (get-grammatical-type this-object)))


; This function returns the grammatical type of the word passed in.
(dex get-grammatical-type (this-object)
     (car (get (degenerate-sym this-object) 'word-definition)))


; This function is a predicate indicating whether or not the pronoun passed in
; as an argument has been resolved yet.
(dex unresolved-pronoun? (pronoun-name)
     (null (element? pronoun-name
                     (mapcar *matched-pronouns* 'car))))


; This function gets the object being referred to by a referent.
(dex get-referent-object (this-referent)
     (cond
       ((boundp this-referent)
        (nth (eval this-referent) 4))
       (t this-referent)))


; This function gets the features associated with the word passed in as an
; argument.
(dex get-word-features (word-symbol)
     (nth (get (degenerate-sym word-symbol) 'word-definition) 3))


; This function returns the name associated with the slot name passed in for
; the current event. If there is no slot with that name for the current event
```

```
; the function returns nil.
(dex get-slot-filler (slot-name)
     (cadr (element? slot-name *this-event*)))


; This function takes an object and a time frame, and returns the owner
; of that object at that time.
(dex find-possessor (temp-object time-frame)
     (loop (initial temp-possessions *possession-indicators*
                    possessor nil)
           (while temp-possessions)
           (do
             (cond
               ((and
                  (equal temp-object (caar temp-possessions))
                  (equal time-frame (caddar temp-possessions)))
                (setq possessor (cadar temp-possessions)))))
           (next temp-possessions (cdr temp-possessions))
           (result possessor)))


; This function takes a pronoun name as an argument, and returns the pair in
; *matched-pronouns* whose first element is that pronoun.
(dex get-pronoun-noun-pair (pronoun-name)
     (let (sublist-length
            (length (element? pronoun-name
                              (mapcar *matched-pronouns* 'car))))
          (cond
            ((greaterp sublist-length 0)
             (nth *matched-pronouns*
                  (add1 (difference (length *matched-pronouns*)
                                    sublist-length))))
            (t ()))))


; This function gets the actual word associated with a variable whose name
; begins with "word".
(dex get-word-value (word-name)
     (cond
       (word-name
         (nth (eval word-name) 4))))


; This function gets the full slot value of a noun or pronoun.
(dex get-slot-value (value-name slot-name)
     (let (value-referred-to (get-referent-object value-name))
          (cond
            ((not (boundp value-name))
             value-name)
            ((noun? value-referred-to)
             value-referred-to)
            ((and
               (pronoun? value-referred-to)
               (unresolvable-pronoun? value-referred-to))
             value-referred-to)
            ((pronoun? value-referred-to)
             (cond
               ((unresolved-pronoun? value-referred-to)
                (resolve-pronoun slot-name value-referred-to)))
             (let (resolved-object
                    (get-pronoun-noun-pair value-referred-to))
                  (cond
                    (resolved-object (cadr resolved-object))
                    (t nil)))))))


; This function determines whether the program has already unsuccessfully tried
; to resolve the pronoun passed in as an argument.
(dex unresolvable-pronoun? (this-pronoun)
     (element? this-pronoun *unmatched-pronouns*))


; This function determines whether the time frame of the discourse should be
; advanced. It does this by looking through the *CONCEPT-MEM* for a
; time-advance prototype, which is triggered by words like "then".
(dex advance-time? ()
     (loop (initial temp-concept (eval *CONCEPT-MEM*)
                    advanced-flag nil)
           (while temp-concept)
           (do
             (cond
               ((equal (degenerate-sym (get-event-name temp-concept))
                       'time-advance)
```

```
        (setq advanced-flag t)
        (return advanced-flag))))
(next temp-concept (get-previous-cell temp-concept))
(result advanced-flag)))
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-1)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john bought a honda period)
  (he likes it period)

Parsing this sentence: (john bought a honda period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: bought
        - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

bought2-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = MONEY
                                        source = referent_0
                                        recipient = nil

        - - - - - - - - - - - - - - - -
Processing a new word: a
        - - - - - - - - - - - - - - - -
Processing a new word: honda
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

bought1-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_1
                                        source = nil
                                        recipient = referent_0


*Event-Mem* => c_20

Adding john_0 to *referent-bag*.

Adding honda_0 to *referent-bag*.

Parsing this sentence: (he likes it period)


        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: likes
        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

likes-MBUILD_0:
                                        actor = referent_2
                                        act = MBUILD
                                        object = referent_3


*Event-Mem* => c_38

Resolving the pronoun of he_0 in the actor slot.

The pronoun he_0 refers to john_0
  because it is the only noun whose features match those of he_0.

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to honda_0
  because it is the only noun whose features match those of it_0.
nil
```

```
CLisp:
(parse-and-resolve-pronouns discourse-ast-2)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (it was big period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil


*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (it was big period)


        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: was
        - - - - - - - - - - - - - - - -
Processing a new word: big
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

describe-object_0:
                                        object = referent_3
                                        property = word_8


        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_36

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
  because it is the only noun whose features match those of it_0.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-3)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (he wanted to period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil


*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (he wanted to period)


        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: wanted
        - - - - - - - - - - - - - - - -
Processing a new word: to
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

want-MBUILD_0:
                                        actor = referent_3
                                        act = CONTROL
                                        to-value = word_8
                                        object = nil


        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_36

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Since the actor must have been an actor in an earlier event,
  and the only actor in an earlier event is john_0, the pronoun he_0
 refers to john_0.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-4)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (then he read it period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil


*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (then he read it period)


        - - - - - - - - - - - - - - - -
Processing a new word: then
        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: read
        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

read-ATRANS_0:
                                        actor = referent_3
                                        act = ATRANS
                                        object = referent_4
                                        time-period = nil


*Event-Mem* => c_43

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
  because it is the only noun whose features match those of it_0.

Since the sentence refers to the current owner of book_0,
  the pronoun he_0 refers to bill_0.
nil
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-5)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (but he read it first period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil


*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (but he read it first period)


        - - - - - - - - - - - - - - - -
Processing a new word: but
        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: read
        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: first
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

read-ATRANS_0:
                                        actor = referent_3
                                        act = ATRANS
                                        object = referent_4
                                        time-period = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_42

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
  because it is the only noun whose features match those of it_0.

Since the sentence refers to the past owner of book_0,
  the pronoun he_0 refers to john_0.
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-6)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (then he took it from him period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil


*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (then he took it from him period)


        - - - - - - - - - - - - - - - -
Processing a new word: then
        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: took
        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: from
        - - - - - - - - - - - - - - - -
Processing a new word: him
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

took-ATRANS_0:
                                        actor = referent_3
                                        act = ATRANS
                                        object = referent_4
                                        source = referent_5
                                        recipient = referent_3


*Event-Mem* => c_50

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
  because it is the only noun whose features match those of it_0.
```

```
Resolving the pronoun of him_0 in the source slot.

These are the nouns to which him_0 could refer: (john_0 bill_0)

Since bill_0 is the current possessor of book_0,
  him_0 refers to bill_0

By eliminating the current owner of book_0 from contention,
  he_0 must stand for john_0
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-7)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (then he returned it period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil

*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (then he returned it period)


        - - - - - - - - - - - - - - - -
Processing a new word: then
        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: returned
        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

returned-ATRANS_0:
                                        actor = referent_3
                                        act = ATRANS
                                        object = referent_4
                                        source = referent_3
                                        recipient = nil
                                        description = nil


*Event-Mem* => c_43

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
  because it is the only noun whose features match those of it_0.

Since the actor and the source are the same, the current possessor
  of book_0, bill_0, is who the pronoun he_0 refers to.
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-8)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (he returned it from last week period)

Parsing this sentence: (john gave bill the book period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: gave
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: the
        - - - - - - - - - - - - - - - -
Processing a new word: book
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                        actor = referent_0
                                        act = ATRANS
                                        object = referent_2
                                        source = referent_0
                                        recipient = referent_1
                                        description = nil


*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (he returned it from last week period)


        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: returned
        - - - - - - - - - - - - - - - -
Processing a new word: it
        - - - - - - - - - - - - - - - -
Processing a new word: from
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

returned-ATRANS_0:
                                        actor = referent_3
                                        act = ATRANS
                                        object = referent_4
                                        source = referent_3
                                        recipient = nil
                                        description = nil


        - - - - - - - - - - - - - - - -
Processing a new word: last
        - - - - - - - - - - - - - - - -
Processing a new word: week
        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_41

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
```

```
  because it is the only noun whose features match those of it_0.

Since the actor and the source are the same, the past possessor
  of book_0, john_0, is who the pronoun he_0 refers to.

Adding week_0 to *referent-bag*.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-9)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john gave bill the book period)
  (he gave it to him as a gift period)

Parsing this sentence: (john gave bill the book period)


      - - - - - - - - - - - - - - - -
Processing a new word: john
      - - - - - - - - - - - - - - - -
Processing a new word: gave
      - - - - - - - - - - - - - - - -
Processing a new word: bill
      - - - - - - - - - - - - - - - -
Processing a new word: the
      - - - - - - - - - - - - - - - -
Processing a new word: book
      - - - - - - - - - - - - - - - -
Processing a new word: period
      - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_0:
                                    actor = referent_0
                                    act = ATRANS
                                    object = referent_2
                                    source = referent_0
                                    recipient = referent_1
                                    description = nil

*Event-Mem* => c_23

Adding john_0 to *referent-bag*.

Adding book_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (he gave it to him as a gift period)


      - - - - - - - - - - - - - - - -
Processing a new word: he
      - - - - - - - - - - - - - - - -
Processing a new word: gave
      - - - - - - - - - - - - - - - -
Processing a new word: it
      - - - - - - - - - - - - - - - -
Processing a new word: to
      - - - - - - - - - - - - - - - -
Processing a new word: him
      - - - - - - - - - - - - - - - -
Processing a new word: as
      - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

gave-ATRANS_1:
                                    actor = referent_3
                                    act = ATRANS
                                    object = referent_4
                                    source = referent_3
                                    recipient = referent_5
                                    description = nil

      - - - - - - - - - - - - - - - -
Processing a new word: a
      - - - - - - - - - - - - - - - -
Processing a new word: gift
      - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_47

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)
```

```
Resolving the pronoun of it_0 in the object slot.

The pronoun it_0 refers to book_0
  because it is the only noun whose features match those of it_0.

Since the actor and the source are the same, the past possessor
  of book_0, john_0, is who the pronoun he_0 refers to.

Resolving the pronoun of him_0 in the recipient slot.

These are the nouns to which him_0 could refer: (john_0 bill_0)

Since bill_0 is the current possessor of book_0,
  him_0 refers to bill_0

Adding gift_0 to *referent-bag*.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-10)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john asked bill to leave period)
  (he did period)

Parsing this sentence: (john asked bill to leave period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: asked
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: to
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

asked-MBUILD_0:
                                    actor = referent_0
                                    act = CONTROL
                                    to-value = word_3
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: leave
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

leave-PTRANS_0:
                                    actor = referent_1
                                    act = PTRANS
                                    source = nil
                                    destination = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_19

Adding john_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (he did period)


        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: did
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

did-CONFIRM_0:
                                    actor = referent_2
                                    act = CONFIRM
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_32

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Since we generally don't confirm what is already explicitly stated,
  and the only actor in an implicitly stated event is bill_0,
  the pronoun he_0 refers to bill_0.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-11)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john asked bill to leave period)
  (he didnt period)

Parsing this sentence: (john asked bill to leave period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: asked
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: to
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

asked-MBUILD_0:
                                    actor = referent_0
                                    act = CONTROL
                                    to-value = word_3
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: leave
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

leave-PTRANS_0:
                                    actor = referent_1
                                    act = PTRANS
                                    source = nil
                                    destination = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_19

Adding john_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (he didnt period)


        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: didnt
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

didnt-NEGATE_0:
                                    actor = referent_2
                                    act = NEGATE
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_32

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Since we generally don't contradict what is already explicitly stated,
  and the only actor in an implicitly stated event is bill_0,
  the pronoun he_0 refers to bill_0.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-12)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john asked bill to leave period)
  (but he didnt want to period)

Parsing this sentence: (john asked bill to leave period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: asked
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: to
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

asked-MBUILD_0:
                                    actor = referent_0
                                    act = CONTROL
                                    to-value = word_3
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: leave
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

leave-PTRANS_0:
                                    actor = referent_1
                                    act = PTRANS
                                    source = nil
                                    destination = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_19

Adding john_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (but he didnt want to period)


        - - - - - - - - - - - - - - - -
Processing a new word: but
        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: didnt
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

didnt-NEGATE_0:
                                    actor = referent_2
                                    act = NEGATE
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: want
        - - - - - - - - - - - - - - - -
Processing a new word: to
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

want-MBUILD_0:
                                    actor = referent_2
                                    act = CONTROL
                                    to-value = word_10
                                    object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period
```

```
*Event-Mem* => c_36

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

The possible actors in earlier events who could be the
  actor in this event are: (john_0 bill_0)

Since the negated event is a control event,
  and we cannot determine the actor in the controlled event,
  we cannot resolve the pronoun he_0.
nil

CLisp:
```

```
CLisp:
(parse-and-resolve-pronouns discourse-set-13)

Parsing the following sentences and resolving the pronouns
in those sentences:
  (john asked bill to leave period)
  (but he likes fish period)

Parsing this sentence: (john asked bill to leave period)


        - - - - - - - - - - - - - - - -
Processing a new word: john
        - - - - - - - - - - - - - - - -
Processing a new word: asked
        - - - - - - - - - - - - - - - -
Processing a new word: bill
        - - - - - - - - - - - - - - - -
Processing a new word: to
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

asked-MBUILD_0:
                                        actor = referent_0
                                        act = CONTROL
                                        to-value = word_3
                                        object = nil

        - - - - - - - - - - - - - - - -
Processing a new word: leave
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

leave-PTRANS_0:
                                        actor = referent_1
                                        act = PTRANS
                                        source = nil
                                        destination = nil

        - - - - - - - - - - - - - - - -
Processing a new word: period

*Event-Mem* => c_19

Adding john_0 to *referent-bag*.

Adding bill_0 to *referent-bag*.

Parsing this sentence: (but he likes fish period)


        - - - - - - - - - - - - - - - -
Processing a new word: but
        - - - - - - - - - - - - - - - -
Processing a new word: he
        - - - - - - - - - - - - - - - -
Processing a new word: likes
        - - - - - - - - - - - - - - - -
Processing a new word: fish
        - - - - - - - - - - - - - - - -
Processing a new word: period
        - - - - - - - - - - - - - - - -
Instantiating a new memory structure ...

likes-MBUILD_0:
                                        actor = referent_2
                                        act = MBUILD
                                        object = referent_3


*Event-Mem* => c_39

Resolving the pronoun of he_0 in the actor slot.

These are the nouns to which he_0 could refer: (john_0 bill_0)

Since we cannot use the object fish_0 or the action in this case,
  we cannot resolve the pronoun he_0

Adding fish_0 to *referent-bag*.
nil
```