The PLUM Users Manual

W. Lehnert and S. Rosenberg

COINS Technical Report 87-114

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.

TABLE OF CONTENTS

Table of Contents

1.0 INTRODUCTION

PLUM (the Predictive Language Understanding Mechanism) provides a control structure for integrated language analysis [Dyer 1983]. 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 content addressable, but does not impose constraints on what sorts of frame structures are to be used. 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.

While PLUM emphasizes the use of frame-oriented representation, its architecture does not prohibit the use of other analysis strategies. For example, it is possible to implement an augmented transition network in PLUM, although ATN's are not typically defined in terms of frames and slot-filling operations.

Within our own department of COINS, we see two different kinds of demand for natural language processing facilities. Some people want to concentrate on theoretical issues in natural language, while others only wish to design a natural language interface component for some project whose primary goals are not strictly linguistic. PLUM has been designed with these two distinct types of user populations in mind.

We want PLUM to serve as a vehicle for experimentation by students specializing in natural language processing. To this end, PLUM provides a basic foundation from which to base a variety of investigations, i.e. interactions between syntax and semantics, possible applications of connectionism, the coordination of multiple knowledge sources, etc. By using PLUM as a starting point for all such efforts, we hope to factor out a lot of the overhead work that would otherwise be duplicated across individual research projects.

We also want to use PLUM as the foundation for a variety of natural language processing applications ranging from database query applications to intelligent tutorial dialogues. In these projects the major thrust of the research effort is not on the natural language component of the system, yet a natural language facility is crucial for demonstrating the advantages of specific system designs. It is hoped that a research assistant will be able to design an adequate interface for these projects without having to devote all of his/her energy to that end.

The diverse demand for natural language processing capabilities within our own environment has given us a unique opportunity to evaluate the generality of PLUM and determine when it is and isn't providing a truly flexible base for various language investigations and interface projects. As our experience with PLUM increases we will continue to make refinements and alterations to the version described here.

2.0 AN OVERVIEW OF PLUM 3.0

Many of the design ideas behind PLUM were inspired by other conceptually-based parsers, most notably Riesbeck's ELI [Schank & Riesbeck, 1981] and Dyer's DYPAR [Dyer, 1983]. PLUM might be viewed as a direct descendent of DYPAR in the sense that extensive experience with DYPAR motivated many of the design decisions behind PLUM. For readers who know something about these systems, much of what follows will feel familiar. However, we will not presuppose any previous experience with DYPAR or any other language analyzer in describing PLUM. An occasional reference to DYPAR will be made when a clear contrast is appropriate, but our description of PLUM is otherwise self-contained.

2.1 Memory Organization

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 working memory representation being constructed will be *content-addressable* -- that is, memory structures will be placed on appropriate *type-lists* and accessed by addressing their *type*.

In addition to the working memory which is used by PLUM (mentioned above), an auxiliary memory representation is also employed to allow the designer to directly examine instantiated structures in memory. This auxiliary memory representation is 'layered' within various levels of memory. For example, a local construct concerning a pronominal referent will not be stored at the same level of auxiliary memory as a frame instantiation. Exactly what each layer of memory should contain is up to the designer, but we assume that the lowest levels of memory are used for relatively local phenomena within each sentence, while higher levels are used for 'macro-structures' like instantiated knowledge structures. Ideally, an arbitrary number of levels can be used, starting with level 1 (the lowest possible level). A set of four memory levels that is commonly used is:

1. LEXICAL-MEM,

2. SYNTAX-MEM,

3. CONCEPT-MEM, and

4. EVENT-MEM.

2.2 Memory Processes

Working memory is a collection of *type-lists*, linked lists which are indexed by the type of structures they contain. Before a structure can be added to memory, it has to first complete an *activate-instantiate* cycle [DeJong, 1979]. A potential memory structure is activated (henceforth, referred to as being triggered) when a prediction is made about that structure. Each structure specifies a fixed set of required slots which must then be filled before that structure is considered to have been instantiated. Only when a structure is instantiated can it be added to memory. If additional optional slots within that structure can be filled later, they will be added to the instantiated structure accordingly.

Each level of auxiliary memory is a linked structure. These lists are chronologically ordered so that structures instantiated earlier appear before structures instantiated later. All memory structures are associated with one level of memory and cannot migrate from level to level.

PLUM does not prohibit the development of memory management strategies; if a designer wants to experiment with a theory of forgetting, deletion operators can be introduced to manipulate PLUM's memory. However, these long-term phenomena do not appear to be tightly bound to the mechanisms of text comprehension which presumably function in a short-term context. We have therefore not included any basic forgetting features within PLUM. As far as PLUM's memory is concerned, everything is strictly cumulative. Once a structure has been added to memory, it stays there. This does not imply that all structures added to memory must be correct, however. It is possible to instantiate competing interpretations which will be resolved at a later time. For example, certain ambiguity problems will be handled with competing memory structures. But in all cases, only one structure will surface as the slot-filler for a more complex frame structure. We will not carry multiple slot-fillers around as a disjunctive set of choices, but we will allow multiple slot-filler candidates to appear in memory as independent entities.

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* [Schank & Riesbeck, 1981]. A predictive demon is associated with each slot that needs to be filled in a frame. These demons are instructed to search for a particular type of memory structure, and they are told where in memory to

look. Each search is restricted to a specific type-list, so we do not have to engage in the more exhaustive searches that DYPAR performed when it scanned its single-level list organization of working 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.

2.3 Prediction Prototypes

Every memory structure that PLUM constructs must be defined beforehand by a 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 prediction prototype which contains additional information about the frame structure that is useful during run-time. The prediction prototype 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 triggered if the current prediction succeeds in instantiating its concept-frame.

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

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 prediction prototype. The declarative nature of these definitions supports a potentially powerful debugging environment (the design monitor) which is discussed in a separate report [Johnson, 1985].

2.4 System Control

As more structures are added to memory, more and more predictions will be triggered, activating more and more demons. 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 demons may be pending. It is therefore necessary to impose some sort of control on these pending demons.

In DYPAR a simple queue was used to regulate the testing of pending demons. Each active demon assumed a location in the queue, to be removed from the queue by a variety of mechanisms. Processing the queue then meant testing each demon as it surfaced in the queue, until all the demons had been tested (and possibly executed). If any demon `fired' (tested positive), the entire queue would then be processed again, to see if any other demon could fire in response. When all of the demons were found to be in a quiescent state, control returned to a read routine that processed the next input word.

This general strategy is reminiscent of production system control mechanisms and has been used in other parsing systems as well [Thibadeau, Just, & Carpenter 1982]. It is an adequate approach, albeit somewhat lacking in efficiency and elegance. PLUM employs a similar mechanism, but distributes the load over multiple *waiting lists*. When a prediction is triggered, 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 a waiting list for the type of structure for which they are looking. Then, when a memory structure is instantiated, all demons on its waiting list are tested, thus avoiding testing demons that couldn't possibly fire.

PLUM's 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 triggered. Since the waiting demons are tested before any new predictions are triggered, timing problems are minimized.

2.5 Implementation

PLUM was first implemented in UMass Clisp which runs on a VAX 780 or VAX 750 under VMS. A major reimplementation was undertaken which is now running on the Symbolics 3600 in Zetalisp as well as Clisp (using the exact same code!). PLUM does not require closures or flavors.

The current implementation of PLUM is divided into eight groups of functions. They are the:

1. top-level driver,
2. dictionary defining functions,
3. search routines for examining working memory,
4. memory building functions,
5. prediction-instantiating functions,
6. prediction-defining functions,
7. demon-defining functions, and
8. general utilities.

In addition to these functions, a working version of PLUM also requires data files containing lexical definitions and prediction prototypes.

3.0 PREDICTION PROTOTYPES

A prediction prototype is a user defined data structure that defines a frame for memory along with a set of expectations that seek to fill the frame's empty slots. The terms "prediction" and "expectation" will be used to convey distinct concepts in the discussion that follows. A "prediction" refers to a prediction prototype, i.e., a frame prototype and all of the process-oriented information needed to create an instantiation of that frame. In fact, we will use the terms "prediction" and "prototype" interchangeably. On the other hand, an "expectation" refers to the specific processes responsible for filling a slot in some particular frame.

The user defines a prediction prototype by invoking a special function called create-pred. This function is designed to accept its arguments in any order, as long as they adhere to the general format <keyword1 arg1 keyword2 arg2 .... >. Create-pred recognizes seven legal keywords:

type	(required)
comment	(optional)
concept-frame	(required)
control-structure	(required)
predicts	(optional)
required-slots	(optional)
memory-level	(required)

The two most important parts of the prediction prototype are the concept-frame and the control-structure. The concept-frame defines the frame structure; its slot-names, and slot-constraints. The control-structure defines the expectations that operate to fill specific slots in the concept-frame. If either of these two arguments is omitted from the prototype definition, a message will be printed out:

"INADEQUATE INFORMATION GIVEN TO CREATE-PRED"

and create-pred will enter the break package. However, these are not the only arguments which create-pred requires. If the keywords type or memory-level are missing, the same error condition will apply.

The type argument supplies the prediction prototype with a name that can be used by lexical definitions and other prediction prototypes when a reference must be made to the prototype being defined. It also determines the *type-list* in working memory that the structure will be added to should it be successfully instantiated. The memory-level argument determines what level of auxiliary memory will receive the concept-frame.

In order for a concept-frame to be added to working memory, all of its required-slots must be filled by their associated expectations. To determine which slots in the concept-frame are required, we look at the required-slots argument. This should be a list of slot-names, each of which must appear in the concept-frame. If a required-slot is listed which does not appear in the concept-frame, a warning message will appear, but the prototype will be defined nevertheless. *It is important to redefine any such prototype, as it can never create instantiated memory structures under these circumstances.*

If a prototype omits the optional required-slots argument, the concept-frame will be instantiated whenever *any one* of its slots is filled. In this case, it is enough for just one of the associated expectations to succeed in its quest for a slot-filler. This facility is useful for any prototype that should automatically create a structure whenever it is triggered. An expectation can readily fill it's slot with some unconditional structure (like the last word), in which case the frame will always be added to working memory as soon as its prediction prototype is triggered.

We will use the term triggered when talking about prediction prototypes. A prediction is triggered in one of two ways:

1. a lexical definition can trigger a prediction prototype (see section 5), or

2. other predictions can trigger a prediction prototype by use of the predicts argument.

When a prototype is triggered, a copy of the prediction prototype is created, with its own working copy of the concept-frame, and the expectations needed to fill that frame. At the same time, all expectations defined in the prediction prototype are translated into internal data structures called *demons* (the demons are activated). These demons are then either tested immediately, or placed on an appropriate waiting list to be tested later (see section 4).

We will also speak about instantiating a concept-frame. A concept-frame is instantiated when all of the demons associated with its required-slots have been successful in filling their respective slots. As soon as all of the required-slots are filled, the concept-frame is placed on a type-list in working memory. (A type-list is a linked list of instantiated concept-frames containing only frames of the same type.)

The structure is also added to whatever level of auxiliary was specified by the memory-level argument in the prediction prototype definition. After the concept-frame is added to memory, any predictions specified by the predicts argument of its prediction prototype are triggered.

Once a frame has been instantiated, some of its demons may still be active (on a waiting list), seeking to fill optional slots in the instantiated concept-frame. If one should succeed, the new slot-filler is added to the already instantiated frame, but additional structures are not created in memory. A frame can only be instantiated once, although it may receive additional slot-fillers after it has been instantiated. Pending concept-frames will not be instantiated until all of their required-slots are filled. The most important distinction between instantiated concept-frames and pending concept-frames is that instantiated frames are part of memory – pending frames are not.

The comment argument in a prediction prototype definition is designed to receive a text string (in the form of a list) which can be used to print messages whenever that prototype is triggered. These messages are useful as part of the trace facility for keeping track of which prototypes are being triggered for what reasons.

The most complicated arguments in a prototype definition are the concept-frame and control-structure, so we will discuss these in detail. The example at the end of the report (section 8) will be useful in illustrating the features about to be described.

3.1 Concept-Frames

A concept-frame is defined by a list structure of the form:

```
(slot-name1 slot-type1 slot-constraints1
 slot-name2 slot-type2 slot-constraints2
 ..... etc .....)
```

A slot-name can be any Lisp atom. The slot-type must be one of three keywords; the atom "=", the atom "==", or the atom "=&" (quotation marks excluded). The first is used when the slot being defined is limited to one slot-filler. The second, when a constant list is used to fill a slot. The last is used when the slot can accept multiple slot-fillers. The slot-constraint can be either an atom or a list.

When an atom is used as a slot-constraint, we assume that the slot is always filled with a constant value corresponding to that atom. For example, if we want to define an *ATRANS* concept-frame, we can define the *act* slot:

act = ATRANS

in which case the *act* slot-filler will always be *ATRANS*.

Likewise, the slot can be filled with a list of constant values:

act == (ATRANS PTRANS)

If the slot-constraint is a list, that list will contain constraints that must be met by any candidate slot-filler before the slot can be filled. For example, in the *ATRANS* frame, we could define the *actor* slot:

actor =& (animate)

which tells us that the *actor* slot can assume multiple slot-fillers, but all of them must have the feature `animate´ associated them. In order for PLUM to know exactly how to determine whether or not such an association exists, we must precisely specify the slot-filler being searched for and its target-environment. This will be discussed in more detail in section 3.2.

It is possible to have multiple slot-constraints as long as they operate as disjunctive constraints. If we want our definition of *ATRANS* to work for banks and governments as well animate beings, we could say:

actor =& (animate or institution)

with the understanding that all appropriate slot-fillers will be either one or the other. A slot can have arbitrarily many slot-constraints, as long as each pair of constraints is separated by the keyword or.

actor =& (animate or institution or deity)

There is one other convention useful for defining slot-constraints. If a slot is expected to always assume a slot-filler that is identical to some other slot-filler in the same concept-frame, we can define the duplicate slot-filler with the keyword same-as followed by the name of the defining slot. So in our *ATRANS* frame, we might always assume that the *source* slot of the *ATRANS* is identical to the *actor* slot of the *ATRANS*:

```
(actor =& (animate)
 act = ATRANS
 object =& (physobj)
 source = (same-as actor)
 recipient =& (animate))
```

When a slot is being defined as a same-as slot, it doesn't matter which slot-type is used in that slot specification. The slot-type will be overridden by the slot-type associated with the defining slot. In this example, the *actor* slot is defined to take a multiple slot-filler, so the *source* slot is automatically assumed to take a multiple slot-filler as well, in spite of the "=" slot-type.

An arbitrary number of slots can be defined for any concept-frame, and there are no restrictions on the order in which those slots appear in the concept-frame definition.

In order for a concept-frame to be instantiated and added to memory, *it is necessary for at least one non-constant slot to be filled*, even if there are no required slots in the definition of the prediction prototype. So if you find yourself wanting to define a frame whose slots are all constants, be sure to include some sort of non-constant slot which can always be filled when the prototype is triggered. Without such a slot, your concept-frame will never be added to memory.

3.2 Control-Structures

There are four types of instructions that can be defined within the control-structure field of a prediction prototype; *expect-clauses, do-first-clauses, do-last-clauses,* and *insert-clauses*. Expect-clauses are used to specify search strategies; do-first-clauses and do-last-clauses are used for miscellaneous procedure specifications; and insert-clauses are used for bottom-up insertion.

3.2.1 Expect-Clause

In order to fill a slot in a concept-frame, we must provide some instructions to PLUM explaining where to look for likely slot-fillers. All slot-fillers must reside in working memory, so our directions will always refer to instantiated memory structures. We specify where to look in working memory by means of an expect-clause in the control-structure field of the prediction prototype definition. All expect-clauses conform to the following syntax:

(expect <slot-name> **in** <search-direction> <slot-filler>
[check <target-environment>]
[until <type>]
[after <slot-name>])
[1]

Expect and in are keywords which must always appear in the expect-clause. Check, until and after are also keywords and are optional.

The slot-name must be an atom which serves as a slot-name in the associated concept-frame definition. PLUM currently recognizes five keywords as valid search-directions:

any, future, past, next, last

The slot-filler is the type value of the memory structure. Since all memory structures of the same type reside on a single type-list, when PLUM searches for a structure in memory, it restricts its search to the type-list of the targeted memory structure. The search can then proceed in basically one of two ways:

1. We expect the slot-filler to be a structure that already exists in working memory.

2. We expect the slot-filler to be a structure that has not yet been instantiated.

The keywords past, and last are used if we are in case (1), while the keywords future and next are used to cover case (2). If the slot-filler could be found in either manner, we use the keyword any. More details will be provided on the exact search procedures associated with all of these keywords in section 4.

[1] The square-bracketed code is optional, depending on the slot-constraint.

The keywords last and next are used when it is appropriate to consider only one candidate memory structure of the targeted type. These keywords should therefore be used only when the associated slot is restricted to a single slot-filler. If a slot is designed to take multiple slot-fillers, its associated expect-clause should use only the keywords past, future, or any.

It is possible to have more than one expect-clause associated with a given slot-name. When a slot has multiple expect-clauses, they specify competing search strategies. In the event that one of the expect-clauses succeeds in locating a valid slot-filler, the demons associated with all other competing expect-clauses will die. *This happens even if the slot is defined to take multiple slot-fillers.*

There are two other keywords which may appear in expect-clauses; until and after. Until specifies a terminating condition for a search – search until a specified memory structure is found. For example:

> (**expect** actor **in past** referent check referent token features **until** direction-to)

while after specifies a condition for beginning a search – begin searching after a specified slot is filled. As in:

> (**expect** recipient **in future** referent check referent token features **after** actor)

The target-environment is a path specification within some memory structure. It is used to specify the constraints to be checked when searching for a slot-filler. We will postpone the discussion of complex target-environments until section 4.

3.2.2 Do-First-Clauses

Another facility available to us within a control-structure specification is the do-first-clause. This is one of two places in PLUM where a user is allowed to insert arbitrary Lisp code for the sake of modifying memory or creating side-effects. We have not yet arrived at any general guidelines for the use of do-first-clauses, but we can describe one usage that appears to have some general applicability.

The syntax for a do-first-clause is usually:

(**do-first** <function-name> **on** <search-direction> <memory-structure>)

where function-name is the name of a function of one argument, search-direction is one of the keywords in expect-clauses (with the same restrictions), and memory-structure is the type value of a memory frame. The user must then define the function named in the do-first-clause.

There can be only one do-first-clause in each control-structure specification (one per prototype definition). If more than one do-first-clause is included, no error will result, but only the first one will be recognized. The do-first-clause is then executed each time the prediction prototype is triggered, before any of its demons are tested.[2]

The user-defined function in a do-first-clause must be defined to accept a concept-frame as the value of its argument. If the lambda variable inside your function is called *c-frame*, for example, then *c-frame* will be bound to a list structure of slot-names and slot-fillers. Which concept-frame actually gets returned as a value to the function is determined by your search-direction and memory-structure specifications, and the state of working memory at the time the do-first-clause is executed.

If no argument is found for the do-first-clause function, then no function execution will take place. It is therefore useless to define a do-first-clause with the keywords next or future, as no such structure can be located at the time the do-first-clause is executed (a do-first-clause can only execute once, it cannot wait around for delayed execution like a demon). PLUM will not flag an error if these keywords are used for a do-first-clause, although you may see a runt time trace message of the form:

"WARNING: SEARCH FOR DO-FIRST PARAM FAILED"

depending on what trace mode is running (see section 7). A slightly different trace message will always appear at the time create-pred is executed if a do-first-clause is missing the keyword on. In that case, you should see:

---

[2] The function *execute-first* builds the function call specified by the do-first-clause when a prediction prototype is first defined by create-pred. The function *run-first* interprets the argument specification for a do-first-clause at the time of execution and executes the function. Note that the argument must be interpreted at the time of execution since we are passing an instantiated concept-frame that can only be accessed at the time of function execution.

"WARNING: NO ARGUMENT FOUND IN DO-FIRST CLAUSE"

In some cases it is appropriate to use a slight variation of the syntax described above. It is also possible to use a do-first-clause of the form:

(do-first <function-name> **on** <memory-level>)

where memory-level is one of the memory-level names used in the prediction prototype definitions. If a do-first-clause is defined in this way, the argument passed to the function will be the entire memory level specified. A do-first-clause defined in this manner can only fail to find its argument in the event that no frames have been added to the specified memory level.

If a user wants to execute a function that receives no input argument, it is best to define the argument in terms of a non-empty type-list, and simply ignore this argument in the function definition.

We have occasionally found it useful to use do-first-clauses for the purpose of passing information onto a blackboard. This blackboard can then be accessed by demons defined in expect-clauses. At the moment, PLUM is designed to recognize a global variable blackboard which is a simple Lisp atom. If a do-first-clause function assigns a value to blackboard, this value can then be accessed during subsequent demon execution by any demon whose search-direction and target-environment have been defined with the keywords current blackboard. Thus, there are actually six legal search-directions, not five. *However, this is the only way in which the search-direction* current *may be used in an expect-clause*.

For example, section 8 we define the prototype for a referent:

```
(create-pred type (referent)
      comment (this is activated 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*))
```

A simple form of *memtok* might simply take the head noun of the last noun phrase, retrieve a *gensym* of that atom, and store the *gensym* in the global variable blackboard. The expect-clause then locates its token in blackboard.

3.2.3 Do-Last-Clauses

Do-last-clauses are identical to do-first-clauses except that instead of being executed as soon as the frame is triggered, they are executed immediately after the frame is instantiated. As with the do-first-clause, the do-last-clause may appear anywhere in the control-structure and only one do-last-clause will be executed per control-structure. The proper syntax is:

(do-last <function-name> **on** <search-direction> <memory-structure>)
or
(do-last <function-name> **on** <memory-level>)

3.2.4 Insert-Clauses

The final type of instruction which may appear in the control-structure of a prediction prototype definition is the insert-clause. The insert-clause facilitates the bottom-up insertion of slots, along with values, into instantiated or non-instantiated concept-frames. Consequently, it is not necessary to attempt to anticipate every conceivable slot that might be useful in a prototype prediction when defining it. If useful information, for which there is no appropriate slot in the concept-frame, is encountered, a new slot may be inserted "from the bottom, up." An insert-clause looks like this:

(insert current (<slot-name>) **in** <search-direction> <memory-structure>)

A good example of the use of an insert-clause would be to insert a time slot into an *ATRANS* frame. Consider parsing the sentence:

"Johnny gave Mary the book in 1982."

where "1982' triggers the prediction prototype:

```
(create-pred type (time)
   concept-frame   (value = (year))
   control-structure   ((expect value in last word)
                        (insert current (time) in last atrans))
   required-slots   (value)
          .
          .
          .    )
```

Like the expect-clause, the insert-clause may appear anywhere in the control-structure and there is no limit on how many there are.

4.0 MEMORY MANAGEMENT

When a demon is tested, it invokes a search routine (*search-wm*) that determines what type-list in memory must be searched and in what direction the search should go. This information is found in the demon's internal representation. The demon also knows what it is looking for and what constraints may be imposed on its target. The object being sought is specified by the demon's expect-clause, and its constraining features are found in the concept-frame definition. To show how it all goes together, we'll step through a concrete example.

Consider the *recipient* slot of an *ATRANS* prediction. Within the concept-frame we see:

recipient =& (animate)

which tells us that the *recipient* slot can be filled by an arbitrary number of slot-fillers, and that each slot-filler must be associated with the feature `animate.' There are two competing demons that work to fill this slot:

E1: (**expect** recipient **in future** referent
check referent token features)
E2: (**expect** recipient **in next** direction-to
check value token features)

Whenever more than one demon is seeking to fill a slot, PLUM makes sure that they all share a common status value. That way, as soon as one demon is successful in its quest, all the other competing demons will die off. The *recipient* slot is associated with two such competing demons, E1 and E2.

In each expect-clause, PLUM recognizes a slot-name (recipient), a search-direction (future, next), a target-environment (referent token features, value token features), and a slot-filler (referent, direction-to). The slot-filler determines what memory structure will be used to fill the slot if the search is successful. When the target-environment is simple (one atom), the target is a memory structure of the type named.

When the target-environment is more complex (more than one atom), it's necessary to path through some intermediate structures to find the target structure. The first keyword in the target-environment tells the search function what structure we need to find as a top-level memory structure. In the case of E2, *search-wm* is instructed to look for a *direction-to* structure. Once a *direction-to* structure is

located, the expect-clause says to go inside the *value* slot of that structure. To see exactly what kinds of things qualify as the *value* slot-fillers inside *direction-to* structures, we have to look at the prediction prototype for a *direction-to* frame:

```
(create-pred type (direction-to)
        ....

        concept-frame (value = (location or physobj or animate)
                       orientation = to)
        control-structure ((expect value in next referent check
                            referent token features))

        ....
                                )
```

This tells us that any structure we find inside the *direction-to value* slot will be a referent, which is also marked as a location, physical object or animate object. But if the same slot-filler is going to succeed as an *ATRANS recipient*, the *ATRANS* frame requires an `animate' referent, so it is not the case that all *direction-to values* will qualify as *ATRANS recipients*.

If the search function cannot locate the top-level structure in working memory, the demon fails. But suppose the E2 demon does find a *direction-to* structure. Now it must test the structure to determine whether or not the particular referent in the *value* slot has been characterized as an animate.

Continuing along the path specified in the expect-clause, we now look inside the *token* slot. We can examine the referent prediction prototype to confirm that it has a *token* slot:

```
(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*))
```

To see that structures going into that *token* slot must themselves have *features* slots (as the path prescribed in the expect-clause continues), we have to know what the internal organization of a *memory token* is.

As it turns out, all *memory tokens* have two slots:[1]

1. a *type* slot that is filled with the constant identifier *memory-token*, and
2. a *features* slot that is filled with a list of semantic features extracted from lexical definitions and optional modifiers picked up from modifying adjectives.

(This is a rather impoverished notion of a memory token, but it is adequate for at least some applications.)

Once the search function has pathed its way down into the *features* slot of the token slot-filler inside the referent structure, it is ready to compare what it finds there against the constraints specified in the concept-frame. In this case, it needs to find the identifier `animate´ in the features list. If it does, the referent in question will be added to the recipient slot of the *ATRANS* frame. If it doesn't, the search continues in accordance with the search-direction that was specified (see section 4.1).

4.1 Search-Direction Keywords

As we saw in section 3, there are six keywords that can be used to specify a search direction.

next, last, future, past, any, current

These can be divided into two basic categories, forward-searching, and backward-searching.

4.1.1 Forward-searching

When a demon is activated that uses the keywords next or future in its expect-clause, we expect that the memory structure it is searching for has not yet been instantiated. It is therefore placed on a waiting-list, and tested when a frame of that particular type is instantiated.

---

[1] See PLUM source code for implementation details.

When the keyword next is used, the demon will only be tested once, this occurs at the next instantiation of the sought memory structure. As soon as an instance of this structure is encountered, it is checked according to the slot constraints specified. If it passes the constraints it fills the specified slot. However, if the structure fails, the demon will die and no further memory searches will take place. Therefore, a demon with a search-direction of next, expects to be satisfied with the first instance it finds of the specified target structure. Due to the restrictive cut-off in this search specification, it is not appropriate to use this keyword if you are trying to fill a slot that admits multiple slot-fillers.

When a search uses the keyword future, we engage in a search that moves forward in time indefinitely - that is, we are not restricted to the first instantiation of a target memory structure. If the slot we are trying to fill takes a single slot-filler, the demon will remain active as long as it takes to find a memory structure that satisfies the target-environment and slot constraints. It will die only after the slot is filled. If the associated slot takes multiple slot-fillers, the demon will live forever, continuing to search for more and more slot-fillers in all future memory structures. Unless, of course, the keyword until appears in the expect-clause. If this is the case, the demon will die if the specified slot-name is encountered.

4.1.2 Backward-searching

When a demon is activated that uses the keywords last or past in its expect-clause, we expect that the memory structure it is searching for is already instantiated. These demons can, therefore, be tested immediately, starting with the most recently instantiated element on the appropriate type-list. If no structure exists (the list is empty), the demon dies immediately, regardless of which keyword is used.

The keyword last is analogous to the keyword next. If the type-list specified is not empty and the keyword last is used, then the demon will be tested just once, on the most recently instantiated element. If it fails because of slot-constraints, the demon dies. This keyword should not be used for slots that take multiple slot-fillers.

The keyword past is analogous to future in the same way that last is analogous to next. A past search moves backwards in time (through the type-list), looking for memory structures that satisfy the concept-frame's slot-constraints. If the slot it is trying to fill takes only a single slot-filler, the demon dies when one such slot-filler is found. If the slot accepts multiple slot-fillers, the demon continues backward through memory testing all available memory structures as slot-filling candidates. However, there is an additional feature that distinguishes past from future. All demons using the past keyword necessarily die when they exhaust the type-list they are searching, regardless of whether or not they have succeeded in filling their slots.

4.1.3 Special Cases

Finally, there are two special cases, the keywords any and current.

The keyword any is used for searches that should go back in time and then forward. Demons that use this keyword are treated initially as though they had used the keyword past. If after testing all existing memory structures on the appropriate type-list they have still not completed their slot-filling responsibilities, they are treated as though they had used the keyword future and are placed on the appropriate waiting-list.

The keyword current may only be used in referencing the global variable blackboard. It referes to whatever value blackboard is bound to at the time the demon is tested. (See section 3.2.2 for discussion of do-first-clauses.)

****

It is possible to define a prediction prototype whose expect-clauses are not defined properly for slots that take multiple slot-fillers. That is, you can set up a slot in your concept-frame for multiple slot-fillers and then define an expect-clause for that slot that uses the keywords next or last. PLUM will notice the discrepancy and print out a warning message at the time of prototype definition:

"WARNING: MULTIPLE SLOT FILLERS CANNOT BE OBTAINED USING ONE OF THE EXPECT-CLAUSES SPECIFIED FOR THE <SLOT-NAME> SLOT"

If this warning message is generated, PLUM will then go on to internally encode the expect-clause just as it would for any other next or last demon, and it will function with those search specifications. So the demon will function insofar as it may find one slot-filler, but it will not be capable of providing more than one slot-filler.

4.2 Waiting List Processing

When a concept-frame's required-slots are filled, it is instantiated and added to the appropriate type-list in working memory, and the appropriate level of auxiliary memory. Then any demons on the waiting for that type are tested. For example, if a type *X1* frame is instantiated, it is added to the *X1*-type-list and then the demons on the *X1*-waiting-list are tested.

Each demon on the waiting-list is tested against the newly instantiated memory structure in turn. If it succeeds, it fills the appropriate slot in the concept-frame with which it is associated, as discussed above.

Once all of the demons on the waiting-list have been tested, any predictions associated with the newly instantiated memory structure (any prototypes listed in the predicts slot of the newly instantiated frame) are triggered. This in turn causes the activation of all of the demons belonging to the newly triggered prediction, which are either tested immediately or put on waiting-lists (see previous section).

If at any time a demon succeeds in filling the last unfilled required-slot of a concept-frame, the sequence described above is interrupted and the new frame is immediately instantiated, added to memory, and all of the demons on *its* waiting-list are tested (with thte possibility that one of these demons will fill the last unfilled required-slot of some other frame, causing an interruption of *this* sequence, etc.).

When all demons are in a quiescent state, the next word is read in from the input stream.

5.0 THE LEXICON AND DICTIONARY ENTRIES

Every word in PLUM's dictionary must be defined with the special function defword:

(defword word (indices) (predictions) (conceptual-features))

The first entry to defword is the lexical item being defined. PLUM does not incorporate a morphology package for stripping roots and interpreting suffixes, so any facility required along these lines must be supplied by the system designer.

The second entry to defword is a list of prediction indices that the analyzer will check when testing its predictions. These indexing features can be either syntactic (for traditional grammars) or semantic (for semantic grammars), but they should be features associated with word senses rather than their resulting conceptual structures.

The third field in the defword function definition is a list of prediction prototypes that should be triggered by the word. An arbitrary number of predictions can be included here, and in the case of words with multiple word senses, we should include all competing predictions. Run-time procedures will be responsible for the disambiguation of multiple word senses, so we need not try to include any information about those choices here. Additional information about word sense determination will be included in the prediction prototypes themselves. Note that all entries to the predicting-features list must end with the suffix *-prediction* even though the type value of the corresponding prediction prototype does not include this suffix.

The final field in the defword structure is a list of conceptual features that will show up in any memory tokens associated with the word. PLUM 3.0 is currently using this field only for words that reference memory structures.

For example, in:

(defword john (noun) (adjective-group-prediction np-terminator-prediction)
(animate proper-noun location))

john is indexed as a noun which will trigger predictions for an *adjective-group* and an *np-terminator*, and will be represented in memory as an *animate*, a *proper noun*, and a *location*.

When two words should be treated as being synonymous, the function syn is used. For example:

```
(syn john (mary bill))
```

assumes that *john* has already been defined via defword. It then processes all words listed in its second argument so that they will be treated the same as *john*.

If you wish to include a phrasal lexicon, PLUM allows you to define phrases consisting of *two words* via the function phrase:

```
(phrase ((water bed) (disk drive) (dog house))
```

which will create the lexical entries water-bed, disk-drive, and dog-house. The user must then be careful to create defword definitions for these three new lexical items. Any strings containing these sequences in the input stream will be read as a string containing the corresponding hyphenated forms.

6.0 INTERNAL DATA STRUCTURES & LISTS

6.1 Data Structures

PLUM uses the *defstruct* record package to maintain internal data structures related to demons and memory structures. It is useful to have some familiarity with these internal record formats during debugging.

6.1.1 Demons

The most frequently encountered structure is the demon spawned by an expect-clause in a prediction prototype. The demon structure is:

(spawning-prediction slot-name slot-constraint predicted-environment
predicted-filler search-direction also-fills life-expectancy
status killers self-pointer)

The slots for *spawning-prediction* and *status* are filled with pointers to internal structures and memory structures. The *status* pointer assumes the value t when the demon is alive. This value changes to nil when the demon dies. It is possible to find a dead demon on a waiting list, any such demon will be removed from the waiting list the next time the waiting list is run. If a demon is dead, it will not be tested again, nor can it be revived.

The *also-fills* slot is bound to a list of slots that the demon also fills, which would be used in the event of another slot in the associate concept-frame being filled with the keyword same-as.

The *killers* slot is bound to a list of memory types that will kill this demon if instantiated. This is for use with the keyword until.

Once a demon is created, most of its field values are fixed for life One exception is a demon defined with the keyword any. If a demon has been spawned from an expect-clause with the keyword any, the search-direction will start out with the value any and then change to the value future after its memory search has exhausted all existing memory structures. The life-expectancy for an any demon will also change from `wait-for-one´ (if it´s a single slot-filler), to `repeat´ at the same time that its search-direction changes.

To see how expect-clauses correspond to search-direction values and life-expectancy values, look at the functions *select-time* and *select-life-expectancy*.

6.1.2 Prediction Prototypes & Instantiated Frames

Prediction prototypes are described in detail in section 3 so there is no need to say more about them here. An example of a prediction prototype might be:

```
(create-pred type (referent)
      comment (this is activated 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*))
```

Instantiated frames are nothing more than disembodied property lists which correspond to concept-frames (see section 3.1). They are destructively modified when a slot is filled, and might look like this:

```
(type slot1 filler1
      slot2 filler2

      ..........)
```

****

If you wish to understand the inner workings of PLUM, it is useful to track the creation of these various structures during the parse of a sample sentence. However, most users should be able to operate PLUM effectively without this level of understanding.

6.2 Lists

There are three different kinds of lists used in PLUM 3.0. They are:

1. *waiting-lists*
2. *type-lists*
3. *memory-level-lists*

Each allows the user to access different pieces of information from within the system.

Waiting-lists are a linked list of activated demons. To find out what demons are waiting for a np, for example, enter:

(get np 'waiting-list)

Type-lists are a linked lists of instantiated concept-frames. To find out what memory structures had already been instantiated, a special function get-instances is available. For example, to find out what np's had already been instantiated, enter:

(get-instances 'np)

Finally, there are memory-level-lists. These are also linked lists of instantiated concept-frames. These lists are chronologically ordered so that structures instantiated earlier appear before structures instantiated later. All memory structures are associated with one level of memory and cannot migrate from level to level. To find the last frame to be added to a level of auxiliary memory, enter:

(car '<memory-level>)

7.0 THE TRACE FACILITY

The PLUM trace facility *ptrace* provides a fairly flexible facility for monitoring the execution of PLUM. *Ptrace* relies on a set of software `switches' which control the trace output. These switches allow a user to track information about function execution, prediction prototype triggering, concept-frame instantiation, and demon execution. While all of the switches can be toggled individually, *ptrace* supports four `modes' which are useful pre-defined combinations of switches.

When PLUM is initially loaded, *ptrace* is set to run by default in "debug' mode. This means that trace output will automatically be produced whenever the following situations are encountered by PLUM:

1. a new word is read
2. a prediction prototype is triggered
3. a concept-frame is instantiated
4. a demon is tested

7.1 Debugging Switches

The following is a listing of the current debugging switches,[1] along with a brief description of what they do:

demon-trace	Traces demons tested & the results
search-trace	Traces searches and checks
search-trace2	Same as *search-trace* + constraint checking
instpred-trace	Traces triggering of prediction prototypes & demons

[1] Note: all switches must assume the values t or nil.

instpred-trace2	Same as *instpred-trace* + shows entire prototype
memory-trace	Traces memory structures as they're created (Frame instantiation)
only-words-trace	Prints each word from input stream *quietly* when read
word-trace	Prints each word from input stream *loudly* when read

All of these switches can be set manually, – that is, without the use of the *ptrace* function. For example, to turn on the trace of demon testing, execute (setq *demon-trace* t). Likewise, (setq *demon-trace* nil) will turn off that trace.

7.2 Trace Modes

This is a listing of the current trace modes along with a list of the debugging switches which are turned on (set to t) when they are implemented:

debug	demon-trace, search-trace, instpred-trace, memory-trace, word-trace
full	all but only-words-trace
quiet	only-words-trace
memory	memory-trace, word-trace

If none of these modes are suitable, the trace can be customized by either altering an existing mode or adding a new user-defined mode. For example, to see only an echo of the words processed, first execute (ptrace memory), and then execute (setq *memory-trace* nil). These two calls set all switches to nil except

word-trace. If a user expects to use a customized trace mode frequently, it is easy to define new trace modes by defining a new function which calls *ptrace* in combination with individual switch settings. It is not necessary to redefine *ptrace* in these cases.

8.0 EXAMPLE: IMPLEMENTING CONCEPTUAL ANALYSIS

In this section we will define some dictionary entries and prediction prototypes that can be used to analyze some simple sentences describing a transfer of possession. We assume the reader is familiar with the representational techniques of conceptual dependency.[1]

In this example, we utilize a simple noun phrase mechanism which is not intended to be a generally adequate for noun phrase recognition. It merely suffices for some simple examples.

A careful study of the prototypes in this example will illustrate the use of the blackboard and the do-first-clause (see the prototypes for the "adjective-group" and "referent"). Note also how the prototype for "word" uses the input-stream as its target-environment. This is an exceptional prototype definition which PLUM has been specifically designed to handle as a special case for the sake of efficient lexical access. This prototype is actually already written into PLUM and is thus redundant here. It is included only for clarity.

Using these definitions, you should be able to execute some simple *ATRANS* sentences. After loading all the PLUM source code, load these definitions and examine the traces for:

```
(parse '(john gave mary a book period))
(parse '(john gave a book to mary and bill period))
(parse '(john and mary gave a book to bill period))
```

Parse will return as its value, any instantiated frames on the *EVENT-MEM* list. Make sure you understand what happens when you try:

```
(parse '(john gave mary period))
```

There are many sentences that will fail to parse correctly using this lexicon with its given definitions. See if you can predict what will happen when you execute:

---

[1] If not, a good introduction can be found in Inside Computer Understanding by Schank and Riesbeck.

```
(syn book (record))
(parse '(john gave mary a book and bill a record period))
```

Since the conjunction "and" is defined to do nothing more than terminate a noun phrase, PLUM cannot hope to handle elliptical constructions using these definitions. Sentences of this sort illustrate the need for a scoping mechanism in PLUM's expect-clause definitions.

As you experiment with different sentences, try changing the trace around as well. PLUM is set to start up with a default trace mode of "debug" (see section 7). See if you can track the demon executions and predict when specific demons will be triggered (test positively) or die (be removed from a waiting list). When you feel at home with the execution of these examples, you are ready to start designing your own prototype definitions for PLUM.

8.1 Lexical Definitions

```
;*****************************************************************
; LEXICAL DEFINITIONS
;*****************************************************************

; All dictionary entries are of the form:
; (defword item prediction-index predicts features)
;
; Nouns require additional associations to conceptual features that
; are frequently used as slot constraints. These features would be
; indexed through an extra level of indirection if we were using a
; system of primitive decomposition for representing referents.
; Until we incorporate such representations, we'll make the
; associations directly with the lexical items themselves (via the
; features field).

(defword gave (verb) (active-atrans-prediction) nil)
(defword john (noun) (np-terminator-prediction adjective-group-prediction)
   (animate proper-noun location))
(syn john (mary bill))
(defword book (noun) (np-terminator-prediction
                        adjective-group-prediction) (physobj))
(defword to (preposition) (direction-to-prediction) nil)
(defword a (article) (determiner-prediction) nil)
(defword period (period) nil nil)
(defword and (conjunction) nil nil)
(syn period ($period))
; (phrase ((water bed) (pig sty) (disk drive)))
```

8.2 Prototype Predictions

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

(create-pred type (active-atrans)
   comment  ( ATRANS -> REF gave REF OBJ ! REF gave OBJ to REF )
   concept-frame  (actor =& (animate)
                   act = ATRANS
                   object =& (physobj)
                   source = (same-as actor)
                   recipient = (animate))
   control-structure  ((expect actor in past referent
                        check referent token features)
                       (expect object in future referent token features
                        return referent)
                       (expect recipient in future referent token features
                        return referent
                       (expect recipient in next direction-to value token features
                        return value))
   memory-level (*EVENT-MEM*)
   required-slots  (actor object))

(create-pred type  (pp)
   comment (this is triggered by any preposition)
   concept-frame (preposition = (preposition)
                  object = (np))
   control-structure ((expect preposition in last word part-of-speech
                       return word)
                      (expect object in next np return np))
   required-slots (preposition object)
   memory-level (*SYNTAX-MEM*))

(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 value part-of-speech
                        return determiner)
                       (expect modifiers in last adjective-group modifying-adjectives)
                       (expect headnoun in last adjective-group last-entry))
```

```
    predicts (referent-prediction)
    required-slots (headnoun)
    memory-level (*SYNTAX-MEM*))

(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 part-of-speech
                        return word))
    predicts (np-prediction)
    required-slots (value)
    memory-level (*SYNTAX-MEM*))

(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 last adjective-group)
                       (expect modifying-adjectives in current blackboard)
                       (expect last-entry in last word part-of-speech
                        return word))
    required-slots (last-entry)
    memory-level (*SYNTAX-MEM*))

(create-pred type (determiner)
    comment (this is triggered by any article)
    concept-frame (value = (article))
    control-structure ((expect value in last word part-of-speech
                        return word))
    required-slots (value)
    memory-level (*SYNTAX-MEM*))

(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*))

(create-pred type (direction-to)
    comment (this is triggered by any directional preposition
    concept-frame (value = (location or physobj or animate)
```

```
                orientation = (to))
   control-structure ((expect value in next referent
                       check referent token features))
   required-slots (value)
   memory-level ('SYNTAX-MEM'))

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

9.0 REFERENCES

1. DeJong, G., "Prediction and Substantiation: A New Approach to Natural Language Processing," Cognitive Science, 1979, 3, pp. 251-73.

2. Dyer, M., In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension. MIT Press, Cambridge, MA. 1983.

3. Johnson, P., "Requirements Definition for a Plumber's Apprentice." Proceedings: Second Annual Workshop on Theoretical Issues in Conceptual Information Processing, May, 1985, Yale.

4. Schank, R., & Riesbeck, C.; Inside Computer Understanding: Five Programs Plus Miniatures. Lawrence Erlbaum Assoc., Hillsdale, NJ. 1981.

5. Thibadeau, R., Just, M., & Carpenter, P., "A Model of the Time Course and Content of Reading," Cognitive Science, 1982, 6, pp. 157-203.

Release Notes for PLUM and VPLUM

(Common Lisp - Version 4.0)

K. Arvind
Wendy Lehnert
Philip Johnson
Brian Stucky

July 17, 1986

1. Introduction.

PLUM Version 4.0 is significantly different than prior implementations, both internally and externally. Internally, it has been completely rewritten in common lisp, so that it can be ported with only minor changes to any common lisp site. Version 4.0 also contains several additions and modifications to the PLUM language. The most notable enhancements are the development of an insert clause mechanism, which allows the user to add slots to frames during parsing, and language constructs to define a context for the operation of demons. These enhancements and others are described more fully below. Finally, this document also describes the compatible version of VPLUM, the Symbolics Lisp Machine interface.

2. Loading PLUM.

2.1 File Path Names.

Please note that the designated path names for files in `loadplum.lsp` and `lispm-init.lsp` are specific for UMass users and will have to be appropriately modified for users at remote sites.

2.2 The VAX Interface.

To load PLUM in the VAX/VMS environment, simply type:

```
$ lisp/init=nlp$disk:[nlp.plum]loadplum.lsp
```

After receiving the `PLUM>` prompt, the prototype and dictionary files should be loaded. For example, to load the prototype definitions for a parse of the sentence "John gave Mary a book," type:

```
PLUM> (load ''nlp$disk:[nlp.plum]example.lsp'')
```

Parsing the sentence is accomplished in the normal fashion:

```
PLUM> (parse '(john gave mary a book period))
```

2.3 The Symbolics Interface.

Loading PLUM and VPLUM is accomplished by:
`(login 'vplum)`

This single command will load in all the plum and vplum source files, and pop up a window of screen configurations for the user to choose from. After making a choice, the user must load in the prototype and dictionary definitions. Note that loading files from the vax is slightly idiosyncratic; there must be only one colon between the machine node specification and the directory specification. For example, to load in the "John gave Mary ..." prototypes, type:
Command: `(load ''vax1:nlp$disk:[nlp.plum]example.lsp'')`

(The normal VMS syntax is "vax1::nlp$disk..."). For more information on the use of VPLUM and other tools, see (the soon to be written) *The Parser's Apprentice User Manual.*

3. Differences between PLUM Version 4.0 and earlier releases.

3.1 Return Clauses.

VaxPlum does not support the use of the keyword RETURN in prototype definitions. The keyword CHECK may be used instead.

3.2 -Prediction Suffixes.

In previous versions of Plum, the suffix '-prediction' was used in the entries filling the 'predicts' slots of prediction prototypes and the 'predictions' field of a defword. In VaxPlum, this suffix should not be used. For example, the prediction field of an np-terminator prototype (see [nlp.plum]example.lsp) formerly contained `(np-prediction)`. In version 4.0, this should simply be `(np)`.

3.3 Defpred Supercedes Create-pred.

The function `create-pred` has been renamed `defpred` in order to be consistent with `defword`, `defstruct`, `defflavor`, etc. `Create-pred` will continue to be supported for the conceivable future, however.

3.4 Tracing the Parse.

The function ptrace has been replaced by the function `set-trace-mode` and a new utility called `set-trace-switches` has been added. Calling `set-trace-switches` returns a menu of the following trace-switches so that each can be set individually:

- `*trace-demons*` `*trace-demons-in-detail*`
- `*trace-memory*` `*trace-memory-in-detail*`
- `*trace-search*` `*trace-search-in-detail*`
- `*trace-predictions*` `*trace-predictions-in-detail*`
- `*trace-words*` `*trace-words-in-detail*`

`Set-trace-mode` can be called with the following arguments: `:quiet`, `:debug`, `:memory`, and `:full`. It sets combinations of switches in a manner exactly analogous to `ptrace`.

3.5 Accessing Internal Data Structures.

Due to differences in the data structures that VaxPlum and CLisp Plum use, some of the functions used to access data structure values in VaxPlum are different from those used in CLisp Plum. For example, use `(get 'np 'waiting-demons)` instead of `(get 'np 'waiting-list)` and `(get 'np 'realizations)` instead of `(get-instances 'np)`.

3.6 INFO and NEWS Utility.

INFO (short for information) offers help on various topics including the Lisp functions that constitute VaxPlum. INFO is invoked by typing in: `(INFO <topic>)` or just `(INFO)`. At present INFO is not comprehensive; there are a lot of topics for which help text has not yet been added.

NEWS may be used to broadcast news to Plum users. To broadcast news, enter it in the file nlp$disk:[nlp.plum]plumnews.lsp. To read news type in: `(NEWS)`.

4. PLUM Version 4.0 Source Files.

Plum currently resides in the directory: nlp$disk:[nlp.plum]. A brief description of each component file follows:

- `loadplum.lsp`: This file loads the other files and initializes Plum.
- `globals.lsp`: This file contains declarations for all the global variables and structures used in VaxPlum.
- `tools.lsp`: This file contains tool functions used by the various components of VaxPlum.
- `syntax.lsp`: This file contains the syntax checker for prediction prototypes.
- `translator.lsp`: This file contains the functions that translate prediction prototypes to the internal representation used by VaxPlum.
- `parser.lsp`: This file contains the actual working components of VaxPlum.
- `utilities.lsp`: This file contains code for the various utilities (info, news etc.) supported by VaxPlum.
- `helptext.lsp`: This file contains text used by the Plum Help Facility.
- `plumnews.lsp`: This file contains the text used by the Plum News Facility.
- `domain.lsp`: This file contains domain specific functions (like memtok).
- `lispm-init.lsp`: This file contains the Symbolics login file for VPlum.
- `cl-plum-useful-frame.lsp`: This file contains the Symbolics code for constraint frame definition.
- `vplum4.lsp`: This file contains the VPlum source code.

5. Enhancements to PLUM.

5.1 The PLUM Insert Clause.

Insert clauses facilitate the insertion of new slots (slots that do not appear in the prediction prototype definition) into a concept frame. Insert clauses are specified within the control structure field of a prototype definition. Any number of insert clauses may appear in any order within the control structure field. The format of an insert clause is:
`(insert current (` *slot-name*`) in` *search-direction memory-structure*`)`

Whenever a prototype containing an insert clause is instantiated (i.e., a memory structure is created for its concept frame), a new slot with name *slot-name* is inserted into the memory structure given by *search-direction memory-structure*. The value of this slot is set to point to this instantiation of the prototype. If the *search-direction* is *next* or *future* then an insert

demon is created that waits for the insertee memory structure to be created. If the *search-direction* is *any*, then the new slot is inserted into all existing memory structures that satisfy the specification and then an insert demon is also created to insert the slot into future memory structures. The action for *past* is the same as that for *any* except that the insert demon is not created. For *last* and *next* insertion is done into only one memory structure, while for the other search directions the new slot may be inserted into several memory structures. The memory structure *current blackboard* is not a valid insertee structure.

For example, consider parsing the sentence "John gave Mary the book at noon." where 'at' triggers the memory-structure 'time' defined below:

```
(create-pred (time)
  comment (this is triggered by preposition at)
  concept-frame (value = (hour))
  control-stucture
    ((expect value in next word check word features)
     (insert current (time) in last active-atrans))
  required-slots (value)
  memory-level (*concept-mem*))
```

The word 'noon' is defined as:

```
(defword noon (noun) nil (hour))
```

Parsing this sentence results in the creation of an 'active-atrans' structure into which the slot 'time' is inserted. The value of the 'time' slot is an instance of the prototype 'time'.

5.2 Reverse Until.

Backward scoping on the UNTIL clause is now supported in PLUM. This feature allows a *boundary* to be established that limits the range of a reverse search. For example:

```
EXPECT actor IN PAST atrans UNTIL period
```

would search for an ATRANS memory structure until either it is found and the actor slot is filled *or* a PERIOD memory structure is encountered, in which case the demon is killed. In other words, the search will only be successful if the instantiation of the ATRANS occurred after that of a PERIOD frame. The PERIOD memory structure acts as a boundary that restricts the scope of the backward search.

5.3 The PLUM Demon Stack Processing.

This feature allows the user to create and parse within a specific level during the course of processing (much like pushing and popping a recursive net within an ATN). Use of *level stamping* results in saving the current context (currently active demons) and initializing a new parsing context that is operative until the old state is popped back.

Level stamping is handled by means of two functions that will normally be used only with the DO-FIRST and DO-LAST clauses:

PUSH-LEVEL takes a single argument (which is ignored). This function creates a new context for demons. It saves the demons associated with the existing context in context stacks associated with each type of memory structure.

POP-LEVEL takes a single argument (which is ignored). This function kills all demons associated with the current context and restores the last stacked context for each type of memory structure. *Every* execution of POP-LEVEL should normally be matched by a previous execution of PUSH-LEVEL.

As an example, consider the sentence:

```
John, who left yesterday, gave Mary a shirt.
```

With level stamping, we can process the relative clause in an independent context and use the results at the sentential level. The word 'who' triggers the relative clause frame that initializes a new parsing context. Any active demons will be saved.

```
(create-pred (relative-clause)
  concept-frame      (...)
  control-structure  ((do-first PUSH-LEVEL on SYNTAX-MEM)
                      (              ...              ))
  required-slots     (...)
  memory-level       (SYNTAX-MEM))
```

When the relative clause is completed, the activation of an ATRANS frame from the word 'gave' kills the current context and restores the demons that are on the stack. Processing continues on this level for the remainder of the sentence.

```
(create-pred (ATRANS)
  concept-frame      (...)
  control-structure ((do-first POP-LEVEL on SYNTAX-MEM)
```

```
                    (                ...                ))
required-slots      (...)
memory-level        (EVENT-MEM))
```

Note that the specification of the memory-level SYNTAX-MEM within the do-first clause is arbitrary and has no bearing on the processing. It is done to maintain consistency with required syntax and alternative uses of the do-first and do-last clauses.

6. Using VPlum and Miscellaneous Symbolics Information.

6.1 Boot Dopey or Grumpy under 6.1

1. Always start up VPLUM in a fresh environment. If the previous user did not logout, do it for them by entering (logout).
2. "Chord" the 3 keys: cntrl-hyper-function. This will cause the machine to respond 'Lisp stopped itself" at the top of the screen, and prompt you for a FEP command.
3. Type "boot " (boot with a space) and check what the default load file is. For version 6.1, you need >boot.boot, for version 5.3, you need >boot-5-3.boot. `VPlum version 3 runs in version 5.3, VPlum version 4 runs in version 6.1.` If the default is not what you want, type in the correct boot file, otherwise simply type return.
4. If all goes well, the machine will be initiated and loaded in about 2 minutes. Do not try to interrupt during this loading process. To see if the system is still loading, watch the system indicators at the very bottom of the screen display.

6.2 Loading VPLUM

1. Once you have a lisp listener, type (login 'vplum)
2. Wait for a pop-up menu to appear. This menu contains several options, but only 3 should be considered: VPLUM Displays 1-3. To select one of these configurations select it with the mouse and click-any. A split screen display will replace the old screen.
3. Before any more s-expressions can be evaluated, you have to activate a lisp listener.
   - **Version 5.3 (VPlum version 3):** Do this by either typing SELECT-L, or by moving the mouse into the lisp listener and click-left.
   - **Version 6.1 (Vplum version 4):** Do this by either typing SELECT-SYMBOL-SHIFT-L, or by moving the mouse into the *common* lisp listener and click-left.
4. If you are running VPlum version 3, then type (load-vplum). Under version 4, VPlum is loaded automatically for you.

6.3 Load your prototype and dictionary files

1. Select the lisp listener.
2. (load "vax1:nlp$disk:[nlp.plum]example.lsp") or whatever file you want. (Note: it is best to store files in your VAX account and bring them in over the net- files are not backed up on the lisp machines!)

6.4 Running VPLUM

Running VPLUM is just like running PLUM except you call VPLUM instead of PARSE. E.g. (vplum '(john gave mary a book period))

VPLUM will provide a dynamic display of the parse as PLUM works through the sentence. Prototypes that have been activated but not instantiated are represented with dotted boxes; instantiated prototypes are represented with solid boxes.

The display will either show slot filling relations or prediction sources. To toggle between these two display modes, click-middle. If the prototype references another form of memory (for example, memory tokens in REFERENT processing), no slot filling arrows from the referent prototype will be displayed, since its slot is not filled by a PLUM memory structure.

To scroll the display to the right, click-right.

To scroll the display to the left, click-left.

To display a prototype in the lisp listener, click-any while the mouse is positioned over the appropriate box in the display window.

To see a partial parse of the sentence, or a reparse of the whole sentence, click-any while the mouse is positioned over a word in the display window. VPLUM will parse up to and including that word.

6.5 Changing Screen Configurations

If you've brought up VPLUM under one screen configuration, and you want to change to another one, bring up the VPLUM display menu by (1) double-click-right (this brings up the system menu), and (2) click-right on 'useful frame" (in the last row, first column)

7. Staying Alive and Healthy on the Symbolics.

7.1 Useful Lisp Machine Functions

- If you kill the mouse (it refuses to respond to anything), try the following: get into a common lisp listener (use SELECT-SYMBOL-SHIFT-L, or simply SELECT-L for version 5.3 lisp listeners), then type (send tv:mouse-process :reset). Sometimes this works. Sometimes it doesn't. Sometimes moving the mouse vigorously works. If not, resort to a soft reset (the cntrl-hyper-function chord, followed by START).
- If you seem to have killed the system (no response to anything), try hitting ABORT. If that doesn't help, try cntrl-ABORT. Then try cntrl-meta-ABORT. If all of these fail, reset (using cntrl-hyper-function START).
- Just as SELECT-SYMBOL-SHIFT-L will get you a common lisp listener, SELECT-E will put you in the editor, and SELECT-T will put you in a vax terminal emulator. In version 6.1 only, SELECT-D gives you a document examiner. If you ever want to get at on-line documentation for a command (like maybe SELECT-D), just type SELECT-HELP and then enter the command.
- cntrl-shift-a will show arguments to a function named FOO if you position the point anywhere within the definition of FOO.

7.2 General display tricks:

If you have a lot of stuff being written to the lisp listener window, the display will freeze with **MORE** until you hit the space bar. Note that this will hold up the VPLUM visual display as well! If you want to turn this screen hold off, type function-M. This same toggle will turn the screen hold on again if you want to reinstate it. If you get **MORE** during the display of a prediction prototype, simply click over the prototype again to move past the **MOVE**.

function-C toggles the black-on-white/white-on-black screen display.

7.3 Some Editting Commands

If you are in the habit of using KEYMACS or some other varient on EMACS, you will probably want to go back and learn the standard EMACS commands for cursor control. Here are some of the most useful ones:
cntrl-F move right one character
cntrl-B move left one character
cntrl-K kill to end of line

meta-F move right one word
meta-B move left one word
cntrl-N move down one line
cntrl-P move up one line
cntrl-meta-F move right one s-expression
cntrl-meta-B move left one s-expression
cntrl-meta-K kill the following s-expression
cntrl-Y yank from kill buffer (continue with meta-Y)
cntrl-meta-Q format following s-expression
cntrl-X-cntrl-S saves a file

You can scroll the screen with the mouse by 'bumping" the mouse up against the edge of the window in the right corners.

The cursor can be positioned by the mouse using click-left.

Regions of text can also be marked by using the mouse. Once the cursor is positioned, 'drag" the mouse over a region by holding down the left mouse button as you mouse the mouse. When you release the button, your region will be marked.

7.4 File I/O

Once inside an editting window, you can load a file by typing cntrl-X-cntrl-F. A small window will activate and prompt you for the file name. Don't forget to specify the host machine! E.g. VAX1:NLP$DISK:[lehnert.plum]myfile.lsp

cntrl-X-cntrl-S saves a file

To evaluate a lisp expression in an editting window (i.e. 'stuff" that expression into the lisp listener), use cntrl-shift-E.

To compile a lisp expression from inside the editor, using cntrl-shift-C.

cntrl-X-cntrl-B displays all files editted during your session. To display one in your window, use the mouse to select it and click-left.

meta-X puts you in extended command mode inside the editor. This is useful if you want to evaluate or compile a whole buffer (or file). With the buffer (or file) currently displayed in the editting window, type meta-X followed by 'eval buffer" or 'compile buffer" in the extended command window. (no quotes needed). Note: it is generally better to compile expressions rather than just evaluate them.

meta-. will locate a function definition for you in another file.