USING THE PEARL AI PACKAGE

(Package for Efficient Access to Representation in Lisp)

by

M. Deering, J. Faletti and R. Wilensky

Memorandum No. UCB/ERL M82/19

31 March 1982
(Revised 27 May 1983)

Using the PEARL AI Package

(*P*ackage for *E*fficient *A*ccess to *R*epresentations in *L*isp)*

Michael Deering
Joseph Faletti
Robert Wilensky

Computer Science Division
Department of EECS
University of California, Berkeley
Berkeley, California 94720

February 1982

ABSTRACT

This document is a tutorial and manual for PEARL (Package for Efficient Access to Representations in Lisp), an AI language developed with space and time efficiencies in mind. PEARL provides a set of functions for creating hierarchically-defined slot-filler representations and for efficiently and flexibly inserting and fetching them from a forest of associative data bases. In addition to providing the usual facilities such as demons and matching, PEARL introduces stronger typing on slots and user-assisted hashing mechanisms.

* This research was sponsored in part by the Office of Naval Research under contract N00014-80-C-0732 and the National Science Foundation under grant MCS79-06543.

Table of Contents

1. Introduction 1
2. Running PEARL 2
    2.1. Under Franz Lisp 2
    2.2. Under UCI Lisp 3
3. Creating Simple Objects 4
    3.1. Defining Symbols 4
    3.2. Defining Structures 5
4. Creating Individual Instances of Defined Structures 6
5. Accessing Slots of Structures 8
6. Storing In and Retrieving From the Data Base – The Simplest Way 11
    6.1 Storing In the Data Base: *Insertdb* and *Removedb* 11
    6.2 Retrieving Hash Buckets From the Data Base: *Fetch* 12
    6.3 Accessing the Results of *Fetch*: *Nextitem* 13
7. The Default Values for Unspecified Slots 14
8. Using *Patterns* For More Flexible and Powerful Retrieval 15
9. Marking Structures During Creation For More Efficient Retrieval 17
10. Printing Structures, Symbols and Other PEARL Objects 20
11. Error Messages, Bugs, and Error Handling Abilities 21
12. Short-Circuiting and Redirecting *Create* Using !, $ and Atoms 22
13. More Flexible Hash Selection 23
14. Using Predicates to Constrain Fetching 27
15. More Useful Slot Types 30
16. Attaching Hooks to Structures (If-Added Demons) 32
17. Creating and Manipulating Multiple Data Bases 38
18. Creating a Forest of Data Bases 39
19. Creating Expanded Subtypes of Previously Defined Objects 40
20. Fetching Expanded Structures 42
21. How Two Objects *Match* 43
    21.1 When Is a Pattern not a *Pattern*? 43
    21.2 The Matching Process 44
22. Binding Blocks of Structures Together Via Common Variables 47
23. Controlling the Unbinding of Variables by *Match* 48
24. Function Structures 49
25. More About the PEARL Top Level Loop and History Mechanism 51
26. Looping and Copying Functions 54
27. Miscellaneous Variations and Abbreviations 55
28. Low Level Access Functions 57
29. Appendix of UCI Lisp functions added to Franz PEARL 58
30. Appendix of Franz Lisp functions added to UCI Lisp PEARL 62
31. Bibliography 63
32. Index of Global Variables and Functions With Their Arguments 64
33. Concept Index 71

Update of Changes
Through
PEARL 3.8
April 1983

Table of Contents

1. Introduction 76
2. Running PEARL 76
   2.1. Under Franz Lisp 76
5. Accessing Slots of Structures 76
10. Printing Structures, Symbols and Other PEARL Objects 76
    10.1. Abbreviations 77
11. Error Messages, Bugs, and Error Handling Abilities 78
12. Short-Circuiting and Redirecting *Create* Using !, $ and Atoms 78
13. More Flexible Hash Selection 78
16. Attaching Hooks to Structures (If-Added Demons) 79
17. Creating and Manipulating Multiple Data Bases 80
19. Creating Expanded Subtypes of Previously Defined Objects 80
20. Fetching Expanded Structures 80
21.2 The Matching Process 80
26. Looping and Copying Functions 81
29. Appendix of UCI Lisp functions added to Franz PEARL 81
32. Index of Global Variables and Functions With Their Arguments 81
34. Compiling Lisp+PEARL Files 82

Using the PEARL AI Package

(*P*ackage for *E*fficient *A*ccess to *R*epresentations in *L*isp)*

Michael Deering
Joseph Faletti
Robert Wilensky

Computer Science Division
Department of EECS
University of California, Berkeley
Berkeley, California 94720

February 1982

1. Introduction

PEARL (Package for Efficient Access to Representations in Lisp) is a set of functions for creating hierarchically-defined slot-filler representations and for efficiently and flexibly inserting and fetching them from a forest of data bases. Its intended use is in AI programming and it has been used at Berkeley in the development of several AI programs including PAM [7] and PANDORA [8].

PEARL has the expressive power found in other AI knowledge representation languages, but is extremely time-space efficient. For example, using a data base of 4000 entries, PEARL takes only about 4.2 CPU milliseconds for an average unsuccessful query and 7.3 CPU milliseconds of an average successful query on a PDP-10.

This document describes PEARL's use and is intended for the beginning user. (A description of the implementation of PEARL will be available shortly.) The best way to approach PEARL is to read this document up through section 11 and then to take it to a terminal and reread it, typing in the examples and observing their effects.

PEARL was implemented by Michael Deering and Joseph Faletti. It was originally developed on a DEC PDP-10 under UCI Lisp and was subsequently moved to a DEC VAX 11/780 under Franz Lisp with help from Douglas Lanam and Margaret Butler. Both PEARL and its documentation are still being developed, improved, and debugged. Any comments or suggestions will be appreciated. Send them to Joe Faletti via Arpanet or Unix mail (Kim.Faletti@Berkeley or ucbvax!kim.faletti).

* This research was sponsored in part by the Office of Naval Research under contract N00014-80-C-0732 and the National Science Foundation under grant MCS79-06543.

2. Running PEARL

PEARL is implemented as a set of functions compiled and loaded into Lisp. Thus the full power of Lisp is available in addition to the added power of PEARL.

Since PEARL runs under two different Lisps on two different machines, there are a few differences between versions. Most of these differences are in the method of starting PEARL up and in the names of external files accessed by PEARL. The two parts of this section describe how to start up PEARL either under Franz Lisp or under UCI Lisp. You need only read the section which is applicable to your Lisp.

2.1. Under Franz Lisp

To access PEARL, simply run the core version of Lisp containing PEARL. On Berkeley Unix, this is available by typing the command:

```
% ~bair/bin/pearl
```

or, if ~bair/bin is in your search path, simply:

```
% pearl
```

During the startup process, PEARL will read in two files, **.init.pearl** and **.start.pearl**, if they exist. These files are designed for purposes similar to those of *.lisprc*. However, they split these functions into two groups. In your *.init.pearl* file you should include any expressions which change the user-settable parameters to PEARL. (For example, methods for setting the size of data bases, the print function, and the prompt are described below.)

When you wish to have other files read in at startup time, this usually needs to be done after PEARL's parameters are set. PEARL is set up so that after the reading of *.init.pearl*, it sets any necessary parameters which you have not set in your .init.pearl and then reads in the file *.start.pearl* if you have one. This is where any processing which requires the attention of PEARL (such as the existence of its data bases) should be placed. Thus *.init.pearl* is primarily for initializing PEARL and *.start.pearl* is for starting up your use of PEARL. **Note**: unlike most Unix programs which look for startup files only in your home directory, thereby limitting you to only one environment for each program, PEARL looks for each file first in the current directory and if there is none, then it looks in your home directory. This allows you to tailor invocations of PEARL to the kind of work you do in a particular directory.

After reading in these two files, PEARL will then place you in a modified prompt-read-eval-print loop, with a default prompt of "PEARL> ". This can be changed by setting the special variable ***pearlprompt*** to the desired value. If you want the standard Lisp prompt "-> " to be used by PEARL, you must set **pearlprompt** to *nil* in your *.init.pearl* and PEARL will do the right thing.

The primary feature of the PEARL prompt-read-eval-print loop is that it uses a different print function. The default function is

```
(lambda (*printval*)
        (valprint *printval* 4) )
```

but this can be changed to whatever you desire by giving a new function definition to **pearlprintfn**. The PEARL prompt-read-eval-print loop also contains a number of features to improve upon the standard Lisp top level. These include a history mechanism and are described in chapter 25.

There are quite a few functions from UCI Lisp which have been added to PEARL to make it easier to move programs to Franz Lisp. A list of these with brief documentation of differences is included in an appendix.

2.2. Under UCI Lisp

To access PEARL, simply run the core version of Lisp containing PEARL. On the Berkeley KL-10 system, this is available by typing the system call

```
RU PEARL[5500,504,PEARL]
```

During the startup process, PEARL will read in two files, INIT.PRL and START.PRL, if they exist. The file INIT.PRL is designed for purposes similar to those of INIT.LSP. In this file you should include any expressions which change the user-settable parameters to PEARL. (For example, methods for setting the size of data bases, the print function, and the prompt are described below.) If you wish to use the REALLOC function to enlarge your memory space, this call should be the last call in your INIT.PRL file.

When you wish to have other files read in at startup time, this usually needs to be done after the REALLOC. The common kludge with UCI Lisp to solve this is to define an INITFN (initialization function) which does this and then to reset the INITFN to *nil* which returns you to the standard Lisp prompt-read-eval-print loop. However, PEARL sets the INITFN for its own purposes so that this common "solution" will not work. Instead, PEARL is set up so that after the reading of INIT.PRL, it sets any necessary parameters which you have not set in your INIT.PRL and then reads in the file START.PRL if you have one. This is where any processing which requires the attention of PEARL should be placed. Thus INIT.PRL is primarily for initializing PEARL and START.PRL is for starting up your use of PEARL.

After reading in these two files, PEARL will then place you in a modified prompt-read-eval-print loop, with a default prompt of "PEARL> ". The ">" portion is the (modified) Lisp prompt which is printed whenever ***read*** is invoked and can be changed with the UCI Lisp function INITPROMPT. The "PEARL" is PEARL's addition and can be set by setting the special variable ***pearlprompt*** to the desired value. If you do not want any prompt added by PEARL other than the Lisp prompt you must set **pearlprompt** to *nil* in your INIT.PRL and PEARL will do the right thing.

The main feature of the PEARL prompt-read-eval-print loop is that it uses a different print function. The default function is

```
(lambda (*printval*)
        (valprint *printval* 4) )
```

but this can be changed to whatever you desire by giving the function **pearlprintfn** a new definition. Note that *dskin* and the break package have been changed slightly to also use of this print function. Also, although the functions names and examples below are in lower case, PEARL in UCI Lisp expects them all in upper case, just as the rest of the UCI Lisp functions.

3. Creating Simple Objects.

PEARL allows four basic types of objects. The first two are integers and arbitrary Lisp objects and are created in the usual Lisp fashion. The second two are structured types provided by PEARL, and are stored in an internal form as blocks of memory. These latter types are called **symbols** and **structures**.

3.1. Defining Symbols

Symbols are PEARL's internal atomic symbols. Semantically they are like Lisp atoms, but are represented and used differently to make PEARL more efficient. Before they are used, symbols must be declared (and thus defined to PEARL) by a call to the function **symbol**, which takes as arguments any number of atoms whose names will be used to create symbols. For example,

```
(symbol John)
```

creates one symbol called John and

```
(symbol Bob Carol Ted Alice Home
        Office School Healthy NewYork)
```

creates several symbols at one time. *Symbol* is an nlambda (fexpr) and returns a list containing the names of the symbols it created. A one-argument lambda (expr) version is available as **symbole**.

There are two ways to get at the actual (unique) symbol: you can use the function **getsymbol** or you can evaluate the atom whose name is built out of the symbol name with the characters **s**: on the front. The function **symatom** will build this atom for you when given a symbol name. For example, to set B to the symbol Bob use any of:

```
(setq B (getsymbol 'Bob) )
(setq B s:Bob)
(setq B (eval (symatom 'Bob))
```

Given an internal symbol, you can find out its print name by passing it to the function **pname** (which also will return the print name of other types of PEARL objects).

3.2. Defining Structures

Structures in PEARL provide the ability to define and manipulate logical groupings of heterogeneous data and are essentially objects with slots and slot fillers. As such, they act more like "records" in Pascal or "structures" in C than Lisp lists. In reality they are more than both, but for the moment the reader should keep records in mind.

Just as you must define the form of a record in Pascal before defining the value of a variable whose type is that kind of record, it is necessary to define each particular form of structure you wish to use in PEARL before creating an object with that form. PEARL provides one function called **create** which is used both to define kinds of structures and to create individual instances of these structures. (One function is provided for both because a special individual is created as a side effect of each definition. More on this is provided in section 7 on defaults.) The first argument to *create* distinguishes between a call which defines and one which creates an individual. There are three kinds of defining calls (*base*, *expanded* and *function*) and two kinds of instance-creating calls (*individual*, *pattern*) to *create*. Only one of each (*base* and *individual*) is described in this section. The rest are left for later.

To start off with an example, let us suppose that you wish to represent the conceptual act "PTrans" from the Conceptual Dependency (CD) notation of Schank. (The examples in this documentation assume a passing familiarity with CD but lack of this should not hurt you too badly and PEARL itself does not restrict you in any way to CD. PTrans stands for Physical Transfer which has four "cases": actor doing the transfer, object being transferred, original location and final location.) First we must define the form which PTrans structures will take. In C this would be a type definition for the type PTrans as follows (assuming a system-provided definition of the type *symbol*):

```
struct PTrans {
        symbol Actor;
        symbol Object;
        symbol From;
        symbol To;
};
```

In Pascal this would be

```
type PTrans = record
                  Actor : symbol;
                  Object : symbol;
                  From : symbol;
                  To : symbol
              end;
```

In PEARL,

```
(create base PTrans
        (Actor symbol)
        (Object symbol)
        (From symbol)
        (To symbol) )
```

does the job. Note first of all that in order to define a new form of structure, the first argument to *create* must be **base**. Note also that the second argument to *create* is the name of the structure form to be created. Following this is a list of (<slotname> <type>) pairs. Structures are currently allowed to have up to 32 slots in Franz PEARL or 18 in UCI Lisp PEARL as long as all slots within a particular structure have mutually distinct names. Different structures may have slots of the same name. Thus in applications of PEARL to CD twenty different structure types might all have an Actor slot.

Five types are allowed for slots: **symbol**, **struct**, **int**, **lisp**, and **setof <type>**. *Symbol* and *struct* are the types just described. *Int* is a normal Lisp integer value. The type *lisp* allows arbitrary **non-atomic** Lisp values. Finally, *setof <type>* allows you to define sets consisting of all symbols (*setof symbol*) or all structures (*setof struct*) and can be done recursively (*setof setof struct*).

4. Creating Individual Instances of Defined Structures

Once you have defined a specific form of structure like PTrans, you can create an individual PTrans using **individual** as the first argument to *create* and the name of the base structure you want an individual instance of as the second argument. The rest of the arguments are (<slotname> <value>) pairs in which the <value> must be of the type that the slot was declared to be. The slots may be listed in any order and need not be in the same order as defined. For example, to create an instance of John going home from the office (i.e., John PTransing himself from the office to home) you would use this call to *create*:

```
(create individual  PTrans
           (Actor  John)
           (Object  John)
           (From  Office)
           (To  Home) )
```

Create will return an object of type PTrans, with the slots filled in as indicated. The object returned has been created and stored as a *hunk* of memory in Franz Lisp or a block of memory in Binary Program Space in the UCI Lisp (rather than Free Storage where most Lisp objects are stored). Since you are using the PEARL prompt-read-eval-print loop, the object returned by *create* will be printed in an external list form, something like the above. However, if you print a structure using the standard Lisp print functions (as for example some break packages will do), it will be printed by Franz Lisp in the normal way it prints hunks. (Warning: Since the structure actually contains a circular reference to another hunk, this will cause problems with programs which do not set *prinlevel* in Franz Lisp so general packages which you wish to add to PEARL should be modified to use some PEARL print function.) With UCI Lisp's normal print function, it will show up as an address in Binary Program Space, looking something like "#31534".

As with any Lisp function that returns an object, we must store (a pointer to) the result of *create* somewhere (for example, in the atom Trip) if we wish to reference it in the future. Otherwise, the created object will be inaccessible. (This point is clearer if you consider that Pascal would insist that you do something with the result of the function call, although PEARL and many languages like Lisp and C in which every subprogram is a value-returning function allow you to construct a value and then blithely go on your way without using it.)

To store (a pointer to) the instance returned by *create* in the atom Trip, you could do the following:

```
(setq Trip (create individual PTrans
                   (Actor John)
                   (Object John)
                   (From Office)
                   (To Home) ) )
```

Since this is a common operation, *create* provides the option of having (a pointer to) the newly created instance automatically assigned to a Lisp atom. This is accomplished by including the name of the atom as the third argument to *create*. If the third argument to *create* is an atom rather than a (<slotname> <value>) pair, *create* stores the new object in this atom. Thus the effect of the previous example can be achieved by:

```
(create individual PTrans Trip
        (Actor John)
        (Object John)
        (From Office)
        (To Home) )
```

(In addition, when *create base PTrans* is used, an assignment is automatically made to the atom PTrans, thus making the defaultinstance of a structure easily available. To preserve this, calls to create of the form *(create individual PTrans PTrans ...)* are disallowed (that is, ignored). In case you should actually wish to use the atom PTrans for other purposes, evaluating the atom built by prepending **i:** onto the structure name will give you the default instance of a base structure and evaluating the atom built by prepending **d:** will give you the actual definition. Changing the value of these atoms is **very dangerous**. Given the name of a structure, the functions **instatom** and **defatom** will construct these atoms for you. For more information about the item assigned to *PTrans* and *i:PTrans*, see the section 7 on defaults.)

PTrans is an example of a structure whose slots are all of the type *symbol*. A more complex example is that of MTrans (Mental Transfer: an actor transfering a concept (Mental Object) from one place to another (usually from himself to someone else). The MObject slot is some other act and so would be of type *struct* resulting in the following definition:

```
(create base MTrans
        (Actor symbol)
        (MObject struct)
        (From symbol)
        (To symbol) )
```

A sample instance of MTrans is *John told Carol that he was going home from the office* and would be created with

```
(create individual MTrans InformLeaving
        (Actor John)
        (MObject (PTrans Leaving
                         (Actor John)
                         (Object John)
                         (From Office)
                         (To Home) ) )
        (From John)
        (To Carol) )
```

Note that to fill a slot of type *struct* with a structure value within a *create* one just embeds the appropriate arguments for a recursive call to *create*, *except* that you may leave out *individual* since it would just be repetitive. If you should want to create an object of another type within an individual or base structure, you must include the alternative argument (*individual*, *base*, *pattern*, *expanded*, or *function*) before the type name. This is particularly useful when you wish to create a pattern with an individual instance in one of its slots.

The optional third argument of an atom name for storing in may be included at each level if you wish. In the example above, *create* actually will create two new instances, an MTrans which will be stored in InformLeaving, and a PTrans which is pointed to by the MObject slot of the MTrans and is also pointed to by Leaving. In this case, neither InformLeaving nor Leaving is required. If Leaving were left out, then one would still have a way to get at the PTrans via the MObject slot of the MTrans that InformLeaving points to. However, if InformLeaving were left out and the result of the call to *create* were not stored any other way, there is one more way that the MTrans would be accessible. The value of the most recently created object is always stored in the special variable ***lastcreated*** by *create* so the value of the MTrans would remain accessible until the next call to *create*. Note that if there are recursive calls to *create* during this time in order to process structure values in slots, the value of **lastcreated** is continually changing to the most recent one and the setting of **lastcreated** is the last thing *create* does. There is also a special variable called ***currenttopcreated*** which is set by *create* at the top level call as soon as the space for an individual or default instance is allocated. Since it is sometimes handy for a piece of user code which runs during *create* (see the sections on !, $, predicates and demons) to be able to access the topmost object, **currenttopcreated** is sometimes quite useful.

As in C and Pascal, one can embed to any level. *Create* does not have facilities for more complex networks of structures, as there are other functions in PEARL which allow their construction. *Create* is mainly used to create objects for other functions to manipulate.

5. Accessing Slots of Structures

In C and Pascal one can access the slots of a record or structure by using dot notation. For example, in Pascal the To slot of the MObject slot of the MTrans pointed to by InformLeaving would be accessed with the expression InformLeaving.MObject.To (or perhaps more accurately InformLeaving^.MObject^.To since slots really contain pointers to objects). In Pascal and C, there are essentially only two things that one can do to a slot of a record or structure: access it (get its value) and assign to it (give it a new value). In PEARL the macro **path** provides a large number of ways to access and/or change the values in slots of individual structures. (In the UCI Lisp version this is called *ppath* to distinguish it from the system function *path.*) A call to *path* is of the following general form:

```
(path <Selector> <Structure> <Slot-Name-or-List> <Value>)
```

<Selector> determines the action to be performed and is not evaluated. <Structure> should evaluate to the object in which the slot occurs (or in whose depths the object occurs). <Slot-Name-or-List> should evaluate either to the atom name of the slot desired in <Structure> or a list of the slot names which one must follow to get down to the slot. <Value> (which is only needed when it makes sense) should evaluate to the value to be put into the slot (or otherwise used in performing the function). At this point, we will only describe the two <Selector>s corresponding to accessing and assigning. These are **get** and **put** respectively. Thus to access the value of a slot, you would use

```
(path get <Structure> <Slot-Name-or-List>)
```

(No value is needed; the purpose of this call is to get the value.) To assign a value to a slot, you would use

```
(path put <Structure> <Slot-Name-or-List> <Value>)
```

For example, to access the Actor slot of the PTrans in Trip, either of the following is appropriate:

```
(path get Trip 'Actor)
(path get Trip '(Actor) )
```

This is essentially equivalent to a reference to *Trip^.Actor* in Pascal.

To access a slot within a structure in a slot of type *struct*, add the slot names to the <Slot-Name-or-List>, just as we access embedded fields within fields in Pascal by adding more slots to the accessing expression. For example, to access the place John told Carol he was going in our MTrans example above, we want the To slot of the MObject slot of the MTrans stored in InformLeaving:

```
(path get InformLeaving '(MObject To) )
```

This is essentially equivalent to a reference to *InformLeaving^.MObject^.To* in Pascal. PEARL will check each slot reference, and will indicate if a slot name is not found (perhaps due to a misspelling or an unbound slot).

Similarly, to change the Actor of our PTrans in Trip to be Bob:

```
(path put Trip '(Actor) (getsymbol 'Bob) )
```

and to change the To slot within the MObject of the MTrans:

```
(path put InformLeaving '(MObject To) (getsymbol 'School) )
```

which is essentially equivalent to assigning a value to *InformLeaving^.MObject^.To* in Pascal. Note that the order of the arguments to these functions is in **not like** the argument ordering of *putprop*.

CAUTION: *Path* does not check values to ensure that they are of the correct type before putting them in a slot. Also, a change in a structure with *path* does not cause it to be reinserted in the data base (see the next section). Thus, before changing a structure, one should remove it from the data base and then reinsert it after the change.

These functions were gathered under the macro *path* because of their similarity. However, if you prefer to have the action being performed lead off the function name in keeping with *putprop*, *get*, *putd*, *getd*, etc., these two functions are also available as **putpath** and **getpath** with similar names also provided for all the other forms of path described below. Thus the name "path" may be tacked onto the end of one of the action selectors to *path* but the rest of the arguments to these functions remain the same.

There are quite a few other operations which are allowed through *path* which you will not need or understand until you have read the rest of this documentation. We describe them here for completeness but suggest you skip to the next section during your first reading. If you feel there is one missing, feel free to suggest it since they are easy to add.

path **clear** or **clearpath** -- sets the selected path to the standard default value for its type (*nilsym*, *nilstruct*, zero or *nil*). Note that this is only the standard default and does not inherit a default from above. See section 7 for more on default values.

path **addset** or **addsetpath** -- add the specified value to a slot of type *setof*.

path **delset** or **delsetpath** -- delete the specified value from a slot of type *setof*.

path **getpred** or **getpredpath** -- get the list of predicates on the slot.

path **addpred** or **addpredpath** -- add the specified predicate to the predicates on the slot.

path **delpred** or **delpredpath** -- delete the specified predicate from the predicates on the slot.

path **gethook** or **gethookpath** -- get the assoc-list of hooks (demons) on the slot.

path **apply** or **applypath** -- arguments to this function are a <Function-or-Lambda-Body>, followed by the <Structure>, and <Slot-Name-or-List>. The <Function-or-Lambda-Body> is applied to the value of the slot. (In the UCI Lisp version, *apply#* is used so that macros will work. In the Franz Lisp version, a PEARL-supplied version of *apply* called **apply*** is used which also handles macros "right".)

(Skip this next paragraph until you have read about hashing and the data bases.) The method of processing the path in **path** functions allows a form of indirection through the data base that is sometimes helpful when you use symbols in slots as unique pointers to other structures. Suppose you had the following definitions:

```
(create base Person
        (* Identity symbol)
        (  Name lisp) )
```

```
(dbcreate individual Person
          (Identity John)
          (Name (John Zappa) )
```

and you want to ask "*what is the Name of the Person in the Actor slot of Trip (above)*" which you might normally write as:

```
(getpath (fetch (create pattern Person
                         (Identity ! (getpath Trip 'Actor) ) ) )
         'Name)
```

This is very hard to understand. A shorthand for this is the following:

```
(getpath Trip '(Actor Person Name) )
```

which behaves like this: when *path* gets to the symbol in the Actor slot of Trip, it notices that there is still more path to follow. It then interprets the next symbol in the path as the name of a type and does a quick equivalent of fetch of a Person with its first slot set to John. It then continues to follow the path specified in this new structure, finishing up with the value of the Name slot of the structure. (Note that this depends on Person structures being hashed by the relevant slot and will fail otherwise. Also note that the tendency of most users of PEARL has been away from the use of symbols as indirections to larger structures and toward actually putting the larger structure in the slot. In this case this would imply putting the Person structure in the Actor slot of PTrans and eliminate the need for "Person" in the path list.)

6. Storing In and Retrieving From the Data Base – The Simplest Way

So far we have shown how to create structures and have treated them pretty much like C structures or Pascal records. However, PEARL's most important departures from these languages involve its data base facilities. PEARL's data base can be thought of as one large bag into which any structure can be placed. The data base can hold hundreds, even thousands of separate objects. There are two basic operations that can be performed upon the data base, inserting with the function *insertdb* and retrieving with a combination of the functions *fetch* and *nextitem*.

6.1. Storing In the Data Base: *Insertdb* and *Removedb*

While the simplest forms of these actions are relatively straightforward, the power and efficiency of PEARL derives from the nuances and controls available with these functions which take up much of the rest of this document. Much of the power develops from knowledge provided by the user about how each kind of structure is likely to be retrieved (and therefore how it should be stored). Thus, the data base benefits from knowing as much as possible in advance about the objects that will be placed within it. This information is provided by using a large variety of extra flags during definition calls to *create*. It is used by *insertdb* to hash objects into a specific *hash bucket* in the data base, by *fetch* to retrieve the correct hash bucket from the data base, and by *nextitem* to filter the desired objects from this bucket.

PEARL allows the construction and use of multiple data bases which are described in detail later. Without exerting any effort, a data base is automatically created called *maindb* and pointed to by the special variable ***db***. In general, all PEARL functions which deal with a data base have an optional last argument which specifies which data base to use. If it is missing, then the default data base pointed to by **db** is assumed. Thus you can change the default data base simply by assigning the desired data base to **db**. For simplicity, this optional data base argument is not mentioned in the following discussion.

The function **insertdb** takes a single structure argument and inserts it into the data base. In its simplest form *insertdb* requires no user flags on the definitions of structures. In this case, *insertdb* simply hashes on the type of the structure regardless of its specific contents so that each entry ends up in a bucket with all other entries of that type. For example, to insert into the data base the PTrans which was saved in the Lisp variable Trip above, you simply provide it as an argument to *insertdb*:

```
(insertdb Trip)
```

We could also put the PTrans (saved in Leaving whose To slot was changed to School) which was the MObject of the MTrans above in the data base with:

```
(insertdb Leaving)
```

Since no information has been provided by the user about how to efficiently distinguish PTranses in general, these two will be stored in the same bucket (as will all PTranses). When inserting an item into a bucket, *insertdb* will check to ensure that this specific item is not already in that bucket (using *eq*) and will only insert it if it is not already there, thus avoiding duplicates.

The function **removedb** takes a single structure argument and removes it from any place in the data base where it has been put using *eq* to determine equality.

Since one often wants to create an individual and then insert it into the data base, there is a macro **dbcreate** provided whose arguments are precisely like *create*. Thus, *(dbcreate individual PTrans ....)* expands into *(insertdb (create individual PTrans ....))*.

6.2. Retrieving Hash Buckets From the Data Base: Fetch

The simplest case of fetching from the data base is equivalent to asking if a particular, completely defined object is in the data base. This is performed by a combination of the functions flfetch and *nextitem*. The first step is to retrieve the hash bucket(s) for the object. For example, to determine whether the object stored in Trip is in the data base, the first step is to call the function **fetch** and store what it returns (the form of what is returned is described below):

```
(setq PotentialTrips (fetch Trip) )
```

The function *fetch* takes a single structure argument which is called the **pattern**. What *fetch* returns includes this pattern and the hash bucket(s) from the data base which contain those structures which are most likely to "match". The rules of "matching" are fairly complex and are described in detail in section 20, but for the moment it is enough to know that two structures match if their respective slots contain equal values. Thus matching is closer to Lisp's *equal* than to *eq*.

6.3. Accessing the Results of a Fetch: Nextitem.

Conceptually, what *fetch* returns is a restricted type of **stream**. A stream is a "virtual" list, whose items are computed only as needed. When a fetch from the data base is performed, the pattern provided is only used to construct a stream containing that pattern and the appropriate hash bucket from the data base; no matching (comparing) between the pattern and objects in the data base occurs. Thus the stream contains pointers to all data base items in the same hash bucket, regardless of their likelihood of matching the pattern. Therefore, the *stream* or "virtual list" returned by *fetch* is in fact bigger than the list of actual items which match. (For this reason, the default PEARL print function only prints how many potential items are in the stream.)

For our purposes, you should regard the object that *fetch* returns to be a funny sort of black box, whose only use is as an argument to the function **nextitem**. *Nextitem* will compute the next element to be removed from the stream. When elements are extracted from the stream with the function *nextitem*, the pattern is "matched" against successive items from the hash bucket until one matches (and is returned) or until the potential items run out (and *nil* is returned).

Nextitem is very much like the function *pop* in Lisp because it updates the status of the stream to reflect the extraction of the "topmost element" similar to the way *pop* replaces its list argument with its *cdr*. *Nextitem* does this by destructively modifying the stream (but not the hash bucket); once the top item has come off it is no longer a part of the stream. Like *pop*, *nextitem* returns *nil* if the stream is empty.

A stream, as returned by *fetch* in PotentialTrips, will **never** be *nil* but instead will be a list. In all cases, the first element will be the atom ***stream***. In most cases, the second element (*cadr*) is the pattern (object being fetched) and the rest (*cddr*) is the hash bucket that the object hashes to. However, it is entirely possible for the hash bucket to either fail to contain any instances of the object, or to contain multiple instances of the object. The form that is printed by PEARL's default print function is: the atom ***stream:***, the object being fetched, and the number of potential items in the stream, avoiding the prining of a lengthy result. (If the pattern is actually a function structure, then the atom used is ***function-stream:***.)

Thus, to actually determine whether the object in Trip is in the data base, it is necessary to ask for the *nextitem* in the bucket of the stream PotentialTrips (that is, in the *cddr*) which matches the object being fetched (that is, the *cadr* of PotentialTrips):

```
(setq FirstResult (nextitem PotentialTrips) )
(setq SecondResult (nextitem PotentialTrips) )
```

If nothing matching Trip occurred in the data base, FirstResult would contain *nil;* otherwise, it would contain an object in the data base which matches Trip. If you have typed in all the examples we have shown you above, then FirstResult will contain the same value as Trip. SecondResult will be *nil*. (The only other object in the same bucket is the value of Leaving, but that does not match because its To slot contains School after the *path put* above.) If the two structures in Trip and Leaving both contained the same slot fillers, they would both match Trip and each would be returned by *nextitem* on successive calls.

This is essentially the only type of fetching that is useful with the information presented so far, but more powerful types will be described shortly.

Since the functions *create*, *fetch*, and *nextitem* are often used in combination, several macros combining them are provided by PEARL:

When you wish to create a pattern only long enough to use it as an argument to *fetch*, you can use the macro **fetchcreate** which is defined in such a way that *(fetchcreate blah)* is equivalent to *(fetch (create blah))*).

If you want to do a *fetchcreate* in a function definition and you wish the pattern to be created only once but used each time this function is called (a potential savings in space and time), the macro **inlinefetchcreate** will call *create* when it expands and then expand to a call to fetch with this pattern.

If you want to do a *create* in a function definition and you wish the object to be created only once, the macro **inlinecreate** will call *create* when it expands and effectively replace itself with the result.

When you wish to fetch but only need the resulting stream long enough to use it as an argument to *nextitem*, you can use the macro **firstfetch** which is defined in such a way that *(firstfetch blah)* is equivalent to *(nextitem (fetch blah))*).

If your only goal in fetching some fully-specified object is to test for its existence in the data base, the function **indb** which expects a single structure argument will return *nil* if it is not there, and non-*nil* if it is. (Note that *indb* uses *strequal* rather than *match*.)

It is sometimes convenient to have a list of all the items which would be returned by successive calls to *nextitem* on a stream. The function **streamtolist** expects a stream argument and returns a list of the items which the stream would produce.

7. The Default Values for Unspecified Slots

When creating an instance of a given type, one may not always wish to fill in all the slots of the structure, either because the slot value is unknown or immaterial. PEARL has a mechanism for filling unfilled slots with default values. The simplest form of defaulting involves two predefined objects, **nilsym** and **nilstruct**. *Nilsym* is a *symbol*, and roughly corresponds to Lisp's *nil* when *nil* is viewed as an atom. *Nilstruct* is a structure without any slots, and corresponds to a null structure. In the absence of a specified value, PEARL will fill in slots of an individual instance of a structure being created with *nilsym* if the slot type is *symbol*, *nilstruct* if the slot type is *struct*, zero if the slot is of type *int*, and Lisp *nil* if the slot is of type *lisp* or *setof <any type>*. Note that it is up to the user to decide upon the meaning of *nilsym* and *nilstruct* during further processing. For example, you must decide for your own application whether a *nilstruct* filling the

MObject slot of a MTrans indicates that nothing was said or that what was said is unknown.

Often you may desire closer control over the default values of a particular slot within individual instances. For example, suppose you had a definition of Person that includes several pieces of information about a person:

```
(create base Person
         (Identity symbol)
         (Age int)
         (Salary int)
         (Health symbol) )
```

The Identity slot would be filled in with a unique symbol for that person (such as the symbol John), the Age slot would contain the age in years, the Salary slot would get the person's monthy salary in dollars, and the Health slot would contain a *symbol* indicating their state of health. Now in creating an individual instance of a Person the Identity slot should be always filled in, but we may desire the other slots to be defaulted to 30 (years), 20000 (dollars) and Healthy. However, under the default system described so far, these would be defaulted to zero, zero and *nilsym*. PEARL provides the ability to specify individual defaults for each slot of a particular structure type. This is done at *base* creation time by following the type within a slot with the new default value. Thus our definition of Person would be:

```
(create base Person
         (Identity symbol)
         (Age int 30)
         (Salary int 20000)
         (Health symbol Healthy) )
```

Although the main purpose of a call to *create base* is to define a structure, it also creates a special individual of the type being defined which has its slots filled with the default values. For this reason this individual is called the **default instance** of that type. It is a structure instance like any other, only a pointer to it is kept with the type definition, and it is consulted whenever an instance of that type is constructed. Thus the default values (either the user-defined defaults or the standard defaults) will always be used whenever the user declines to fill in a slot of a structure instance. For more on defaults, see the discussion of inheriting in section 19 on creating expanded structures.

8. Using Patterns For More Flexible and Powerful Retrieval

If the fetching mechanisms described so far were the only sort of fetching that we could do, *fetch* (and PEARL) would not be very useful. What is needed is a way to only partially specify the values in the structure to be fetched. Note that the default mechanism does not accomplish this, since all slots are specified at creation time, even if they get *nilsym*, *nilstruct*, or *nil* for a value. What is needed is the ability to specify a *don't care* value for slots whose values should not affect the matching process during retrieval. The easiest way to accomplish this in PEARL is to create a type of object called a **pattern**. A *pattern* is similar to an *individual* instance of a structure except that a special pattern-matching variable called **?*any*** which means *don't care* or *match anything* is used as the default value for unspecified slots. (The reason for its name will be clear after the description of pattern-matching variables later in this section. Even more detail on pattern-matching variables and more powerful patterns is provided in sections 21-23 on the matching process, blocks, lexically scoped variables, and the unbinding of variables.)

Patterns are created with *create* using *pattern* as the first argument. Other than this, their syntax is exactly the same as individuals. An example of a *pattern* creation is:

```
(create pattern PTrans JohnWentSomewhere
        (Actor John)
        (Object John) )
```

This pattern would match any instance of PTrans in which John was both the actor and the object being moved. (Note that this pattern is stored in the Lisp variable JohnWentSomewhere in the same way as other individuals.) The main use of patterns is as arguments to *fetch*, as in:

```
(setq PotentialGoings (fetch JohnWentSomewhere) )
```

Fetch will return a stream containing all PTranses in the data base in which John was the actor and object, regardless what the From and To slots contain. More complex examples can be created. Patterns can be embedded as in:

```
(create pattern MTrans InformJohnGoingSomewhere
        (MObject (PTrans (Actor John)
                         (Object John) ) ) )
```

Since all unspecified slots are filled with ?*any*, this pattern will return any MTranses concerning any of John's PTranses when passed to *fetch*. Thus, if we insert InformLeaving from above in the data base:

```
(insertdb InformLeaving)
```

then the following will fetch that object:

```
(nextitem (fetch InformJohnGoingSomewhere) )
```

Usually one is interested in a more specific piece of information. For example, if you knew that John told Carol something and wanted to find out what it was, then you could do this two ways. One is to create a pattern, fetch it and look at the MObject slot of the result:

```
(create pattern MTrans WhatDidJohnTellCarol
        (Actor John)
        (From John)
        (To Carol) )
(setq Result (firstfetch WhatDidJohnTellCarol) )
(path get Result 'MObject)
```

However, you might prefer to use a pattern which explicitly shows that you want that value and gives you a slightly easier way to get at it. In this case, you can specify a pattern-matching variable in the MObject slot of the pattern. A pattern-matching variable is created by preceding an atom with a question mark ? as in ?*What*. The question mark is a read macro character which reads the next atom and builds the list *(*var* What)* out of it (or *(*global* What)* if *What* has previously been declared global to PEARL; see below for more on global variables.). During matching, this variable will get bound to the value of the slot it gets matched against:

```
(create individual MTrans WhatDidJohnTellCarol
        (Actor John)
        (MObject ?What)
        (From John)
        (To Carol) )
(firstfetch WhatDidJohnTellCarol)
```

To access the value of a pattern-matching variable in a structure, one uses either the function **valueof** (which is an expr) or the fexpr **varvalue**. Both functions have two arguments: the name of the pattern-matching variable whose value you want and the structure it occurs in (which is evaluated internally by *varvalue*). Thus both of the following are equivalent:

```
(valueof 'What WhatDidJohnTellCarol)
(varvalue What WhatDidJohnTellCarol)
```

9. Marking Structures During Creation For More Efficient Retrieval

Besides specifying what type each structure is, the simplest piece of information that *insertdb* would like the user to give it through *create* concerns which slot(s) of a type would be most likely to contain unique information about a particular instance of that type. This information is used to differentiate instances of the type from each other, so that they will be hashed into different hash buckets. This is similar to the "keys" that many data base systems ask for. For PTrans, the Actor slot is often the best choice for this role. For Person, the Identity slot would be the best choice for this role. Such unique slots are indicated to *create* when defining a type by placing an asterisk '*' before the slotname in a (<slotname> <type>) pair. For example, our new definitions of PTrans and Person to take advantage of this would look like:

```
(create base PTrans
        (* Actor symbol)
        (  Object symbol)
        (  From symbol)
        (  To symbol) )
(create base Person
        (* Identity symbol)
        (  Age int 30)
        (  Salary int 20000)
        (  Health symbol Healthy) )
```

If you execute this when you have already executed the other examples in this document, PEARL will warn you that you are redefining the base structures PTrans and Person. This is all right, since that is precisely what we want to happen. However, the previous instances of PTrans will remain hashed in the more inefficient way and will not match any later PTrans structures that are defined. If you find these warnings annoying when redefining structures, they may be turned off by setting the special variable ***warn*** to *nil* instead of the initial *t*. (Note that the Lisp scanner requires a space (or other white space) to separate the asterisk from the slotname. Otherwise one would have the slotname **Actor*).

Any number of starred slots may be provided within a structure definition, but usually only one or two are necessary. How one decides which slots should be starred is an art, and depends significantly on your application and the nature of your data. The basic rule of thumb is to choose those slots whose values vary the most widely from instance to instance. A bad choice will not usually cause the system to bomb or operate incorrectly in any way, but when it comes time to fetch an object back out of the data base the system may have to take the time to scan a large amount of the data base rather than finding the desired object very rapidly. Thus execution time is usually the only penalty one pays for an improper choice of slots to star.

However, there is one type of use of a slot which can cause problems in combination with hashing information. It involves the use of pattern-matching variables and will serve as a useful example of how to use variables and of how *insertdb* and *fetch* use the hashing information to insert and find objects. The problem situation occurs when you wish to insert items into the data base which contain a variable in the starred slot (representing general knowledge) and then use, as a pattern, a structure with that slot filled. Thus, the following sequence will fail to find Thing in the data base and instead will return *nil*:

```
(create base Thing
        (* One int) )

(dbcreate individual Thing DBThing
          (One ?O)
          (Two 2) )

(nextitem (fetchcreate individual Thing PatThing
                       (One 1)
                       (Two 2) )
```

This fails *simply because of the hashing*. Let us see why. When *insertdb* is asked to put something into the data base, it seeks to put it as many places as the hashing information indicates are good places to want to look for it. With no hashing information at all, the only thing *insertdb* can do is to put the object with all other objects of its type. Thus, with no hashing information, all Things are put together in the same hash bucket. With the hashing information, *insertdb* would like to put DBThing in a second place based on the fact that it is a Thing *and* on the value of its One slot. Unfortunately, its One slot has an unbound variable in it and does not provide any information which is useful. Thus the hashing process puts DBThing into the data base in only one place. However, when *fetch* indexes PatThing, it uses the hashing information to determine that it should look in the data base under the combination of Thing and 1. Since DBThing is not there, it is not found. If we remove the asterisk, this sequence will return DBThing with ?O bound to 1 because both DBThing and PatThing will get indexed into the same spot (along with all other Things). Thus you should be very careful when determining how to hash types of structures when you intend to insert them into the data base with variables in them.

After more of the system has been described and examples of fetching and inserting have been given the user will have a better understanding of this process.

As another example, let us now create and insert an instance of our new PTrans which has the Actor slot starred:

```
(dbcreate individual PTrans Trip
                (Actor John)
                (Object John)
                (From Office)
                (To Home) )
```

This would insert Trip with all other PTranses and, because of the starred slot Actor, also with any other PTranses with a value of John in the Actor slot. Next we redefine and recreate the MTrans:

```
(create base MTrans
          (* Actor symbol)
          (  MObject struct)
          (  From symbol)
          (  To symbol) )

(create individual MTrans InformLeaving
          (Actor John)
          (MObject (PTrans Leaving
                              (Actor John)
                              (Object John)
                              (From Office)
                              (To Home) ) )
          (From John)
          (To Carol) )
```

reinsert the PTrans from the MTrans:

```
(insertdb Leaving)
```

and finally create and insert two other instances of a PTrans, one with different values in the From and To slots and one with different values in the Actor and Object slot:

```
(create individual PTrans Trip
          (Actor John)
          (Object John)
          (From Home)
          (To School) )

(create individual PTrans
          (Actor Ted)
          (Object Ted)
          (From Office)
          (To Home) )
```

Note that this last PTrans will be indexed under the combination of PTrans and Ted and thus will not be in the same hash bucket we look in when fetching Trip (which is indexed by PTrans and John):

```
(setq PotentialTrips (fetch Trip) )
(setq Result (nextitem PotentialTrips)
PotentialTrips
```

Notice the form of the stream PotentialTrips at this point.

10. Printing Structures, Symbols and Other PEARL Objects

As mentioned in the beginning, PEARL stores symbols and structures (and their definitions) in hunks of memory that are circularly linked. Lisp cannot print out the contents of these blocks in a useful way. Franz Lisp sometimes goes into an infinite loop trying to print them and the best UCI Lisp can do is tell you its address, like #2934, which is not very informative. As mentioned above, the PEARL prompt-read-eval-print loop knows how to print these in symbolic form. However, when you want your own programs to print them, PEARL provides you with two pairs of functions to convert these blocks into more readable form. The first we will discuss is the function **valform**. *Valform* takes a *struct*, a *symbol* or any other type of PEARL or Lisp object as an argument and returns a Lisp S-expression describing the object. Thus if one calls *valform* on our PTrans in Trip:

```
(setq TripAsList (valform Trip) )
```

the Lisp variable TripAsList will contain the S-expression:

```
(PTrans (Actor John) (Object John) (From Home) (To School) )
```

Note that *valform* does not cause the PTrans to be printed out at the user's terminal, it is merely a function that returns an S-expression (just as the Lisp function *list* does.) PEARL functions will operate upon structures and symbols only when they are in their internal form, so the primary reason for converting structures to S-expressions is to print them (or to modify them for use as new input to *create*). So PEARL provides a more useful function **valprint** that is effectively *(sprint (valform <argument>))*. (**Sprint** is a function provided by UCI Lisp or Franz PEARL which prints a Lisp expression in a nicely indented form. There are more details on *sprint* in the appendix on UCI Lisp functions added to PEARL.) *Valprint* is normally used within a Lisp program to print out any PEARL construct onto the user's terminal and it is also used by the default print function in the PEARL prompt-read-eval-print loop. Try typing the following and notice that they are the same except that the second value is slightly indented.

```
(valprint Trip)
Trip
```

Like *sprint*, *valprint* will accept a second integer argument telling it which column to start printing in. The default *pearlprintfn* uses a value of 4 for this argument to make the items typed by the user more distinguisable from the results typed by PEARL.

There is one other form of each of these two functions. The functions **fullform** and **fullprint** are like *valform* and *valprint* but they print more complete information. If you type

```
(fullprint Trip)
```

you will notice that the result has two mysterious *nils* in each slot. These represent other forms of information (predicates and hooks or demons) which can be added to structures and which will be described later. At the moment therefore, *valform* and *valprint* are all that the user need remember.

Note also from above that when a pattern with ?**any** is printed, only the name of that variable is printed, and not its value. (Try typing JohnWentSomewhere and InformJohnGoingSomewhere if you do not remember what this looked like.) If a PEARL pattern-matching variable has not been bound, PEARL indicates this by printing no value. If a variable is bound, both the variable name and its value are printed. Later when you learn how to put your own variables in slots, this will become more useful.

When given a data base, these functions assume that the user does not really want the complete contents of the data base printed out and so simply print *(database: <databasename>)*. To actually have the complete contents of a data base printed out, use the function **printdb**. With no argument, it prints the default data base. Otherwise, it expects its argument to evaluate to a data base. A print function which prints all the internal information contained in a structure or its definition is **debugprint**.

11. Error Messages, Bugs, and Error Handling Abilities

Due to the complex implemention of PEARL and the lack of facilities in Lisp for easily distinguishing between types of input, a user's error in using PEARL will not show up as soon as it occurs, but may instead cause some unfathomable part of PEARL to bomb out sometime later. If this should happen, the user might try using the Lisp trace facilities, but will often find little useful information. This sad state of affairs is currently unavoidable due to the difficulty of catching user errors where they first occur. This is partly due to our inability to predict what kinds of errors users are most likely to make.

PEARL checks as much as it can, but many features are impossible or prohibitively expensive to check. The best strategy for the user to follow is to examine his last interaction with PEARL. If the error occurred in the bowels of *create*, then there is a good chance that the user did something wrong in the details of a slot description in the call to *create*, since gross structural errors in such calls are checked for. Inspect your call closely.

Other errors can be even more difficult, since a function call may blow up or fail to produce the desired result due to bad data passed to *create* several calls ago. A general rule of thumb to use in tracking down mystifying errors is to check out the definitions of the structures involved in the function that failed. Thus if *path* blows up, one should determine the type of the structure passed to *path*, and then check the *create* call that defined that type.

Sometimes PEARL may appear to the user to be doing the wrong thing, but due to PEARL's complex semantics, the user is really at fault. To make matters worse, there is of course always the chance that the error really **is** in PEARL. Every effort has been made to minimize this chance, and at the moment there are no known major errors (except those indicated in this documentation), but as with any complex evolving software system there is always the chance of obscure errors. It has been our experience that most errors are due to the user (including the implementors) not completely understanding the semantics of some PEARL feature. This documentation is an effort to minimize this type of error. For any error which you commit in which PEARL gives what you consider an unreasonable response, feel free to report it and we will consider trying to catch it. These or any other complaints, bugs or suggestions should be mailed to Joe Faletti via Arpanet or Unix mail (Kim.Faletti@Berkeley or ucbvax!kim.faletti).

12. Short-Circuiting and Redirecting Create Using !, $ and Atoms

Sometimes, when creating an individual structure, one may want to fill a slot with an already created structure that is pointed to by some atom or returned by some function (or with whatever type of value the slot requires). In this case, one does not wish to (or cannot) describe the value for a slot as a list of atoms. To handle this situation, PEARL allows you to list a Lisp expression which evaluates to the desired internal form (that is, a form which needs no processing by *create*), preceding it with an exclamation point "!". The structure (or other object) resulting from evaluating the Lisp expression will be tested to ensure it is the right type of value and, if it is, inserted in the newly created structure as the value of that slot. (The mnemonic idea of this symbol is as a sort of "barrier" meaning *Stop processing here!!! and take this (almost) literally!!!*) For example, after using

```
(create individual PTrans Ptrans23
        (Actor John)
        (Object John)
        (To Home) )
```

to create an individual PTrans, leaving it in internal form in the atom Ptrans23, you can then insert this PTrans into a new MTrans with:

```
(create individual MTrans
        (Actor Bob)
        (MObject ! Ptrans23)
        (To Carol) )
```

At other times the user may want to take the result of evaluating some Lisp code and splice it into the Lisp expression describing the structure being created at the point where the description of the value of a slot would occur. In this case, you wish some Lisp code to be evaluated and then you wish *create* to build a value for this slot by further processing (scanning) the result of this evaluation. To this end, PEARL will evaluate any slot value preceded by a "$" and insert its result into the *create* call, proceeding to process it just as if the user had typed it in directly. So if one stores the atom Alice in Name with

```
(setq Name 'Alice) ; the atom Alice, not the symbol Alice
                   ; (or the value of s:Alice).
```

or possibly

```
(setq Name (read) )
```

with the user having typed *Alice*, then *$ Name* in

```
        (create individual  PTrans
                (Actor $ Name)
                (Object $ Name)
                (From Home)
                (To NewYork) )
```

is equivalent to having Alice typed as the Actor and Object values: *create* evaluates Name and then processes its value *Alice* as input. Thus, the PTrans will be equivalent to

```
        (create individual  PTrans
                (Actor Alice)
                (Object Alice)
                (From Home)
                (To NewYork) )
```

The power of this construct occurs when Name is a atom whose value changes at run time (as when it is read above) or the *create* call is within a loop in which Name takes on many different values.

In summary, both ! and $ cause the evaluation of the Lisp expression following them. However, ! stops the usual processing and expects an internal value, whereas $ continues the usual processing and expects a Lisp description of the value. When you need either ! or $, you will know it! Until then, do not worry if you do not understand them very well!

There is currently one type of slot which allows the effect of ! without requiring that the ! be typed. In *int* slots, if PEARL finds an atom which evaluates to an integer, rather than an actual integer, it will evaluate the atom and store the result as the value of the slot. (PEARL will eventually be modified to try this in other cases also.)

13. More Flexible Hash Selection

The use of stars (asterisks *) to indicate useful slots for hashing described earlier is only one of many hashing schemes that PEARL allows. This section describes the others, and how the user can control them. The first point to note is that even with the star hashing a single structure can be hashed in several different ways. Thus if one thinks that in a particular program PTrans will be frequently fetched from the data base given only the Actor **or** only the Object (that is, the program might only know the Actor and desire the whole PTrans, or know only the Object and desire the whole PTrans) the user should star **both** the Actor and Object slots within the definition of PTrans. When PEARL stores a PTrans into the data base, it will index it under both (PTrans + Actor) and (PTrans + Object) in addition to the usual indexing with all other PTranses. In general, any number of slots can be starred.

Another type of hashing does not use the type of the structure in creating a hash index. If the symbol colon (:) is used before the name of a slot, objects of that type will be hashed under that slot value only. Thus if the Actor slot of the PTrans definition was preceded by a colon instead of a star, then instances of PTrans would be hashed under the value of the Actor slot alone rather the value of the (PTrans + Actor). This would be useful in the case in which one were interested in fetching any structure in which a particular value, say the symbol John, appered in a coloned slot. For example all structures in which John appeared in the Actor slot could be fetched at once (and very efficiently).

A third type of hashing is **star-star** or **double-star (**) hashing**. If a double star is used instead of a single star, PEARL will use **triple hashing**. Only one triple hashing is allowed per structure. Triple hashing requires that two, and only two slots be double starred. If PTrans were to be defined in the following way:

```
(create base  PTrans
              (** Actor  symbol)
              (** Object symbol)
              (    From  symbol)
              (    To  symbol) )
```

then when an instance of a PTrans is created, it will be hashed into the data base under a combination of the three values (PTrans + Actor + Object). As with all hashing, if a slot is necessary to a particular type of hashing but is unfilled (or filled with *nilsym* or *nilstruct*) the hashing will not occur. Triple hashing is used when one wants fast access to all individuals of a particular type with two slots likely to have fairly unique values. In the case of PTrans, this would allow one fast access to all PTranses in which John PTranses Mary somewhere. Distinctions this fine are not usually necessary, and as it is slightly more expensive at creation and fetching time, it should only be used when the user is sure of the need for it.

A fourth type of hashing is **colon-colon** or **double colon (::) hashing**. It has the same relation to colon hashing as double star hashing has to star hashing. If the **'s in the above are replaced with ::, the hashing will be on (Actor + Object) ignoring the fact that the structure is a PTrans. This might be useful in fetching all structures involving John and Mary. As with double star hashing, double colon hashing should be used sparingly and only one such hashing pair may be used in a type.

However, it is possible to combine the use of any of these hashing methods in a single structure. Thus one could have both double colon hashing and double star hashing, as well as several * and : hashings as well. Given several ways, PEARL uses the one which the most complex one is used during fetching, since that should provide the greatest degree of discrimination between items (that is, most likely to narrow down the choices). If the value in a slot intended to take part in hashing is unbound or otherwise not useful, then the next most specific method it used. Given the values which are considered useful and the hashing information for the type of structure, the hierarchy of buckets to be chosen is as follows:

** hashing
:: hashing
* hashing
: hashing

In addition to these four methods of hashing, there are several hashing labels which are modifiers on these methods and affect what values are used to compute the index.

The remaining hashing flags do not introduce any new types of hashing, but rather modify the way the existing types work. To motivate these, consider the implementation of Goal withing CD. In early versions of CD, there were several different types of goals, including Delta-Prox (goal of being near something), Delta-Poss (goal of possessing something), and so on. In general these delta goals were of the form (Delta-<some CD primitive> (Actor ...) (Objective ...)). This lead to an explosion of Delta-goals (e.g. Delta-Move-Fingers-Within-Telephone-Dial), and a new way of handling goals was established. This was simply that all Goals were to have the form:

```
(create base Goal
        (Planner symbol)
        (Objective struct) )
```

where the Objective would be filled with the appropriate structure, whether it was a simple Poss or the $DialPhone script. This change makes CD much cleaner, but poses somewhat of a problem for hashing. One of the major uses of hashing within some AI programs written in PEARL is to associate plans with goals. So it is best if this process is efficient.

As an example of this problem (using the early form of Delta-goals):

```
; Declaration of PlanFor rules.
(create base PlanFor
        (* Objective struct)
        (* Plan struct) )

(create base Delta-Prox
        (Planner symbol)
        (Location symbol) )

(create base Walk-Plan
        (Planner symbol)
        (From symbol)
        (To symbol) )

; Store in the data base the fact that walking is a way of accomplishing
;   a Delta-Prox goal.
(dbcreate individual PlanFor
           (Goal (Delta-Prox (Planner ?X)
                             (Location ?Y) ) )
           (Plan (Walk-Plan (Planner ?X)
                            (From nilsym)
                            (To ?Y) ) ) )
```

This structure simply says the fact that if one has a goal of being somewhere, then one plan for doing this is to walk. Or, using the rule in reverse, if you note that someone is walking to some location, then you might infer that they had a goal of being at that location. Note that after being put into the data base, the rule can be easily fetched by presenting either half of it as a pattern.

Thus if a planning program has a goal of doing the action in the atom GoalAct, then it can query the data base for any direct plans for doing Act by:

```
        (fetchcreate individual PlanFor
                     (Goal ! GoalAct)
                     (Plan ?*any*) )
```

So if GoalAct happened to be a Delta-Prox goal, then the rule above would be fetched. However the revised form of goals hides the unique nature of the Delta-goal, and the best one could do is fetch all PlanFor rules that have a structure of type Goal in their Goal slot. This is a serious loss since *all PlanFors* have a Goal in their Goal slot; thus the system would have to look through all PlanFors whenever it was trying to fetch one. What is needed is a way of telling PEARL that when hashing on Goals, never hash the structure type Goal, but rather use the item that fills the Objective slot of the Goal. This would solve our problem nicely, as now all PlanFors would be hashed on the name of the Objective (Prox, Dial-Phone, etc.), and a list of all PlanFors would not have to be searched to find a particular one, rather the system could just hash directly to it.

To indicate to PEARL that this **hash aliasing** is desired, place an ampersand '&' before the slot name to be substituted for the structure name when defining the structure. Thus Goal would be declared:

```
        (create base Goal
                 (  Planner symbol)
                 (& Objective struct) )
```

Naturally only one slot can be selected for hash aliasing.

In this way, Goals change the way in which other structures use them to index but the way in which Goals themselves are indexed will not be affected. Since many other types of structures are likely to contain Goals, we must be careful about how this affects the hashing of all of them. It might be the case that PlanFor was the only structure indexed based on Goals which would benefit from hash aliasing and that some structures would actually be hurt by this because they expected Goals to be only one of many types of values. In this case, putting the control of how Goals get used by other structures into the definition of Goal is a bad idea. Instead, the control can be moved up into only the problematic structures. These structures can simply add the ">" hash label to a starred slot, causing PEARL to use the first starred slot of the slot-filling structure instead of its type. For example, when we put a both "*" and ">" on the Goal slot of PlanFor then it will always use the first starred slot of the Goal in its Goal slot:

```
        (create base Goal
                 (  Planner symbol)
                 (* Objective struct))

        (create base PlanFor
                 (* > Goal struct)
                 (    Plan struct))
```

Thus, the use of ">" hashing is called **forced aliasing** since the structure filling a slot has very little control over it.

However, there is one way for a structure to affect how forced aliasing happens. If the user wanted to also star the Planner slot of Goal, but wanted the Objective slot to be used in cases of forced aliasing, then the use of an "^" on the Objective slot will allow that:

```
(create base  Goal
        (*    Planner  symbol)
        (* ^ Objective  struct))
```

thus allowing Goals inserted directly into the data base to be indexed by the combinations *Goal + Planner* and *Goal + Objective* while other objects containing Goals would use the Objective slot rather than Goal *OtherObject + Objective.*

On the other hand, if most structures containing Goals would benefit from the use of the hash aliasing label "&" in Goal, but only one or two are hurt by it, the use of "&" in Goal can be overridden by the structures which will contain Goals by adding the "<" hash label to the starred slot to produce **anti-aliasing**. This gives the controlling structure the last word over how it is hashed.

```
(create base  Goal
        (  Planner  symbol)
        (& Objective  struct))

(create base  OffendedStructure
        (* < Slot  struct))
```

Thus, the anti-aliasing "<" means *just for this hashing, turn off hash aliasing (if any) of any structure filling this slot.*

The proper use of hash aliasing and anti-aliasing, like all the hashing specifiers is an art that must be learned by applying them to real systems, and the correct hash directives for a particular system rely critically upon the statistics of that particular system operating upon a particular set of data. The hashing mechanism was designed to give the user benefit in proportion to the effort expended in determining hash labels. With no effort, the structure type provides some help. With the addition of each label or pair of labels, an item to be inserted into the data base is indexed into another location in the hash table. Thus the cost of *extra* labels is simply the time to find another hash bucket (a few adds and multiplies), and add the item to the front of the list implying the time and space incurred by one cons-cell.

14. Using Predicates to Constrain Fetching

Sometimes when you are creating a pattern to fetch a structure, giving the overall form of the structure is not specific enough. In particular, it is often desirable to restrict the value of a slot to a subrange. For example, using the structure Health:

```
(create base  Health
        (Actor  symbol)
        (Level  int) )
```

one might want to find out who is sick by creating a pattern that only matches those Health structures in which the Level is less than -1 (on a scale from -10 to 10 perhaps). This can be done by simply writing a predicate (say Sick) which expects to be given the value of the slot being matched against as its one argument:

```
(de Sick  (Num)
    (lessp  Num  -1) )
```

Then you simply add its name after the value within the <slotname filler> pair of the pattern:

```
(create pattern Health HealthPattern
        (Actor ?Person)
        (Level ?Level Sick) )
```

Given these definitions, a (fetch HealthPattern) would pass the Level slotfiller of each Health structure it found in the data base to the predicate Sick. If Sick returned true (non-*nil*) then it would consider the slot to have matched whereas a *nil* from Sick would be considered a mismatch. There are no standard predicates for users to use for these purposes, but they are relatively easy to create as needed.

However, one often has a predicate which has more than one argument only one (or none) of which are the slot value. For example, one might want to include a special variable or the value of some other slot of the structure or the structure itself. To make this easy PEARL allows predicates to be arbitrary s-expressions which may contain any of several special forms for which PEARL substitutes the current slot or structure.

If a predicate includes an asterisk *, this is replaced by the value of the current slot (in the structure being matched against). If it includes a double asterisk **, this is replaced by the whole structure being matched against. If you want the value of another slot in the current structure, precede its name with an equal sign (as in **=SlotName** to have the value of the slot named SlotName inserted). There is a readmacro "=" which converts *=S* into *(*slot* S)*, just as the readmacro "?" converts ?X into *(*var* X)* (or *(*global* X)*) for pattern-matching variables. While processing predicates before executing them, PEARL will look for these three constructs and replace any of them with the appropriate value, so pattern-matching variables can also be used in predicates.

If there are several predicates on a slot, they are run in succession until one returns nil or they have all been run. Thus, a list of predicates provides the effect of a conditional *and*. Thus, although PEARL knows nothing special about logical connectives like *or* and *and*, the effect of a the usual Lisp *and* is automatically implied and the conditional *or* of Lisp can be had by using the s-expression type of predicate. If you wish things to run regardless of their results, providing the effect of unconditional *and*, use hooks (demons).

The above was one of two types of predicates available. To motivate the other type, consider the case of wanting to fetch all MTranses about the occurence of a PTrans. This could be accomplished in one of two ways. The first is:

```
; In this pattern example, all slots are automatically filled
;    with ?*any* except the MObject which must be a PTrans.
(create pattern MTrans
        (MObject (PTrans) ) )
```

Since this method actually results in ?*any* being matched against the fillers in each of the PTrans's slots, it is a bit inefficient.

The second way uses **structure predicates** to avoid this matching by specifying merely that the filler of the MObject slot must be a PTrans structure. This is done by listing the name of a previously defined structure after a pattern-matching variable:

```
        (create pattern MTrans
                (MObject ?Obj PTrans))
```

PEARL will then bind Obj to any structure that is a PTrans (or expanded PTrans) and match successfully without examining any of the slots of that PTrans. PEARL can tell the difference between these two types of predicates since one will have some sort of function declaration and the other will be the name of a defined structure. In the case of a function with the same name as a structure (which the user should never do as it invites errors) the name's structure role takes precedence.

Since a similar effect is sometimes desired on slots of type *symbol*, a similar but more complex mechanism is provided with symbols and with structures which failed the above test. If the name of a predicate on a slot of type symbol or structure is the name of a type of structure, PEARL will assume that what you want to know about the value in this slot is whether there is anything in the database of the type specified by the structure predicate with the slot value in its first slot. Thus, if the data base contains an item saying that the symbol John represents a person:

```
        (symbol John)
        (dbcreate individual Person
                  (Identity John))
```

then fetching a pattern with a symbol slot which has a Person predicate on it:

```
        (fetchcreate pattern Thing
                     (Slot ?X Person))
```

will cause the equivalent of a fetch from the (default) database of the pattern (Person (Identity John)). Note that this implies that the first slot of a structure enjoys somewhat of a pre-eminence and that this means that one should carefully choose which slot to put first. For efficiency however, ***fetch*** is not actually used. The function actually used is **disguisedas** which expects the slot filler, the structure definition (not default instance) and an optional data base to look in. Slot filler may be either a symbol or structure.

This second type of predicate can also result in a kind of inefficiency which you might like to avoid. By putting a variable in the MObject slot of the MTrans along with a PTrans structure predicate, we preclude PEARL from hashing the object in any useful way, forcing it to look through all MTranses instead of only MTranses with PTranses in their MObject slot. Since patterns are most often less specific than the objects in the data base, this can make a big difference. Another problem with a variable plus a structure predicate is that the structure predicate is either based on fetches and the first slot or it is limited to matching the type only. We might sometimes want a more complicated structure to be used as a predicate. However, if we opt instead for the more efficient fetching and matching by putting a structure in the slot, we have lost the ability to have a variable bound during the match.

To allow you both to help improve the hashing and matching of a structure and also to bind a variable as a side effect, PEARL provides a mechanism to attach an **adjunct variable** to the slot. This adjunct variable in a slot is bound as a side effect whenever the values in the slot of the two structures were already bound, have already been matched successfully and all predicates and slot hooks have been run. Adjunct variables may be local, lexically scoped or global, just as any other variable. To use an adjunct variable, include the variable *after* the value preceded by a colon and preceding any predicates or slot hooks. For example,

```
(create pattern MTrans
        (MObject (PTrans (Actor John)) : ?Obj))
```

would match any MTrans about John PTransing something, and also bind the adjunct variable ?Obj to the actual PTrans structure that applied.

Since PEARL uses hunks to create so many types of values of its own, it also provides a set of predicates to test an item to see what type it is. Many of them are quite definitely kludges since they depend upon certain bizarre structures existing only in PEARL-created items and not in user-created items and thus should not be depended upon totally. These functions are **streamp**, **databasep**, **blockp**, **definitionp**, **psymbolp** (to distinguish from Franz Lisp *symbolp*), **structurep**, **symbolnamep**, and **structurenamep**.

15. More Useful Slot Types

These last few examples begin to show the restricted nature of basic integer values and of labelling slots as being of type *struct*. If the values in an integer slot will range between -10 and 10, then you would like to say that. If the values which will fill a slot of type structure will be Events or Acts or States, you would like to specify that. PEARL provides mechanisms to fill both of these needs.

In the case of an integer slot to be filled with values from a range of -10 to 10, these integer values do not represent "levels of health" very well either. Rather than saying that a person's "health level" is -2, you might like to say it was "Sick". In fact, you would probably like to say that the values of the slot will be one from among the set of values "Dead, Critical, Sick, OK, Healthy and InThePink". Moreover, you might like to specify that these values are to be associated with integer values in such a way that the ordering you specified holds and you may or may not want to specify precisely what integer values should be associated with these atoms. In other words, you would like a type which consists of a set of values with a linear ordering on them, similar to the Pascal scalar or enumeration type.

Such a type exists in PEARL and is created by a call to the function **ordinal**. For example, to create an ordered set of values to represent levels of various states when you want the actual integer values to be created by PEARL, you would say:

```
(ordinal Levels (Low Middle High))
```

which would associate the numbers 1, 2, and 3 with Low, Middle and High respectively. If you want to specify the values to be associated with each name, you simply list the value after each name. Thus, to create a set of values for use in the integer Level slot of Health above, you might say the following (the values need not be listed in order):

```
        (ordinal HealthLevels (Dead -10 Critical -6 Sick -2 OK 2
                               Healthy 6 InThePink 10))
```

Among the actions that ***ordinal*** performs are the following:

1. The assoc-list of names and values for the ordinal type can be accessed by evaluating the atom built by prepending **o:** to the name of the ordinal type. Given the name of an ordinal type, the function **ordatom** builds this atom. Thus *o:Levels* contains (and *(eval (ordatom 'Levels))* returns) the value *((Low . 1) (Middle . 2) (High . 3))*.
2. Atoms consisting of the name of the ordinal type concatenated with a colon and the value name are created and set to the value they represent. Thus *Levels:Low* is set to 1, *Levels:Middle* is set to 2, etc.
3. Two atoms with **:min** and **:max** concatenated to the name of the ordinal type are created and set to the lowest and highest integer values in the type. Thus *HealthLevels:min* is -10, and *HealthLevels:max* is 10.
4. The name of the ordinal type is added the list of all ordinal type names kept in the special variable ***ordinalnames***.
5. The name of the ordinal type is stored with the slot so that the print functions can convert from the integer value back into the name. Since the default value for integers is zero but most ordinals will not have a zero value, the print functions will print ***zero-ordinal-value*** instead of zero.

Having created an ordinal type, it is then possible to declare in a structure definition that a slot will contain values of that type. The use of values from this type is **not enforced** by PEARL but allows the definitions of integer slots to be more readable, allows the use of the names of values instead of their associated integers when creating individuals and allows PEARL to print the more readable information when printing an integer slot. The special atoms created allow predicates, hooks (demons) and other functions to refer to these values without knowing their associated integers. We can now redefine Health to use HealthLevels:

```
        (create base Health
                (Actor symbol)
                (Level HealthLevels) )
```

and create an individual which says that John is in the pink of health:

```
        (create individual Health
                (Actor John)
                (Level InThePink) )
```

Declaring a slot to be of type *struct* is similarly unenlightening, so PEARL will accept the name of a structure type in its place. For example, we can make the following definitions:

```
        (create base  Person
                (* Identity symbol) )
        (create base  Health
                (Actor Person)
                (Level HealthLevels) )
```

and the Actor slot of Health will be of type *struct*. However, there is currently no extra type checking implied by this declaration (although it is being considered), but again it improves the readability of declarations tremendously.

16. Attaching Hooks to Structures (If-Added Demons)

A fairly old construct within AI is that of demons. In their pure form they could be thought of as asynchronous parallel processes that watch everything going on within a system, lying in wait for a particular set of conditions to occur. These conditions might be a block-manipulating program stacking some blocks too high to be stable, or a data base program violating a consistency constraint. The main problem with classical demons was that in their most flexible form they gobble up far too much system time, as well as being very hard to program as it was hard to see just when they might pop up during the execution of a program.

In an attempt to control the implementation of demons and at the same time provide the user with increased control over the built-in PEARL functions, PEARL allows the user to attach pieces of code to structures that will be run when specific PEARL (or user) functions access particular types of data or pieces of data at particular places in the code. Thus, PEARL provides a general but restricted and fairly efficient ability to control the operation of specific functions on specific pieces of data by providing **hooks** in the PEARL functions which check for requests within structures that certain functions be run when they are accessed in certain ways. Thus PEARL has two useful sub-breeds of **hooks** which watch over either

a. the value of a particular slot of a particular individual structure, referred to as *slot hooks*.

b. operations upon all individuals of a particular base structure type referred to as *base hooks*.

Like predicates, hooks can either be the name of a function to run or a Lisp s-expression to be evaluated. If an s-expression, they can include the special forms ** representing the current structure or * representing the value of the current slot on slot hooks and of the current structure on base hooks. Variables or slot names preceded by = are also allowed (just as in predicates), referring to variables or slots in the current structure. If hooks are run by functions which take two items as arguments, like *match*, then the special form >** may be used to represent the **other** structure (which > is meant to suggest) and >* may be used for the value in this slot of the other structure. (In the case of functions of only one argument, >* and >** are the same as ** and *.) In functions which take two arguments, the special form ? may be used to represent the result that the function intends to return. (This will be **pearlunbound** in hooks which run before the function has done its job.)

When hooks run in the context of a call to *path*, two special variables are available: ***pathtop*** which is the topmost structure passed to path and ***pathlocal*** which is the current innermost structure whose slot is being accessed. When hooks are run in the context of a call to a function which deals with a data base, then the special variable **db** will contain the data base currently being used.

The functions used to fill in the special forms like *, **, =slot, and variables before evaluation come in two flavors and are called **fillin1** and **fillin2**. *Fillin1* is designed for hooks which run on single structures and expects as arguments:

a. the function (s-expression) to fill in,

b. the slot value (or item if a base hook) to use for *,

c. the structure to use for **, and

d. the definition for the item provided as the third argument (for interpretation of *=slot* forms).

Fillin2 is designed for hooks which run on two structures and produce a result and expects as arguments:

a. the function (s-expression) to fill in,

b-c. the slot values (or structures if a base hook) to use for * and >*,

d-e. the structures to use for ** and >**,

f. the definition for the structure provided as the fourth argument, and

g. the result the function intends to return to use for ?.

Four functions for running hooks are provided for the user, two for running slot hooks and base hooks for single items and two for running slot hooks and base hooks for pairs of items. **Runslothooks1** expects to be given the invoking function's name, the structure and name of the slot on which to run the slot hooks, and the value to be used for *. **Runslothooks2** expects to be given the invoking function's name, the two structures and name of the slot in them on which to run the slot hooks, and the values to be used for * and >*. **Runbasehooks1** expects to be given the invoking function's name and the structure whose base hooks are to be run. **Runbasehooks2** expects the invoking function's name, the two structures whose base hooks are to be run and the result the calling function plans to return.

If present, base hooks are run by most major PEARL functions. If a base hook is labelled with *<foo* then the function *foo* will execute the hook just after entry and whatever initialization is necessary. If a base hook is labelled with *>foo* then the function *foo* will execute the hook just before exitting. Slot hooks are run by most major PEARL functions which look through the slots of a structure. If a slot hook is labelled with *<foo* then the function *foo* will execute the hook just before processing the slot. If a slot hook is labelled with *>foo* then the function *foo* will execute the hook just after processing the slot.

However, hooks can be turned off selectively or completely. By setting the atoms ***runallslothooks*** and ***runallbasehooks*** to nil, you can completely disable the running of all hooks. This is useful for debugging and also helps improve efficiency a bit if you do not use hooks at all. There is also an atom to go with each PEARL function (of the form ***run...hooks***) which can be used to disable hooks for selected functions. The following is a complete table of what PEARL functions run hooks and the names of the labels that invoke them and the atoms that control their running:

Base hooks are run by:	invoked by hooks labelled:
create expanded	<expanded or >expanded
create individual	<individual or >individual
create pattern	<pattern or >pattern
smerge	<smerge or >smerge
nextitem	<nextitem or >nextitem
standardfetch *	<fetch or >fetch
expandedfetch *	<fetch or >fetch
insertdb	<insertdb or >insertdb
removedb	<removedb or >removedb
nextequal	<nextequal or >nextequal
indb	<indb or >indb
standardmatch	<match or >match
basicmatch	<match or >match
strequal	<strequal or >strequal

* *fetch* does not run hooks on function structures.

Slot hooks are run by:	invoked by hooks labelled:
standardmatch	<match or >match
basicmatch	<match or >match
strequal	<strequal or >strequal
path put	<put or >put
path clear	<clear or >clear
path addset	<addset or >addset
path delset	<delset or >delset
path addpred	<addpred or >addpred
path delpred	<delpred or >delpred
path get	<get or >get
path getpred	<getpred or >getpred
path gethook	<gethook or >gethook
path apply	<apply or >apply

Hooks of both kinds are controlled by these atoms, initially t:
runallslothooks -- controls all slot hooks.
runallbasehooks -- controls all base hooks.

runputhooks	*runclearhooks*
runaddsethooks	*rundelsethooks*
runaddpredhooks	*rundelpredhooks*
rungethooks	*rungetpredhooks*
rungethookhooks	*runapplyhooks*
runmatchhooks	*runsmergehooks*
runindividualhooks	*runexpandedhooks*
runpatternhooks	*runnextitemhooks*
runfetchhooks	*runinsertdbhooks*
runremovedbhooks	*runindbhooks*
runnextequalhooks	*runstrequalhooks*

It is likely that hooks attached to a particular function would like to run the same function in such a way that hooks will not be invoked. Or in general, it is possible that you will want to run some PEARL function in such a way that it is "hidden" from hooks. To make this easy, a macro is provided called **hidden** which temporarily sets the atom **run...hooks** to nil, runs a command and then restores the former value of that atom. For this to work correctly, you must invoke the function you wish hidden with the name corresponding to its **run...hooks** atom. Thus, you can hide the creation of an individual from hooks by executing:

```
(hidden (individual PTrans ....) )
```

(see Section 27 for the macro *individual*) but **not** by executing:

```
(hidden (create individual PTrans ....) )
```

A parallel function **visible** temporarily sets the associated atom to *t* before evaluating the function.

One of the reasons that hooks are checked for both before and after a PEARL function does its job is to provide the user with the opportunity to affect the result of the particular task. In the simplest case, a hook simply executes a piece of code and does not directly affect the function it is labelled with. However, if the value returned by a hook is a list whose *car* is either ***done***, ***fail***, and ***use***, then the action of that function will be modified. If the result of a hook which runs before the task starts with **done**, then the hook is presumed to have accomplished what the PEARL function was supposed to have done and the function will return immediately with the *cadr* of the hook's result if there is one, or else with the structure being operated on (for base hooks) or the value in the slot (for slot hooks). If the result of a hook which runs after the task starts with **done**, then the function will return immediately with the *cadr* of the hook's result if there is one, or else with the result that was going to be return anyway.

If the result of a hook which runs before the task starts with **fail**, then the hook is presumed to have determined that the PEARL function should quit and the function will return immediately with the *cadr* of the hook's result if there is one, or else with the atom **fail**. If the result of a hook which runs after the task starts with **fail**, then the function will return immediately with the *cadr* of the hook's result (which may be nil).

If the result of a hook which runs before the task starts with **use**, then the hook is presumed to have determined that the PEARL function should use a different value instead of the originally provided one and the function will use the *cadr* of the hook's result for the rest of the task. If the result of a hook which runs after the task starts with **use**, then the function will replace its intended result with the *cadr* of the hook's result (which may be nil). Thus, for example, a slot hook labelled with *<match* can short-circuit the matching of a slot and one labelled with *<match* can reverse the decision made by matching of a slot. Similarly, a base hook labelled with *<match* can use its own matching algorithm and one labelled with *>match* can modify the result of the whole match.

Obviously, these all should be used with great care. Note that *return immediately* means without even running any other slot hooks on that slot for slot hooks or without running any other base hooks on that structure for base hooks.

For example consider the case of a structure representing someone's order in a Chinese restaurant. As items are added to the order, it would be nice if there was a magical slot TotalBill that contained the current running total of the cost of the items ordered. Demons, being such magical creatures, fill the bill nicely. However, we only wish to have our demon-like hooks activated when particular slots are filled (added to or accessed). First consider the simple case in which an order consists of three items only, the name of the soup and one or two entrees:

```
(create base  Chinese-Food-Entree
        (Name  lisp)
        (Price  int) )

(create base  Chinese-Dinner-Order
        (Soup  Chinese-Food-Entree)
        (Entree1  Chinese-Food-Entree)
        (Entree2  Chinese-Food-Entree)
        (TotalBill  int) )

(create individual  Chinese-Food-Entree
        (Name  (Hot And Sour Soup) )
        (Price  323) )

(create individual  Chinese-Food-Entree
        (Name  (Sizzling Rice Soup) )
        (Price  349) )

(create individual  Chinese-Food-Entree
        (Name  (Lingnan Beef) )
        (Price  399) )

(create individual  Chinese-Food-Entree
        (Name  (Mandarin Chicken) )
        (Price  367) )

(create individual  Chinese-Food-Entree
        (Name  (Shrimp Cantonese) )
        (Price  479) )

; an undetermined meal is created.
(create individual  Chinese-Dinner-Order Meal
        (Soup  ^  if  >put  (Maintain-Total  *  **  =TotalBill) )
        (Entree1  ^  if  >put  (Maintain-Total  *  **  =TotalBill) )
        (Entree2  ^  if  >put  (Maintain-Total  *  **  =TotalBill) )
        (TotalBill  0) )
```

Note that a slot hook is put after the value in a slot by using the word **if** (or **hook**) followed by the appropriate label for the invoking function followed by the function name or s-expression to be evaluated. Note also that when you want to put hooks on slots of an individual but do not want to specify a value, the use of "^" will instruct *create* to copy the default value instead. If the Maintain-Total function is properly specified, whenever one replaces one of the food slots with a real dish using the *putpath* function, the Maintain-Total function would be activated and would add the price of that meal to the running total in the TotalBill slot. If one changed one's mind a lot, it would be necessary to include another hook Remove-Price which would be activated by a *clearpath*. This would require adding the *if-cleared* hook "*if >clear Remove-Price*" after the *if-put* hook:

```
        (create individual  Chinese-Dinner-Order  ChangingMeal
                (Soup  ^  if  >put  (Maintain-Total  *  **  =TotalBill)
                          if  >clear  (Remove-Price  *  **  =TotalBill) )
                (Entree1  ^ if  >put  (Maintain-Total  *  **  =TotalBill)
                            if  >clear  (Remove-Price  *  **  =TotalBill) )
                (Entree2  ^ if  >put  (Maintain-Total  *  **  =TotalBill)
                            if  >clear  (Remove-Price  *  **  =TotalBill) )
                (TotalBill  0) )
```

The code for the two hooks follows:

```
        (de Maintain-Total  (Food  Meal  CurrentMealTotal)
            (putpath  Meal  '(TotalBill)
                     (*plus  CurrentTotal
                             (getpath  Food  '(Price) ) ) ) )

        (de Remove-Price  (Food  Meal  CurrentMealTotal)
            (putpath  Meal  '(TotalBill)
                     (*plus  CurrentTotal
                             (getpath  Food  '(Price) ) ) ) )
```

A more flexible meal order structure would not have three slots for food, but rather a single slot of type *setof struct*. Then entries would be added by the *addsetpath* functions, and the *if-put* hook would be an *if-addset* hook but the code would essentially be the same.

To attach a base hook to a structure, the first "slot" in its definition must start with one of the atoms **if** or **hook**. The rest of the slot must then contain a sequence of labels for invoking functions and function names or s-expressions to be evaluated. For example, to invoke *valprint* before and a user function called *verify* afterwards whenever a PTrans is inserted into the data base, you would define PTrans as follows:

```
        (create base  PTrans
                (if <insertdb (valprint * 5)
                    >insertdb (verify *))
                (* Actor  symbol)
                (  Object  symbol)
                (  From  symbol)
                (  To  symbol) )
```

Recall that PEARL provides a print function called **fullprint** which for most structures seen so far printed two extra *nil*s in each slot. If a slot has predicates, the first *nil* will be replaced by a list of them. If the slot has hooks, the second *nil* will be replaced by a list of cons-cells with the invoking function in the *car* and the hook in the *cdr*.

The invocation of hooks labelled with other forms of *path* are similar except for *apply*. If *(path <apply Fcn ...)* or *(path >apply Fcn ...)* is executed, then any hooks which are labelled with Fcn will be run.

At this point the syntax of a slot in a definition or individual has become quite complicated, so we summarize with the following BNF grammar:

{ a b c }	means select one of a, b, or c.
[XXX]	means optionally XXX.
XXX *	means zero or more XXX's
x | y	means x or y

```
<BaseSlot> ::= (
                 <HashLabels>
                 <SlotName>
                 <SlotType>
                 <InheritOrValue>
                 <AdjunctVariable>
                 <PredicatesAndHooks>
                 )
<IndividualSlot> ::= (
                       <SlotName>
                       <InheritOrValue>
                       <AdjunctVariable>
                       <PredicatesAndHooks>
                       )
<ExpandedSlot> ::= <BaseSlot> | <IndividualSlot>

<HashLabels> ::= { "&" "^" "*" "**" ":" "::" ">" "<" } *
<SlotType> ::= { "struct" "symbol" "int" "lisp" } |
               "setof" <SlotType> | <OrdinalName> |
               <StructureName>
<InheritOrValue> ::= <Value> | "^" | "nil" |
                     "==" <Value> | ":=" <Value>
<Value> ::= <integer> | <atom> | <list> | <Variable>
<AdjunctVariable> ::= [ ":" <Variable> ]
<Variable> ::= ?<atom>
<PredicatesAndHooks> ::= { <Predicate> | <Hook> } *
<Predicate> ::= <StructureName> | <S-Expression>
<Hook> ::= "if" <atom> <HookFunction>
<HookFunction> ::= <atom> | <S-Expression>
```

17. Creating and Manipulating Multiple Data Bases

Without any effort on the user's part, a single data base of a defalt size is created by PEARL as it starts up. It is called ***maindb*** and is pointed to by the special variable ***db*** which is assumed by all functions which use a data base to point to the default data base (that is, the data base to be used when an expected data base argument is missing).

To build another data base, choose a name for it and call the function **builddb** which is an nlambda (fexpr) expecting the name of the new data base. You may build as many as you wish and store whichever one you want in **db**. If **db** is already bound after your *.init.prl* file has been read in, then **maindb** will not be built.

Sometimes one may wish to clear out the data base and start out with a clean slate. To make this easy, there is a special function **cleardb** which expects either zero or one data bases as arguments and does the job. If it receives no arguments, then the default data base is cleared. *Cleardb* removes everything from the data base, but does not actually delete (or reclaim the storage space of) the objects within the data base. But if the objects inside are not pointed to by any program variables, they are gone for good.

Data bases contain two parts, referred to as *db1* and *db2*. *Db1* contains items which are indexed under only their type or using single-colon hashing. Its default size is 29. *Db2* contains items which are indexed under two or three values. Its default size is 127. These sizes are chosen to be prime numbers which are just barely smaller than a power of two. (This choice was made to take full advantage of hunks in Franz Lisp are always allocated to be a power of two.) The ratio between the two sizes is approximately 1 to 4. The size for data bases may be chosen by specifying the power of two that you wish *db2* to close to.

The function **setdbsize** expects an integer between 2 and 13 representing the power to which two should be raised. The default data base size is thus the result of calling *setdbsize* with an argument of 7. To change the default size, you **must** call *setdbsize* in your *.init.prl* file. The data base size may be set only once and if it is not changed by the *.init.prl* file, it is set to the default. (In the Franz Lisp version, although this full range of values is accepted, the largest a data base in the 1 to 4 ratio can be is 29 + 127 since hunks are limited to 128 words. However, an argument of 9 to *setdbsize* will set the sizes of both data bases to 127.) Related relevant special variables are ***db1size*** and ***db2size*** which are set by *setdbsize* and ***availablesizes*** which contains the assoc-list used to associate the power of two to a size.

18. Creating a Forest of Data Bases

Although having multiple data bases which are unconnected is often enough, it is sometimes convenient to build onto an already existing data base in a tree-like fashion. For example, in a story understanding program, one might want to have the default data base containing long-term knowledge and then add a data base to contain the knowledge specific to a particular story being processed. In large applications, it can also help to split up special kinds of knowledge to improve efficiency even more than PEARL's hashing already does. With only the ability to build separate data bases, searching for a fact which might be either general knowledge or specific knowledge learned from the story would require two fetches, one from each data base. However, if the story data base is built on top of the main data base then simply fetching an item from the story data base will also include fetching from the main data base. To build another data base upon an existing one, use the function **builddb** with two arguments, the name of the new data base and the name of the old one to build onto:

```
(builddb *story* *maindb*)
(builddb *future* *maindb*)
```

These two statements will build two data bases on top of the main one such that fetching from *story* will look both in it and in *maindb* but not in *future*. You can then build further upon any of these if you wish. Note however, that the second argument must be ***the name of the data base to build upon*** and cannot be ***db*** to build upon the default data base. Also, if the second argument is missing, then the new data base is isolated, not built on top of the default data base.

If your program builds many data bases, it is likely that some of them will be temporary ones. If this is so, it is possible to release a data base so that the space can be garbage collected or reused for a later data base. To release a data base, pass the actual data base (not its name) to the function **releasedb**. If the data base is not a leaf of the data base tree, then the space will not actually be released until all its children are released also but PEARL will no longer accept it as a data base argument.

A list of the names of the currently active data bases is maintained by PEARL in the special variable ***activedbnames***.

19. Creating Expanded Subtypes of Previously Defined Objects

Within CD, as in many applications, you may have many different structures with some slots with the same name. PEARL allows this, as it can always tell which type of structure you are using, and thus it behaves just as if you had used unique names for all slots. But sometimes the fact that two different structure types have slots with the same names is more than a coincidence: there may be various semantic similarities between the similar parts of the two structures. PEARL has a mechanism for creating such structures using the **expanded** selector to *create*. Basically, you must first define a base structure that contains all the identical parts of two or more structures, and then you must define the structures themselves as *the base plus the differences*. A good example of this from CD involves Acts. All Acts within CD have an Actor slot, and all of these slots have the same meaning. That is, whatever is going on, the person in the actor slot is the motivating force. So we may first define this common part as a normal base structure:

```
(create base Act
        (* Actor symbol) )
```

and then we can define the various acts as expansions upon this base:

```
(create expanded Act PTrans
        (Object symbol)
        (From symbol)
        (To symbol) )

(create expanded Act MTrans
        (MObject struct)
        (From symbol)
        (To symbol) )

(create expanded Act ATrans
        (Object symbol)
        (From symbol)
        (To symbol) )
```

```
(create expanded Act Injest
        (Object symbol)
        (Through symbol) )
```

Note that we did **not** have to list the Actor slot, it was **inherited** from the base structure Act. The structure to be expanded need not be a base structure, but could itself be an *expanded* structure. Thus we can capture the similarities of the various Transfers with:

```
(create expanded Act Trans
        (From symbol)
        (To symbol) )
```

followed by

```
(create expanded Trans PTrans
        (Object symbol) )

(create expanded Trans MTrans
        (MObject symbol) )

(create expanded Trans ATrans
        (Object symbol) )
```

In expanded definitions as in base definitions one can specify hashing and default information in the usual way. However one can selectively inherit some of this information from the structure being expanded. Thus in our first Act example, since we specified star hashing on the Actor slot, all the structures that we defined in terms of Act have star hashing on their Actor slot by default. If we had not wanted this for ATrans, we could have specified this simply by listing the Actor slot over again without the asterisk. However, since PEARL requires old slots in expanded structures to also provide a new value, we need some way to say *inherit the same old value*. This is done by putting an up-arrow "^" where PEARL expects to find a value, just as when you want to inherit the default value but add hooks or predicates when creating individuals.

```
(create expanded Act ATrans
        (Actor ^)
        (From symbol) )
```

We also could have added colon hashing to the Actor slot by listing it above as normal. However, we cannot change the type of a slot and including a type name after *Actor* will cause PEARL to try to interpret that type name as a value, (resulting in any of several errors, depending on the type). Thus, the hashing information for any slot is inherited from above, *unless* it the slot appears in the expanded structure.

Default values are inherited in almost the same way. The exception is that if in the original structure the default is preceded by the symbol ":=" (rather than being preceded by either nothing or the symbol "=="), expansions of that structure will not inherit this value, but instead will get the standard default for that type. So if one defines:

```
(symbol Pandora)
```

```
(create base Act
        (Actor symbol Pandora) )
```

or

```
(create base Act
        (Actor symbol ·== Pandora) )

(create expanded Act PTrans
        (From symbol) )
```

then all PTranses will have Pandora as their default Actor, whereas with:

```
(create base Act
        (Actor symbol := Pandora) )

(create expanded Act PTrans
        (From symbol) )
```

only the default instance of Act will have Pandora in its Actor slot and the default Actor of PTrans will just be the usual default for *symbol*-valued slots which is *nilsym*. Which type of default inheritance to use depends upon the application, and must be decided on a case by case basis.

Given this hierarchy, it is often useful to check whether an object is of a certain type or an expanded version of it. Two functions provide this ability with slightly different arguments. **Isa** expects an item and the name of the type you want to check for. **Isanexpanded** expects two instances. Thus the following are always true for any structure X:

```
(isa X (pname X))
(isanexpanded X X)
```

Two related functions are **nullstruct** and **nullsym** which are functions for testing for *nilstruct* and *nilsym* (similar to *null* for *nil*).

20. Fetching Expanded Structures

To make the extra information that *expanded* structures provide more useful, a special version of *fetch* called **expandedfetch** is provided which takes the hierarchy of structures defined into account when fetching. For example, using the above hierarchical definitions of Act, Trans, PTrans, MTrans, and ATrans, you can insert three different Transes into the data base:

```
(dbcreate individual PTrans
          (Actor Pandora)
          (Object Pandora) )

(dbcreate individual MTrans
          (Actor Pandora)
          (To Pandora) )
```

```
        (dbcreate individual  ATrans
                  (Actor  Pandora)
                  (From  Pandora) )
```

and then to fetch all Transes performed by Pandora, you could use:

```
        (create pattern  Trans  TransPattern
                (Actor  Pandorà) )

        (expandedfetch  TransPattern)
```

Once you start using expanded structures, you usually want to be able to use the function name *fetch* and mean *expandedfetch*. To this end, the standard fetch function is actually called **standardfetch**. This leaves the function **fetch** to be bound to whichever fetch function you wish. It is normally given the same function definition as *standard-fetch*.

21. How Two Objects Match

When a fetch from the data base is performed, the pattern provided is only used to construct a stream containing that pattern and the appropriate hash bucket from the data base; no matching (comparing) between the pattern and objects in the data base occurs. Thus the stream contains pointers to all data base items in the same hash bucket, regardless of their likelihood of matching the pattern. When elements are extracted from the stream with the function *nextitem*, the pattern is "matched" against successive items from the hash bucket until one matches (and is returned) or until the potential items run out (and *nil* is returned).

21.1. When Is a Pattern Not a Pattern?

To understand the process with which two objects are matched, it is necessary to understand what is meant by a *pattern* in the context of matching. The term *pattern* has been used in two ways in PEARL. It has been used previously in this documentation in a specialized sense which is only relevant in the context of creating a *pattern*. The use of the *pattern* selector to *create* is simply a variation on *create individual* which uses the match-anything variable ?*any* as the default for unspecified slots instead of the usual default values (either the one inherited from the base definition or the default for the type of slot). It is called creating a *pattern* because the change of default is usually only useful for constructing a pattern.

However, the use of the function *create* with object selector *pattern* is **not** the only way to create a pattern which can be matched; in fact, it is only useful for forming simple patterns. **Any** individual structure in PEARL can be used as a pattern. If a fully specified structure (that is, one with an actual value in all of its slots) is used as a pattern for fetching, it will only match objects which are equal to it in a manner similar to *equal* (versus *eq*) in Lisp. (An exception to this occurs when patterns with pattern-matching variables are stored in the data base.) Thus a fully specified pattern is only useful for determining whether a particular fact (object) is in the data base. Any object is a pattern but the interesting patterns will not be fully specified; rather, they will have unspecified slots which contain pattern-matching variables instead of values. The details of the matching process will now be described.

21.2. The Matching Process

In general, the matching procedure takes two structures and either, neither or both may contain pattern-matching variables. So conceptually, both are patterns. If the structures are not definitionally the same type then the match fails automatically. Otherwise, each structure is viewed as a sequence of slots which are successively "matched" between the two structures. Two structures of the same type match if and only if each of their slots "matches" the corresponding slot of the other structure. Each slot is of one of four types (*struct*, *symbol*, *int*, or *lisp*), or is a *setof* one of these types. Regardless of its type, each slot is filled in one of four ways:

(1) The slot may contain an actual value of its type (for example, a slot of type *struct* may contain a PTrans).

(2) The slot may contain a variable which is local to the structure (pattern-matching variables are local unless otherwise specified).

(3) The slot may contain a global variable, declared previously by a call to the function *global* with the variable's name as argument.

(4) The slot may contain the special match-anything variable ?*any*.

If the slot contains a variable (other than ?*any*) which has not been bound then it may become bound as a side effect of the matching process. All local pattern-matching variables are unbound at the start of the matching process. When a local variable is bound to a real value during the matching process (it will never be bound to a variable), it will not be unbound again but for the purposes of matching will be treated as if the slot were filled with that value.

Let us now examine each of the pairings of slot values which may occur and how they are matched. If either of the two slots being matched contains the special variable ?*any*, then the slots match by definition, regardless of the contents of the other slot. If both slots contain variables that are unbound, the slots do not normally match, (even if the two variables are textually the same name). (Since some users want two unbound variables to match, the value to be returned in this case is stored in the special variable ***matchunboundsresult*** whose initial value is *nil*. Setting this variable to non-*nil* will cause two unbound variables to match immediately but will not cause their predicates to be run.) If one slot contains an unbound variable (and the other a bound variable or a value), then the predicates and restrictions of the slot with the unbound variable are tested, and hooks on that slot labelled with *match* are run to see if the unbound variable should be bound to the bound value. If so, then the unbound variable is bound to the value of the other slot, and the two slots match. Note that only the predicates and hooks on the structure containing the unbound variable are run while the symbols *, **, and =<slotname> refer to the other structure (with the bound value in it). If the predicates or restrictions return *nil*, the two slots do not match, the variable is not bound, and the entire match fails.

If both slots contain either bound variables or values, then the values of the two slots are compared. If the slot is of type *struct*, then the entire matching algorithm is recursively applied. If the slot is of types *int* or *lisp*, then *equal* is used. If the type is *symbol*, than the two values must be the same symbol. Regardless of the type, restrictions associated with the slot are until one fails or there are no more to run. All must succeed for the match to succeed. If the match succeeds, then any hooks with the label *match* are run.

The difference between the two types of variables is one of scope. Normal variables (for PEARL) do not need to be declared, and may be used in any structure by typing in ?<*var*> during a *create* (note that *putpath* is incapable of installing variables). The scope of these variables is only over the structure in which they are typed. Thus the variable ?*V* typed into two different creations of structures are in no way connected (in the same manner as two local variables V in different Pascal subroutines are unrelated.) If one becomes bound, the other is unaffected. On the other hand, if a variable name is previously declared as **global**:

```
(global G)
```

then all instances of the variable name ?G are the same (similar to global variables in Pascal). The list of global variables is kept in the special variable ***globallist***.

As mentioned before, when two structures are matched, all normal (local) variables in both structures are unbound (bound to the value **pearlunbound*) before any slots are compared. This is to ensure that any bindings induced by a previous unsuccessful (or successful for that matter) match are removed. This rule is useful because the type of matching that early PEARL users have needed is in matching most patterns against fully-specified values (that is, cases in which one slot is always bound and the other either bound or unbound). Global variables are **not** unbound before each match, so they can be used to reflect global contexts. They are given the value **pearlunbound** at the time they are declared and remain bound thereafter unless explicitly unbound by the user. To unbind a global variable, you may use use the function **unbind**, a fexpr which requires the name of a (previously declared) global variable:

```
(unbind G)
```

or use *setq* and the function **punbound** which simply returns the atom **pearlunbound**:

```
(setq G (punbound) )
```

The function **pboundp** will test the value of a Lisp (not PEARL) variable to see if it is **pearlunbound**. The function **globalp** will determine whether the variable passed to it has been declared global.

Global variables should be used with care so that they are not set by unsuccessful matches. Generally this is achieved by first collecting the value desired into a local variable via a series of matches (only the last of which succeed), and then using the result of this success to cause a further action which is guaranteed to correctly bind the value of the global variable. (These actions may be hooks which rebind the global variable every time the local one is bound. Effectively, this is a way to say *always unbind this particular global variable before matches*. The action also could be performed by the user's program when the right value is found.)

Each structure or tree of structures built by a call to *create* constructs an individual assoc(association)-list of all the local variables in that structure. This assoc-list is stored with the root of the tree, thus achieving local uniqueness of variables within a structure. Global variables are bound values of the Lisp atom of the same name and are accessed in the usual way. To access the value of a local variable in a structure, one uses either the function **valueof** (which is an expr) or the fexpr **varvalue** both of which have two arguments: the name of the variable whose value you want and the structure it occurs in (evaluated internally by *varvalue*). For example, to get the value of ?G in X, use either of:

```
(valueof 'G X)
(varvalue G X)
```

Thus PEARL uses both deep and shallow binding.

The match algorithm is available to the user as a separate function by the name **standardmatch**. This function unbinds all local variables before proceeding with the match (using the macro **unbindvars**) and again afterwards if the match failed. A function which assumes that all local variables have been unbound already and proceeds just as *standardmatch* would is **basicmatch**. The function name used to access the matching function by *nextitem* and all other built-in PEARL functions is **match** which is normally given the same function definition as *standardmatch* but can be bound to whichever match function you wish. A function which compares two structures for equality without affecting the values of their variables is available as **strequal**. Since it does not bind variables, it also does not execute predicates although it does run base hooks and slot hooks labelled with *strequal*. A function parallel to *nextitem* which uses *strequal* instead of *match* is available as **nextequal**.

This rest of this section covers other ways to access and affect the values of variables. It will make more sense after reading the next section on blocks but fits in better here so you should probably leave it for your second reading.

Recall that the question mark read macro expands into either *(*var* <varname>)* or *(*global* <varname>)*. These two forms are not normally meant to be evaluated. However, for convenience, there are two functions ***var*** and ***global*** which return the value of the variable whose name is their argument. That is, if *?X* expands into *(*global* X)*, executing it will returned the value of the atom X. Thus *X* and *?X* are equivalent for a global variable. For a local or lexically scoped variable, in which *?X* expands into *(*var* X), the function* **var** looks in three places for a variable with the name *X*.

1. First it looks to see if the special variable ***currentstructure*** has been bound to a structure by the user, and if so, looks in its variable list.
2. If this fails, it looks in the special variable ***currentpearlstructure*** for a structure. This variable is set by various PEARL functions like *create*, *fetch*, *path*, and *nextitem* to the top level structure they last operated on.
3. If this fails, it looks in the currently open block on top of **blockstack** if there is one.

4. If this fails, it returns *nil*.

Note that the atom **currentstructure** is there simply for the use of the user and is never set by PEARL.

A related function is **setv** which takes a question-mark variable, a value and an optional environment and sets that variable in that environment or else in the default environment described above to that value. The environment can be either a structure or a block. This stops with an error message if it fails to find a variable by that name in the specified or default environment.

22. Binding Blocks of Structures Together Via Common Variables

It is sometimes the case that you wish to create a group of structures which are closely related in some way and which you wish to tie together via pattern-matching variables. For example, a *frame* might be considered such a loosely connected group of structures. In this case what is desired is for the pattern-matching variables to *actually be the same*. Normally however, if you create several structures in PEARL with variables having the same name, each has its own local variable with that name and they are totally unrelated. If on the other hand, you declared them to be global, then all structures having variables with that name would refer to the same variable and it would no be unbound before matching. For this purpose, PEARL provides variables of an intermediate nature which are local to only a small group of structures and which are all unbound before any one of the structures takes parting in matching.

These variables are called **lexically scoped** (although if the related functions *block* and *endblock* are called dynamically, they also provide a breed of dynamic scoping). To declare a set of lexically-scoped variables, thus opening a (nested) scope for them, use the function **block**, so named because of the similarity to the concept of a block in Algol-like languages. The function *block* is a fexpr which in its simplest form expects one argunent which should be a list of new variables:

```
(block (A B C))
```

Such a call to *block* creates an unnamed block containing these variables and any occurrences of variables with these names in any structures *created* after this call will refer to these lexically-scoped variables. Thus, no structure created after the above call to *block* can contain a local variable called A, B, or C. (However, if a variable has been previously declared to be global this overrides **all** future declarations with *block*. Once again, global pattern-matching variables are to be used with *extreme caution.*)

If you use several blocks, especially nested blocks, it is helpful to give them names. For this purpose, *block* will accept two arguments, the first an atom to name the block and the second the list of new variables. For example:

```
(block Name (A B C))
```

To end the most recent block, use the fexpr **endblock**. This function accepts any of three types of arguments. If last block was unnamed, simply use:

```
        (endblock)
```

If the last block was named, you must provide *endblock* with this name:

```
        (endblock Name)
```

This is provided as a protection against unbalanced calls to *block* and *endblock*. If you wish to end the most recent block, regardless of what its name is, use

```
        (endblock *)
```

To end several blocks at once, you can use the fexpr **endanyblocks** which ends all blocks back through the one whose name matches its argument. Again no argument (*nil*) means the last unnamed block. An argument of "*" causes PEARL to end all currently open blocks. A shorthand for *(endanyblocks *)* is **(endallblocks)**.

The function *block* builds an assoc-list of the variables listed. If the block is nested, the assoc-list of the enclosing block is hooked to the end of its assoc-list, thus providing a complete assoc-list of all the variables available in the block. A side effect of *block* is that this assoc-list is bound to the name of the block. The block itself (the block's name plus this assoc-list) is available as *b:<blockname>* so that the above call to block binds *Name* to

```
((A . *pearlunbound*) (B . *pearlunbound*) (C . *pearlunbound*))
```

and *b:Name* to

```
        (Name (A . *pearlunbound*) (B . *pearlunbound*)
              (C . *pearlunbound*))
```

If a block is unnamed, PEARL calls it *unnamedblock* and the corresponding variables are set. The special variable ***blockstack*** contains a stack of all the currently active blocks. The effect of ending a block is to pop it off this stack. Once a block is closed, it is still accessible through the Lisp variable *b:<blockname>*. Given the name of a block, the function **blockatom** will build this atom for you.

It is possible to return to the scope of an earlier block with the fexpr **setblock** which expects the name of a named block. This will have the effect of ending all currently open blocks and setting the current block stack to contain this block. Note that this block will contain all the variables of any blocks it is nested in but that it is not possible to close off these block selectively. Thus, the block stack will contain only one block with all the variables in its complete assoc-list.

23. Controlling the Unbinding of Variables by Match

It is sometimes desireable to use the filled-in result pattern of a *fetch* or *match* as a pattern for a further *fetch* (or *match*) or to otherwise store and restore the current values of variables (for example, to allow backtracking algorithms and/or hypothetical assertions). Since all bound local variables would normally be unbound during this further fetching or matching, this would not be possible given the mechanism described so far. To accomplish this action, which can be considered as "pushing" the context of the current assoc-list, you should use one of several functions provided for this purpose. The function **freezebindings** takes a structure as argument and moves all bound variables from its normal assoc-list to a backup so that *fetch* will not unbind them. The function **thawbindings** takes a structure as argument and will undo this action, restoring the assoc-list to its complete state. These two functions affect the structure plus any bound variables in all enclosing blocks. To freeze or thaw only a single structure, use **freezestruct** and **thawstruct**. To freeze or thaw only a single block, use **freezeblock** and **thawblock** which expect the name of a block as an argument.

Above it was mentioned that two structures will match if and only if they both are of the same type. Actually the system has been extended to allow the matching of a structure of one type with another of a type derived from the first via a *create expanded*. The extra slots of the larger (expanded) structure are ignored during the match.

Lastly it should be mentioned that the matching rules are an evolving system, and may be amended as experience with their use is accumulated. The rules may seem a bit complex at first, but in use they are fairly natural. The rules are biased towards efficiency (like much of PEARL). The designers felt that hiding exponential time-complexity processing within the language would lead users to construct inefficient programs without realizing it. Thus several "features" of other complex AI matchers are not built in. The user must implement these individually at a higher level. It has been our experience that this leads to much cleaner designs.

24. Function Structures

In using PEARL, it is sometimes handy to escape into Lisp in a "*structured*" way. Although PEARL allows ad hoc escapes by way of its hooks and the ! and $ evaluation operators defined above, the philosophy in PEARL **function structures** is to allow structured escapes that restrict the generality of the escape to the minimum necessary for the task. At times you may wish to equate Lisp functions with their expected arguments with PEARL structures with their associated slots. For example while you may wish to describe an action in a program as fetching an item from the data base, you may actually be unable to describe the item as a structure and/or be unable or unwilling to actually store it in the data base. Instead, you will sometimes want the value to be provided by a function called at fetching time instead of a structure in the data base.

Take as an example the case of keeping track of whether any two objects are near each other. One possible way to do this is to keep structures in the data base which record for each pair of objects that are near each other the fact that they are near each other:

```
(create base Near
        (Object1 struct)
        (Object2 struct))
```

Then determining whether two objects are near each other would require a simple fetch. However, if you are dealing with a large number of objects which are moving around quite a bit but only want to know about nearness once in a while, it might be easier or more efficient to compute whether two objects are near each other only on demand. In this case, you might like to write a function called Near which expects two arguments. However, for consistency, you may not want to design your program so that it knows what things can be fetched and what things need computing. So you would like to define a structure which looks like our definition of Near above but which actually invokes the function Near.

To do this, one may create the function Near (which must be an expr) and also a structure of type *function* named Near:

```
(de Near (x y)
    ... mechanism to actually determine nearness ... )

(create function  Near
        (Object1  struct)
        (Object2  struct))
```

and then can create an individual of it for fetching:

```
(create individual  Near  IsNear
        (Object1  John)
        (Object2  Office))

(fetch  IsNear)
```

Note that the format of function structures within PEARL is the same as that of structures. However, the name of the actual Lisp function to be called must match the type name of the *function* structure, and the arguments must occur in the same order and be of the same types as the slots which will contain the actual arguments to the function.

As another simple example, to define a *function* structure to correspond to the function *getpath*, we would use the following:

```
(create function  getpath
        (Item  struct)
        (Path  lisp) )
```

and then an actual instance:

```
(create individual  getpath  Minst
        (Item  !  Mtrans1)
        (Path  '(MObject) ) )
```

This example is not too useful. As a more realistic use, consider a program to return all the MObjects of all MTranses that are in the data base:

```
(create function  nextitem
        (Stream  lisp) )

(create pattern  MTrans  MPat1
        (MObject  ?X) )

(global  MStream)
(setq  MStream  (fetch  MPat1) )

(create individual  getpath  Minst2
        (Item  (nextitem  (Stream  ?MStream) ) )
        (Path  '(MObject) )
```

```
(setq Stream1 (fetch Minst2))
```

Note the recursive use of the data base: the *fetch* of Minst2 will cause a *getpath* to be executed. But PEARL must first get the two arguments to pass on to *getpath* which causes the function *nextitem* to be evaluated, getting the next MTrans in MStream to pass to *getpath*.

Thus, function structures provide a way to describe a function and its arguments through a PEARL structure and then to include, in a pattern to fetch or in a structure slot, a function call which will provide the desired value at fetching time. However, this only works during fetching.

The function used by PEARL to execute a function structure is **evalfcn**. It takes an item as its argument and returns the result of applying the associated expr to its slot values if the item is a function structure. If the item is a single structure it returns the item untouched. If the item is a list of structures, it applies itself recursively with *mapcar*. No other PEARL functions currently know about function structures as being any different than other individual structures.

25. More About the PEARL Top Level Loop and History Mechanism

The PEARL prompt-read-eval-print loop includes two features which make PEARL easier to work with than the usual top level of Lisp. Both features were designed in imitation of the Berkeley Unix shell program *csh*.

The first is an aliasing mechanism which provides the ability to use various atoms as aliases for commonly executed s-expressions. If you type an atom to the top level and it has the property **alias**, the value of its *alias* property will be evaluated instead. Thus, if you do a

```
(putprop 'dir '(dir) 'alias)  ; in UCI Lisp
         or
(putprop 'ls '(exec ls) 'alias)  ; in Franz Lisp
```

then if you type the atom *dir* or *ls* repectively to the top level, you will get the contents of your directory printed out. Two such built-in atoms are **history** which will run the function *history* and print out your last 64 commands (see below) and **h** which will print the last 22 commands (one crt screenful). The aliasing mechanism can be turned off (saving a *get* for each atom you use at the top level) by setting the special variable ***usealiases*** to *nil*.

PEARL's top level also includes a simplified command-history mechanism. As you type in expressions to the top level of PEARL, they are stored away for future reference. The results of evaluating each expression are also kept. The commands and their results are kept in two hunks whose default size is 64. The hunk containing the commands is kept in the special variable ***history*** and the hunk containing the results is kept in the special variable ***histval*** To change the number of commands remembered, set the special variable ***historysize*** to something other than 64 in your *.init.prl*. It cannot be changed later. (If you are a novice user of PEARL, we recommend that you not change it to be smaller, since the history command can sometimes be helpful to someone helping you to debug something after you have fiddled with it a while.)

The commands you type are squirrelled away so that you can ask PEARL to re-execute them, thus saving the pain of retyping a complicated expression. To access the previous commands, the readmacro "!" is provided. To access the results of the previous commands, the readmacro "$" is provided. (The exclamation point is in imitation of the cshell; the dollar sign is meant to suggest "value".) These readmacros peek at the next character to determine what to do. We discuss the variations available on these two readmacros in parallel, since many of them coincide.

The simplest and most useful forms are "!!" and "$$" which effectively re-execute and reprint the last command or its result. Actually, both forms are executed, but the dollard sign macro always returns its value quoted so that its effect is usually to just reprint the result of the previous command. Note that since these are readmacros which simply return the last s-expression typed or its value, you can use them to build up more complex commands. For example:

```
pearl> (fetch Item)
    (*stream:* . . .)
pearl> (nextitem !!)
```

will cause the fetch to be repeated and then do a *nextitem* on it. However, it is much more efficient to use the *$$* form in this case, since what you really want is to do a *nextitem* on the result of the *fetch* in the last command:

```
pearl> (fetch Item)
    (*stream:* . . .)
pearl> (nextitem $$)
```

The commands are numbered as you type them, starting with zero. Although the values wrap around in the hunks, the *history number* continues to climb. The current history number is available in the special variable ***historynumber***. To access a particular command or its value, you may type you may follow an exclamation point or dollar sign with the number of the command. Thus **!23** and **$23** are the 23rd command and its result. If you don't remember the command's number you can use the function name or a prefix of it. Thus **!fetch** and **$fetch** will access the last *fetch* or its value. Or **!fe** and **$fe** will access the last command starting with *fe* or its value. If there was a reference to an atom (instead of a list) with that name or with that as a prefix somewhere, then the atom will be evaluated again. For exclamation point, this is a waste of typing except for long atom names. For dollar sign, it provides you a way of recovering the value of a variable that has since changed. (As a side effect of implementing this, PEARL contains a function **prefix** which expects two lists and determines whether the first is a prefix of the second, considered as a list of atoms. Thus, PEARL just calls *prefix* on the results of *explodeing* two atoms.)

Here the parallel between the two macros ends.

There are five forms which work only with exclamation point and refer only to the last s-expression typed. They are essentially ways to pick individual top-level elements out of the last command:

!^	the first argument
!$	the last argument
!*	the complete set of arguments
!:0	the function name
!:n	the nth argument

Both macros are splicing macros so that their values may be spliced into the current s-expression. !* is designed so that the following will work:

```
pearl> (add 1 2 3 4)
    10
pearl> (times !*)
(times 1 2 3 4)
    24
```

To see the last 64 commands you gave printed out, use the function **history** (or type the atom **history**). If you don't want all 64 commands, *history* will accept an integer argument telling how many you want. Thus the aliases on *history* and *h* are:

```
(putprop 'history '(history) 'alias)
(putprop 'h '(history 22) 'alias)
```

If you use the command numbers often, you might like to have the history number printed out before each command. To have the history number printed just before the PEARL prompt, set the special variable ***printhistorynumber*** to a non-*nil* value. The default value is flnilfR.

Whenever you use the ! or $ history mechanisms, the line you type in will be reprinted in its expanded form on the next line using the current *pearlprintfn*. If you wish to modify your own read macros so that they also will cause this reprinting, simply have them set the special variable ***readlinechanged*** to a non-*nil* value.

It is sometimes useful to have a function return no value. That is, you often do not want the value of the function to be printed by the top level loop. In particular, functions which print values often return ugly values afterward. To get around this problem, the PEARL top level disables printing of the value returned by a function if it returns the atom ***invisible***. All of the PEARL print functions return this value.

It is sometimes useful to be able to save the current state of a PEARL run for later. There are two functions to allow this. If you wish to save a version which will continue exactly where you left off (at the top level), use the function **savecontinue** which expects zero, one or two arguments. If you wish to save a version which will read in the *.start.prl* file when it starts up, use **savefresh**. (If you also want *.init.prl* read in, change the value of the special variable ***firststartup*** to *t* beforehand but be careful not to put functions which may only be run once in it.) Note however that you cannot save Franz PEARL on top of the file you are running; trying to will result in the *Dumplisp failed* error message from Franz Lisp. Note also that a saved PEARL uses about 1500 blocks or 750kbytes on the disk so this should be used sparingly. (Exceeding the disk quota will result in the same error message.) In the Franz Lisp version, if the number of arguments to either of these functions is:

0: It will be saved as *pearl* in the current directory.

1: The argument is assumed to be a (relative) file name to save under.

2: The result of concatenating the two arguments together with a / between them will be the file name used. (This is for UCI Lisp compatibility.)

In the UCI Lisp version, if the number of arguments is:

0: It will be saved as *pearl* in the current directory.

1: The argument is assumed to be a file name for the current directory.

2: They must be a directory and a file name to save in.

26. Looping and Copying Functions

PEARL includes several loop macros. The first two were included simply for use by the implementation but might be useful to the user. They are the **for** and **while** macros which both expand into a *prog* wrapped around a *progn*. A call to the *while* macro should be of the form:

```
(while <test>
       EXPR1
       EXPR2
       ...
       EXPRn)
```

The <test> is evaluated before each execution of the loop. If it is non-*nil*, the EXPRi are evaluated in sequence. This continues until <test> return nil in which case the last value returned by EXPRn is returned. Since the while expands into a *prog*, any of the EXPRi may call the function *return*, terminating the loop prematurely and returning the value given to *return*.

A call to the *for* macro should be of the form:

```
(for <var> <initial> <final>
     EXPR1
     EXPR2
     ...
     EXPRn)
```

<initial> and <final> should evaluate to integers. The EXPRi are repeatedly evaluated in sequence with <var> being set to the values ascending from <initial> to <final>. If <initial> is greater than <final>, nothing is done. <var> is a prog variable which disappears after the *for* executes. The value returned is the last value of EXPRn and *return* provides a premature exit with a value as in *while*.

The fexpr **foreach** expects a stream and a function (or macro) and applies the function to each element returned by successive calls to *nextitem* on the stream. Unfortunately it only returns *nil* at this time. Eventually, other useful looping structures may be provided.

Since PEARL provides several new types of values, it provides a few functions to copy them. In particular, the standard Lisp function **copy** has been redefined to avoid trying to copy anything that is not a cons-cell. There are several ways to copy structures, described below. The rest of PEARL values either are too complicated to copy (data bases), can be copied with *copy* (streams) or else make no sense to copy (symbols, blocks).

For copying structures, there are currently two functions. The one you are most likely to want is **scopy** which expects a single structure argument and returns a new structure with the same values in it. However, the new structure will differ from the old in several important ways. First of all, copying a bound variable will result in the actual value being inserted in the new copy. When copying an unbound variable, the new structure will receive a local variable with the same name and this variable will be installed in the slot. All variables so installed will be installed in the top level structure regardless of where they came from in the original. The only exception to this is lexically-scoped variables. When the new structure is built, it will be built within any currently open blocks and any of its unbound variables whose names match variables from the current block(s) will be identified with those block variables. Global variables are similarly reinstalled only if they are unbound. Adjunct variables are also installed *only if* they are unbound, since if they are bound their purpose will already have been served and their bound values installed in other slots referring to them.

A variation on *scopy* which replaces all unbound variables from the original with ?**any** is called **patternize**. After (and during) the running of these copying functions, the resulting top-level structure is kept in the special variable ***currenttopcopy***.

The situation sometimes arises where you have already built a structure and have a new structure with information that should be merged into the old one. Rather than use *path* to copy each relevant slot, you can use **smerge** which expects as arguments the old structure to merge into and the new structure from which to take values. All unfrozen variables in the old structure are unbound first and then any unbound variable whose counterpart in the new structure is bound gets replaced (**not set**) with this value. The old structure being merged into must be of the same type or an expanded version of the new structure.

27. Miscellaneous Variations and Abbreviations

People very quickly get tired of typing the relatively long function names that PEARL uses. As a result, a large number of abbreviations and macros have been included in PEARL. We recommend that the shortest ones be used primarily at the top level, since they are easily subject to typographic errors. Most the abbreviations are in *create* and are summarized by the following table:

The function or atom:	May be abbreviated:
create	cr
individual	ind
pattern	pat
expanded	exp
function	fn

Thus, *(cr pat ....)* is equivalent to *(create pattern ....)*.

In addition, a large number of macros for popular combinations of functions are included:

The s-expression:	Is expanded into by the macro:
(create base ...)	(cb ...)
	(base ...)
(create individual ...)	(ci ...)
	(individual ...)
	(ind ...)
(create expanded ...)	(ce ...)
	(expanded ...)
	(pexp ...)
(create pattern ...)	(cp ...)
	(pattern ...)
	(pat ...)
(create function ...)	(cf ...)
	(pfunction ...)
	(fn ...)
(insertdb (create ...) nil)	(dbcreate ...)
	(dbcr ...)
'(quote ,(create ...))	(inlinecreate ...)
(fetch (create ...) nil)	(fetchcreate ...)
'(fetch (quote ,(create ...)) nil)	(inlinefetchcreate ...)
(nextitem (fetch ...))	(firstfetch ...)
(valprint ...)	(vp ...)
(fullprint ...)	(fp ...)

(*pexp* and *pfunction* are so named to avoid conflict with the exponential function *exp* and the function quoting function *function*.)

The automatic setq feature of *create* that causes an atom to be bound to the item created is available throughout *create*. In all cases, the special variable ***lastcreated*** is set to the item. In addition:

This combination:	Causes this atom to be set:
(create base X ...	X
(create base X Y ...	Y
(create expanded X Y ...	Y
(create expanded X Y Z ...	Z
(create individual X ...	(none)
(create individual X Y ...	Y
(create individual X X ...	(none, the second X is ignored)
(create pattern X ...	(none)
(create pattern X Y ...	Y
(create pattern X X ...	(none, the second X is ignored)

When creating an object, wherever a recursive call to *create* is implied by a structure in a slot of type structure, you may start with one of the types *individual*, *pattern*, *base*, *expanded*, *function* to change the type of object being created. Whenever it isn't given, the type of the toplevel *create*, which is kept in the special variable ***currentcreatetype*** is used. For example, in

```
(create pattern x
        (a (individual y))
        (b (base z (s1 ...) ...))
        (c (w)))
```

where a, b, and c are all slots of type structure, slot a will contain an individual y which the attendant defaults filled in, slot b will contain the default instance of a newly created type z, and slot c will contain a pattern w with ?*any* as defaults.

Since each Lisp stores its functions in a different place, PEARL includes a macro **aliasdef** which expects the names of an new and a old function name and copies the function definition of the old one to the new one. In the case of Lisps which store the function definition on the property list, *aliasdef* requires a third argument which is the name of the property that the definition is kept under.

28. Low Level Access Functions.

There are a large number of functions for setting and accessing the various part of structures, symbols, and data bases which are primarily intended for the use of PEARL. In general, the access functions are called **get**... where "..." is the name of the information about the structure. The functions which change information are called **put**.... It is not generally safe to use the *put*... functions but the *get*... functions can sometimes be useful to the user. For a complete list of the functions, see the index. If you don't recognize the function by name, you don't need it so we don't bother to further document them. Since most of them expect a slot number, it is useful to know about the macro **numberofslot** which requires the name of a slot and the definition of a structure (which can be accessed with *defatom* or *d:<structurename>*.) and returns the corresponding slot number.

29. Appendix of UCI Lisp functions added to Franz PEARL

Since PEARL was originally written in UCI Lisp, there are many functions from UCI Lisp that it needed. We also wrote others to move our other programs. The number is too great to document each one. If the function is described with an equal sign, as in "*fn* = *other*" then the function definition of the Franz Lisp function *other* has been put under *fn*. Thus it might not behave quite the same as in UCI Lisp. If no equivalence is given, it was written from scratch which is slightly more likely to mimic UCI Lisp. In this case, see the UCI Lisp manual for details.

The functions used for the PEARL top level loop in the Franz Lisp version plus changes to the fixit debugger and the trace package are briefly described here also.

The Franz Lisp version of PEARL is normally loaded with both the Fixit debugger and the trace package already loaded. This is done to avoid getting the versions which do not know how to print PEARL objects. In addition, the Fixit debugger is attached to all available hooks for going into the break package, since it is much more similar to the UCI Lisp break package than the standard Franz Lisp break package is. Both the debugger and trace package use the function **breakprintfn** to print values. The *msg* function uses the function **msgprintfn** to print values. Either can be bound to whatever function you wish. To disengage the Fixit debugger, read the Franz manual chapter on exception handling. See Note 4 below for more on features added to the Fixit debugger.

Atoms and Variables:
dskin -- special variable -- initial value: t. See Note 1 below.
file -- special variable -- initial value: nil. Used by *dskin*
    and function definition functions.
invisible -- special atom -- not printed by *dskin* if returned
        by a value when it is evaluated.

Functions:
*append = append
(breakprintfn value lmar rmar) -- used by *trace* and *debug*.
*dif = diff
*eval = eval
*great = greaterp
*less = lessp
*max = max
(msgprintfn value lmar rmar) -- used by *msg*.
*nconc = nconc
*plus = plus
*times = times
(addprop 'id 'value 'prop)
(allsym itemorpair) -- fexpr
(apply* 'fcn 'args) -- macro -- This is provided to act like UCI Lisp's
        apply#. The asterisk is used because of the special meaning
        of # in Franz Lisp. Unlike Franz Lisp's *funcall* and
        apply, this does what you would expect with macros!
atcat = concat
(boundp 'item)
clrbfi = drain

consp = dtpr
(de fcnname arglist &rest body) – macro – See Note 2 below.
(debug-replace-function-name 'cmd 'frame) -- Used by the modified
        Fixit debugger to handle the "> newfcnname" facility.
(defp 'to 'from [prop]) – macro – Ignores *prop* and just
      copies the function definition.
(defv var val) – fexpr
(df fcnname arglist &rest body) – macro -- See Note 2 below.
(dm fcnname arglist &rest body) -- macro – See Note 2 below.
(dremove 'elmt 'l)
(drm char lambda) – macro – See Note 2 below.
(dskin filename1 filename2 ....) – See Note 1 below.
(dskin1 '*file*)
(dskin2 'port)
(dsm char lambda) – macro – See Note 2 below.
(enter 'v 'l)
(every 'fcn 'args) – macro -- Potential problem when compiled.
expandmacro = macroexpand
(funl &rest body) – macro -- Expands into (function (lambda ...)).
(ge 'x) – macro
(gensym1 'ident 'val)
gt = >
(initsym atomorpair1 ...) – fexpr
(intersection 'set1 'set2)
(islambda 'fcn) – Is *fcn* a lambda (expr)?
(le 'x) – macro
(length '*u*)
lineread = readl (below)
(litatom 'x) – macro
lt = <
mapcl = mapcar
memb = member
(msg ...) -- macro – Some features may be missing. The function
        used to print is *msgprintfn*, initially bound to
              (or (eq '*invisible* ...)
                  (patom (valform ...)))
(nconc1 'l 'elmt)
(nequal 'arg1 'arg2)
(newsym atom) – fexpr
noduples = union (below)
(nth 'l 'num)
(oldsym atomorpair) – fexpr
(pearl-break-err-handler) – Should be tied to ER%tpl if you want the
        standard Franz Lisp break (not much of a) package.
        Same as standard Franz Lisp *break-err-handler* except
        that it uses the function *breakprintfn*.
(pearl-top-level) – The PEARL top level loop.
(pearl-top-level-init) – The initial function called when PEARL starts up.
        This is the code that reads in the init files and sets any unset
        PEARL parameters.
peekc = tyipeek
(pop q) – macro
(push var 'val) – macro
(readl ['flag]) – fexpr

(readl1 'flag)
remove = delete
(remprops 'item 'proplist)
(remsym atomorpairlist) -- fexpr
(save fcnname) -- fexpr -- Saves function or macro definition under
the property *olddef*. Saves macro character definitions
under *oldmacro*.
(selectq ...) -- macro
(some 'fcn 'list) -- macro -- Potential problem when compiled.
(sprint 'item ['lmar ['rmar]]) -- See Note 3 below.
(subset 'fcn 'list) -- macro
(timer (defun timer fexpr (request)$?
(unbound) -- macro
(union 'list1 ['list2 ...])
(unsave fcnname) -- fexpr -- See *save*.

Note 1: A simplified but extended imitation of the UCI Lisp function **dskin** is provided in PEARL. It is an nlambda which requires the file extensions to be provided. There is a special variable ***dskin*** which controls whether the expression read in is printed and/or whether the result of evaluating it is printed.

dskin = nil	means neither
dskin = t	means result only
dskin = 'name	means the name of the variable in setq *or* the name of the function in de, df, dm, dsm, drm, defmacro, defun, or def *or* the name of the type in create.
dskin = 'both	means both t and 'name.

The default value of *dskin* is t.

File names are always printed before they are opened. The print function used for values is the current function definition of **dskprintfn**. The default function definition in PEARL is:

```
(de dskprintfn (*printval*)
    (cond ((atom *printval*) (patom *printval*))
          ( t (print (valform *printval*)))))
```

Note 2: For better compatibility with UCI Lisp, PEARL contains macros for the function and read macro definition functions **de**, **df**, **dm**, **dsm**, and **drm**. They have been defined to save the old definitions automatically and to return *(fcnname Redefined)* when this is the case. *De*, *df*, and *dm* save the old definition under the property '*olddef*. *Dsm* and *drm* save the old definition under the property '*oldmacro*. (The current definition of a readmacro is kept by Franz under the property '*macro*.) If the function definition is read in by *dskin*, then the current file name which is in the special variable ***file*** is put under the property '*sourcefile*.

Note 3: A function similar to the UCI Lisp **sprint** is included, including the printmacro facility and the optional second argument saying which column to start in. In addition, there is an optional third argument saying which column to try not to go beyond (that is a right margin). A slight addition has been made to the printmacro feature (feature 1 below). During *sprinting*, if the atom in the function position in a list has the printmacro property one of four things will happen during *sprinting*:

1. If the printmacro property value is a string and the item to be printed has a nil *cdr*, then the string will be printed instead of the item.
2. If the printmacro property value is a string and the item to be printed has two items in it, then the string will be printed followed immediately by the *cadr* of the item.
3. If the printmacro property value is a string but the item to be printed is longer than two elements, then it will be *sprinted* in the normal fashion (i.e., the printmacro will be ignored).
4. Otherwise, the printmacro property value will be applied to the rest of the arguments. It should be a function which expects three arguments, the item to be printed, a left column to start in and a right column to try not to go beyond. A good default value for the right column argument seems to be zero. If the function under the printmacro property returns nil, then *sprint* assumes that it decided not to print the item and prints it in the usual way.

Note 4: The Fixit debugger now accepts a command of the form > **newname** whenever either an undefined function or unbound variable error occurs. As in UCI Lisp, newname is not evaluated in the case of an undefined function but is evaluated in the case of an unbound variable. Note that the blank is required (unlike UCI Lisp). This is not guaranteed to work if you move around the stack first.

30. Appendix of Franz Lisp functions added to UCI Lisp PEARL

The following is a summary of the functions added to the UCI Lisp version of PEARL to make it compatible with Franz Lisp. Where the details are not obvious, see the Franz Lisp manual. **Note**: Most *macros* listed in the index which are labelled with asterisks are not available in UCI Lisp PEARL, since the implementor must specifically request that they stick around.

Dskin, the break package, and *msg* have been changed to use the functions **dskprintfn**, **breakprintfn**, **msgprintfn**for printing.

(addtoaddress 'n 'address) -- expr -- Used by *cxr* and
rplacx. Written in LAP code.
(apply* 'fcn 'args) -- macro -- Equivalent to *apply#*.
(buildalist ...) — expr — Used by *defmacro*.
(combineskels ...) -- expr -- Used by *quasiquote*.
(convert ...) — expr — Used by *defmacro*.
(cxr 'index 'hunk) -- expr -- A hunk is a block of memory. Provides
random access to a single cell of a hunk. (Uses
addtoaddress and *even*.)
(defmacro macroname arglist body) -- macro -- *Defmacro* provides
a slightly more intelligent macro facility. *Body* is
processed to look for occurrences of the arguments in
arglist which are replaced with the appropriate form
of *ca..r*. If an argument is preceded by *&rest*,
then it gets the list of the rest of the arguments.
The Franz Lisp version has many more features not included
in the PEARL version.
(even 'x) -- expr -- Is *x* even? Used by *cxr* and
rplacx to determine which half of a cons-cell to use.
(isconst ...) -- expr -- Used by *quasiquote*.
(makhunk 'size) -- expr -- Calls the UCI Lisp function *getblk*,
requesting a block of memory which is half of *size*, since
each piece of a UCI Lisp block of core is a cons-cell.
(msg ...) -- fexpr -- Modified to use *msgprintfn* to print
values of evaluated elements of the print list.
(pearl-top-level) -- the PEARL top level loop.
(pearl-top-level-init) -- The initial function called when PEARL starts up.
(rplacx 'index 'hunk 'val) -- expr -- Provides random access storage into
a block of memory. (Uses *addtoaddress* and *even*.)
(quasiquote 'skel) -- expr -- called by the quasi-quote readmacro
character backquote '. Equivalent to the quasiquote
functions defined in Charniak[2] with different invoking
characters to match those of Franz Lisp.
Unquote is comma "," and splice-unquote is ",@".
Uses *combineskels* and *isconst*.

31. Bibliography

[1] Bobrow, D., and Winograd, T. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science* 1:1 (1977).

[2] Charniak, E., Riesbeck, C., and McDermott, D. *Artificial Intelligence Programming*. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1980.

[3] Faletti, J., and Wilensky, R. "The Implementation of PEARL: A Package for Efficient Access to Representations In Lisp", forthcoming ERL technical report, UCB.

[4] Greiner, R., and Lenat, D. "A Representation Language Language." In *Proc. First NCAI*. Stanford, CA, August, 1980, 165-169.

[5] Roberts, I., and Goldstein, R. "NUDGE, A Knowledge-Based Scheduling Program." In *Proc. IJCAI-77*. Cambridge, MA, August, 1977, 257-263.

[6] Schank, R. *Conceptual Information Processing*. Amsterdam: North Holland, 1975.

[7] Wilensky, R. "Understanding Goal-Based Stories", Technical Report 140, Computer Science Department, Yale University, New Haven, CT, September 1978.

[8] Wilensky, R. "Meta-Planning: Representing and Using Knowledge about Planning in Problem Solving and Natural Language Understanding." *Cognitive Science* 5:3 (1981).

32. Index of Global Variables and Functions With Their Arguments

All functions are exprs (or lexprs) unless otherwise listed. Functions with one or more asterisks for a page number are not documented other than in this index because they were not actually intended for use by the PEARL user. A single asterisk * means it is primarily intended for use by PEARL but might be useful and will generally work right. A double asterisk ** means it will generally only work within PEARL's code, since it expects certain external prog variables to exist and be set correctly. A triple asterisk *** means it is dangerous to use. Note that it is dangerous to redefine any functions in this list, although it should be all right to redefine any macros.

activedbnames -- special variable -- initial value: nil 40
*any*conscell* -- special variable -- value: '(*any* . *pearlunbound*) *
availablesizes -- special variable -- value: 39
((-1. . 1.) (0. . 1.) (1. . 1.) (2. . 3.) (3. . 7.)
(4. . 13.) (5. . 29.) (6. . 61.) (7. . 127.) . . . .
Franz Lisp: . . . (8. . 127.) (9. . 127.) (10. . 127.)
(11. . 127.) (12. . 127.) (13. . 127.))
UCI Lisp: . . . (8. . 251.) (9. . 509.) (10. . 1021.)
(11. . 2039.) (12. . 4093.) (13. . 8191.))
blockstack -- special variable -- initial value: nil 48
currentcreatetype -- special variable -- initial value: base 56
currentpearlstructure -- special variable -- initial value: nil 46
currentstructure -- special variable -- initial value: nil 46
currenttopcopy -- special variable -- initial value: <UNBOUND> 55
currenttopcreated -- special variable -- initial value: (nilstruct) 8

db -- special variable -- default initial value: <UNBOUND> 33
db -- special variable -- default value: the *maindb* database 12
db1size -- special variable -- default initial value: 29 39
db2size -- special variable -- default initial value: 127 39
done -- special atom 35

fail -- special atom 35
file -- special variable -- initial value: nil 60
firstartup -- special variable -- initial value: t 53
function-stream: -- special atom 13
globallist -- special variable -- initial value: nil 45

history -- special variable -- value: command history hunk 51
historynumber -- special variable -- initial value: 0 52
historysize -- special variable -- default value: 64 51
histval -- special variable -- value: value history hunk 51
invisible -- special atom 53

lastcreated -- special variable -- initial value: (nilstruct) 8
lastsymbolnum -- special variable -- initial value: -1 *
maindb -- special variable -- default value: the main data base 11
matchunboundsresult -- special variable -- initial value: nil 44
ordinalnames -- special variable -- initial value: nil 31

pathlocal -- special variable -- initial value: <UNBOUND> 33
pathtop -- special variable -- initial value: <UNBOUND> 33
pearlprompt -- special variable -- default value: "pearl> " 3, 4
pearlunbound -- special atom 45
printhistorynumber -- special variable -- initial value: nil 53

readlinechanged -- special variable -- initial value: nil 53
runaddpredhooks -- special variable -- initial value: t 34
runaddsethooks -- special variable -- initial value: t 34
runallbasehooks -- special variable -- initial value: t 33
runallslothooks -- special variable -- initial value: t 33

runapplyhooks -- special variable -- initial value: t 34
runclearhooks -- special variable -- initial value: t 34

rundelpredhooks — special variable — initial value: t 34
rundelsethooks — special variable — initial value: t 34
runexpandedhooks — special variable — initial value: t 34
runfetchhooks — special variable — initial value: t 34
rungethookhooks — special variable — initial value: t 34
rungethooks — special variable — initial value: t 34
rungetpredhooks — special variable — initial value: t 34
runindbhooks — special variable — initial value: t 34
runindividualhooks — special variable — initial value: t 34
runinsertdbhooks — special variable — initial value: t 34
runmatchhooks — special variable — initial value: t 34
runnextequalhooks — special variable — initial value: t 34
runnextitemhooks — special variable — initial value: t 34
runpatternhooks — special variable — initial value: t 34
runputhooks — special variable — initial value: t 34
runremovedbhooks — special variable — initial value: t 34
runsmergehooks — special variable — initial value: t 34
runstrequalhooks — special variable — initial value: t 34

stream — special atom 13
stream: — special atom 13
toplevelp — special variable — initial value: <UNBOUND> *
unhashablevalues — special variable — initial value:
(0 unbound *pearlunbound* nilsym (nilstruct)) *
use — special atom 35
usealiases — special variable — initial value: t 51
warn — special variable — initial value: t 17
zero-ordinal-value — special variable — initial value: 0 31

! — splicing macro 52
$ — splicing macro 52
= — read macro 28
? — read macro 16

(addalist 'var 'inst) — macro *
(addbasehook 'conscell 'item) — macro *
(addhistory 'line) *
(addpredpath 'item 'path 'pred) 10
(addsetpath 'item 'path 'value) 10

(addtoexpansionlists) — macro **
(adjvarset 'var 'val) — macro *
(allocdef numofslots) — macro *
(allocval numofslots) — macro *
(applypath 'fcn 'item 'path) 10

(base name [storage] slot1 ...) — macro 56
(basicmatch 'item1 'item2) 46
(block [blockname] varlist) — fexpr 47
(blockatom 'symbol) 48
(blockp 'potblock) 30

(breakprintfn '*printval*) 58, 59
(builddb newdb [olddb]) — fexpr 38
(buildintvalue 'intval 'bppset) — macro *
(buildslot) — macro **
(buildstructvalue 'structdesc) — macro *
(buildsymbolvalue 'symname) — macro *
(buildvalue 'value 'typenum 'ppset) *

(cb name [storage] slot1 ...) — macro 56
(ce basename newname [storage] slot1 ...) — macro 56
(cf name [storage] slot1 ...) — macro 56
(checkandrunbasehooks2 'fcn 'item1 'item2) — macro **
(checkandrunslothooks2 'fcn 'hooks 'val1 'val2 'item1 'item2) — macro **

(checkrunhandlebasehooks1 'fcn 'runhooksatom) – macro ••
(checkrunhandleslothooks1 'fcn 'runhooksatom) – macro ••
(ci basename [storage] slot1 ...) – macro 56

(cleardb ['db]) 39
(cleardb1 'db) 39
(cleardbactive 'db) – macro •
(clearhashandformat 'slotnum 'defblock) – macro •
(clearpath 'item 'path) 10

(compatible 'slotnum 'item1 'item2) – macro •
(connectdb 'newdb 'olddb) •
(consistentvalue 'val 'predlist 'typenum 'item) – macro •
(constructvalue) – macro ••
(convertpreds 'pred) •
(copy 'list) 54
(copypatternslot) – macro ••
(copyslice) – macro ••
(copyslot 'nameblock) – macro ••

(cp basename [storage] slot1 ...) – macro 56
(cr selector ...) – fexpr 55
(create selector ...) – fexpr 5
(createbase 'newname 'slots) •
(createexpanded 'oldname 'newname 'slots) •
(createfunction 'fcnname 'slots) •
(createindividual 'basename 'slots) •
(createpattern 'basename 'slots) •

(databasep 'potdb) 30
(dbcr selector ...) – macro 56
(dbcreate selector ...) – macro 12, 56
(debugprint 'item) 21
(defatom 'symbol) 7

(defaultfortype 'typenum) – macro •
(definitionp 'potdef) 30
(delpredpath 'item 'path 'pred) 10
(delsetpath 'item 'path 'value) 10
(disguisedas 'filler 'struct ['db]) 29
(disguisedas1 'filler 'struct 'db) 29

(dobasehooks2< 'fcn 'runhookatom) – macro ••
(dobasehooks2> 'fcn 'runhookatom) – macro ••
(doslothooks2< 'fcn 'runhookatom) – macro ••
(doslothooks2> 'fcn 'runhookatom) – macro ••
(dskprintfn '*printval*) 60

(endallblocks) 48
(endanyblocks blockname) – fexpr 48
(endblock [blockname]) – fexpr 47
(enforcetype 'value 'typenum) •
(equalvalue 'xval 'yval 'typenum) – macro •
(evalfcn 'item) 51

(executehook1 fcn value item defblock) – macro ••
(executehook2 fcn val1 val2 item1 item2 defblock result) – macro ••
(expanded basename newname [storage] slot1 ...) – macro 56
(expandedfetch 'item ['db]) 42
(expandedfetch1 'item 'db) 42

(fcnslot) – macro ••
(fetch 'item ['db]) 12, 43
(fetch1 'item 'db) 12, 43
(fetchcreate selector ...) – macro 14, 56

(fillbaseslot) -- macro **
(fillin1 'fcn 'value 'item 'defblock) 33
(fillin2 'fcn 'val1 'val2 'item1 'item2 'defblock 'result) 33
(fillindivslot) -- macro **

(findnextblockstart) -- macro **
(findslotnum) -- macro **
(findstructsymbolpair 'defblock 'symbol) -- macro **
(firstfetch pattern) -- macro 14, 56
(fn name [storage] slot1 ...) -- macro 56

(followpath 'item 'path) *
(for val 'init 'final &rest 'body) -- macro 54
(foreach 'stream fcn) -- fexpr 54
(fp 'item ['lmar ['rmar]]) 56
(freezebindings 'struct) 48
(freezeblock 'blockname) 49
(freezestruct 'struct) 49

(fullform 'item) 20
(fullprint 'item ['lmar ['rmar]]) 20, 37
(fullprint1 'item 'lmar 'rmar) 20, 37
(fullslotform) -- macro **

(getalist 'inst) -- macro 57
(getalistcp 'inst) -- macro 57
(getbasehooks 'defblock) -- macro 57
(getdb1 'db) -- macro 57
(getdb2 'db) -- macro 57
(getdbactive 'db) -- macro 57
(getdbchildren 'db) -- macro 57
(getdbname 'db) -- macro 57
(getdbparent 'db) -- macro 57
(getdefaultinst 'defblock) 57
(getdefinition 'valblock) 57
(getenforce 'slotnum 'defblock) -- macro 57
(getexpansionlist 'defblock) -- macro 57
(getformatinfo 'slotnum 'defblock) -- macro 57
(gethash* 'slotnum 'defblock) -- macro 57
(gethash** 'slotnum 'defblock) -- macro 57
(gethash1 'num1 'db1) -- macro 57
(gethash2 'num1 'num2 'db2) -- macro 57
(gethash3 'num1 'num2 'num3 'db2) -- macro 57
(gethash: 'slotnum 'defblock) -- macro 57
(gethash:: 'slotnum 'defblock) -- macro 57
(gethash< 'slotnum 'defblock) -- macro 57
(gethash> 'slotnum 'defblock) -- macro 57
(gethashalias 'defblock) -- macro 57
(gethashinfo 'slotnum 'defblock) -- macro 57
(gethashvalue 'slotnum 'item 'defblock) *

(gethookpath 'item 'path) 10
(getisa 'valblock) -- macro 57
(getpath 'item 'path) 10
(getpname 'defblock) -- macro 57
(getppset 'slotnum 'defblock) -- macro 57
(getpred 'slotmum 'inst) -- macro 57
(getpredpath 'item 'path) 10

(getsinglevalue 'slotmum 'item) *
(getslot 'slotnum 'inst) -- macro 57
(getslothooks 'slotnum 'inst) -- macro 57
(getslotname 'slotnum 'defblock) -- macro 57
(getslottype 'slotnum 'defblock) -- macro 57
(getstructlength 'defblock) -- macro 57

(getstructorsymnum 'strsym) -- macro 57

(getsymbol 'symname) 4
(getsymbolpname 'symbolitem) -- macro 57
(getuniquenum 'defblock) -- macro 57
(getvalue 'slotnum 'inst) 57
(getvarandvalue 'slotnum 'inst 'var) 57
(getvarval 'slotnum 'inst) -- macro 57

(*global* varname) -- fexpr 46
(global variable) -- fexpr 45
(globalp 'variable) 45
(handlehookresult 'oldval 'newval) -- macro **
(hashablevalue 'slotnum 'item 'defblock) -- macro **
(hashslot) -- macro **

(hidden 'command) -- macro 35
(higheroreq 'item1 'item2) -- macro *
(history ['num]) 53
(ind basename [storage] slot1 ...) -- macro 56
(indb 'item ['db]) 14
(indb1 'item 'db) 14
(individual basename [storage] slot1 ...) -- macro 56

(inheritvalue 'structdef) -- macro **
(inlinecreate selector ...) -- macro 14, 56
(inlinefetchcreate selector ...) -- macro 14, 56
(insertdb 'item ['db]) 12
(insertdb1 'item 'db) 12

(insidecreate selector ...) -- fexpr **
(insidefetch patdef expdefs) -- macro **
(insidefetcheverywhere patdef expdefs) -- macro **
(insidepatternize 'item) **
(insidescopy 'item) **
(installadjunct 'adjunctvar) -- macro **
(installglobal 'globalvar) -- macro **
(installvar 'varname) -- macro **

(instatom 'symbol) 7
(isa 'item1 'name) 42
(isanexpanded 'item1 'item2) 42
(islambda 'fcnname) *

(match 'item1 'item2) 46
(msgprintfn '*printval*) 58, 62
(newnum) -- macro *
(nextequal 'stream) 46
(nextitem 'stream) 13
(noalias) -- macro **

(nullstruct 'item) 42
(nullsym 'item) 42
(numberofslot 'slotname 'defblock) -- macro 57
(onesymbol) -- macro **
(ordatom 'symbol) 31
(ordinal name vallist) -- fexpr 30

(pat basename [storage] slot1 ...) -- macro 56
(path fcn 'item 'pathlist ['val]) -- macro 9
(pattern basename [storage] slot1 ...) -- macro 56
(patternize 'item) -- macro 55
(patternizeslot) -- macro **
(pboundp 'a) 45

(pearlprintfn '•printval•) 3, 4
(pexp basename newname [storage] slot1 ...) -- macro 56
(pfunction name [storage] slot1 ...) -- macro 56
(pname 'item) 4
(ppsetform 'slotval 'ppsetname) •

(prefix 'item1 'item2) 52
(prefixcommandhistory) •
(prefixcommandvalue) •
(printdb ['db]) 21
(printdb1 'db) 21
(psymbolp 'potsymbol) 30
(punbound) 45

(punboundatomp 'yyy) •
(putalist 'alist 'inst) -- macro •
(putalistcp 'alist 'inst) -- macro •
(putbasehooks 'hooklist 'defblock) -- macro •
(putdb1 'db1 'db) -- macro •••
(putdb2 'db2 'db) -- macro •••
(putdbchildren 'childlist 'db) -- macro •••
(putdbname 'name 'db) -- macro •
(putdbparent 'parent 'db) -- macro •••
(putdef 'defblock 'valblock) -- macro •••
(putdefaultinst 'valblock 'defblock) -- macro •••
(putenforce 'slotnum 'defblock) -- macro •••
(putexpansionlist 'explist 'defblock) -- macro •••
(putformatinfo 'slotnum 'hashnum 'defblock) -- macro •••
(puthash• 'slotnum 'defblock) -- macro •••
(puthash•• 'slotnum 'defblock) -- macro •••
(puthash1 'num1 'db1 'item) -- macro •
(puthash2 'num1 'num2 'db2 'item) -- macro •
(puthash3 'num1 'num2 'num3 'db2 'item) -- macro •
(puthash: 'slotnum 'defblock) -- macro •••
(puthash:: 'slotnum 'defblock) -- macro •••
(puthash< 'slotnum 'defblock) -- macro •••
(puthash> 'slotnum 'defblock) -- macro •••
(puthashalias 'hashnum 'defblock) -- macro •••
(puthashinfo 'slotnum 'hashnum 'defblock) -- macro •••
(putisa 'isa 'valblock) -- macro •••

(putpath 'item 'path 'value) 10
(putpname 'name 'defblock) -- macro •••
(putppset 'slotnum 'setname 'defblock) -- macro •
(putpred 'slotnum 'value 'inst) -- macro •
(putslot 'slotnum 'value 'inst) -- macro •••
(putslothooks 'slotnum 'slothooklist 'inst) -- macro •
(putslotname 'slotnum 'slotname 'defblock) -- macro •••
(putslottype 'slotnum 'typenum 'defblock) -- macro •••
(putstructlength 'size 'defblock) -- macro •••
(putsymbolpname 'name 'block) -- macro •••
(putuniquenum 'num 'defblock) -- macro •••
(putvarval 'slotnum 'value 'inst) -- macro •••
(reallitatom 'potatom) •

(releasedb 'db) 38
(removedb 'item ['db]) 12
(removedb1 'item 'db) 12
(removeslot) -- macro ••
(revassq 'value 'alist) •
(runbasehooks1 'fcn 'item) 33
(runbasehooks2 'fcn 'item1 'item2 'result) 33
(runslothooks1 'fcn 'item 'slotname 'value) 33
(runslothooks2 'fcn 'item1 'item2 'slotname 'val1 'val2) 33

(savecontinue 'directory 'name)	53
(savefresh 'directory 'name)	53
(savepearl)	•
(scopy 'item) -- macro	55
(scopyslot) -- macro	••
(setblock blockname) -- fexpr	48
(setdbactive 'db) -- macro	•••
(setdbsize 'poweroftwo)	39
(setv var 'val 'environment) -- fexpr	47
(slotequal 'slotnum 'item1 'item2)	•
(slotnametonumber 'slotname 'defblock) -- macro	••
(smerge 'build 'from)	55
(standardfetch 'item ['db])	43
(standardfetch1 'item 'db)	43
(standardmatch 'item1 'item2)	46
(streamp 'potstream)	30
(streamtolist 'stream)	14
(strequal 'item1 'item2)	46
(structurenamep 'potname)	30
(structurep 'potstruct)	30
(symatom 'symbol)	4
(symbol name1 name2 ...) -- fexpr	4
(symbole 'symname)	4
(symbolnamep 'potname)	30
(thawbindings 'struct)	49
(thawblock 'blockname)	49
(thawstruct 'struct)	49
(unbind globalvar) -- fexpr	45
(unbindvars 'structure) -- macro	46
(unboundatomp 'yyy)	•
(valform 'item)	20
(valprint 'item ['lmar ['rmar]])	20
(valprint1 'item 'lmar)	20
(valslotform) -- macro	••
(valueof 'var 'struct)	17
(*var* varname) -- fexpr	46
(varset 'var 'val) -- macro	•
(varvalue var 'val) -- fexpr	17
(visible 'command) -- macro	35
(vp 'item ['lmar ['rmar]])	56
(while 'val &rest 'body) -- macro	54

33. Concept Index

abbreviations 55-56
accessing slots of structures 8-10
accessing structure default instances 7
accessing structure definitions 7
accessing symbols 4

adding slots to structures 40
adding to the data base 12
adjunct variables 30
affecting forced aliasing (^) 27
ako's (expanded structures) 40-42

aliasing of commands 51
aliasing in hashing 27
ampersand (&) hashing 26
and, in predicates 28
anti-aliasing in hashing (<) 27
any 15
automatic storing of structures 6, 56

base hooks 32-37
bases 5
blocks 47-48
building structures 5
building upon data bases 38, 39

changing slots of structures 8
clearing data bases 39
colon (:) hashing 23
colon-colon (::) hashing 24

command aliasing 51
command history 51-53
command history, printing 53
compatibility functions (UCI, Franz) 58-62

controlling running of hooks 33-34
controlling results with hooks 35
controlling unbinding of variables 48-49
converting from internal form 20
copy redefined 54
copying structures 55

creating data bases 38, 39
creating patterns 15-16
creating base structures 5
creating individual structures 6
creating symbols 4

data bases 11
data bases, building upon 39
data bases, clearing 39
data bases, creating 38
data bases, fetching from 12, 19, 25, 42. 43, 46
data bases, freeing 40
data bases, inserting into 12
data bases, printing 21
data bases, releasing 40
data bases, removing from 12
data bases, setting size of 39

debugging 21
debugging print 21
declaring global variables 45

default fetch function	43
default instance for a structure	15
default instance, accessing	7
default match function	46
default printing functions	20, 58, 60-61, 62
default values for slots	14-15
defaults, inherited	41-42
defining structures	5
defining symbols	4
definitions of structures, accessing	7
deleting from the data base	12
demons (hooks)	32-37
disguising in path	10-11
disguising in predicates	29
don't-care matching variable	15
double-colon (::) hashing	24
double-star (**) hashing	24
dumping PEARL for later	53
efficiency despite variables	30
enumerated (ordinal) types	30
environment for variable evaluation	46-47
environment, top level	51-53
environments, in hooks	33
equality of structures	46
equivalences of functions (UCI-Franz)	58-62
error messages	21
evaluating function structures	51
evaluating in create	22
expanded structures	40
expanded structures, fetching	42
feedback, sending	21
fetch, standard	46
fetching expanded structures	42
fetching from the data base	12, 19, 25, 42, 43, 46
fetching with equality (not matching)	46
filling in special forms (in hooks)	33
for loop	54
forced aliasing (>)	26
forest of data bases	39-40
freeing data bases	40
freezing variables	48-49
function equivalences (UCI-Franz)	58-62
function structures	49-51
function structures, evaluating	51
getting symbols	4
global variables	45
greater-than (>) hashing	26
hash aliasing (&)	26
hash marking	17, 23-27
hashing problems	18
hashing with variables	30
hiding functions from hooks	35
hierarchy of structures	40
history mechanism	51-3
history number, printing in prompt	53
hooks	32-37

hooks, affecting result with	35
hooks, controlling running of	33-34
hooks, hiding functions from	35
hooks, making functions visible to	35
hooks, multi-argument	28
hooks, running	33-34
if-added functions (hooks)	32-37
indirection in path	10-11
individuals	6
inheritance in structures	41-42
(.)init.prl file	2-3
inserting in the data base	12
instances	6
integer slots	30
internal access functions	57
internal form printing	21
invisible functions to hooks	35
invisible results from functions	53
isa's (expanded structures)	40-42
less-than (<) hashing	27
lexically scoped variables	47-48
looping functions	54
low level access functions	57
macros, special	56
main data base	11
marking structures for hashing	17, 23-27
match, standard	46
match, without unbinding variables	46
match-anything variable	15
matching process	44
matching two structures	43
matching unbound variables	44
matching-variables	16-17
merging structures	55
modified input line, printing	53
multi-argument matching predicates	28, 32
next item in a stream	13
nilstruct(ure)	14
nilsym(bol)	14
or, in predicates	28
ordinal types	30-31
path functions	10
path indirection	10-11
pattern-matching variables	16-17
patterns	12, 15, 43
patterns in matching	43
predicates for object types	30
predicates in matching	27-29
predicates in matching, when run	44
predicates in matching, multi-argument	28
print names	4
printing PEARL objects	20
printing command history	53
printing data bases	21
printing functions	20
printing functions, standard	3-4, 58, 60-61, 62

printing history number in prompt	53
printing modified input line	53
printing warnings	17
processing a stream	13
prompt	3-4
prompt-read-eval-print loop	2-3, 51
read-eval-print loop	2-3, 51
redirecting in create (! and $)	22
releasing data bases	40
removing from the data base	12
reporting bugs	21
retrieving from the data base	12
returning invisible results	53
running hooks	33-34
running under Franz Lisp	2
running under UCI Lisp	3
saving PEARL for later	53
scalar types	30
short-circuiting in create	22
side effect setting of adjunct variables	30
size of data bases	39
slot hooks	32-37
slot names to numbers	57
slot types	6
slot types, more specific	30
slot values	8-10
slot values in hooks	32
slot values in predicates	28
special forms in hooks	32
special forms in predicates	28
special forms, filling in	33
special macros	56
standard fetch function	43
standard match function	46
star (*) hashing	17, 23
star-star (**) hashing	24
(.)start.prl file	2-3
startup files	2-3
storing structures in the data base	12
storing of structures in atoms	8, 56
streams	13
structure equality	46
structure matching	44-45
structure predicates	28-29
structure slots, further typing	30
structured escapes to Lisp	49-51
structures	5
structures, copying	55
structures, expanded	40
structures, function	49-51
structures, merging	55
symbols	4
testing for nilstruct	42
testing for nilsym	42
testing for object types	30
thawing variables	48-49

top level loop 2-3, 51
top level loop functions 59, 62
triple (**) hashing 24
type tests for objects 30
types in structure slots 31-2

unbinding global variables by match (lack of) 45
unbinding global variables by user 45
unbinding local variables by match 45-6
unbinding local variables by user 46
unbinding of variables, controlling 48-49
up-arrow (^) hashing 27

values of variables 17, 46
values of variables, setting 47
variables in hooks 32
variables in predicates 28
variables with hashing 30
variable, accessing values 17, 46
variables, adjunct 30
variables, controlling unbinding 48-49
variables, freezing 48-49
variables, global 45
variables, lexically scoped 47-48
variable, setting values 47
variables, side effects 30
variables, thawing 48-49
variables, unbinding 46

visible functions to hooks 35
warnings 17
while loop 54

Update of Changes Through PEARL 3.8 April 1983

1. Introduction

This appendix describes the changes that have been made to PEARL since the original manual was produced. It is designed to parallel the sections of the manual so that the original index can be used to find changes.

PEARL is now distributed with Franz Lisp (starting with Opus 38.58). The earliest version of PEARL distributed (with 38.58) was PEARL 3.6. The current update corresponds to version 3.8. The current major and minor version numbers for PEARL are stored in the special variables *pearlmajorversion* and *pearlminorversion* respectively.

With the change in mail protocols and addition of new machines at Berkeley, the form of addresses for bugs and suggestions have been simplified. Bugs, suggestions or queries should be sent to *Pearl–Bugs@Berkeley* or *ucbvax!pearl–bugs*.

2. Running PEARL

PEARL is currently only maintained under Franz Lisp. The current version could be moved back to UCI Lisp (or to other Lisps) fairly easily but has not been for lack of need. Lisp Machine Lisp is the most likely Lisp that PEARL will be moved to next but it has not been done, mostly because of conflicts in the use of the colon character and lack of access to a Lisp Machine.

2.1 Under Franz Lisp

Since PEARL is now part of Franz Lisp, it should be available as */usr/ucb/pearl* or wherever you find *lisp* on your system.

The *.start.pearl* and *.init.pearl* files are actually called *start.prl* and *init.prl* and may optionally be prefixed with a dot "." and/or suffixed with either ".o" or ".l" just as in Franz. The use of the dot prefix and of the ".o" or ".l" is preferred and fastest. Thus PEARL will read the first file found in the following sequence: *.init.prl.o*, *.init.prl.l*, *.init.prl*, *init.prl.o*, *init.prl.l*, or *init.prl* and similarly for *start.prl*. Franz's special variable *$ldprint* is lambda-bound to *nil* during the reading of these two files to disable the printing of "[load .init.prl]".

5. Accessing Slots of Structures

Doing a "*path* **put**" on a slot containing a variable will not set the variable. Rather it replaces the variable with the value provided.

10. Printing Structures, Symbols and Other PEARL Objects

The various printing functions still exist but all call a single formatting function with various options controlled by special atoms. The principle functions are **allform** which does the building of a printable list form for internal PEARL structures and **allprint** which calls *allform*. *Allform* uses the following global variables to determine what form to build:

1. ***abbrevprint*** – a non-*nil* value causes abbreviations to be used whenever possible for any structure except the top level structure passed to a print function. Abbreviations are described at the end of this section. The new functions **abbrevform** and **abbrevprint** lambda-bind this to *t* and then call *allform*. *fullform* binds this to *nil*.
2. ***fullprint*** – a non-*nil* value causes complete information including hooks and predicates to be given when present. *Fullform* (and thus *fullprint*) lambda-binds this to *t* and calls *allform*. *Abbrevform* binds this to *nil*.

Valform lambda-binds both to *nil*. The default value of both is are also *nil*, so that the default action of *allform* when used by itself will be like *valform* unless these special variables are changed. All the default print functions automatically use *allprint* so that they can all be changed by changes to the default values of **abbrevprint** and **fullprint**.

Two other special atoms which affect the behavior of all the printing functions are:

3. ***uniqueprint*** – a non-*nil* value causes a structure which is encountered more than once during the same top-level call to a print function to be translated into exactly the same cons-cells. This saves on cons-cells and also makes it possible for the —*form* functions to handle circular structures, although *sprint* and thus the —*print* functions cannot handle the result. Since most people seldom have duplications within a structure, the default is *nil* (off). The assoc–list of already translated structures is stored in the special atom ***uniqueprintlist***.
4. ***quiet*** – a non-*nil* value disables all printing by any of PEARL's print functions, providing an easy way to disable printing all at once. There is also a function called **quiet** which behaves like *progn*, except that it lambda-binds **quiet** to *t* during the evaluation of its arguments, providing a local island of "quiet".

The standard print functions are designed to handle any Lisp structure. Thus, they spend a significant amount of time determining what kind of object they have been passed. For situations in which you know exactly what type of object you want printed, the functions **structureform/structureprint**, **symbolform/symbolprint**, and **streamform/streamprint** are provided. They assume you know what you are doing and do not ensure that you give them the right type of value.

Adapting PEARL to fit an improvement in Franz, the atoms *showstack-printer* and *trace-printer* are bound to the functions **pearlshowstackprintfn** and **pearltraceprintfn**. **Note** the addition of "pearl" to the beginning of these. The name of *breakprintfn* was also changed to **pearlbreakprintfn** but it is not currently lambda-bindable.

10.1. Abbreviations

As people build larger deeper structures it becomes useful to have some of them abbreviated during printing if they are internal to the structure being printed. When an individual (including default instance) structure is created, an abbreviation atom is stored in it.

This abbreviation is chosen as follows:

1. If the option in *create* of having a structure automatically stored in an atom is used, then that atom is the one used as an abbreviation. Thus the structure created by *(create individual x Pete)* will be given the abbreviation *Pete*.
2. If that option is not used, then default instances will be given the abbreviation *i:x* (where x is the structure type name) and individuals at the top level will be given a name *newsym*-ed from the name of their type. Thus *(create base x)* will make a default instance abbreviated *i:x* and the first structure created with *(create individual x)* will be abbreviated *x0*.
3. *Scopy* and related functions that create new structures from old ones *gensym* the new structure's abbreviation from that of the old structure.

11. Error Messages, Bugs, and Error Handling Abilities

Bugs, complaints and suggestions of useful features (to be added to the current list of 30 or so things on the wish list) should be mailed by electronic mail to **Pearl-Bugs@Berkeley** or **ucbvax!pearl-bugs**.

12. Short-Circuiting and Redirecting Create Using !, $ and Atoms

If an atom is encountered where a value-description was expected in any type of slot, and it is bound to a value of the right type, its value is inserted into the slot. For *symbols*, this is done if the atom is not a symbol name. For *structures*, the atom must evaluate to a structure. For *Lisp* slots, it must simply be bound. For *setof* slots, its value is checked for being of the appropriate type, including depth of nesting.

Note also that a change in the internal representation has made it possible to allow **even atoms** in slots of type *lisp*.

13. More Flexible Hash Selection

Because we have never gotten around to adding fetch functions to take advantage of colon and colon-colon hashing and these two methods really are not useful in normal fetching, they are currently ignored.

For situations in which you wish to create an expanded structure and add new hashing marks to an old slot (rather than replace them), preceding new hash marks with a plus ("+") will cause the old hashing information to be copied before processing the new hashing.

Thus, the sequence

```
(cb x (* a int))
(ce x y (a ^))
(ce x z (+ : a ^))
(ce x w (: + a ^)) ; anomalous use of +
```

will result in:

* hashing in x,
no hashing in y,
both * and : hashing in z, and
only * hashing in w (because of misplacement of +).

Several new hashing methods have been added to PEARL.

A hashing mechanism using the label *** has been added called "triple-star hashing". If slots are labeled with *** and **all** slots so marked are filled with useful values, then the item is hashed under the type of structure plus the values of all these slots. During fetching, this is considered the most useful (that is, specific) hash method.

A hashing mechanism using the label **&&** has been added called "hash focusing". It is designed for people using a data base all of whose entries are of the same type (not required, just common for this application) and enables the contents of a single slot to be used to better discriminate them. Examples of such structures are "planfors", inference rules, or almost any other such extremely-common binary predicates. If a slot labeled && is found when inserting into the database then the item is hashed as if it were the item in the slot so labeled. At fetching time, && is considered less useful than *** or ** and more useful than * or nothing.

This differs from & (hash aliasing) in that hash focusing affects how a structure itself is inserted and fetched, while & simply affects how structures containing this type of structure are treated. For example, suppose the unique numbers of A, B, and C respectively are 1, 2, and 3. C is a symbol. A has one slot X with * and && hashing. B has one slot Y of type symbol with * hashing. Then a structure like (A (X (B (Y C)))) will be indexed the following ways and *fetcheverywhere* (see below) will find it in the following order: the && method will be used first which uses the 2 and 3 from B and its C, (ignoring the 1 of A), and also simply 2 from B; the * on A uses the type of B thus using 1 and 2; it is also looked for under the 1 of A without using 2 or 3. If B had an & in its slot then the * on A is affected by & on B thus using 1 and 3 (ignoring the 2 of B).

Thus, if you consider A, B, and C to be three levels of information in the structure, an item can be hashed under any combination of two of those levels. The normal * method uses levels 1 and 2, the aliasing & method ignores level 2 and uses levels 1 and 3, and the new focussing && method ignores level 1 and uses levels 2 and 3. In addition, the item can be put under 1, 2 or 3 individually by various combinations of marks (1 = none, 2 = :, 3 = :+&). The only unavailable combination of the three is all of them.

16. Attaching Hooks to Structures (If-Added Demons)

Slot hooks are now always inherited and added to, rather than replaced. If the hooks and predicates of a slot are preceded by **instead** then inheriting does not happen and hooks and predicates are replaced.

The atoms for path hooks were misnamed in such a way that you could not use *hidden* and *visible*. Instead of **rungethooks**, and other **run...hooks** forms, they are now ***rungetpathhooks*** and other ***run...pathhooks***. Note that they must be called as (*XXX*path ...) and not (path *XXX* ...) when used with *hidden* and *visible*.

17. Creating and Manipulating Multiple Data Bases

The function *setdbsize* can now be done at any time and will remove all current databases before changing the size, warn the user (if **warn** is set) and recreate **maindb** with the special variable **db** pointing to it.

The function *cleardb* is now a local database clearer and its effects do not extend up the database hierarchy.

19. Creating Expanded Subtypes of Previously Defined Objects

Hashing in old slots inherited by new expanded structures can now be added to by preceding the new hash marks with plus ("+"). See section 13 above.

The name of an old slot inherited by a new expanded structure may be changed by following the new name by the old slotname preceded with an equal sign. Thus for example:

```
pearl> (create base X (A struct))
   (X (A (nilstruct)))
pearl> (create expanded X Y (B =A) (C .....))
   (Y (B (nilstruct)) (C .....)))
```

Note that there may not be a space between the equal sign and the slot name since = is a read macro which expands *=A* into *(*slot* A)* but leaves a single space-surrounded equal sign alone. The actual effect is to add another name to the slot so that it can be later referenced with either name.

20. Fetching Expanded Structures

A fetching function called **fetcheverywhere** exists which gathers **all** the buckets the object could have been hashed into and builds a stream out of all of them (potentially five buckets). There is currently no "expanded" counterpart, since it has the potential of returning *5 times the-depth-of-the-hierarchy* buckets.

21.2 The Matching Process

During matching, if an unbound global variable is set and the match later fails, the value is restored to **pearlunbound**. The names of variables that are set are saved in the special variable ***global-savestack***.

Formerly, there was only one match function which was used by both *standardfetch* and *expandedfetch* and which therefore would match two structures if they were hierarchically related. This is really inappropriate for the standard fetching, so there are now two regular match functions, *standardmatch* and *basicmatch*, which will only match two structures of the same type, and two expanded match functions, *standardexpandedmatch* and *basicexpandedmatch*, which will match two structures which are related hierarchically (one above the other) on the slots they have in common. Streams built by *standardfetch* use the regular versions and and streams built by *expandedfetch* use the expanded versions.

There are now two functions **memmatch** and **memstrequal** which are like *memq* except that they use *match* and *strequal* respectively instead of *eq*.

As of version 3.8, PEARL will now do **unification** of variables in pattern matching. To turn it on, call the function **useunification**. (The current implementation precludes turning it off once it is on but this may be remedied in later versions if we can figure out what it means to stop unifying.)

28. Looping and Copying Functions

The function *scopy* no longer deletes bound adjunct variables.

The standard Franz function *copy* is no longer redefined since the standard version now avoids the copying of hunks.

The functions *scopy* and *patternize* are now exprs rather than macros.

The new function **varreplace** permanently "freezes" the values of slots containing bound variables by replacing all bound variables in an item with their values.

A variation on *scopy* called **intscopy** ("internal scopy") is designed to do the copying as if the copied item were internal to another outer item, thus sharing its local and block variables. Its arguments are the item to be copied and the outer item in whose scopy the copying should be done.

29. Appendix of UCI Lisp functions added to Franz PEARL

The definitions of *de*, *df*, *dm*, *drm* and *dsm* have been modified so that if the special variable ***savedefs*** is *nil* then old definitions of functions are not saved. This is especially useful in compiling (and as a result, assembly and loading) since it will speed them up quite a bit. This also disables the saving of the name of the file that the definition was in. The variable **savedefs** is normally *t* which causes these macros to act as before, saving the definition, etc. If **savedefs** is *nil*, then they simply expand into the appropriate *defun* or *setsyntax*. The following lines should be included in a file to have this effect only at compile time:

```
(eval-when (compile)
        (declare (special *savedefs*))
        (setq *savedefs* nil))
```

If you also want to permanently disable this feature in a lisp, that loads *ucisubset.l*, simply put a *(setq *savedefs* nil)* in your *.lisprc* file AFTER the loading of *ucisubset.l*.

The function *remove* is no longer made equivalent to Franz's *delete* so that Franz's *remove* can be used. The functions *nth*, *push* and *pop* are no longer defined by PEARL, since the new Franz versions are better. (UCI Lisp users note: This switches the arguments to *push*.)

32. Index of Global Variables and Functions With Their Arguments

All special variables in PEARL are now defined with *defvar* so that *fasl*'ing in *pearl.o* at compile time will automatically declare them special again.

All the exprs whose names were of the form *XXXX1* where *XXXX* was the name of a lexpr which was a principle function of PEARL were eliminated (i.e., absorbed by the other form).

34. Compiling Lisp+PEARL Files.

To compile a file of mixed Lisp and PEARL functions with *liszt*, you must first load in the function definitions and special declarations of PEARL by loading the object code. This is the file *pearl.o* which is normally kept in the */usr/lib/lisp* directory and will found automatically by *load*.

Thus, the following should normally be included at the beginning of a PEARL file you wish to compile:

```
(eval-when (compile)
        (declare (special defmacro-for-compiling))
        (setq defmacro-for-compiling t)
        (load 'pearl.o))
(declare (macros t))
```