F.G.H.
5/7/85

LispCourse #25: The Record Package; The Inspector

Records: Data Abstraction in Interlisp

There is a package in Interlisp called the Record Package that makes data abstraction syntactically easy. The Record Package provides a simple syntax that supports writing constructors, selectors, and mutators.

The Record Package is part of CLISP ž statements written in the Record Package syntax are translated by the Lisp interpreter (or compiler) into standard Lisp and then executed.

> All of the syntax described here is just a "pretty" way of stating in CLISP what you could do directly, but less clearly, with CARs, CDRs, GETPROPs, ASSOCs, etc. in straight Interlisp.

The Record Package actually includes an interface to two related functionalities in Interlisp ÿ records and datatypes. We will discuss records first and datatypes a bit later.

Records: Basic Stuff

A ***record*** is basically a description of a list structure. The description names the whole structure (i.e., the record) **and** names each of the parts of the structure.

> In the terminology of the Record Package, each part of the overall record is called a ***field*** of the record.

Once you have described a record structure, you can create instances of the record and can access any field of these instances ***by name*** using the special Record Package syntax.

> Example:
>
> > *(fetch (Message Header) of NextMessage)* is a statement that retrieves the Header part of a Message record that is the value of the atom NextMessage.

Important note: the word *record* is used to denote both the *description* of the list structure and the actual lists that are *instances* of the structure described. Which of these is intended should be clear from the context.

Record Declarations

Before a record can be used it must be declared.

A **record declaration statement** names the record and describes all of the fields in the record. It has the form:

(RECORD *RecordName Fields ExtraStuff1 ExtraStuff2 ...*)

RECORD is a keyword indicating that this is a record declaration.

RecordName is the name of the record. RecordNames must be unique in the whole system (among both records and datatypes).

Fields is a list of the parts (i.e., fields) of the record. Each element in the list is a non-NIL litatom that serves as the name of the field.

The list can also contain NILs, which stand for unnamed (and therefore unaccessible) fields. Finally, the list can contain integers, which stand for the specified number of unnamed fields.

The *ExtraStuffI* staements are optional. If present, each *ExtraStuffI* can be any of several kinds of information.

Most importantly, the *ExtraStuffI* can be an assignment statement of the form:

FieldName _ Form

FieldName is one of the named fields from the *Fields* list.

Form is any Lisp form.

This assignment statement specifies the default value for the field named *FieldName* in the record. When you create an instance of the record, if *FieldName* is not explicitly given a value in the CREATE statement, then the value returned by evaluating *Form* will be used to fill-in the field.

Examples:

(RECORD PersonsName (First Middle Last)
First _ 'John Middle _ 'Dunce Last _ 'Doe)

(RECORD DatabaseEntry (Name OfficeNumber PhoneNumber))

Note: A record declaration statement is NOT an executable statement ÿ i.e., it doesn't evaluate to anything. It just serves to describe the record to CLISP, which will use this information to translate the executable statements that access instances of this record.

Constructing Record Instances: The CREATE statement.

Once a record has been declared, you can construct instances of the record using the CREATE statement.

The CREATE statement has the following format:

(CREATE *RecordName Assignment1 Assignment2 ...*)

RecordName is the name of the record you want to create an instance of.

The *AssignmentI* statements are optional. If present, each *AssignmentI* is a statement that specifies the value to be given to a particular field of the record when it is created. This statement should have the format:

FieldName _ Form

FieldName is the name of one of the fields of *RecordName.*

Form is any Interlisp form, the value of which will be placed in the field *FieldName* when the record is created.

When the CREATE statement is evaluated, it returns a list that is an instance of the specified record. The value of each field in the record is determined as follows:

If there was an assignment statement for that field in the CREATE statement, then the value of the form in that assignment statement is used.

Otherwise, if there was default assignment statement for that field in the record declaration for the record, then the value of the form in that assignment statement is used.

Otherwise, the field is set to NIL.

Examples:

```
30_(RECORD PersonsName (First Middle Last)
First _ 'John Last _ 'Doe)
PersonsName
31_(CREATE PersonsName)
(John NIL Doe)
32_(CREATE PersonsName First _ 'Sam)
(Sam NIL Doe)
33_(CREATE PersonsName First _ 'Sam Last _ 'Smith
Middle _ (CAR (LIST 'A. 'B.)))
(Sam A. Smith)
```

Selecting Fields in a Record: The fetch statement

Given a record (instance), you can select any of its fields using the ***fetch*** statement.

The ***fetch*** has the following format:

(fetch (*RecordName FieldName*) of *Form*)

fetch and **of** are keywords.

RecordName is the name of the record (description) being used.

FieldName is the name of the field to be selected from the record instance of type *RecordName*.

Form is an Interlisp form that evaluates to a record of type *RecordName*.

When the ***fetch*** statement is evaluated, it will return the value of the named field from the record that is the value of the given form.

Examples:

33_(SETQ Person (CREATE PersonsName First _ 'Sam Last _ 'Smith
Middle _ 'A))

(Sam A. Smith)

34_Person

(Sam A. Smith)

35_ (fetch (PersonsName Last) of Person)

Smith

36_ (fetch (PersonsName First) of Person)

Sam

37_ (fetch (PersonsName First) of (CREATE PersonsName))

John

Note: if the *FieldName* is unambiguous, ***FieldName*** can be used in place of **(*RecordName FieldName*)**. *FieldName* is unambiguous if there is only one record *in the entire system* having a field with that name.

Opinion: Using *FieldName* instead of *(RecordName FieldName)* is just lousy programming style because it can lead to lots of trouble when you (or more likely someone else) adds a second record to the system that just happens to use *FieldName*.

Mutating Fields in a Record: The replace statement

Given a record (instance), you can mutate (i.e., change) any of its fields using the ***replace*** statement.

The ***replace*** has the following format:

(replace (*RecordName FieldName*) of *Form1* with *Form2*)

replace, of, and **with** are keywords.

RecordName is the name of the record (description) being used.

FieldName is the name of the field to be changed from the record instance of type *RecordName*.

Form1 is an Interlisp form that evaluates to a record of type *RecordName*.

Form2 is an Interlisp form whose value will be used as the new value of the field.

When the ***replace*** statement is evaluated, it will replace the old value of the specified field in the specified record (i.e., the record returned by evaluating *Form1*) using the specified new value (i.e., the value returned by evaluating *Form2*). The value returned by the ***replace*** statement is the new value.

Examples:

38_Person
(Sam A Smith)
39_ (replace (PersonsName Last) of Person with 'Jones)
Jones
40_ (fetch (PersonsName Last) of Person)

Jones
41_Person
(Sam A Jones)
42_(replace (PersonsName First) of Person with 'Jane)
Jane
43_(replace (PersonsName Middle) of Person with 'Maria)
Maria
44_Person
(Jane Maria Jones)

Records as Data Abstraction

Since records allow you access to data structures by *name* rather than by *structure*, they (to a great extent) isolate the program from changes in the underlying data structure.

Consider the alternative record definitions: *(RECORD Name (First Middle Last))* and *(RECORD Name (Last First Middle))* and *(RECORD Name (SS# BirthPlace Last Middle First)*

CREATE, fetch, and replace statements for all of these would be identical.

> For example: *(fetch (Name First) of NewName)* would always return the first name field, although it would be the CAR of the first type of record, the CADR of the second type of record, and the 5th element of the third type of record.

Therefore changing the underlying record stucture between any of these three record declarations (and many more alternative declarations) would have no effect on the programs that accessed the records using CREATE, fetch, and replace.

Hierarchical Data Structures

Often data structrues are hierarchical, with smaller data structures embedded in more global data structures.

An example from last time is the DatabaseEntry list structure. Each DatabaseEntry consisted of three items: a name, a phone number, and an office number. Each of these three items was in turn a list structure consisting of two or three atoms.

DatabaseEntry: List of three items
Name: List of three items
First: atom
Middle: atom
Last: atom
PhoneNumber: List of two items
Prefix: atom
Extension: atom
OfficeNumber: List of two items
BldgNumber: atom
RoomNumber: atom

To deal with this hierarchical data structure using the Record Package, you would define four independent records for DatabaseEntry, Name, PhoneNumber, and OfficeNumber.

For example:

(RECORD DatabaseEntry (Name PhoneNumber OfficeNumber))

(RECORD Name (First Middle Last))

(RECORD PhoneNumber (Prefix Extension) Prefix _ 494)

(RECORD OfficeNumber (Bldg RoomNumber) Bldg _ 35)

To select the Extension or RoomNumber from a given database entry you could use embedded **fetch** statements:

Example:

45_ Entry

((Sam A Smith)(494 4431)(35 2319))

46_ (fetch (PhoneNumber Extension) of
(fetch (DatabaseEntry PhoneNumber) of Entry))

4431

47_ (fetch (OfficeNumber RoomNumber) of
(fetch (DatabaseEntry OfficeNumber) of Entry))

2319

The Record Package provides a shorthand syntax for these embedded access (i.e., **fetch** and **replace**) statements, provided that you have named the fields of the various records correctly.

In particular, if the field name of the *embedding* record is the same as the record name of the *embedded* record, then you can combine the **fetch** statements by using a ***path name*** as the field specification in a single fetch statement. This ***path name*** should name the highest level record followed by the field that contains the second level record followed by the field that contains the third level record and so on until the target field is named.

Examples:

48_Entry

((Sam A Smith)(494 4431)(35 2319))

49_ (fetch (DatabaseEntry PhoneNumber Extension) of Entry)

4431

50_ (fetch (DatabaseEntry OfficeNumber RoomNumber) of Entry))

2319

Example that doesn't work with the shorthand syntax because the embedding record field name is not the same as the embedded record name:

60_(RECORD DatabaseEntry (Name ***Phone*** OfficeNumber))

DatabaseEntry

61_(RECORD ***PhoneNumber*** (Prefix Extension) Prefix _ 494)

PhoneNumber

62_(SETQ Entry
(CREATE DatabaseEntry Name _ 'Foo
Phone _ (CREATE PhoneNumber
Extension _ 4456)

```
                    OfficeNumber _ 99))
(Foo (494 4456) 99)
63_(fetch (DatabaseEntry Phone Extension) of Entry)
 no such record path
 at   ... (DatabaseEntry Phone Extension) of Entry)
in    (fetch (DatabaseEntry Phone Extension) of Entry)

UNDEFINED CAR OF FORM
fetch

64_(fetch (DatabaseEntry PhoneNumber Extension) of Entry)
 no such record path
 at   ... (DatabaseEntry PhoneNumber Extension) of Entry)
in    (fetch (DatabaseEntry PhoneNumber Extension) of Entry)

UNDEFINED CAR OF FORM
fetch

65_(fetch (PhoneNumber Extension) of
              (fetch (DatabaseEntry Phone) of Entry))
4456
```

Records constructed from different list structures

The (RECORD ...) statement declares a record data structure that is to be constructed from an ordinary list. Each field in the record declaration is to be an element in that list.

There are many alternative list structures that can be constructed using the Record Package. These alternative list structures are declared, created, and accessed using the same statements as are RECORDs, except that the declaration statement starts with the given record type instead of RECORD.

Some of the most important alternative list structures are:

TYPERECORD ž just like a RECORD except that the first element of the list is always the name of record.

Example:

```
51_ (TYPERECORD Name (First Middle Last))
Name
52_ (CREATE Name
          First _ 'Sam Middle _ 'Ishikawa Last _
          'Watanabe)
(Name Sam Ishikawa Watanabe)
53_ (fetch (Name First) of IT)
Sam
```

ASSOCRECORD ž the list created is in ASSOC list format with the field names as keys.

Example:

```
54_ (ASSOCRECORD Name (First Middle Last))
Name
55_ (CREATE Name
          First _ 'Sam Middle _ 'Ishikawa Last _
          'Watanabe)
((First . Sam)(Middle . Ishikawa)(Last . Watanabe))
56_ (fetch (Name First) of IT)
Sam
```

PROPRECORD ž the list created is in Prop list format with the field names as props.

Example:

```
57_ (PROPRECORD Name (First Middle Last))
Name
58_ (CREATE Name
            First _ 'Sam Middle _ 'Ishikawa Last _
            'Watanabe)
(First Sam Middle Ishikawa Last Watanabe)
59_ (fetch (Name First) of IT)
Sam
```

Determining the record type of an object: The TYPE? statement

There is a fourth tool (in addition to constructors, selectors, and mutators) that is very handy in writing programs with data abstraction ž the **predicator**.

A predicator for a given data structure is a predicate that returns a non-NIL value if ist argument is an example of the data structure and NIL otherwise.

For example, a predicator for the name list from last week might be:

```
(DEFINEQ (LC.NameP
    (LAMBDA (List)
        (* * Check if List is a name list structure)
        (AND
            (LISTP List)
            (EQUAL (LENGTH List) 3)
            (LITAOM (CAR List))
            (LITATOM (CADR List))
            (LITATOM (CADDR List))))))
```

The ***TYPE?*** statement is the Record Package mechanism for dealing with predicators.

To use the TYPE? mechanism, you must add a predicator as one of the *ExtraStuffI* arguments in the record declaration statement for each record. The argument must have the format:

(TYPE? *Form*)

TYPE? is a keyword.

Form can be an Interlisp expression using the variable DATUM.

DATUM is bound to the record instance to be tested and then the expression is evaluated, returning a NIL or a non-NIL value. Form can also be an atom which is the name of a function having one argument, the record instance to be tested. In this case, the function is applied to the record instance to get the NIL or non-NIL value.

Example:

```
(RECORD Name (First Middle Last)
        (TYPE?
                (AND
                        (LITATOM (fetch (Name First) of
                        DATUM))
                        (LITATOM (fetch (Name Middle) of
                        DATUM))
                        (LITATOM (fetch (Name Last) of
                        DATUM)))))
```

Once you have a TYPE? clause in the record declaration for a record, you can ask of any Lisp object whether it is an instance of that record type. To do this, use the ***TYPE?*** statement.

The ***TYPE?*** statement has the following format:

(TYPE? *RecordName Form*)

TYPE? is a keyword.

RecordName is the name of an already declared record type.

Form is any Interlisp form.

When the ***TYPE?*** statement is evaluated, the TYPE? entry is retrieved from the record declaration for *RecordName* and applied (as described under the TYPE?

clause above) to the value returned by evaluating *Form*. The value of the ***TYPE?*** statement is the value of this application.

For example, the record declarations for the database entries might be ammended as follows:

```
(RECORD DatabaseEntry (Name PhoneNumber OfficeNumber)
        (TYPE?
                (AND (EQUAL (LENGTH DATUM) 3)
                (TYPE? Name (fetch (DatabaseEntry Name) of DATUM))
                (TYPE? PhoneNumber
                        (fetch (DatabaseEntry PhoneNumber) of DATUM))
                (TYPE? OfficeNumber
                        (fetch (DatabaseEntry OfficeNumber) of
                        DATUM)))))

(RECORD Name (First Middle Last)
        (TYPE?
                (AND
                        (LITATOM (fetch (Name First) of DATUM))
                        (LITATOM (fetch (Name Middle) of DATUM))
                        (LITATOM (fetch (Name Last) of DATUM)))))

(RECORD PhoneNumber (Prefix Extension) Prefix _ 494
        (TYPE?
                (AND
                        (NUMBERP (fetch (PhoneNumber Prefix) of
                        DATUM))
                        (NUMBERP (fetch (PhoneNumber Extension) of
                        DATUM)))))

(RECORD OfficeNumber (Bldg RoomNumber) Bldg _ 35
        (TYPE?
                (AND
                        (NUMBERP (fetch (OfficeNumber Bldg) of
                        DATUM))
                        (NUMBERP (fetch (OfficeNumber RoomNumber)
                        of DATUM)))))
```

With these declarations, you can do the following:

60_(TYPE? DatabaseEntry ’(Foo (494 4456) Bar))
NIL
61_(TYPE? DatabaseEntry ’((Frank G Halasz)(494 4320)(35 1654)))
1654

Final note: For TYPERECORD records you don’t need to specify a TYPE? clause in the reord declaration statement for the TYPE? statement to work. There is a default TYPE? clause that just checks for a name match between the *RecordName* given in the TYPE? statement and the CAR of the record instance.

DATATYPES: Defining New Objects in the Interlisp System

Records build data structures out of list structures. As such, records don’t really create new types of objects in your Interlisp system; they just provide a convenient interface to lists that have a particular format.

In contrast, a ***DATATYPE*** creates a whole new type of object in the Interlisp system.

Basically, a DATATYPE behaves much like a RECORD. The CREATE, fetch, replace, and TYPE? statements all work exactly the same for DATATYPEs and for RECORDs.

But when you declare a DATATYPE, Interlisp goes through a lot of overhead to create a new type of object in its world.

Then, when you create an instance of the DATATYPE, you are not creating a list structure but a object of the new type with its own internal representation different from lists. Similarly, when you access the DATATYPE, you are accesings this new object type rather than a list.

The advantage of DATATYPEs over RECORDs is that access to any field in a DATATYPE is much, much faster.

The disadvantage of DATATYPEs is that there is a big overhead for the system in defining and managing new DATATYPEs.

Thus, if you are going to create lots and lots of instances of a given data structure and access these instances frequently, then you might want to make that structure a DATATYPE. If you are creating a few instance that are accessed infrequently, then you might want to make the structure a RECORD.

Since RECORDs and DATATYPEs are created and accessed in the same way, you can easily write and test a program using RECORDs and then switch to DATATYPEs when the program goes into real use.

Example:

To make the Name RECORD into a DATATYPE, all you would have to do is declare a DATATYPE instead of a RECORD as follows:

(DATATYPE Name (First Middle Last))

Note that TYPE? is automatically defined when the DATATYPE is declared, you do not need a TYPE? clause in the DATATYPE declaration statement.

Thereafter, Name will behave exaclty as in all the examples above, except it will be a Name object rather than a list structure.

In particular, when you CREATE a Name, you will get back a Name object rather than a list.

```
87_ (CREATE Name First _ 'Sam Last _ 'Smith Middle _ 'A)
{Name}#51,142770
88_ (fetch (Name First) of IT)
Sam
89_ (TYPE? Name NewName)
T
90_ (TYPE? Name 'Foo)
NIL
```

System DATATYPEs

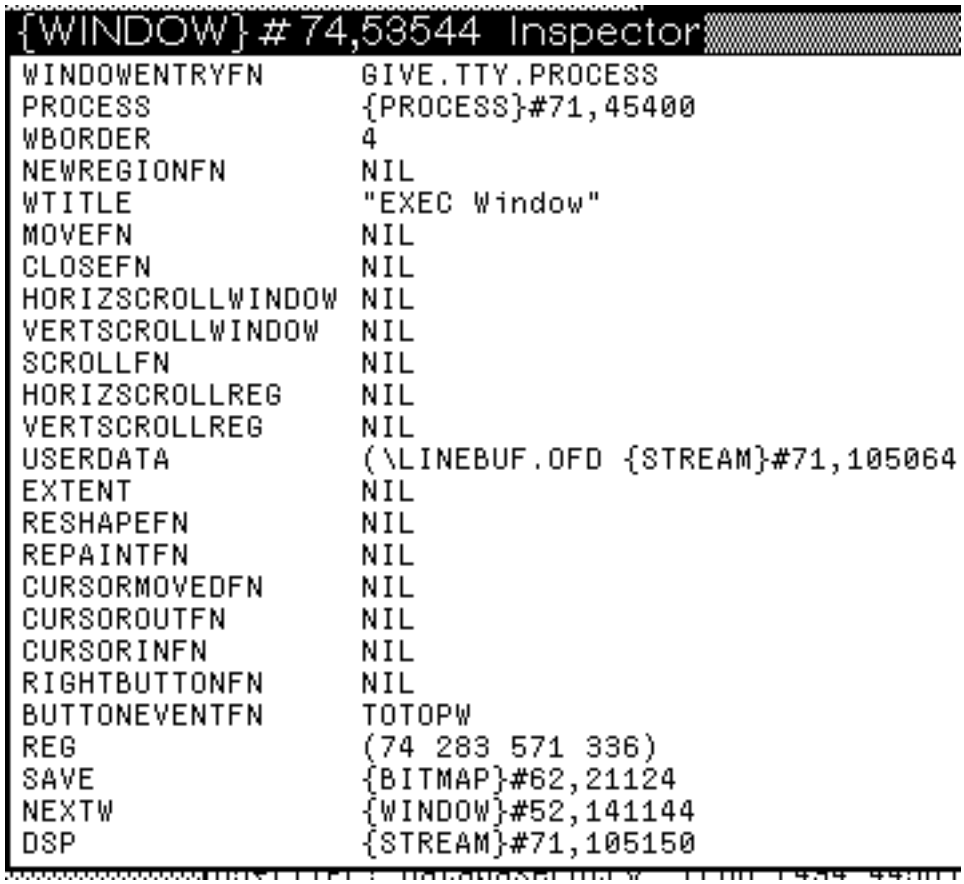
Many, many data structures in the Interlisp system are implemented in terms of DATATYPES.

For example, Windows are just a DATATYPE data structure. The function CREATEW calls (CREATE WINDOW ...) and eventually returns the WINDOW object returned by this CREATE statement.

In fact, the following is the DATATYPE declaration for WINDOW from the Interlisp system code:

```
(DATATYPE WINDOW
    (DSP NEXTW SAVE REG BUTTONEVENTFN RIGHTBUTTONFN
    CURSORINFN CURSOROUTFN CURSORMOVEDFN REPAINTFN
    RESHAPEFN EXTENT USERDATA VERTSCROLLREG
    HORIZSCROLLREG SCROLLFN VERTSCROLLWINDOW
    HORIZSCROLLWINDOW CLOSEFN MOVEFN WTITLE
    NEWREGIONFN WBORDER PROCESS WINDOWENTRYFN))
```

A typical window might be filled in as follows:



Editing record declarations and record instance: EditRec and The Inspector

To be completed in class

References

Chapter 3 of the IRM.

Exercises

Redo the problem from LispCourse #23 using the Record Package.