

CHAPTER 3

THE RECORD PACKAGE

The advantages of “data abstraction” have long been known: more readable code, fewer bugs, the ability to change the data structure without having to make major modifications to the program, etc. The record package encourages and facilitates this good programming practice by providing a uniform syntax for creating, accessing and storing data into many different types of data structures (arrays, list structures, association lists, etc.) as well as removing from the user the task of writing the various manipulation routines. The user declares (once) the data structures used by his programs, and thereafter indicates the manipulations of the data in a data-structure-independent manner. Using the declarations, the record package automatically computes the corresponding Interlisp expressions necessary to accomplish the indicated access/storage operations. If the data structure is changed by modifying the declarations, the programs automatically adjust to the new conventions.

The user describes the format of a data structure (record) by making a “record declaration” (see page 3.5). The record declaration is a description of the record, associating names with its various parts, or “elds”. For example, the record declaration (RECORD MSG (FROM TO . TEXT)) describes a data structure called MSG, which contains three elds: FROM, TO, and TEXT. The user can reference these elds by name, to retrieve their values or to store new values into them, by using the FETCH and REPLACE operators (page 3.1). The CREATE operator (page 3.3) is used for creating new instances of a record, and TYPE? (page 3.4) is used for testing whether an object is an instance of a particular record. (note: all record operators can be in either upper or lower case.)

Records may be implemented in a variety of different ways, as determined by the first element (“record type”) of the record declaration. RECORD (used to specify elements and tails of a list structure) is just one of several record types currently implemented. The user can specify a property list format by using the record type PROPRECORD, or that elds are to be associated with parts of a data structure via a specified hash array by using the record type HASHLINK, or that an entirely new data type be allocated (as described on page 3.14) by using the record-type DATATYPE.

The record package is implemented through the DWIM/CLISP facilities, so it contains features such as spelling correction on eld names, record types, etc. Record operations are translated using all CLISP declarations in effect (standard/fast/undoable); it is also possible to declare local record declarations that override global ones (see page 16.9).

The lisp package includes a RECORDS lisp package command for dumping record declarations (page 11.25), and FILES? and CLEANUP will inform the user about records that need to be dumped.

3.1 FETCH AND REPLACE

The elds of a record are accessed and changed with the FETCH and REPLACE operators. If the record MSG has the record declaration (RECORD MSG (FROM TO . TEXT)), and X is a MSG data structure, (fetch FROM of X) will return the value of the FROM eld of X, and (replace FROM of X with

FETCH and REPLACE

Y) will replace this eld with the value of Y. In general, the value of a REPLACE operation is the same as the value stored into the eld.

Note that the form (fetch FROM of X) implicitly states that X is an instance of the record MSG, or at least it should to be treated as such for this particular operation. In other words, the interpretation of (fetch FROM of X) never depends on the value of X. Therefore, if X is not a MSG record, this may produce incorrect results. The TYPE? record operation (page 3.4) may be used to test the types of objects.

If there is another record declaration, (RECORD REPLY (TEXT . RESPONSE)), then (fetch TEXT of X) is ambiguous, because X could be either a MSG or a REPLY record. In this case, an error will occur, AMBIGUOUS RECORD FIELD. To clarify this, FETCH and REPLACE can take a list for their “eld” argument: (fetch (MSG TEXT) of X) will fetch the TEXT eld of an MSG record.

Note that if a eld has an *identical* interpretation in two declarations, e.g. if the eld TEXT occurred in the same location within the declarations of MSG and REPLY, then (fetch TEXT of X) would *not* be considered ambiguous.

Another complication can occur if the elds of a record are themselves records. The elds of a record can be further broken down into sub-elds by a “subdeclaration” within the record declaration (see page 3.10). For example,

```
(RECORD NODE (POSITION . LABEL) (RECORD POSITION (XLOC . YLOC)))
```

permits the user to access the POSITION eld with (fetch POSITION of X), or its subeld XLOC with (fetch XLOC of X).

The user may also elaborate a eld by declaring that eld name in a *separate* record declaration (as opposed to an embedded subdeclaration). For instance, the TEXT eld in the MSG and REPLY records above may be subdivided with the separate record declaration (RECORD TEXT (HEADER . TXT)). Fields of subelds (to any level of nested subelds) are accessed by specifying the “data path” as a list of record/eld names, where there is some path from each record to the next in the list. For instance, (fetch (MSG TEXT HEADER) of X) indicates that X is to be treated as a MSG record, its TEXT eld should be accessed, and *its* HEADER eld should be accessed. Only as much of the data path as is necessary to disambiguate it needs to be specified. In this case, (fetch (MSG HEADER) of X) is sufficient. The record package interprets a data path by performing a tree search among all current record declarations for a path from each name to the next, considering first local declarations (if any) and then global ones. The central point of separate declarations is that the (sub)record is *not* tied to another record (as with embedded declarations), and therefore can be used in many different contexts. If a data-path rather than a single eld is ambiguous, (e.g., if there were yet another declaration (RECORD TO (NAME . HEADER)) and the user specified (fetch (MSG HEADER) of X)), the error AMBIGUOUS DATA PATH is generated.

FETCH and REPLACE forms are translated using the CLISP declarations in effect. FFETCH and FREPLACE are versions which insure fast CLISP declarations will be in effect, /REPLACE insures undoable declarations.

THE RECORD PACKAGE

3.2 CREATE

Record operations can be applied to arbitrary structures, i.e., the user can explicitly creating a data structure (using CONS, etc), and then manipulate it with FETCH and REPLACE. However, to be consistent with the idea of data abstraction, new data should be created using the same declarations that define its data paths. This can be done with an expression of the form:

```
(CREATE RECORD- NAME . ASSIGNMENTS )
```

A CREATE expression translates into an appropriate Interlisp form using CONS, LIST, PUTHASH, ARRAY, etc., that creates the new datum with the various elds initialized to the appropriate values. ASSIGNMENTS is optional and may contain expressions of the following form:

FIELD-NAME _ FORM	Specifies initial value for FIELD-NAME .
USING FORM	Specifies that for all elds not explicitly given a value, the value of the corresponding eld in FORM is to be used.
COPYING FORM	Similar to USING except the corresponding values are copied (with COPYALL).
REUSING FORM	Similar to USING, except that wherever possible, the corresponding <i>structure</i> in FORM is used.
SMASHING FORM	A new instance of the record is not created at all; rather, the value of FORM is used and smashed.

The record package goes to great pains to insure that the order of evaluation in the translation is the same as that given in the original CREATE expression if the side effects of one expression might affect the evaluation of another. For example, given the declaration (RECORD CONS (CAR . CDR)), the expression (CREATE CONS CDR_X CAR_Y) will translate to (CONS Y X), but (CREATE CONS CDR_(FOO) CAR_(FIE)) will translate to ((LAMBDA (\$\$1) (CONS (PROGN (SETQ \$\$1 (FOO)) (FIE)) \$\$1))) because FOO might set some variables used by FIE.

Note that (CREATE RECORD REUSING FORM ...) does not itself do any destructive operations on the value of FORM. The distinction between USING and REUSING is that (CREATE RECORD REUSING FORM ...) will incorporate as much as possible of the old data structure into the new one being created, while (CREATE RECORD USING FORM ...) will create a completely new data structure, with only the contents of the elds re-used. For example, CREATE REUSING a PROPRECORD just CONSeS the new property names and values onto the list, while CREATE USING copies the top level of the list. Another example of this distinction occurs when a eld is elaborated by a subdeclaration: USING will create a new instance of the sub-record, while REUSING will use the old contents of the eld (unless some eld of the subdeclaration is assigned in the CREATE expression.)

If the value of a eld is neither explicitly specified, nor implicitly specified via USING, COPYING or REUSING, the default value in the declaration is used, if any, otherwise NIL. (Note: For BETWEEN elds in DATATYPE records, N₁ is used; for other non-pointer elds zero is used.) For example, following (RECORD A (B C D) D _ 3),

```
(CREATE A B_T) ==> (LIST T NIL 3)
```

```
(CREATE A B_T USING X) ==> (LIST T (CADR X) (CADDR X))
```

TYPE?

```
(CREATE A B_T COPYING X) ==> [LIST T (COPYALL (CADR X)) (COPYALL (CADDR X])  
(CREATE A B_T REUSING X) ==> (CONS T (CDR X))
```

3.3 TYPE?

The record package allows the user to test if a given datum “looks like” an instance of a record. This can be done via an expression of the form

```
(TYPE? RECORD- NAME FORM )
```

TYPE? is mainly intended for records with a record type of DATATYPE or TYPE-RECORD. For DATATYPEs, the TYPE? check is exact; i.e. the TYPE? expression will return non-NIL only if the value of FORM is an instance of the record named by RECORD- NAME . For TYPE-RECORDs, the TYPE? expression will check that the value of FORM is a list beginning with RECORD- NAME . For ARRAY-RECORDs, it checks that the value is an array of the correct size. For PROP-RECORDs and ASSOC-RECORDs, a TYPE? expression will make sure that the value of FORM is a property/association list with property names among the eld- names of the declaration.

Attempting to execute a TYPE? expression for a record of type ACCESS-FNS, HASH-LINK or RECORD will cause an error, TYPE? NOT IMPLEMENTED FOR THIS RECORD. The user can (re)define the interpretation of TYPE? expressions for a particular declaration by inclusion of an expression of the form (TYPE? COM) in the record declaration (see page 3.9).

3.4 WITH

Often it is necessary to manipulate the values of the elds of a particular record. The WITH construct can be used to talk about the elds of a record as if they were variables within a lexical scope:

```
(WITH RECORD- NAME RECORD- INSTANCE FORM 1 FORM N )
```

RECORD- NAME is the name of a record, and RECORD- INSTANCE is an expression which evaluates to an instance of that record. The expressions FORM 1 FORM N are evaluated so that references to variables which are eld- names of RECORD- NAME are implemented via fetch and SETQs of those variables are implemented via replace.

For example, given

```
(RECORD REC-N (FLD1 FLD2))  
(SETQ INST (CREATE REC-N FLD1 _ 10 FLD2 _ 20))
```

Then the construct

```
(with REC-N INST (SETQ FLD2 (PLUS FLD1 FLD2))
```

is equivalent to

THE RECORD PACKAGE

(replace FLD2 of INST with (PLUS (fetch FLD1 of INST) (fetch FLD2 of INST))

Note that the substitution is lexical: this operates by actually doing a substitution inside the forms.

3.5 RECORD DECLARATIONS

A record is defined by evaluating a record declaration,¹ which is an expression of the form:

```
(RECORD- TYPE RECORD- NAME FIELDS . RECORD- TAIL)
```

RECORD- TYPE specifies the “type” of data being described by the record declaration, and thereby implicitly specifies how the corresponding access/storage operations are performed. RECORD- TYPE currently is either RECORD, TYPE-RECORD, ARRAY-RECORD, ATOM-RECORD, ASSOC-RECORD, PROP-RECORD, DATATYPE, HASHLINK, ARRAY-BLOCK or ACCESS-FNS. RECORD and TYPE-RECORD are used to describe list structures, DATATYPE to describe user data-types, ARRAY-RECORD to describe arrays, ATOM-RECORD to describe (the property list of) litatoms, PROP-RECORD to describe lists in property list format, and ASSOC-RECORD to describe association list format. HASHLINK can be used with any type of data: it simply specifies the data path to be a hash-link. ACCESS-FNS is also type-less; the user specifies the data-paths in the record declaration itself, as described below.

RECORD- NAME is a litatom used to identify the record declaration for creating instances of the record via CREATE, testing via TYPE?, and dumping to lisp via the RECORDS lisp package command (page 11.25). DATATYPE and TYPE-RECORD declarations also use RECORD- NAME to identify the data structure (as described below).

FIELDS describes the structure of the record. Its exact interpretation varies with RECORD- TYPE :

RECORD

[Record Type]

FIELDS is a list structure whose non-NIL literal atoms are taken as eld- names to be associated with the corresponding elements and tails of a list structure. For example, with the record declaration (RECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CAR X).

NIL can be used as a place marker to fill an unnamed eld, e.g., (A NIL B) describes a three element list, with B corresponding to the third element. A number may be used to indicate a sequence of NILs, e.g. (A 4 B) is interpreted as (A NIL NIL NIL NIL B).

TYPE-RECORD

[Record Type]

Similar to RECORD, except that RECORD- NAME is also used as an indicator in CAR of the datum to signify what “type” of record it is. This type- eld is used by the record package in the translation of TYPE? expressions. CREATE will insert an extra eld containing RECORD- NAME at the beginning of the structure, and the translation of the access and storage functions will take this extra eld into

¹Local record declarations are defined by including an expression of this form in the CLISP declaration for that function, rather than evaluating the expression itself (see page 16.10).

Record Declarations

account. For example, for (TYPE-RECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CADR X), not (CAR X).

ASSOC-RECORD [Record Type]

FIELDS is a list of literal atoms. The elds are stored in association-list format:

```
((FIELDNAME1 . VALUE1) (FIELDNAME2 . VALUE2) )
```

Accessing is performed with ASSOC (or FASSOC, depending on current CLISP declarations), storing with PUTASSOC.

PROP-RECORD [Record Type]

FIELDS is a list of literal atoms. The elds are stored in “property list” format:

```
(FIELDNAME1 VALUE1 FIELDNAME2 VALUE2 )
```

Accessing is performed with LISTGET, storing with LISTPUT.

Both ASSOC-RECORD and PROP-RECORD are useful for defining data structures in which it is often the case that many of the elds are NIL. A CREATE for these record types only stores those elds which are non-NIL. Note, however, that with the record declaration (PROP-RECORD FIE (H I J)) the expression (CREATE FIE) would still construct (H NIL), since a later operation of (replace J of X with Y) could not possibly change the instance of the record if it were NIL.

ARRAY-RECORD [Record Type]

FIELDS is a list of eld-names that are associated with the corresponding elements of an array. NIL can be used as a place marker for an unnamed eld (element). Positive integers can be used as abbreviation for the corresponding number of NILs. For example, (ARRAY-RECORD (ORG DEST NIL ID 3 TEXT)) describes an eight element array, with ORG corresponding to the first element, ID to the fourth, and TEXT to the eighth.

Note that ARRAY-RECORD only creates arrays of pointers. Other kinds of arrays must be implemented by the user with ACCESSFNS.

HASHLINK [Record Type]

FIELDS is either an atom FIELD-NAME, or a list (FIELD-NAME HARRA YNAME HARRA YSIZE). HARRA YNAME indicates the hash-array to be used; if not given, SYSHASHARRAY is used. HARRA YSIZE is used for initializing the hash array: if HARRA YNAME has not been initialized at the time of the declaration, it will be set to (LIST (HARRAY (OR HARRA YSIZE 100))). HASHLINKs are useful as subdeclarations to other records to add additional elds to already existing data-structures. For example, suppose that FOO is a record declared with (RECORD FOO (A B C)). To add an additional eld BAR, without modifying the already-existing data structures, redeclare FOO with:

```
(RECORD FOO (A B C) (HASHLINK FOO (BAR BARHARRAY)))
```

Now, (fetch BAR of X) will translate into (GETHASH X BARHARRAY), hashing on the existing list X.

ATOM-RECORD [Record Type]

FIELDS is a list of property names, e.g., (ATOM-RECORD (EXPR CODE MACRO

THE RECORD PACKAGE

BLKLIBRARYDEF)). Accessing is performed with GETPROP, storing with PUTPROP. As with ACCESSFNS, CREATE is not initially defined for ATOMRECORD records.

DATATYPE

[Record Type]

Specifies that a new user data type with type name RECORD-NAME be allocated via DECLAREDATATYPE (page 3.14). Unlike other record-types, the records of a DATATYPE declaration are represented with a completely new Interlisp type, and not in terms of other existing types.

FIELDS is a list of field specifications, where each specification is either a list (FIELDNAME . FIELDTYPE), or an atom FIELDNAME. If FIELDTYPE is omitted, it defaults to POINTER. Options for FIELDTYPE are:

- POINTER Field contains a pointer to any arbitrary Interlisp object.
- BITS N Field contains an N-bit unsigned integer.
- BETWEEN N₁ N₂ A generalization of BITS. Field may contain an integer x, such that x is greater than or equal to N₁ and less than or equal to N₂. Enough bits are allocated to store a number between 0 and 2^{N₂-N₁}; N₁ is appropriately added or subtracted when the field is accessed or stored into.
- INTEGER or FIXP Field contains a full word signed integer (the size is implementation- dependent).
- FLOATING or FLOATP Field contains a full word floating point number.
- FLAG Field is a one bit field that "contains" T or NIL.

For example, the declaration

```
(DATATYPE FOO
  ((FLG BITS 12)
   TEXT
   (CNT BETWEEN 10 25)
   HEAD
   (DATE BITS 18)
   (PRIO FLOATP)
   (READ? FLAG)))
```

would define a data type FOO which occupies (in Interlisp-10) three words of storage with two pointer fields (one word), a full word floating point number, fields for an 18, 12, and 4 bit unsigned integer, and a flag (one bit), with 1 bit left over. Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. To store this information in a conventional RECORD list structure, e.g., (RECORD MSG (FLG TEXT CNT DATE PRIO . HEAD)), would take 5 words of list space and up to three number boxes (for FLG, DATE, and PRIO).

Since the user data type must be set up at run-time, the RECORDS package command will dump a DECLAREDATATYPE expression as well as the DATATYPE

Record Declarations

declaration itself. The INITRECORDS le package command (page 11.25) will dump only the DECLAREDATATYPE expression.

Note: DATATYPE declarations should be used with caution within local declarations, since a new and different data type is allocated for each one with a different name.

ARRAYBLOCK

[Record Type]

(Not implemented in Interlisp-D) Similar to a DATATYPE declaration, except that the objects it creates and manipulates are arrays. As with DATATYPE's, the actual order of the elds of the ARRAYBLOCK may be shuffled around in order to satisfy garbage collector constraints.

For example,

```
(ARRAYBLOCK FOO
  ((F1 INTEGER)
   (F2 FLOATING)
   (F3 POINTER)
   (F4 BETWEEN -30 -2)
   (F5 BITS 12)
   (F6 FLAG)))
```

ACCESSFNS

[Record Type]

FIELDS is a list of elements of the form (FIELD-NAME ACCESSDEF SETDEF), i.e. for each eldname, the user species how it is to be accessed and set. ACCESSDEF should be a function of one argument, the datum, and will be used for accessing. SETDEF should be a function of two arguments, the datum and the new value, and will be used for storing. SETDEF may be omitted, in which case, no storing operations are allowed. ACCESSDEF and/or SETDEF may also be a LAMBDA expression or a form written in terms of variables DATUM and (in SETDEF) NEWVALUE. For example, given the declaration

```
[ACCESSFNS ((FIRSTCHAR (NTHCHAR DATUM 1)
                      (RPLSTRING DATUM 1 NEWVALUE))
            (RESTCHARS (SUBSTRING DATUM 2])
```

(replace FIRSTCHAR of X with Y) would translate to (RPLSTRING X 1 Y). Since no SETDEF is given for the RESTCHARS eld, attempting to perform (replace RESTCHARS of X with Y) would generate an error, REPLACE UNDEFINED FOR FIELD. Note that ACCESSFNS do not have a CREATE definition. However, the user may supply one in the defaults and/or subdeclarations of the declaration, as described below. Attempting to CREATE an ACCESSFNS record without specifying a create definition will cause an error CREATE NOT DEFINED FOR THIS RECORD.

ACCESSDEF and SETDEF can also be a property list which specify FAST, STANDARD and UNDOABLE versions of the ACCESSFNS forms, e.g.

```
[ACCESSFNS LITATOM ((DEF (STANDARD GETD FAST FGETD)
                        (STANDARD PUTD UNDOABLE /PUTD])
```

means if FAST declaration is in effect, use FGETD for fetching, if UNDOABLE, use

THE RECORD PACKAGE

/PUTD for saving.

The ACCESSFNS facility allows the use of data-structures not specified by one of the built-in record types. For example, one possible representation of a data-structure is to store the elds in *parallel* arrays, especially if the number of instances required is known, and they do not need to be garbage collected. Thus, to implement a data structure called LINK with two elds FROM and TO, one would have two arrays FROMARRAY and TOARRAY. The representation of an “instance” of the record would be an integer which is used to index into the arrays. This can be accomplished with the declaration:

```
[ACCESSFNS LINK
  ((FROM (ELT FROMARRAY DATUM)
    (SETA FROMARRAY DATUM NEWVALUE)))
  (TO (ELT TOARRAY DATUM)
    (SETA TOARRAY DATUM NEWVALUE)))
  (CREATE (PROG1 (SETQ LINKCNT (ADD1 LINKCNT))
    (SETA FROMARRAY LINKCNT FROM)
    (SETA TOARRAY LINKCNT TO)))
  (INIT (PROGN (SETQ FROMARRAY (ARRAY 100))
    (SETQ TOARRAY (ARRAY 100))))]
```

To CREATE a new LINK, a counter is incremented and the new elements stored (although the CREATE form given the declaration should actually include a test for over ow).

RECORD- TAIL is optional. It may contain expressions of the form:

FIELD-NAME _ FORM

Allows the user to specify within the record declaration the default value to be stored in FIELD-NAME by a CREATE (if no value is given within the CREATE expression itself). Note that FORM is evaluated at CREATE time, not when the declaration is made.

(CREATE FORM) Defines the manner in which CREATE of this record should be performed. This provides a way of specifying how ACCESSFNS should be created or overriding the usual definition of CREATE. If FORM contains the eld- names of the declaration as variables, the forms given in the CREATE operation will be substituted in. If the word DATUM appears in the create form, the *original* CREATE definition is inserted. This effectively allows the user to “advise” the create.

Note: (CREATE FORM) may also be specified as ‘RECORD- NAME _ FORM ’, e.g. C _ (CONS A D).

(INIT FORM) Specifies that FORM should be evaluated when the record is declared. FORM will also be dumped by the INITRECORDS le package command (page 11.25).

For example, see the example of an ACCESSFNS record declaration above. In this example, FROMARRAY and TOARRAY are initialized with an INIT form.

(TYPE? FORM) Defines the manner in which TYPE? expressions are to be translated. FORM may either be an expression in terms of DATUM or a function of one argument.

Note: (TYPE? FORM) may also be specified as ‘RECORD- NAME @ FORM ’, e.g. C @ LISTP.

Defining New Record Types

(SUBRECORD NAME . DEFAULTS)

NAME must be a field that appears in the current declaration and the name of another record. This says that, for the purposes of translating CREATE expressions, substitute the top-level declaration of NAME for the SUBRECORD form, adding on any defaults specified.

For example: Given (RECORD B (E F G)), (RECORD A (B C D) (SUBRECORD B)) would be treated like (RECORD A (B C D) (RECORD B (E F G))) for the purposes of translating CREATE expressions.

a subdeclaration (i.e., a record declaration.)

The RECORD- NAME of a subdeclaration must be either the RECORD- NAME of its immediately superior declaration or one of the superior's field-names. Instead of identifying the declaration as with top level declarations, the record-name of a subdeclaration identifies the parent field or record that is being described by the subdeclaration. Subdeclarations can be nested to an arbitrary depth.

Giving a subdeclaration (RECORD NAME₁ NAME₂) is a simple way of defining a *synonym* for the field NAME₁.

Note that, in a few cases, it makes sense for a given field to have more than one subdeclaration. For example, in

```
(RECORD (A . B) (PROPRECORD B (FOO FIE FUM)) (HASHLINK B C))
```

B is elaborated by both a PROPRECORD and a HASHLINK. Similarly,

```
(RECORD (A B) (RECORD A (C D)) (RECORD A (FOO FIE)))
```

is also acceptable, and essentially “overlays” (FOO FIE) and (C D), i.e. (fetch FOO of X) and (fetch C of X) would be equivalent. In such cases, the *rst* subdeclaration is the one used by CREATE.

3.6 DEFINING NEW RECORD TYPES

In addition to the built-in record types, users can declare their own record types by performing the following steps:

- (1) Add the new record-type to the value of CLISPRECORDTYPES;.
- (2) Perform (MOVD 'RECORD RECORD- TYPE), i.e. give the record-type the same definition as that of the function RECORD;
- (3) Put the name of a function which will return the translation on the property list of RECORD- TYPE, as the value of the property USERRECORDTYPE. Whenever a record declaration of type RECORD- TYPE is encountered, this function will be passed the record declaration as its argument, and should return a *new* record declaration which the record package will then use in its place.

THE RECORD PACKAGE

3.7 RECORD MANIPULATION FUNCTIONS

The user may edit (or delete) global record declarations with the function:

```
(EDITREC NAME COM1 COMN) [NLambda NoSpread Function]
```

Nospread nlambda function similar to EDITF or EDITV. EDITREC calls the editor on a copy of all declarations in which NAME is the record-name or a eld name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If NAME is NIL, *all* declarations are edited.

COM₁ COM_N are (optional) edit commands.

When the user redeclares a global record, the translations of all expressions involving that record or any of its elds are automatically deleted from CLISPARRAY, and thus will be recomputed using the new information. If the user changes a *local* record declaration, or changes some other CLISP declaration, e.g., STANDARD to FAST, and wishes the new information to affect record expressions already translated, he must make sure the corresponding translations are removed, usually either by CLISPIFYing or applying the !DW edit macro.

```
(RECLOOK RECORDNAME _ _ _ _ ) [Function]
```

Returns the entire declaration for the record named RECORDNAME ; NIL if no record declaration with name RECORDNAME . Note that the record package maintains internal state about current record declarations, so performing destructive operations (e.g. NCONC) on the value of RECLOOK may leave the record package in an inconsistent state. To change a record declaration, use EDITREC.

```
(FIELDLOOK FIELDNAME ) [Function]
```

Returns the list of declarations in which FIELDNAME is the name of a eld.

```
(RECORDFIELDNAMES RECORDNAME ) [Function]
```

Returns the list of elds declared in record RECORDNAME . RECORDNAME may either be a name or an entire declaration.

```
(RECORDACCESS FIELD DATUM DEC TYPE NEWV ALUE ) [Function]
```

TYPE is one of FETCH, REPLACE, FFETCH, FREPLACE, /REPLACE or their lowercase equivalents. TYPE = NIL means FETCH. If TYPE corresponds to a fetch operation, i.e. is FETCH, or FFETCH, RECORDACCESS performs (TYPE FIELD OF DATUM). If TYPE corresponds to a replace, RECORDACCESS performs (TYPE FIELD OF DATUM WITH NEWV ALUE). DEC is an optional declaration; if given, FIELD is interpreted as a eld name of that declaration.

Note that RECORDACCESS is relatively inefficient, although it is better than constructing the equivalent form and performing an EVAL.

3.8 CHANGETRAN

A very common programming construction consists of assigning a new value to some datum that is a function of the current value of that datum. Some examples of such read-modify-write sequences include:

Changetrans

<code>(SETQ X (IPLUS X 1))</code>	Incrementing a counter
<code>(SETQ X (CONS Y X))</code>	Pushing an item on the front of a list
<code>(PROG1 (CAR X) (SETQ X (CDR X)))</code>	Popping an item off a list

It is easier to express such computations when the datum in question is a simple variable as above than when it is an element of some larger data structure. For example, if the datum to be modified was `(CAR X)`, the above examples would be:

```
(CAR (RPLACA X (IPLUS (CAR X) 1)))
(CAR (RPLACA X (CONS Y (CAR X))))
(PROG1 (CAR X) (RPLACA X (CDR X)))
```

and if the datum was an element in an array, `(ELT A N)`, the examples would be:

```
(SETA A N (IPLUS (ELT A N) 1))
(SETA A N (CONS Y (ELT A N)))
(PROG1 (CAR (ELT A N)) (SETA A N (CDR (ELT A N))))
```

The difficulty in expressing (and reading) modification idioms is in part due to the well-known asymmetry of setting versus accessing operations on structures: `RPLACA` is used to smash what `CAR` would return, `SETA` corresponds to `ELT`, and so on.

The “Changetrans” facility is designed to provide a more satisfactory notation in which to express certain common (but user-extensible) structure modification operations. Changetrans defines a set of CLISP words that encode the kind of modification that is to take place, e.g. pushing on a list, adding to a number, etc. More important, the expression that indicates the datum whose value is to be modified needs to be stated only once. Thus, the “change word” `ADD` is used to increase the value of a datum by the sum of a set of numbers. Its arguments are an expression denoting the datum, and a set of items to be added to its current value. The datum expression must be a variable or an accessing expression (involving `fetch`, `CAR`, `LAST`, `ELT`, etc) that can be translated to the appropriate setting expression.

For example, `(ADD (CADDR X) (FOO))` is equivalent to:

```
(CAR (RPLACA (CDDR X)
              (PLUS (FOO) (CADDR X))))
```

If the datum expression is a complicated form involving subsidiary function calls, such as `(ELT (FOO X) (FIE Y))`, Changetrans goes to some lengths to make sure that those subsidiary functions are evaluated only once (it binds local variables to save the results), even though they logically appear in both the setting and accessing parts of the translation. Thus, in thinking about both efficiency and possible side effects, the user can rely on the fact that the forms will be evaluated only as often as they appear in the expression.

For `ADD` and all other changewords, the lower-case version (`add`, etc.) may also be specified. Like other CLISP words, change words are translated using all CLISP declarations in effect (see page 16.9).

The following is a list of those change words recognized by Changetrans. Except for `POP`, the value of all

THE RECORD PACKAGE

built-in changeword forms is defined to be the new value of the datum.

(ADD DATUM ITEM₁ ITEM₂) [Change Word]
Adds the specified items to the current value of the datum, stores the result back in the datum location. The translation will use IPLUS, PLUS, or FPLUS according to the CLISP declarations in effect.

(PUSH DATUM ITEM₁ ITEM₂) [Change Word]
CONSES the items onto the front of the current value of the datum, and stores the result back in the datum location. For example, (PUSH X A B) would translate as (SETQ X (CONS A (CONS B X))).

(PUSHNEW DATUM ITEM) [Change Word]
Like PUSH (with only one item) except that the item is not added if it is already FMEMB of the datum's value.

Note that, whereas (CAR (PUSH X 'FOO)) will always be FOO, (CAR (PUSHNEW X 'FOO)) might be something else if FOO already existed in the middle of the list.

(PUSHLIST DATUM ITEM₁ ITEM₂) [Change Word]
Similar to PUSH, except that the items are APPENDED in front of the current value of the datum. For example, (PUSHLIST X A B) would translate as (SETQ X (APPEND A B X)).

(POP DATUM) [Change Word]
Returns CAR of the current value of the datum after storing its CDR into the datum. The current value is computed only once even though it is referenced twice. Note that this is the only built-in changeword for which the value of the form is not the new value of the datum.

(SWAP DATUM₁ DATUM₂) [Change Word]
Sets DATUM₁ to DATUM₂ and vice versa.

(CHANGE DATUM FORM) [Change Word]
This is the most flexible of all change words, since it enables the user to provide an arbitrary form describing what the new value should be, but it still highlights the fact that structure modification is to occur, and still enables the datum expression to appear only once. CHANGE sets DATUM to the value of FORM*, where FORM* is constructed from FORM by substituting the datum expression for every occurrence of the literal DATUM. For example, (CHANGE (CAR X) (ITIMES DATUM 5)) translates as (CAR (RPLACA X (ITIMES (CAR X) 5))).

CHANGE is useful for expressing modifications that are not built-in and are not sufficiently common to justify defining a user-changeword. As for other changeword expressions, the user need not repeat the datum-expression and need not worry about multiple evaluation of the accessing form.

It is possible for the user to define new change words. To define a change word, say sub, that subtracts items from the current value of the datum, the user must put the property CLISPWORD, value (CHANGETRAN . sub) on both the upper and lower-case versions of sub:

User Defined Data Types

```
(PUTPROP 'SUB 'CLISPPWORD '(CHANGETRAN . sub))
(PUTPROP 'sub 'CLISPPWORD '(CHANGETRAN . sub))
```

Then, the user must put (on the *lower-case* version of *sub* only) the property *CHANGEWORD*, with value *FN*. *FN* is a function that will be applied to a single argument, the whole *sub* form, and must return a form that *Changetran* can translate into an appropriate expression. This form should be a list structure with the atom *DATUM* used whenever the user wants an accessing expression for the current value of the datum to appear. The form (*DATUM_* *FORM*) (note that *DATUM_* is a single atom) should occur once in the expression; this species that an appropriate storing expression into the datum should occur at that point. For example, *sub* could be defined with:

```
(PUTPROP 'sub 'CHANGEWORD
  '(LAMBDA (FORM)
    (LIST 'DATUM_
      (LIST 'IDIFFERENCE
        'DATUM
        (CONS 'IPLUS (CDDR FORM)))))))
```

If the expression (*sub* (*CAR* *X*) *A* *B*) were encountered, the arguments to *SUB* would first be dwimied, and then the *CHANGEWORD* function would be passed the list (*sub* (*CAR* *X*) *A* *B*), and return (*DATUM_* (*IDIFFERENCE* *DATUM* (*IPLUS* *A* *B*))), which *Changetran* would convert to (*CAR* (*RPLACA* *X* (*IDIFFERENCE* (*CAR* *X*) (*IPLUS* *A* *B*)))).

Note: The *sub* *changeword* as defined above will always use *IDIFFERENCE* and *IPLUS*; *add* uses the correct addition operation depending on the current *CLISP* declarations.

3.9 USER DEFINED DATA TYPES

Note: The most convenient way to define new user data types is via *DATATYPE* record declarations (see page 3.7).

In addition to built-in data-types such as atoms, lists, arrays, etc., *Interlisp* provides a way of defining completely *new* classes of objects, with a fixed number of fields determined by the definition of the data type. Facilities are provided for declaring the name and *type* of the fields for a given class, creating objects of a given class, accessing and replacing the contents of each of the fields of such an object, as well as interrogating such objects.

In order to define a new class of objects, the user must supply a name for the new data type and specifications for each of its fields. Each field may contain either a pointer (i.e., any arbitrary *Interlisp* datum), an integer, a floating point number, or an *N*-bit integer. This is done via the function *DECLAREDATATYPE*:

```
(DECLAREDATATYPE TYPENAME FIELDSPECS ) [Function]
TYPENAME is a literal atom, which specifies the name of the data type. FIELDSPECS
is a list of "field specifications". Each field specification may be one of the following:
```

POINTER	Field may contain any <i>Interlisp</i> datum.
FIXP	Field contains an integer.

THE RECORD PACKAGE

FLOATP	Field contains a floating point number.
--------	-----------------------------------------

(BITS N) Field contains a non-negative integer less than 2^N .

`DECLAREDATATYPE` returns a list of “eld descriptors”, one for each element of `FIELDSPECS`. A eld descriptor contains information about where within the datum the eld is actually stored.

If `TYPENAME` is already declared a datatype, it is re-declared. If `FIELDSPECS` is `NIL`, `TYPENAME` is “undeclared”.

```
(FETCHFIELD  DESCRIPTOR  DATUM  )
```

[Function]

Returns the contents of the eld described by `DESCRIPTOR` from `DATUM`. `DESCRIPTOR` must be a “eld descriptor” as returned by `DECLAREDATATYPE`. If `DATUM` is not an instance of the datatype of which `DESCRIPTOR` is a descriptor, causes error `DATUM OF INCORRECT TYPE`.

In Interlisp-10, if `DESCRIPTOR` is quoted, `FETCHFIELD` compiles open. This capability is used by the record package.

```
(REPLACEFIELD  DESCRIPTOR  DATUM  NEWV AL UE )
```

[Function]

Store `NEWV ALUE` into the `eld` of `DATUM` described by `DESCRIPTOR`. `DESCRIPTOR` must be a `eld descriptor` as returned by `DECLAREDATATYPE`. If `DATUM` is not an instance of the datatype of which `DESCRIPTOR` is a descriptor, causes error `DATUM OF INCORRECT TYPE`. Value is `NEWV ALUE`.

(NCREATE TYPENAME FROM) [Function]

Creates and returns a new instance of datatype `TYPENAME` .

If FROM is also a datum of datatype `TYPENAME`, the elds of the new object are initialized to the values of the corresponding elds in FROM.

NCREATE will not work for built-in datatypes, such as ARRAYP, STRINGP, etc. If `TYPENAME` is not the type name of a previously declared *user* data type, generates an error, ILLEGAL DATA TYPE.

(GETFIELDSPECS TYPENAME) [Function]

Returns a list which is `EQUAL` to the `FIELDSPECS` argument given to `DECLAREDATATYPE` for `TYPENAME` ; if `TYPENAME` is not a currently declared data-type, returns `NIL`.

```
(GETDESCRIPTORS  TYPENAME  )
```

[Function]

Returns a list of `eld` descriptors, `EQUAL` to the `value` of `DECLAREDATATYPE` for `TYPENAME` .

(USERDATATYPES) [Function]

Returns list of names of currently declared user data types.

Note that the user can define how user data types are to be printed via `DEFPRINT` (page 6.23), how they are to be evaluated by the interpreter via `DEFEVAL` (page 5.11), and how they are to be compiled by the compiler via `COMPILETYPELST` (page 12.9).

The DATATYPE facility in Interlisp-D is an extension of that found in Interlisp-10. Interlisp-D also accepts BYTE, WORD, and SIGNEDWORD as datatype eld descriptors equivalent to BITS 8, BITS 16,

User Defined Data Types

and BETWEEN -2^{15} and $2^{15}-1$ respectively. Interlisp-D will not move elds around in a user declaration if they pack into words and pointers as speci ed. POINTER elds take 24 bits and must be 32-bit right-justi ed.