8. 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 8.6). The record declaration is a description of the record, associating names with its various parts, or "fields". For example, the record declaration **(RECORD MSG (FROM TO . TEXT))** describes a data structure called **MSG**, which contains three fields: **FROM**, **TO**, and **TEXT**. The user can reference these fields by name, to retrieve their values or to store new values into them, by using the **FETCH** and **REPLACE** operators (page 8.2). The **CREATE** operator (page 8.3) is used for creating new instances of a record, and **TYPE?** (page 8.5) 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 fields 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 8.20) 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

field 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 21.12).

The file package includes a **RECORDS** file package command for dumping record declarations (page 17.38), and **FILES?** and **CLEANUP** will inform the user about records that need to be dumped.

8.1 FETCH and REPLACE

The fields 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** field of **X**, and **(replace FROM of X with Y)** will replace this field with the value of **Y**. In general, the value of a **REPLACE** operation is the same as the value stored into the field.

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

An exception to this rule is that "user" record declarations take precedence over "system" record declarations, in cases where an unqualified field name would be considered ambiguous. System records are declared by including **(SYSTEM)** in the record declaration (see page 8.15). All of the records defined in the standard Interlisp-D system are defined as system records.

Another complication can occur if the fields of a record are themselves records. The fields of a record can be further broken down into sub-fields by a "subdeclaration" within the record declaration (see page 8.14). For example,

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

permits the user to access the **POSITION** field with (fetch **POSITION** of X), or its subfield **XLOC** with (fetch **XLOC** of X).

The user may also elaborate a field by declaring that field name in a *separate* record declaration (as opposed to an embedded subdeclaration). For instance, the **TEXT** field in the **MSG** and **REPLY** records above may be subdivided with the seperate record declaration **(RECORD TEXT (HEADER . TXT))**. Fields of subfields (to any level of nested subfields) are accessed by specifying the "data path" as a list of record/field 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** field should be accessed, and *its* **HEADER** field 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 (page 21.13) 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 field 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 (see page 21.12). **FFETCH** and **FREPLACE** are versions which insure fast CLISP declarations will be in effect, /**REPLACE** insures undoable declarations.

8.2 CREATE

Record operations can be applied to arbitrary structures, i.e., the user can explicitely creating a data structure (using **CONS**, etc), and then manipulate it with **FETCH** and **REPLACE**. However, to be consistant 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 fields 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 fields not explicitly given a value, the value of the corresponding field 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 *structure* 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 fields re-used. For example, **REUSING** a **PROPRECORD** just **CONS**es the new property names and values onto the list, while **USING** copies the top level of the list. Another example of this distinction occurs when a field is elaborated by a subdeclaration (page 8.14): **USING** will create a new instance of the sub-record, while **REUSING** will use the old contents of the field (unless some field of the subdeclaration is assigned in the **CREATE** expression.)

If the value of a field 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** fields in **DATATYPE** records, N_1 is used; for other non-pointer fields 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))

(create A B←T copying X))
```

```
    = = > [LIST T (COPYALL (CADR X)) (COPYALL (CADDR X]
(create A B←T reusing X)
    = = > (CONS T (CDR X))
```

8.3 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 **TYPERECORD**. For **DATATYPE**s, 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 **TYPERECORD**s, the **TYPE?** expression will check that the value of *FORM* is a list beginning with *RECORD-NAME*. For **ARRAYRECORD**s, it checks that the value is an array of the correct size. For **PROPRECORD**s and **ASSOCRECORD**s, a **TYPE?** expression will make sure that the value of *FORM* is a property/association list with property names among the field-names of the declaration.

There is no built-in type test for records of type **ACCESSFNS**, **HASHLINK** or **RECORD**. Type tests can be defined for these kinds of records, or redefined for the other kinds, by including an expression of the form **(TYPE?** *COM*) in the record declaration (see page 8.14). Attempting to execute a **TYPE?** expression for a record that has no type test causes an error, **TYPE? NOT IMPLEMENTED FOR THIS RECORD**.

8.4 WITH

Often one wants to write a complex expression that manipulates several fields of a single record. The **WITH** construct can make it easier to write such expressions by allowing one to refer to the fields 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 field-names of *RECORD-NAME*

are implemented via **FETCH** and **SETQ**s of those variables are implemented via **REPLACE**.

For example, given

(RECORD RECN (FLD1 FLD2))
(SETQ INST (create RECN FLD1 ← 10 FLD2 ← 20))

Then the construct

(with RECN INST (SETQ FLD2 (PLUS FLD1 FLD2]

is equivalent to

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

Warning: **WITH** is implemented by doing simple substitutions in the body of the forms, without regard for how the record fields are used. This means, for example, if the record **FOO** is defined by **(RECORD FOO (POINTER1 POINTER2))**, then the form

(with FOO X (SELECTQ Y (POINTER1 POINTER1) NIL]

will be translated as

(SELECTQ Y ((CAR X) (CAR X)) NIL]

The user should be careful that record field names are not used except as variables in the **WITH** forms.

8.5 Record Declarations

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

(*RECORD-TYPE RECORD-NAME RECORD-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. The different record types are described below.

RECORD-NAME is à litatom used to identify the record declaration for creating instances of the record via **CREATE**, testing via **TYPE?**, and dumping to files via the **RECORDS** file package command (page 17.38). **DATATYPE** and **TYPERECORD** declarations also use *RECORD-NAME* to identify the data structure (as described below).

RECORD-FIELDS describes the structure of the record. Its exact interpretation varies with *RECORD-TYPE*. For most record types it defines the names of the fields within the record that can be accessed with **FETCH** and **REPLACE**.

RECORD-TAIL is an optional list that can be used to specify default values for record fields, special **CREATE** and **TYPE?** forms, and subdeclarations (described below).

Normally, record declaration forms are typed in to the top-level executive or read from a file, and they define the structure of the record globally. Local record declarations within the context of a function are defined by including a record declaration form in the CLISP declaration for the function, rather than evaluating the expression itself (see page 21.13).

Note: Although record declarations are evaluatable forms, and thus can be included in functions, changing a record declaration dynamically (at run-time) is not recommended. When a **FETCH** or **REPLACE** operation is interpreted, and the record declaration has changed, the form has to be re-translated. If a function containing **FETCH** or **REPLACE** operations has been compiled, it may be necessary to re-compile. For applications which need to change record declarations dynamically, users should consider using more flexible data structures, such as association lists or property lists.

8.5.1 Record Types

Records can be used to describe a large variety of data objects, that are manipulated in different ways. The *RECORD-TYPE* field of the record declaration specifies how the data object is created, and how the various record fields are accessed. Depending on the record type, the record fields may be stored in a list, or in an array, or on the property list of a litatom. The following record types are defined:

RECORD [Record Type]

The **RECORD** record type is used to describe list structures. *RECORD-FIELDS* is interpreted as a list structure whose non-**NIL** literal atoms are taken as field-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 field, 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 **NIL**s, e.g. **(A 4 B)** is interpreted as **(A NIL NIL NIL NIL B)**.

TYPERECORD [Record Type]

The **TYPERECORD** record type is similar to **RECORD**, except that the record name is added to the front of the list structure to signify what "type" of record it is. This type field is used by the

record package in the translation of **TYPE?** expressions. **CREATE** will insert an extra field containing *RECORD-NAME* at the beginning of the structure, and the translation of the access and storage functions will take this extra field into account. For example, for **(TYPERECORD MSG (FROM TO . TEXT))**, (fetch **FROM** of **X**) translates as **(CADR X)**, not **(CAR X)**.

ASSOCRECORD [Record Type]

The **ASSOCRECORD** record type is used to describe list structures where the fields are stored in association list format:

((*FIELDNAME*$_1$. *VALUE*$_1$) (*FIELDNAME*$_2$. *VALUE*$_2$) ...)

RECORD-FIELDS is a list of literal atoms, interpreted as the permissable list of field names in the association list. Accessing is performed with **ASSOC** (or **FASSOC**, depending on current **CLISP** declarations, see page 21.12), storing with **PUTASSOC**.

PROPRECORD [Record Type]

The **PROPRECORD** record type is used to describe list structures where the fields are stored in property list format:

(*FIELDNAME*$_1$ *VALUE*$_1$ *FIELDNAME*$_2$ *VALUE*$_2$ ...)

RECORD-FIELDS is a list of literal atoms, interpreted as the permissable list of field names in the property list. Accessing is performed with **LISTGET**, storing with **LISTPUT**.

Both **ASSOCRECORD** and **PROPRECORD** are useful for defining data structures in which it is often the case that many of the fields are **NIL**. A **CREATE** expression for these record types only stores those fields which are non-**NIL**. Note, however, that with the record declaration **(PROPRECORD 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**.

ARRAYRECORD [Record Type]

The **ARRAYRECORD** record type is used to describe arrays. *RECORD-FIELDS* is interpreted as a list of field names that are associated with the corresponding elements of an array. **NIL** can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of **NIL**s. For example, **(ARRAYRECORD (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 **ARRAYRECORD** only creates arrays of pointers. Other kinds of arrays must be implemented by the user with the **ACCESSFNS** record type (page 8.12).

HASHLINK [Record Type]

The **HASHLINK** record type can be used with any type of data object: it specifies that the value of a single field can be accessed by hashing the data object in a given hash array. Since the **HASHLINK** record type describes an accessing method, rather than a data structure, the **CREATE** expression is meaningless for **HASHLINK** records.

RECORD-FIELDS is either an atom *FIELD-NAME*, or a list (*FIELD-NAME HARRAYNAME HARRAYSIZE*). *HARRAYNAME* is a variable whose value is the hash array to be used; if not given, **SYSHASHARRAY** is used. If the value of the variable *HARRAYNAME* is not a hash array (at the time of the record declaration), it will be set to a new hash array with a size of *HARRAYSIZE*. *HARRAYSIZE* defaults to 100.

The **HASHLINK** record type is useful as a subdeclaration to other records to add additional fields to already existing data structures (see page 8.14). For example, suppose that **FOO** is a record declared with **(RECORD FOO (A B C))**. To add an aditional field **BAR**, without modifying the already existing data strutures, redeclare **FOO** with:

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

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

ATOMRECORD [Record Type]

The **ATOMRECORD** record type is used to describe property lists of litatoms. *RECORD-FIELDS* is a list of property names. Accessing is performed with **GETPROP**, storing with **PUTPROP**. The **CREATE** expression is not initially defined for **ATOMRECORD** records.

DATATYPE [Record Type]

The **DATATYPE** record type is used to define a new user data type with type name *RECORD-NAME* (by calling **DECLAREDATATYPE**, page 8.21). 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.

RECORD-FIELDS is interpreted as 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**. Possible values for *FIELDTYPE* are:

POINTER Field contains a pointer to any arbitrary Interlisp object.

INTEGER
FIXP Field contains a signed integer. Note that an **INTEGER** field is not capable of holding everything that satisfies **FIXP**, such as bignums (page 7.1).

FLOATING
FLOATP Field contains a floating point number.

SIGNEDWORD Field contains a 16-bit signed integer.

FLAG Field is a one bit field that "contains" **T** or **NIL**.

BITS *N* Field contains an *N*-bit unsigned integer.

BYTE Equivalent to **BITS 8**.

WORD Equivalent to **BITS 16**.

XPOINTER Field contains a pointer like **POINTER**, except that the field is *not* reference counted by the Interlisp-D garbage collector. **XPOINTER** fields are useful for implementing back-pointers in structures that would be circular and not otherwise collected by the reference-counting garbage collector.

Warning: **XPOINTER** fields should be used with great care. It is possible to damage the integrity of the storage allocation system by using pointers to objects that have been garbage collected. Code that uses **XPOINTER** fields should be sure that the objects pointed to have not been garbage collected. This can be done in two ways: The first is to maintain the object in a global structure, so that it is never garbage collected until explicitly deleted from the structure, at which point the program must invalidate all the **XPOINTER** fields of other objects pointing at it. The second is to declare the object as a **DATATYPE** beginning with a **POINTER** field that the program maintains as a pointer to an object of another type (e.g., the object containing the **XPOINTER** pointing back at it), and test that field for reasonableness whenever using the contents of the **XPOINTER** field.

For example, the declaration

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

would define a data type **FOO** with two pointer fields, a floating point number, and fields for a 12 and 18 bit unsigned integers, and a flag (one bit). Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. Generally, a **DATATYPE** record is much more storage

compact than the corresponding **RECORD** structure would be; in addition, access is faster.

Since the user data type must be set up at *run*-time, the **RECORDS** file package command will dump a **DECLAREDATATYPE** expression as well as the **DATATYPE** declaration itself. If the record declaration is otherwise not needed at runtime, it can be kept out of the compiled file by using a **(DECLARE: DONTCOPY --)** expression (see page 17.40), but it is still necessary to ensure that the datatype is properly initialized. For this, one can use the **INITRECORDS** file package command (page 17.38), which will dump only the **DECLAREDATATYPE** expression.

Note: When defining a new data type, it is sometimes useful to call the function **DEFPRINT** (page 25.16) to specify how instances of the new data type should be printed. This can be specified in the record declaration by including an **INIT** record specification (page 8.14), e.g. **(DATATYPE QV.TYPE ... (INIT (DEFPRINT 'QV.TYPE (FUNCTION PRINT.QV.TYPE))))**.

Note: **DATATYPE** declarations cannot be used within local record declarations (page 21.13).

BLOCKRECORD [Record Type]

The **BLOCKRECORD** record type is used in low-level system programming to "overlay" an organized structure over an arbitrary piece of "unboxed" storage. *RECORD-FIELDS* is interpreted exactly as with a **DATATYPE** declaration, except that fields are *not* automatically rearranged to maximize storage efficiency. Like an **ACCESSFNS** record, a **BLOCKRECORD** does not have concrete instances; it merely provides a way of interpreting some existing block of storage. Thus, one cannot create an instance of a **BLOCKRECORD** (unless the declaration includes an explicit **CREATE** expression), nor is there a default **type?** expression for a **BLOCKRECORD**.

Warning: The programmer should exercise caution in using **BLOCKRECORD** declarations, as they enable one to write expressions that fetch and store arbitrary data in arbitrary locations, thereby evading the normal type system. Except in very specialized situations, a **BLOCKRECORD** should never contain **POINTER** or **XPOINTER** fields, nor be used to overlay an area of storage that contains pointers. Such use could compromise the garbage collector and storage allocation system. The programmer is responsible for ensuring that all **FETCH** and **REPLACE** expressions are performed only on suitable objects, as no type testing is performed.

A typical use for the **BLOCKRECORD** type in user code is to overlay a non-pointer portion of an existing **DATATYPE**. For this use, the **LOCF** macro is useful. **(LOCF (fetch** *FIELD* **of** *DATUM*)**)**

can be used to refer to the storage that begins at the first word that contains *FIELD* of *DATUM*. For example, to define a new kind of Ethernet packet (page 31.26), one could overlay the "body" portion of the **ETHERPACKET** datatype declaration as follows:

```
(ACCESSFNS MYPACKET
  ((MYBASE (LOCF (fetch (ETHERPACKET EPBODY) of DATUM))))
  (BLOCKRECORD MYBASE
    ((MYTYPE WORD)
     (MYLENGTH WORD)
     (MYSTATUS BYTE)
     (MYERRORCODE BYTE)
     (MYDATA INTEGER)))
  (TYPE? (type? ETHERPACKET DATUM)))
```

With this declaration in effect, the expression **(fetch MYLENGTH of PACKET)** would retrieve the second 16-bit field beyond the offset inside **PACKET** where the **EPBODY** field starts. For more examples, see the EtherRecords library package.

ACCESSFNS [Record Type]

The **ACCESSFNS** record type is used to define data structures with user-defined access functions. For each field name, the user specifies how it is to be accessed and set. This allows the use of the record package with arbitrary data structures, with complex access methods.

RECORD-FIELDS is interpreted as a list of elements of the form (*FIELD-NAME ACCESSDEF SETDEF*). *ACCESSDEF* should be a function of one argument, the datum, and will be used for accessing the value of the field. *SETDEF* should be a function of two arguments, the datum and the new value, and will be used for storing a new value in a field. *SETDEF* may be omitted, in which case, no storing operations are allowed.

ACCESSDEF and/or *SETDEF* may also be a form written in terms of variables **DATUM** and (in *SETDEF*) **NEWVALUE**. For example, given the declaration

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

(replace (FOO FIRSTCHAR) of X with Y) would translate to **(RPLSTRING X 1 Y)**. Since no *SETDEF* is given for the **RESTCHARS** field, attempting to perform **(replace (FOO 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 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 **/PUTD** for saving (see CLISP declarations, page 21.12).

Note: *SETDEF* forms should be written so that they return the new value, to be consistant with **REPLACE** operations for other record types. The **REPLACE** record operator does not enforce this, though.

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 fields 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 fields **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))
         (SETQ LINKCNT 0)]
```

To create a new **LINK**, a counter is incremented and the new elements stored. (Note: The **CREATE** form given the declaration probably should include a test for overflow.)

8.5.2 Optional Record Specifications

After the *RECORD-FIELDS* item in a record declaration expression there can be an arbitrary number of additional expressions in *RECORD-TAIL*. These expressions can be used to specify default values for record fields, special **CREATE** and **TYPE?** forms, and subdeclarations. The following expressions are permitted:

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 field-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*".

(INIT *FORM***)** — Specifies that *FORM* should be evaluated when the record is declared. *FORM* will also be dumped by the **INITRECORDS** file package command (page 17.38).

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.

(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 — If a record declaration expression occurs among the record specifications of another record declaration, it is known as a "subdeclaration." Subdeclarations are used to declare that fields of a record are to be interpreted as another type of record, or that the record data object is to be interpreted in more than one way.

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** $NAME_1$ $NAME_2$) is a simple way of defining a *synonym* for the field $NAME_1$.

It is possible for a given field to have more than one subdeclaration. For example, in

(RECORD FOO (A B) (RECORD A (C D)) (RECORD A (Q R)))

(Q R) and **(C D)** are "overlayed," i.e. **(fetch Q of X)** and **(fetch C of X)** would be equivalent. In such cases, the *first* subdeclaration is the one used by **CREATE**.

(SYNONYM *FIELD* **(**SYN_1 ... SYN_N**))** *FIELD* must be a field that appears in the current declaration. This defines SYN_1 ... SYN_N all as synonyms of *FIELD*. If there is only one synonym, this can be written as **(SYNONYM** *FIELD SYN***)**.

(SYSTEM) If **(SYSTEM)** is included in a record declaration, this indicates that the record is a "system" record rather than a "user" record. The only distinction between the two types of records is that "user" record declarations take precedence over "system" record declarations, in cases where an unqualified field name would be considered ambiguous. All of the records defined in the standard Interlisp-D system are defined as system records.

8.6 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.

8.7 Record Manipulation Functions

(EDITREC *NAME* COM_1 ... COM_N**)** [NLambda NoSpread Function]

EDITREC calls the editor on a copy of all declarations in which *NAME* is the record name or a field name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If *NAME* is **NIL**, *all* declarations are edited.

COM_1 ... 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 fields are automatically deleted from **CLISPARRAY**, and thus will be recomputed using the new information. If the user changes a *local* record declaration (page 21.13), or changes some other CLISP declaration (page 21.12), 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 **CLISPIFY**ing or using the **DW** edit macro.

(RECLOOK *RECNAME* — — — —**)** [Function]

Returns the entire declaration for the record named *RECNAME*; **NIL** if there is no record declaration with name *RECNAME*. 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 field.

(RECORDFIELDNAMES *RECORDNAME* —**)** [Function]

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

(RECORDACCESS *FIELD DATUM DEC TYPE NEWVALUE***)** [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** *NEWVALUE*). *DEC* is an optional declaration; if given, *FIELD* is interpreted as a field name of that declaration.

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

(RECORDACCESSFORM *FIELD DATUM TYPE NEWVALUE*) [Function]

Returns the form that would be compiled as a result of a record access. *TYPE* is one of **FETCH**, **REPLACE**, **FFETCH**, **FREPLACE**, **/REPLACE** or their lowercase equivalents. *TYPE* = **NIL** means **FETCH**.

8.8 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:

Incrementing a counter:

(SETQ X (IPLUS X 1))

Pushing an item on the front of a list:

(SETQ X (CONS Y X))

Popping an item off a list:

(PROG1 (CAR X) (SETQ X (CDR X)))

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 (CAAR X) (RPLACA X (CDAR 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 set what **CAR** would return, **SETA** corresponds to **ELT**, and so on.

The "Changetran" facility is designed to provide a more satisfactory notation in which to express certain common (but

user-extensible) structure modification operations. Changetran 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 (envolving **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))**, Changetran 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 21.12).

The following is a list of those change words recognized by Changetran. Except for **POP**, the value of all built-in changeword forms is defined to be the new value of the datum.

(ADD *DATUM ITEM*$_1$ *ITEM*$_2$ ...**)** [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 (see page 21.12).

(PUSH *DATUM ITEM*$_1$ *ITEM*$_2$ ...**)** [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 ITEM*$_1$ *ITEM*$_2$ ...**)** [Change Word]

Similar to **PUSH**, except that the items are **APPEND**ed 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 *DATUM*$_1$ *DATUM*$_2$**)** [Change Word]

Sets *DATUM*$_1$ to *DATUM*$_2$ 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 litatom **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**:

```
(PUTPROP 'SUB 'CLISPWORD '(CHANGETRAN . sub))
(PUTPROP 'sub 'CLISPWORD '(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 specifies 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 dwimified, 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 (see page 21.12).

8.9 Built-In and User Data Types

Interlisp is a system for the manipulation of various kinds of data; it provides a large set of built-in data types, which may be used to represent a variety of abstract objects, and the user can also define additional "user data types" which can be manipulated exactly like built-in data types.

Each data type in Interlisp has an associated "type name," a litatom. Some of the type names of built-in data types are: **LITATOM**, **LISTP**, **STRINGP**, **ARRAYP**, **STACKP**, **SMALLP**, **FIXP**, and **FLOATP**. For user data types, the type name is specified when the data type is created.

(DATATYPES —) [Function]

Returns a list of all type names currently defined.

(USERDATATYPES) [Function]

Returns list of names of currently declared user data types.

(TYPENAME *DATUM*) [Function]

Returns the type name for the data type of *DATUM*.

(TYPENAMEP *DATUM TYPE***)** [Function]

Returns T if *DATUM* is an object with type name equal to *TYPE*, otherwise **NIL**.

Note: **TYPENAME** and **TYPENAMEP** distinguish the logical data types **ARRAYP**, **CCODEP** and **HARRAYP**, even though they may be implemented as **ARRAYP**s in some Interlisp implementations.

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. 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.

Note: The most convenient way to define new user data types is via **DATATYPE** record declarations (page 8.9) which call the following functions.

(DECLAREDATATYPE *TYPENAME FIELDSPECS* — —**)** [Function]

Defines a new user data type, with the name *TYPENAME*. *FIELDSPECS* is a list of "field specifications." Each field specification may be one of the following:

POINTER	Field may contain any Interlisp datum.
FIXP	Field contains an integer.
FLOATP	Field contains a floating point number.
(BITS *N***)**	Field contains a non-negative integer less than 2^N.
BYTE	Equivalent to **(BITS 8)**.
WORD	Equivalent to **(BITS 16)**.
SIGNEDWORD	Field contains a 16 bit signed integer.

DECLAREDATATYPE returns a list of "field descriptors," one for each element of *FIELDSPECS*. A field descriptor contains information about where within the datum the field is actually stored.

If *FIELDSPECS* is **NIL**, *TYPENAME* is "undeclared." If *TYPENAME* is already declared as a data type, it is undeclared, and then re-declared with the new *FIELDSPECS*. An instance of a data type that has been undeclared has a type name of ****DEALLOC****.

(FETCHFIELD *DESCRIPTOR DATUM***)** [Function]

Returns the contents of the field described by *DESCRIPTOR* from *DATUM*. *DESCRIPTOR* must be a "field descriptor" as returned by **DECLAREDATATYPE** or **GETDESCRIPTORS**. If *DATUM* is not an

instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error **DATUM OF INCORRECT TYPE**.

(REPLACEFIELD *DESCRIPTOR DATUM NEWVALUE***)** [Function]

Store *NEWVALUE* into the field of *DATUM* described by *DESCRIPTOR*. *DESCRIPTOR* must be a field 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 *NEWVALUE*.

(NCREATE *TYPE OLDOBJ***)** [Function]

Creates and returns a new instance of datatype *TYPE*.

If *OLDOBJ* is also a datum of datatype *TYPE*, the fields of the new object are initialized to the values of the corresponding fields in *OLDOBJ*.

NCREATE will not work for built-in datatypes, such as **ARRAYP**, **STRINGP**, etc. If *TYPE* 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 field descriptors, **EQUAL** to the *value* of **DECLAREDATATYPE** for *TYPENAME*. If *TYPENAME* is not an atom, **(TYPENAME** *TYPENAME***)** is used.

Note that the user can define how user data types are to be printed via **DEFPRINT** (page 25.16), how they are to be evaluated by the interpreter via **DEFEVAL** (page 10.13), and how they are to be compiled by the compiler via **COMPILETYPELST** (page 18.11).