Hiding the details of your code makes it more readable, and lets you program more efficiently. Data structures are a good example: You're better off if you can say "Fetch me the SPEED field from this AIRPLANE" rather than having to say (CAR (CDDDR (CADR AIRPLANE))). You can declare data structures used by your programs, then work with field names rather than access details. Using the declarations, Medley performs the access/storage operations you request. If you change a data structure's declaration, your programs automatically adjust.

You describe the format of a data structure (record) by making a "record declaration" (see the Record Declarations section below). 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, that has three fields: FROM, TO, and TEXT. You can refer to these fields by name, to get their values or to store new values into them, by using FETCH and REPLACE:

```
(fetch (MSG FROM)of MYMSG)
(replace (MSG TO) of MYMSG with "John Doe")
```

You create new MSGs with CREATE:

```
(SETQ MYMSG (create MSG))
```

and TYPE? tells you whether some object is a MSG:

```
(IF (TYPE? MSG THIS-THING) then (SEND-MSG THIS-THING))
```

So far we've said nothing about *how* your MSG is represented—when you're writing FETCHes and REPLACEs, it doesn't matter. But you *can* control the representation: The symbol RECORD in the declaration above causes each MSG to be represented as a list. There are a number of options, up to creating a completely new Lisp data type; each has its own specifier symbol, and they're described in detail below.

The record package is implemented using DWIM and CLISP, so it will do spelling correction on field names, record types, etc. Record operations are translated using all CLISP declarations in effect (standard/fast/undoable).

The file manager's RECORDS command lets you give record declarations (see Chapter 17), and FILES? and CLEANUP will tell you about record declarations that need to be dumped.

FETCH and REPLACE

The fields of a record are accessed and changed with FETCH and REPLACE. If 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 (fetch FROM of X) assumes that X is an instance of the record MSG—the interpretation of (fetch FROM of X) never depends on the *value* of X. If X is not a MSG, this may produce incorrect results.

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, give FETCH and REPLACE a list for their "field" argument: (fetch (MSG TEXT) of X) will fetch the TEXT field of a MSG record. 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 ambiguous.

If there's a conflict, "user" record declarations take precedence over "system" record declarations. System records are declared by including (SYSTEM) in the declaration (see the Record Declarations section below). All of the records defined in the standard Medley system are 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. For example,

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

lets you access the POSITION field with (fetch POSITION of X), or its subfield XLOC with (fetch XLOC of X).

You may also declare that field name in a *separate* record declaration. For instance, the TEXT field in the MSG and REPLY records above may be subdivided with the seperate record declaration (RECORD TEXT (HEADER TXT)). You get to fields of subfields (to any level of nesting) 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)
```

treats X as a MSG record, fetches its TEXT field, and fetches *its* HEADER field. You only need to give enough of the data path to disambiguate it. In this case, (fetch (MSG HEADER) of X) is sufficient: Medley searches among all current record declarations for a path from each name to the next, considering first local declarations (see Chapter 21) and then global ones. Of course, if you had two records with HEADER fields, you get an Ambiguous data path error.

FETCH and REPLACE are translated using the CLISP declarations in effect (see Chapter 21). FFETCH and FREPLACE are fast versions that don't do any type checking. /REPLACE insures undoable declarations.

Record Declarations

You define records by evaluating declarations of the form:

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

RECORD-TYPE specifies the "type" of data you're declaring, and controls how instances will be stored internally. The different record types are described below.

RECORD-NAME is a symbol used to identify the record declaration for CREATE, TYPE?, FETCH and REPLACE, and dumping to files (see Chapter 17). 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. Generally, it names the fields within the record that can be accessed with FETCH and REPLACE.

RECORD-TAIL is an optional list where you can specify default values for record fields, special CREATE and TYPE? forms, and subdeclarations (described below).

Record declarations are Lisp programs, and could be included in functions, changing a record declaration at run-time. *Don't do it.* You risk creating a structure with one declaration, and trying to fetch from it with another—complete chaos results. If you need to change record declarations dynamically, consider using association lists or property lists.

Record Types

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 a symbol's property list. The following record types are defined:

RECORD [Record Type]

The fields of a RECORD are kept in a list. *RECORD-FIELDS* is a list; each non-NIL symbol is a field-name to be associated with the corresponding element or tail of a list structure. For example, with the declaration `(RECORD MSG (FROM TO . TEXT))`, `(fetch FROM of X)` translates as `(CAR X)`.

NIL can be used as a place marker for 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 NILs, e.g. `(A 4 B)` is interpreted as `(A NIL NIL NIL NIL B)`.

DATATYPE [Record Type]

Defines a new user data type with type name *RECORD-NAME*. Unlike other record types, the instances of a DATATYPE are represented with a completely new Lisp type, and not in terms of other existing types.

RECORD-FIELDS is a list of field specifications, where each specification is either a list (*FIELDNAME FIELDTYPE*), or an symbol *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. Caution: An INTEGER field is not capable of holding everything that satisfies FIXP, such as bignums.

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, but the field is *not* reference counted by the 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: Use XPOINTER fields with great care. You can 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 Chapter 17), but it is still necessary to ensure that the datatype is properly initialized. For this, one can use the INITRECORDS file package command (see Chapter 17), which will dump only the DECLAREDATATYPE expression.

Note: When defining a new data type, it is sometimes useful to call the function DEFPRINT (see Chapter 25) 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 (see the Optional Record Specifications section below), e.g. (DATATYPE QV.TYPE ... (INIT (DEFPRINT 'QV.TYPE (FUNCTION PRINT.QV.TYPE)))).

DATATYPE declarations cannot be used within local record declarations (see Chapter 21).

TYPERECORD [Record Type]

Similar to RECORD, but 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 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]

Describes lists where the fields are stored in association list format:

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

RECORD-FIELDS is a list of symbols, the permissable field names in the association list. Access is done with ASSOC (or FASSOC, if the current CLISP declarations are FAST, see Chapter 21), storing with PUTASSOC.

PROPRECORD [Record Type]

Describes lists where the fields are stored in property list format:

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

RECORD-FIELDS is a list of symbols, the permissable field names in the property list. Access is done with LISTGET, storing with LISTPUT.

Both ASSOCRECORD and PROPRECORD are useful for defining data structures where many of the fields are NIL. CREATEing one these record types only stores those fields that 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]

ARRAYRECORDs are stored as arrays. *RECORD-FIELDS* is 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 NILs. 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.

ARRAYRECORD only creates arrays of pointers. Other kinds of arrays must be implemented with ACCESSFNS (see below).

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 access method, rather than a data structure, CREATE is meaningless for HASHLINK records.

RECORD-FIELDS is either a symbol *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 the Optional Record Specifications section below). For example, suppose that FOO is a record declared with `(RECORD FOO (A B C))`. To add a new field BAR, without modifying the 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]

ATOMRECORDs are stored on the property lists of symbols. *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.

BLOCKRECORD [Record Type]

BLOCKRECORD is used in low-level system programming to "overlay" an organized structure over an arbitrary piece of raw 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. So you can't 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: Exercise caution in using BLOCKRECORD declarations, as they let you fetch and store arbitrary data in arbitrary locations, thereby evading Medley's 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. You are 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 a BLOCKRECORD 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, you 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 place inside PACKET where the EPBODY field starts.

ACCESSFNS [Record Type]

ACCESSFNS lets you specify arbitrary functions to fetch and store data. For each field name, you specify how it is to be accessed and set. This lets you use arbitrary data structures, with complex access methods. Most often, ACCESSFNS are useful when you can compute one field's value from other fields. If you're representing a time period by its start and duration, you could add an ACCESSFNS definition for the ending time that did the obvious addition.

RECORD-FIELDS is 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, you 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, see Chapter 21).

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

ACCESSFNS let you use 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 needn't be garbage collected. To implement LINK with two fields FROM and TO, you'd have two arrays FROMARRAY and TOARRAY. The representation of an "instance" of LINK would be an integer, 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.)

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 you 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 you to "advise" the create.

(INIT *FORM*) Specifies that *FORM* should be evaluated when the record is declared. *FORM* will also be dumped by the INITRECORDS file package command (see Chapter 17).

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 Medley system are defined as system records.

CREATE

You can create RECORDs by hand if you like, using CONS, LIST, etc. But that defeats the whole point of hiding implementation details. So much easier to use:

(create *RECORD-NAME . ASSIGNMENTS*)

CREATE translates into an appropriate Interlisp form that uses CONS, LIST, PUTHASH, ARRAY, etc., to create the new datum with the its fields initialized to the values you specify. *ASSIGNMENTS* is optional and may contain expressions of the following form:

FIELD-NAME ← *FORM* Specifies initial value for *FIELD-NAME*.

USING *FORM* *FORM* is an existing instance of *RECORD-NAME*. If you don't specify a value for some field, the value of the corresponding field in *FORM* is to be used.

COPYING *FORM* Like USING, but the corresponding values are copied (with COPYALL).

REUSING *FORM* Like USING, but wherever possible, the corresponding *structure* in *FORM* is used.

SMASHING *FORM* A new instance of the record is not created at all; rather, new field values are smashed into *FORM*, which CREATE then returns.

When it makes a difference, Medley goes to great pains to make its translation do things in the same order as the original CREATE expression. 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.

How are USING and REUSING different? (create *RECORD* reusing *FORM* ...) doesn't do any destructive operations on the value of *FORM*, but *will* incorporate as much as possible of the old data structure into the new one. On the other hand, (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 CONSes 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: 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. (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))
```

TYPE?

The record package allows you 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 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 TYPERECORDs, the TYPE? expression will check that the value of *FORM* is a list beginning with *RECORD−NAME*. For ARRAYRECORDs, it checks that the value is an array of the correct size. For PROPRECORDs and ASSOCRECORDs, 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 the Record Declarations section below). Attempting to execute a TYPE? expression for a record that has no type test causes an error, Type? not implemented for this record.

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 SETQs 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]
```

Be careful that record field names are not used except as variables in the WITH forms.

Defining New Record Types

In addition to the built-in record types, you can declare your 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*).
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.

Manipulating Record Declarations

(**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 you redeclare 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 you change a *local* record declaration (see Chapter 21), or change some other CLISP declaration (see Chapter 21), e.g., STANDARD to FAST, and wish the new information to affect record expressions already translated, you must make sure the corresponding translations are removed, usually either by CLISPIFYing 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* —) [Funtion]

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.

Changetran

Often, you'll want to assign a new value to some datum that is a function of its current value:

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)))`

Those are simple when you're working with a variable; it gets complicated when you're working with structured data. For example, if you want to modify (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 you're changing 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))))
```

Changetran is designed to provide a simpler way to express these common (but user-extensible) structure modifications. Changetran defines a set of CLISP words that encode the kind of modification to take place—pushing on a list, adding to a number, etc. More important, you only indicate the item to be modified 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 the datum, and a set of numbers to be added to it. The datum 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 X 1) is equivalent to:

```
(SETQ X (PLUS X 1))
```

and (ADD (CADDR X) (FOO)) is equivalent to:

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

If the datum is a complicated form involving 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, even though they are used in both the setting and accessing parts of the translation. You can rely on the fact that the forms will be evaluated only as often as they appear in your expression.

For ADD and all other changewords, the lowercase version (`add`, etc.) may also be specified. Like other CLISP words, change words are translated using all CLISP declarations in effect (see Chapter 21).

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 Chapter 21).

(**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 APPENDed in front of the current value of the datum. For example, `(PUSHLIST X A B)` translates 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: You give an arbitrary form describing what the new value should be. But it still highlights the fact that structure modification is happening, and still lets the datum 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 symbol 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 common enough to justify defining a user-changeword.

You can define new change words. To define a change word, say sub, that subtracts items from the current value of the datum, you 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, you 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 symbol DATUM used whenever you want an accessing expression for the current value of the datum to appear. The form (DATUM← *FORM*) (note that DATUM← is a single symbol) 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 as:

```
(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 Chapter 21).

Built-In and User Data Types

Medley is a system for manipulating various kinds of data; it comes with a large set of built-in data types, which you can use to represent a variety of abstract objects; you can also define additional "user data types" that you can manipulate exactly like built-in data types.

Each data type in Medley has an associated "type name," a symbol. 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 crated.

(**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.

In addition to built-in data-types like symbols, lists, arrays, etc., Medley provides a way to define completely *new* classes of objects, with a fixed number of fields determined by the definition of the data type. To define a new class of objects, you must supply a name for the new data type and specifications for each of its fields. Each field may contin 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 (see Chapter 8) which call the following funtions.

(**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.

You can define how a user data type prints, using DEFPRINT (see Chapter 25), how they are to be evaluated by the interpreter via DEFEVAL (see Chapter 10), and how they are to be compiled by the compiler via COMPILETYPELST (see Chapter 18).

[This page intentionally left blank]