YTools: A Package of Portable Enhancements to Common Lisp
Version 1.3

Drew McDermott

June 25, 2004

Contents

1 Introduction

I have been using Lisp for a long time, and have built several tools for doing things that Common Lisp doesn't do, or, in my opinion, doesn't do right. This manual describes them.

Each tool is introduced by a line in the following form

`Toolname` *Category* *Location:* where-to-find-it

The *Category* is either *Macro* or *Function*. The *Location* specifies where the tool is to be found. This information is explained in section 9. It tells you how to tell the file manager to load the tool. If you're reading this manual just to get acquainted with what YTools offers, you can ignore the location information.

There are various lexical conventions used in YTools, which are explained in section A.2. One that is pervasive is that underscore ("_") may be used as the name for a function parameter that is to be ignored. I will point out the contexts where this convention is allowed, but basically it is used wherever it would make sense (in the parameters of a `defun`, but not the bound variables of a `let`, for instance).

Another terminological convention is that I avoid reference to `t` and `nil`, especially `nil`. (See "NIL Considered Harmful"
`http://cs-www.cs.yale.edu/homes/dvm/nil.html`.)
Instead I refer wherever possible to "*True*" for the denotation of `t` and "*False*" or "empty list" for the denotation of `nil`.

The YTools package has been developed for many years at Yale, and many people have made a contribution to it, especially Eugene Charniak, Denys Duchier, Jim Firby, Steve Hanks, Jim Meehan, Chris Riesbeck, and Larry Wright.

2 Binding Local Functions (And Iterating While You're At It)

Some people like to put subroutine definitions before the definitions of their callers, some people after. Although Lisp allows either order at the top level, in the `labels` construct all the definitions must come before the body. It can make code much more readable if we allow them to come after, flagged by the keyword `:where`.

2.1 `let-fun`: Improved version of `labels`

`let-fun` *Macro* *Location:* YTools, `binders`

The basic idea is embodied in the `let-fun` macro:

```
(let-fun (---local-defs-1---)
   ---body---
 [:where ---local-defs-2---])
```

Each *local-def* is in the same format as for `labels`, except that an optional `:def` is allowed before each definition. Example:

```
(defun apply-to-leaves (fn tree)
   (let-fun ((walk-tree (tr)
                (cond ((atom tr)
                       (leaf-handle tr))
                      (t
                       (mapcar #'walk-tree tr)))))
      (walk-tree tree)

    :where
```

```
        (:def leaf-handle (x) (funcall fn x))))
```

The `:def` is purely optional; its presence is mainly to help search for local functions in text editors.

The only other notational variation from `labels` is that "_" may be used instead of a parameter in any of the local function definitions. The "_" indicates a parameter whose value is ignored. So

```
    (let-fun ()
       #'foo
     :where
        (:def foo (a _ c)
             ...))
```

is equivalent to

```
    (let-fun ()
       #'foo
     :where
        (:def foo (a b c)
            (declare (ignore b))
            ...))
```

Actually, you can just write `(ignore b)` instead of `(declare (ignore b))`.

The existence of `:where` makes possible a liberalization of the usual rules for indenting Lisp code. If the very last thing in a function definition is a `:where` followed by some local function definitions and closing parentheses, then I sometimes allow myself to move those definitions to the left of the column containing the left paren before `let-fun`. So I might write the definition above thus:

```
(defun apply-to-leaves (fn tree)
   (let-fun ((:def leaf-handle (x) (funcall fn x)))
      (walk-tree tree)

    :where

 (:def walk-tree (tr)
    (cond ((atom tr)
           (leaf-handle tr))
          (t
           (mapcar #'walk-tree tr))))))
```

Of course, this device is wasted on small examples, but for large procedures it can save one from dividing a program into arbitrary globally defined chunks just to avoid "Vietnamization." [1]

There is an analogous facility called `let-var`:

```
(let-var (---local-vars---)
   ---body---
 [:where ---local-vars---])
```

Each *local-var* is in the standard form used in `let`.

2.2 `repeat`: A Clean Loop Facility

The complex `loop` macro of Common Lisp is, in my opinion, an aberration. Its syntax is un-Lisp-like and its semantics are obscure. However, it does supply certain facilities that are useful, especially the ability to collect values in a list in the order they are generated.

[1] The tendency of Lisp code to wander off to the right, then eventually back to the left, until it resembles the map of Vietnam.

`repeat` *Macro* *Location:* Ytools, `repeat`

The YTools `repeat` facility duplicates the useful parts of the built-in version, adds some other useful features, and looks more Lisp-like:

```
(repeat [:for (---varspecs---
               [[:collectors | :collector] ---vars---])]
  ---repeat-clauses---
[:where ---local-fundefs])
```

where

```
varspec ::= var | (var val)
            | (var = start [:by inc] [:to thresh])
            | (var = start :then subsequent)
            | (var = start :then :again)
            | (var :in list [:tail var])

repeat-clause ::= exp+
    | :when test
    | [ :collect | :nconc | :append ] collect-spec
    | :within expression
    | [ :while | :until ] test
    | :result exp

collect-spec ::= exp | (:into var exp)
```

The keyword `:for` signals that we are binding variables for the scope of the `repeat`. There are two classes of variable: collector variables and all the other kinds. The collectors are declared after all the others, preceded by the keyword `:collectors` (or `:collector`, if it looks better). Each collector variable is initialized to an empty collector.

The abstract datatype `Collector` is essentially a queue. Most of the time the operations on collectors are implicit in various `repeat` constructs, but there are times when you need to manipulate collectors directly. You create a collector by evaluating `(empty-Collector)`. The objects the collector c contains are obtained by evaluating `(Collector-elements` c`)`. A new element x is added to the end of the elements by evaluating `(one-collect` c x`)`. An entire list can be added by `(list-collect` c l`)`. `list-collect` does *not* copy its list argument, so that subsequent collection operations may destructively alter that argument. To reset a collector c to its empty state, execute `(collector-clear` c`)`.

`repeat` provides several constructs for binding, initializing, and stepping non-collector variables (where v means a symbol used as a variable):

- v: Bind v for the duration of the `repeat`.
- `(`v e`)`: Bind v to e; it may be reset inside the `repeat`.
- `(`v `=` e `...)`: Binds v to e initially. What happens on subsequent iterations varies among subcases:
  - `(`v `=` e`)`: Equivalent to `(`v e`)`.
  - `(`v `=` e `[:by` i`] [:to` r`])`: On each iteration, add i (default 1) to v. Stop when v passes threshold r (default: don't stop). The phrase "passes threshold" means "is greater than" unless i is a negative constant, when it means "is less than." The expressions i and r are evaluated just once, before the iteration begins and any of the variables are bound.
  - `(`v `=` e `:then` s`)`: On each iteration after the first, v is set to the value of s, which is reevaluated on each iteration, with all the repeat variables in scope.

  – `(`v `=` e `:then :again)`: Equivalent to `(`v `=` e `:then` e`)`.

- `(`v `:in` l `[:tail` v_t`])`: v is bound to successive elements of l. Iteration stops when the elements are exhausted. If `:tail` v_t is present, v_t is a symbol that is bound to the tail of l starting with v. *It is okay to reset v_t in the body of the `repeat`,* thus changing the rate at which the loop advances; setting it to `()` causes the loop to terminate before the next iteration. If you care about the value of v_t, but not about the value of v, you can replace v with _. This is the only valid use of _in the bound variables of a `repeat`.

Here is how the `repeat` is executed:

1. Auxiliary values (such as the `:to` expression e in `(`v `= ...:to` e`)`) are evaluated.

2. All variables are bound simultaneously.

3. All the *local-fundefs* are created, just as for `let-fun`. These functions can "see" all the bindings of the variables.

4. Every test implied by the variable-binding constructs is performed. That is, for every binding of the form `(`v `:in` l `...)`, it is checked whether l has any (remaining) elements. For every binding of the form `(`v `=` e `...:to` l`)`, it is checked whether v is $\leq l$. Iteration stops if a test fails.

5. Repeat clauses are executed in order. An ordinary expression is evaluated. A test of the form `:while` e or `:until` e causes e to be evaluated; iteration stops if a test fails. A test of the form `:when` e causes e to be evaluated; if e is *False*, the rest of the repeat clauses are skipped and control advances to the next iteration.

6. A clause of the form `:collect` c, `:nconc` c, or `:append` c, where c is of the form `(:into` v e`)`, is handled by evaluating e and adding to the elements of v, which must be one of the collectors declared after `:collectors`. If c is just e, with no `:into`, the first collector is implied. The difference between the different clause forms is that

   - `:collect` causes one element to be added to the end of the collector elements.
   - `:nconc` causes a list of elements to be added (as `list-collect` would).
   - `:append` is the same as `:nconc` except that the list is copied before being added.

7. A clause of the form `:within` e is equivalent to e by itself, except that anywhere within e may appear:

   `(:continue` —*repeat-clauses*—`)`

   These clauses are evaluated within the lexical context established by e, but are otherwise interpreted as if they appeared at the top level of the `repeat`. (See example below.)

8. If control reaches the end of the `repeat` body (because a `:when` test came up *False* or the last clause was performed), all the variables with changes indicated by the binding constructs are changed simultaneously, like the variables in a `do`. Then control branches back to the implied tests at the beginning of the clauses.

9. Whenever a test indicates that iteration ends, the first `:result` e following the test is found, e is evaluated in the scope of the variables, and the result is returned. However, a `:result` inside a `:continue` is not counted as "following" tests outside that `:continue`. That is, when a test indicates that the `repeat` is to end, the operative `:result` is the one found by searching through the current `:continue`, then the immediate enclosing `:continue`, and so on, up to the top level of the `repeat`.

Important note: In a result clause, the value of a collector variable is the list of values accumulated. Everywhere else, including inside local functions called by a result clause, the value is the actual Collector.

If there are no tests in the `repeat`, implied or explicit, the macro expander will issue a warning message. Sometimes not having any tests is actually correct, because the `repeat` is going to exit some nonstandard way (say, by throwing a value). (`repeat` establishes a block named `nil`, so `return` can be used to exit it as well.) To make the warning go away, put in a `:while` or `:until` whose test is a string. This will be discarded and the warning suppressed.

Any atom in the repeat body is ignored unless it is one of the keywords specified above (or one allowed by the abbreviation convention below). So you can write `:else :result` if it makes the control flow clearer.

The syntax of `repeat` is often considerably simplified by the use of the following abbreviations:

- If no `:collectors` are declared, but there are `:into`-less clauses of the form `:collect` *e*, then a default collector *d* is created, and the collect clause is interpreted as `:collect (:into` *d e*`)`.
- If there is an (explicit or implied) termination test with no `:result` after it, the result of the repeat is the contents of the first or default collector, if any; if there are no collectors, the `repeat` has no predictable result, and is presumably executed just for its effects.
- If there is just one variable binding, whose second element is `:in` or `=`, then the parens around the variable bindings can be omitted. That is, you can write `(repeat :for (x :in l) ...)` instead of `(repeat :for ((x :in l)) ...)`.

These rules have the consequence that if there is just one collector var, then you may usually omit all its occurrences. So `(repeat :for (... :collector c) :collect (:into c e) :result c)` can be written `(repeat :for (...) :collect e)`.

As an example of `repeat`, a `do` loop

```
(do ((v1 e1 b1)
     (v2 e2 b2))
    ((test v1 v2) (res v2 v1))
  (format srm "~s~%" (foo v1 v2)))
```

can be written as

```
(repeat :for ((v1 = e1 :then b1)
              (v2 = e2 :then b2))
 :until (test v1 v2)
    (format srm "~s~%" (foo v1 v2))
 :result (res v2 v1))
```

The `:within`-`:continue` construct can be used to wrap variable binders and conditional tests around repeat clauses. Example:

```
(repeat :for ((x :in l)
              :collector c)
 :within
    (let ((y (expen x)))
       (cond ((proper y)
              (:continue
                :collect y
                :until (final x)))))
 :result c)
```

(We could omit all explicit mentions of the collector `c`, but the result would probably be more obscure.)

3 Facilities for setting and matching

YTools defines various facilities for setting variables.

3.1 The `!=` macro

`!=` *Macro* *Location:* Ytools, `setter`

`(!=` *place newval*`)` is a generalization of `setf`, providing these extra features:

1. If *place* is a sequence of the form `<` v_1 `...`v_n `>`, then `(!= <` v_1 `...`v_n `>` *newval*`)` is equivalent to `(multiple-value-setq (`v_1 `...`v_n`)` *newval*`)`. This is the only case where `!=` has more than two arguments. The spaces before and after the brackets are necessary. You can write _for any value that is not used., as in `(!= < a _ c > (foo c d))`, which uses only the first and third values returned by `foo`.

2. `(!= (<` v_1 `...`v_n `>)` *newval*`)` sets v_i to the i'th element of *newval*, which must be a list of at least n elements. Again, the spaces separating the brackets from the v's are necessary; and the _notation may be used to skip elements of *newval*.

3. If neither of these cases applies, then *newval* may contain occurrences of `*-*`, which stands for the contents of *place* before the assignment. For example, `(!= (car (get foo 'tally)) (+ *-* 3))` augments `(car (get foo 'tally))` by 3. The `!=` macro uses setf-expanders to avoid recomputing the left-hand side to the extent possible. The example above might expand to

   ```
   (let* ((#:g11726 (get foo 'tally))
          (*-* (car #:g11726))
          (#:g11725 (+ *-* 3)))
     (rplace #:g11726 #:g11725)
     #:g11725)
   ```

   Just like `setf`, `!=` always returns its second argument, the new value.

3.2 Qvars and the `matchq` macro

The question mark character is reserved by YTools for use in writing "match variables," or *qvaroids*. A special case of the qvaroid is the *qvar*, which is written (and readable) as `?`*sym*. A qvaroid is an abstract object consisting of four slots:

1. `sym`: A symbol
2. `notes`: A list of stuff, used for various purposes
3. `atsign`: A boolean
4. `comma`: A boolean

A qvaroid can be constructed using `make-Qvaroid`: `(make-Qvaroid` a c s l`)` makes a qvaroid with `atsign`=a, `comma`=c, `sym`=s, and notes=l. To retrieve a field of a qvaroid q, write `(Qvaroid-atsign` q`)`, `(Qvaroid-sym` q`)`, and so forth.

The external representation of a qvaroid is `?[@][,](`s `.` l`)`. E.g., a qvaroid with `atsign`=*False*, `comma`=*True*, `sym`=`foo`, and notes=`(`a b`)` is `?,(foo a b)`. This representation is also readable. (When read, the comma and atsign may appear in either order.) If the "notes" field is the empty list, the parens may be dropped.

A *qvar* is a qvaroid with `atsign`=`comma`=*False* and notes=empty list. It is read and printed as in `?foo`. Its constructor is `(make-Qvar` *sym notes*`)`. (Okay, so `make-Qvar` will make a qvaroid if its second argument is non-empty.)

These datatypes may be used for any purpose you see fit, including especially writing unification algorithms and the like. There is also a built-in list matcher:

`matchq` *Macro* *Location:* Ytools, `setter`

`(matchq` *pattern datum*`)`, where the *pattern* and *datum* are list structures, tries to match the pattern against the datum. If it succeeds, it returns *True*, else *False*. It has as side effect that the variables in some of the qvaroids in the pattern get new values drawn from the parts of the datum that they match.

A simple example is `(matchq (P ?x ?y) d)`. The match succeeds if `d` is a list of exactly 3 elements, of which the first is `P`. The variable `x` is set to the second element and `y` to the third.

It is important to realize that `matchq` is expanded at compile time, so that the pattern does not need to be scanned at run time. The example above expands to something like this:

```
(let ((ttt d))
  (and (consp ttt)
       (eq (car ttt) 'P)
       (let ((ttt (cdr ttt)))
         (and (consp ttt)
              (progn (!= x (car ttt)) true)
              (let ((ttt (cdr ttt)))
                (and (consp ttt)
                     (progn (!= y (car ttt)) true)
                     (null (cdr ttt)))))))))
```

Assignments are done immediately as pieces of the datum are matched. So even if the match fails, some of the variables in the pattern could be set. Matching is done left-to-right, so if a program can detect which variable got set and which didn't, it can infer where the match failed, or at least a set of places where it might have failed.

If a qvaroid has the comma flag set, then it matches the current value of its sym. So `(matchq (P ?x ?,y) d)` succeeds (and sets `x`) if `d` is of the form `(P` *anything* v`)`, where v is the value of `y`. And `(matchq (P ?x ?,x) d)` succeeds if `d` is of the form `(P` a a`)`, setting `x` to a.

If a qvaroid has the atsign flag set, then it matches a *segment* of values in the datum. A segment is a sequence of zero or more values. If a qvaroid matches a segment, its variable is assigned to a list of the values in the segment. For example, `(matchq (P ?@x z) d)` succeeds if `d` is a list beginning with `P` and ending with `z`. The variable `x` is bound to all the values between them; if `d` = `(P ying yang z)`, then `x` gets value `(ying yang)`; if `d` = `(P z)`, then `x` gets value `()`. *Important restriction:* there can be at most one segment (atsigned) variable in a given list (i.e., a given sublist of a list structure). This restriction is required to ensure that the matcher doesn't have to backtrack.

If both the atsign and comma flags are set, then the qvaroid matches a sequence of values equal to the current value of its sym.

If the notes field of a qvaroid is non-empty, then the qvaroid is a special match construct, one of the following. (I use the word "datum" here to mean "the piece of the datum being matched against this construct.")

- `?(:&` p_1 `...` p_n`)`: matches the datum if each of the p_i does.
- `?(:\|` p_1 `...` p_n `[:&` $p_\&$`])`: matches the datum if one of the p_i does. If the fragment "`:&` $p_\&$" is present, then the datum must also match $p_\&$, which is useful for binding a variable to a datum that matches one of several patterns. For instance, the pattern `(a ?(:\| huey dewey louie :& ?name))` matches any of the data `(a huey)`, `(a dewey)`, and `(a louie)`, setting `name` to the `cadr` of the datum. Another way of putting it is that the example pattern is equivalent to `(a ?(:& ?name ?(:\| huey dewey louie)))`, but more concise.

- `?(:+` p r_1 `...` r_k`)` matches the datum if p matches it, and it satisfies all the predicates r_i. Each predicate is the name of a function; write `fun` instead of `#'fun`.

In these qvaroids, the atsign flag may be used, but not the comma flag. When the atsign flag is present, the piece of the datum matched is a segment of the original datum.

Examples:

```
(matchq (P ?(:& ?x (foo ?,y)))
        d)
```

succeeds if `d` is of the form `(P (foo` v`))`, where v is the value of `y`. In addition, it sets `x` to `(foo` v`)`.

```
(matchq (P ?@(:+ ?l (\\ (x) (is-list-of x #'is-Integer))))
        d)
```

succeeds if `d` is of the form `(P` n_1 `...` n_k`)`, where each n_i is an integer. In addition, it sets `l` to the list of all the n_i.

If you care about the structure of the datum, but don't want to assign a variable, you can use the `_` convention. So the pattern `(P ?_ ?@_)` matches any list of length at least two starting with a `P`, without setting anything.

3.3 Applications of the matcher

`match-cond` *Macro* *Location:* Ytools, `setter`

The matcher is used to implement the `match-cond` macro:

```
(match-cond datum
   [declarations]
   ---clauses---)
```

The *datum* is evaluated, and then the *clauses* are handled the same as in `cond`, except for clauses that are qvaroids. Any clause of the form

```
(:? pat
  ---body---)
```

is handled by matching the pattern *pat* against the datum. If the match succeeds, the *body* is executed, and no further clauses are examined.

All the (comma-free) variables in the patterns of a `match-cond` are bound with scope equal to the `match-cond`. In addition, the variable `match-datum` is bound to the *datum* being matched.

Example:

```
(match-cond (get x 'dat)
   (:? (P ?u ?v)
     (list 'P u v))
   (:? ((lambda (?v) (Q ?,v)) ?@_)
     (list 'lambda v))
   (t 'nomatch))
```

is equivalent to

```
(let ((match-datum (get x 'dat)))
   (let (u v)
       (cond ((matchq (P ?u ?v) match-datum)
               (list 'P u v))
              ((matchq ((lambda (?v) (Q ?,v))
                         match-datum)))
              (t 'nomatch))))
```

`match-let` *Macro* *Location:* YTools, `setter`

The `match-let` macro is used as a “destructuring binder”:

```
(match-let pattern datum
   ---body---)
```

The *pattern* must match *datum*; assuming it does, the *body* is executed. If the match fails, an error is signaled.

4 An Improved Formatted I/O Facility

4.1 The `out` Macro

`out` *Macro* *Location:* YTools, `outin`

The `out` macro is an alternative to the awful `format` facility
`http://www.cs.yale.edu/homes/dvm/format-stinks.html`.
Unlike `format`, which separates a “control string” from the data to be output, the `out` macro interleaves them. For instance, the `format` in the `do` example of the previous section

```
(format srm "~s~%" (foo v1 v2))
```

can be written

```
(out (:to srm) (foo v1 v2) :%)
```

Note that the “`out` directive” `:%` has a name similar to the corresponding `format` directive `~%`.

As a slightly more complex example, to output `x`, `y`, and their sum, with appropriate annotations: one could write

```
(out "x = " x ", y = " y :% 3 "x+y = " (+ x y) :%)
```

If `x` is 10 and `y` is 13, then this would cause the following output:

```
x = 10, y = 13
   x+y = 23
```

The “3” means “insert 3 spaces.”

The general form of `out` is

```
(out [(:to stream)] ---out-directives---)
```

If `(:to stream)` is present, output goes to the value of the expression *stream*, otherwise to `*standard-output*`. If *stream* is the symbol `:string`, then output goes to a new string which is eventually returned as the value of the `out` form. Each *out-directive* is one of those shown in table 1.

For example, here is how you might output a list `dtl` of defective objects of type `Tribbly`:

```
(out (:to *error-output*)
   "The following tribblies have problems:"
   :% (:e (repeat :for ((dt :in dtl)))
             (:o (Tribbly-name dt) :%
                 "Problems:")
             (repeat :for ((tt :in (Tribbly-troubles dt)))
                (:o tt :%))
             (:o :%)))
```

If we want each tribbly to be indented, and each problem to be indented under its tribbly, and also to avoid plural nouns when grammatically inappropriate, we could get fancier:

An integer n	If $n > 0$, insert this many spaces into the output. If $n < 0$, insert $-n$ newlines into the output. If $n = 0$, insert a newline unless already at the beginning of a line.
`:%`	Insert a newline.
A string	`princ` the string.
`(:a` e`)`	Evalute the expression e and `princ` the result.
`(:t` n`)`	Tab to column n (which is *not evaluated*).
`(:_` e`)`	Evaluate expression e, producing an integer. Treat it as though it occurred as an out-directive (i.e., print spaces or newlines).
`(:i=` e`)`	Evaluate e, getting an integer n; indent all further lines by n spaces from the left margin. This indentation carries over to *all* calls of `out` on the same stream until the current `out` form finishes.
`(:i>` e`)`	Like `:i=`, except that new indentation is relative to the current indentation.
`(:i<` e`)`	Equivalent to `(:i> (-` e`))`.
`(:q ---`*clauses*`---)`	Each *clause* is of the form `(`*test* `---`*out-directives*`---)`. Resume processing out-directives with the list from the first *test* that evaluates to a non-nil value.
`(:e [(:stream` v`)] ---`*exps*`---)`	Evaluate each *exp* and discard the results. Any subexpression of *exp* of the form `(:o` $d\ldots$`)` is, every time it is executed, treated as though $d\ldots$ had occurred among the top-level *out-directives*. If the `(:stream` v`)` part is present, then the variable v is bound to the stream being printed to for the duration of the `e` form. See text for explanation.
`(:pp-block [(:pre` p`)]` $d\ldots$ `[(:suf` s`)])`	Print a logical block, with the given optional prefix and suffix. $d\ldots$ is a sequence of out-directives.
`(:pp-ind [:block |:current]` n`)`	Indent subsequent lines in the current logical block by the value of n. Indentation is relative to the block or the current indentation depending on whether `:block` or `:current` is the first argument.
`(:pp-nl [:linear | :fill | :miser | :mandatory])`	Possibly insert a newline into a logical block. See the documentation for `pprint-newline`.
`(:f` c `---`*args*`---)`	Output the *args* under the control of the `format` control string c. Still the best way of printing floating-point numbers.
`(:v` e`)`	Evaluate e and save its value(s). The value(s) of e become the value of the `out`-form.
Anything else	Evaluate it and print the result (just like the "~s" `format` directive, which requires no counterpart in `out`).

Table 1: `out`-directives

```
   (out (:to *error-output*)
      (:q ((null dtl)
           "All tribblies are okay" :%)
          (t
           "The following tribbl"
           (:q ((> (length dtl) 1) "ies have")
               (t "y has"))
           " problems: "
           :%
           (:e (repeat :for ((dt :in dtl) num)
                   (setq num (length (Tribbly-troubles dt)))
                   (:o (Tribbly-name dt)
                       (:i> 3) :%
                       "Problem"
                       (:q ((not (= num 1)) "s"))
                       ": " (:i> 3) :%
                       (:e (repeat :for ((tt :in (Tribbly-troubles dt)))
                              (:o tt :%)))
                       (:i< 6) :%))))))
```

The `out` macro fiddles with its output stream behind the scenes, in much the way `pprint-logical-block` does. (The two manipulations are entirely orthogonal, and you can freely intermingle calls to one with calls to the other.) Hence the result of writing to the same stream outside the `out` regime is undefined. This is the reason for the `:stream` field in the `:e` out-directive. It causes the specified variable to be bound to the modified stream object being output to. It is safe to pass this object to calls to `out` from within subroutines called inside an `:e` directive. For example, suppose we want to create a subroutine that behaves similarly to the `~p` directive in a format control string (except it takes a list or a number as input):

```
(defun pluralize (n srm &optional (alt-endings '("" "s")))
   (cond ((not (is-Number n))
          (setq n (length n))))
   (out (:to srm)
        (:q ((= n 1) (:a (car alt-endings)))
            (t (:a (cadr alt-endings))))))
```

Now we can write our example as

```
   (out (:to *error-output*)
      (:e (:stream errsrm)
          (:o (:q ((null dtl)
                   "All tribblies are okay" :%)
                  (t
                   "The following tribbl"
                    (:e (pluralize dtl errsrm '("y has" "ies have")))
                   " problems: "
                   :%
                   (:e (repeat :for ((dt :in dtl) num)
                           (setq num (length (Tribbly-troubles dt)))
                           (:o (Tribbly-name dt)
                               (:i> 3) :%
                               "Problem"
                               (:e (pluralize num errsrm))
                               ": " (:i> 3) :%
                               (:e (repeat :for ((tt :in (Tribbly-troubles dt)))
                                      (:o tt :%)))
                               (:i< 6) :%))))))))
```

The stream passed to `pluralize` is essentially the same as `*error-output*`, but safe to write to using inner calls to `out`.

The indentation level used by `out` can be altered by using the macro `out-indent` instead of the `:i>`,`:i<` directives.

```
(out-indent srm n
    ---body---)
```

binds the indentation level of the stream *srm* to its current value + n and executes the *body*. `(out-indent` s n `(out ...)` e`)` is roughly equivalent to `(out :to` s `(:i>` n`) ...(:v` e`))` but can be less obscure and more efficient. For example,

```
(out-indent *error-output* 3
   (recurse))
```

may well end up being equivalent to a straightforward call to `recurse`, if nothing is printed to `*error-output*`. In the same circumstance, `(out (:to *error-output*) (:i> 3) (:v (recurse)))` must make a list of the values returned by `recurse` and turn it back into a row of values when `out` returns. Whether or not this efficiency loss is important can be debated, but, because the `out`-form doesn't actually seem to print anything, it looks puzzling.

The `dbg-out` macro is often handy. `(dbg-out` *gate-var* *—out-directives—*`)` is equivalent to

```
(cond (gate-var
         (out (:to *error-output*) ---out-directives---)))
```

except that if the *out-directives* don't end with `:%`, a newline is inserted into the stream after everything is printed.

4.2 The `in` Macro

`in` *Macro* *Location:* YTools, `outin`

For input, YTools supplies a simple facility called `in`. The form is similar to that of `out`:

```
(in [(:from stream)] ---in-directives---)
```

This reads in a number of objects and returns them as multiple values. If an end of file occurs, no error is signaled, but instead the value `eof*` is returned instead of the object sought. More precisely, if the `in` would normally return N values, and only $M < N$ can be read, then values $1, \ldots, M$ are the objects read, and values $M+1, M+2, \ldots, N$ are `eof*`.[2]

The repertoire of `in` directives is considerably smaller than the set of `out` directives:

- `:obj` — A Lisp object is read from the input stream.
- `:char` — A single character is read from the input stream.
- `:peek` — A character is peeked at and returned (but left in the input stream).
- `:string` — A whitespace-delimited string is read. If an end-of-file is encountered, `eof*` is returned if the string is so far empty. Otherwise, the `eof*` behaves like a whitespace, and simply ends the string.
- `:linestring` — A line is read and returned, as if by `read-line`.

[2]It is a property of the `in` macro that the number of objects returned can be ascertained at compile time.

- `:linelist` — A line is read and returned as a list of Lisp objects. If an object is a list, the `:linelist` reader may well have to go on to other lines to read the whole thing. So a more precise definition of `linelist` is: Return the shortest list of Lisp objects `(`b_1 `...` b_n`)` such that (a) readable representations of b_1, b_2, $\ldots$, b_n are the first n things in the input; and (b) only whitespace remains on the line where the readable representation of b_n ends.
- `:keyword` — A string is read (as `:string` would), and the result is interned as the name of a symbol in the keyword package.

5 Signaling Conditions

In Common Lisp, `error`, `cerror`, `signal`, and other constructs take "condition designators" as arguments. These can include (ugh) `format` arguments, so we replace all of these with macros that allow `out` instead. The main macro is:

`signal-problem` *Macro* *Location:* YTools, `signal`

```
(signal-problem [place-spec]
     [condition-spec]
     [proceed-spec])
place-spec ::= [:place] p | :noplace
condition-spec ::= (:condition c)
                   | (:class condition-class ---args---)
                   | ---outargs---
proceed-spec ::= :fatal | :proceed
                 | (:proceed ---restart-description---)
                 | (:prompt-for ---object-description-- default)
```

The main thing `signal-problem` does is create a condition object and signal it. There are three ways to describe the object to be created:

1. `(:condition` *c*`)`: *c* evaluates to the condition.
2. `(:class` *c* `---`*args*`---)`: The condition is obtained by evaluating `(make-condition '`*c* `---`*args*`---)`.
3. *Anything else:* is interpreted as describing a vanilla condition that prints as though the *condition-spec* were arguments to `out`.

As with the standard Lisp condition signalers, if the condition is handled, then control transfers to the handler. Otherwise, the debugger is entered, which is where the *place-spec* and *proceed-spec* come in. The debugger prints a message such as

```
Error:  p broken
```

where p is the object specified by the *place-spec*; p is usually a symbol, but can be anything; it isn't evaluated. The guide symbol `:place` can be omitted if p is a symbol. This convention can lead to bugs; if you write `(signal-problem x " < 0")`, the debugger will print:

```
 Error:  x broken
< 0
```

which is probably not your intention. You can avoid `x` being taken for the place name by writing `:noplace` where the place designation goes. If `x` has value -5, you can write `(signal-problem :noplace x " < 0")` to get

```
  Error:  BREAK
-5 < 0
```

When the debugger is entered, the *proceed-spec* influences the displayed restarts. If it's `:fatal` (the default), there is no way to continue from the error. If it's `:proceed`, there will be a restart with a bland message such as "I will try to proceed." You can tailor the message by writing `(:proceed` *`form`*`)`, where *form* evaluates to the string you want associated with that restart.[3]

If the *proceed-spec* is of the form `(:prompt-for` *`string default`*`)`, then the message associated with the "continue" restart is

`You will be prompted for:` *string*

If you take that continuation, you will be given the choice of typing `:ok` or of typing `:return` e. In the former case `:signal-problem` returns the value of *default*; in the latter, of e.

Some relatives of `signal-problem`:

`signal-condition` *Macro* *Location:* YTools, `signal`

`(signal-condition` *condition-spec*`)` signals the condition described by *condition-spec* (see above). If it is not handled, `signal-condition` returns `nil`.

`breakpoint` *Macro* *Location:* YTools, `signal`

`(breakpoint` *—out-directives—*`)` is just like `break`, except that `out` is used to print its arguments instead of `format`.

6 Classes and Structures

One of the cool things about Common Lisp is that you can specialize a generic function on any or all arguments, and any or all datatypes. It is just as easy to specialize on an Integer as on some hairy CLOS class. Because this is so, the distinction between classes and structures is quite blurry. Structures are in some sense "light-weight" classes. But the macros used to define them, `defstruct` and `defclass`, have rather different syntax and several unnecessary differences in effect. It would be nice to have a way of defining an abstract data type that deemphasized whether it was a class or a struct. That way, you could flip easily between implementing it as a class or as a struct, without having to change lots of code.

`def-class` *Macro* *Location:* YTools, `object`

The `def-class` macro does exactly that. In addition, it focuses attention on a subset of object-oriented programming, which happens to be the subset I use. The result is a somewhat more concise language for describing the usual cases. If you love really hairy OOP, then this tool is not for you. On the other hand, if you don't even want to know whether an abstract datatype is a structure or a class, give it a whirl.

The full syntax is thus:

[3]Older versions of YTools used `:continue` instead of `:proceed`, and it is still allowed, but deprecated because using it inside a `repeat-:within` can cause confusion.

```
(def-class name
         ---slot-defns----
         [(:handler
              ---meth-defns---)]
         [(:options [(:include ---components---)]
                    [(:medium [:list | :vector
                                | :structure | :object]
                              [:named]
                              [:already-defined])]
                    [:key])])
```

A given call to `def-class` defines either a structure or a class type. For conciseness, I will use the term *classoid* to refer to the datatype defined by a given call to `def-class`.

Each *slot-defn* is either a symbol naming a slot, or a list of the form

```
 (slotname initform [:type type])
```

Each *meth-defn* in the "handler" is of the form required by `defmethod` (section 6), without the explicit `defmethod`. (The `:handler` field and the `:options` field can be in either order, and can appear anywhere in the body of the `def-class`, even in the slot list.)

The `:include` field specifies the components of the classoid, i.e., the superclasses or included structures.

The `:medium` option gives the choice of four "media": `:list`, `:vector`, `:structure`, and `:object`. The first three cause `def-class` to expand into a `defstruct`, the last, into a `defclass`. If the `:medium` option is omitted, then the `def-class` defines a class (that is, it expands into a `defclass`) if and only if there is more than one component classoid, or the only component is a class. (Of course, if there is more than one component, they must all be classes or an error will be signaled.) If the medium is `:vector` or `:list`, the new objects defined by this `def-class` will be implemented as ordinary vectors and lists. If the `:named` flag is present, the first slot of such a vector or list will be reserved for the name of the class; otherwise, it will just be an anonymous vector or list. The `:named` flag is redundant for all other media.

The `:already-defined` flag means that someone else defined the classoid, and this call to `def-class` is just for the purpose of declaring it. If `:already-defined` is present, the classoid can have slots but not a handler or any components.

Unlike `defstruct` and `defclass`, `def-class` by default creates a positional constructor, always called `make-`*classname*. The order of the arguments is determined as follows: Find the class(oid) precedence list for all the components and reverse it; now enumerate all the slots of each component in that list; for each classoid, the slots are included in the order they were declared in.

Here's an example of a couple of `def-class`es that expand into a couple of `defstruct`s:

```
(def-class Animal
   blood-temp numlegs)
(def-class Mammal
          (:options (:include Animal))
    (lays-eggs false :type boolean))
```

The constructor `make-Mammal` for `Mammal` takes three arguments, `blood-temp`, `numlegs`, and `lays-eggs`, in that order.

Non-key constructors are useful for simple classoids (which most of mine are), but become unwieldy for classoids with many slots or components. To declare a key constructor instead, add the `:key` option. If you don't, you can still use a key-constructor, because `def-class` defines an extra constructor `make-`*classname*`-key` that uses `&key` arguments.

The `def-class` macro will warn you if you defined a classoid with a non-key constructor that has more than 10 slots or 2 classes. In addition, if the class being defined has a component with a key constructor, then the macro will give this one a key constructor, too. [[There is currently no way to override this behavior.]]

`make-inst` *Macro* *Location:* YTools, `object`

`initialize` *Generic Function* *Location:* YTools, `object`

Finally, if the classoid is a class, one can make an instance by writing `(make-inst` *classname —key-args—*`)`. `make-inst` is just like `make-instance`, except that the first argument is not evaluated. It's the name of the class, not something that evaluates to that name.

Once an object is created, by any of the methods above, the generic function `initialize` is applied to it. The methods that are called as a result can perform tasks such as filling slots that still don't have values after using their initforms. The following are helpful for this task:

- `(slot-is-filled` *ob slot*`)` tests whether the given *slot* is bound in *ob*. (Both *ob* and *slot* are evaluated.) One difference between class and structure instances is that the former can have truly unbound slots, whereas structure slots are always filled, traditionally with `nil`. In YTools, a structure slot has default value `+unbound-slot-val+`, a constant bound to a unique object. So `slot-is-filled` tests whether a slot is truly unbound *or* has value `+unbound-slot-val+`.

- `(slot-defaults` *ob* s_1 v_1 `...` s_n v_n`)` fills unfilled slots of *ob*. If slot s_1 is unfilled, v_1 is evaluated and used to fill it. Then if s_2 is unfilled, v_2 is evaluated and use to fill s_2. The slotnames are not evaluated. The order of evaluation and slot filling is left-to-right, so later values may use earlier ones.

The `handler` of a classoid defines methods in the usual way. If for some reason you want to use the `:print-function` or `:print-object` options of a `defstruct`, you write a handler clause with the corresponding keyword where the generic function name should be, as in this example:

```
(def-class Sec-method
   public-key private-key
   (:handler
        (:print-function (sec-meth srm lev)
             (out (:to srm) "#<Security method, public key = "
                            (:q ((> lev *print-level*)
                                 (Sec-method-public-key sec-meth))
                                (t "####"))
                            ">"))
        (cough-it-up (sec-meth)
           (Sec-method-private-key sec-meth))))
```

Note that the first argument of the second method definition doesn't look right. It should be of the form `(sec-meth Sec-method)`. `def-class` will fix up such discrepancies, but only for the first argument. The discrepancies go both ways: the `:print-function` option for `defstruct` does not expect a specializer on its first argument, and if you include one it will be removed.

Classes and structures have another difference that `def-class` smooths over. Suppose in my mammal example we have an instance of `mammal` stored in `v1`. We can refer to its `numlegs` slot by writing `(Mammal-numlegs v1)` *or* `(Animal-numlegs v1)`. But suppose we now switch to implementing them as objects, thus:

```
(def-class Animal (:options (:medium :object))
    blood-temp numlegs)
(def-class Mammal
            (:options (:include Animal) (:medium :object))
    (lays-eggs false :type boolean))
```

Do the calls to `Mammal-numlegs` still work? The answer is Yes. YTools defines all the functions required to make `defclass` behave like `defstruct` in this regard. This spares us from having to change all occurrences of `Mammal-numlegs` to `Animal-numlegs`, or from having to define the auxiliary functions by hand.

[[One feature missing from `def-class` is `conc-name`. It could be included without too much effort]]

7 Miscellaneous Features

7.1 The BQ Backquote Facility

Backquote is an indispensable feature of Lisp. Yet the standard spec for it leaves something to be desired. I have two main complaints:

1. There are three things to implement when implementing a facility like backquote: a reader, a macro-expander, and a writer. The reader converts a character sequence such as `` `(foo ,x) `` into an internal form such as `(backquote (foo (bq-comma x)))`. (This is what Allegro reads it as.) The macro-expander then turns calls to `backquote` into constructor forms such as `(list 'foo x)`. The writer prints `(backquote (foo (bq-comma x)))` as `` `(foo ,x) ``.

   Unfortunately, the Common Lisp spec does not specify what the macros are. They are, therefore, implementation-dependent. Compare the situation with ordinary "quote," where there is a well-defined internal form `(quote` x`)`, and therefore a well-defined transformation from the external form `'`x. The problem with leaving it unspecified is that there is no way to write a portable code walker that does something special with backquoted expressions. In fact, an implementation is not required even to *have* an internal representation for backquotes. The reader and the macro-expander can be merged, so that `` `(f ,x) `` is read as `(list 'f x)`. Then the backquote writer's behavior is not well defined, because it is impossible to tell wheter a list-constructing form came from a backquote or not.

2. The rule for interpreting nested backquotes is that a comma is paired with the innermost backquote surrounding (and "raises" its argument out of that context, so that the next comma matches up with the next backquote, and so forth).

   I think this is wrong, or at least wrong in some cases. I read backquotes left-to-right, and hence see the outermost backquote first. One would like it to be the case that from that backquote's point of view, everything inside it is "inert" (quoted), except stuff marked with a comma. This is true for all expressions that might occur inside it, except another backquote. So if you are editing a complex backquote expression:

   ```
   `(foo (bazaroo '(fcn a ,x)))
   ```

   the inner quote doesn't "shield" `x` from evaluation. But if you convert the inner quote to a backquote, that's exactly what happens. You have to convert it to this:

   ```
   `(foo (bazaroo `(,fcn a ,',x)))
   ```

   The `,',` construct is just plain ugly. Its sole purpose is to raise its argument out of the innermost backquote; you can't say `,,x`, because that would mean "Evaluate `x` when the outer backquote is expanded, getting e, and then evaluate e when the innermost backquote is expanded." Notice how the order of evaluation is outside-in, while the nested-backquote rule is inside-out. Very, very confusing.

These are not huge defects; 99.9% of all backquotes are not nested, and almost no one cares what the internal representation of a backquote is. But if you're interested, the file `bq.lisp` provides an alternative implementation. It defines a portable macro, `ytools::bq-backquote`, for a backquote to expand into.

`ytools::bq-backquote` *Macro* *Location:* `bq`

`ytools::bq-comma` *Macro* *Location:* `bq`

The revised backquote is available as "`!‘`"; the original backquote is not disturbed by loading `bq`. You can use the new backquote just like the old: `!‘(f ,x)` evaluates to the same value as `‘(f ,x)`. The main innovation is that after the backquote and comma characters can come a single digit (between 1 and 9) that shows directly how to match up the backquotes with the commas. Compare the following two forms:

```
!‘1(foo (let ((x (k 3))) !‘2(baz ,1x ’,2y)))
!‘1(foo (let ((x (k 3))) !‘2(baz ,2x ’,1y)))
```

which, in an environment with `x = (car y)` and `y = ((d e f) b c)`, evaluate, respectively, to

```
(foo (let ((x (k 3))) !‘2(baz (car y) ’,2y)))
(foo (let ((x (k 3))) !‘2(baz ,2x ’((d e f) b c))))
```

Actually, to improve readability, the digit after a comma may be followed by a "`#`" character, and the backquote pretty-printer puts the character in when the expression following the comma-digit is an atom. So the two examples above actually print as

```
(foo (let ((x (k 3))) !‘2(baz (car y) ’,2#y)))
(foo (let ((x (k 3))) !‘2(baz ,2#x ’((d e f) b c))))
```

The digits after comma and backquote are optional, and default to `1`. Obviously, you can't nest a 1-labeled backquote inside another 1-labeled backquote, so as soon as you nest them you must use at least one explicit label. The digits needn't come in any order, so the inner backquote can be labeled `1` and the outer `2`, or vice versa. In the "`,@`" construct, the digit comes between the comma and the atsign.

Suppose you want to get an expression e to be evaluated when the outer backquote is expanded, and that value to be evaluated when the inner one is expanded. You must write them in this order: `!‘1( ... !‘2(... ,2,1`e`))`. The "`,2`" is treated as constant when the outer backquote is expanded, but its argument is evaluated and substituted, yielding `(...!‘2(... ,2`v`))`, where v is the value of e.

The typical use for nested backquotes is where you have a macro that expands into the definitions of one or more macros. For example, you might have a recursive data structure that is processed in several different ways. A *processor* of this data structure is a data-driven function that delegates most of the work at a node to a procedure that depends on the *identifier* of the node, that is, a symbol stored in the node that says what kind of node it is. For concreteness, picture the data structure as the parse tree of a sentence, with identifiers such as `noun-phrase`, `word`, `sentence`, and so forth. One processor of the parse tree might generate voice output for a sentence. Another processor might check the parse tree for errors. Another might verify that the tree is a legal parse of a given sentence. For the voice-output task you create a macro

```
(def-voice-handler n ...)
```

where n is the identifier for a node. Then you use the macro as in these examples:

```
(def-voice-handler noun-phrase ...)
(def-voice-handler verb-phrase ...)
```

Within the body of each macro, the variables `node` and `subnodes` should correspond to the node being processed and its subnodes.

For the parse-test task you create a macro

```
(def-parse-check-handler (n) ...)
```

In this macro, we can refer to `node` and `subnodes` as before, but also to `frag`, which is bound to the part of the sentence we are trying to verify.

A typical macro definition would look like the one for `def-voice-handler`:

```
     (defmacro def-voice-handler (name &body body)
         !`(!= (gethash ',#name voice-handler-tab*)
                (\\ (node subnodes) ,@body)))
```

The idea is that the voice-output processor finds the identifier for the node, looks up the handler in the hash table, then calls it, passing it the node and subnodes.

The definition of `def-parse-check-handler` would look almost identical, except that we would use `parse-check-handler-tab*` instead of the `voice-handler-tab*`, and would add the `frag` argument to the lambda expression.

If there are many different processors, one might want to write a general-purpose macro to define these macros automatically. Here is what it would look like:

```
(defmacro define-process-handler-macro (processor args)
   (let ((macro-name (build-symbol def- (< processor) -handler))
         (handler-table-name (build-symbol (< processor) -handler-tab*)))
   !`2(defmacro ,2#macro-name (name &body body)
        !`(!= (gethash ',name ,2#handler-table-name)
               (\\ (node subnodes ,2@args) ,body))))
```

Now we can just write

```
   (define-process-handler-macro (voice ()))
   (define-process-handler-macro (parse-check (frag)))
```

The second one expands into

```
(defmacro def-parse-check-handler (name &body body)
          !`(!= (gethash ',name parse-check-handler-tab*)
                 (\\ (node subnodes frag) ,body)))
```

which is exactly what we would have written by hand. Note that it was nearly effortless to turn constant parts of the original macro into evaluable expressions matching the outer backquote. (For an explanation of the `build-symbol` macro, see section 7.4.)

7.2 The Mappers

`<#`	*Macro*	*Location:* YTools, `mapper`
`<!`	*Macro*	*Location:* YTools, `mapper`
`<$`	*Macro*	*Location:* YTools, `mapper`
`<&`	*Macro*	*Location:* YTools, `mapper`
`<v`	*Macro*	*Location:* YTools, `mapper`
`<<`	*Macro*	*Location:* YTools, `mapper`
`</`	*Macro*	*Location:* YTools, `mapper`
`<?`	*Macro*	*Location:* YTools, `mapper`
`neg`		*Location:* YTools, `mapper`

These are all versions of the usual mapping functions, such as `mapcar` and `mapcan`, with a few twists thrown in. For instance, `(<#` *f —lists—*`)` means the same as `(mapcar` *f —lists—*`)`, except that *f* is treated as though it were in functional position. You can write `(mapcar #'foo l)` as `(<# foo l)`. The following table gives the correspondences between the YTools mappers and the built-in facilities:

```
<# = mapcar
<! = mapcan
<$ = mappend (see below)
<& = every
<v = some
<< = apply
</ = mapreduce (see below)
<? = remove-if, with predicate negated
```

The last line requires a bit of elaboration. `(<?` *pred list*`)` returns a new list consisting of all the elements *list* that satisfy *pred*.

If the symbol `neg` appears before the function designator in one of these mapping constructs, then it applies `not` to the value of the function. This is most useful in conjunction with the mappers `<&`, `<v`, and `<?`, but will work with any of them. Example: `(<? neg atom l)` returns a list of all the elements of `l` that are not atomic.

7.3 Data-driven Programming

Many Lisp procedures "walk" through S-expressions recursively, performing some operation on each subexpression and collecting the results in some way. In most of these procedures, some subexpressions have to be treated in an idiosyncratic way. For instance, if a procedure is walking through a Lisp program performing an operation that is sensitive to variable bindings, `let` and `lambda` expressions must be handled by a subprocedure that notes the new variables that are bound inside these expressions. One way to handle these special subexpressions is by using a `cond` to check for each case. This method becomes unwieldy and hard to maintain if there are many special cases. At that point it's appropriate to use the object-oriented approach, and "ask" the S-expression how it "wants" to be handled. Less poetically, we associate a table with the tree-walking procedure, and every time the tree walker comes to an expression E whose `car` is the symbol f, it looks in the table for a *handler* for expressions whose car is f, and if it finds one, calls it to handle E. This is called *data-driven programming*.

The `datafun` facility is a simple set of tools for implementing this technique. Obviously, there are three things you have to do to apply data-driven programming to a particular task:

1. Decide where the handler for f will be stored. The obvious choices are in an association list, in a hash table, or on the property list of f.

2. Write a snippet of code to store a handler.

3. Write handlers for all the fs of interest.

The last bit is handled by the `datafun` macro.

`datafun` *Macro* *Location:* YTFM

```
(datafun taskid f
    (defun :^ (---args---)
       ...))
```

defines a function named f-*taskid*. The *taskid* is an arbitrary symbol you choose to represent the task. For instance, if your S-expression walker is counting free variables, you might give it the *taskid* `freevar-count`. Then

```
(datafun freevar-count let
    (defun :^ (e env)
       ...))
```

This defines a function `let-freevar-count` which is to be called (with two arguments `e` and `env`) by the freevar counter.

Let's suppose you decide to store in a hash table `freevar-count-handlers*`. You retrieve the handler for an S-expression beginning `(`*f*`...)` by doing `(gethash` *f* `freevar-count-handlers*)`.

The only remaining bit is to tell the `datafun` macro where to store the handlers. In general the way to do this is by using the same design idea at a "meta-level," supplying a "datafun attacher" handler:

```
(datafun attach-datafun freevar-count
   (defun :^ (id sym fname)
      Code to attach function named  fname  to symbol  sym
       under taskid  id))
```

So we could write

```
(datafun attach-datafun freevar-count
   (defun :^ (_ sym fname)
      (!= (href freevar-count-handlers* sym) (symbol-function fname))))
```

Hash tables and association lists are used so often for data-driven programming that these two cases can be abbreviated. Just write `(datafun-table` *table-var taskid* `&key (size 100))` to allocate a hash table for the given *taskid* with the given size. That is, we could just write `(datafun-table freevar-count-handlers* freevar-count 10)`.

Similarly, `(datafun-alist` *alist-var taskid*`)` to declare a global alist. In the previous example, if we had written `(datafun-alist freevar-count-handlers* freevar-count)`, then the global variable `freevar-count-handlers*` would be allocated (with initial value `()`), and declaring a new handler for symbol *sym* will change the entry for *sym* in `freevar-count-handlers*` to be that handler; of course, if there is no entry, one will be added to the front of the alist.

There are two shorter forms of `datafun`:

- `(datafun` *taskid sym sym′*`)` makes the handler for *sym* be the same as the handler for *sym′*.
- `(datafun` *taskid sym* `#'`*fcn*`)` makes function *fcn* the handler for *sym*.

7.4 Other Functions and Macros

In this section I summarize various "small" functions and macros, in mostly alphabetical order.

`alref`	*Macro*	*Location:* YTFM
`alref.`	*Macro*	*Location:* YTFM

`(alref` a x `[`d`])` is like `(cadr (assq` x a`))`, but if there is no entry for x in the alist a, d (or if d is missing, *False*) is returned. Note that the order of arguments to `alref` is like that of `aref` — table first, then key. (Cf. `href`, below.) `alref.` is like `alref` except that `cdr` of the pair is returned instead of the `cadr`. Both of these macros are `setf`-able.

`bind`	*Macro*	*Location:* YTFM

`bind` is synonymous with `let`, except that it declares all the variables that it binds `special`.

`assq`	*Function*	*Location:* YTFM

`assq` is a synonym for `assoc` with test `eq`.

`build-symbol`	*Macro*	*Location:* YTFM

`(build-symbol [(:package` p`)]` *—pieces—*`)` creates a symbol. The package argument p is evaluated to yield the package where the symbol will reside. If the package argument is missing, the symbol will be in the package that is the value of `*package*`. If p evaluates to *False*, the symbol will be uninterned.

The "pieces" of the `build-symbol` form specify pieces of the name of the symbol. Each "piece" specification x yields a string according to the type of x:

Symbol The name of the symbol

String x; but if x contains alphabetic characters you will get a warning (see below)

any other atom The string obtained by `princ`-ing x

a list `(:<` e`)` The string obtained by `princ`-ing the value of e

a list `(:++` v`)` The string obtained by `princ`-ing the value of `(incf` v`)`

The warning for strings containing alphabetic characters is generated because the characters will be used to produce the name of a symbol, and it is hard to do this in a portable way. In ANSI CL, the string should be uppercase, but in Allegro's Modern CL, it should be lowercase. Using symbols solves the problem.

Example of `build-symbol`:

```
(build-symbol :foo ":" (++ foonum*))
=> |FOO:7| in ANSI CL
   |foo:7| Modern CL
```

assuming `foonum*` is 6 before the call to `build-symbol`; `foonum*` has value 7 afterward.

`car-eq` *Function* *Location:* YTFM

`(car-eq` x y`)` is true if x is a pair whose car is `eq` y.

`classify` *Function* *Location:* YTFM

`(classify` l p`)`, where l is a list and p is a predicate, returns two values: a list of all elements of l that satisfy p, and a list of all the elements that don't.

`debuggable` *Macro* *Location:* YTFM

`(debuggable` K`)`, where K is either -1, 0, or 1, expands into a declaration of compiler-optimization quantities such as `speed` and `debug`. (The exact settings depend on implementation.) The expansion also sets the variable `debuggability*` to K, thus allowing your own macros to expand differently depending on the declared debuggability level. (None of this would be necessary if there were a portable way to find out the current settings of `speed`, `debug`, and company.)

`drop` *Function* *Location:* YTFM

`(drop` n l`)`, where n is an integer and l is a list, returns a new list consisting of all but the first n elements of l, or the last $-n$ elements if $n < 0$. If $n > 0$, `(drop` n l`)` is equivalent to `(take` n' l`)`, where $n' = n -$ *length*(l). If $n < 0$, `(drop` n l`)` is equivalent to `(take` n' l`)`, where $n' = n +$ *length*(l).

`eof*` *Constant* *Location:* YTools, `outin`

The `in` macro (section 4) returns this constant if it encounters the end of the stream being read from.

`false` *Constant* *Location:* YTFM

`false` denotes *False*, i.e., `nil`.

`funktion` *Macro* *Location:* YTFM

`(funktion` s`)`, which may be written `!'`s is equivalent to `(function` s`)` if s is a lambda-expression or `debuggability*` is ≤ 0. If s is a symbol and `debuggability*` is > 0, it's equivalent to `(quote` s`)`. The latter is logically less clean, but allows you to redefine s without finding and fixing every place containing a pointer to its old definition.

`href` *Macro* *Location:* YTFM

`(href` h k `[`d`])` is like `(gethash` k h`)`, except for the argument-order change (see `alref`, above), and the fact that if there is no entry for k in h, d (default *False*) is returned. `href` returns exactly one value in either case. `href` is `setf`-able.

`include-if` *Macro* *Location:* YTFM

In a backquote `,@(include-if` s e`)` expands to the value of e if s evaluates to *True*, otherwise to nothing. E.g., `` `(foo ,y ,@(include-if (p x) x)) `` expands like `` `(foo ,y ,x) `` if `(p x)` evaluates to *True*, and to `` `(foo ,y) `` otherwise.

`is-list-of` *Function* *Location:* YTFM

`(is-list-of` x p`)`, returns *True* if x is a list of objects that all satisfy predicate p.

`is-whitespace` *Function* *Location:* YTFM

`(is-whitespace` c`)` is *True* if and only if character c is whitespace.

`lastelt` *Function* *Location:* YTFM

`(lastelt` l`)` is equivalent to `(car (last` l`))`.

`len` *Function* *Location:* YTFM

`len` is equivalent to `length`, except that it works only on lists, not arbitrary sequences.

`make-Printable` *Function* *Location:* YTFM

`(make-Printable` f creates an object whose printed representation on stream s is `(`f s`)`. Such an object may sound useless, but it is convenient for defining special "marker" objects that indicate that some condition is true. For example, a language interpreter might return an error flag defined as

```
(defconstant +err-flag+
   (make-Printable (\\ (srm) (out (:to srm) "#<Error during evaluation>"))))
```

The constant `eof*` is a Printable.

`memq` *Function* *Location:* YTFM

`memq` is a synonym for `member` with test `eq`.

`multi-let` *Macro* *Location:* Ytools, `multilet`

`(multi-let` *clauses —body—*`)` is a cross between `multiple-value-bind` and `let`. Each *clause* is of the form

`(`*varspec exp*`)`

but each *varspec* is either a single variable name, as in `let`, or a list of variables as in `multiple-value-bind`. Example:

```
(multi-let ((x (foo))
            ((y z)
             (baz u v 3)))
   (* (+ x y) z))
```

`x` is bound to the single value of `(foo)`, and `y` and `z` are bound to the two values of `(baz u v 3)`.

There is a crucial difference between `multi-let` and `multiple-value-bind`. The latter doesn't care whether the number of variables being bound is the same as the number of values returned; it discards values or introduces `nil` values to make everything match. Many hard-to-track-down bugs are produced by this behavior. If `debuggability*` (see below) is ≥ 0, `multi-let` will signal an error if the two don't match up. If `debuggability*` < 0, `multi-let` will expand into more efficient code that doesn't check for alignment of the variables and values.

`nodup` *Function* *Location:* YTFM

`(nodup` l `[:test` p`])` returns a copy of list l with duplicates removed. The equality test is p, default `eql`.

`occurs-in` *Function* *Location:* YTFM

`(occurs-in` x r`)` is true if x occurs as a subtree or leaf (`eql`-tested) of S-expression r.

`->pathname` *Function* *Location:* YTFM

`(->pathname` x`)` converts x to a pathname, if necessary and possible. It converts strings in the usual way, and converts symbols by converting their names, with case suitably adjusted. Once it has a string, it expands YTools logical pathnames, so that the result is a regular Common Lisp pathname.

`pathname-get` *Function* *Location:* YTFM

`(pathname-get` p s`)` is the value of property s of pathname p. It can be changed using `setf`.

`printable-as-string` *Function* *Location:* YTFM

`(printable-as-string` s`)` creates an object that prints as the string s. Equivalent to `(make-Printable (\\(srm) (out (:to srm) (:a` s`))))`.

`series` *Function* *Location:* YTFM

`(series [`l`]` h `[`i`])` is the list of numbers `(`l, $l+i$, $l+2i$, `...`, $l+\lfloor\frac{(h-l)}{i}\rfloor i$`)`. If i is absent, it defaults to 1. If in addition l is absent, it defaults to 1. Examples:

```
(series 10 20 3) ⇒ (10 13 16 19)
(series 10 22 3) ⇒ (10 13 16 19 22)
(series 10 20)  ⇒ (10 11 12 13 14 15 16 17 18 19 20)
(series 10)   ⇒ (1 2 3 4 5 6 7 8 9 10)
```

`shorter` *Function* *Location:* YTFM

`(shorter` l n`)`, where l is a list and n is an integer, returns the length of l if that length is $< n$, and *False* if it isn't.

`symno*` *Variable* *Location:* YTFM

A global variable useful as a counter in constructions such as `(build-symbol foo- (:++ symno*))`. Of course, you can use your own counters if it yields more intelligible symbols.

`take` *Function* *Location:* YTFM

`(take` n l`)`, where n is an integer and l is a list, returns a new list consisting of the first n elements of l, or the last $-n$ elements if $n < 0$.

`true` *Constant* *Location:* YTFM

`true` is a constant denoting *True*, i.e., `t`.

8 Downloading the Software and Getting it Running

The YTools package is available at my website
`http://www.cs.yale.edu/homes/dvm`.
You download a tar file that creates two subdirectories. Let's suppose you expand it in a directory `~/prog`; you'll get `~/prog/ytload/` and `~/prog/ytools/`. The former directory contains basic code for loading YTtools (and other systems, which don't concern us here). Then put the following in your Lisp initialization file (e.g., `.clinit.cl` in Allegro Common Lisp):

```
(load "ytload-dir/ytload.lisp")
(setq ytools::config-directory* "~/")
(setq ytools::ytload-directory* "~/prog/ytload/")
```

The `config-directory*` should be your home directory if you are using your own private copy of YTools. If you're installing it for use at a site, use some other appropriate directory. I'm assuming that your Lisp can accept "`/`" as a directory delimiter; if not, change the pathnames to something legal.

`yt-install` *Function* *Location:* YTFM
`yt-load` *Function* *Location:* YTFM

Once these lines have been executed, you can install a system by executing `(yt-install :`*`sysname`*`)`; the YTools package itself is loaded by executing `(yt-install :ytools)`. You can also load subsets and supersets of the package, as described in section 9.9. The only two systems discussed here are `:ytools` and `:ytfm`, but others can be downloaded from my website.

All the installation process does is prompt you for the values of various global variables, which are then stored in a file named `ytconfig.lisp` in the configuration directory. . You can edit the values in the file whenever and however you want. You cannot, however, easily introduce new variables into the file. After the system is installed, you load it on all subsequent occasions by typing `(yt-load :`*`sysname`*`)`. Actually, it is unnecessary to call `yt-install` explicitly; if you try to load an uninstalled system, `yt-load` will install it first, after asking if that's what you want to do. `yt-install` and `yt-load` are defined in the `:cl-user` package, and exported from the `:ytools` package.

The `:ytools` package is analogous to the `:cl-user` package. You can do all your work there, but for serious projects you will define one or more packages that "use" the `:ytools` package, and you will encounter the following issue: The basic macros `defun`, `defmacro`, and `eval-when` are shadowed in `:ytools` (so that the ignorable variable _ can be handled properly, and so that `eval-when` understands the `:slurp-toplevel` symbol; see section 9.4). So, if your package uses both `:ytools` and `:common-lisp`, as it certainly will, there will be a conflict between the two versions of each of these symbols. You may resolve it either way you like, but if you take the standard versions (from `:cl-user`), you will not be able to use "_"; why would you want to do that? Instead, write

```
(defpackage :mypkg
   (:use :common-lisp :ytools)
   (:shadowing-import-from :ytools
            ytools::defun ytools::defmacro ytools::eval-when)
   . . .)
```

Of course, if you want the standard versions of these facilities, import them from `:cl-user` instead.

9 The YTools File Manager (YTFM)

YTools provides utilities, collectively called the "YTools File Manager," or YTFM, for loading and compiling files. In particular, it keeps track of whether a file needs to be recompiled before it is loaded, and what files depend on what other files. Most CL implementations provide extensions to the built-in `load` function, plus some variant of `defsystem`, to accomplish these tasks. `defsystem`, like `make` in Unix, puts all the information about a group of related files in one place.

YTFM takes a somewhat different approach, in which each file starts with a record of what other files it depends on. There are still modules, but a module is simply a name for an expression that loads a group of files. If you find this approach misguided or unnecessary, you may want to skip to section 9.9. However, one thing to be aware of is the meaning of the "Location" information in the presentation of YTools features. Instead of loading the entire `:ytools` system, one can load just the YTFM, plus the pieces of YTools that you want.

If the "Location" specified for a YTools tool is *"YTFM"*, then the macro is loaded as soon as you do `(yt-load :ytfm)`.

If the location includes a file name F (in `typewriter` font), then the function or macro is to be found in the file `%ytools/`F`.lisp`. If you want just a few tools, evaluate `(yt-load :ytfm)`, then `(fload %ytools/ `F`)`, for each file F you want. Of course, if the file depends on other files, they will be loaded, too. The details of what `fload` does are described in the next section.

Most tools have "YTools" in their "Location" information, which means the tool is loaded when you execute `(yt-load :ytools)`. Some have just a file name, meaning that in addition to loading `:ytools`, you have to `fload` the file in question.

9.1 Loading files: `fload`

`fload` *Macro* *Location:* YTFM

To make the YTFM work, one must use the `fload/fcompl` facility instead of the usual `load`, `compile-file`, and such. The format of `fload` is:

`(fload [`*fload-flag*`]* ---`*filespecs*`---)`

where *filespecs* is a list of directories and files. Example:

```
        (fload "/home/smith/prog/" macros support
               %utils/ mailhack)
```

when this file is being compiled, loads files

```
/home/smith/prog/macros.fasl
/home/smith/prog/support.fasl
%utils/mailhack.fasl
```

(assuming that `fasl` is the appropriate extension for an object file in the host Lisp system; see section 8). If a file has already been loaded (and not changed subsequently), it is skipped, unless this behavior is overridden by the `-f` or `-c` flags. If an object file doesn't exist, or is not up to date, the YTools system decides whether to (re)compile its source file, or use an old binary if there is one. It usually asks the user, but its exact behavior depends on the global variable `fload-compile*` (see below).

After the `support` file has been loaded, `fload` loads `%utils/mailhack.fasl`. However,`%utils/` is not really the name of a directory. The `%` in front of it indicates that it is a *YTools logical pathname*, which normally expands into a directory, as specified by `def-ytools-logical-pathname`, described in section 9.2.

If the filespecs are omitted, then the ones used on the previous call to `fload` are re-used. The values set by the flags (see below) are recovered as well.

This general behavior is modified by various flag arguments to `fload`, and a global variable, `fload-compile*`. The variable takes on one of these values:

- `:compile`: Always compile, without asking the user.
- `:source`: Never compile. Load the source file if it is newer than the object file.
- `:object`: Never compile. Load the object file even if it is older than the source.
- `:ask` (the default): Ask the user whether a file should be recompiled. The user has four possible responses:
  1. `y`: Yes, compile it
  2. `n`: No, don't. YTools will follow up by asking whether to load object or source, and whether that decision should be permanently associated with the file.
  3. `+`: Yes, and set `fload-compile*` to `:compile` from now on.
  4. `-`: No, and set `fload-compile*` to `:object` from now on.

Exactly what happens when a file is compiled is discussed in section 9.4.

The possible flag arguments for `fload` are:

- Flag `-f`: Force the file to be loaded even if has been loaded already and not changed since.
- Flag `-c`: Force the file to be recompiled and loaded even if `fload` normally would not do one or both of these operations.
- Flag `-`: Clear `-f` and `-c` flags. This operation makes sense because the `-f` and `-c` flags are remembered as long as you keep evaluating `fload` calls with no explicit files arguments. So, after, e.g., `(fload -c)`, which causes the most recently `fload`ed files to be recompiled and reloaded, `(fload)` will do the same thing; the `-c` is "sticky." Writing `(fload -)` causes the file to be reloaded *without* mandatory recompilation.
- Flag `-a`: If the user has previously been asked whether to load the source or object version of the file, and in that dialogue said to do the same thing from now on without asking, then discard that information and ask again when necessary.

9.2 YTools Logical Pathnames

The *filespecs* in a call to fload are interspersed directories and file names. These can be strings or symbols. If you type `(fload foo)`, then in ANSI Common Lisp `foo` will actually be read as `FOO`, but YTools can figure out that the intended file name is `"foo"` from its knowledge of the usual case of file and symbol names.

The character `'%'` has a special meaning in a filespec. It signals the beginning of a *YTools logical pathname*. These have nothing to do with Common Lisp logical names, although they play a similar (and complementary) role, allowing the physical location of a file to vary from implementation to implementation without changing references to it. A YTools logical name is defined by executing

```
(def-ytools-logical-pathname name pathname [object-code-loc])
```

After this, any reference to `%`*name* is interpreted as *pathname*. For instance, after

```
(def-ytools-logical-pathname ded "~/prog/deduction/")
```

executing `(fload %ded/ unify)` will try to load `~/prog/deduction/unify.lisp` or its object file. The optional *object-code-loc* argument specifies where to put object files compiled from source files in this directory. If omitted, they are put in the same directory. A relative pathname such as `"../bin/"` is interpreted as follows: Suppose we are in a directory ending `.../`c`/`d`/`. We ascend one level, remembering d, then descend one level (through `bin`), then one more, through d again, yielding `.../`c`/bin/`d`/`. Analogous operations are performed if you have to ascend more than one level.

Note that the slashes at the end of pathname definitions are meaningful and required. A form such as `(def-ytools-logical-pathname foo "x.lisp")` is okay, but simply defines `foo` as a synonym for `x.lisp`. If you want `foo` to stand for a directory (the usual case), you have to put the slash in.

The function `filespecs->pathnames` converts a filespecs list to a list of ordinary Lisp pathnames. E.g., `(filespecs->pathnames '(%ytools/ hunoes fileutils))` might return

```
(#P"/usr/local/ytools/hunoes" #P"/usr/local/ytools/fileutils")
```

The pathnames have no defaults filled in, and may or may not point to files that already exist.

Note on directory delimiters: Different OS's have different directory delimiters. In my experience, most Lisps allow a forward slash even if the underlying OS uses some other character. Thank God. One of the first queries in the installation process for YTools is for the value of the "directory delimiter" on your computer. Try typing '`/`' and switch to the actual delimiter for your filesystem only if '`/`' doesn't work out.

9.3 File Dependencies: `depends-on`

YTools expects that near the front of each file will appear an expression exemplified by the following

```
        (depends-on (:at :run-time) %ded/ unify index)
```

Suppose this form occurs in the file `thisfile.lisp`. The `depends-on` facility tells YTools that when this file is loaded, two other files should be loaded. They are both found in a directory identified by the YTools logical name `ded`. So if you execute

```
 (fload thisfile)
```

`fload` will check whether `unify.lisp` has been loaded in an up-to-date form, and if not will compile it if necessary and load it, and similarly for `index.lisp`.

The general form of `depends-on` is

`(depends-on` ---*dep-groups*---`)`
where each `dep-group` is of the form
`[`*time-spec*`]` ---*filespecs*---

and *filespecs* are as described for `fload`, above. `depends-on` declares that the given *filespecs* are needed when the file containing the `depends-on` is processed. The *time-specs* state exactly when they are needed, and will be discussed in detail below.

To understand the *time-specs*, I first need to review the YTools model of file dependency. Every file is divided into a *header* and a *body*. The header is the few lines at the top of the file that identify its package, what symbols it exports, what files it depends on, and so forth. There is no need to indicate the end of the header explicitly, but YTools assumes that the end occurs as soon as it finds a piece of code that doesn't seem to belong to the header (e.g., a function or datatype definition). That means all "headerish" material should appear at the front of the file.

As a file f_1 is `fload`ed (whether compiled or not), if YTools finds `(depends-on ... (:at :run-time) ...` f_2 `...)` in it, then it `fload`s f_2 before resuming the load of f_1.

Similarly, if f_1 contains a `(depends-on ... (:at :compile-time) ...` f_2 `...)` form, then when f_1 is compiled, f_2 will be loaded. Time-specs are described in complete detail in section 9.5.

9.4 Compiling and Slurping Files

`fcompl` *Macro* *Location:* YTFM

A file is compiled when `fload` sees that its object version is out of date (see section 9.1), or when the following function is called and the object is out of date:

`(fcompl [`*fcompl-flag*`]* ---`*filespecs*`---)`

Here *filespecs* have the same format as for `fload`.
The flags for `fcompl` are:

- Flag `-f`: Force the file to be compiled even if its source has not been changed since the last time it was compiled.
- Flag `-`: As for the same flag in conjunction with `fload`, this causes `fcompl` to "erase" the `-f` flag.

After a successful compilation, `fcompl` will normally try to load the new object file. Exactly what it does depends on the value of the global variable `fcompl-reload*`:

- If it's *False*, the new object file is never loaded.
- If it's `:ask` (the default), it will ask if you whether to load the object file.
- If it's any other value, the new object file is always loaded.

The process of compiling a file, whether by `fload` or `fcompl`, differs from the standard Lisp model in that the file and the files it depends on are examined in a preprocessing pass before the regular Lisp compiler looks at them. This preprocessing is called *slurping*.[4] Most of the time, slurping just examines the *header* of a file to extract file dependencies. The header is the portion of the file at the beginning that describes the relationship of this file (and its packages) to other files (and their packages). It is intended that users not have to focus too hard on the definition of the header, because Lisp files will just naturally organize themselves the right way, but here is the technical definition: the header is the segment at the beginning of the file that consists entirely of *headerish* forms, those whose function or special-operator is one of `depends-on`, `in-package`, `defpackage`, `declaim`, `eval-when` (sometimes), or `slurp-whole-file`. (The novelties in this list are explained later in this section.) Of course, a form `(`progn $e_1, \ldots, e_n$`)` is counted as headerish if each e_i is, and a macro form is headerish if it expands into something headerish. The macros `in-header` and `end-header` can be used to control the exact boundary of the header. See below.

For example, suppose file `foo.lisp` contains

[4] It's reminiscent of the Java process, except the the Java compiler examines class files, not source files.

```
    (depends-on (:at :run-time) baz)
```

When `foo.lisp` is compiled (say, by writing `(fcompl foo)`), the first thing that happens is that YTools checks to see if any of the files `foo.lisp` depends on has changed since the last time it was loaded, by slurping `foo.lisp`, `baz.lisp`, and all the other files supporting `foo`. If it finds any, it consider recompiling and reloading them, meaning it consults the variable `fload-compile*`, and may end up asking the user whether to recompile, reload, or both.

This may take some getting used to. The YTools model of file dependency and compilation is not the usual Lisp approach, which is roughly: To compile a set of files, load them all in as source files, then compile them all, then load them all in as object files. (A well organized file set can be compiled by a more incremental process, in which each file is loaded as source, compiled, then loaded as object.) YTools separates compilation and loading in a more elegant way. When a file is compiled, normally the files it will depend on at run time are only slurped. Then when it is loaded, those same files are loaded. In this way, it is possible to recompile a system without ever loading any of its files.

The basic assumption of slurping is that during compilation of a file F, the only thing one needs from a supporting file S is information about the supporting file's supporters. But there are a few other possibilities:

1. If file S contains a macro definition, it is usually the case that the macro should be available during the compilation of F. If the macro calls functions defined in S, those should be available, too.

2. In the extreme case, S is nothing but a set of macros that are needed at compilation time, and not needed at all when F is actually loaded.

The first case is handled by ensuring that when the header of a file is slurped, so is the rest of the file. This is accomplished by a local declaration, which can take two forms:

1. Put the form `(slurp-whole-file)` anywhere in the header.

2. End the header with the form `(end-header :continue-slurping)`.

The latter looks prettier, but the former is somewhat more flexible in practice. The file manager will print a message "`Slurping` *filename*..." when it starts to slurp the whole file, and a message "...`Slurped` *filename*" when it finishes. *No message is printed* when the header alone is slurped.

The only other flag that can come in an `end-header` form is `:no-compile`. Its presence tells YTools that this file should never be compiled.

The macro `end-header` can be used to declare the end of the header of a file. This may clarify the structure of some files, but is normally necessary only when it carries a special flag like `:no-compile`. The header normally stops when a non-headerish form is encountered. This fact can lead to frustrating situations where YTools thinks it has left the header before you think it has. To debug in such a case, set the variable `end-header-dbg*` to *True*, and YTools will tell you when it thinks it has reached the end of each header it examines.

To force a set of forms to be "headerish," use `(in-header` $e_1, \ldots, e_n$`)`. It's equivalent to `(progn` $e_1, \ldots, e_n$`)`, when compiled, but when slurped it will not end the header, no matter what each e_i does.

In YTools, `eval-when` has been augmented with a new "situation," `:slurp-toplevel`. That is,

```
   (eval-when ( ... :slurp-toplevel)
       ---forms---)
```

causes the *forms* to be evaluated when the `eval-when` is encountered during the slurping of a file. An `eval-when` form is "headerish" if and only if it includes `:slurp-toplevel` among its situations.

The form

```
         (needed-by-macros ...)
```

`needed-by-macros` is equivalent to

```
    (eval-when (:compile-toplevel :load-toplevel :execute :slurp-toplevel)
        ...)
```

That is, the forms nested within `needed-by-macros` are evaluated whenever the `needed-by-macros` is encountered, even during slurping. The reason for the name is that a typical use for this construct is in a file that contains a macro definition, plus some subroutines used by the macro. Slurping the file normally causes the macro definition to be taken but other function definitions to be ignored. Wrapping the subroutine definitions with `(needed-by-macros ...)` fixes the problem.

When compiling, you will see messages announcing when files are being compiled and loaded. You will also see messages announcing when they are being slurped (although, as explained above, not when only their headers are being examined). All the messages can be suppressed by setting `fload-verbose*` to *False*.

9.5 Time Specs in `depends-on`

The second case mentioned above is where file F has a direct supporter S that consists *entirely* of macros, constant definitions, and the like. In this case, you probably want S to be loaded when F is compiled, but not when the object version of F is loaded.

A *time-spec* in a `depends-on` is of the form

```
  (:at [:run-time] [:compile-time] [:slurp-time])
```

Suppose the form

```
(depends-on ... (:at t_1 ... t_n) ---filespecs--- ...)
```

occurs in file F. The time-specs t_i specify when the files G denoted by *filespecs* are to be compiled and/or loaded and/or slurped.

- `:run-time` means that when F is compiled, the G are to be slurped; when F is loaded, the G are to be loaded; when F is slurped, the G are to be slurped.
- `:compile-time` means that when F is compiled or slurped, the G are to be loaded; when F is loaded, nothing is to be done with the G. If you write `(:at :run-time :compile-time)`, then you get the "union" of these effects: the file is loaded at both run time and compile time. Note that all this loading is done by `fload`, so only the first of multiple calls to load it have any effect, unless the file changes.
- `:slurp-time` means that when F is slurped, the G are to be loaded. This is hardly ever necessary, unless F uses read macros that are defined in G.

The forms `(:at :run-time)` and `(:at :compile-time)` may be abbreviated `:at-run-time` and `:at-compile-time`. If the time spec is omitted, that's equivalent to `(:at :run-time :compile-time)`.

9.6 Long-range File Dependencies

Suppose `file1` depends on `file2`, which depends on `file3`, both dependencies being `:at-run-time`. Suppose `file3` changes and the user `floads` `file1`. There are two possibilities to think about: `file3` should probably be reloaded, and `file1` and `file2` may need to be recompiled if either of them used a macro defined in `file3`. YTools will deal with all these possibilities. That is, when a file is `fload`ed, YTools rebuilds the tree of dependencies the file is the root of, and recompiles and reloads all the files between the root and a changed file. Actually, it will normally query the user about each case; if this gets to be tedious, just set `fload-compile*` to `:compile` (explicitly, or by responding "+" to a query).

If you have just reloaded a file and are not sure what other files may now need to be changed, use `fload-recheck`, defined in section 9.8.

9.7 YTools Modules

A *module* is, intuitively, a set of files that work together.

The logical pathname `%module` has a special meaning. It expects to be followed by the names of entire modules rather than file names. For example:

```
  (fload %module/ utilities graphplan)
```

loads in the module `utilities` followed by the module `graphplan`.

A YTools module is a device for giving a short name to a set of files, but the actual definition is a bit more complicated. The best way to think of a module is as comprising two components:

1. A tiny virtual file, which normally contains `depends-on` statements.
2. An *expansion*, a set of forms that behave as if they occurred in the file that depends on the module, at the point where the dependency is declared.

To see the distinction, suppose that the file `amazing.lisp` contains this statement:

```
(depends-on (:at :run-time) %module/ specutils)
```

Now suppose that `specutils` contains three forms:

```
   (depends-on (:at :run-time) %utildir/ foundation objorp whizbang)
   (def-class C1 ...)
   (defmacro buzz (...) `(car ,(C1-b ...)))
```

The situation is (almost) as if there is a file `specutils.lisp` containing only these three forms. Think of `specutils` as being a *virtual file*.

Now suppose that we want to include in `specutils` the declaration `(slurp-whole-file)` (see secref-fcomplthe section on compiling files). There is an ambiguity, because the declaration may apply to the virtual file `specutils`, or to the file `amazing.lisp` that depends on `specutils`.

The ambiguity is resolved simply: To make it apply to `amazing.lisp`, put the declaration in the expansion of the module, so that it will actually appear in the top level of `amazing.lisp`. To make it apply to the module itself, put it in the contents of the module. You can, of course, do both.

To define a module, write

```
   (def-ytools-module name
      [(:contents ---forms---)]
      [(:expansion ---forms---)])
```

For example, the `specutils` module might be defined thus:

```
    (def-ytools-module specutils
       (:contents
          (depends-on (:at :run-time) %utildir/ foundation objorp whizbang)
          (def-class C1 ...)
          (defmacro buzz (...) `(car ,(C1-b ...))))
       (:expansion
          (slurp-whole-file)))
```

indicating that the file to slurp the entirety of is `amazing.lisp`, or any other file that depends on `specutils`.

There is one difference between `specutils` and a file with the same contents. In section 9.6, I said that YTools searches the entire tree of file dependencies to find files changes in which might make it necessary to recompile the current file. There is one exception: it will not search through a module boundary. This feature is based on the assumption that a module gets created after its components have been designed, debugged, and redesigned. Furthermore, someone who just wants to use the module might have no idea what to make of queries about its components.

If you really want a module to be completely transparent, move all the forms in its "contents" field to its "expansion" field, as in the following definition of the "specutils" module:

```
(def-ytools-module specutils
   (:expansion
      (slurp-whole-file)
      (depends-on (:at :run-time) %utildir/ foundation objorp whizbang)
      (def-class C1 ...)
      (defmacro buzz (...) `(car ,(C1-b ...)))))
```

Now `(depends-on %module/ specutils)` behaves exactly like

```
(slurp-whole-file)
(depends-on (:at :run-time) %utildir/ foundation objorp whizbang)
(def-class C1 ...)
(defmacro buzz (...) `(car ,(C1-b ...)))))
```

The only built-in module is named `ytools`. Any file that wants to use all the YTools should have

```
(depends-on %module/ ytools)
```

in its header. The YTools module does not include every facility described in this document. Optional facilities occupy files in the YTools directory, and you indicate a dependence on one of them by writing

```
(depends-on %ytools/ file)
```

I mention in passing my convention of using the name M`.lsy` for a file that defines a module M, plus ancillary facilities such as packages and logical pathnames. Generally speaking, to make a module loadable you have to first load its ".lsy" file.

9.8 Other file-management facilities

The following are not part of the YTools module, but may be found in the file `fileutils` in the YTools directory. Either `fload` the file when you need it (`(fload %ytools/ fileutils)`) or put `(depends-on :at-run-time %ytools/ fileutils)` in the header of any file that requires one of them.
`fload-versions`: If is often the case during system debugging that you want to use an experimental version of a file. If you write

```
(fload-versions directory/ (gps gps-experimental))
```

then wherever `directory/gps` would normally be loaded (or slurped, or compiled), `directory/gps-experimental` will be used in its place.

In general, the arguments to `fload-versions` are just like those to `fload`, except that lists (of 1 or 2 elements) appear instead of the files. The first element of the list is always the name of a file. The second element (if present) determines the new version of that file. The possibilities for the second element are:

- A symbol (other than a keyword or "`-`"): The symbol's name, with case suitably modified, is the new version.
- A string or a keyword: The new filename is the old with the string (or keyword symbol's name) appended.
- *Absent*: Treated as if the value of the global variable `fload-version-suffix*` occurred instead. This value is initially `:-new`. It should always be a string or keyword.
- `-`: All versioning information for the file is discarded. It reverts to its original self.

`fload-recheck`: In the normal course of loading a file, `fload` will check the entire dependency tree below it. So if it depends on a file $F2$ that depends on a file $F3$, and $F3$ has changed, it will offer to recompile and reload $F2$.

If you want to be even more obsessive, execute `(fload-recheck)`. This will find every loaded file that might have to be reloaded and/or recompiled, and ask you about them.

Note that both of these scenarios get to be boring quickly. You'll be asked about whether to recompile dozens of files. Most probably don't need to be recompiled, but it takes longer to figure that out than to recompile them. So set `fload-compile*` to `:compile`. You can do that by typing `+` when asked whether you want to compile a file. Remember to set it back to `:ask` when the recompilations are done, if that's the long-term behavior you want.

9.9 *Is It Possible to Avoid the YTools File Manager?*

It may sound like a heavy investment of effort to use the YTools file manager, especially when it's taken you years to learn some version of `defsystem`. Even worse, it may be that once you start down this slippery slope, you'll end up having to make tiny edits in thousands of files.

Well, rest easy. All of the little extras in YTools files, such as `end-header`, `depends-on`, and such, are defined as harmless macros. You may get a warning message or two about the `fload` environment not being present, but all the files should load properly. If you want the whole system to just go away, set the global variable `depends-on-enabled*` to nil.

If you decide that all you want from YTools is, say, `repeat` and `out`, just copy them and compile the copies. Most compilers will tell you which functions are used in `repeat.lisp` but not defined there. The definitions can be found either in one of the core YTFM files (`base`, `datafun`, `pathname`, `module`, `slurp`, `files`, `depend`, all with extension `.lisp`), or in a file that `repeat` depends on. Just copy them to your version of `repeat` and you're all set.

A Some Notational Conventions

A.1 Upper and Lower Case

I like running my code in Allegro's "modern" mode, in which the case of symbols is preserved when they are read and printed. All my code is written in such a way that it ports without change to an ANSI CL in which symbols are converted to upper case when read.

Most of my code is lower case. I use upper case only as the first letter of a datatype name (and sometimes subsequent letters if the datatype name is an acronym), and only in functions defined as part of the datatype definition. That is, I might write

```
    (defstruct Foo a b c)
```

to define a datatype `Foo`. As a consequence, the constructor for this type is named `make-Foo`, the predicate is `Foo-p`, the slot reader for slot `a` is `Foo-a`, and so forth. But if I have a function that transmogrifies objects of this type, it's called `foo-transmogrify`. That is, the *only* functions that have the upper-case "F" are the ones automatically defined by `defstruct`. *Exception:* If I need an alternative constructor for `Foo`s, I might name it `new-Foo` or `create-Foo`.

These conventions are close to those used by Java and Haskell. However, I do *not* use the "case foothills" style such as `fooCacheIfMarked`, partly because Emacs can't see the word boundaries, and partly because in ANSI CL it is unreadable. Instead I use hyphens as exemplified above: `foo-cache-if-marked`.

A.2 Special Characters

YTools has its own readtable (`ytools-readtable*`),[5] in which two macro characters are reserved: exclamation point and question mark. Exclamation point is used for several purposes (discussed as they come up below).

Question mark is used only as the printed representation of *qvars*, which are used in applications of pattern matching. The object `?x` is of type `Qvar`. You test whether an object `obj` is of this type by evaluating `(is-Qvar ob)`; you extract the symbol (i.e., `x`) by evaluating `(Qvar-sym ob)`. To make one, evaluate `(make-Qvar 'x '())`. All the details are given in section 3.

Exclamation point is reserved for use in packages built on top of YTools. YTools itself uses it for two purposes:

1. `!()`: This expression is read as `(empty-list)`, a macro that expands to `'()`. The pretty-printer prints `(empty-list)` as `!()`, thus hoping to reduce somewhat the number of occurrences of the ambiguous `nil` `http://www.cs.yale.edu/homes/dvm/nil.html` in the universe. Actually, you can write `!(`*anything*`)`; I use `(let ((l !(Symbol))) ...)` to suggest that `l` is a list of symbols, initially empty.

2. `!"..."`: This is an ordinary string, except that any substring of the form "~newline whitespace" is deleted. Helpful for long string constants that start at a column awkwardly far to the right.

3. `!'`s: An abbreviation for `(funktion` s`)`, q.v.

Unless stated otherwise, `"!"` behaves like an ordinary symbol constituent. In particular, `"!="` is an ordinary symbol with name `"!="`; see section 3.

In addition, there are a couple of other usages that may look like the invocation of macro characters, but aren't:

- `(\\ ...)` is an abbreviation for `#'(lambda ...)`.
- "_" may be used in most contexts where a bound variable is expected; it denotes a variable whose value is ignored. These contexts are: `\\`, `defun`, and as indicated in the rest of this document. Example: `(defun foo (a _ c) ...)` is equivalent to `(defun foo (a b c) (declare (ignore b)) ...)`. By the way, you can just write `(ignore b)` instead of `(declare (ignore b))`.
- `%` as a marker for logical pathnames in `fload` et al. Nothing funny happens at read time; macros that expect filespecs just look for symbols whose names begin with "`%`."

A.3 Coding style

Here are a few of my hard-won prejudices on the subject of how to write good, intelligible code.

Function Names: Many functions can be considered to take an object of type Y, perform operation P on it, and return an object of type Z. I tend to name such a function Y-P-Z. I prefer `list-copy` to `copy-list`. Functions that "coerce" an object to a different type have the operation `->` in their names, as in `list->vector` or `->pathname`. Note that in the last case the operation is not preceded by any type; that indicates that it should work on any object.

My predicates tend to be distinguished by the occurrence of auxiliary verbs such as "is" and "has," as in `foo-is-empty` or `bar-has-no-children`; or by the use of an adjective instead of an operation, as in `foo-empty` or `bar-greater-than`. I never end a predicate name with the letter `p` or a question mark.

[5] I have been marking global variables with a single asterisk since before the convention existed of putting an asterisk at both the bow and the stern of a global-variable name. I prefer my convention. Using a single asterisk has the advantage of making it obvious which global variables belong to YTools and which to Lisp itself. I do write constants as +*name*+, because an isolated plus would look like it had something to do with addition.

A "category tester" is a predicate that takes any object at all and returns *True* if it belongs in the category. For instance, the function that tests whether an object is of type `Bar` is called `is-Bar`. Sometimes what you require instead is a function that tests whether an object in one category belongs to a subcategory as well. The name of such a function might start with the wide category and end with the narrow one, as in `foo-is-bar`. The clue is whether the verb has anything in front of it.

Dynamically bound (special) variables: Avoid them wherever possible. Here are a few unavoidable uses:

- As repositories of global debugging flags or other information.
- As variables that must be visible through a function you didn't write. For example, if you want to communicate with a macro during file compilation, you have to bind some special variables for the macro to read, because there is no other way to communicate through `compile-file`.
- As places to hold global tables. However, you should think hard about what exactly needs to be global.

Let me amplify on that last point. Suppose you are writing a program to process natural language. It's natural to define a global variable `grammar*`, another one `lexical-rules*`, and so forth. The problem with this approach is that switching between grammars and such requires resetting all those variables. A better approach is to have a single table `nl-regimes*` that contains definitions of different languages. A language definition consists of a grammar, some lexical rules, etc. Procedures to parse a sentence or produce an answer to a question can be passed as arguments the entire language definition or just the parts each procedure needs. The one remaining special variable is now less central to the operation and less likely to be the place where something goes haywire.

The idea of passing a few extra arguments sounds like it could get out of hand. One way to avoid an explosion in the number of arguments is to package a few of them together using a new datatype and pass the package. Of course, this strategy makes sense only if the arguments belong together.

Use `cond` I don't know why people like `if`, `when`, and `unless`. I revise my code starting thirty seconds after I first start writing it, and I find the revisions required to be particularly annoying for all conditionals except `cond`. By supplying one extra level of parens, it allows you to change your mind about which tests go first or what happens when a test is true without a huge fuss.

John Foderaro, in his coding-style recommendation
`http://www.franz.com/~jkf/coding_standards.html`
states:

> I've found that the key to readability in Lisp functions is an obvious structure or framework to the code. With a glance you should be able to see where objects are bound, iteration is done and most importantly where conditional branching is done. The conditionals are the most important since this is where the program is deciding what to do next. Bugs often occur when the conditional can't handle all possible cases. If you're planning on extending the function you need a clear idea of the possible inputs it's willing to accept and what you can assert to be true for each branch of the conditional.

I agree entirely.

He then goes on to recommend using his own `if*` macro.

> The if* macro along with proper indenting support in the editor makes glaringly apparent the structure of conditionals. They stand out like a sore thumb. Furthermore one can easily extend an if* conditional adding expressions to the then or else clauses or adding more predicates with elseif. Thus once you've laid out the conditional you can easily extend it without changing the expression itself (contrast that to having to go from when to if to cond as you grow the conditional using the built-in Common Lisp conditional forms).

To each his own. It seems to me that `cond` has all the virtues of `if*`, but far be it from me to criticize a fellow macro-hacker! However, read on.

A Couple of Observations about Macros: Many macros have auxiliary symbols that play a crucial role in "parsing" calls to the macro. To see examples, look at `let-fun` (section 2.1). It has two such symbols: `:where` and `:def`.

A key question in designing a macro is how many of these symbols, which I call *guide symbols*, to use. At one extreme we find a construct like `do`, which has no guide symbols at all, and distinguishes all its subparts using parentheses. Many people (including me) find its syntax confusing at first, then merely ugly. At the other extreme we find the "complex" form of `loop`, which has dozens of guide symbols, enough to form a little subgrammar. This subgrammar has few parentheses, which raises issues about the precedence of the guide symbols, the order in which subexpressions are executed, and ultimately of how the different features combine semantically. In fact, I never use the complex version of `loop`; `repeat` does everything I want and its semantics are quite clear.

The design of a good macro is determined in the end by aesthetic decisions, but I would propose a few principles:

1. Keywords are used for two purposes in the built-in constructs of Lisp: as keyword arguments, and as "clause markers." It's a good idea to limit them to these two purposes. That is, a keyword `:gizmo` may either come before its argument, as in

   ```
   (fcn ...  :gizmo arg ...)
   ```

   or as the first element of a list structure

   ```
   (big-construct con-th
       (:ying ...)
       (:gizmo
            ...
            (arg ...)
            ...lots of other stuff...)
       ...)
   ```

2. Guide symbols should be located in the keyword package. One reason is that they stand out visually; someone reading an occurrence of the macro can instantly see that *this* is a guide symbol, and *that* is variable. Another is that putting them in the keyword package means one less package headache for the macro user. If a guide symbol is not a keyword, and if the macro implementer used `eq` to test for equality with a guide symbol being looked for, then the macro user will have to import the guide symbol into the package at the point of use of the macro (or write `impl-pkg:gizmo`, which is awful). The macro implementors may want to advertise (as the creators of the complex `loop` did) that their guide symbols are compared by testing for string equality of the symbols' names, and hope that all users are aware of the advertisement.

I can't resist pointing out that the documentation for Foderaro's `if*` macro
`http://www.franz.com/support/documentation/6.0/doc/pages/operators/excl/if_s.htm`
describes four guide symbols (`then`, `elseif`, `else` and `thenret`), and fails to mention the equality test used to test for occurrences of them. In fact, if you look at the code, the test is for string equality of their names, so you *don't* have to import them into your package.

Of course, many people dislike `cond` precisely because it uses no guide symbols and sticks in an extra layer of parens to compensate. If you are one of them, go ahead and use `if*` (if you're not an Allegro user, I believe you can just grab the implementation

`http://www.franz.com/~jkf/ifstar.txt`).
Then use keyword versions of the guide symbols. That is, write `:then` instead of `then`, and so forth. The name of `:then` is the same as that of `then` (a fact that is easy to forget), so the equality tests will succeed when they're supposed to, and your code will be more readable.

Another principle of macro design is:

3. Don't duplicate lisp control structures inside a macro. Instead weave the macro into the structures.

Let me give an example. In the development of the `repeat` macro, I noticed that I was often having to use assignment when I'd rather use `let`. Here's what I was forced to write

```
(repeat :for (... x ...)
   (!= x ...)
 :until (member 'foo x)
 :result (remove 'foo x))
```

when I really wanted something like this:

```
(repeat :for (...)
   (let ((x ...))
      ???)
 ???)
```

The problem is that the body of the `let` is invisible to the `repeat` macro. My first reaction was to think of adding a new guide symbol, `:let`, which would allow the user to bind a variable over the remainder of this iteration of `repeat`:

```
(repeat :for (...)
 :let ((x ...))
 :until (member 'foo x)
 :result (remove 'foo x))
```

This would not be hard to implement, but there are so many other patterns, such as

```
(repeat :for (... (x ...) ...)
   (cond ((member 'foo x)
          ... Some longish amount of code
            (cond ((foo y)
                   ??? the equivalent of :return y))
            ...)
           (t
            ??? More shenanigans))
 )
```

One could add more guide symbols, but the net return would be a largish implementation of a substantial subset of Lisp (with clumsy syntax). This is what the designers of `loop` did, as well as the designers of `format`, if you think about it.

A better idea is to introduce two new guide symbols, one (the *diver*) that says "We're going to dive into some Lisp code now," and another (the *snorkeler*) that says, "We've encountered a place in that Lisp code where we should resume processing as if we're in the body of the macro." In the case of `repeat`, the diver is `:within` and the snorkeler is `:continue`.

The same pattern occurs inside `out`. Here `:e` is the diver and `:o` is the snorkeler.

B Alternative Names for Lisp Constructs

There are certain built-in functions whose names are less than felicitous. Who can remember whether `multiple-value-list` returns as multiple values the elements of a list, or whether that job is performed by `values-list`? Is the best convention for constructing the name of a predicate to tack a "p" on the end of something?

For reasons like these, but sometimes flimsier, YTools provides synonyms for built-in Lisp functions using names that I think are better. The synonymy is accomplished in the interpreter by setting `symbol-function` of the synonym to `symbol-function` of the original name; and in the compiler by making the synonym be a compiler macro that expands into a call to the original. So there should be no performance hit at all from using the synonyms.

In some cases the synonym is the same as the original name except for the case of some of the letters in it. In ANSI CL, because it is case-insensitive, the synonym is already available, and trying to define it (as itself!) would cause infinite loops. YTools is careful to check for this situation and avoid actually defining anything.

Anyway, here are the synonyms, in alphabetical order:

`=<` *Function* *Location:* YTFM

`=<` is a synonym for `<=`, which looks like an arrow to me, not an inequality.

`Array-dimension` *Function* *Location:* YTFM
`Array-dimensions` *Function* *Location:* YTFM

`Array-dimension` and `Array-dimensions` are synonyms for the functions with the same names downcased.

`is-`*Type* *Function* *Location:* YTFM

To maintain the convention that types are capitalized, YTools supplies alternative names for `symbolp`, `numberp`, ..., namely `is-Symbol`, `is-Number`, ... Another reason for this choice of names is to help eliminate the "trailing p" convention for predicates in favor of simple declarative constructs such as `is-...` or `...-has-...`. One nonobvious decision is to provide a substitute for `consp` named `is-Pair`. There is, however, no `is-Atom`, because `Atom` is not a Common Lisp type. You can use `atom` or `(not (is-Pair ...))`.

Here is a complete list of all the type-testing synonyms defined by YTools: `is-Array`, `is-Char`, `is-Float`, `is-Integer`, `is-Keyword`, `is-Number`, `is-Pair`, `is-Pathname`, `is-Ratio`, `is-Stream`, `is-String`, and `is-Symbol`

`list-copy` *Function* *Location:* YTFM

`list-copy` is a synonym for `copy-list`.

`list->values` *Function* *Location:* YTFM

`list->values` is a synonym for `values-list`.

`make-Pathname` *Function* *Location:* YTFM

`make-Pathname` is a synonym for `make-pathname`.

`make-Symbol` *Function* *Location:* YTFM `make-Symbol` is a synonym for `make-symbol`.

`off-list`	*Macro*	*Location:* YTFM
`on-list`	*Macro*	*Location:* YTFM

These are synonyms for `pop` and `push`, respectively, which look to me like operations on a stack. They're rarely used for that purpose, because most uses of stacks in Lisp are handled by recursion, but I'd still prefer different names for them.

`pathname-equal`	*Function*	*Location:* YTFM

`pathname-equal` is a synonym for `equal`.

`Pathname-host`	*Function*	*Location:* YTFM
`Pathname-device`	*Function*	*Location:* YTFM
`Pathname-directory`	*Function*	*Location:* YTFM
`Pathname-name`	*Function*	*Location:* YTFM
`Pathname-type`	*Function*	*Location:* YTFM
`Pathname-version`	*Function*	*Location:* YTFM

These are all synonyms for the corresponding functions with lowercase names.

`pathname->string`	*Function*	*Location:* YTFM

`pathname->string` is a synonym for `namestring`.

`Symbol-name`	*Function*	*Location:* YTFM
`Symbol-plist`	*Function*	*Location:* YTFM

Synonyms for `symbol-name` and `symbol-plist`.

`tuple`	*Function*	*Location:* YTFM

`tuple` is a synonym for `list`, intended for contexts where the list being built is thought of as having a fixed length, like a record. Example: `(!= x (list 'a 3))` initializes `x` to a list of two S-expressions. If the programmer wants to signal to the reader of his code that `x` will remain a list of two objects, he or she should write `(!= x (tuple 'a 3))` instead; this also suggests that the first element is going to be a symbol and the second a number. The use of `list` can then be used to signal that the list might grow or shrink, and that its elements are some type that includes both symbols and numbers.

`values->list`	*Function*	*Location:* YTFM

`values->list` is a synonym for `multiple-value-list`.

Index