YTools: A Package of Portable Enhancements to Common Lisp Version 2.1.42

Drew McDermott

November 30, 2008

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.

To summarize my overall philosophy: Macros are good. They capture patterns that would otherwise be clumsily repeated or have to be translated into awkward functions with odd lambda-expression arguments. I don't hesitate to define a macro if I find myself writing a page full of rule definitions that are all of the same form, even if the macro will never be used again. The macros described here are, for the most part, reusable many times.

Tools that are used a lot should have short names. Names involving special characters are often shorter and, once one gets used to them, more suggestive than alphabetic names. Lisp should be used as a functional language to the extent possible, and if you're going to use `remove-if` a lot, it's much easier to write `(<?  atom l)` than `(remove-if-not #'atom l)`, and clearer, too. It's an unfortunate side effect of my fondness for such constructs that the newcomer to my code is often taken aback by the number of strange symbols. Hopefully this manual will make it relatively painless to learn and love them. All the symbols are listed in the index, near the front. Many of the odd symbols occur as "guide symbols" in other macros rather than stand-alone constructs such as `<?`, and the documentation for that macro should be consulted in order to figure out what one of them means. For a quick reference, tables 2– 4 in section A.3 lists all the symbols with a quick explanation of their meaning.

The manual is organized as follows. Each tool is introduced by a line in the following form

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

The *Category* is either *Macro*, *Function*, or *Variable*. 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.

One special symbol that is used in many different contexts is the underscore ("_"). In particular, it may be used as the name for just about any bound variable 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`. These conventions are backed up by the definitions of the constants `true`, `false`, `empty-list`), and `_`, which are documented in section 7.5.

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. They are probably collectively and severally appalled at what it has become.

2 Facilities for Iteration and Function Binding

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.

`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] [:initbind val])

  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.

[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.

The abstract datatype `Collector` is a list built backwards. Instead of consing elements onto the front, you cons them onto the back. 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*`)`. Two successive calls to `Collector-elements` are not guaranteed to return `eq` results. 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` *may or may not* copy its list argument, so that subsequent collection operations *might* destructively alter that argument.[2] To reset a collector *c* to its empty state, execute `(collector-clear` *c*`)`. However, let me emphasize again that normally one doesn't manipulate collectors directly, but only implicitly through `repeat` operations whose syntax is specified above and whose meaning will be described below.

`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*`)`.

  If *v* will not be accessed in the body of the loop, it can be `_`. (E.g., to print `foo` 10 times, you can do `(repeat :for ((_ = 1 :to 10)) (print 'foo))`.)
- `(`*v* `:in` *l* `[:tail` v_t`] [:init` *val*`])`: *v* is bound to *val*, and then 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 don't actually need the value of *v* (say, because all you care about is the value of v_t),you can replace *v* with `_`. The purpose of `:init` is to provide a value for *v* before the first test whether the list is empty. If the list is empty to start with, then this is the final value of *v*. The default `:init` value is `nil`. Even if the list is known to be non-empty, it may be necessary to provide an `:init` field if *v* is declared within the `repeat` to have a type incompatible with a value of `nil`.

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.

[2]It is legal for collectors to be implemented as ordinary lists that are reversed when their elements are retrieved, or as lists with an extra pointer to the last cons cell.

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, in the order they are bound. 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 differences between the alternative clause forms are:

   - `: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, which (see above) means that its cells may or may not be reused).
   - `: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`

   `(: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 in 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 conventions 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 and other "places."

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)))
  (rplaca #:g11726 #:g11725)
  #:g11725)
```

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

`setter` *Macro* *Location:* YTools, `setter`

`(setter` *p*`)`, where *p* is a place, is the function

```
(lambda (new-val fcn)
    (!= p (funcall fcn *-* new-val)))
```

If you pass `(setter` *e*`)` to a function as an argument (call it `s`), the function can alter the value of *e* by calling `s`. For example, to subtract 1 from *e*, it could execute `(funcall s 1 #'-)`. Note that setter returns the new value of *p*. To simply set *e* to a value *v*, you can use the built-in global variables `<-this-val` or `^-this-val`. These are both synonyms for `(\\ (_ x) x)`, but they are *variables,* not functions, so they don't need `#'` tacked on their noses. To retrieve the current value of *e*, you can use the built-in global variable `retrieve-val`, which is a synonym for `(\\ (x _) x)`.

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*. 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 printed `?,(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 are optional.

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. In writing such an algorithm, you will want to refer to the fields of a qvar(oid), using `(Qvar-atsign` *q*`)`, `(Qvar-sym` *q*`)`, and so forth.

There is also a built-in list matcher:

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

`matches` *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. `(matches` ***datum pattern***`)` is synonymous, but often more readable when emphasizing the test rather than the assignment of variables.

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 a list from the datum. A segment is a sequence of zero or more list elements. If a qvaroid matches a segment, its variable is assigned to a list of the items 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 items 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 items 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`.
- `?(:~` *p*`)` matches the datum if and only if *p* does not.

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. ("\\" is an abbreviation for `lambda`.)

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 headed by `:?`. 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)
                  (list 'lambda v))
                 (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:

```
(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

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*`)`	Evaluate 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-*False* value.
`(:e [(:stream` *v*`)]` `---`*exps*`---)`	Evaluate each *exp* and discard the results. Any subexpression of *exp* of the form `(:o` *d*...`)` is, every time it is executed, treated as though *d*... 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*... `[(:suf` *s*`)])`	Print a logical block, with the given optional prefix and suffix. *d*... 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 to print 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*)
   (: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. `dbg-out-indent` is to `dbg-out` as `out-indent` is to `out`.

(`dbg-out-indent` *gate-var* *n*
   ---*body*---)

behaves like `(out-indent *error-output*` *n* —*body*—`)` if *gate-var* is *True*; otherwise like *body* alone.

Several other macros expand into forms that produce output using the same conventions as `out`. In this manual, these forms are indicated by the subexpression —*out-directives*—. A case in point is `read-y-or-in`, which replaces `y-or-n-p`, as part of our campaign to stamp out `format`:

`read-y-or-n` *Macro* *Location:* YTools, `outin`

`(read-y-or-n [:yes-no]` *—out-directives—*`)` follows the `out-directives`, which should produce a question, and waits for the user to answer `y` or `n`, after which it returns *True* or *False*. If the flag `[:yes-no]` is present, then the user must answer `yes` or `no` instead of `y` or `n`.

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*`.[3]

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`.
- `: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 use `out` instead. The main macro is:

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

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

```
(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 ---`*out-directives*`---)`, where —*out-directives*— produce a string paired with the restart.[4]

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

`You will be prompted for:` *description*

where *description* is the string produced by the *out-directives*. 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`:

[4]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.

`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 *False*.

`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:

```
(def-class name
         ---slot-defns----
         [(:handler
              ---meth-defns---)]
         [(:options [(:include ---components---)]
                    [(:medium [:list | :vector
                                | :structure | :object]
                               [:named | :slots-only ]
                               [: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`, 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; if the `:slots-only` flag is present, it will just be an anonymous vector or list; `:named` is the default. The `:slots-only` flag is illegal, and hence the `:named` flag redundant, for media `:structure` and `:object`. (This nice simple rule, that `:named` is the default regardless of medium, is different from that of `defstruct`.)

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]]

`with-slots-from-class` *Macro* *Location:* YTools, `object`

`(with-slots-from-class (`*—slots—*`)` *x* `-` *c* *—body—*`)` is like the Common Lisp macro `(with-slots (`*—slots—*`)` *x* *—body—*`)`, except that it uses the access functions from classoid *c* instead of `slot-value` to access the slots of *x*. For instance, in the body of

```
 (with-slots-from-class (nickname (fullname entire-name-incl-middle))
                        p - Person
    ---body---)
```

an occurrence of `nickname` is treated as though it were `(Person-nickname p)`, and an occurrence of `fullname` is treated as though it were `(Person-entire-name-incl-middle p)`. In the corresponding `with-slots` form, occurrences of `nickname` become `(slot-value p 'nickname)`, and occurrences of `fullname` become `(slot-value p 'entire-name-incl-middle)`. The only problem with `with-slots` is that not all Lisp implementations allow the use of `with-slots` with structures instead of genuine objects.

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

```
(def-op name arglist
           [ | (:method-combination —combo-spec—)
             | (:argument-precedence-order —param-names—)]*
           —body—)
```

This is a near-synonym for `defgeneric`; the only difference is that if the *`body`* is non-empty it is used to make a default method for the newly defined generic function *`name`*, that is, a method that qualifies all the arguments with type `t`. (The `op` in the name of this macro is short for `operation`, which was the name used for generic functions in the T dialect of Scheme (**?**).)

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 it is impossible to write your own tools that fit together with the reader, macro-expander, or writer. For instance, 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 whether a list-constructing form came from a backquote or not.

   That's an example of interfacing with the macro-expander. You might also want to interact with the reader. Suppose you wanted to create a generalized backquote readmacro (call it `!@`) that built something other than list structures. You might write `!@(make-a-foo (baz ,a) ,@l)` as short for `(apply #'make-a-foo (list 'baz a) l)`. Many Lisp implementations will signal an error of type "Comma not inside a backquote" when the expression `!@(...)` is read, and there is no portable way to intervene in the read process to make this legal.

2. The rule for interpreting nested backquotes is that a comma is paired with the innermost backquote surrounding it (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.

A 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 7.5.)

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`
`<?`	*Macro*	*Location:*	YTools, `mapper`
`neg`		*Location:*	YTools, `mapper`

Except for `><` and `neg`, 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 section 7.5)
`<&`	=	`every`
`<v`	=	`some`
`<<`	=	`apply`
`</`	=	`mapreduce` (see section 7.5)
`<?`	=	`remove-if`, with predicate negated
	=	(returns a new list consisting of all the elements of *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.

`><` is tossed into this section because it seems to belong. `(><` *f* —*args*—`)` is just a synonym for `(funcall` *f* —*args*—`)`.

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*`)` can be used 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 Keeping Track of "Extra" Multiple Values

`track-extra-vals` *Macro* *Location:* Yools, `multilet`

Isn't it annoying when you change a function from returning a single value to returning more than one, and suddenly you have to rewrite the entire control structure around the function call, introducing `multiple-value-bind` or whatever? This macro is supposed to ameliorate the pain a bit, by allowing you to treat the newly introduced ("extra") values as afterthoughts. Inside the *body* of

```
(track-extra-vals :extra-vars bdgs
                  [ :principal-values vars
                    | :num-principal-values n ]
                  [ :values exps
                    | :extra-values exps ]
   —body—)
```

an occurrence of `extra-vals` allows you to separate the processing of the "extra" values from the "main" flow.

The general form of `extra-vals` is discussed below, but the idea is that

`(extra-vals` *e* `:+ —`***updates***`—)`

behaves just like *e*, but also does bookkeeping (the ***updates***) on extra variables declared in *bdgs*. In the simplest case, the ***updates*** are just variable names, one per "extra" value returned by *e* beyond its primary value. These names are drawn from ***bdgs***, each a pair of the form `(`***var init***`)`. Each var is initialized as prescribed by ***bdgs***, then set on each execution of `extra-vals` to its corresponding returned value. (The keyword arguments of `track-extra-vals` may come in any order, and may be interspersed among the elements of *body*.)

An example may clarify. Suppose you want to bind a sequence of variables using `let*`:

```
(let* ((x (fb false))
       (y (fb x))
       (z (fb y)))
   ...)
```

Then you realize that `fb` needs a state variable to keep track of what it is looking for. It now takes two arguments, the symbol and the state, and returns two values, a boolean and a new state. Here is how we rewrite the code:

```
(track-extra-vals :extra-vars ((state (init-state)))
   (let* ((x (extra-vals
                (fb false state)
                :+ state)
```

```
           (y (extra-vals
                 (fb x state)
                 :+ state))
           (z (fb y state))))
       ...)
    :extra-values ())
```

Compare this with

```
 (multiple-value-let (x state)
                     (fb false (init-state))
    (multiple-value-let (y state)
                        (fb x state)
       (multiple-value-let (z _)
                           (fb y state)
          ...)))
```

in which the original structure of the code has been completely buried.

The optional keyword arguments to `track-extra-vals` are there to handle every possible special case. The `:extra-values` field specifies what is to be returned beyond the values returned by *body*. The field `:num-principal-values` is necessary because there is no way for the macro to figure out how many values are normally returned. An alternative mechanism is to use the `:principal-values` argument to provide names for those arguments. If you don't like either of those ideas, wait! there's more! An alternative to the use of `:extra-values`, and one of the "principal values" arguments is to use the `:values` argument to specify *all* the values to be returned. If none of the above are specified, the macro assumes the number of "principal" values produced by ***body*** is 1, and it returns this value plus the final values of the extra variables. (If this assumption is wrong, the values produced beyond that primary value are, ironically enough, lost.) If the empty list is supplied as the argument `:extra-values`, then the macro doesn't need the principal-values arguments; it just returns whatever ***body*** would normally return.

The `:values` and `:extra-values` arguments cannot refer to any variables except those in ***bdgs*** and those bound at the point where they are declared. The only exception is that they may refer to the elements of the `:principal-values` argument, if it is supplied.

The general form of `extra-vals` is

```
(extra-vals [(---vars---)]
    —body—
    [:after n]
    :+ —updates—)
```

The `:after` argument says how many values the `extra-vals` expression is expected to return; it defaults to 1. Those values are returned as the value of the `extra-vals` expression; the values of *body* beyond that number become the values of the bound variables ***vars***. Each ***update*** is either a variable or a pair `(`***var newval***`)`. When the ***newval*** is omitted, it defaults to the corresponding ***var***. When all the updates are of this form, the ***vars*** may be omitted entirely. Without these conventions, the first `extra-vals` expression in our `let*` example would have been written

```
    (extra-vals (next)
       (fb false state)
       :after 1
       :+ (state next))
```

7.5 Other Functions and Macros

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

`_` *Constant* *Location:* YTFM

In addition to its use as the "name" of an ignored variable, `_` is a constant bound to `nil`. It is intended to be used in contexts where a value is not going to be looked at by anyone. The standard example is a program that attempts to retrieve or compute a piece of data, and returns two values: the first says whether a useful value could be found, and the second is that value, but (obviously) only in the case where the first value is *True*.

```
(defun retrieve-it (...)
   (let ((v (some-intermediate-value ...)))
      (cond ((crucial-test v)
             (values true (useful value we hope)))
            (t
             (values false _)))))
```

A caller of `retrieve` would use a calling sequence such as

```
(multi-let (((found val)
             (retrieve-it ...)))
   (cond (found
          (do-something-with val))
         (t
          (do-something-not-involving-val ...))))
```

(`multi-let` is described below.)

`\\` *Macro* *Location:* YTFM

`(\\ (...)  ...)` is an abbreviation for `(function (lambda (...)  ...))`.

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

`(alref` *a x* `[`*d* `[:test` *e*`]])` 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. The equality test defaults to `eq` (*not* `eql`). 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.

`control-nest` *Macro* *Location:* YTools, `multilet`

```
(control-nest
 [$tag_1$]
    $exp_1(tag_2)$
 $tag_2$
    $exp_2(tag_3)$
 ...
    $exp_{n-1}(tag_n)$
 $tag_n$
    $exp_n$)
```

is meaningful if and only if every exp_i contains exactly one occurrence of tag_{i+1}. Using $exp_i(e)$ to mean exp_i with the occurrence of tag_{i+1} replaced by e, the `control-nest` expression is equivalent to $exp_1(exp_2(\ldots(exp_{n-1}(exp_n))))$. The only purpose of this construct is to avoid "Vietnamization" (see section 2.1). For instance, there is no analogue of `let*` for `multiple-value-let`, but we can use `control-nest` to get the same effect:

```
(multiple-value-let (x y z)
                    (foo ...)
   (multiple-value-let (p q)
                       (baz x y)
      (multiple-value-let (r s)
                          (zap x p)
         ...)))
```

can be written

```
(control-nest
   (multiple-value-let (x y z)
                       (foo ...)
      :bind-p-q)
 :bind-p-q
   (multiple-value-let (p q)
                       (baz x y)
      :bind-r-s)
 :bind-r-s
   (multiple-value-let (r s)
                       (zap x p)
      ...)))
```

(This is an alternative to using `track-extra-vals` to generalize `let*` as suggested by the example in section 7.4.) Note that the tags are usually from the keyword package, but don't have to be; and that tag_1 is optional, and ignored, but it is occasionally useful in calling attention to a symmetry. For instance:

```
(control-nest
 :test-mother
   (let ((m (mother x)))
      (cond ((ruled-out m)
             :test-father)
            (t
             (succeed m))))
 :test-father
   (let ((f (father x)))
      (cond ((within-bounds f)
             (succeed f))
            (t
             :test-daughter)))
 :test-daughter
   (let ((d (daughter x)))
      (cond ((has-prop d)
             (succeed d))
            (t (fail m f d)))))
```

Exactly where to break the expressions depends on the structure one is trying to call attention to.

Warning: There is a chance, if a fairly remote one, that by using `control-nest` within another macro call you can confuse it. Be sure that whatever syntax the outer macro is expecting to find is not fatally rearranged by `control-nest`. It's actually hard to find a convincing example, but here's one:

```
(repeat ...
 :within
    (control-nest
     (repeat
         ...
         dumb-continue
         ...)
    dumb-continue
     (:continue
      :until test
          ...)))
```

When the outer `repeat` is expanded it may not realize that when the `control-nest` is expanded the `:continue` clause is going to wind up inside, and hence belong to, the inner `repeat`. So it may expand it as though it the *test* should be evaluated inside its environment and the outer loop should stop if it comes out true. However, exactly what happens depends on the implementation of `repeat`; smart alecks should not use the above construct as a way of sneaking an outer-`repeat` `:continue` into an inner `repeat`.

`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 $n' = n - \mathit{length}(l)$. If $n < 0$, `(drop` n l`)` is equivalent to `(take` n' l`)`, where $n' = n + \mathit{length}(l)$.

`empty-list` *Macro* *Location:* YTFM

The expression `()` evaluates to an empty list. `(empty-list)` evaluates to `()`, the empty list. It also prints (when `*print-pretty*` is true) as `!()`, and `!()` will be read as `(empty-list)`. In either case, you can put any well-formed Lisp expression between the parens as a comment. This is typically used to show the type of elements of the list, as in `!(Symbol)`.

`endtail` *Function* *Location:* YTFM

`(endtail` n l`)` is synonymous with `(last` l n`)`.

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

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

`exists` *Macro* *Location:* YTools, `repeat`

`(exists [(`v_i` :in `l_i`)]+` —*body*—`)` is short for `(some (lambda (`—v_i—`)` —*body*—`)` —l_i—`)`. (`some` is a built-in Lisp function.) Here $l_1, l_2, \ldots, l_n$ are lists ($n \geq 1$) and $v_1, v_2, \ldots, v_n$ are variables. `(lambda (`v_1 v_2 ...v_n`)` —*body*—`)` is a "semipredicate"; that is, it returns *False* for some arguments, and some non-*False* value for others, but not necessarily *True*. The v_i are bound to successive elements of the l_i. Each such assignment, of each v_i to the j'th element l_{ij} of l_i, is called a *cross section* of the l_i. The lists don't have to be the same length, so the number of cross sections is the length of the shortest l_i. If all cross sections are examined, and *body* evaluates to *False* on each of them, then `exists` returns *False*. But if the jth cross section makes *body* have a non-*False* value, that value is returned. This explanation is a bit wordy; 99% of the time there is only one list, and we care only whether the value of `exists` is *False* or *True*, so we can summarize by saying: `exists` tests whether the list contains an element satisfying —*body*—.

`false` *Constant* *Location:* YTFM

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

`forall` *Macro* *Location:* YTools, `repeat`

`(forall [(`v_i` :in `l_i`)]+` —*body*—`)` is short for `(every (lambda (`—v_i—`)` —*body*—`)` —l_i—`)`. (`every` is a built-in Lisp function.) `forall` is dual to `exists`. It returns *True* if —*body*— is not *False* on every cross section of the lists l_i. In the usual case where there is just one list, it returns *True* if and only if every element of that list satisfies —*body*—. (Note that `forall` differs from `exists` in that the only thing that matters about the value of the *body* is whether it is *False* or not.)

`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. (See `debuggable`, above.)

`gen-var` *Function* *Location:* YTools, `multilet`

`(gen-var` *v*`)`, where *v* is a symbol, creates a new, uninterned symbol whose name is *v*-*k*, where *k* is an integer (similar to `gensym`).

`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.

`ignore` *Declaration* *Location:* YTFM

`(ignore ...)` may be used in almost all contexts where `(declare (ignore ...))` is legal, including `defun`, `defmacro`, `defmethod`, `def-op`, `def-meth`, \\, `multiple-value-let`, and `let-fun`.

`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. If *e* is missing, it defaults to *s*.

`intercept` *Macro* *Location:* YTools, `binders`

`(intercept` *g —body—*`)` is equivalent to `(catch '`*g —body–*`)`; that is, the catch tag must be a constant symbol, not a form to be evaluated.

`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. (Its argument is declared to be a list.)

`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. This function is used by `printable-as-string`, described below, which is almost always adequate. The constant `eof*` is a Printable.

`mappend` *Function* *Location:* YTools, `mapper`

`(mappend` *f —lists—*`)` is like `mapcan`, except that the values of *f* applied to successive cross sections of the *lists* are appended together nondestructively rather than being `nconc`'ed.

`mapreduce` *Macro* *Location:* YTools, `mapper`

`(mapreduce` *f i l*`)` is equivalent to `(reduce` *f l* `:initial-value` *i*`)`, except that any number of list arguments is allowed. If there are more than one, then *f* is applied to their cross sections just as other mapping functions are.

`memoize-val` *Macro* *Location:* YTools, `misc`

`(memoize-val` *e* `:store-as` *p* `[:missing-if` *m*`])` returns the value of *e*. The first time it is called for a particular *e* it evaluates it and stores the result in the place *p*. On subsequent occasions it just returns the contents of *p*. It decides whether *p* contains the value of *e* by evaluating *p* and comparing it to the value of *m* (default: *False*). Example:

```
(let ((fibtab (make-array 100 :element-type 'integer
                              :adjustable true
                              :initial-element -1)))
   (defun fibonacci (n)
      (cond ((< n 2) 1)
            (t
             (cond ((>= n (array-dimension fibtab 0))
                    (adjust-array fibtab (* 2 (array-dimension fibtab 0))
                                  :initial-element -1)))
             (memoize-val (+ (fibonacci (- n 1))
                             (fibonacci (- n 2)))
                          :store-as (aref fibtab n)
                          :missing-if -1)))))
```

`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.

The error message generated by `multi-let` in case of parameter-argument mismatch might look like this in the case of the `(y z)/(baz u v 3)` example above:

```
Wrong number of arguments.  Wanted (y z)
 got ((oops tilt))
 as value of (baz u v 3)
```

In this instance `baz` has returned a single value, `(oops tilt)`, when two were expected.

`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*.

`pass` *Macro* *Location:* YTools, `binders`

`(pass` *g e*`)` is equivalent to `(throw '`*g e*`)`; that is, the catch tag *g* must be a constant symbol, not a form to be evaluated.

`->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* (in `princ` mode). Equivalent to `(make-Printable (\\(srm) (out (:to srm) (:a` *s*`))))`. For example, a language interpreter might return an error flag defined as

```
(defconstant +err-flag+ (printable-as-string "#<Error during evaluation>"))
```

`series` *Function* *Location:* YTFM

`(series [`*l*`]` *h* `[`*i*`])` is the list of numbers $(l,\ l+i,\ l+2i,\ \ldots,\ l+\lfloor\frac{(h-l)}{i}\rfloor i)$. 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:* YTools, `misc`

`(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.

`string-concat` *Function* *Location:* YTFM

`(string-concat` *—strings—*`)` is equivalent to `(concatenate 'string` *—strings—*`)` + declarations that all the arguments are strings.

`string-length` *Function* *Location:* YTFM

`(string-length` *s* `)`

`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`.

`val-or-initialize` *Macro* *Location:* YTools, `misc` `(val-or-initialize` *place* `:init` *v* `[:missing-if` *m*`])` returns the contents of *place*. However, if the contents currently equal *m* (default *False*), then *v* is evaluated and its value used to initialize *place*. Example: Suppose we maintain two representations of a data structure (type `Formula`). The normal one is "internal form," but for human consumption we also have an "external form." We compute the latter only when we need it:

```
(defun get-external-form (fmla)
   (val-or-initialize (Formula-external-form fmla)
                      :init (compute-external-form fmla)))
```

Amusingly, `memoize-val` and `val-or-initialize` are the same function, with its required arguments swapped:

```
(memoize-val e :store-as p :missing-if m)
  = (val-or-initialize p :init e :missing-if m)
```

But each is appropriate in completely different circumstances. The difference is that the *place* (*p*) specified in a call to `val-or-initialize` is expected to be altered in the future, so that the value *e* stored there initially will probably be overwritten later. Whereas the *e* specified in a call to `memoize-val` is computed once and stored for future reference; altering it later would make no sense. One consequence of this asymmetry is that a compiler should treat the `memoize-val` expression as being of the same type as *e*, and the `val-or-initialize` expression as being of the same type as *p*.

`with-gen-vars` *Macro* *Location:* YTools, `multilet`

In a macro, a common idiom is calling `gensym` or `gen-var` (qv.) repeatedly to create a set of new, uninterned variables that can be used to bind values without fear of capturing existing variables. `(with-gen-vars (`—***vars***—`)` —***body***—`)` takes a list of symbols (the ***vars***) and creates a set of uninterned symbols using `gen-var`, then binds local variables to them for use inside —***body***—. If *v* occurs in the list ***vars***, then the generated variable has the name *v*-*k* for some integer *k*, and the variable bound to it for use in body has the name *v*`$`. For example:

```
(defmacro set-to-quo-sum-dif (x y)
   (with-gen-vars (x y)
      `(let ((,x$ ,x) (,y$ y))
          (!= ,x
              (/ (+ ,x$ ,y$)
                 (- ,x$ ,y$))))))
```

The expression `(set-to-quo-sum-dif (cadr z) (!= u (+ u 1)))` expands to

```
(let ((#:x-23 (cadr z)) (#:y-24 (!= u (+ u 1))))
   (!= (cadr z)
       (/ (+ #:x-23 #:y-24)
          (- #:x-23 #:y-24))))
```

8 Downloading the Software and Getting it Running

The YTools package is available at my website
`http://www.cs.yale.edu/homes/dvm.`
and at CLOCC, the Common Lisp Open Code Collection
`http://sourceforge.net/projects/clocc.`
You download a tar file that creates two subdirectories. Let's suppose you expand it in a directory `~/prog/`; you'll get `~/prog/clocc/src/ytools/` and `~/prog/clocc/src/ytools/ytload/`. The latter directory contains basic code for loading YTtools. Then put the following in your Lisp initialization file (e.g., `.clinit.cl` in Allegro Common Lisp):

```
(setq ytload-directory* "~/prog/clocc/src/ytools/ytload/")
(load (concatenate 'string ytload-directory* "ytload.lisp"))
(setq yt::ytload-directory* ytload-directory*)
(setq yt::config-directory* "~/ytconfig/")
(setq yt::config-file* "dialect.lisp")
```

The `ytload-directory*` is where the "bootstrap loader" for YTools is located; this is the `ytload` subdirectory of `ytools` (unless you move it). The slightly clumsy sequence above involves defining this variable both in the `:cl-user` package and (after it is created by `ytload.lisp`) the `:ytools` package, whose nickname is `yt`. I'm assuming that you're installing YTools for your private use, and that "~" can be used to refer to your home directory. Many Lisps or OSes don't recognize this idiom, in which case replace it with your actual home directory. (In some Windows implementations, this whole issue has been resolved in a way to cause maximum pain, so that you have to use `"c:\\Docume~1\\"`*yourname*`\\"` instead of `"~/"`.) If you're installing YTools for use by a bigger group, then use `"/homes/classes/ai-class/"` or even `"/usr/sbin/"` instead of the home directory in what follows.

Anyway, the other two variables, `config-directory*` and `config-file*`, tell YTools where configuration information is kept. Every Lisp dialect should have its own configuration file, but they can all be in the same directory. If you are running just one dialect, you can let `config-file*` keep its default value of `"ytconfig.lisp"`, but you must specify a value for `config-directory*`. I recommend setting `config-directory*` to `"~/ytconfig/"` and `config-file*` to `"`*dialect*`.lisp"`, as in `"clisp.lisp"`.

`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 9.8. 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, so get all the way through the installation interview before making any changes. If you don't feel comfortable guessing what the variables mean, you can make all changes through the interview process by executing `(yt-install :ytools :start-over` *True*`)`, which will ask all the questions again. Another keyword argument to `yt-install` is `:if-installed`; it has four possible values that determine what to do if the system is already installed: *False* means do nothing; `:warn` (the default) causes the system to stop and ask if you want to reinstall; `:reinstall` causes it to reinstall without asking; `:start-over` is equivalent to the argument `:start-over` *True*. Note that without specifying `:start-over`, the system will ask only questions that have not already been asked; typically, it will just recompile and reload the system without asking anything.

After a 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`, `defmethod`, `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 9.2). 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::defmethod
           ytools::eval-when)
   . . .)
```

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

YTools defines three macro characters

`,  ?  !`

described in detail in section A.3. To take advantage of them, you should set `*readtable*` to the value of `ytools-readtable*`. If you have your own macro characters, you may want to build your readtable by copying `ytools-readtable*`. Unfortunately, the readtable system is not as flexible or well designed as the package system, so you may have to be more imaginative about copying YTools's macro characters. (For further discussion about declaring readtables in files, see section 9.1.)

The current version number for YTools is the value of the constant `+ytools-version+`. It can also be found in the file `CHANGELOG`; look for the first expression of the form

`[- Version` *major.minor.patch*... `-]`

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. Files are organized into *systems,* collections of files that have a specified effect on compiling and running the files that depend on them. Systems and their associated modules, packages, and whatnot are defined in files with names of the form *mod*`.lsy`.[5]

If this sounds too bizarre to be worth exploring, you may want to skip to section 9.8.

However, one thing to be aware of is the meaning of the "Location" information in the presentation of YTools features. If you want just a few pieces of the YTools system, you can load just the YTFM, plus the files declared as the locations of the pieces of YTools 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)`. Most tools have "YTools" in their "Location" information, which means the tool is loaded when you execute `(yt-load :ytools)`. If the location includes a file name F (in `typewriter` font), then the function or macro is to be found in the file F`.lisp`. You can load just the one file to get that function or macro, although you may have to include other files to get macros or subroutines it uses. Of course, if you use YTFM all the files F`.lisp` needs will be loaded automatically, but you decided not to . . . .

If the location is of the form "YTools + F", that means that, in addition to loading `:ytools`, you have to `fload` the file F`.lisp`. Evaluate `(yt-load :ytools)` (once), then `(fload %ytools/` F`)`, for each such F. At present (version 1.9.55 of this manual), although there some extra utilities to

[5]At the top of the hierarchy are *load modules*, which are defined by files with names of the form *lm*`.lmd`. These are the items dealt with by `yt-load` and `yt-install`. Load modules are different from *YTools modules,* which are discussed in section 9.5.

be found in the YTools directory that are not loaded as part of YTools, they are not documented here. The only documentation is whatever comments there are in the code. It might be worth your while to take a look anyway.

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-flags*— —*filespecs*— [:readtable *R*])

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

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

loads files

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

assuming that all three files have been compiled since they were changed, but not yet loaded.[6] Unlike `/home/smith`, `%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 section 9.4.

If `fload` is called with no arguments, or its only arguments are flags, then it works with the same files that were specified the last time it had filespec arguments. If it has any flag arguments, then they replace the previous flag arguments. If it has no arguments at all, then both the flags and the filespecs are the same as they were the last time `fload` was called.

The `:readtable` argument is occasionally useful in addressing the awkward lack of symmetry between packages and readtables. Most files use `in-package` to declare which package should be used to read them, but, oddly, there is no analogue for readtables, which are just as important. YTools adopts the following strategy in figuring out the readtable to be used for a file, based on Allegro Common Lisp's *named readtables.* In non-Allegro implementations, a rudimentary named-readtable facility is defined in the file `ytools.lsy`. `(named-readtable` *symbol*`)` returns the readtable named *symbol* or *False* if there isn't one. To associated a name with a readtable, evaluate `(setf (named-readtable` *symbol*`)` *readtable*`)`. The readtable is actually associated with the keyword symbol of the same name as the *symbol*. The only built-in readtable is named `:ytools`, which is the value of the global variable `ytools-readtable*`. To make this the readtable to be used when processing a file, make sure the mode line at the front of the file has a `Readtable` field, as in this example:

```
;-*- Mode: Common-lisp; Package: ydecl; Readtable: ytools; -*-
```

YTFM looks for this declaration as its main source of information about a file's read syntax. In addition, there is a special operator `(in-readtable` *name*`)` to be used in a file to declare which readtable is to be used in reading it (at least until the next occurrence of `in-readtable`). There is also a convenience operator `(in-regime` *pname* `[`*rtname*`])`, which is equivalent to `(in-package` *pname*`)` and `(in-readtable` *rtname*`)`. If *rtname* is omitted, it defaults to *pname*.

If neither of these options is used, but the file has already been loaded or compiled, then YTFM has probably recorded the appropriate readtable, and will use it again. If that recourse proves

[6]And, of course, assuming that `fasl` is the appropriate extension for an object file in the host Lisp system; see section 8.

futile, it just uses the global value of `*readtable*`, unless this behavior is overridden by providing a `:readtable` argument to `fload` (or `fcompl`; see section 9.2).[7]

A key feature of `fload` is that it does not simply load a file; it also records that an up-to-date version of the file be kept in memory from now on (or until the request is canceled (using the `-x` flag, described below). It uses the YTools chunk system (**?**) for this job. The idea is that whenever an edit occurs that might change the contents or meaning of the `fload`ed file, the file should be reloaded (perhaps after being recompiled). Using chunk terminology, we say that the chunk `(:loaded` *pathname*`)` is *managed.* The exact meaning of a chunk depends on how it is connected to other chunks and what the procedure responsible for keeping it up to date does. (This procedure is called its *deriver.*) Hopefully when I introduce a chunk and informally describe what it means, its name will be vivid enough to remind you of that meaning. When a chunk *manages* a chunk with a propositional name, that means it undertakes to keep the proposition true. When it manages a chunk whose name sounds like a dataset, that means it undertakes to recompute the data when the data they depend on change.

Because of the chunk system, `fload`ing a file that has already been `fload`ed usually has no effect; if the file needs to be reloaded, chances are the chunk system has already done so. However, this behavior can be changed using flag arguments to the `fload` macro. The possible flag arguments are:

- `-f`: Force the file to be loaded even if has been loaded already and not changed since.
- `-c`: Force the file to be recompiled and loaded even if `fload` normally would not do one or both of these operations.
- `-s`: Indicates that the source version of the file should henceforth be preferred to the object version, even if it is older. Also forces the source to be reloaded, unless the `-x` flag is present.
- `-o`: Indicates that the object version of the file should henceforth be preferred to the source version, even if it is older. Also forces the object to be reloaded, unless the `-x` flag is present.
- `-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.
- `-x`: Declare that the chunk `(:loaded` *file*`)` is not to be managed any longer. One can reverse this decision later simply by calling `fload` again. If the request to cancel cannot be undone without undoing other requests, the user will be asked whether they really mean it. E.g., if file $F1$ depends on $F2$, and the user has `fload`ed $F1$, a request to cancel management of `(:loaded` $F2$`)` will not be honored until the user has confirmed that `(:loaded` $F1$`)` is no longer to be managed either.
- `-z`: Normally bringing a chunk up to date causes all the chunks that depend on it to be brought up to date, or *rederived* (**?**). In the case of `fload`, if a file is reloaded, then any file that depends on it (see section 9.3) is also reloaded — and often recompiled. Sometimes, if a file is being repeatedly revised and reloaded, one would like to postpone rederivation of the files (and other chunks) that depend on it. The `-z` flag causes this to happen.
- `-`: Clears all flags. This operation makes sense because the flags are "sticky," and are remembered as long as you keep evaluating `fload` calls with no explicit file 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. Writing `(fload -)` causes the file to be reloaded *without* mandatory recompilation.

[7]One consequence of these rules is that the previously recorded value of the readtable for a file overrides the explicit `:readtable` argument to `fload`. This is probably wrong, but since an explicit annotation in the file itself overrides both, it hasn't seemed worth fixing.

(Flags are not case-sensitive; `-X` is eqivalent to `-x`.) The operations postponed by the `-z` can be executed, thus bringing the Lisp session up to date, by executing `(postponed-files-update)`. As long as there are postponed file operations, `fload` and `fcompl` (sect. section 9.2) will remind you of their existence every time they are called. You can turn off the warnings by setting `warn-about-postponed-file-chunks*` to *False*.

When the `-c` flag is not given, `fload` decides what to do based on a global switch, `fload-compile*`. This is *not* a variable, but a symbol macro (so that YTFM can detect when it is set). The switch takes on one of these values:

- `:source` — Never compile. Load a source file if it has been changed since the last time it was loaded.
- `:object` — Never compile. Load the object file if it has been changed since the last time it was loaded. The age of the source file, if there is one, is irrelevant.
- `:compile` — If the object file is older than the source file, compile the source and load the resulting object file.
- `:ask-once` — Ask the user whether the object or source version of the file should be loaded. The user has four possible responses (some of which can be given in a short or long version, indicated as "*c*`[`*cccc*`]`"):

  1. `o[bject]`: Use object; further questions will be asked about whether the file should be freshly compiled.
  2. `s[ource]`: Use source.
  3. `+`: Yes, and set `fload-compile*` to `:compile` from now on.
  4. `-`: No, and set `fload-compile*` to `:object` from now on.
  5. `\!`: Cease managing the chunk corresponding to the loaded file; as if the `-x` flag had been given to `fload`.

  In addition, one can respond `\\` to any query from . YTFM in order to cancel the entire current call to `fload` or `fcompl`

- `:ask-every` — Like `:ask-once`, except that the answers are not recorded. This gets to be very tedious very quickly.
- `:ask-ask` — Like `:ask-every`, except that the user is asked whether the asking should stop (for the particular file being inquired about).
- `:fresh-object` (the default) — If an up-to-date version of the object file exists, it is loaded. If it does not, then we switch to `:ask-ask`.

For large systems, the seemingly reasonable value `:fresh-object` can result in questions about 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`, or just type `+` when YTFM asks you what to do. Remember to set it back to `:ask` when the recompilations are done, if that's the long-term behavior you want.

Because `fload-compile*` is not a variable, you can't bind it with `let` or `bind`. The macro `(bind-fload-compile*` *v —body—*`)` changes its value within the *body* (dynamically scoped).

When loading and compiling files, you will see messages announcing when each file is processed. You will also see messages announcing when they are being body-slurped (section 9.3). All the messages can be suppressed by setting `fload-verbose*` to *False*.

9.2 Compiling Files: `fcompl`

`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 macro is invoked and the object code is out of date:

```
(fcompl  —fcompl-flags—  —filespecs—  [:readtable R])
```

The arguments have meanings analogous to those for `fload`, with similar defaulting behavior: If no arguments are supplied, then all default; if only flags are specified, then the other two default; if the readtable is unspecified, it defaults. Here *filespecs* have the same format as for `fload`.

The flags for `fcompl` are:

- `-f`: Force the file to be compiled even if its source has not been changed since the last time it was compiled.
- `-l`: If the file is successfuly compiled, force the file to be loaded even if `fcompl-load*` (see below) is *False*. (The chunk `(:loaded` *file*`)` is managed from now on.)
- `-x`: Declare that the file's "compiled" chunk is not to be managed any longer. One can reverse this decision later simply by calling `fcompl` again.
- `-z`: As for the `-z` flag for `fload` (section 9.1), postpones updating all chunks that depend on the filespecs' being compiled. The chunks can be brought up to date later by executing `(postponed-files-update)`.

  Note that if a file has been `fload`ed, calling `fcompl` to compile it will normally cause it to be reloaded, because "*f* is loaded" depends on "*f*'s object file is up to date." The `-z` flag will postpone the load, which can be useful if one wants to put off loading until the compilation seems to occur without any errors.
- `-`: As for the same flag in conjunction with `fload`, this causes `fcompl` to "erase" the `-f` and `-z` flags.

(The flags are not case-sensitive; `-X` is equivalent to `-x`.)

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-load*`:

- If it's any value other than *False* or `:ask`, the new object file is always loaded.
- If it's *False*, the new object file is never loaded (although the `-l` flag will override this decision).
- If it's `:ask` (the default), it will ask if you whether to load the object file. If you respond `y` or `yes`, it will be loaded; if you respond `+` or `-`, `fcompl-load*` will be set to *True* or *False*, respectively.

However, as noted above in the paragraph describing the `-z` flag, if the file has been previously `fload`ed, then the value of `fcompl-load*` is irrelevant; the call to `fload` is sufficient reason to load the file, unless the `-z` or `-x` flag is used to postpone or cancel the load request.

9.3 File Dependencies: `depends-on`

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.*[8] There is more than one kind of slurping, but we will distinguish between two basic sorts: `header` and `body` slurping. The latter examines the entire file, while the former examines just its *header*, to extract information about the resources it requires in order to be loaded or compiled. It is normally unnecessary to distinguish the header explicitly, because its principal components are calls to the macro `depends-on`, whose natural location is the front of the file, along with calls to `in-package` and the like.

Suppose file `foo.lisp` starts with a mode line, perhaps an `in-package form`, and then

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

When `foo.lisp` is compiled (say, by writing `(fcompl foo)`), the first thing that happens is that YTools header-slurps `foo.lisp` to see that `baz.lisp` (or its compiled version) must be loaded in order for the functions in `foo.lisp` to work properly. Once the file `baz.lisp` has been noted, the system also commits to slurping *it.* The word "commits" indicates that the YTFM will re-examine the header of a file whenever it changes.[9] In chunk terminology (**?**), the YTFM manages the chunks `(:file-header-scanned "foo.lisp")` and `(:file-header-scanned "baz.lisp")`. It will stop managing such chunks only when it stops managing the corresponding `(:loaded` *filename*`)` chunks.

The general form of `depends-on` is

`(depends-on [`*---filespecs---*`]` *---timed-groups---*`)`

where each `timed-group` is of the form
`(:at [`*time-spec*`]+)` *---filespecs---*

and where a `time-spec` is one of
```
:run-time    :compile-time    :read-time
```

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. The time-specs default to `:run-time` and `:compile-time` for the *filespecs*, if any, that come first in the macro expression, before any explicit time-specs.

Suppose the form

`(depends-on ...`
`(:at` t_1 `...` t_n`)` G_1 `...`G_k`)`

occurs in file F. If one of the t_i= `:compile-time`, then whenever F is compiled, all the G_j should be loaded. If one of the t_i= `:run-time`, then whenever F is loaded all the G_j should be loaded. A `:run-time` dependency also has an impact at compile time. If F depends on G_j `:at :run-time`, it is assumed that any macro defined in G_j may be used in F. To find the macro definitions, G_j must be examined. The entire file must be scanned, so this is a form of *body slurping.*

There is one remaining time-spec, that indicates files G_j that must be loaded before F can be read, meaning, before F can be compiled, loaded (as source), or body-slurped. Such a dependency must be noted as soon as possible when F is being read, so that the relevant G's can be loaded before the processing of F can continue. Not surprisingly, this is the `:read-time` time-spec (synonym: `:slurp-time`).

In case you need some more detail on how headers work, here goes: 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,

[8] It's reminiscent of what the Java compiler does, except the the Java compiler examines class files, not source files.

[9] It tests for whether it has changed only when an explicit `fload` or `fcompl` is issued for a file connected to it by dependencies.

and so forth. Expressions that play these roles are called *headerish* forms. A form is headerish if its function or special-operator is one of `depends-on`, `in-package`, `defpackage`, or `self-compile-dep` (explained below). Function and datatype definitions are obviously not headerish, but neither are declarations. Like the Lisp compiler, the slurp mechanism expands any macro form it encounters and examines the result as if it occurred in the file. The header can come to an end in the middle of `(progn` $e_1 \ldots e_n$ if some e_i is not headerish.

In almost all cases, the separation between the header and the body of the file occurs naturally. If it seems that YTFM is putting it in the wrong place, set the global variable `end-header-dbg*` to *True*. This causes messages to be printed about YTFM's decision processes in classifying header forms, so you can see where it goes off the rails. If you want to change the status of a form, use one of the following facilities.

`end-header`	*Macro*	*Location:*	YTFM
`in-header`	*Macro*	*Location:*	YTFM

The macro `end-header` can be used to declare the end of the header of a file. The format is

```
(end-header [:no-compile])
```

It usually takes no arguments, but if the flag `:no-compile` is present, it tells YTFM that the file should never be compiled.

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 executed or compiled, but when slurped it will not end the header, no matter what form each e_i takes.

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.

The operator `(self-compile-dep [ :load-source |` *—sub-file-types—*`])`, when used in the header of a file, indicates how the file depends on itself when it is compiled. The flag `:load-source` indicates that before the file is compiled it should be loaded in source form. It's possible to narrow the scope of what is looked for and avoid loading the entire file. For instance, if you write `(self-compile-dep :macros)`, that indicates that the macro definitions are to be found and evaluated before the file is compiled. This is useful only if you want to put macro definitions after a point where they are used, or if you want forms wrapped with `needed-by-macros` to be evaluated before the file is compiled. In the basic YTools release `:macros` is the only sub-file type, but systems built on top of YTools can define others.

9.4 YTools Logical Pathnames

As mentioned, 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. 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`). Having reached the end, we tack on the remembered sequence, *d*, yielding `.../`*c*`/bin/`*d*`/`. With more than one level to ascend through, the remembered sequence gets one directory per level. The whole sequence is recorded and included in the final result, unless directory layers with the name "`--`" are encountered. Each such occurrence means that the layer that would otherwise have been included should be skipped. The default relative location for the binary files for YTools itself is `../../bin/--/`. If YTools is in `~/prog/src/ytools/`, then binary files will be placed in `~/prog/bin/ytools/`. The first "`..`" moves us to `~/prog/src/`, remembering `ytools/`; the second moves us to `~/prog/` remembering `src/ytools/`. The `bin` moves us to `~/prog/bin/`, and the `--` discards `src`, so that the final result is `~/prog/bin/ytools/`.

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.

If the optional *object-code-loc* argument is omitted, the existing association between source directory and object directory, if any, is not changed. The default object directory is the source directory itself. An explicit *False* as the value of the *object-code-loc* argument restores this default.

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.

9.5 YTools Modules

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 M has three components, distinguished by how they behave when `(depends-on %module/` M`)` is found in file F:

1. Its *run support* for F: What should be evaluated when F is being loaded (in either source or object form).

2. Its *compile support* for F: What should be evaluated when F is being compiled.

3. Its *expansion*: a set of forms that are treated as if they had occurred in F at the point where the `depends-on` form was found.

A module is defined by `def-ytools-module`:

```
(def-ytools-module name
   ((---context-specs---) ---forms---)+)
```

where each *context-spec* is either `:run-support`, `:compile-support`, or `:expansion`. The *forms* are arbitrary forms to be evaluated, but there is a special operator you should use to indicate that files should be loaded: `module-elements` has the same syntax as `fload`, and behaves the same, but signals to the module system that loading these files is not a random act, but is constitutive of the module. This makes a difference when using `fcompl` to compile a module.

Here is a sample module definition, for the Nisp system, which adds strong typing to Lisp (**?**):

```
(def-ytools-module nisp
   (:run-support
       (module-elements %ydecl/ runnisp))
   (:compile-support
       (module-elements %ydecl/ compnisp))
   (:expansion
       (depends-on-file-type-decls)
       (self-compile-dep :nisp-types :macros)))
```

If the form `(depends-on %module/ nisp)` occurs in F`.lisp`, then three things are supposed to happen:

1. When F's object file is loaded, so is `runnisp.lisp` in directory `%ydecl`. (It will in turn load several other files.)

2. When F is compiled, `compnisp.lisp` is to be loaded. (It also loads other files, which happen to be a superset of the ones loaded by `runnisp`.)

3. The forms `(depends-on-file-type-decls)` and `(self-compile-dep :nisp-types :macros)` are to be inserted at the top level of F. The first causes files F depends on at run time to be slurped looking for the types they define. The second causes F itself to be slurped, both for type definitions and for macro definitions.

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 file in the `ytools` directory. If a file requires one of them, you indicate an extra dependence on it by expanding thus:

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

You can use `fcompl` to compile a module. Just write `(fcompl` *flags* `%module/` *name*`)` and the effect will be the same as if each file declared using `module-elements` were `fcompl`'ed separately. There is a subtlety here to be aware of. Suppose a module `titanic` contains 5 files, but one of them (call it `main.lisp`) depends on the other 4. Then for loading purposes one may define the module using something like

```
(def-ytools-module titanic
    ((:compile-support)
     (module-elements %mydir/ main))
    ...)
```

Loading the (compile-time piece of) the `titanic` module will result (in effect) in issuing the call `(fload %mydir main)`, which will load all 5 files in. But `(fcompl -f %module/ titanic)` is equivalent to `(fcompl -f %mydir/ main)`, and so is sure to compile `main`, but unlikely to compile the 4 supporting files unless there is some independent reason to compile them. This is seldom an important issue, unless you find yourself wanting to compile a module without loading it.

9.6 File Transduction

One well known problem with `load` is that loading a file cannot change the current package, even if the only thing in the file is `(!= *package* ...)`. The reason is that `*package*` is always "bound" with file scope during `load`.

`after-file-transduction` *Macro* *Location:* YTFM

`fload` and `fcompl` have their own scoping mechanisms, which allow you to control what gets evaluated outside of their scopes after a file is loaded or compiled. The form `(after-file-transduction` *—forms—*`)` causes the *forms* to be evaluated when the current call to `fload` or `fcompl` is done. The *forms* are evaluated in reverse chronological order of posting, and left-to-right within each form list. That is, if you execute

`(after-file-transduction` e_1 e_2`)`
then
`(after-file-transduction` e_3 e_4`)`

then when the file is finished the forms will be evaluated in order e_3, e_4, e_1, e_2. Of course, `unwind-protect` is used to ensure that the forms are evaluated even if the transduction terminates abnormally.

In addition, the form `(setf-during-file-transduction` *form value*`)` causes *form*'s current value to be saved, then for it to be given the value *value* until the current file transduction is over. At that point the saved value is restored. Because the value is restored using `after-file-transduction`, the saved value is restored even if the transduction terminates abnormally.

If you want to implement your own file transducer, you can use this macro:

`with-post-file-transduction-hooks` *Location:*
Macro *Location:* YTFM

`(with-post-file-transduction-hooks` *—body—*`)` causes the *body* to be evaluated with the regime described above in place. `fload` and `fcompl` use this macro to make `after-file-transduction` work.

To test whether a file transduction is in progress, use the function `(during-file-transduction)`, which returns *True* if and only if it is evaluated inside the body of `with-post-file-transduction-hooks`.

9.7 Other File-Management Facilities

`fload-versions` *Macro* *Location:* YTFM

It is often the case during system debugging that you want to use an experimental version of a file. One way to do that is to use a version-control system such as CVS (?) to create a separate "development branch," which can be merged back into the main branch if the experiment works out. This sometimes seems like an overly heavy-handed approach, especially given how clumsy systems like CVS can be. An alternative is to tell YTFM to intercept all references to the file and pretend that another file had been referred to instead. You do this by writing `(fload-versions` *—filespecs—*`)`, where the *filespecs* are as for `fload` and `fcompl`, except that instead of filenames one writes pairs of filenames, the second being the temporary version of the first.

```
(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.

The format of a file-version pair allows for several other possibilities. 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.

`def-file-segment` *Macro* *Location:* YTools, `fileseg`

`def-file-segment` is a device for defining an interdependent network of chunks, each consisting of a piece of a file. As alluded to in secrefytfmthe section on YTFM, a chunk manages a dataset or invariant, using a piece of code called its deriver. The deriver is run to bring the chunk up to date (recomputing the dataset or restoring the invariant) whenever it appears to be necessary.

`(def-file-segment` *chunk-name* `(`*—base-chunk-names—*`)` *—deriver—*`)` defines *chunk-name* as *deriver*. This form should occur only inside a loaded file, hence the name. The *base chunk names* are names of previously defined file segments. The chunk is assumed to require updating whenever the file it was defined in is reloaded or one of its base chunks is updated.

The standard use for this device is in the maintenance of tables. Suppose a table is initialized in file `A.lisp`, then added to in files `B.lisp` and `C.lisp`. We put the following in `A.lisp`:

```
(def-file-segment init-table ()
   (defvar magic-table*)
   (!= magic-table* (make-hash-table)))
```

and the following in files `B.lisp`

```
(def-file-segment B-table-aug (init-table)
   (!= (href magic-table 'foo) 13))
```

. . . and `C.lisp`:

```
(def-file-segment C-table-aug (init-table)
   (!= (href magic-table 'oof) 14))
```

If file `A` is reloaded, then both the chunks will be rederived, thus ensuring that the table stays up to date, even if files `B` and `C` are not reloaded.

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

To switch from your current system-definition facility to YTools, all you need to do is put one line at the beginning of each file saying what files it depends on. Does this make it impossible to switch back to a traditional facility? Can you use YTools without giving up `defsystem` or `asdf`?

Relax. All of the little extras in YTools files, such as `end-header`, `depends-on`, and such, are defined as harmless macros. If you `load` a file instead of `floading` it, they will do nothing, except possibly generating a warning message or two. If you want the whole system to just go away, including the warning messages, set the global variable `depends-on-enabled*` to *False*.

A Some Notational Conventions

A.1 Syntactic Notation Used in This Manual

To illlustrate the syntax of YTools constructs, I use an informal BNF-style notation, in which nonterminals appear in an *italic font*. The notation —*xs* — means "zero or more occurrences of an *x*," with the definition of an *x* given shortly thereafter. As a particular case, —*body* — means an "implicit `progn`," that is, a sequence of expressions to be evaluated, with the value(s) of the last returned as the value of the sequence. I use the notation `[`*x*`]+` to indicate one or more *x*s. An optional syntactic constituent is written `[`*x*`]`. A choice of constituents is written `[` *x* `|` *y* `|` `...|` *z* `]`. If one of the choices is to omit the constituent entirely, I write `[|` *x* `|` `...]`.

A.2 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.3 Special Characters

YTools has its own readtable (`ytools-readtable*`), in which three macro characters are reserved: exclamation point, comma, and question mark. Exclamation point is used for several purposes (discussed as they come up below). Comma is redefined so it works inside `!‘`; outside that context it behaves as it normally does.

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 these 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. See `empty-list` in section 7.5.

2. `!"..."`: This is an ordinary string, except that any substring of the form "~*whitespace*" is deleted. (Something like the "~*newline*" `format` directive, but more robust with respect to the vagaries of newline representation in different operating systems.) The substring "~%" becomes a newline; and "~~" becomes a "~". Helpful for long string constants that start at a column awkwardly far to the right.

3. `!'`*s*: An abbreviation for `(funktion` *s*`)`, section 7.5

4. ``!`(...)``: The improved backquote (section 7.1).

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 ...)`. `\\` is an ordinary macro.

- "_" may be used in most contexts where a bound variable is expected; it denotes a variable whose value is ignored. In evaluation contexts, `_` evaluates to `nil`. See section 7.5

- `:||` is used in match patterns (section 3.2) to indicate a disjunctive pattern. The reader reads it as the symbol in the keyword package whose name is the empty string.

- `%` 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 "`%`."

Tables 2–4 list all of the nonalphabetic symbols used as or in YTools macros. Each entry gives the symbol, the context it occurs in, a one-phrase summary of its meaning, and a mnemonic that explains why the symbol should be easy to remember. If you have forgotten it anyway, a fuller explanation of its meaning is found somewhere in this manual; consult the index.

You may have noticed that the names of my global variables have an asterisk at the end, but not at the beginning. I developed this habit before the convention existed of putting an asterisk at both the bow and the stern of a global-variable name, and I never bothered to change. I do write constants as `+`*name*`+`, because an isolated plus would look like it had something to do with addition. By the way, a constant is just as likely to be defined with `defvar` and `defparameter` as `defconstant`. In many implementations, `defconstant` is so cantankerous as to be almost useless: compilers often refuse to generate code to load any constant besides a number or a boolean, and some systems make it almost impossible to redeclare a constant without killing Lisp and starting over. Having a naming convention such as `+...+` provides a more reliable way of telling the reader what the constants are; it's a pity it's so difficult to tell the compiler.

A.4 Coding style

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

Symbol	*Context*	*Meaning*	*Mnemonic*
`_`*(underscore)*	Various (See subsequent entries)	Ignorable, unimportant, place-holding, …	Looks blank, contentless, unbindable, unassignable, …
`:?`	`(match-cond` *e* `(:?` *p* `...)` `...)`	*p* is to be matched against *e*	Anticipating match variables
`?`	`?`*`var`*	Match variable	Asking for a value
`@`	`?@`*`var`*	Segment match variable	‘`@`’ means segment in ‘`,@`’
`,`	`?,`*`v`* or `?,@`*`v`*	Matches current value of *`v`*	‘`,`*`v`*’ means current value of *`v`* in backquote
`:||` *or* `:\\\|`	`?(:||` $p_1 \ldots p_k$`)`	Matches *exp* if one of p_1 to p_k matches	‘`|`’ means “or” in C
`:&`	`?(:&` $p_1 \ldots p_k$`)`	Matches *exp* if p_1 to p_k match	‘`&`’ means “and” in C
`:&`	`?(:|| ... :&` *q*`)`	In addition to one of the disjuncts matching, *q* must match as well	Ditto
`:~`	`?(:~` *p*`)`	Matches *exp* if and only if *p* does not	‘`~`’ often means “not”
`_`	`?_`	Ignored match variable	Looks unassignable
`:%`	`(out ... :% ...)`	Print newline	Looks like `~%`
`%`	`%`*`directory`*`/`	*`directory`* is YTools logical pathname	‘`%`’ looks like ‘`/`’
`:+`	`?(:+` *p* $r_1 \ldots r_k$`)`	Matches *exp* if *p* matches and all predicates r_i are true of *exp*	‘`+`’ connotes “in addition”
`:+`	`(extra-vals` *`e`* `:+` —*`vars`*—`)`	Update *`vars`* from extra values returned by *`e`*	‘`+`’ connotes “and also”
`:++`	`(build-symbol` `(:++` *k*`)` `...)`	*k* is incremented and its characters are included in symbol	`++` — just like in C!

Table 2: YTools symbols, installment 1

Symbol	*Context*	*Meaning*	*Mnemonic*
`!`	`!`*char*`...`	Meaning depends on character *char*.	`#` was taken.
`!'`	!'*fcn*	Like `#'` if safety/speed optimization is tilted toward speed; else like `'`	Looks like `#'`
`` !` ``	`` !` ``n`(... ,`n` ...)`	Super-duper backquote; n is a digit that allows us to match up backquotes with commas	It's excl-*backquote*
`!"` *and* `~`	`!"`*long* `~` *string*`"`	String that's allowed to stretch over multiple lines	Super-duper string
`!()`, `!(`t`)`	Expression	Empty list (of objects of type t)	Looks like an empty list (if t isn't too big)
`_`	Expression	Unimportant ("don't care") value	Looks contentless
`:_`	`(out ... (:_ `n`) ...)`	Insert n spaces	Looks like a space
`=`	`(loop for ((`v` = `i` ...)) ...)`	Initialize v to i in loop	Pretty obvious
`:i=`	`(out ... (:i= `n`) ...)`	Indent by n spaces	"i[ndentation] ="
`!=`	`(!= `e` ...)`	≈`setf`	Become equal!
`*-*`	`(!= `l` (... *-* ...))`	Value of left-hand side of assignment (l)	`*-*` looks like a blank spot into which l should be copied.

Table 3: YTools symbols, installment 2

Symbol	*Context*	*Meaning*	*Mnemonic*
`< >`	`(!= <...>` e`)`	Equivalent to `(setf (values ...)` e`)`	`<...>` looks like a tuple.
`< >`	`(!=(<...>)` e`)`	Sets variables to elements of list e.	Added parens connote list.
`_`	`(!= <..._...>` e`)`	Value is not stored	Looks unassignable
`:i>` / `i<`	`(out ... (:i>` n`) ...)`	Add (`:i>`) or subtract (`:i<`) n from indentation	`>` and `<` point right and left
`:^`	`(datafun` ***`task flag`*** `(defun :^ ...))`	Placeholder for absent function name	^ points up to actual name, ***`flag-task`***
`:<`	`(build-symbol (:<` s`) ...)`	Characters of s are included in symbol	`<` connotes expansion
`<`c	`(<`c f `...)`	Calls function f in some way that depends on character c.	Left angle-bracket is a warped left paren
`<#`	`(<#` f `...)`	= `mapcar`	Same number (#) of elements out as in
`<!`	`(<!` f `...)`	= `mapcan`	Destructive!
`<$`	`(<$` f `...)`	= `mappend` (like `mapcan`, but `append`s instead of `nconc`king)	Constructive, like capitalism
`<v`	`(<v` p l`)`	= `some`	`(`p l_1`) v (`p l_2`) v ...`
`<&`	`(<&` p l`)`	= `every`	`(`p l_1`) & (`p l_2`) & ...`
`<?`	`(<?` p l`)`	Returns elements of l that satisfy p	Querying p
`<<`	`(<<` f l`)`	`= apply`	Extra bracket for extra paren layer
`><`	`(><` f `...)`	= `(funcall` f `...)`	It's almost `apply`
`</`	`(</` f l`)`	Reduction operator	APL used "/" for reduce .... Remember? APL?
`\\`	*Query response*	Abort	Close to backspace key
`\\`	`(\\ ...)`	`(lambda ...)`	Looks like λ
`_`	Parameter list	Ignored parameter	Looks nameless

Table 4: YTools symbols, installment 3

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 more than one type of 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.

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 `is` has anything in front of it.

Dynamically bound (special) variables: Avoid them wherever possible. They make it hard to track down all the influences on a buggy program. Here are a few unavoidable uses:

- As repositories of pervasive debugging flags, printer settings, or the like.
- 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 called `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 objects of that type. 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 guide symbols to these two purposes. That is, a guide symbol `:gizmo` may either come before its argument(s), as in

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

   or as the first element of a macro subclause:

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

   I violate my own rule in a couple of places, notably in the variable declarations of the `repeat` (section 2.2), `exists`, and `forall` (section 7.5), where the guide symbols `=` and `:in` function as markers for subclauses, but are not the first elements of those clauses. I also violate it for the `-` symbol for `with-slots-from-class`(section 6). Type declarations are a special case; the use of `-` for them emphasizes that they are being overlaid on normal Lisp at compile time, and that they will fade away at run time. The same syntax is used for Nisp (**?**) and PDDL (**?**).

There is one rule I perhaps should follow, but don't: Make sure guide-symbol arguments fit the pattern required by `&key`, i.e., one argument per guide symbol, in a sequence of even length coming at the end of the macro call or one of its sublists. In particular, it seems okay to allow 0 or more than 1 argument to follow a guide symbol, even though it may make life more difficult for pretty printers.

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 a 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 unbearable). 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.

   A partial exception to this rule is to allow guide symbols to come from the `common-lisp` package, so that they will be virtually certain to be visible in users' code without any explicit importing. Examples are the `=` guide symbol for `repeat` (section 2.2) and the `-` symbol for `with-slots-from-class` (section 6). The exception applies only when the symbols can't possibly be confused with variables or functions. It's not a coincidence that both `=` and `-` are used as "infix" operators, in contexts where they can't be read as an equality test or a subtraction. A case could be made that they should have been put in the keyword package anyway, but `:=` connotes the wrong thing, and `:-` just looks "too big."

   I violate the rule completely for the flags used by `fload` and `fcompl` (section 9). These rules seem like Unix-style command-line arguments to me, and they are treated that way: In `-f` or `-x`, it's the *character* after the hyphen (`'f'` or `'c'`) that matters. The package `-f` was placed in before being that character was extracted is irrelevant (although the possibility of it having been upcased when read makes case insensitivity unavoidable for YTFM flags).

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`? (I can't.) I cringe every time I have to tack a "p" on the end of a predicate name.

There are certain functions that, while great from a nostalgia point of view, often conceal the programmer's intent, and, I'm sad to say, the chief culprits are `cons`, `car`, and `cdr` and functions built on them, such as `nthcdr` and `caddadr`. The problem with them is that list structures can be used to implement different data structures, including tuples, lists, and tree structures. A *tuple* is built out of cons cells, but is of fixed length, with each position occupied by a separate type. A *list proper* often starts off small and grows using `cons`. All its elements are usually thought of as being the same type (even if that type is some general category). 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. A list built by `tuple` should be decomposed using `first`, `second`, . . . rather than `car`, `cadr`, ... A real list should be decomposed using `head` and `tail`. Similarly, `pair` should be used to build binary trees, to be decomposed using `left` and `right`.

What role does that leave for `car` and `cdr`? I use them for decomposing S-expressions. S-expressions are usually built using backquote.

All these synonyms and several others are listed below in alphabetical order. 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.

`head` *Function* *Location:* YTFM

`head` is a synonym for `car`, to be used when its argument is to be considered a list rather than a tuple or S-expression. (See `tail`.)

`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-list`, `is-Number`, `is-Pair`, `is-Pathname`, `is-Ratio`, `is-Stream`, `is-String`, and `is-Symbol`.

`left` *Function* *Location:* YTFM

`left` is a synonym for `car`, to be used when its argument is to be considered a list rather than a tuple or S-expression. (See `right`.)

`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`.

`multiple-value-let`	*Macro*	*Location:* YTFM

`multiple-value-let` is a synonym for `multiple-value-bind`. It exists because (a) there is no reason to use "let" for one and "bind" for the other, since they bind in exactly the same way; (b) `bind` is used in YTools to bind special variables.

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

These are synonyms for `pop`, `push`, and `pushnew`, 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.

`nthrest`	*Function*	*Location:* YTFM
`nthtail`	*Function*	*Location:* YTFM

`(nthrest` n l`)` and `(nthtail` n l`)` are both synonyms for `(nthcdr` n l`)`, but `nthrest` should be used when l is a tuple, `nthtail` when l is a list. (Remark: `nthrest` returns a shorter tuple when n is greater than 0. This is something most other programming languages don't allow you to do, probably for good reasons. However, if a tuple was built using `tuple.`, then `nthrest` can be used to retrieve a legitimate field.)

`pair`	*Function*	*Location:* YTFM

`pair` is a synonym for `cons`, to be used when its value is to be considered a tree rather than a tuple, list, or S-expression. (See `left` and `right`.)

`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`.

`right` *Function* *Location:* YTFM

`right` is a synonym for `car`, to be used when its argument is to be considered a list rather than a tuple or S-expression. (See `left`.)

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

Synonyms for the corresponding functions with lowercase names.

`tail` *Function* *Location:* YTFM

`tail` is a synonym for `cdr`, to be used when its argument is to be considered a list rather than a tuple or S-expression. (See `head`.)

`tuple` *Function* *Location:* YTFM
`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, as explained at the beginning of this section. A list built by `tuple` should be decomposed using `first`, `second`, ... rather than `car`, `cadr`, ...

`tuple.` is a synonym for `list*`; it's used in contexts where some *tail* of the entity being built is being used as the last field. For such tuples, `rest` or `nthrest` should be used to access the last field.[10]

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

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

[10]One might suppose there should be a record-oriented synonym for `cons`, in terms of which `list*` is defined; but for records, unlike lists, the case of two arguments is not fundamental in any sense.

Index