TR-I-0247

The Beholder TOOLBOX Manual
Part 1
ビホルダ ツールボクス マニュアル
パート 1

John K. Myers
真龍主·星音

March 2, 1992

Abstract

This manual presents user documentation for the ATR Interpreting Telephony Research Laboratories BEHOLDER TOOLBOX package, Part 1. This package is designed to be a comprehensive general-purpose toolbox that supports parallel programming on the Sequent computer. The Part 1 manual presented here consists of new commands, explanations of problems with the Sequent CLiP system, and brief discussions of system timing information and agenda queueing theory. The commands are broken down into four main groups: commands extending the normal sequential Allegro Common Lisp, heap commands, new-flavors commands, and commands extending the parallel features of Lisp. The TOOLBOX commands are explained and grouped by topic in the front. An appendix in the back provides a dictionary listing the commands in alphabetical order.

Contents

1 INSTALLING THE BEHOLDER PACKAGE

The following lines should be installed at the end of the user's invisible `.clinit.cl` file in the user's home directory:

```
(require ":/usr/bhldr/beholder-1")           ;Basic system utilities.
(require ":/usr/bhldr/beholder-2")           ;Parallel utilities.
(require ":/usr/bhldr/beholder-heaps")       ;The heaps package.
(require ":/usr/bhldr/beholder-new-flavors") ;The new-flavors package.
```

(It is important to use the double-quote syntax, in order to support future filenames that might contain both upper and lower-case letters.) After this has been done, the user may automatically use all of the capabilities described in this manual from a `clip` shell or inside any `clip` programs.

In general, any of the packages may be omitted if the user does not need it, or if there is a naming conflict. However, it is best to load all of the packages.

2 DATA AND COMMAND EXPLANATION

This section presents a description of the system's data and commands. First, the types of data used by the system are described. Next, the most commonly used commands are detailed. The commands are arranged in the order in which they are typically used. After this, other support commands that can be used are listed, and important system variables are also described.

2.1 Beholder-1: Language Augmentation Commmands

There are a number of assorted functions that are useful and should be available in Common Lisp. Some of these come from Symbolics Lisp; some of them are just common-sense. The routines in this section are found in the file `beholder-1`.

2.1.1 String Commands

(string-append "string1" ... "stringN") Returns a string that is the concatenation of the previous strings. Example: `(string-append "Foo" "bar" "Baz") ==> "FoobarBaz"`

(f-string item) Coerces the item into a string, by returning its printed representation. The name comes from "forced" `string`—this is an improvement over the normal `(string)` function, which breaks when given numbers, and sometimes when given lists. This function should work no matter what `item` is. See also `trunc`.

(fstring item) This is the same as `(f-string)`.

(string-length item) This is an improved version of `length`, that tries to "do the right thing". It offers basic compatability with the Symbolics Lisp function of the same name. If `item` is a string, it returns a count of the number of characters in the string. If `item` is a char, it returns 1. If `item` is an array, e.g. an array of characters, it returns the length of the array. Otherwise, it uses `f-string` to coerce the value of the argument into a string, and then returns the length of that string.

(chararray-to-string chararray &optional(start 0)(end+1 (length chararray))) This function converts an array of characters into an equivalent string. The array must contain characters, not strings. The first optional argument indicates the array index containing the starting letter of the string (inclusive). The second optional argument indicates the array index of the ending letter of the string, plus one (exclusive).

```
my-array                              ==>  #(#\a #\b #\c)
(chararray-to-string my-array)        ==>  "abc"
(chararray-to-string my-array 1)      ==>  "bc"
(chararray-to-string my-array 0 2)    ==>  "ab"
```

(trunc item length) This function returns a string which is the name of the item, or a truncated version if the string would be longer than the given length. The item is forced to be converted into a name by using `(f-string)`. If the resulting name is longer than `length` characters, the name string is truncated to (length - 1) characters and a tilde character "~" is appended to indicate truncation; this resulting string is then returned. `item` can be just about anything. `length` must be an integer that is 1 or greater. This function does not pad the resulting string to ensure that it is exactly equal to length; it could be smaller. Note that this function returns a string, it does not do any printing; however, the results can be used as an argument to `format`. The function has been used under Symbolics Lisp for printing labels of graphical nodes.

(sys-make-name "STRING-1" ... "STRING-N") This macro first forms a name by concatenating the (one or more) given strings. It then searches the current name-space and returns the existing symbol which uses that name as its print-name. The resulting expression returns the variable; it must be eval'ed one more time to reference the contents of the variable. This code is useful for referencing variables if you will only know or be able to compute the name of an existing variable at runtime.

```
(setq foo2 0)
(sys-make-name "FOO2")                          ==>    FOO2    :INTERNAL
(set (sys-make-name "FOO" "2")  5)        ;note this is not a setq
foo2                                            ==>    5
(+  (eval (sys-make-name "FO" "O" "2")) 5)     ==>    10
(setq bar (sys-make-name "FOO2"))               ==>    FOO2
bar                                             ==>    FOO2
(eval bar)                                      ==>    5
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function only finds existing symbols. If the symbol has not been interned as part of the name-space yet, the function returns NIL.

```
(setq bar (sys-make-name "WEIRD"))              ==>    NIL
bar                                             ==>    NIL
```

(sys-make-new-name "STRING-1" ... "STRING-N") This macro first forms a name by concatenating the one or more given strings. It then searches the current name-space and returns the symbol which uses that name as its print-name, if it exists; otherwise, it creates a new named symbol and returns that. The resulting expression returns the variable; it must be eval'ed one more time to reference the contents of the variable. This code is useful for making new variables.

```
(setq foo2 0)                                    ==>    0
(sys-make-new-name "FOO2")                       ==>    FOO2     NIL
(set (sys-make-new-name "FOO" "2") 5)          ;note this is not a setq
foo2                                             ==>    5
(+  (eval (sys-make-new-name "FO" "O" "2")) 5)   ==>    10
(setq bar (sys-make-new-name "FOO2"))            ==>    FOO2
bar                                              ==>    FOO2
(eval bar)                                       ==>    5
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function finds existing symbols. If the symbol has not been interned as part of the name-space yet, the function returns a new symbol.

```
(setq bar (sys-make-new-name "WEIRD"))       ==>    WEIRD
bar                                          ==>    WEIRD
(set bar 20)                                 ==>    20
weird                                        ==>    20
```

(sys-make-keyword "STRING-1" ... "STRING-N") This macro first forms a keyword name by concatenating the (one or more) given strings. It then searches the current name-space and returns the existing keyword symbol which uses that name as its print-name. The resulting expression returns the keyword variable. This code is useful for referencing keywords if you will only know or be able to compute the name of an existing keyword at runtime. The user should *not* include the ":" character in the keyword definition. A keyword symbol should evaluate to itself.

```
(sys-make-keyword "STR" "EAM")               ==>    :STREAM   :EXTERNAL
(setq bar (sys-make-new-name "STREAM"))      ==>    :STREAM
bar                                          ==>    :STREAM
(eval bar)                                   ==>    :STREAM
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function only finds existing keyword symbols. If the symbol has not been interned as a keyword yet, the function returns NIL.

```
(setq bar (sys-make-keyword "WEIRD"))        ==>    NIL
```

(sys-make-new-keyword "STRING-1" ... "STRING-N") This macro first forms a keyword name by concatenating the one or more given strings. It then searches the current name-space and returns the keyword symbol which uses that name as its print-name, if it exists; otherwise, it creates a new keyword symbol and returns that. This code is useful for making new keywords. The user should *not* include the ":" character in the keyword definition. A keyword symbol should evaluate to itself.

```
(sys-make-new-keyword "MY-" "STR" "EAM")       ==>  :MY-STREAM    :EXTERNA
(setq bar (sys-make-new-keyword "MY-STREAM")) ==>  :MY-STREAM
bar                                            ==>  :MY-STREAM
(eval bar)                                     ==>  :MY-STREAM
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function finds existing keyword symbols. If the keyword symbol has not been interned as part of the name-space yet, the function returns a new keyword symbol.

```
(setq bar (sys-make-new-name "WEIRD"))      ==>    :WEIRD
bar                                         ==>    :WEIRD
(eval bar)                                  ==>    :WEIRD
```

2.1.2 Basic Commands

(type-the-time &optional (stream T)) Types out the time in a pretty format. Returns NIL. If the optional stream argument is NIL, does not type the time out, but returns a string of what it would have typed out.

```
(type-the-time nil)   ==>  "4:47:26 PM   Thur  Jun 20, 1991"
```

(neq a b) Returns NIL if a is eq to b, T otherwise.

(beep &optional (stream T)) Prints a Cntrl-G on the given stream. This rings the bell on most terminals, including the Symbolics. Example: (beep)

(alarm1 &optional (stream T)) Prints five beeps on the given stream.

(div2 x) Integer divide-by-2. Returns the integer representation of the number x represented as a signed binary number, and then shifted right one place, filling the sign bit. Takes floating-point numbers as input, but basically forgets the decimal. This function does the right thing when working with binary negative numbers. Note that this might not be what you'd expect if you didn't think about it:

```
(div2  4) ==>   2    [0100  ==>  0010]
(div2  3) ==>   1    [0011  ==>  0001]
(div2 -4) ==>  -2    [1100  ==>  1110]
(div2 -3) ==>  -2    [1101  ==>  1110]
```

(intern-soft string &optional package) This is the Symbolics-compatable version of the Common Lisp `find-symbol` function. It returns the symbol associated with the string, or NIL if the symbol has not yet been interned. It is used by `sys-make-name`.

(boundpq varname) This is the literal version of `boundp`. It does not evaluate its argument. It does not break if its argument is unbound.

```
(setq foo 2)        ==>  2
(boundpq foo)       ==>  T
(boundp  weird)     ==>  ERROR: Attempt...
(boundpq weird)     ==>  NIL
```

(boundqp varname) Same as `boundpq`.

(second-value multi-valued-function) This function returns the second value of a multiple-valued function. As a bonus, the first value of the multiple-valued function is returned as a second value from this function call.

```
(second-value   (values 1 2))     ==>  2    1
```

(++ number-loc) This function is a different name for `incf`. It pulls a number out of a general variable-location, increments the number by 1, puts the results back into the same place, and returns the incremented results. The number can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`++l`) described in Section 2.4.1.

(+= number-loc increment) This function is a different name for `incf`. It pulls a number out of a general variable-location, increments the number by the given increment, puts the results back into the same place, and returns the incremented results. The number and/or the increment can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`+=l`) described in Section 2.4.1.

(-- number-loc) This function is a different name for `decf`. It pulls a number out of a general variable-location, decrements the number by 1, puts the results back into the same place, and returns the decremented results. The number can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`--l`) described in Section 2.4.1.

(**-= number-loc decrement**) This function is a different name for `decf`. It pulls a number out of a general variable-location, decrements the number by the given decrement, puts the results back into the same place, and returns the decremented results. The number and/or the decrement can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`-=l`) described in Section 2.4.1.

(***= number-loc multiplicand**) The multiply-in-place macro. This macro pulls a number out of a general variable-location, multiplies the number by the given multiplicand, puts the results back into the same place, and returns the new results. The number and/or the multiplicand can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

Currently there is no locked version of this macro.

(**/= number1 .. numberN**) Surprise! (`/=`) is the system-defined *not-equal* function. There is no in-place div-equals function as of yet. Use (`*= X (/ 1 Y)`) for now, and let me know if you'd like a better one.

2.1.3 List Commands

(**squish nested-list**) This function takes a simple nested list and non-destructively returns a copy of the same list without all of the interior parenthises–the list gets "squished" down to one level. Order is preserved. Given a non-list atom, the function returns a list of that atom.

```
(setq foo '(a ((b (c) d))) )          ==>  (A ((B (C) D)))
(squish foo)                          ==>  (A B C D)
foo                                   ==>  (A ((B (C) D)))
(squish 'a)                           ==>  (A)
```

(**only-once list**) This function nondestructively returns a copy of the given list in which every atom is listed only once–duplicates are eliminated. The function uses `eq` for comparison. The resulting order is reversed. This is unfortunately an $O(\frac{1}{2}n^2)$ operation.

```
(only-once '(a a a b b b a c b c c c))   ==>  (C B A)
```

(**apush key item alist**) Assoc-list push. Pushes a new assoc entry (`key . item`) onto the given assoc-list.

(pullq inlist atom1 ... atomN) Destructively Pulls atoms onto the back of the argument `inlist`, in place. Returns the new list. Designed to complement `push`, which puts things on the front of the list. Note carefully that the list is eval'ed, but that the atoms aren't.

```
(setq x '(a b c))  ==>  (a b c)
(pullq x d e)      ==>  (a b c d e)
x                  ==>  (a b c d e)
(setq x NIL)       ==>  NIL
(pullq x d e)      ==>  (d e)
x                  ==>  (d e)
```

This new, improved version of pullq is just a hair slower but it does the right thing when `x` is NIL. It also does the right thing in list-bashing a new copy of the atoms onto the back, avoiding those kinds of strange stack problems.

Please do NOT use this function to pull local variables (from a `let` or from a function's arguments) onto the back of a list, and then exit from the local lexical definition. You will reuse part of the Lisp stack that is being assigned to something else, and you will be sorry.

If `x` isn't a list you've got problems.

(pull inlist item1 ... itemN) Destructively Pulls items onto the back of the argument `inlist`, in place. Returns the new list. Designed to complement `push`, which puts things on the front of the list. Note that both the list and the items are eval'ed.

```
(setq x '(a b c))  ==>  (a b c)
(setq y '(d f))    ==>  (d f)
(pull x (cdr y) 'e)      ==>  (a b c (f) e)
x                  ==>  (a b c (f) e)
(setq x NIL)       ==>  NIL
(pull x 'd 'e)     ==>  (d e)
x                  ==>  (d e)
```

This new, improved version of pull is just a hair slower but it does the right thing when `x` is NIL. It also does the right thing in list-bashing a new copy of the atoms onto the back, avoiding those kinds of strange stack problems. Also, since this macro evaluates all of its arguments, you would have to be really creative to get yourself into problems storing local variables on a global list. It should be safe.

If `x` isn't a list you've got problems.

2.1.4 File Commands

These commands implement a primitive-style output-file stream manager. The commands automatically open and close the stream `OS` (for "Output Stream"). There can be only one OS stream open at a time. It is up to the user to ensure that the opened stream gets closed–there are no unwind-protects used to auto-ensure this. If the user aborts out of a program in the middle, he or she should do a `(close-file)` to make sure that things are alright.

Another convention, which does not need routines to be supported, is to use the stream `ES` for errors. The programmer can write all error-message format commands using the stream ES. This stream will normally be directed at the Listener terminal `T` [i.e., ES=T], instead of into the output file. However, if the user wants the output file to record the error messages too, the error stream can be set equal to the output stream [ES=OS] by `(setq ES OS)`.

(open-file filename) Opens output stream `OS` for serial output into a new copy of the given file. `filename` should evaluate to a string or file descriptor indicating the appropriate file. This function breaks if `filename` indicates a nonsense path, e.g. to a machine that does not exist. The scratch variable `*using-file*` is set to the string or value passed in `filename`. When this function is called from a terminal (and not from inside a running program), it prints out a reminder message. No useful value is returned if this function is called from within a program; however, `OS` is set to a legitimate file stream on a successful call, as a side-effect. In general, it is important to specify the full pathname of the file; the default directory on the Sequent seems to be `/usr`, which should not be used by normal users, and will probably give you a Permission Denied error anyway.

Franz Lisp does not seem to support remote files. Certainly the `"MACHINE:pathname"` colon syntax is NOT supported for at least output files.

Note that `pformat` supports the use of various streams, including `ES` and `OS`.

```
(open-file "/usr1/myers/recording.text")

"Opened stream #<stream writing /usr1/myers/recording.text @ #x9294e9>
Do a (setq ES OS) if desired.
"
```

(use-file filename) Opens a file for output to OS. The same function as `open-file`.

(close-file) Closes an existing OS file. Prints an error message to the terminal if it is called twice, or if the file is already closed. Resets both `OS` and `ES` to `T`. Returns the string or descriptor that was stored in `*using-file*`. Does not take any arguments.

```
(close-file)

"/usr1/myers/recording.text"
```

(file-p stream) This function is supposed to test whether a given stream is an output file or not. Currently it tests to see whether a file is not the terminal IO stream or if it is, which gives similar but not precise answers. If you want to use this, please talk to me.

(filep stream) The same as `(file-p stream)`.

(with-open-file (IS input-filename :direction :input) BODY) This system-defined function is the approved method of reading from an input file stream IS. It uses an unwind-protect to ensure that the input file is closed if an error occurs in the body of the routine.

OS This variable is used as the Output-file Stream. You can use it for all output in your program. It is bound to `T` as a default. It is used by routines `open-file` and `close-file`.

ES This variable can be used as the Error Stream. You can use it for all minor error-message output in your program. It is bound to `T` as a default. Perform a `(setq ES OS)` after opening a file if you want error messages to go to the recording output file. Note that major error messages should use the stream T to the terminal; otherwise, the user will not be notified when something bad has happened. ES is reset to T by routine `close-file`.

using-file This variable is bound to the string or descriptor that was used to open the currently open file by routine `open-file`. It is used for information purposes only.

2.1.5 Benchmarking and Testing Commands

(exact-string-test function-call expected-results-string) This macro is used for automatically testing functions that are supposed to return well-known answers. For instance, if a routine is known to work properly, but then the code gets changed, it is useful to run a suite of test programs on the routine in order to check out that it still works correctly. Exact-string test accepts a function call, executes the function, and gathers all of the output to stream OS into an internal string variable. It then does a `string-equal` comparison against the expected-results string, which should exactly correspond. If the strings are equal, the routine prints a small verification message, and returns T. If the strings are not equal, the routine prints a large complaining message, and prints out the output that was actually obtained; it then returns NIL. Watch out for spaces and carriage returns in the expected string. It is important that the tested routine send its output to OS; the results that are *returned* by the tested routine are not examined. Examples:

```
(defun my-func (x) (format OS "Results: ~A" (* x x)))
(exact-string-test (my-func 3)  "Results: 9")
  Exact-string-test:  (MY-FUNC 3) passes the test.    ==> T
(exact-string-test (my-func 3)  "BadMatch")
  EXACT-STRING-TEST:  (MY-FUNC 3) FAILS THE TEST.
  Actual output:
  Results: 9       ==> NIL
```

(burn-cycles iteration-count) This command can be used for microsecond timing. See section 2.4.2.

2.2 Beholder-New-Flavors: A Flavors Syntax Package

The Beholder New Flavors package is found in the `/usr/bhldr/beholder-new-flavors` file.

Object-oriented programming is very useful, if not required, for many applications. A good, clean, object-oriented syntax makes programs easy and faster to read, write, and debug. There have been a number of different syntaxes developed for supporting object-oriented programming. The CLiP compiler currently supports the Symbolics Old Flavors syntax, which is build on sending "messages" and "set-messages" to programs. About seven years ago, people decided that this syntax was too clumsy and hard to read, and Symbolics replaced this with the cleaner so-called "New Flavors" syntax based on function calls and using `setf` to set slot variables. The Beholder New Flavors package supports this syntax. Programmers can define objects, define methods, access slot variables, set slot variables, and call methods, in the manner defined by the Symbolics New Flavors package. In addition, the Beholder New Flavors package implements some helpful *option codes* for frequently-used options, which make the programs even easier to type and to read. These are defined below.

The command definition section is not intended to be an introduction to the advantages and uses of flavors and methods. For a good discussion of this, see pages 85-206 of the Symbolics Common Lisp Programming Constructs Book 8, or pages 368-490 of the old Symbolics Common Lisp Language Concepts 2A. Read the introductory sections in these chapters for a brief overview.

The current New Flavors package is implemented on top of the CLiP old flavors package. It works by defining macros which translate into old flavors calls. This means that programs can use both New Flavors and old flavors at the same time. This is very important, because some CLiP system routines[2] are written using the old-style flavors, and these would not work if New Flavors and the old flavors were incompatible.

However, for this reason, it is currently necessary to use different names for the New Flavors functions. Therefore, the flavor defining function is known as `defflav`, and the method defining function is known as `defmeth`.

This introduces some incompatibilities with the Symbolics machine. So, the `beholder-new-flavors` file also supports the Symbolics. If the

[2]Especially `defflavor`, which quietly calls `defmethod`.

beholder-new-flavors file is loaded on a Symbolics machine, then files using defflav and defmeth can be run on the Symbolics, too. The file is implemented using machine-dependent compilation, so the same source file that the Sequent uses can be loaded. The Symbolics version of Beholder New Flavors also supports the option codes.

The Beholder New Flavors package requires the use of beholder-1. This file must have been previously loaded in order to work.

2.2.1 New Flavor Commands

Flavor and Method Definition Commands

(defflav flavor-name (slot-vars) (parents) :options :option-codes)

Defines a flavor named flavor-name that inherits from parents, uses the given slot variables, and is built using the given (optional) options and/or option codes. Example:

```
(defflav ship   ( x  (y 0) ) (vehicle thing) :C)
(defflav rocket ( x  y  z  ) (vehicle thing) :D)
```

This defines a flavor ship. It has slots x and y. The y slot has a default initialization of 0; the x slot has no default initialization. The ship inherits slots from parent flavors vehicle and thing. Note multiple parents are no problem. Often the parent list will simply be nil: (). The ship is defined using option code :C, which, as explained below in section 2.2.1, makes y and x readable, writable, and initable, and creates the creation function (make-ship) for creating instances of the flavor.

It is recommended that the option flag :C be used in most cases.

(defmeth (method-name flavor-name) (arg1 ... argN) &body) Defines a method that is used by the given flavor. It is permissable to have different flavors use the same method name; the system automatically determines which method is correct based on what flavor of object is used in the call. The different flavor methods can even have a different number of arguments. As usual, the slot variables of the flavor instance are local variables inside the method and can be referenced and setq'd directly; there is no need to use the access functions inside the method. Also as usual, the special local variable self is bound to the flavor instance. The body can be a sequence, it does not have to be a list.

Examples:

```
(defmeth (move ship) (new-x new-y)
  "This method moves the ship to a new (x,y) coordinate."
(setq x new-x)
(setq y new-y)
)
```

```
(defmeth (move rocket) (new-x new-y new-z)
  "This method moves the rocket to a new (x,y,z) coordinate."
(setq x new-x)
(setq y new-y)
(setq z new-z)
)
```

This shows that different flavors can have methods with the same name.

Flavor Instance Creation Commands

(make-flavorname &optional :slotvar1 init1 ... :slotvarM initM) Instance creation routine used in the New Flavors package if the option-code :C is used in `defflav`. Creates an instance of the given flavorname. Example:

```
(setq my-ship  (make-ship :x 5) )
```

Any slot variables that are not specified receive their default initialization values. Any slot variables that are not specified and do not have initialization values are set to NIL, they are not left unbound.

(flavorname &optional :slotvar1 init1 ... :slotvarM initM) Instance creation routine used in the New Flavors package if the option-code :D is used in `defflav`. Creates an instance of the given flavorname. Example:

```
(setq my-ship  (ship :x 5) )
```

Any slot variables that are not specified receive their default initialization values. Any slot variables that are not specified and do not have initialization values are set to NIL, they are not left unbound.

(make-instance 'flavorname &optional :slotvar1 init1 ... :slotvarM initM) Instance creation routine used by the old flavors package. This is the default method of constructing a flavor instance that is used if the :C or :D option code is not specified. Also, even if the :C or :D code is used for a flavor, this syntax is still valid and can be mixed with that of the New Flavors package.

This function creates an instance of the given flavorname. Example:

```
(setq my-ship  (make-instance 'ship :x 5) )
```

Any slot variables that are not specified receive their default initialization values. Any slot variables that are not specified and do not have initialization values are set to NIL, they are not left unbound.

Method Invocation Commands

(methodname flavor-instance arg1 ... argN) Invokes the method function on the given flavor instance object. Which method function is invoked depends upon which flavor the object is an instance of. Example:

```
(move my-ship 10 20)      ==> 20
```

This calls method `move` on the object `my-ship`. Since `my-ship` is an instance of flavor `ship`, it uses the `move ship` method defined in the previous `defmeth` example. The ship x and y slots are set to 10 and 20 respectively; the method returns the last line in the method definition, which evaluates to 20.

Slot-Variable Referencing Commands

(slotvarname flavor-instance) This is the normal method of referencing a slot's value. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were used. Example:

```
(x my-ship)      ==> 10
```

(flavorname-slotvarname flavor-instance) This is the default method of referencing a slot's value, as provided by the New Flavors package. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were **NOT** used. Example:

```
(ship-x my-ship)      ==> 10
```

(send flavor-instance :slotvarname) This is the default method of referencing a slot's value, as provided by the old flavors package. This syntax is still valid, even if the New Flavors package is being used. Example:

```
(send  my-ship :x)      ==> 10
```

Slot-Variable Setting Commands

(setf (slotvarname flavor-instance) newvalue) This is the normal method of setting a slot's value. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were used. Example:

```
(setf (x my-ship) 10)      ==> 10
```

(setf (flavorname-slotvarname flavor-instance) newvalue) This is the default method of setting a slot's value, as provided by the New Flavors package. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were **NOT** used. Example:

```
(setf (ship-x my-ship) 10)      ==> 10
```

(SET-slotvarname flavor-instance newvalue) This is a bonus function for setting a slot's value that is automatically defined by the New Flavors package. This syntax is usable for any flavor, New Flavors or not, as long as the New Flavors package has been loaded. This syntax is still valid, even if the New Flavors package is being used.

```
(set-x my-ship 10)      ==> 10
```

(send flavor-instance :SET-slotvarname newvalue)
This is the default method of setting a slot's value, as provided by the old flavors package. This syntax is still valid, even if the New Flavors package is being used. Example:

```
(send  my-ship :x)      ==> 10
```

Flavor Description Commands

(describe flavor-instance) This is the system command for printing out the slots and slot values of a flavor instance.

Defflav Option-Code Descriptions

:A The :A option to defflav sets all variables to set, get, and init. It is equivalent to typing the flags `:settable-instance-variables :gettable-instance-variables :initable-instance-variables`.

:all Same as :A. The :all option to defflav sets all variables to set, get, and init. It is equivalent to typing the flags `:settable-instance-variables :gettable-instance-variables :initable-instance-variables`.

:B The :B option to defflav sets all variables to set, get, and init, the same as option :A. In addition, it sets the conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(x my-ship)`.

:C The :C option to defflav sets all variables to set, get, and init, and also sets the conc-name to NIL, the same as option :B. In addition, it sets the constructor function name to `make-`*name*. So, for instance, to create an instance of a ship, the user types `(make-ship)`. This option is recommended.

:D The :D option to defflav sets all variables to set, get, and init, and also sets the conc-name to NIL, the same as option :B. In addition, it sets the constructor function name to *name*. So, for instance, to create an instance of a ship, the user types `(ship)`.

:concnil This option to defflav sets the flavor conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(x my-ship)`.

:nilconc This option to defflav sets the flavor conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(x my-ship)`.

(:conc-name newname) This option to defflav sets the flavor conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(newname-x my-ship)`.

:readable-instance-variables Option to defflav. Translates the Symbolics syntax into :gettable-instance-variables. Makes all slot variables gettable.

:read Option to defflav. See :readable-instance-variables. Short for :gettable-instance-variables.

:get Option to defflav. See :readable-instance-variables. Short for :gettable-instance-variables.

:writable-instance-variables Option to defflav. Translates the Symbolics syntax into :settable-instance-variables and :getable-instance-variables. Makes all slot variables settable and gettable.

:write Option to defflav. See :writable-instance-variables. Short for :gettable-instance-variables and :settable-instance-variables.

:set Option to defflav. See :writable-instance-variables. Short for :gettable-instance-variables and :settable-instance-variables.

:init Option to defflav. Short for :initable-instance-variables. Makes all slot variables initable.

(:readable-instance-variables sequence-of-slotnames) Option to defflav. Translates the Symbolics syntax into :gettable-instance-variables. Makes the given slot variables gettable.

(:read sequence-of-slotnames) Option to defflav. See (:readable-instance-variables). Short for :gettable-instance-variables.

(:get sequence-of-slotnames) Option to defflav. See (:readable-instance-variables). Short for :gettable-instance-variables.

(:writable-instance-variables sequence-of-slotnames) Option to defflav. Translates the Symbolics syntax into :settable-instance-variables and :getable-instance-variables. Makes the given slot variables settable and gettable.

(:write sequence-of-slotnames) Option to defflav. See (:writable-instance-variables). Short for :gettable-instance-variables and :settable-instance-variables.

(:set sequence-of-slotnames) Option to defflav. See (:writable-instance-variables). Short for :gettable-instance-variables and :settable-instance-variables.

(:init sequence-of-slotnames) Option to defflav. Short for :initable-instance-variables. Makes the given slot variables initable.

2.3 Beholder-Heap: A Simple Heap Package

A "heap" is a data-structure similar to a stack, except that each entry is tagged with a priority. Instead of being ordered in a first-in, first-out relationship, the entries in a heap are ordered by their priorities, with the *lowest* number belonging to the first-out item on top of the heap. (Note that if two items have the same priority, there is no guarantee as to which order the system will choose.)

A heap is a very common, useful data structure. The Allegro CLiP system by itself does not support heaps. It was necessary to write a simple heap facility to allow this behavior.

The routines in this section are found in the file `beholder-heap`. Heap functions are currently not used by the `beholder-1`, `beholder-2`, nor `beholder-new-flavors` files. The Heap system is independent and does not use routines supplied in other files.

2.3.1 Heap Commands

(heap) Returns an empty heap. Same as `(make-heap)`. Currently an empty heap is implemented as an empty list `()`.

(make-heap) Returns an empty heap. Same as `(heap)`. Currently an empty heap is implemented as an empty list `()`. Be careful; "make-heap" returns a Symbolics heap structure when this code is run on the Symbolics computer.

(push-heap heap key item) Pushes the given item on the given heap, using the given key. The key *must* evaluate to a number.

(pop-heap heap) Pops the top item off the heap (the item with the *lowest* number as its key). Returns two values: the item, and also its priority key. Returns `NIL NIL` if the heap is empty. The popped item is removed from the top of the heap.

(top-of-heap heap) Returns two values: the top item of the heap, and also its priority key. Returns `NIL NIL` if the heap is empty. The top is the item with the *lowest* number as its key. The top item is *not* removed from the top of the heap.

(find-heap-item heap key item) Searches for the given item and key inside the given heap. Both the item and the key must match, using `equal`. Returns a heap object representing the subheap that the item was found in. Returns NIL if the item was not found. The subheap object is replaceable in the main heap using setf, for heap twiddling. The searched-for item is *not* guaranteed to be at the top of the returned subheap object.

(heap-find-item heap item) Searches for the given item inside the given heap. The item must match, using `equal`. Returns a heap object representing the

subheap that the item was found in. Returns NIL if the item was not found. The subheap object is replaceable in the main heap using setf, for heap twiddling. The searched-for item is *not* guaranteed to be at the top of the returned subheap object. This function searches exhaustively and will take longer than find-heap-item.

(pull-heap heap) Pulls the bottom item off of the heap (the item with the *highest* number as its key). Returns two values: the item, and also its priority key. Returns `NIL NIL` if the heap is empty. The pulled item is removed from the bottom of the heap.

(bottom-of-heap heap) Returns two values: the bottom item of the heap, and also its priority key. Returns `NIL NIL` if the heap is empty. The bottom is the item with the *highest* number as its key. The bottom item is *not* removed from the bottom of the heap.

(top-of-heap-key heap) Returns the lowest number in the heap. Returns NIL if the heap is empty.

(bottom-of-heap-key heap) Returns the highest number in the heap. Returns NIL if the heap is empty.

(delete-from-heap heap key item) Deletes a given keyed item from the heap. Returns two values: the deleted item, and also its key. Both the key and the item must match the corresponding entry in the heap with `equal`.

(rekey-heap-item heap key item newkey) Labels an existing item in the given heap with the given new key; reorders the heap to reflect the new status. The new key must be a number. Both the old key and the item must match the corresponding entry in the heap with `equal`. This routine should be faster than deleting the item from the heap and then pushing it in again with the new key.

(heap-empty-p heap) Tests to see whether a given heap is empty or not. Returns T or NIL.

(heap-full-p heap) Tests to see whether the given heap has at least one entry or not. Returns the heap or NIL. This is the preferred test.

(clear-heap heap) Clears a given heap out; makes it empty. Currently this is implemented by setting the variable to a new heap.

(list-of-heap-items heap) Returns a list of the items in the heap. The list is ordered from lowest to highest, by key.

(list-of-heap-keys heap) Returns a list of the keys of the items in the heap. The list is ordered from lowest to highest, by key.

(list-of-heap-items-and-keys heap) Returns a list of pairs of (item key), the items in the heap paired with their keys. The list is ordered from lowest to highest, by key.

(heap-top-N-items heap N) Returns a list of the top N items with the lowest keys, and a second value of the number of entries returned. The list is ordered from lowest to highest, by key.

(heap-bottom-N-items heap N) Returns a list of the bottom N items with the highest keys, and a second value of the number of entries returned. The list is ordered from highest to lowest, by key.

(heap-top-Nth-item heap N) Returns the item that is Nth from the top counting up from the lowest entry, and a second value of its ranking, which will usually be N. If the heap has less than N entries, the bottommost entry is returned.

(heap-bottom-Nth-item heap N) Returns the item that is Nth from the bottom counting down from the highest entry, and a second value of its inverse ranking, which will usually be N. If the heap has less than N entries, the topmost entry is returned.

(size-of-heap heap) Returns a count of the number of items in the heap.

2.3.2 Implementation notes

In version 1.0, heaps are currently implemented using simple lists instead of structures. This makes the heaps fast, but does not support `describe`. The current implementation stores heaps using binary trees. No attempt is made to balance the tree. Heap entries are of the form `(key entry left-subtree right-subtree)`. An empty heap consists of the empty list, `()`.

2.4 Beholder-2: Parallel Language Augmentation Commands

The following commands are found in the `beholder-2` file (see Section 1).

2.4.1 Lock Commands

The Lock commands are included in the `beholder-2` file.

Locked Counter Increment/Decrement Commands One of the most basic functions that a computer can do is to increment or decrement a counter, either by 1 or by a specified number. However, when the computer is actually a parallel multi-processor, even a simple increment can cause a write-write collision. These functions circumvent this problem by supporting operations on locked counters. The user provides both a counter location and a lock dedicated to that counter. The command grabs the lock, performs the requested operation on the counter, and returns the new resulting value. It is recommended, although not necessary, that the counter's contents be of the integer type. The counter-lock should be a spin-lock. Since these commands use something similar to a `setf`, it is not necessary for the counter to be a variable name–any esoteric locator (e.g., an array reference, or

a slot in a def'd structure) will perform the proper operation. The functions are intentionally named after the C language functions of similar operation, with the addition of the "l" for "lock".

(++l counter-location counter-lock) Grabs the lock and increments the given counter by 1. Returns the new value.

(– –l counter-location counter-lock) Grabs the lock and decrements the given counter by 1. Returns the new value.

(+=l counter-location incr counter-lock) Grabs the lock and increments the given counter by the given incr. Returns the new value.

(–=l counter-location decr counter-lock) Grabs the lock and decrements the given counter by the given decr. Returns the new value.

Lock Extension Functions In some rare cases, there are times when a lock is either required or not required at all. In these cases, you want the computer to use the lock if it's there, and don't worry about it if it hasn't been allocated. The following function supports this behavior.

(with-spin-lock-or-NIL lock body...) This enclosing macro supports optional locks. If `lock` is NIL, the macro goes ahead and evaluates the body. If `lock` is a lock, the macro is the same as a `with-spin-lock` form.

2.4.2 Time Commands

The Time commands are included in the `beholder-2` file.

(get-time) Returns the "internal real time", in internal-tick units (milliseconds). This command is only accurate to a few hundreths of a second.

There is a rumour that the time is different when asked between different processors. This is currently being investigated.

(get-elapsed-time start-time-in-internal-ticks) Returns the amount of time that has elapsed, in seconds, from the time that the start-time was recorded. Start-time must be in internal-tick units. Unfortunately, this function is only accurate to a few hundreths of a second–however, a basic operation takes about a microsecond to perform. Thus, this function is useless in timing anything that does not have more than about 5000-10000 basic operations in it. [Besides this, since it uses simple subtraction, there may be a possibility that this function will break once a month. E.g., for five seconds at midnight on the 31st, this function will return wrong answers. Do not use it to time elapsed durations of more than about a day.] See also `burn-cycles`.

Example:

```
(setq old-time (get-time))
...
(when (> (get-elapsed-time old-time) 5.0))
   (pformat T "You're taking longer than five seconds to think!!!~%"))
```

(wait nsecs) Wait implements a "sit-and-spin" function that ties up a processor until the given number of seconds has elapsed. Wait uses `get-elapsed-time` and should not be used to time durations of more than about a day, to be safe. **Wait can only be used safely for durations with hundreths of seconds precision.** It fixes the problem introduced by `sleep`, which quietly truncates its argument down to integer values and is basically useless. See the discussion in Section 3.4.

(burn-cycles iteration-count) This routine does nothing but use up computer time. It counts up to the given total. Since each iteration takes almost exactly 1.925 microseconds to execute, this command can be used for microsecond timing.

2.4.3 Random Number Commands

The Random Number commands are included in the `beholder-2` file.

The system's random-number generator produces exceedingly strange values when used by multiple lwps. It is necessary to use a locked version in order to get usable numbers.

(rand x) This is a locked version of the CL function `(random x)`, which takes a positive number x and returns a number of the same type from 0 (inclusive) up to but not including x. I.e., if x is an integer, numbers from 0 to (x-1) are returned; if x is a floating-point number, numbers from 0 to $(x - \epsilon)$ are returned. It is an error not to supply the argument x when calling this function.

2.4.4 Normal Mailbox Commands

The Normal Mailbox commands are included in the `beholder-2` file. See also the Fast Mailbox commands, which will be presented in Part 2 of the Beholder Toolbox manual.

(flush-mailbox my-mailbox &optional (mailbox-spinlock NIL)) This command first grabs the spinlock, if it exists, and then empties the current contents of the mailbox by repeatedly receiving messages until the box is empty. The contents of the mailbox are lost. The spinlock is released after the mailbox is empty. It is not necessary to have a lock for the mailbox. The function returns the number of messages that were thrown away.

Example:

```
(flush-mailbox parser-mailbox)
```

2.4.5 Resource Commands

In certain applications, the memory space allocated by the system may be especially tight, and memory space management may become particularly acute. In such cases, it is useful for the user to be able to allocate and deallocate his own items in memory,

rather than trusting to the garbage collection of the system to perform this task for him or her properly.

Symbolics Lisp has a well-developed system known as "resources" that performs this task. The Resources section of the Symbolics manual may be referenced for further information on the general flavor of the types of operations required in a resources package. Version 1.0 of the BEHOLDER package implements an (incompatable) version of a resources package that is simpler to use, but less powerful. The user is only allowed to allocate and deallocate structures, (not other objects such as arrays, etc.). The package is invoked by using `defstruct-resource` exactly in place of the normal `defstruct`. This automagically defines functions `allocate-`*foo* and `deallocate-`*foo*, where *foo* is the name of the structure defined using the `defstruct-resource`. The user should use `allocate-`*foo* in place of the normal call to `make-`*foo*, to create an instance of the structure. In version 1.0, this function takes no initialization arguments. It is necessary for the user to initialize all the slots of the structure by himself, using `setf`. In particular, the contents of the structure slots will most probably contain leftover garbage that could damage the user's data structures if it is left there. When the user is done with a particular instance of a resource structure, and wants to explicitly garbage-collect the instance, the user should call `deallocate-`*foo* on the instance.

All the options normally used with `defstruct` can be used with `defstruct-resource`. However, it does not make much sense to specify initializers for the slots, as these may or may not be used when a new structure is allocated.

It is completely possible to specify many different types of resources. Each type of resource is handled separately by its own allocate and deallocate calls.

If the structure is redefined using another `defstruct-resource` call, all the previously deallocated garbage structures under that resource name are lost, and the system starts with a new storage stack. Apart from using up more space, this behavior should not affect the user in any way.

For reasons of speed, no checking is performed on a deallocation call to see whether the structure instance has been deallocated or not. It is a serious error to deallocate the same structure instance twice.

(defstruct-resource (myresource :named) slot1 ... slotn) This function is the defstruct function for resource structures. The syntax is exactly the same as `defstruct`. This function defines a resource. After this function is called, the user can call the allocate-*myresource* program to get instances. Default initializations should not be specified on the slots, since the user should explictly fill in each slot whenever an instance is allocated.

(allocate-MYRESOURCE) This function is the allocation function for resources. It should be used instead of the standard structure-instance allocation function `make-myresource` to allocate an instance of a resource that has been defined using `defstruct-resource`. It returns the newly allocated structure. The new structure will most probably contain evil garbage in its slots, and should be initialized by the user. This function is automagically defined when a `defstruct-resource` is executed.

(deallocate-MYRESOURCE my-instance) This function is the deallocation function for resources. It throws an instance back into the resource pool. The next time the same kind of resource is allocated, instead of using new data space, the allocation function will return this old instance. Since the instance will contain garbage, it is up to the programmer to clear it out and initialize it properly. It is a mistake to continue using an instance that has been deallocated; the user program should deallocate an instance only after it is sure that the program is finished with that instance.

2.4.6 Output Commands: The pformat Package

The beginning programmer should load `beholder-2` and use `pformat` in all cases where a `format` command would be used in normal Lisp. The syntax is exactly the same.

One of the first things that a user discovers about a parallel computer such as the Sequent is that the output from multiple parallel processors is close to useless. If multiple processors are allowed to write into the same output stream, characters will be badly interleaved and mixed up, and occasionally characters, words, or even whole lines will get lost. Characters are not guaranteed to be printed. Right away, this renders numbers meaningless, because you cannot be sure that when the computer typed `10.5` it did not actually mean `1000.56`, or even `19999 200.5`. It is important to guarantee that all output gets printed, and that the output from a single format statement is printed continguously (without being shuffled with other output). This is accomplished by the `pformat` command. The `pformat` command implements a parallel version of the familiar `format` command; the arguments are exactly the same. A single lwp called the PFORMAT-PROCESS is allocated (by `init-pformat` and dedicated to the task of printing output. When a process calls `pformat`, instead of sending the output directly to the terminal, the output arguments get evaluated, quoted, and then placed in a special system mailbox (`pformat-MB`). The PFORMAT-PROCESS does nothing but read this mailbox and print out the messages in it. If there are no messages, the process blocks until a message comes in. The process runs forever (until it is killed). Since, in practice, an lwp keeps running on the same processor-task until it blocks, this means that the application processes will tend to run for a long time until a couple of these processes block, and then finally the PFORMAT-PROCESS is allowed in and you get a lot of (old) output at the same time.

Since the output uses one lwp, this could tend to take one processor-task away from the other lwps that want to run. However, since the output only runs when there is actually output to be typed, this should not cause too much extra overhead. There does not seem to be any alternative.

In the past, the output would only be typed out when the lwps are running. Thus, if you typed a `pformat` command at the top level, it was stored until `(start-lwps)` was typed again. This was inconvenient. The latest version of `pformat` tests to see whether only one processor is allocated or not. If so, it assumes that the user is testing a program directly at the top level, and prints out the message directly, using `format`. If there is more than one processor allocated, it assumes that a parallel program is running, and sends the message to the pformat mailbox as before. Note that this test takes some time to perform. If the user is certain that only parallel

programs will be run, and that the programs will not be tested at the top level, then ppformat sends the message directly to the pformat mailbox.

In certain pathological circumstances, unprinted output will be saved up until (start-lwps) is called, resulting in leftover output that does not belong to the current program run. For this reason, it is currently recommended that you call (reset-pformat) at the beginning of your program. This function flushes out the old mailbox without disturbing the pformat lwp itself.

Currently (init-pformat) is automatically called when the beholder-2 file is loaded. In addition, the PFORMAT-PROCESS is *not* placed on the list of *lwps* used by kill-all-lwps. Thus, there should be no need for a user to call init-pformat; and the user can use kill-all-lwps as often as possible without having to worry that the output will go away. If, for some reason, two PFORMAT-PROCESSes are allocated, it is conceivable that this could cause a problem, as both of them would try to read from the current pformat-MB mailbox and then try to type at the same time.

pformat should be used for all output done by lwps. If, for some reason, you want the main process to perform output, you may use either format or pformat for this task. Note again that the pformat output will not appear until after the start-lwps command starts up, if more than one processor is allocated. Remember also that the process running the start-lwps command goes away until all the lwps are completely finished, so pformat output provided *after* the start-lwps command in the main-process code will be pushed on the queue and not seen this time.

Current thinking says to use format for :print-functions inside defstructs; it is unclear whether this is correct or not. Sometimes it is necessary to use ',stream instead of stream for the stream in a :print-function.

It is important not to call the system routine (reset-queues) after the beholder-2 package is loaded, as this will break pformat by terminating the pformat process, causing all messages to stack up in the mailbox and not get printed. If (reset-queues) is called, the user must call (init-pformat) afterwards to initialize a new pformat process. This also wipes out the mailbox.

If pformat appears to be broken, it should be possible to dump the contents of the mailbox using the command dump-pformat. This is similar to reset-pformat, except it prints the dead messages out, in order, as it flushes them.

Pformat Basic User Routines

(pformat stream control-string args...) This function is the parallel version of the regular format command. The arguments are exactly the same as format. First, the function tests to see whether only one processor has been allocated or not. If so, it assumes that a top-level program is running, and it prints out the formatted message directly. If not, the function evaluates its arguments, quotes the result, and pushes a print request containing the quoted evaluations onto a system mailbox. Later, a special dedicated lwp executes the request and prints the output. pformat should be used in lwp code in all cases instead of format.

(ppformat stream control-string args...) This function is the direct version of pformat, the parallel version of format. It does not test how many processors

have been allocated, but rather sends the output directly to the pformat mailbox. It is slightly faster than pformat, and should be used when the programmer is certain that only parallel processes will be used and there is no need for running the program at the monitor level. It does not print out its messages until start-lwps has been called. The arguments are exactly the same as format. The ppformat function evaluates its arguments, quotes the result, and pushes a print request containing the quoted evaluations onto a system mailbox. Later, a special dedicated lwp executes the request and prints the output.

(reset-pformat) This function performs a flush of the pformat-MB. Any and all old format messages currently in the mailbox are discarded. The function returns the number of old format messages that were thrown out. The messages are not printed out. See dump-pformat.

Pformat Advanced User Routines and System Routines The normal user should not have to worry about these commands.

(init-pformat) This function is called by sq-system when it is loaded. It allocates the PFORMAT-PROCESS lwp and stores it in system variable pformat-process. It sets pformat-MB to a new mailbox. The pformat process should not be blocked or killed by the user. For this reason, the user should never have to call this function.

It is important not to call the system routine (reset-queues) after the beholder-2 package is loaded, as this will break pformat by terminating the pformat process, causing all messages to stack up in the mailbox and not get printed. If (reset-queues) is called, the user must call (init-pformat) afterwards to initialize a new pformat process. Remember that this also wipes out the mailbox.

(check-pformat) This function returns the state of the pformat process. It should be :RUNNABLE. If it is :TERMINATED, the process must be re-initialized with init-pformat.

(dump-pformat) This function performs a dump of the pformat-MB. Any and all old format messages that are currently in the mailbox are printed out, and then discarded. The function returns the number of old format messages that were dumped out. This routine is useful for seeing dead messages if pformat is broken, or if the lwps are not running at the moment. It should only be used in unusual cases. See reset-pformat.

pformat-MB This system variable stores a mailbox that is used to implement pformat. It should not be examined or modified by the user.

pformat-process This system variable stores the lwp process that is used to implement pformat. It should not be examined or modified by the user.

2.4.7 Blocking and Unblocking Commands

(kill-all-lwps) Kills all the lwps on the *lwps* list.

(block-lwp LWP) The same as suspend. Note that (block) is a system command that means something completely different.

(unblock LWP) The same as resume. Note that (unblock-lwp) is apparently already a system command.

(block-me) A very useful command. Suspends the lwp that executes this statement. When the lwp is unblocked by someone else, execution picks up on the next line.

2.4.8 Creation Commands: Spawn, Build, Fork, and Join

Basic LWP Creation–Spawn and Build One of the major problems of the Sequent is that the primary function `make-lwp` does not evaluate its arguments until the lwp actually *runs*. This means that, if you are lucky, local variables used in the function definition might be bound to something new but vaguely reasonable; if you are unlucky, they might point to something completely random in the middle of the stack. For example:

```
<Initial lwp> (dotimes (i 3)
                (make-lwp (format T "Process variable ~A.~%" i) :run T)) NIL
<Initial lwp> (start-lwps)
Using 1 processor
Process variable 3.
Process variable 3.
Process variable 3.
```

In this case, the variable i is still bound to 3 at the time that the processors start. All three processors use that value.

The predicted and desired behavior, of course, is to have the arguments evaluate at the time that the process is *defined and allocated*, not at the time that it is *started*. This capability is implemented by the `spawn-lwp` and `build-lwp` commands. `spawn-lwp` starts a live process, i.e. one with the :run flag of T; the process will start running as soon as `(start-lwps)` is called. Of course, if `spawn-lwp` is called from a running lwp, the resulting created process starts running immediately (well, as soon as it can get a free processor-task). `build-lwp` starts a blocked process–one that waits until it is unblocked (resumed) by someone in order to run. Remember that `start-lwps` does not unblock blocked processes, it just starts ones that are unblocked already.

The main argument to both `spawn-lwp` and `build-lwp` is a list that looks like a function call. Each of the arguments in the function call is evaluated, quoted, and then packaged into a new make-lwp call. Since the new make-lwp only accepts quoted constants, the expected proper behavior is produced. Note that this means it is no longer possible to define an unnamed process consisting of a `let` or a `progn` enclosing a bunch of statements–since `spawn-lwp` would evaluate each of

the statements at allocation-time, instead of at lwp run-time, this would probably not produce the desired results. It is much better to package the desired program of an lwp into a single routine, and then to use that routine name, together with any required arguments, in a call to `spawn-lwp`.

Both `spawn-lwp` and `build-lwp` come in two flavors. The normal kind considers the first atom in the function-call list (i.e., the function name) to be a literal; this is normally what you want. The second kind, distinguished by `-ev` at the end of the name, considers the first atom in the function-call list to be a variable; this variable is evaluated just like the other arguments, and then the result is used to call the function. For instance, in the following example the two constructed lwps are identical:

```
(spawn-lwp (parse *input-sentence*))

(setq *which-routine* 'parse)
(spawn-lwp-ev (*which-routine* *input-sentence*))
```

Spawning and Building Commands

(spawn-lwp (my-routine args...) &optional (name "An LWP"))

Allocates and creates an lwp that starts out running. The lwp will run the given routine. Evaluates and then quotes its arguments. Does not evaluate the routine specification; this should be a literal. Returns the resulting lwp.

```
(spawn-lwp (parse *input-sentence*))
```

(build-lwp (my-routine args...) &optional (name "An LWP"))

Allocates and creates an lwp that starts out blocked. The lwp will run the given routine. Evaluates and then quotes its arguments. Does not evaluate the routine specification; this should be a literal. Returns the resulting lwp.

```
(build-lwp (parse *input-sentence*))
```

(spawn-lwp-ev (my-routine-expr args...) &optional (name "An LWP"))

Allocates and creates an lwp that starts out running. The lwp will run the given routine. Evaluates and then quotes its arguments. Evaluates the routine specification as well. Returns the resulting lwp.

```
(setq *which-routine* 'parse)
(spawn-lwp-ev (*which-routine* *input-sentence*))
```

(build-lwp-ev (my-routine-expr args...) &optional (name "An LWP"))

Allocates and creates an lwp that starts out blocked. The lwp will run the given routine. Evaluates and then quotes its arguments. Evaluates the routine specification as well.

```
(setq *which-routine* 'parse)
(build-lwp-ev (*which-routine* *input-sentence*))
```

Fork and Join A common medium-low-level problem is to set up many parallel processes that fork off from the stream of execution, and have each process take care of its own business. In addition, it also may be desired to join these processes together, so that when all of them are finished executing a new process is called to continue the stream of execution. This model of parallel control is called the "fork and join" model. Various processes are "forked" off, and "join" back together again later when they are finished.

The fork-and-join model is fully supported by the routines `make-fork`, `fork-lwp`, and `join-and-when-finished-do`, along with their variants. First, the user should use `make-fork` to create a fork structure, which is used for bookkeeping by the system. The fork (structure) is important, because several forked sets of parallel processes could be going on at the same time, and it is important to tell *which* fork a new process belongs to. The user should store this structure in a handy fork variable, and use this variable as a bookkeeping device. The user never needs to examine this structure, merely to pass it around for identification purposes.

Next, the user should create a number of parallel processes, using `fork-lwp`, which takes the previously-mentioned fork variable as an argument. The `fork-lwp` call replaces the normal call to `spawn-lwp`. Inside the process function definition, as the last line to get executed, the process should call `join-and-when-finished-do` with the argument of a block of routines to be executed after the join occurs.

It is important to remember that the parent process continues executing after it forks off a child-fork process. Indeed, there is no need for there to be a single parent; many parent processes can call `fork-lwp`, and as long as they all use the same fork variable, they can all contribute new child processes to the fork.

Typically, a parent will set up a number of forked processes, and then die; or loop, go to sleep and wait for the next processing iteration (e.g. the next utterance). The forked children execute, and then perish; all except one, which gets to be the lwp that executes the join routine and its subsequent processing. However, if the desired effect is for the parent to fork off a series of routines and then wait until they are all done, and then to have the parent continue as the join, a slightly different programming style is necessary. The parent should pass itself, `*this-lwp*`, as an argument `parent-lwp` to each of the child forked processes. The parent should call `(block-me)` after it has finished forking off all of the children. Each child should then have the expression `(join-and-when-finished-do (unblock parent-lwp))` as its last line. The parent will fork the children and then block; the children will execute; and the last child will unblock the parent. In this manner, an inline fork-and-join is built.

There have been questions as to whether the fork variable should be stored in a local variable inside the parent routine, and then passed down as an argument to the child forks; or whether the fork variable should be a global variable (perhaps in an indexed array) and then have the child call this global by name. Either way will work. It is probably better to design the child to accept a fork variable as an argument, rather than hardwiring the name in, in case more than one parallel parent desires to start up different (simultaneous) forks using the same child-process function. There is no need to worry about whether the fork variable is local or global in the parent, as the fork variable stores the fork *structure*, which is allocated out of main memory, not the parent's lwp memory. Even if the structure is stored in a

local variable in the parent, and the parent process dies (thereby rendering its local variable storing the structure invalid), the forked child processes will still continue on correctly—because they have been given pointers to the structure, not copies, and not the variable itself. About the only way to mess up is if the user attempts to pass the fork variable in by declaring a local variable name "special" in the parent and forked-child processes, and then hoping that the child processes will be able to reference the special variable by name. The parent's local variable is taken from its processing stack. If the parent process terminates or even becomes blocked, its stack is no longer available, and the special references will fail.

It is *important* to be careful about *pathological race conditions* when using forks. If `fork-lwp` is used to build lwps one by one in an indiscriminant fashion, and the resulting processes are small and fast, it could happen that all of the existing processes finish just as the next lwp in the fork is about to get built. If all the processes in a fork finish, the fork's `join` routine is executed, and that turn or round of the fork is basically finished. Putting a late forked process or two onto a fork that has already fired its join routine is the same as starting up another turn for the fork–when those few processes finish, the join routine will be called again. This is normally not what is desired by the programmer. The solution is to normally use `build-fork-lwp` to create blocked fork processes, and to collect these and push them onto a list as they are allocated. Then, use `start-processes` to unblock them all at the same time. Note that in theory, since `start-processes` uses a loop to unblock the processes one at a time, it could be possible for a started process to finish before all of the processes are unblocked. However, in practice, since an lwp is not swapped out of a processor-task until it blocks, this should not cause any problems.

All of a fork's `join-and-when-finished-do` routines should execute the same code, since it is unknown which one of them will actually be used to complete the join and start the next process.

The fork-and-join model of parallel processing has to some extent been replaced by the agenda-and-worker model, which allows higher-level control of the amount of resources allocated to a subsystem. Although the fork-and-join routines are probably alright for experimental use, it would probably be better to develop serious code using the agenda-and-worker routines instead, to allow integration with other subsystems at a later date.

In the current version, the variable `*lwps*` is maintained by the user. If the user is interested in having `(kill-all-lwps)` work, it is now the user's responsibility to push each created process onto the `*lwps*` list.

Forking and Joining Commands

(make-fork) Creates and returns a fork record-keeping structure. Store this in a variable and use it to control the following functions.

(fork-lwp fork-var (my-routine args...) &optional (name "A Forked Process"))
Spawns a **running** process to execute the given routine. The name of the routine is a literal that is not evaluated. Increments the active process count in the given fork. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq i 1)
(fork-lwp parse-fork  (parser *sentence1-A*)
(string-append "Parser-" (my-string i))  )
(++ i)
(fork-lwp parse-fork  (parser *sentence1-B*)
      (string-append "Parser-" (my-string i))  )
```

(build-fork-lwp fork-var (my-routine args...) &optional (name "A Forked Process")) Builds a **blocked** process to execute the given routine. The name of the routine is a literal that is not evaluated. Increments the active process count in the given fork. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq parse-lwps ())
(setq i 1)
(push
  (build-fork-lwp parse-fork  (parser *sentence1-A*)
(string-append "Parser-" (my-string i))  )
  parse-lwps)
(++ i)
(push
  (build-fork-lwp parse-fork  (parser *sentence1-B*)
(string-append "Parser-" (my-string i))  )
  parse-lwps)

...

(start-processes parse-lwps)
```

(fork-lwp-ev fork-var (my-routine-expr args...) &optional (name "A Forked Process")) Spawns a **running** process to execute the given routine. The name of the routine is contained in an expression that is evaluated. Increments the active process count in the given fork. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq which-func 'parser)
(setq i 1)
(fork-lwp-ev parse-fork  (which-func *sentence1*)
(string-append "Parser-" (my-string i))  )
(++ i)
(fork-lwp-ev parse-fork  (which-func *sentence1*)
(string-append "Parser-" (my-string i))  )
```

(build-fork-lwp-ev fork-var (my-routine-expr args...) &optional (name "A Forked Process")) Builds a **blocked** lwp that is part of a parallel fork.

The name of the routine is contained in an expression that is evaluated. Grabs and increments the active process count in the given fork, pushes the lwp on the fork list. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq which-func 'parser)
(setq i 1)
(push
  (build-fork-lwp-ev parse-fork  (which-func *sentence1-A*)
(string-append "Parser-" (my-string i))  )
  parse-lwps)
(++ i)
(push
  (build-fork-lwp-ev parse-fork  (which-func *sentence1-B*)
(string-append "Parser-" (my-string i))  )
  parse-lwps)

...

(start-processes parse-lwps)
```

(join-and-when-finished-do old-fork-var body...) Decrements the number of active processes in the fork by 1. Executes the code contained in **body** when the last process finishes (calls this function). This routine should come at the end of the body of the lwp procedure.

(join old-fork-var) This is a low-level routine that could be used by users attempting to build custom join routines. It decrements the active processes count and returns the new value. Most users will want to use `join-and-when-finished-do` instead.

(end-of-fork old-fork-var) This is a low-level routine that could be used by users attempting to build custom join routines, or to seal off forks that have no joins. It is the same as "join". It decrements the active processes count and returns the new value.

(start-processes list-or-sequence-of-processes) Starts a list of blocked processes: sends a resume signal to each process on the list.

```
(start-processes lwp1 lwp2 lwp3)

(setq parser-lwps (list lwp1 lwp2 lwp3))   ;could be '',lwp1 .
(start-processes parser-lwps)
```

2.4.9 Workers and Agenda Commands

The Worker and Agenda model is a very useful model for designing parallel control structures. An agenda is a First In, First Out (FIFO) queue of tasks to be performed.

Each agenda has a set of worker processes. When a worker process finishes its current task, it goes to the agenda and requests another task. Another, unspecified part of the system keeps the agenda full by entering tasks on it. If the agenda is empty when a worker requests a task, then the worker blocks, goes to sleep, and waits for the agenda to fill up again.

A simple Worker and Agenda system can be implemented using mailboxes or Fast Mailboxes.

This section offers a more complex version, that is designed for stronger control. Each agenda has a certain number of workers, but only some of these are *active* at one time–the rest of them are *leashed*. Commands exist for creating, leashing, and unleashing workers. The resulting framework allows control of the amount of resources devoted to executing different parts of a system, represented by different agendas.

The Worker and Agenda package is still under research. A more advanced version is being prepared to replace this one.

For a further discussion of the theory behind this package, see Section 6.

(make-agenda) Returns an agenda.

(agenda-empty-p agenda) Tests to see whether an agenda is empty or not.

(wait-for-empty-agenda agenda) Waits until an agenda is empty. Detects this by waiting for the agenda to send an "empty" message to itself, which happens when the agenda notices the number of running active workers is 0.

(push-agenda item agenda) Pushes a given item onto the given agenda.

(pushnew-agenda item agenda) Pushes a new given item onto the given agenda, if it wasn't in the agenda waiting to be processed already.

(*my-worker-function* **item**) User-defined worker functions for a particular agenda must take one and only one argument, which gets bound to the item that is popped off the agenda–the same as the item that was originally pushed onto the agenda.

(make-worker my-worker-function agenda) Allocates a worker for the given agenda. My-worker-function must be an unquoted literal.

(make-N-workers N my-worker-function agenda) Allocates N workers for the given agenda. N should be a positive integer. My-worker-function must be an unquoted literal.

(unleash-all-workers agenda) Activates all of an agenda's workers. Normally call this right after allocating all workers. Active workers stay active whether they are executing an agenda item or waiting for the agenda to fill.

(unleash-N-workers N agenda) Activate some of an agenda's workers.

(leash-all-workers agenda) Deactivate all of an agenda's workers.

(leash-N-workers N agenda) Deactivate some of an agenda's workers.

2.5 Significant Variables

lwps This variable holds all the lwps that the system knows about. This includes live, blocked, and dead lwps. This variable was previously used by `fork`, `build-fork`, `init-pformat`, etc., but is now completely maintained *by the user*. It is mainly useful because `kill-all-lwps` uses it as input. When you create a process by yourself (e.g., by using `fork`), you should push the process onto `*lwps*`, so that you can kill it later if you need to.

Example:

```
(push (spawn (my-process args))  *lwps*)
```

master-process This variable is used by `Start-Master` to determine which process to start. It is used for looping process chains. It should store a process, if you want to take advantage of the looping feature. Set this variable to a process that sets up your system, and then call `Start-Master` as the last thing before the last join dies.

2.6 Flag Variables

recording-statistics This flag is used by the worker/agenda package. If it is non-NIL, the agenda package gathers statistics on how long each worker takes to accomplish its job. These can be used by the BEHOLDER package to determine expected process durations.

2.7 System (Non-user) Variables

pformat-MB This system variable stores a mailbox that is used to implement `pformat`. It should not be examined or modified by the user.

pformat-process This system variable stores the lwp process that is used to implement `pformat`. It should not be examined or modified by the user.

rand-lock This system variable stores a lock that is used to implement `rand`. It should not be examined or modified by the user.

3 KNOWN FEATURES OF THE SEQUENT AND ALLEGRO PARALLEL COMMON LISP

3.1 "Make-Lwp" Evaluates its Routine Arguments at Start-Time

One of the major problems of the Sequent is that the primary function `make-lwp` does not evaluate its arguments until the lwp actually *runs*. The lexical string containing the lwp process definition is apparently quoted and pushed on a stack; later on, when `start-lwps` is called, the definition is recovered and evaluated to determine what constants and variable values the process should use. This means that, if you are lucky, local variables used in the function definition might be bound to something new but vaguely reasonable; if you are unlucky, they might point to something completely random in the middle of the stack. For example:

```
[1] <Initial lwp> (dotimes (i 3)
                    (make-lwp (format T "Process variable ~A.~%" i) :run T)) NIL
[1] <Initial lwp> (start-lwps)
Using 1 processor
Process variable 3.
Process variable 3.
Process variable 3.
```

In this case, the variable i is still bound to 3 at the time that the processors start. All three processors use that value, instead of 0, 1, and 2.

The predicted and desired behavior, of course, is to have the arguments evaluate at the time that the process is *defined and allocated*, not at the time that it is *started*. This capability is implemented by the `spawn-lwp` and `build-lwp` commands, discussed in Section 2.4.8.

The problem is made more interesting by the fact that the :run and the :name arguments inconsistently evaluate at allocation-time, not start-time.

3.2 Characters Interleaved on Multiple Output Processes

Using the CLiP system, when multiple processes print information out to the user's terminal by using `format`, not only the words but also the characters are interleaved in the output stream. If multiple processes are printing out at the same time, as is normally the case, it is almost impossible to read the output. This problem has been fixed by the `pformat` command.

3.3 Characters Lost on Multiple Output Processes

When multiple processes print characters to the terminal, not only does the order of the characters get jumbled, but sometimes output gets lost. Whole characters, words, or lines of output can get lost in the output stream and never show up.

Just think of the differences between reading a printed timing result of 1234.56 seconds, 14.56 seconds, or 123456 seconds, and it is easy to understand why this is unacceptable. This is another important reason for using `pformat`.

3.4 "Sleep" Truncates to an Integer

Because Franz-Lisp is implemented on top of Unix, the `(sleep nsecs)` program quietly converts `nsecs` from a (possible) floating-point number into an integer, in order to call the Unix `sleep` function. It does this by truncating down to the next lowest integer. Thus, a call to `(sleep 0.99)`, instead of sleeping for about 1 second, does not sleep at all–it sleeps for 0 seconds. This renders the sleep function basically useless for subsecond timing, required for simulating mock processes and testing race conditions. Users requiring this functionality should use the new `(wait nsecs)` function defined in this manual. Unfortunately, the `wait` function can only be accurate within a hundreth of a second or two, whereas basic operations take on the order of a microsecond to perform.

3.5 Compiling a Function Once Does Not Deinstall a Macro with the Same Name

Sometimes known as the "Undead Macro" or the "Macro From Hell" problem, this feature is particularly nasty. If a macro is changed into a function with the same name, the system will still use the macro definition instead of the function definition for the *source code*, apparently until the time after the file containing the new function has been compiled and loaded. Thus, the next compilation will reuse the old macro definition. The symptoms of the problem are consistent with the following behavior: The function definitions are saved as the file is compiled, and installed on their symbols after the end of the file is reached. However, macro definitions are installed on their symbols as they are defined. Any macro that was previously defined remains in effect until it is redefined in the source code by another defmacro, or until after the file containing a new function definition has been compiled and loaded. A function does not redefine a macro until after it has been loaded. Thus, if a function redefines a macro at the beginning of a file, the redefinition will not take effect until the file has been completely compiled and loaded–and all the calls to the function will have been compiled as calls to the old macro. The way to fix this feature seems to be to both compile and load the program, twice. Note that the source code must be a different version in order for `:cl` to consider recompiling it–modify one character up and back and then save the new result to get a new version.

As an example, say that you are debugging a difficult macro (e.g. `my-format`) that has problems with the level of evaluation of its arguments–it comes up with the wrong answer. Of course, the macro is defined before the test routines that use it in the source code. Now say that you decide to turn the macro into a function-building function, because the level of evaluation should be different. You change the source appropriately, compile and load the program. You run the compiled test routine, but you get the same mistakes. The compiler secretly reuses the old definition of the macro, even if the function definition is before the test code calling it in the

source. However, if you test `my-format` by hand directly at the interactor level, it uses the *new* definition, thus adding to the confusion. If you are lucky, you change something else in the code, recompile, and the old macro definition has gone away so you get the new function definition this time. If you're not lucky, you spend half a day chasing down a bug that's not there.

Note that no warning messages are given when a function redefines a macro in the environment. Also, even if a function redefines a macro in the same file, no warning message is presented. Also, if a macro redefines a macro in the same file, no warning message is presented.

As is typical, normal macros get expanded inside routines at compile time. Thus it is impossible to redefine a macro call inside compiled code by redefining the macro from one defmacro even to another defmacro; since the code has already been expanded and compiled, there is no way to touch it. The routine calling the macro must be recompiled.

3.6 Machine Wedge on Too Many LWPs

If too many LWPs are *allocated* by a user, the entire machine wedges. Not only does that user's terminal freeze up, but no other users can log on, and basically all commands (including `ls`, etc.) will not run. The symptom is a message similar to `Insufficient memory to run command: ls`. It does not matter whether the processes are running or blocked; this problem will typically occur even before `(start-lwps)` is called in the code.

This problem is caused by the entire computer running out of disk space on the user's partition. The system (actually the Sequent Symmetry Parallel Programming Library) uses virtual memory to represent each LWP. Each LWP consumes a default of about 75K of memory, for Lisp stack space and a few other details. The virtual memory is represented by an actual, temporary swap file on disk that is stored in the user's current working directory, ".". The name of this file is "`shared memory PIDNUM`"; however, it is *unlinked*, so `ls` will not show it. This file dynamically expands as the user requests allocation for more LWPs; it gets deleted when the user `quit`s a CLiP session, specifically until the CLiP process responsible for creating it dies. When the user requests the creation of too many light-weight processes (say, more than 100), the system attempts to grow the temporary file past the limits of the disk space and crashes. However, when it crashes, the temporary file is not removed. At this point, if any other user attempts to open a single LWP, his system will crash too. Since most shell commands (even simple shell commands such as `ls`) require virtual memory space to run, even the simple shell commands will not work for other users. The entire machine is blocked.

The solution is to log in as the root (i.e., system) programmer on the system console, and to kill the user's processes. This should free up enough memory to get the system back up and running again.

3.7 Incremental Error Compilation

The current version of the CLiP compiler is extremely inconvenient to use, in that it compiles until the first error is found in the source file, prints a cryptic error

message, and then stops. There is no error recovery. This means that if there are 30 errors in your source code, you will have to fix an error and recompile the code 30 times before the code is bug-free, as opposed to getting a list of 30 errors, fixing them, and recompiling the code only once.

One solution is to develop and compile the code on the Symbolics machine, using the `sq-compat` file discussed in Section 5, and make sure as much as possible that there are no compilation errors in the code by using the Symbolics' facilities. After that, download the debugged code to the Sequent.

Another very good solution is to use the ILISP editor package, which does allow incremental compilation.

3.8 No Checks for Unbound Function Names

The CLiP compiler does not check for unbound function names when it compiles your code; no error messages are generated. This means that if you have any typographical errors in your code, or any functions that you were "going to define later" but forgot to, you get to discover them at run-time (if you are lucky). The program will bomb because it attempts to call an undefined function. Of course, if the error is in a rare case and the program does not call that case very often, the program might run correctly for several times and you might not discover your error until much later.

This feature is particularly inconvenient. It is scheduled to go away in the next release of CLiP.

3.9 Structures Do Not Evaluate to Themselves

In most Lisp compilers, an atom that is an instance of a structure is bound to a value that consists of itself. That is, when a structure is evaluated, the structure is returned. This means that the expressions `'#<instance>` and `#<instance>` can both be used as legal input to a function; both will evaluate to the instance itself.

However, in the current version of CLiP (6.0.1), an atom that is an instance of a structure is not bound to any value. That is, when a structure is evaluated, the program crashes with an unbound variable reference. This means that the expression `#<instance>` cannot be used as legal input to a function.

This problem is especially pernicious when implementing general-purpose deeply nested macros.

This feature may go away in future releases of CLiP.

Note that numbers do in fact evaluate to themselves; this problem appears to be restricted to things implemented using structures only.

3.10 Compiler Error Messages

The current compiler prints out a running list of the routines that it has ALREADY compiled, but it uses `; Compiling ROUTINE-A` instead of `; Compiled ROUTINE-A`. If there is a syntactic error in the *following* macro, structure, or routine (say, `ROUTINE-B`), the compiler prints out the following sequence:

```
; Compiling ROUTINE-A
Error: <some error message>
```

This normally means that the error is in routine **B** *following* routine A, not that the error is in routine A.

3.11 Flavor Instances Do Not Receive Newly Defined Methods

If a programmer defines a flavor type (using the old flavor system or the new-flavors system built on top of it) and then starts creating instances of the flavor, and *then* defines some methods THIS IS UNCLEAR.

4 THINGS A PROGRAMMER SHOULD KNOW ABOUT THE SEQUENT AND ALLEGRO

4.1 "Real" processors, Timesharing, and User Interference

The CLip manual talks about virtual LWPs being allocated, and then assigned to actual processors. A programmer must allocate actual processors in order to be allowed to run. A programmer is only allowed by the system to allocate up to 11 processors, the current number of hardware processor boards inside the ATR Sequent. The manual also states that processors communicate via global memory.

There are two logical consequences of the material presented in the manual. First, it would seem that if one user logs on and allocates, say, 8 or 11 processors, then that would exclude those processors' usage from other users. Perhaps a third or a second user could not log on. Second, it would seem that one user's program could communicate with another user's program by storing appropriate values in common global variables.

These consequences are not the case. Any number of users can log in to the Sequent computer at the same time, and each of them can allocate 11 processors. Each user program runs in a separate space; programs running on two different terminals cannot communicate with each other through shared variables.

The confusion is resolved by examining the details of the implementation more closely. The CLiP system is implemented on top of a (multi-processor) version of the UNIX timesharing system. What the CLiP manual calls a "processor" is not a hardware processor at all, it is actually a UNIX software process image. The operating system forks a UNIX process for each "processor" that is allocated, and then uses that process for running an active LWP. The operating system takes care of the low-level details of the actual hardware processor allocation; this is transparent to (and apparently uncontrollable by) the user. The limit of 11 allocated "processors" imposed by the system is arbitrarily put in to insure that no one user bogs down the system

4.2 Memory Allocation and LWPs

Each LWP consumes a default of about 75K of virtual memory. Most of this (64K) is a stack for run-time Lisp. The code and the data used by the LWPs are *shared* and are charged *once* to the initial process heap–the system does *not* make a separate copy of the executable code for each LWP. Note that this affects self-modifying code.

This means that the overhead for allocating another LWP is slight–basically, only space is allocated, and very little state initialization is performed. The LWP takes a copy of the executable code defining the process (one line), allocates a separate stack plus local variables, and then uses a common executable image of the system routines to actually run the code. So each routine shares code but does not share data. Both C and CLiP work in this manner.

It is possible to allocate an LWP with a smaller stack size, using the `:stacksize 65532` keyed argument to `make-lwp`. (Note that the default is 64K). However, any LWPs with smaller stacks will not get garbage-collected and recycled on termination in the current CLiP version. If your program works with immortal processes that never terminate, this will only be a problem if you try and run your program twice in one CLiP session. This feature is expected to change in the next version of CLiP. The system has been tested with stacks as small as 1024. It is probably smart to make the stack size be a multiple of a large power of two.

Note that the system garbage-collects the memory of dead LWPs. Thus, the allocation of every LWP with a non-standard size is guaranteed to require a low-level system call to the disk to grow the memory space, whereas the allocation of a standard-size LWP should reuse the space of an old dead LWP without requiring a system call, if any dead standard LWPs exist. This may cost more time for allocation. On the other hand, if less memory is required for the LWP's stack, theoretically less time should be taken in allocating the memory for a single LWP. It is unclear what kind of trade-off exists between these two factors.

The results indicate that the number of LWPs that can be allocated depends on the size of the LWPs and the amount of space left on disk. See Section 3.6 for further discussion.

Apparently, the system keeps a list of the inactive processes. When a LWP is activated, it is removed from this list. Note that this operation is O(n) [linear] in computational time, n being the number of inactive LWPs. This seems to limit the total number of LWPs that can be used effectively, to around 100-500. These results have not yet been confirmed.

4.3 Arrays and Lists

CLiP arrays are printed out as **lists** by the CLiP system. This may lead some people to think that the CLiP arrays are implemented as lists, and that array indexing is implemented as the horribly inefficient `nth` function. This is incorrect. CLiP arrays are implemented as actual arrays in the normal manner; access consists of adding the index to the base pointer to compute an indirect pointer, and then following the indirect pointer twice, as usual. The printout of arrays as lists is an artifact, and should be ignored. It is speculated that this printing method comes from the fact that both arrays and lists are sequences, although this does not justify the confusion.

4.4 Flavors and Object Oriented Packages

Three object-oriented programming packages are available on the Sequent. The first is known as Flavors (standard Flavors, or Old Flavors), and was written by Franz. The second is called New-Flavors, and was written by John Myers. The third is called PCL; the author is unknown. A fourth package, called CLOS, is currently not yet available on the Sequent.

4.4.1 Overview and Discussion

Flavors Flavors is a mature product that has been well-implemented. It supports the old style of syntax used in the Symbolics Version 6 operating system, which was popular around 1984. Franz reimplemented the Flavors system, they are not using Symbolics code. The old Flavors syntax uses `send` messages to set and access variables, and is slightly clumsy. Although old Flavors code will compile and run on a Symbolics, the compiler complains about the syntax used by each `defmethod`, which is inconvenient.

New-Flavors "New-Flavors" is the name of a small but clean hack designed to sit on top of the Sequent Flavors system and provide compatability with the Symbolics New Flavors syntax, popular around 1988. New Flavors-style calls are translated into old Flavors calls by using macros. Thus, both New-Flavors calls and old Flavors calls can coexist in the same piece of code. New-Flavors code should compile directly on a Symbolics machine. Conversely, using the New-Flavors package, New Flavors code from the Symbolics can be run on the Sequent. However, the system has not yet been tested extensively.

PCL PCL stands for Portable Common LOOPS (the Lisp Object-Oriented Programming System). It is a dirty system that leaves many needed functions unimplemented in the version we have. It is an early (1988) precursor to CLOS and uses the CLOS syntax (on those functions that work). Although PCL has been ported to a number of machines, it really should be considered to be an alpha version of CLOS and should be ignored in favor of CLOS.

CLOS CLOS stands for the Common Lisp Object System. It is a new language (1989) that is an attempt by many vendors to create an object-oriented system that sets a standard and can be used on any machine running Common Lisp. Franz has not yet come out with a version of CLOS for CLiP. CLOS is slightly more powerful than New Flavors, but apparently requires much more defining to be done by hand. "classes" are used instead of "flavors". CLOS code compiles and runs on Genera 8 Symbolics.

4.4.2 Problems with PCL

In the PCL flavors (classes) package, the printname of object instances is represented as `#<Standard-Instance ###>`. The object prints correctly as `#<My-Class ###>` in Flavors and New-Flavors.

(type-of my-instance) always returns `PCL::IWMC-CLASS`, no matter what the class, for PCL. (type-of my-instance) correctly returns the lowest flavor for Flavors and New-Flavors.

In PCL, (typep my-instance my-class) returns a list of the class types in the hierarchy from my-class up to T. If the instance is not part of the class, it returns NIL. This is more useful than the simple T/NIL returned by the Flavors and New-Flavors packages.

`subtypep` always returns `NIL ^ NIL` for Flavors, New-Flavors and PCL.

The `(defgeneric)` and `(slot-boundp)` functions defined in CLOS are not supported in PCL.

In PCL, `(describe my-instance)` does not report the slot names nor values, and is basically useless. It works correctly in Flavors and New-Flavors.

Sometimes it is desired that a new description be created for a class of objects. In the PCL system, it is impossible to modify the function `(describe)` with a defmethod, as is described in the CLOS manual. Flavors already completely supports both forms of syntax: `(describe my-instance)` and `(send my-instance :describe)` both print a description; it is also possible to create a custom `(defmethod describe)` for a flavor. New-Flavors also completely supports `describe` by using the old Flavors routines, and by making `describe` be a special case that does not use the New-Flavors compilation procedure (otherwise `describe` would get rebound, which is not desired).

PCL does not support (print-object my-instance) nor (print-self my-instance). Obviously `(send my-instance :print-self)` is also unsupported (the syntax is incorrect). Both Symbolics (Genera 7,8) and the Sequent Flavors support `(send my-instance :print-self stream list-depth slashification)`. List-depth should be around 3-8. Slashification should be T or NIL. Strangely, none of these arguments are optional; the system will break if one is left out. The function `(print-self my-instance &optional (stream T) (list-depth 4) (slashification T))`, using optional arguments, is not supported by Sequent Flavors but is supported by New-Flavors. The Symbolics also supports the `print-self` function (besides the message), but the arguments are required, not optional.

The `:print-self` method is used to output the standard print-representation of the object and can be modified on the Symbolics, under Flavors, or New-Flavors by writing a new method that accepts the same arguments. PCL apparently does not support this capability.

5 THE SYMBOLICS SEQUENT-COMPATABILITY FILE

There is a need to develop Allegro CLiP code on the Symbolics (as discussed in Section 3.7). However, if a CLiP file is compiled on a normal Symbolics, there will be many error messages, as the CLiP parallel system primitives are not defined on the Symbolics. These many unnecessary error messages will mask the few actual error messages that the undebugged program will generate.

The solution is to load a Symbolics Sequent-Compatability file into your Symbolics, that defines each of the CLiP parallel system primitives. Then, the Symbolics compiler will not complain about unbound functions, and only the actual syntactic mistakes in your program will be flagged by the compiler.

This file is currently found in

```
LM01:>myers>sq-compat.lisp
```

There exist both `.bin` and `.ibin` versions. The appropriate version should be loaded into your Symbolics when you first boot the Symbolics.

At present, each of the CLiP system functions has a trivial or null definition. The file is designed for syntactic compilation checking only, and not for run-time simulation. Indeed, it would be difficult to build a Sequent (parallel) simulator to run on the Symbolics (time-sharing) machine. Thus, the Compatability file should only be used for checking the compilation of a Sequent file, and not its run-time characteristics.

6 THE BEHOLDER AGENDA MECHANISM AND THEORY

The BEHOLDER agenda mechanism provides a simple but powerful approach towards allowing a *single subsystem* (e.g. the Parsing subsystem, the Transfer subsystem, or the Generation Subsystem) to be implemented in a parallel fashion. The concept of an *agenda* plus *worker processes* is quite simple. An agenda is a FIFO stack of medium-small processing jobs, called *task descriptions.* A set of "workers" is assigned to an agenda. Each worker uses a different processor. When a worker is free, it pops the next task description off of the top of the agenda, and executes that task. After a worker is finished, it goes back to the agenda and gets another task. Since there are many workers pulling tasks off the agenda and running tasks at the same time, the subsystem's processing is executed in parallel. The whole process continues until the agenda is empty.

When an agenda is created, it can have a number of workers assigned to it. However, it is not necessary that all of the workers be used at any one particular time. Some of the workers can be *active*, while the rest are *inactive.* An active worker watches the agenda and attempts to get tasks to execute. An active worker requires a processor, and thus consumes processing resources. An inactive worker does not take any action; it is implemented with a blocked "lwp" and thus does not use any processing resources. However, just because a worker is active, it does not mean that the worker is executing a task. The worker could be contesting to grab the agenda, or the agenda could be empty.[3] But, in general, the number of active workers determines the number of tasks that the subsystem executes in parallel. *The amount of processing resources allocated to a subsystem can be controlled by varying the number of agenda workers that are active. The number of workers controls how much effort is spent on processing a program, and indirectly how fast that program gets processed.* In an ideal world, the number of active workers would be directly proportional to the speed of processing–e.g., with four workers, the program would execute four times as quickly. However, in actuality, a large number of workers usually tend to get in each other's way when locking data and contesting for the agenda, and program speedup tends to level off sharply after a certain number of workers is activated. The optimal number of active workers depends on the application and must be tuned by benchmarking the program with many different active-worker counts.

The amount of computational resources allocated to a subsystem becomes important to control when many subsystems are integrated together into a complete system. Controlling the levels of computation among multiple subsystems working together is the main job of the BEHOLDER scheduler, which will be discussed in a later report.

[3] In the current implementation, a worker goes to `sleep` and lets go of its processor when the agenda is empty. The worker is still active, but it is not consuming any processor resources. This could change in the future.

6.1 Agenda Queuing Theory

We first define a *simple agenda* as consisting of a FIFO queue with a lock on it. This is the same as a mailbox. We assume that the lock is only on the receive-message (allocate-task) side, and ignore the sending side at first. The lock is necessary so that two workers do not grab the same task-message. We first assume that the queue has previously been filled up with a large number of tasks.

It takes a finite amount of message time M for a worker-process to grab the lock, pop the top message off the FIFO queue, and release the lock. For a mailbox, this time is roughly 100 microseconds[4].

It is important to note that the physical *size* of the message has nothing to do with how fast the message can be read from the agenda. Lisp does not copy data, but rather passes a pointer to the data. A pointer to a long list of data is just the same as a pointer to a brief list of data. Once the task message has been received, it may take longer to *interpret* the data, but this time is classed under the next parameter J.

It takes a non-zero amount of job time J for a worker-process to execute the given job task. The length of the job-task duration may vary, but assume that it is constant. The duration of the task will of course depend on the complexity of the task. A simple one-instruction task, such as a single `setq`, takes about 6 microseconds to execute. A complex program may take many hours to execute. Most worker processes will be somewhere in between.

Assume that there are N parallel workers assigned to the agenda. Each worker grabs the lock when it is free, pops the top message, releases the lock, (taking M microseconds), executes the indicated job (taking J microseconds), and waits for the lock to be free. Assume that, once the lock has been freed, it is instantaneously available for someone else to grab it (this is not a bad assumption, because if this time is finite, it can simply be added on to the release time).

Given these assumptions, the *theoretical maximum speed* of the system is

$$\frac{1,000,000}{M} \text{ jobs/second} \tag{1}$$

or about 10,000 jobs/second using normal mailboxes. Note that this equation is independent of both N and J. Thus, the maximum efficiency of the system *cannot* be increased by adding more processors, since the single agenda is acting as a bottleneck. Each processor has to wait for the agenda, so the maximum speed is not affected by the number of processors. Note that this is the *total* number of jobs processed by the entire parallel worker/agenda system, i.e. the total effort.

This limit assumes that there are no gaps between processor requests, i.e. the agenda is not kept waiting for a processor to finish. Under what conditions does this happen? The agenda is not kept waiting if a processor's job's duration takes less than or equal to the amount of time required to grab a message times the number

[4]This is only the time that the mailbox spends locked. The actual duration time that a single call to read a mailbox takes is closer to 130 microseconds, because of the need to clean up, etc. Only the time that the mailbox spends being locked is significant, and the other 30 microseconds should be added onto J. This second-order effect changes the rough calculations presented here slightly, by degrading the theoretical efficiency due to J having a constant added to it.

of other processors in the system. I.e., the speed limit is in effect if

$$J \leq M * (N - 1) \tag{2}$$

If N is equal to 11 processors, then the speed limit is in effect if the time to execute the job J is less than about 1,000 microseconds (roughly 50 lines of code), using a normal mailbox. If N is equal to 1 (using a parallel setup), the speed limit is always in effect, and even if J is 0 the system is still limited by the speed of the mailbox. With N=15, our current maximum configuration, the speed limit takes effect at 1,400 microseconds (roughly 70 lines of code).

Let's turn this equation around. Given a particular job duration J, what is the maximum number of processors that can be used efficiently? The speed limit is in effect when

$$N \geq \frac{J}{M} + 1 \tag{3}$$

so if an $=$ is used, the maximum effective number can be determined. If the task duration J is less than about 100 microseconds, then two processors is already too many. If the task duration takes up to about 500 microseconds, then up to 5 processors are effective, but more are wasted. So the amount of processors required to be maximally efficient depends on the duration of the task.

What happens if the processors are operating under the speed limit? In this case, there is a neglible delay when the program first starts up, and the $(N - 1)$ processors have to wait for their first assignment. However, after that the processors are out of phase (assuming an exact, constant job-time duration J, and that the workers do nothing but run jobs) and never have to wait for each other again. In this case, the steady-state speed of the system is

$$\frac{1,000,000 * N}{(M + J)} \text{ jobs/second} \tag{4}$$

which is linear in N. So as long as the system has not hit the speed limit, increasing N increases the speed of the system.

This theoretical analysis predicts that the speed curve of the system, measured in jobs per second versus number of worker processes, will consist of two lines with a sharp corner in the middle. The first line will start around the origin and have a slope of $\frac{1,000,000}{(M+J)}$. The second line will be flat and will occur at $\frac{1,000,000}{M}$ jobs per second. The corner will occur at $N = \frac{J}{M} + 1$ processors.

This prediction agrees closely with the shape of curves measured. Of course, since this is a theoretically optimal prediction that assumes J is constant, it is only accurate to a first-order approximation; real times will be more inefficient. In particular, if J varies from task to task, the workers will tend to wait more for each other. Nevertheless, this helps to understand what is going on, and to predict what will happen with changes.

How long does a serial system take to run if there is no parallelization involved? Since a job takes J microseconds to run, the speed of the serial system is by definition

$$\frac{1,000,000}{J} \text{ jobs/second} \tag{5}$$

Using this expression, we can derive the efficiency of the system, in terms of the ratio of the speed of the parallel system versus the speed of the serial system.

At the speed limit, the theoretical maximum efficiency of the system is

$$\frac{\text{speed of parallel system}}{\text{speed of serial system}} = \frac{\frac{1{,}000{,}000}{M}}{\frac{1{,}000{,}000}{J}} = \frac{J}{M} \tag{6}$$

Since M is a constant, this is equal to about $\frac{J}{100}$ in our system. That is, the top efficiency of the system depends upon how long a job-task takes. If the job takes 500 microseconds to execute (roughly 25 lines of code), then the parallel system can only run 5 times as fast as a serial system at maximum efficiency. If the job only takes 50 microseconds to execute (roughly 3 lines of code), then the parallel system can only run 50% as fast as a serial system at top speed. Longer jobs give better results when parallelized.

Under the speed limit, the theoretical efficiency of the system is

$$\frac{\text{speed of parallel system}}{\text{speed of serial system}} = \frac{\frac{1{,}000{,}000N}{(M+J)}}{\frac{1{,}000{,}000}{J}} = \frac{J}{(M+J)}N \tag{7}$$

which is linear in N, and depends upon how much larger J is than M. Again, different duration times will give different efficiencies.

These results predict that if *efficiency* is plotted against number of processors, instead of speed in jobs per second, then the graph will again consist of two lines, but this time the speed limit will occur at different heights ($\frac{J}{M}$), depending upon how long a job takes to perform. The corner of the curve will again occur at $N = \frac{J}{M} + 1$ processors, which is the number of processors that gives the maximum efficiency.

All of the preceding analysis has assumed that the agenda has already been filled by a previous process by the time the workers start working. A more realistic scenario has other processes competing with the workers in order to submit tasks to the agenda. Analysis of such a scenario is beyond the scope of this paper.

6.1.1 Conclusions

The results of this analysis show that the best measurement for an agenda or mailbox is its *speed* in number of jobs per second, and not the *efficiency* of the overall system. The speed depends on the amount of time that the agenda actually spends locked when getting one message, and not on the amount of time required for one processor to get one message. This can be measured by seeing how long it takes many processors to empty an angenda if each processor does nothing but pull messages off. It is expected that this time should be constant with the number of processors, once N is larger than one or two. Measuring the efficiency of an agenda by using a small job for testing does little to predict the efficiency of the agenda when a different-sized job is used, since efficiency is a function of job duration. The theoretical efficiency of an agenda system can be computed by determining the required duration for locking, popping, and unlocking the agenda, M, and by measuring the amount of time required to perform a job (including processing the message), J. M is about 100 microseconds for normal mailboxes; J needs an additional 30 microseconds added for cleanup time inside `receive-mb`. The actual efficiency will probably be slightly

less. This analysis has ignored the complications involved in submitting tasks to the agenda; this will slow things up even further.

7 PRELIMINARY TIMING RESULTS

Timing is difficult to evaluate, because it depends on the load on the time-sharing machine, whether the code has been swapped in or out of core, etc. Nevertheless, some guidelines are offered here.

7.1 Basic Parallel Instructions

A basic instruction, such as `setq`, +, or `if`, takes around a microsecond to run. It takes almost exactly 0.000001925 seconds (2 microseconds) for a processor to go through an empty `dotimes` loop once. This is the basis of the `burncycles` command.

The duration of an lwp that is computing using its own numbers is quite stable, and does not vary with the number of other processors that are running, as long as there is no interaction between the processor and the other processors (i.e., parallel constructs such as locks are not used).

A spin-lock takes 0.000015 (fifteen microseconds) to grab and release, using `with-spin-lock`.

The `pformat` command basically consists of grabbing a lock, pushing a message onto a mailbox, and releasing the lock, along with some evaluations and list manipulations. It takes about 0.0007 seconds (700 microseconds) to execute, if other processors are not printing things out at the same time. Under worst-case conditions, i.e. 15 processors doing nothing but printing things out, it degrades by a factor of 13 or so and takes about 0.009 seconds to execute per message. This is predictable, as one message has to wait for $15 - 1 = 14$ other messages to get accepted before it can be serviced. Note that the pformat process itself seems to wait until there is a free processor left from an application–it does not seem to start out running, so it does not block any application lwps from running.

7.2 Starting lwps from a standstill at the CLiP monitor

One of the most important commands is the `start-lwps` command which runs all of the light-weight processes that have been created. This command has been measured to take between 0.45 seconds and 1.12 seconds from the time that the command `start-lwps` is called, until the time that the *first* process starts up, with a normal time of around 0.53 seconds. This time does not seem to vary with the number of processes waiting to be started.

Processes are started in linear order, they are not turned loose all at once. The next existing process with a single `pformat` statement in it takes between 0.0013 seconds and 0.008 seconds to start from the main branch of a program using `start-lwps`, average around 0.002 seconds, once start-lwps has started running. The `start-lwps` program will continue to start lwps at 0.002-second intervals until all of the existing lwps have been started, or until all of the processors have been used up. In the latter case, waiting processes are started when the previous process dies. It takes between 0.03 seconds and 0.08 seconds, average about 0.045 seconds, for a a finished process to die off and a new one to get started.

A A DICTIONARY OF COMMANDS AND VARIABLES

(++ **number-loc**) This function is a different name for `incf`. It pulls a number out of a general variable-location, increments the number by 1, puts the results back into the same place, and returns the incremented results. The number can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`++l`) described in Section 2.4.1.

(++l **counter-location counter-lock**) Grabs the lock and increments the given counter by 1. Returns the new value.

(+= **number-loc increment**) This function is a different name for `incf`. It pulls a number out of a general variable-location, increments the number by the given increment, puts the results back into the same place, and returns the incremented results. The number and/or the increment can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`+=l`) described in Section 2.4.1.

(+=l **counter-location incr counter-lock**) Grabs the lock and increments the given counter by the given incr. Returns the new value.

(-- **number-loc**) This function is a different name for `decf`. It pulls a number out of a general variable-location, decrements the number by 1, puts the results back into the same place, and returns the decremented results. The number can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`--l`) described in Section 2.4.1.

(--l **counter-location counter-lock**) Grabs the lock and decrements the given counter by 1. Returns the new value.

(-= **number-loc decrement**) This function is a different name for `decf`. It pulls a number out of a general variable-location, decrements the number by the given decrement, puts the results back into the same place, and returns the decremented results. The number and/or the decrement can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

If you are using this function with multiple processes, you probably want the locked version (`-=l`) described in Section 2.4.1.

(-=l **counter-location decr counter-lock**) Grabs the lock and decrements the given counter by the given decr. Returns the new value.

(*= number-loc multiplicand) The multiply-in-place macro. This macro pulls a number out of a general variable-location, multiplies the number by the given multiplicand, puts the results back into the same place, and returns the new results. The number and/or the multiplicand can be floating-point. Similar to setf, the number's location can be any general location (such as an array reference), not just a variable name.

Currently there is no locked version of this macro.

(/= number1 .. numberN) Surprise! (/=) is the system-defined *not-equal* function. There is no in-place div-equals function as of yet. Use `(*= X (/ 1 Y))` for now, and let me know if you'd like a better one.

:A The :A option to defflav sets all variables to set, get, and init. It is equivalent to typing the flags `:settable-instance-variables :gettable-instance-variables :initable-instance-variables`.

(agenda-empty-p agenda) Tests to see whether an agenda is empty or not.

(alarm1 &optional (stream T)) Prints five beeps on the given stream.

:all Same as :A. The :all option to defflav sets all variables to set, get, and init. It is equivalent to typing the flags `:settable-instance-variables :gettable-instance-variables :initable-instance-variables`.

(allocate-MYRESOURCE) This function is the allocation function for resources. It should be used instead of the standard structure-instance allocation function `make-myresource` to allocate an instance of a resource that has been defined using `defstruct-resource`. It returns the newly allocated structure. The new structure will most probably contain evil garbage in its slots, and should be initialized by the user. This function is automagically defined when a `defstruct-resource` is executed.

(apush key item alist) Assoc-list push. Pushes a new assoc entry `(key . item)` onto the given assoc-list.

:B The :B option to defflav sets all variables to set, get, and init, the same as option :A. In addition, it sets the conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(x my-ship)`.

(beep &optional (stream T)) Prints a Cntrl-G on the given stream. This rings the bell on most terminals, including the Symbolics. Example: `(beep)`

(block-lwp LWP) The same as suspend. Note that (block) is a system command that means something completely different.

(block-me) A very useful command. Suspends the lwp that executes this statement. When the lwp is unblocked by someone else, execution picks up on the next line.

(bottom-of-heap heap) A Heap Package command. Returns two values: the bottom item of the heap, and also its priority key. Returns `NIL NIL` if the heap is empty. The bottom is the item with the *highest* number as its key. The bottom item is *not* removed from the bottom of the heap.

(bottom-of-heap-key heap) A Heap Package command. Returns the highest number in the heap. Returns NIL if the heap is empty.

(boundpq varname) This is the literal version of `boundp`. It tests to see whether its argument is bound (has a value) or not. It does not evaluate its argument. It does not break if its argument is unbound.

```
(setq foo 2)        ==>  2
(boundpq foo)       ==>  T
(boundp  weird)     ==>  ERROR: Attempt...
(boundpq weird)     ==>  NIL
```

(boundqp varname) Same as `boundpq`.

(build-fork-lwp fork-var (my-routine args...) &optional (name "A Forked Process")) Builds a **blocked** process to execute the given routine. The name of the routine is a literal that is not evaluated. Increments the active process count in the given fork. Returns the created lwp. Example:

```
  (setq parse-fork (make-fork))
  (setq parse-lwps ())
  (setq i 1)
  (push
    (build-fork-lwp parse-fork  (parser *sentence1-A*)
(string-append "Parser-" (my-string i))  )
    parse-lwps)
  (++ i)
  (push
    (build-fork-lwp parse-fork  (parser *sentence1-B*)
(string-append "Parser-" (my-string i))  )
    parse-lwps)

  ...

  (start-processes parse-lwps)
```

(build-fork-lwp-ev fork-var (my-routine-expr args...) &optional (name "A Forked Process")) Builds a **blocked** lwp that is part of a parallel fork. The name of the routine is contained in an expression that is evaluated. Grabs and increments the active process count in the given fork, pushes the lwp on the fork list. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq which-func 'parser)
(setq i 1)
(push
  (build-fork-lwp-ev parse-fork  (which-func *sentence1-A*)
(string-append "Parser-" (my-string i))  )
  parse-lwps)
(++ i)
(push
  (build-fork-lwp-ev parse-fork  (which-func *sentence1-B*)
(string-append "Parser-" (my-string i))  )
  parse-lwps)

...

(start-processes parse-lwps)
```

(build-lwp (my-routine args...) &optional (name "An LWP"))
Allocates and creates an lwp that starts out blocked. The lwp will run the given routine. Evaluates and then quotes its arguments. Does not evaluate the routine specification; this should be a literal. Returns the resulting lwp.

```
(build-lwp (parse *input-sentence*))
```

(build-lwp-ev (my-routine-expr args...) &optional (name "An LWP"))
Allocates and creates an lwp that starts out blocked. The lwp will run the given routine. Evaluates and then quotes its arguments. Evaluates the routine specification as well. Returns the resulting lwp.

```
(setq *which-routine* 'parse)
(build-lwp-ev (*which-routine* *input-sentence*))
```

(burn-cycles iteration-count) This does nothing but count up to iteration-count. Since each iteration takes almost exactly 1.925 microseconds to execute, this command can be used for microsecond timing.

:C The :C option to defflav sets all variables to set, get, and init, and also sets the conc-name to NIL, the same as option :B. In addition, it sets the constructor function name to `make-`*name*. So, for instance, to create an instance of a ship, the user types `(make-ship)`. This option is recommended.

(chararray-to-string chararray &optional(start 0)(end+1 (length chararray)))
This function converts an array of characters into an equivalent string. The array must contain characters, not strings. The first optional argument indicates the array index containing the starting letter of the string (inclusive). The second optional argument indicates the array index of the ending letter of the string, plus one (exclusive).

```
my-array                                ==>  #(#\a #\b #\c)
(chararray-to-string my-array)          ==>  "abc"
(chararray-to-string my-array 1)        ==>  "bc"
(chararray-to-string my-array 0 2)      ==>  "ab"
```

(check-pformat) This function returns the state of the pformat process. It should be :RUNNABLE. If it is :TERMINATED, the process must be re-initialized with `init-pformat`.

(clear-heap heap) A Heap Package command. Clears a given heap out; makes it empty. Currently this is implemented by setting the variable to a new heap.

(close-file) Closes an existing OS file. Prints an error message to the terminal if it is called twice, or if the file is already closed. Resets both `OS` and `ES` to `T`. Returns the string or descriptor that was stored in `*using-file*`. Does not take any arguments. Example:

```
(close-file)

"/usr1/myers/recording.text"
```

(:conc-name newname) This option to defflav sets the flavor conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(newname-x my-ship)`.

:concnil This option to defflav sets the flavor conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(x my-ship)`.

:D The :D option to defflav sets all variables to set, get, and init, and also sets the conc-name to NIL, the same as option :B. In addition, it sets the constructor function name to *name*. So, for instance, to create an instance of a ship, the user types `(ship)`.

(deallocate-MYRESOURCE my-instance) This function is the deallocation function for resources. It throws an instance back into the resource pool. The next time the same kind of resource is allocated, instead of using new data space, the allocation function will return this old instance. Since the instance will contain garbage, it is up to the programmer to clear it out and initialize it properly. It is a mistake to continue using an instance that has been deallocated; the user program should deallocate an instance only after it is sure that the program is finished with that instance.

(defflav flavor-name (slot-vars) (parents) :options :option-codes) Defines a flavor named `flavor-name` that inherits from `parents`, uses the given slot variables, and is built using the given (optional) options and/or option codes. Example:

```
(defflav ship   ( x  (y 0) ) (vehicle thing) :C)
(defflav rocket ( x  y  z  ) (vehicle thing) :D)
```

This defines a flavor `ship`. It has slots `x` and `y`. The `y` slot has a default initialization of 0; the `x` slot has no default initialization. The ship inherits slots from parent flavors `vehicle` and `thing`. Note multiple parents are no problem. Often the parent list will simply be nil: (). The ship is defined using option code :C, which, as explained below in section 2.2.1, makes y and x readable, writable, and initable, and creates the creation function (`make-ship`) for creating instances of the flavor.

It is recommended that the option flag :C be used in most cases.

(defmeth (method-name flavor-name) (arg1 ... argN) &body) Defines a method that is used by the given flavor. It is permissable to have different flavors use the same method name; the system automatically determines which method is correct based on what flavor of object is used in the call. The different flavor methods can even have a different number of arguments. As usual, the slot variables of the flavor instance are local variables inside the method and can be referenced and `setq`'d directly; there is no need to use the access functions inside the method. Also as usual, the special local variable `self` is bound to the flavor instance. The body can be a sequence, it does not have to be a list.

Examples:

```
(defmeth (move ship) (new-x new-y)
  "This method moves the ship to a new (x,y) coordinate."
(setq x new-x)
(setq y new-y)
)

(defmeth (move rocket) (new-x new-y new-z)
  "This method moves the rocket to a new (x,y,z) coordinate."
(setq x new-x)
(setq y new-y)
(setq z new-z)
)
```

This shows that different flavors can have methods with the same name.

(defstruct-resource (myresource :named) slot1 ... slotn) This function is the defstruct function for resource structures. The syntax is exactly the same as `defstruct`. This function defines a resource. After this function is called, the user can call the allocate-*myresource* program to get instances. Default initializations should not be specified on the slots, since the user should explictly fill in each slot whenever an instance is allocated.

(delete-from-heap heap key item) A Heap Package command. Deletes a given keyed item from the heap. Returns two values: the deleted item, and also its key. Both the key and the item must match the corresponding entry in the heap with `equal`.

(describe flavor-instance) This is the system command for printing out the slots and slot values of a flavor instance.

(div2 x) Integer divide-by-2. Returns the integer representation of the number `x` represented as a signed binary number, and then shifted right one place, filling the sign bit. Takes floating-point numbers as input, but basically forgets the decimal. This function does the right thing when working with binary negative numbers. Note that this might not be what you'd expect if you didn't think about it:

```
(div2  4) ==>   2    [0100  ==>  0010]
(div2  3) ==>   1    [0011  ==>  0001]
(div2 -4) ==>  -2    [1100  ==>  1110]
(div2 -3) ==>  -2    [1101  ==>  1110]
```

(dump-pformat) This function performs a dump of the pformat-MB. Any and all old format messages that are currently in the mailbox are printed out, and then discarded. The function returns the number of old format messages that were dumped out. This routine is useful for seeing dead messages if pformat is broken, or if the lwps are not running at the moment. It should only be used in unusual cases. See `reset-pformat`.

(end-of-fork old-fork-var) This is a low-level routine that could be used by users attempting to build custom join routines, or to seal off forks that have no joins. It is the same as "join". It decrements the active processes count and returns the new value.

ES This variable can be used as the Error Stream. You can use it for all minor error-message output in your program. It is bound to `T` as a default. Perform a `(setq ES OS)` after opening a file if you want error messages to go to the recording output file. Note that major error messages should use the stream T to the terminal; otherwise, the user will not be notified when something bad has happened. ES is reset to T by routine `close-file`.

(exact-string-test function-call expected-results-string) This macro is used for automatically testing functions that are supposed to return well-known answers. For instance, if a routine is known to work properly, but then the code gets changed, it is useful to run a suite of test programs on the routine in order to check out that it still works correctly. Exact-string test accepts a function call, executes the function, and gathers all of the output to stream OS into an internal string variable. It then does a `string-equal` comparison against the expected-results string, which should exactly correspond. If the strings are equal, the routine prints a small verification message, and returns T. If the strings are not equal, the routine prints a large complaining message, and prints out the output that was actually obtained; it then returns NIL. Watch out for spaces and carriage returns in the expected string. It is important that the tested routine send its output to OS; the results that are *returned* by the tested routine are not examined. Examples:

```
(defun my-func (x) (format OS "Results: ~A" (* x x)))
(exact-string-test (my-func 3)  "Results: 9")
  Exact-string-test:  (MY-FUNC 3) passes the test.    ==> T
(exact-string-test (my-func 3)  "BadMatch")
  EXACT-STRING-TEST:  (MY-FUNC 3) FAILS THE TEST.
  Actual output:
  Results: 9       ==> NIL
```

(filep stream) The same as `(file-p stream)`.

(file-p stream) This function is supposed to test whether a given stream is an output file or not. Currently it tests to see whether a file is not the terminal IO stream or if it is, which gives similar but not precise answers. If you want to use this, please talk to me.

(find-heap-item heap key item) A Heap Package command. Searches for the given item and key inside the given heap. Both the item and the key must match, using `equal`. Returns a heap object representing the subheap that the item was found in. Returns NIL if the item was not found. The subheap object is replaceable in the main heap using setf, for heap twiddling. The searched-for item is *not* guaranteed to be at the top of the returned subheap object.

(flavorname &optional :slotvar1 init1 ... :slotvarM initM) Instance creation routine used in the New Flavors package if the option-code :D is used in `defflav`. Creates an instance of the given flavorname. Example:

```
(setq my-ship  (ship :x 5) )
```

Any slot variables that are not specified receive their default initialization values. Any slot variables that are not specified and do not have initialization values are set to NIL, they are not left unbound.

(flavorname-slotvarname flavor-instance) This is the default method of referencing a slot's value, as provided by the New Flavors package. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were **NOT** used. Example:

```
(ship-x my-ship)      ==> 10
```

(fork-lwp fork-var (my-routine args...) &optional (name "A Forked Process")) Spawns a **running** process to execute the given routine. The name of the routine is a literal that is not evaluated. Increments the active process count in the given fork. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq i 1)
```

```
  (fork-lwp parse-fork  (parser *sentence1-A*)
(string-append "Parser-" (my-string i))  )
  (++ i)
  (fork-lwp parse-fork  (parser *sentence1-B*)
        (string-append "Parser-" (my-string i))  )
```

(fork-lwp-ev fork-var (my-routine-expr args...) &optional (name "A Forked Process")) Spawns a **running** process to execute the given routine. The name of the routine is contained in an expression that is evaluated. Increments the active process count in the given fork. Returns the created lwp. Example:

```
(setq parse-fork (make-fork))
(setq which-func 'parser)
(setq i 1)
(fork-lwp-ev parse-fork  (which-func *sentence1*)
(string-append "Parser-" (my-string i))  )
(++ i)
(fork-lwp-ev parse-fork  (which-func *sentence1*)
(string-append "Parser-" (my-string i))  )
```

(fstring item) This is the same as `(f-string)`.

(f-string item) Coerces the item into a string, by returning its printed representation. The name comes from "forced" `string`—this is an improvement over the normal `(string)` function, which breaks when given numbers, and sometimes when given lists. This function should work no matter what `item` is. See also `trunc`.

:get Option to defflav. See :readable-instance-variables. Short for :gettable-instance-variables.

(:get sequence-of-slotnames) Option to defflav. See (:readable-instance-variables). Short for :gettable-instance-variables.

(get-elapsed-time start-time-in-internal-ticks) Returns the amount of time that has elapsed, in **seconds**, from the time that the start-time was recorded. Start-time must be in internal-tick units. Unfortunately, this function is only accurate to a **few hundreths of a second**–however, a basic operation takes about a microsecond to perform. Thus, this function is useless in timing anything that does not have more than about 5000-10000 basic operations in it. [Besides this, since it uses simple subtraction, there may be a possibility that this function will break once a month. E.g., for five seconds at midnight on the 31st, this function will return wrong answers. Do not use it to time elapsed durations of more than about a day.] See also `burn-cycles`.

Example:

```
(setq old-time (get-time))
...
(when (> (get-elapsed-time old-time) 5.0))
   (pformat T "You're taking longer than five seconds to think!!!~%"))
```

(get-time) Returns the "internal real time", in internal-tick units (milliseconds). This command is only accurate to a few hundreths of a second.

There is a rumour that the time is different when asked between different processors. This is currently being investigated.

(heap) A Heap Package command. Returns an empty heap. Same as `(make-heap)`. Currently an empty heap is implemented as an empty list `()`.

(heap-empty-p heap) A Heap Package command. Tests to see whether a given heap is empty or not. Returns T or NIL. See heap-full-p.

(heap-find-item heap item) Searches for the given item inside the given heap. The item must match, using `equal`. Returns a heap object representing the subheap that the item was found in. Returns NIL if the item was not found. The subheap object is replaceable in the main heap using setf, for heap twiddling. The searched-for item is *not* guaranteed to be at the top of the returned subheap object. This function searches exhaustively and will take longer than find-heap-item.

(heap-full-p heap) A Heap Package command. Tests to see whether the given heap has at least one entry or not. Returns the heap or NIL. This is the preferred test.

(heap-bottom-N-items heap N) Returns a list of the bottom N items with the highest keys, and a second value of the number of entries returned. The list is ordered from highest to lowest, by key.

(heap-bottom-Nth-item heap N) Returns the item that is Nth from the bottom counting down from the highest entry, and a second value of its inverse ranking, which will usually be N. If the heap has less than N entries, the topmost entry is returned.

(heap-top-N-items heap N) Returns a list of the top N items with the lowest keys, and a second value of the number of entries returned. The list is ordered from lowest to highest, by key.

(heap-top-Nth-item heap N) Returns the item that is Nth from the top counting up from the lowest entry, and a second value of its ranking, which will usually be N. If the heap has less than N entries, the bottommost entry is returned.

:init Option to defflav. Short for :initable-instance-variables. Makes all slot variables initable.

(**:init sequence-of-slotnames**) Option to defflav. Short for :initable-instance-variables. Makes the given slot variables initable.

(**init-pformat**) This function is called by `sq-system` when it is loaded. It allocates the PFORMAT-PROCESS lwp and stores it in system variable `pformat-process`. It sets `pformat-MB` to a new mailbox. The pformat process should not be blocked or killed by the user. For this reason, the user should never have to call this function.

It is important not to call the system routine `(reset-queues)` after the `beholder-2` package is loaded, as this will break pformat by terminating the pformat process, causing all messages to stack up in the mailbox and not get printed. If `(reset-queues)` is called, the user must call `(init-pformat)` afterwards to initialize a new pformat process. Remember that this also wipes out the mailbox.

(**intern-soft string &optional package**) This is the Symbolics-compatable version of the Common Lisp `find-symbol` function. It returns the symbol associated with the string, or NIL if the symbol has not yet been interned. It is used by `sys-make-name`.

(**join old-fork-var**) This is a low-level routine that could be used by users attempting to build custom join routines. It decrements the active processes count and returns the new value. Most users will want to use `join-and-when-finished-do` instead.

(**join-and-when-finished-do old-fork-var body...**) Decrements the number of active processes in the fork by 1. Executes the code contained in **body** when the last process finishes (calls this function). This routine should come at the end of the body of the lwp procedure.

(**kill-all-lwps**) Kills all the lwps on the *lwps* list.

(**++l counter-location counter-lock**) Grabs the lock and increments the given counter by 1. Returns the new value.

(**--l counter-location counter-lock**) Grabs the lock and decrements the given counter by 1. Returns the new value.

(**+=l counter-location incr counter-lock**) Grabs the lock and increments the given counter by the given incr. Returns the new value.

(**-=l counter-location decr counter-lock**) Grabs the lock and decrements the given counter by the given decr. Returns the new value.

(**leash-all-workers agenda**) Deactivate all of an agenda's workers.

(**leash-N-workers N agenda**) Deactivate some of an agenda's workers.

(**list-of-heap-items heap**) Returns a list of the items in the heap. The list is ordered from lowest to highest, by key.

(list-of-heap-items-and-keys heap) Returns a list of pairs of (item key), the items in the heap paired with their keys. The list is ordered from lowest to highest, by key.

(list-of-heap-keys heap) Returns a list of the keys of the items in the heap. The list is ordered from lowest to highest, by key.

lwps This variable holds all the lwps that the system knows about. This includes live, blocked, and dead lwps. This variable was previously used by `fork`, `build-fork`, `init-pformat`, etc., but is now completely maintained *by the user*. It is mainly useful because `kill-all-lwps` uses it as input. When you create a process by yourself (e.g., by using `fork`), you should push the process onto `*lwps*`, so that you can kill it later if you need to.

Example:

```
(push (spawn (my-process args))  *lwps*)
```

(make-agenda) Returns an agenda.

(make-flavorname &optional :slotvar1 init1 ... :slotvarM initM) Instance creation routine used in the New Flavors package if the option-code :C is used in `defflav`. Creates an instance of the given flavorname. Example:

```
(setq my-ship  (make-ship :x 5) )
```

Any slot variables that are not specified receive their default initialization values. Any slot variables that are not specified and do not have initialization values are set to NIL, they are not left unbound.

(make-fork) Creates and returns a fork record-keeping structure. Store this in a variable and use it to control the following functions.

(make-heap) A Heap Package command. Returns an empty heap. Same as `(heap)`. Currently an empty heap is implemented as an empty list `()`. Be careful; "make-heap" returns a Symbolics heap structure when this code is run on the Symbolics computer.

(make-instance 'flavorname &optional :slotvar1 init1 ... :slotvarM initM) Instance creation routine used by the old flavors package. This is the default method of constructing a flavor instance that is used if the :C or :D option code is not specified. Also, even if the :C or :D code is used for a flavor, this syntax is still valid and can be mixed with that of the New Flavors package.

This function creates an instance of the given flavorname. Example:

```
(setq my-ship  (make-instance 'ship :x 5) )
```

Any slot variables that are not specified receive their default initialization values. Any slot variables that are not specified and do not have initialization values are set to NIL, they are not left unbound.

(make-N-workers N my-worker-function agenda) Allocates N workers for the given agenda. N should be a positive integer. My-worker-function must be an unquoted literal.

(make-worker my-worker-function agenda) Allocates a worker for the given agenda. My-worker-function must be an unquoted literal.

master-process This variable is used by `Start-Master` to determine which process to start. It is used for looping process chains. It should store a process, if you want to take advantage of the looping feature. Set this variable to a process that sets up your system, and then call `Start-Master` as the last thing before the last join dies.

(methodname flavor-instance arg1 ... argN) Invokes the method function on the given flavor instance object. Which method function is invoked depends upon which flavor the object is an instance of. Example:

```
(move my-ship 10 20)      ==> 20
```

This calls method `move` on the object `my-ship`. Since `my-ship` is an instance of flavor `ship`, it uses the `move ship` method defined in the previous `defmeth` example. The ship x and y slots are set to 10 and 20 respectively; the method returns the last line in the method definition, which evaluates to 20.

(*my-worker-function* item) User-defined worker functions for a particular agenda must take one and only one argument, which gets bound to the item that is popped off the agenda–the same as the item that was originally pushed onto the agenda.

(neq a b) Returns NIL if `a` is `eq` to `b`, T otherwise.

:nilconc This option to defflav sets the flavor conc-name to NIL, so that e.g. instead of saying `(ship-x my-ship)` to reference slot variable x, the user says `(x my-ship)`.

(only-once list) This function nondestructively returns a copy of the given list in which every atom is listed only once–duplicates are eliminated. The function uses `eq` for comparison. The resulting order is reversed. This is unfortunately an $O(\frac{1}{2}n^2)$ operation.

```
(only-once '(a a a b b b a c b c c c))   ==>  (C B A)
```

(open-file filename) Opens output stream `OS` for serial output into a new copy of the given file. `filename` should evaluate to a string or file descriptor indicating the appropriate file. This function breaks if `filename` indicates a

nonsense path, e.g. to a machine that does not exist. The scratch variable *using-file* is set to the string or value passed in filename. When this function is called from a terminal, it prints out a reminder message. No useful value is returned if this function is called from within a program; however, OS is set to a legitimate file stream on a successful call, as a side-effect. In general, it is important to specify the full pathname of the file; the default directory on the Sequent seems to be /usr, which should not be used by normal users, and will probably give you a Permission Denied error anyway.

Franz Lisp does not seem to support remote files. Certainly the ''MACHINE:pathname" colon syntax is NOT supported for at least output files.

Note that pformat supports the use of various streams, including ES and OS.

```
(open-file "/usr1/myers/recording.text")

"Opened stream #<stream writing /usr1/myers/recording.text @ #x9294e9>
Do a (setq ES OS) if desired.
"
```

OS This variable is used as the Output-file Stream. You can use it for all output in your program. It is bound to T as a default. It is used by routines open-file and close-file.

(pformat stream control-string args...) This function is the parallel version of the regular format command. The arguments are exactly the same as format. First, the function tests to see whether only one processor has been allocated or not. If so, it assumes that a top-level program is running, and it prints out the formatted message directly. If not, the function evaluates its arguments, quotes the result, and pushes a print request containing the quoted evaluations onto a system mailbox. Later, a special dedicated lwp executes the request and prints the output. pformat should be used in lwp code in all cases instead of format.

pformat-MB This system variable stores a mailbox that is used to implement pformat. It should not be examined or modified by the user.

pformat-process This system variable stores the lwp process that is used to implement pformat. It should not be examined or modified by the user.

(pop-heap heap) A Heap Package command. Pops the top item off the heap (the item with the *lowest* number as its key). Returns two values: the item, and also its priority key. Returns NIL NIL if the heap is empty. The popped item is removed from the top of the heap.

(ppformat stream control-string args...) This function is the direct version of pformat, the parallel version of format. It does not test how many processors have been allocated, but rather sends the output directly to the pformat

mailbox. It is slightly faster than `pformat`, and should be used when the programmer is certain that only parallel processes will be used and there is no need for running the program at the monitor level. It does not print out its messages until `start-lwps` has been called. The arguments are exactly the same as `format`. The `ppformat` function evaluates its arguments, quotes the result, and pushes a print request containing the quoted evaluations onto a system mailbox. Later, a special dedicated lwp executes the request and prints the output.

(pull inlist item1 ... itemN) Destructively Pulls items onto the back of the argument `inlist`, in place. Returns the new list. Designed to complement `push`, which puts things on the front of the list. Note that both the list and the items are eval'ed.

```
(setq x '(a b c))   ==>  (a b c)
(setq y '(d f))     ==>  (d f)
(pull x (cdr y) 'e)      ==>  (a b c (f) e)
x                   ==>  (a b c (f) e)
(setq x NIL)        ==>  NIL
(pull x 'd 'e)      ==>  (d e)
x                   ==>  (d e)
```

This new, improved version of pull is just a hair slower but it does the right thing when `x` is NIL. It also does the right thing in list-bashing a new copy of the atoms onto the back, avoiding those kinds of strange stack problems. Also, since this macro evaluates all of its arguments, you would have to be really creative to get yourself into problems storing local variables on a global list. It should be safe.

If `x` isn't a list you've got problems.

(pull-heap heap) A Heap Package command. Pulls the bottom item off of the heap (the item with the *highest* number as its key). Returns two values: the item, and also its priority key. Returns `NIL NIL` if the heap is empty. The pulled item is removed from the bottom of the heap.

(pullq inlist atom1 ... atomN) Destructively Pulls atoms onto the back of the argument `inlist`, in place. Returns the new list. Designed to complement `push`, which puts things on the front of the list. Note carefully that the list is eval'ed, but that the atoms aren't.

```
(setq x '(a b c))   ==>  (a b c)
(pullq x d e)       ==>  (a b c d e)
x                   ==>  (a b c d e)
(setq x NIL)        ==>  NIL
(pullq x d e)       ==>  (d e)
x                   ==>  (d e)
```

This new, improved version of pullq is just a hair slower but it does the right thing when x is NIL. It also does the right thing in list-bashing a new copy of the atoms onto the back, avoiding those kinds of strange stack problems.

Please do NOT use this function to pull local variables (from a `let` or from a function's arguments) onto the back of a list, and then exit from the local lexical definition. You will reuse part of the Lisp stack that is being assigned to something else, and you will be sorry.

If x isn't a list you've got problems.

(push-agenda item agenda) Pushes a given item onto the given agenda.

(push-heap heap key item) A Heap Package command. Pushes the given item on the given heap, using the given key. The key *must* evaluate to a number.

(pushnew-agenda item agenda) Pushes a new given item onto the given agenda, if it wasn't in the agenda waiting to be processed already.

(rand x) This is a locked version of the CL function `(random x)`, which takes a positive number x and returns a number of the same type from 0 (inclusive) up to but not including x. I.e., if x is an integer, numbers from 0 to (x-1) are returned; if x is a floating-point number, numbers from 0 to $(x - \epsilon)$ are returned. It is an error not to supply the argument x when calling this function.

rand-lock This system variable stores a lock that is used to implement `rand`. It should not be examined or modified by the user.

:read Option to defflav. See :readable-instance-variables. Short for :gettable-instance-variables.

(:read sequence-of-slotnames) Option to defflav. See (:readable-instance-variables). Short for :gettable-instance-variables.

:readable-instance-variables Option to defflav. Translates the Symbolics syntax into :gettable-instance-variables. Makes all slot variables gettable.

(:readable-instance-variables sequence-of-slotnames)
Option to defflav. Translates the Symbolics syntax into :gettable-instance-variables. Makes the given slot variables gettable.

recording-statistics This flag is used by the worker/agenda package. If it is non-NIL, the agenda package gathers statistics on how long each worker takes to accomplish its job. These can be used by the BEHOLDER package to determine expected process durations.

(rekey-heap-item heap key item newkey) A Heap Package command. Labels an existing item in the given heap with the given new key; reorders the heap to reflect the new status. The new key must be a number. Both the old key and the item must match the corresponding entry in the heap with `equal`. This routine should be faster than deleting the item from the heap and then pushing it in again with the new key.

(reset-pformat) This function performs a flush of the pformat-MB. Any and all old format messages currently in the mailbox are discarded. The function returns the number of old format messages that were thrown out.

(second-value multi-valued-function) This function returns the second value of a multiple-valued function. As a bonus, the first value of the multiple-valued function is returned as a second value from this function call.

```
(second-value   (values 1 2))      ==>  2    1
```

(send flavor-instance :SET-slotvarname newvalue)
This is the default method of setting a slot's value, as provided by the old flavors package. This syntax is still valid, even if the New Flavors package is being used. Example:

```
(send  my-ship :x)      ==> 10
```

(send flavor-instance :slotvarname) This is the default method of referencing a slot's value, as provided by the old flavors package. This syntax is still valid, even if the New Flavors package is being used. Example:

```
(send  my-ship :x)      ==> 10
```

:set Option to defflav. See :writable-instance-variables. Short for :gettable-instance-variables and :settable-instance-variables.

(:set sequence-of-slotnames) Option to defflav. See (:writable-instance-variables). Short for :gettable-instance-variables and :settable-instance-variables.

(setf (flavorname-slotvarname flavor-instance) newvalue) This is the default method of setting a slot's value, as provided by the New Flavors package. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were **NOT** used. Example:

```
(setf (ship-x my-ship) 10)      ==> 10
```

(setf (slotvarname flavor-instance) newvalue) This is the normal method of setting a slot's value. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were used. Example:

```
(setf (x my-ship) 10)      ==> 10
```

(SET-slotvarname flavor-instance newvalue) This is a bonus function for setting a slot's value that is automatically defined by the New Flavors package. This syntax is usable for any flavor, New Flavors or not, as long as the New Flavors package has been loaded. This syntax is still valid, even if the New Flavors package is being used.

```
(set-x my-ship 10)      ==> 10
```

(size-of-heap heap) Returns a count of the number of items in the heap.

(slotvarname flavor-instance) This is the normal method of referencing a slot's value. This syntax is used if one of the :B, :C, :D, :nilconc, or :concnil option codes were used. Example:

```
(x my-ship)      ==> 10
```

(spawn-lwp (my-routine args...) &optional (name "An LWP")) Allocates and creates an lwp that starts out running. The lwp will run the given routine. Evaluates and then quotes its arguments. Does not evaluate the routine specification; this should be a literal. Returns the resulting lwp.

```
(spawn-lwp (parse *input-sentence*))
```

(spawn-lwp-ev (my-routine-expr args...) &optional (name "An LWP")) Allocates and creates an lwp that starts out running. The lwp will run the given routine. Evaluates and then quotes its arguments. Evaluates the routine specification as well. Returns the resulting lwp.

```
(setq *which-routine* 'parse)
(spawn-lwp-ev (*which-routine* *input-sentence*))
```

(squish nested-list) This function takes a simple nested list and non-destructively returns a copy of the same list without all of the interior parenthises–the list gets "squished" down to one level. Order is preserved. Given a non-list atom, the function returns a list of that atom.

```
(setq foo '(a ((b (c) d))) )          ==>  (A ((B (C) D)))
(squish foo)                          ==>  (A B C D)
foo                                   ==>  (A ((B (C) D)))
(squish 'a)                           ==>  (A)
```

(start-processes list-or-sequence-of-processes) Starts a list of blocked processes: sends a resume signal to each process on the list.

```
(start-processes lwp1 lwp2 lwp3)

(setq parser-lwps (list lwp1 lwp2 lwp3))   ;could be '',lwp1 .
(start-processes parser-lwps)
```

(string-append "string1" ... "stringN") Returns a string that is the concatenation of the previous strings. Example: `(string-append ''Foo'' ''bar'' ''Baz'') ==> ''FoobarBaz''`

(string-length item) This is an improved version of `length`, that tries to "do the right thing". It offers basic compatability with the Symbolics Lisp function of the same name. If `item` is a string, it returns a count of the number of characters in the string. If `item` is a char, it returns 1. If `item` is an array, e.g. an array of characters, it returns the length of the array. Otherwise, it uses `f-string` to coerce the value of the argument into a string, and then returns the length of that string.

(sys-make-keyword "STRING-1" ... "STRING-N") This macro first forms a keyword name by concatenating the (one or more) given strings. It then searches the current name-space and returns the existing keyword symbol which uses that name as its print-name. The resulting expression returns the keyword variable. This code is useful for referencing keywords if you will only know or be able to compute the name of an existing keyword at runtime. The user should *not* include the ":" character in the keyword definition. A keyword symbol should evaluate to itself.

```
(sys-make-keyword "STR" "EAM")                ==>  :STREAM     :EXTERNAL
(setq bar (sys-make-new-name "STREAM"))       ==>  :STREAM
bar                                           ==>  :STREAM
(eval bar)                                    ==>  :STREAM
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function only finds existing keyword symbols. If the symbol has not been interned as a keyword yet, the function returns NIL.

```
(setq bar (sys-make-keyword "WEIRD"))          ==>    NIL
```

(sys-make-name "STRING-1" ... "STRING-N") This macro first forms a name by concatenating the one or more given strings. It then searches the current name-space and returns the existing symbol which uses that name as its print-name. The resulting expression returns the variable; it must be eval'ed one more time to reference the contents of the variable. This code is useful for referencing variables if you will only know or be able to compute the name of an existing variable at runtime.

```
(setq foo2 0)
(sys-make-name "FOO2")                          ==>  FOO2     :INTERNAL
(set (sys-make-name "FOO" "2")  5)          ;note this is not a setq
foo2                                            ==>  5
(+  (eval (sys-make-name "FO" "O" "2")) 5)    ==>  10
(setq bar (sys-make-name "FOO2"))               ==>  FOO2
bar                                             ==>  FOO2
(eval bar)                                      ==>  5
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function only finds existing symbols. If the symbol has not been interned as part of the name-space yet, the function returns NIL.

```
(setq bar (sys-make-name "WEIRD"))          ==>   NIL
bar                                         ==>   NIL
```

(sys-make-new-keyword "STRING-1" ... "STRING-N") This macro first forms a keyword name by concatenating the one or more given strings. It then searches the current name-space and returns the keyword symbol which uses that name as its print-name, if it exists; otherwise, it creates a new keyword symbol and returns that. This code is useful for making new keywords. The user should *not* include the ":" character in the keyword definition. A keyword symbol should evaluate to itself.

```
(sys-make-new-keyword "MY-" "STR" "EAM")      ==>  :MY-STREAM   :EXTERNAL
(setq bar (sys-make-new-keyword "MY-STREAM")) ==>  :MY-STREAM
bar                                           ==>  :MY-STREAM
(eval bar)                                    ==>  :MY-STREAM
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function finds existing keyword symbols. If the keyword symbol has not been interned as part of the name-space yet, the function returns a new keyword symbol.

```
(setq bar (sys-make-new-name "WEIRD"))      ==>   :WEIRD
bar                                         ==>   :WEIRD
(eval bar)                                  ==>   :WEIRD
```

(sys-make-new-name "STRING-1" ... "STRING-N") This macro first forms a name by concatenating the (one or more) given strings. It then searches the current name-space and returns the symbol which uses that name as its print-name, if it exists; otherwise, it creates a new named symbol and returns that. The resulting expression returns the variable; it must be eval'ed one more time to reference the contents of the variable. This code is useful for making new variables.

```
(setq foo2 0)                               ==>   0
(sys-make-new-name "FOO2")                  ==>   FOO2    NIL
```

```
(set (sys-make-new-name "FOO" "2")  5)   ;note this is not a setq
foo2                                              ==>   5
(+  (eval (sys-make-new-name "FO" "O" "2")) 5)  ==>   10
(setq bar (sys-make-new-name "FOO2"))             ==>   FOO2
bar                                               ==>   FOO2
(eval bar)                                        ==>   5
```

IT IS VERY IMPORTANT THAT THE LETTERS IN THE STRING ARGUMENTS GIVEN FOR THE NAME BE IN UPPER-CASE. This function will not do what you want it to do otherwise. Numerals and non-alphabetic characters are perfectly fine.

This function finds existing symbols. If the symbol has not been interned as part of the name-space yet, the function returns a new symbol.

```
(setq bar (sys-make-new-name "WEIRD"))        ==>    WEIRD
bar                                           ==>    WEIRD
(set bar 20)                                  ==>    20
weird                                         ==>    20
```

(top-of-heap heap) A Heap Package command. Returns two values: the top item of the heap, and also its priority key. Returns `NIL NIL` if the heap is empty. The top is the item with the *lowest* number as its key. The top item is *not* removed from the top of the heap.

(top-of-heap-key heap) A Heap Package command. Returns the lowest number in the heap. Returns NIL if the heap is empty.

(trunc item length) This function returns a string which is the name of the item, or a truncated version if the string would be longer than the given length. The item is forced to be converted into a name by using `(f-string)`. If the resulting name is longer than `length` characters, the name string is truncated to (length - 1) characters and a tilde character "" is appended to indicate truncation; this resulting string is then returned. `item` can be just about anything. `length` must be an integer that is 1 or greater. This function does not pad the resulting string to ensure that it is exactly equal to length; it could be smaller. Note that this function returns a string, it does not do any printing; however, the results can be used as an argument to `format`. The function has been used under Symbolics Lisp for printing labels of graphical nodes.

(type-the-time &optional (stream T)) Types out the time in a pretty format. Returns NIL. If the optional stream argument is NIL, does not type the time out, but returns a string of what it would have typed out.

```
(type-the-time nil)   ==>  "4:47:26 PM   Thur  Jun 20, 1991"
```

(unblock LWP) The same as resume. Note that (unblock-lwp) is apparently already a system command with unknown effect.***

(**unleash-all-workers agenda**) Activates all of an agenda's workers. Normally call this right after allocating all workers. Active workers stay active whether they are executing an agenda item or waiting for the agenda to fill.

(**unleash-N-workers N agenda**) Activate some of an agenda's workers.

(**use-file filename**) Opens a file for output to OS. The same function as `open-file`.

using-file This variable is bound to the string or descriptor that was used to open the currently open file by routine `open-file`. It is used for information purposes only.

(**wait nsecs**) Wait implements a "sit-and-spin" function that ties up a processor until the given number of seconds has elapsed. Wait uses `get-elapsed-time` and should not be used to time durations of more than about a day, to be safe. **Wait can only be used safely for durations with hundreths of a second precision.** It fixes the problem introduced by `sleep`, which quietly truncates its argument down to integer values and is basically useless. See the discussion in Section 3.4.

(**wait-for-empty-agenda agenda**) Waits until an agenda is empty. Detects this by waiting for the agenda to send an "empty" message to itself, which happens when the agenda notices the number of running active workers is 0.

(**with-open-file (IS input-filename :direction :input) BODY**) This system-defined function is the approved method of reading from an input file stream IS. It uses an unwind-protect to ensure that the input file is closed if an error occurs in the body of the routine.

(**with-spin-lock-or-NIL lock body...**) This enclosing macro supports optional locks. If `lock` is NIL, the macro goes ahead and evaluates the body. If `lock` is a lock, the macro is the same as a `with-spin-lock` form.

:write Option to defflav. See :writable-instance-variables. Short for :gettable-instance-variables and :settable-instance-variables.

(**:write sequence-of-slotnames**) Option to defflav. See (:writable-instance-variables). Short for :gettable-instance-variables and :settable-instance-variables.

:writable-instance-variables Option to defflav. Translates the Symbolics syntax into :settable-instance-variables and :getable-instance-variables. Makes all slot variables settable and gettable.

(**:writable-instance-variables sequence-of-slotnames**) Option to defflav. Translates the Symbolics syntax into :settable-instance-variables and :getable-instance-variables. Makes the given slot variables settable and gettable.