Medley uses "deep binding." Every time a function is entered, a basic frame containing the new variables is put on top of the stack. Therefore, any variable reference requires searching the stack for the first instance of that variable, which makes free variable use somewhat more expensive than in a shallow binding scheme. On the other hand, spaghetti stack operations are considerably faster. Some other tricks involving copying freely-referenced variables to higher frames on the stack are also used to speed up the search.

The basic frames are allocated on a stack; for most user purposes, these frames should be thought of as containing the variable names associated with the function call, and the *current* values for that frame. The descriptions of the stack functions in below are presented from this viewpoint. Both interpreted and compiled functions store both the names and values of variables so that interpreted and compiled functions are compatible and can be freely intermixed, i.e., free variables can be used with no `SPECVAR` declarations necessary. However, it is possible to *suppress* storing of names in compiled functions, either for efficiency or to avoid a clash, via a `LOCALVAR` declaration (see the Local Variables and Special Variables section of Chapter 18). The names are also very useful in debugging, for they make possible a complete symbolic backtrace in case of error.

In addition to the binding information, additional information is associated with each function call: access information indicating the path to search the basic frames for variable bindings, control information, and temporary results are also stored on the stack in a block called the frame extension. The interpreter also stores information about partially evaluated expressions as described in the Stack and Interpreter section of Chapter 11.

Spaghetti Stack

The Bobrow/Wegbreit paper, "A Model and Stack Implementation for Multiple Environments" (*Communications of the ACM,* Vol. 16, 10, October 1973.), describes an access and control mechanism more general than a simple linear stack. The access and control mechanism used by Interlisp is a slightly modified version of the one proposed by Bobrow and Wegbreit. This mechanism is called the "spaghetti stack."

The spaghetti system presents the access and control stack as a data structure composed of "frames." The functions described below operate on this structure. These primitives allow user functions to manipulate the stack in a machine independent way. Backtracking, coroutines, and more sophisticated control schemes can be easily implemented with these primitives.

The evaluation of a function requires the allocation of storage to hold the values of its local variables during the computation. In addition to variable bindings, an activation of a function requires a return link (indicating where control is to go after the completion of the computation) and room for temporaries needed during the computation. In the spaghetti system, one "stack" is used for storing all this information, but it is best to view this stack as a tree of linked objects called frame extensions (or simply frames).

A frame extension is a variable sized block of storage containing a frame name, a pointer to some variable bindings (the `BLINK`), and two pointers to other frame extensions (the `ALINK` and `CLINK`). In addition to these components, a frame extension contains other information (such as temporaries and reference counts) that does not interest us here.

The block of storage holding the variable bindings is called a basic frame. A basic frame is essentially

an array of pairs, each of which contains a variable name and its value. The reason frame extensions point to basic frames (rather than just having them "built in") is so that two frame extensions can share a common basic frame. This allows two processes to communicate via shared variable bindings.

The chain of frame extensions which can be reached via the successive ALINKs from a given frame is called the "access chain" of the frame. The first frame in the access chain is the starting frame. The chain through successive CLINKs is called the "control chain".

A frame extension completely specifies the variable bindings and control information necessary for the evaluation of a function. Whenever a function (or in fact, any form which generally binds local variables) is evaluated, it is associated with some frame extension.

In the beginning there is precisely one frame extension in existence. This is the frame in which the top-level call to the interpreter is being run. This frame is called the "top-level" frame.

Since precisely one function is being executed at any instant, exactly one frame is distinguished as having the "control bubble" in it. This frame is called the active frame. Initially, the top-level frame is the active frame. If the computation in the active frame invokes another function, a new basic frame and frame extension are built. The frame name of this basic frame will be the name of the function being called. The ALINK, BLINK, and CLINK of the new frame all depend on precisely how the function is invoked. The new function is then run in this new frame by passing control to that frame, i.e., it is made the active frame.

Once the active computation has been completed, control normally returns to the frame pointed to by the CLINK of the active frame. That is, the frame in the CLINK becomes the active frame.

In most cases, the storage associated with the basic frame and frame extension just abandoned can be reclaimed. However, it is possible to obtain a pointer to a frame extension and to "hold on" to this frame even after it has been exited. This pointer can be used later to run another computation in that environment, or even "continue" the exited computation.

A separate data type, called a stack pointer, is used for this purpose. A stack pointer is just a cell that literally points to a frame extension. Stack pointers print as #ADR/FRAMENAME, e.g., #1,13636/COND. Stack pointers are returned by many of the stack manipulating functions described below. Except for certain abbreviations (such as "the frame with such-and-such a name"), stack pointers are the only way you can reference a frame extension. As long as you have a stack pointer which references a frame extension, that frame extension (and all those that can be reached from it) will not be garbage collected.

Two stack pointers referencing the same frame extension are *not* necessarily EQ, i.e., (EQ (STKPOS 'FOO) (STKPOS 'FOO)) = NIL. However, EQP can be used to test if two different stack pointers reference the same frame extension (see the Equality Predicates section of Chapter 9).

It is possible to evaluate a form with respect to an access chain other than the current one by using a stack pointer to refer to the head of the access chain desired. Note, however, that this can be very expensive when using a shallow binding scheme such as that in Interlisp-10. When evaluating the form, since all references to variables under the shallow binding scheme go through the variable's value cell, the values in the value cells must be adjusted to reflect the values appropriate to the desired access chain. This is done by changing all the bindings on the current access chain (all the name-value pairs) so that they contain the value current at the time of the call. Then along the new access path, all bindings are made to contain the previous value of the variable, and the current value is placed in the value cell. For that part of the access path which is shared by the old and new chain, no work has to be done. The context switching time, i.e. the overhead in switching from the current, active, access

chain to another one, is directly proportional to the size of the two branches that are not shared between the access contexts. This cost should be remembered in using generators and coroutines (see the Generators section below).

Stack Functions

In the descriptions of the stack functions below, when we refer to an argument as a stack descriptor, we mean that it is one of the following:

A stack pointer	An object that points to a frame on the stack. Stack pointers are returned by many of the stack manipulating functions described below.
`NIL`	Specifies the active frame; that is, the frame of the stack function itself.
`T`	Specifies the top-level frame.
A symbol	Specifies the first frame (along the control chain from the active frame) that has the frame name `LITATOM`. Equivalent to `(STKPOS LITATOM -1)`.
A list of symbols	Specifies the first frame (along the control chain from the active frame) whose frame name is included in the list.
A number *N*	Specifies the *N*th frame back from the active frame. If *N* is negative, the control chain is followed, otherwise the access chain is followed. Equivalent to `(STKNTH N)`.

In the stack functions described below, the following errors can occur: The error `Illegal stack arg` occurs when a stack descriptor is expected and the supplied argument is either not a legal stack descriptor (i.e., not a stack pointer, symbol, or number), or is a symbol or number for which there is no corresponding stack frame, e.g., `(STKNTH -1 'FOO)` where there is no frame named `FOO` in the active control chain or `(STKNTH -10 'EVALQT)`. The error `Stack pointer has been released` occurs whenever a released stack pointer is supplied as a stack descriptor argument for any purpose other than as a stack pointer to re-use.

Note: The creation of a single stack pointer can result in the retention of a large amount of stack space. Therefore, one should try to release stack pointers when they are no longer needed (see the Releasing and Reusing Stack Pointers section below).

In Lisp there is a fixed ammount of space allocated for the stack. When most of this space is exhausted, the `STACK OVERFLOW` error occurs and the debugger will be invoked. You will still have a little room on the stack to use inside the debugger. If you use up this last little bit of stack you will encounter a "hard" stack overflow. A "hard" stack overflow will put you into URaid (see the documentation on URaid).

Searching the Stack

(**STKPOS** *FRAMENAME N POS OLDPOS*) [Function]

Returns a stack pointer to the *N*th frame with frame name *FRAMENAME*. The search begins with (and includes) the frame specified by the stack descriptor *POS*. The search proceeds along the control chain from *POS* if *N* is negative, or along the access chain if *N* is positive. If *N* is `NIL`, -1 is used. Returns a stack pointer to the frame if such a frame exists, otherwise returns `NIL`. If *OLDPOS* is supplied and is a stack pointer, it is reused. If

OLDPOS is supplied and is a stack pointer and STKPOS returns NIL, *OLDPOS* is released. If *OLDPOS* is not a stack pointer it is ignored.

(**STKNTH** *N POS OLDPOS*) [Function]

Returns a stack pointer to the *N*th frame back from the frame specified by the stack descriptor *POS*. If *N* is negative, the control chain from *POS* is followed. If *N* is positive the access chain is followed. If *N* equals 0, STKNTH returns a stack pointer to *POS* (this provides a way to copy a stack pointer). Returns NIL if there are fewer than *N* frames in the appropriate chain. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is not a stack pointer it is ignored.

Note: (STKNTH 0) causes an error, Illegal stack arg; it is not possible to create a stack pointer to the active frame.

(**STKNAME** *POS*) [Function]

Returns the frame name of the frame specified by the stack descriptor *POS*.

(**SETSTKNAME** *POS NAME*) [Function]

Changes the frame name of the frame specified by *POS* to be *NAME*. Returns *NAME*.

(**STKNTHNAME** *N POS*) [Function]

Returns the frame name of the *N*th frame back from *POS*. Equivalent to (STKNAME (STKNTH *N POS*)) but avoids creation of a stack pointer.

In summary, STKPOS converts function names to stack pointers, STKNTH converts numbers to stack pointers, STKNAME converts stack pointers to function names, and STKNTHNAME converts numbers to function names.

Variable Bindings in Stack Frames

The following functions are used for accessing and changing bindings. Some of functions take an argument, *N*, which specifies a particular binding in the basic frame. If *N* is a literal atom, it is assumed to be the name of a variable bound in the basic frame. If *N* is a number, it is assumed to reference the *N*th binding in the basic frame. The first binding is 1. If the basic frame contains no binding with the given name or if the number is too large or too small, the error Illegal arg occurs.

(**STKSCAN** *VAR IPOS OPOS*) [Function]

Searches beginning at *IPOS* for a frame in which a variable named *VAR* is bound. The search follows the access chain. Returns a stack pointer to the frame if found, otherwise returns NIL. If *OPOS* is a stack pointer it is reused, otherwise it is ignored.

(**FRAMESCAN** *ATOM POS*) [Function]

Returns the relative position of the binding of *ATOM* in the basic frame of *POS*. Returns NIL if *ATOM* is not found.

(**STKARG** *N POS —*) [Function]

Returns the value of the binding specified by *N* in the basic frame of the frame specified by

the stack descriptor *POS*. *N* can be a literal atom or number.

(**STKARGNAME** *N POS*) [Function]

Returns the name of the binding specified by *N*, in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(**SETSTKARG** *N POS VAL*) [Function]

Sets the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *VAL*.

(**SETSTKARGNAME** *N POS NAME*) [Function]

Sets the variable name to *NAME* of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *NAME*. This function does not work for interpreted frames.

(**STKNARGS** *POS —*) [Function]

Returns the number of arguments bound in the basic frame of the frame specified by the stack descriptor *POS*.

(**VARIABLES** *POS*) [Function]

Returns a list of the variables bound at *POS*.

(**STKARGS** *POS —*) [Function]

Returns a list of the values of the variables bound at *POS*.

Evaluating Expressions in Stack Frames

The following functions are used to evaluate an expression in a different environment:

(**ENVEVAL** *FORM APOS CPOS AFLG CFLG*) [Function]

Evaluates *FORM* in the environment specified by *APOS* and *CPOS*. That is, a new active frame is created with the frame specified by the stack descriptor *APOS* as its ALINK, and the frame specified by the stack descriptor *CPOS* as its CLINK. Then *FORM* is evaluated. If *AFLG* is not NIL, and *APOS* is a stack pointer, then *APOS* will be released. Similarly, if *CFLG* is not NIL, and *CPOS* is a stack pointer, then *CPOS* will be released.

(**ENVAPPLY** *FN ARGS APOS CPOS AFLG CFLG*) [Function]

APPLYs *FN* to *ARGS* in the environment specified by *APOS* and *CPOS*. *AFLG* and *CFLG* have the same interpretation as with ENVEVAL.

(**EVALV** *VAR POS RELFLG*) [Function]

Evaluates *VAR*, where *VAR* is assumed to be a symbol, in the access environment specifed by the stack descriptor *POS*. If *VAR* is unbound, EVALV returns NOBIND and does not generate an error. If *RELFLG* is non-NIL and *POS* is a stack pointer, it will be released after the variable is looked up. While EVALV could be defined as (ENVEVAL *VAR POS* NIL *RELFLG*) it is in fact somewhat faster.

(**STKEVAL** *POS FORM FLG* —) [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*. If *FLG* is not NIL and *POS* is a stack pointer, releases *POS*. The definition of STKEVAL is (ENVEVAL *FORM POS* NIL *FLG*).

(**STKAPPLY** *POS FN ARGS FLG*) [Function]

Like STKEVAL but applies *FN* to *ARGS*.

Altering Flow of Control

The following functions are used to alter the normal flow of control, possibly jumping to a different frame on the stack. RETEVAL and RETAPPLY allow evaluating an expression in the specified environment first.

(**RETFROM** *POS VAL FLG*) [Function]

Return from the frame specified by the stack descriptor *POS*, with the value *VAL*. If *FLG* is not NIL, and *POS* is a stack pointer, then *POS* is released. An attempt to RETFROM the top level (e.g., (RETFROM T)) causes an error, Illegal stack arg. RETFROM can be written in terms of ENVEVAL as follows:

```
(RETFROM
    (LAMBDA (POS VAL FLG)
       (ENVEVAL (LIST 'QUOTE VAL)
                NIL
                (if (STKNTH -1 POS
                       (if FLG then POS))
                else (ERRORX (LIST 19 POS)))
                NIL
                T)))
```

(**RETTO** *POS VAL FLG*) [Function]

Like RETFROM, but returns *to* the frame specified by *POS*.

(**RETEVAL** *POS FORM FLG* —) [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*, and then returns from *POS* with that value. If *FLG* is not NIL and *POS* is a stack pointer, then *POS* is released. The definition of RETEVAL is equivalent to (ENVEVAL *FORM POS* (STKNTH -1 *POS*) *FLG* T), but RETEVAL does not create a stack pointer.

(**RETAPPLY** *POS FN ARGS FLG*) [Function]

Like RETEVAL but applies *FN* to *ARGS*.

Releasing and Reusing Stack Pointers

The following functions and variables are used for manipulating stack pointers:

(**STACKP** *X*) [Function]

Returns *X* if *X* is a stack pointer, otherwise returns NIL.

(**RELSTK** *POS*) [Function]

Release the stack pointer *POS* (see below). If *POS* is not a stack pointer, does nothing. Returns *POS*.

(**RELSTKP** *X*) [Function]

Returns T is *X* is a released stack pointer, NIL otherwise.

(**CLEARSTK** *FLG*) [Function]

If *FLG* is T, returns a list of all the active (unreleased) stack pointers. If *FLG* is NIL, this call is a no-op. The abillity to clear all stack pointers is inconsistent with the modularity implicit in a multi processing environment.

CLEARSTKLST [Variable]

A variable used by the top-level executive. Every time the top-level executive is re-entered (e.g., following errors, or Control-D), CLEARSTKLST is checked. If its value is T, all active stack pointers are released using CLEARSTK. If its value is a list, then all stack pointers on that list are released. If its value is NIL, nothing is released. CLEARSTKLST is initially T.

NOCLEARSTKLST [Variable]

A variable used by the top-level executive. If CLEARSTKLST is T (see above) all active stack pointers *except* those on NOCLEARSTKLST are released. NOCLEARSTKLST is initially NIL.

Creating a single stack pointer can cause the retention of a large amount of stack space. Furthermore, this space will not be freed until the next garbage collection, *even if the stack pointer is no longer being used*, unless the stack pointer is explicitly released or reused. If there is sufficient amount of stack space tied up in this fashion, a STACK OVERFLOW condition can occur, even in the simplest of computations. For this reason, you should consider releasing a stack pointer when the environment referenced by the stack pointer is no longer needed.

The effects of releasing a stack pointer are:

1. The link between the stack pointer and the stack is broken by setting the contents of the stack pointer to the "released mark". A released stack pointer prints as #*ADR*/#*0*.

2. If this stack pointer was the last remaining reference to a frame extension; that is, if no other stack pointer references the frame extension and the extension is not contained in the active control or access chain, then the extension may be reclaimed, and is reclaimed immediately. The process repeats for the access and control chains of the reclaimed extension so that all stack space that was reachable only from the released stack pointer is reclaimed.

A stack pointer may be released using the function RELSTK, but there are some cases for which RELSTK is not sufficient. For example, if a function contains a call to RETFROM in which a stack pointer was used to specify where to return to, it would not be possible to simultaneously release the stack pointer. (A RELSTK appearing in the function following the call to RETFROM would not be executed!) To permit release of a stack pointer in this situation, the stack functions that relinquish control have optional flag arguments to denote whether or not a stack pointer is to be released (*AFLG* and *CFLG*). Note that in this case releasing the stack pointer will *not* cause the stack space to be

reclaimed immediately because the frame referenced by the stack pointer will have become part of the active environment.

Another way to avoid creating new stack pointers is to *reuse* stack pointers that are no longer needed. The stack functions that create stack pointers (STKPOS, STKNTH, and STKSCAN) have an optional argument that is a stack pointer to reuse. When a stack pointer is reused, two things happen. First the stack pointer is released (see above). Then the pointer to the new frame extension is deposited in the stack pointer. The old stack pointer (with its new contents) is returned as the value of the function. Note that the reused stack pointer will be released even if the function does not find the specified frame.

Even if stack pointers are explicitly being released, *creating* many stack pointers can cause a garbage collection of stack pointer space. Thus, if your application requires creating many stack pointers, you definitely should take advantage of reusing stack pointers.

Backtrace Functions

The following functions perform a "backtrace," printing information about every frame on the stack. Arguments allow only backtracing a selected range of the stack, skipping selected frames, and printing different amounts of information about each frame.

(**BACKTRACE** *IPOS EPOS FLAGS FILE PRINTFN*) [Function]

Performs a backtrace beginning at the frame specified by the stack descriptor *IPOS*, and ending with the frame specified by the stack descriptor *EPOS*. *FLAGS* is a number in which the options of the BACKTRACE are encoded. If a bit is set, the corresponding information is included in the backtrace.

1Q - print arguments of non-SUBRs

2Q - print temporaries of the interpreter

4Q - print SUBR arguments and local variables

10Q - omit printing of UNTRACE: and function names

20Q - follow access chain instead of control chain

40Q - print temporaries, i.e. the blips (see the stack and interpreter section below)

For example: If *FLAGS* = 47Q, everything is printed. If *FLAGS* = 21Q, follows the access chain, prints arguments.

FILE is the file that the backtrace is printed to. *FILE* must be open. *PRINTFN* is used when printing the values of variables, temporaries, blips, etc. *PRINTFN* = NIL defaults to PRINT.

(**BAKTRACE** *IPOS EPOS SKIPFNS FLAGS FILE*) [Function]

Prints a backtrace from *IPOS* to *EPOS* onto *FILE*. *FLAGS* specifies the options of the backtrace, e.g., do/don't print arguments, do/don't print temporaries of the interpreter, etc., and is the same as for BACKTRACE.

SKIPFNS is a list of functions. As BAKTRACE scans down the stack, the stack name of each frame is passed to each function in *SKIPFNS*, and if any of them returnS non-NIL, *POS* is skipped (including all variables).

BAKTRACE collapses the sequence of several function calls corresponding to a call to a

system package into a single "function" using BAKTRACELST as described below. For example, any call to the editor is printed as **EDITOR**, a break is printed as **BREAK**, etc.

BAKTRACE is used by the BT, BTV, BTV+, BTV*, and BTV! break commands, with *FLAGS* = 0, 1, 5, 7, and 47Q respectively.

If SYSPRETTYFLG = T, the values arguments and local variables will be prettyprinted.

BAKTRACELST [Variable]

Used to tell BAKTRACE (therefore, the BT, BTV, etc. commands) to abbreviate various sequences of function calls on the stack by a single key, e.g. **BREAK**, **EDITOR**, etc.

Each entry on BAKTRACELST is a list of the form (*FRAMENAME KEY . PATTERN*) or (*FRAMENAME* (*KEY*$_1$. *PATTERN*$_1$) ... (*KEY*$_N$. *PATTERN*$_N$)), where a pattern is a list of elements that are either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns, e.g. (PROGN **BREAK** EVAL ((ERRORSET BREAK1A BREAK1) (BREAK1)))

BAKTRACE operates by scanning up the stack and, at each point, comparing the current frame name, with the frame names on BAKTRACELST, i.e. it does an ASSOC. If the frame name does appear, BAKTRACE attempts to match the stack as of that point with (one of) the patterns. If the match is successful, BAKTRACE prints the corresponding key, and continues with where the match left off. If the frame name does not appear, or the match fails, BAKTRACE simply prints the frame name and continues with the next higher frame (unless the *SKIPFNS* applied to the frame name are non-NIL as described above).

Matching is performed by comparing symbols in the pattern with the current frame name, and matching lists as patterns, i.e. sequences of function calls, always working up the stack. For example, either of the sequence of function calls "... BREAK1 BREAK1A ERRORSET EVAL PROGN ..." or "... BREAK1 EVAL PROGN ..." would match with the sample entry given above, causing **BREAK** to be printed.

Special features:

- The symbol & can be used to match any frame.
- The pattern "–" can be used to match nothing. – is useful for specifying an optional match, e.g. the example above could also have been written as (PROGN **BREAK** EVAL ((ERRORSET BREAK1A) –) BREAK1).
- It is not necessary to provide in the pattern for matching dummy frames, i.e. frames for which DUMMYFRAMEP (see below) is true. When working on a match, the matcher automatically skips over these frames when they do not match.
- If a match succeeds and the *KEY* is NIL, nothing is printed. For example, (*PROG*LAM NIL EVALA *ENV). This sequence will occur following an error which then causes a break if some of the function's arguments are LOCALVARS.

Other Stack Functions

(**DUMMYFRAMEP** *POS*) [Function]

Returns T if you never wrote a call to the function at *POS*, e.g. in Interlisp-10, DUMMYFRAMEP is T for *PROG*LAM, *ENV*, and FOOBLOCK frames (see the Block Compiling section of Chapter 18).

REALFRAMEP and REALSTKNTH can be used to write functions which manipulate the stack and work on either interpreted or compiled code:

(**REALFRAMEP** *POS INTERPFLG*) [Function]

Returns *POS* if *POS* is a "real" frame, i.e. if *POS* is not a dummy frame and *POS* is a frame that does not disappear when compiled (such as COND); otherwise NIL. If *INTERPFLG* = T, returns T if *POS* is not a dummy frame. For example, if (STKNAME *POS*) = COND, (REALFRAMEP *POS*) is NIL, but (REALFRAMEP *POS* T) is T.

(**REALSTKNTH** *N POS INTERPFLG OLDPOS*) [Function]

Returns a stack pointer to the *N*th (or −*N*th) frames for which (REALFRAMEP *POS INTERPFLG*) is *POS*.

(**MAPDL** *MAPDLFN MAPDLPOS*) [Function]

Starts at *MAPDLPOS* and applies the function *MAPDLFN* to two arguments (the frame name and a stack pointer to the frame), for each frame until the top of the stack is reached. Returns NIL. For example,

```
[MAPDL (FUNCTION (LAMBDA (X POS)
   (if (IGREATERP (STKNARGS POS) 2) then (PRINT X)]
```

will print all functions of more than two arguments.

(**SEARCHPDL** *SRCHFN SRCHPOS*) [Function]

Like MAPDL, but searches the stack starting at position *SRCHPOS* until it finds a frame for which *SRCHFN*, a function of two arguments applied to the *name* of the frame and the frame itself, is not NIL. Returns (*NAME* . *FRAME*) if such a frame is found, otherwise NIL.

The Stack and the Interpreter

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function FACT (intentionally faulty):

```
(FACT
   [LAMBDA (N)
          (COND
            ((ZEROP N)
              L)
            (T (ITIMES N (FACT (SUB1 N])
```

In evaluating the form (FACT 1), as soon as FACT is entered, the interpreter begins evaluating the implicit PROGN following the LAMBDA. The first function entered in this process is COND. COND begins to process its list of clauses. After calling ZEROP and getting a NIL value, COND proceeds to the next

clause and evaluates T. Since T is true, the evaluation of the implicit PROGN that is the consequent of the T clause is begun. This requires calling the function ITIMES. However before ITIMES can be called, its arguments must be evaluated. The first argument is evaluated by retrieving the current binding of N from its value cell; the second involves a recursive call to FACT, and another implicit PROGN, etc.

At each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below (from Interlisp-10) illustrates this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```
_ FACT(1)
 u.b.a. L {in FACT} in ((ZEROP NO L)
 (L broken)
 :BTV!
    *TAIL* (L)
    *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
 COND
    *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
    *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
    N 0
 FACT
    *FORM* (FACT (SUB1 N))
    *FN* ITIMES
    *TAIL* ((FACT (SUB1 N)))
    *ARGVAL* 1
    *FORM* (ITIMES N (FACT (SUB1 N)))
    *TAIL* ((ITIMES N (FACT (SUB1 N))))
    *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
 COND
    *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
    *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
    N 1
 FACT
 **TOP**
```

Internal calls to EVAL, e.g., from COND and the interpreter, are marked on the push-down list by a special mark or blip which the backtrace prints as *FORM*. The genealogy of *FORM*'s is thus a history of the computation. Other temporary information stored on the stack by the interpreter includes the tail of a partially evaluated implicit PROGN (e.g., a cond clause or lambda expression) and the tail of a partially evaluated form (i.e., those arguments not yet evaluated), both indicated on the backtrace by *TAIL*, the values of arguments that have already been evaluated, indicated by *ARGVAL*, and the names of functions waiting to be called, indicated by *FN*. *ARG1, ..., *ARGn are used by the backtrace to indicate the (unnamed) arguments to SUBRs.

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated (except for nlambda functions, of course). Also note that the *ARG1, *FORM*, *TAIL*, etc. "bindings" comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the *ARG1 binding, the COND would continue interpreting the new binding as a list of COND clauses. Similarly, if the *ARGVAL* binding were changed, the new value would be given to ITIMES as its first argument after its second argument had been evaluated, and ITIMES was actually called.

FORM, *TAIL*, *ARGVAL*, etc., do not actually appear as variables on the stack, i.e., evaluating *FORM* or calling STKSCAN to search for it will not work. However, the functions BLIPVAL, SETBLIPVAL, and BLIPSCAN described below are available for accessing these internal blips. These

functions currently know about four different types of blips:

FN	The name of a function about to be called
ARGVAL	An argument for a function about to be called
FORM	A form in the process of evaluation
TAIL	The tail of a COND clause, implicit PROGN, PROG, etc.

(**BLIPVAL** *BLIPTYP IPOS FLG*) [Function]

Returns the value of the specified blip of type *BLIPTYP*. If *FLG* is a number N, finds the Nth blip of the desired type, searching the control chain beginning at the frame specified by the stack descriptor *IPOS*. If *FLG* is NIL, 1 is used. If *FLG* is T, returns the number of blips of the specified type at *IPOS*.

(**SETBLIPVAL** *BLIPTYP IPOS N VAL*) [Function]

Sets the value of the specified blip of type *BLIPTYP*. Searches for the *N*th blip of the desired type, beginning with the frame specified by the stack descriptor *IPOS*, and following the control chain.

(**BLIPSCAN** *BLIPTYP IPOS*) [Function]

Returns a stack pointer to the frame in which a blip of type *BLIPTYP* is located. Search begins at the frame specified by the stack descriptor *IPOS* and follows the control chain.

Generators

A *generator* is like a subroutine except that it retains information about previous times it has been called. Some of this state may be data (for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if LISTGEN is defined by:

```
(DEFINEQ (LISTGEN (L)
         (if L then (PRODUCE (CAR L))
                    (LISTGEN (CDR L))))
```

we can use the function GENERATOR (described below) to create a generator that uses LISTGEN to produce the elements of a list one at a time, e.g.,

```
(SETQ GR (GENERATOR (LISTGEN '(A B C))))
```

creates a generator, which can be called by

```
(GENERATE GR)
```

to produce as values on successive calls, A, B, C. When GENERATE (not GENERATOR) is called the first time, it simply starts evaluating (LISTGEN '(A B C)). PRODUCE gets called from LISTGEN, and pops back up to GENERATE with the indicated value after saving the state. When GENERATE gets called again, it continues from where the last PRODUCE left off. This process continues until finally LISTGEN completes and returns a value (it doesn't matter what it is). GENERATE then returns GR itself as its value, so that the program that called GENERATE can tell that it is finished, i.e., there are no more values to be generated.

(**GENERATOR** *FORM COMVAR*) [NLambda Function]

 An nlambda function that creates a generator which uses *FORM* to compute values.

GENERATOR returns a *generator handle* which is represented by a dotted pair of stack pointers.

COMVAR is optional. If its value (EVAL of) is a generator handle, the list structure and stack pointers will be reused. Otherwise, a new generator handle will be constructed.

GENERATOR compiles open.

(**PRODUCE** *VAL*) [Function]

Used from within a generator to return *VAL* as the value of the corresponding call to GENERATE.

(**GENERATE** *HANDLE VAL*) [Function]

Restarts the generator represented by *HANDLE*. *VAL* is returned as the value of the PRODUCE which last suspended the operation of the generator. When the generator runs out of values, GENERATE returns *HANDLE* itself.

Examples:

The following function will go down recursively through a list structure and produce the atoms in the list structure one at a time.

```
(DEFINEQ (LEAVESG (L)
        (if (ATOM L)
         then (PRODUCE L)
         else (LEAVESG (CAR L))
              (if (CDR L)
               then (LEAVESG (CDR L)]
```

The following function prints each of these atoms as it appears. It illustrates how a loop can be set up to use a generator.

```
(DEFINEQ (PLEAVESG1 (L)
        (PROG (X LHANDLE)
          (SETQ LHANDLE (GENERATOR (LEAVESG L)))
          LP (SETQ X (GENERATE LHANDLE))
            (if (EQ X LHANDLE)
             then (RETURN NIL))
              (PRINT X)
                 (GO LP))]
```

The loop terminates when the value of the generator is EQ to the dotted pair which is the value produced by the call to GENERATOR. A CLISP iterative operator, OUTOF, is provided which makes it much easier to write the loop in PLEAVESG1. OUTOF (or outof) can precede a form which is to be used as a generator. On each iteration, the iteration variable will be set to successive values returned by the generator; the loop will be terminated automatically when the generator runs out. Therefore, the following is equivalent to the above program PLEAVESG1:

```
(DEFINEQ (PLEAVESG2 (L) (for X outof (LEAVESG L) do (PRINT X))]
```

Here is another example; the following form will print the first N atoms.

```
(for X outof (MAPATOMS (FUNCTION PRODUCE)) as I from 1 to N do (PRINT X))
```

Coroutines

This package provides facilities for the creation and use of fully general coroutine structures. It uses a

stack pointer to preserve the state of a coroutine, and allows arbitrary switching between *N* different coroutines, rather than just a call to a generator and return. This package is slightly more efficient than the generator package described above, and allows more flexibility on specification of what to do when a coroutine terminates.

(**COROUTINE** *CALLPTR COROUTPTR COROUTFORM ENDFORM*) [NLambda Function]

This nlambda function is used to create a coroutine and initialize the linkage. *CALLPTR* and *COROUTPTR* are the names of two variables, which will be set to appropriate stack pointers. If the values of *CALLPTR* or *COROUTPTR* are already stack pointers, the stack pointers will be reused. *COROUTFORM* is the form which is evaluated to start the coroutine; *ENDFORM* is a form to be evaluated if *COROUTFORM* actually returns when it runs out of values.

COROUTINE compiles open.

(**RESUME** *FROMPTR TOPTR VAL*) [Function]

Used to transfer control from one coroutine to another. *FROMPTR* should be the stack pointer for the current coroutine, which will be smashed to preserve the current state. *TOPTR* should be the stack pointer which has preserved the state of the coroutine to be transferred to, and *VAL* is the value that is to be returned to the latter coroutine as the value of the RESUME which suspended the operation of that coroutine.

For example, the following is the way one might write the LEAVES program using the coroutine package:

```
(DEFINEQ (LEAVESC (L COROUTPTR CALLPTR)
   (if (ATOM L)
    then (RESUME COROUTPTR CALLPTR L)
    else (LEAVESC (CAR L) COROUTPTR CALLPTR)
         (if (CDR L) then (LEAVESC (CDR L) COROUTPTR CALLPTR)))))]
```

A function PLEAVESC which uses LEAVESC can be defined as follows:

```
(DEFINEQ (PLEAVESC (L)
  (bind PLHANDLE LHANDLE
   first (COROUTINE PLHANDLE LHANDLE
                    (LEAVESC L LHANDLE PLHANDLE)
                    (RETFROM 'PLEAVESC))
   do (PRINT (RESUME PLHANDLE LHANDLE)))))]
```

By RESUMEing LEAVESC repeatedly, this function will print all the leaves of list L and then return out of PLEAVESC via the RETFROM. The RETFROM is necessary to break out of the non-terminating do-loop. This was done to illustrate the additional flexibility allowed through the use of *ENDFORM*.

We use two coroutines working on two trees in the example EQLEAVES, defined below. EQLEAVES tests to see whether two trees have the same leaf set in the same order, e.g., (EQLEAVES '(A B C) '(A B (C))) is true.

```
(DEFINEQ (EQLEAVES (L1 L2)
    (bind LHANDLE1 LHANDLE2 PE EL1 EL2
     first (COROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE) 'NO-MORE)
           (COROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE) 'NO-MORE)
     do (SETQ EL1 (RESUME PE LHANDLE1))
        (SETQ EL2 (RESUME PE LHANDLE2))
        (if (NEQ EL1 EL2)
```

```
    then (RETURN NIL))
repeatuntil (EQ EL1 'NO-MORE)
finally (RETURN T)))]
```

Possibilities Lists

A possibilities list is the interface between a generator and a consumer. The possibilities list is initialized by a call to POSSIBILITIES, and elements are obtained from it by using TRYNEXT. By using the spaghetti stack to maintain separate environments, this package allows a regime in which a generator can put a few items in a possibilities list, suspend itself until they have been consumed, and be subsequently aroused and generate some more.

(**POSSIBILITIES** *FORM*) [NLambda Function]

This nlambda function is used for the initial creation of a possibilities list. *FORM* will be evaluated to create the list. It should use the functions NOTE and AU-REVOIR described below to generate possibilities. Normally, one would set some variable to the possibilities list which is returned, so it can be used later, e.g.:

```
(SETQ PLIST (POSSIBILITIES (GENERFN V1 V2))).
```

POSSIBILITIES compiles open.

(**NOTE** *VAL LSTFLG*) [Function]

Used within a generator to put items on the possibilities list being generated. If *LSTFLG* is equal to NIL, *VAL* is treated as a single item. If *LSTFLG* is non-NIL, then the list *VAL* is NCONCed on the end of the possibilities list. Note that it is perfectly reasonable to create a possibilities list using a second generator, and NOTE that list as possibilities for the current generator with *LSTFLG* equal to T. The lower generator will be resumed at the appropriate point.

(**AU-REVOIR** *VAL*) [NoSpread Function]

Puts *VAL* on the possibilities list if it is given, and then suspends the generator and returns to the consumer in such a fashion that control will return to the generator at the AU-REVOIR if the consumer exhausts the possibilities list.

NIL is not put on the possibilities list unless it is explicitly given as an argument to AU-REVOIR, i.e., (AU-REVOIR) and (AU-REVOIR NIL) are *not* the same. AU-REVOIR and ADIEU are lambda nospreads to enable them to distinguish these two cases.

(**ADIEU** *VAL*) [NoSpread Function]

Like AU-REVOIR but releases the generator instead of suspending it.

(**TRYNEXT** *PLST ENDFORM VAL*) [NLambda Function]

This nlambda function allows a consumer to use a possibilities list. It removes the first item from the possibilities list named by *PLST* (i.e. *PLST* must be an atom whose value is a possiblities list), and returns that item, provided it is not a generator handle. If a generator handle is encountered, the generator is reawakened. When it returns a possibilities list, this list is added to the front of the current list. When a call to TRYNEXT causes a generator to be awakened, *VAL* is returned as the value of the AU-REVOIR which put that generator to sleep. If *PLST* is empty, it evaluates *ENDFORM* in the caller's

environment.

TRYNEXT compiles open.

(**CLEANPOSLST** *PLST*) [Function]

This function is provided to release any stack pointers which may be left in the *PLST* which was not used to exhaustion.

For example, FIB is a generator for fibonnaci numbers. It starts out by NOTEing its two arguments, then suspends itself. Thereafter, on being re-awakened, it will NOTE two more terms in the series and suspends again. PRINTFIB uses FIB to print the first N fibonacci numbers.

```
(DEFINEQ (FIB (F1 F2)
  (do (NOTE F1)
      (NOTE F2)
      (SETQ F1 (IPLUS F1 F2))
      (SETQ F2 (IPLUS F1 F2))
      (AU-REVOIR)]
```

Note that this AU-REVOIR just suspends the generator and adds nothing to the possibilities list except the generator.

```
(DEFINEQ (PRINTFIB (N)
    (PROG ((FL (POSSIBILITIES (FIB 0 1))))
          (RPTQ N (PRINT  (TRYNEXT FL)))
          (CLEANPOSLST FL)]
```

Note that FIB itself will never terminate.

[This page intentionally left blank]