SECTION 12

Closures

12.1 INTRODUCTION

A closure is a functional object with some extra information attached to it. In simplest terms, it is a function coupled with some representation of the environment in which it was created. The Explorer system supports two kinds of closures: Dynamic Closures which capture some part of the dynamic variable binding environment in which they are created, and Lexical Closures which capture the surrounding lexical (textual) variable environment in which the closure function was written.

Each of these closures is a separate primitive functional object having a dedicated data type and a unique storage format for its environment structure. Each also implies a different set of support operations which must occur when the closure is funcalled. This section covers these topics.

12.2 DYNAMIC CLOSURES

Conceptually, a dynamic closure object is a function and a remembered binding environment. The remembered binding environment is necessary to resolve possible binding conflicts caused by free references to variables in functions passed as arguments (funargs). The implementation of dynamic closures on the Explorer system is closely tied with the variable binding method chosen for the system.

Some implementations of Lisp use deep binding, where the binding stack can be thought of as an a-list of symbol/value pairs. In this case accessing a variable requires an ASSOC, and takes time proportional to the number of bindings on the a-list. In a deep binding implementation, there is one global binding stack (or one per process, in multi-processing systems). Dynamic closures in such a system can "close over" the entire environment simply by saving a pointer to the current global binding a-list when the dynamic closure is created.

Explorer system Lisp, like MacLisp, uses shallow binding. The goal of shallow binding is to keep the time needed to access a symbols value to a small constant: typically only that of a single memory reference. Conceptually, shallow binding can be thought of as a scheme where there is a stack associated with each symbol which contains a history of the symbol's bindings. The top of the symbol's binding stack (which can be accessed quickly) always contains its current binding.

Shallow binding on the Explorer is a slight variation on this conceptual model. Each symbol's value cell (called the symbol's internal value cell since it is the internal physical location within the symbol data structure itself) always contains the current binding for the function being run. Previous bindings are saved on the binding stack (the Special PDL or SPECPDL) by the BIND primitive and are restored when the binding construct is exited. BIND does this by saving the actual contents of the symbol's internal value cell on the Special PDL along with a pointer (of type DTP-LOCATIVE) to the internal value cell.

Dynamic closure binding is a special kind of binding which may be shared; that is, other functions and dynamic closures in the same binding environment as the CLOSURE function can access and change the dynamic closure's closed over variables. This sharing is accomplished by using a new data type called DTP-EXTERNAL-VALUE-CELL-POINTER (EVCP). It can be thought of as a kind of "environment pointer," referring to the symbol's value when it was closure-bound.

Since bindings are associated with the symbol and not on a global a-list, dynamic closure binding on the Explorer system is on a per-symbol basis. The function CLOSURE takes two arguments: the first argument is a list of symbols (the symbols whose bindings are to be saved), and the second is a function object (such as a LAMBDA expression, or a compiled-code object). First, CLOSURE CDRs down its first argument, assuring that each of the symbols has an external value cell. Whenever it finds one which doesn't, it allocates a word from free storage, places the contents of the symbol's internal value cell into the word, and replaces the internal value cell with a DTP-EXTERNAL-VALUE-CELL-POINTER to the word. Then, CLOSURE allocates a block of 2*N+1 words of storage, where N is the length of CLOSURE's first argument. In the first word of the block, CLOSURE stores its second argument. Then for each symbol in its first argument, it stores a pointer to the internal value cell, and a copy of the symbols EVCP. Finally, CLOSURE returns an object of datatype DTP-CLOSURE which points at the block. This is the dynamic closure itself.

A symbol's internal value cell contains an EVCP only when its binding is being remembered by some dynamic closure; at other times the internal value cell just contains the symbol's value. The presence of the EVCP when a variable is closure-bound allows any modifications to the variable by functions or dynamic closures in CLOSURE's binding environment to be seen by the function closed over. An EVCP is treated in the usual manner (described above) by the BIND and UNBIND operations, but is treated as an invisible pointer by SET (the primitive function for updating a symbol's value) and by SYMEVAL (the primitive function for accessing a symbol's value). The word pointed to by the EVCP is called the symbol's <u>external value cell</u>. Thus, SET and SYMEVAL access and modify the symbol's external value cell, while BIND and UNBIND refer to the internal value cell.

When a CLOSURE is invoked as a function, the first thing that happens is that the saved environment is restored: the contents of the internal value cells of the current environment are saved on the binding PDL. Then the EVCPs stored in the dynamic closure are placed in the symbols' internal value cells, restoring the saved environment. Finally, the function is invoked with the arguments passed to the dynamic closure.

12.3 LEXICAL CLOSURES

While a dynamic closure is a function and an environment which captures a set of dynamic bindings, a lexical closure is a function and a lexical environment which captures a set of variables shared between the function and its lexical parent (and/or any higher lexical grandparents). A dynamic closure takes a snapshot of the global, fluid dynamic binding environment at the point in time that it is created. The environment of a lexical closure, on the other hand, is neither fluid nor temporally changing. A lexical environment is fully defined when the functions are written. Hence all environment is statically determined at compilation time.

When discussing lexical closures it is convenient to distinguish between a <u>closure function</u>, which is the functional part of a lexical closure, and a <u>closure parent function</u>, or just <u>parent function</u>, which is a function inside whose textual context the lexical closure is defined. Any time a function defined within an outer (parent) function freely refers to a local variable name or argument name belonging to its parent function, the function is considered a lexical closure. In the SILLY-ADDER function shown in Table 12-1, SILLY-ADDER is the parent function, and the internal anonymous LAMBDA function is a lexical closure.

Table 12-1 Some Lexical Closure Examples

```
(DEFUN silly-adder (num)
   (funcall #'(lambda () (+ num 1))))

(DEFUN make-summer ()
   (LET ((sum 0))
     #'(lambda (n) (INCF sum n))))

(SETF (SYMBOL-FUNCTION 'add-to-sum) (make-summer))
(add-to-sum 5) ==> 5
(add-to-sum 2) ==> 7
(add-to-sum 6) ==> 13
```

However, such extremely simple creations of lexical closures as that in SILLY-ADDER can be completely "compiled out" and reduced to in-line code. Although the anonymous LAMBDA function accesses its parent's argument, the compiler can statically determine that neither it nor its shared binding of NUM needs to be protected or preserved outside of the execution environment of SILLY-ADDER itself. Hence, it can just compile into a simple (+ NUM 1).

On the other hand, the MAKE-SUMMER parent function creates a lexical closure which is returned as the value of a call to MAKE-SUMMER, so the internal LAMBDA function cannot be compiled out. Furthermore, it shares the variable SUM with its lexical parent MAKE-SUMMER, hence it is a lexical closure. Let's call the closure returned by a call to MAKE-SUMMER ADD-TO-SUM. Calls to ADD-TO-SUM now side-effect the SUM variable in its lexical parent. The lexical closure ADD-TO-SUM, then, must contain both a functional object to perform the computation, and an environment object which captures the referenced variables in its textual scope. The lexical closure environment data structure must provide links back to the defining lexical parent. Moreover, the defining lexical parent MAKE-SUMMER must have a way of keeping track of the closures it creates (since subsequent calls will create more closures). Each such closure must have a separate, protected lexical environment. Hence stack frames of lexical parent functions must have reserved slots for maintaining information about their offspring. The following discussion covers the data structures for maintaining these links.

NOTE

> The implementation of lexical closures is quite different in Explorer Release 3 from earlier releases. The following discussion applies to Release 3 and later only.

12.3.1 Lexical Closure Implementation. The lexical closure object is represented by a word of type DTP-LEXICAL-CLOSURE which points to a cons cell whose car is the function and whose cdr is a locative to the environment. The layout of a lexical closure can be seen in Figure 12-1.

```
        LEXICAL CLOSURE OBJECT
   +----+----------------------+-----------------------+
   |    | DTP-Lexical-Closure  |           |           |
   +----+----------------------+-----------|-----------+
                                           |
+------------------------------------------+
|
|       LEXICAL CLOSURE STORAGE (A CONS CELL)
|  +----+----------------------+-----------------------+
+->|Norm| DTP-Function         | Pointer to FEF storage |
   +----+----------------------+-----------------------+
   |Err | DTP-Locative         |           |           |
   +----+----------------------+-----------|-----------+
                                           |
+------------------------------------------+
|
|       LEXICAL CLOSURE ENVIRONMENT
|  +----+-----------------------------------------------+
|  |Next|     Environment-Vectors-All-Vectors-Link      | -2
|  +----+-----------------------------------------------+
|  |Next|     Environment-Vector-Frame-Pointer          | -1
|  +----+-----------------------------------------------+
+->|Next|     Link to Parent Environment                |  0
   +----+-----------------------------------------------+
   |Next|     0th Lexical Variable Slot                 |  1
   +----+-----------------------------------------------+
                          .
                          .
   +----+-----------------------------------------------+
   |Nil |     Last Lexical Variable Slot                |  N
   +----+-----------------------------------------------+
```

Environment-Vectors-All-Vectors-Link -- Link used to thread together LEX-ALL-VECTORS (not needed if next one is NIL).

Environment-Vector-Frame-Pointer -- Locative to LEX-ALL-VECTORS slot of frame or NIL if already unshared.

Link to Parent Environment -- Link to parent environment or NIL if there is no parent.

Lexical Variable Slots -- EVCPs to the lexical variables on the stack or actual values if unshared.

Figure 12-1 Lexical Closure Structure

12.3.2 Dedicated Locals.

The implementation of lexical closures makes use of up to four dedicated local variable slots the function's runtime stack frame. In all cases it is determinable at compile time exactly which of these locations need be dedicated/reserved within a function. Whenever a particular one (or more) of these is <u>not</u> required, its location is available for general local storage within the function. These locals, if present, have the following uses:

Offset	Name	Description
2	LEX-PARENT-ENV-REG	Parent lexical environment pointer and primary lexical "base register"
3	LEX-ENV-B-REG	Secondary lexical "base register"
4	LEX-CURRENT-VECTOR-REG	Current environment pointer
5	LEX-ALL-VECTORS-REG	Head of environment list

Note that local slot 0 is reserved for &REST args and local 1 to hold the mapping table in combined flavor methods. The symbolic names are in the SYSTEM package and are defined in file SYS:UCODE;LROY-QCOM.

Compile time conditions for dedicating locals:

LEX-PARENT-ENV-REG (Initialized by CALL microcode)

> Dedicated if function is a closure or if function contains a MAKE-LEXICAL-CLOSURE macroinstruction.

LEX-ENV-B-REG (Initialized to NIL)

> Dedicated if compiler determines that a second lexical base register is needed.

LEX-ALL-VECTORS-REG (Initialized to NIL)

> Dedicated if any MAKE-LEXICAL-CLOSURE macroinstruction is emitted which indicates a push onto the environment list.

LEX-CURRENT-VECTOR-REG (Initialized to NIL)

> Dedicated if function contains a MAKE-LEXICAL-CLOSURE macro instruction.

12.3.3 Calling a Lexical Closure.

When a lexical closure is called, the microcode will store the new environment pointer in LEX-PARENT-ENV-REG of the called function. This allows efficient access to the environment from within the function. This approach requires that all lexical closures dedicate LEX-PARENT-ENV-REG as an environment pointer. Furthermore, any function creating an environment must use LEX-PARENT-ENV-REG as an environment "parent" pointer. Simple functions (non closures) are called with no special treatment of LEX-PARENT-ENV-REG in the called function.

12.3.4 Free Lexical References.

References to non-local lexical variables must specify the variable being referenced and which higher lexical environment the variable belongs to. Such references can be compiled into one of the general form instructions LOAD-FROM-HIGHER-CONTEXT, STORE-IN-HIGHER-CONTEXT, or LOCATE-IN-HIGHER-CONTEXT (returns a locative to the variable). Each of these takes from the stack a context descriptor, which is a FIXNUM whose lower 12 bits are the offset within a particular lexical environment. This defines the variable itself. The next 12 bits specify which environment An environment value of 0 represents the immediate parent (taken from LEX-PARENT-ENV-REG); 1 represents the grandparent, and so forth.

The majority of non-local lexical references, however, are either to variables in the immediate lexical parent or in the lexical grandparent. Hence, most non-local lexical references are handled more efficiently by using MAIN-OP instructions with a special base register field which specifies "higher lexical context" addressing (register value 3). The 6-bit offset field of the instruction is then used divided into a 5-bit lexical environment offset and one bit which specifies the base register (0 = LEX-PARENT-ENV-REG, 1 = LEX-ENV-B-REG) These base registers (actually in the function's local block) are assumed to point to valid environment structures. The 5-bit displacement field allows reference to the first 32 slots of the lexical environments pointed to by LEX-PARENT-ENV-REG or LEX-ENV-B-REG. The compiler will revert to the <access>-HIGHER-CONTEXT form instruction only when the short form addresses cannot be used; that is, the lexical environment displacement is greater than 31 or when access to a third environment frame not covered by a base register is required.

It is left up to the compiler to determine which particular environment pointers will reside in LEX-PARENT-ENV-REG and LEX-ENV-B-REG at given points in the code. While it is unlikely that the compiler would choose to have other than the parent pointer reside in LEX-PARENT-ENV-REG, it is quite possible that other than the grandparent pointer might reside in LEX-ENV-B-REG. (Remember that LEX-PARENT-ENV-REG must point to the lexical parent when MAKE-LEXICAL-CLOSURE is executed.) The compiler makes the choice based on the number of references to each lexical level.

Environment pointers are obtained by the instruction LOCATE-LEXICAL-ENVIRONMENT, which accepts an integer specifying the number of generations to go back (0 = parent, 1 = grandparent, etc.) and returns a pointer to the requested environment object on the PDL. This instruction, of course, begins its search at LEX-PARENT-ENV-REG. The simplest approach for the compiler would be to allocate LEX-ENV-B-REG on a global basis, although for some functions local allocation might win. The anticipated "typical" code sequence on entry to a closure would be something like:

```
LOCATE-LEXICAL-ENVIRONMENT  1     ;Grandparent
POP    LOCAL!3                    ;LEX-ENV-B-REG
.
.
```

after which we could have:

```
.
.
PUSH   LEX-A!7              ;7th slot of PARENT
.
.
POP    LEX-B!4              ;4th slot of GRANDPARENT
.
```

Should the compiler determine that a closure has no need of LEX-ENV-B-REG (as a base register) then it is free to use it as a normal local slot. The short form lexical reference instructions reduce the size of the code (1 instruction vs. 2) as well as reducing the time required to access grandparent (or higher) environments. The use of the LEX-ENV-B-REG base register allows us to incur the lookup cost only once; thereafter references cost the same as references to the parent environment.

12.3.5 Constructing Lexical Closures.

Lexical closure objects are constructed using the MAKE-LEXICAL-CLOSURE or MAKE-EPHEMERAL-LEXICAL-CLOSURE macroinstruction. These instructions take two arguments from the stack: the <u>environment descriptor list</u> and the FEF pointer to use. They return (on the stack) a pointer to the constructed lexical closure object. These instructions also use three implicit arguments, found in the invoking function's local slots, as follows:

1. LEX-CURRENT-VECTOR-REG is accessed to retrieve the pointer to the lexical environment to be used in the

constructed lexical closure. If LEX-CURRENT-VECTOR-REG contains NIL, then it is necessary to construct a new environment object according to the descriptor list mentioned above. In this case the pointer to the new environment object is stored into LEX-CURRENT-VECTOR-REG as a side effect of the macro instruction.

2. LEX-ALL-VECTORS-REG is used to maintain a list of environment objects constructed by the current function. See below for details.

3. LEX-PARENT-ENV-REG is used as the lexical parent pointer should it be necessary to construct a new environment object.

The two instructions differ in whether the (possibly new) environment object will be pushed onto the environment list at LEX-ALL-VECTORS-REG. For MAKE-LEXICAL-CLOSURE, the environment object (whether new or old) is pushed onto the list at LEX-ALL-VECTORS-REG, provided that it is not already the first element of the list. For MAKE-EPHEMERAL-LEXICAL-CLOSURE, no attempt is made to push the environment onto the LEX-ALL-VECTORS-REG list.

The decision as to whether an environment object needs to be added to the environment list is made by the compiler and is based on whether the closure being created can possibly outlive the current function's stack frame.

12.3.6 Lexical Environments.

Lexical environment objects are constructed as a side effect of the MAKE-LEXICAL-CLOSURE macroinstruction. Environment objects are always consed in the heap. Each object is a cdr-coded list containing a pointer to its parent environment followed by some number of slots which may contain either the value of a variable or an indirect pointer (EVCP) to the variable's value cell.

Value entries in a freshly constructed environment correspond to locals/args of the current function which are freely referenced and which the compiler has determined are "read only"; i.e., they are never stored into. In this case each constructed environment gets its own copy of the value of the local/arg rather than the EVCP used in the general case.

12.3.7 Environment Descriptor List.

The environment descriptor list is used as a template when constructing an environment object. The first element of the list is the number of slots to be created, and each remaining element is a FIXNUM which defines one lexical variable slot, in the format shown in Figure 12-2.

```
       2 2 2
  bit  4 3 2                                   6 5        0
       +-+-+-----------------------------------+-----------+
       |A|V|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0| offset    |
       +-+-+-----------------------------------+-----------+
       /   \
    arg     value
```

Figure 12-2 Lexical Variable Slot Descriptor

The sign bit is 0 for a local variable, or 1 for an argument. The next bit is the value/reference flag; when it is 0, the slot will hold a DTP-EXTERNAL-VALUE-CELL-POINTER to the variable; when 1, the value of the variable is copied directly into the environment slot. If the number of slots is greater than the number of of slot descriptors provided, then the remaining slots are initialized with a value of NIL and do not correspond to any local variables; the compiler uses such slots as value cells for lexical variables which are in excess of the maximum number of 64 that can be allocated in the stack.

12.3.8 Managing Environments.

We have already discussed the creation of new environments in our discussion of the operation of MAKE-LEXICAL-CLOSURE with respect to LEX-CURRENT-VECTOR-REG, although we did not indicate how LEX-CURRENT-VECTOR-REG might ever be set back to NIL once an environment had been constructed. The motivation for forcing a new environment object to be constructed is to allow representation of environments containing different instances of selected variables. To accomplish this also requires the use of the LEXICAL-UNSHARE instruction.

The LEXICAL-UNSHARE instruction creates an instance of a variable by copying its value (from the stack) into the heap and then relocating all references (from environments in the environment list) to point to the heap allocated value. References to this variable in subsequent environments will point (via EVCPs) into the stack, thus referencing a different instance of the variable. There is also a LEXICAL-UNSHARE-ALL instruction which can be used to unshare all of the local variables at once.

Thus we need to set LEX-CURRENT-VECTOR-REG to NIL whenever we have reason to unshare any of the variables of the environment. This is done implicitly by the LEXICAL-UNSHARE and LEXICAL-UNSHARE-ALL microcode.

The UNSHARE instructions will access the environment list via LEX-ALL-VECTORS-REG. This list is also accessed implicitly upon RETURNing from a frame when the environment pointer points here

(EPPH) flag is set in the frame's Call-Info word. This flag is set only when an environment object is pushed onto the environment list at LEX-ALL-VECTORS-REG. The microcode must perform an UNSHARE-ALL operation when this flag is set.

COMPILER NOTE: Locals/args which are determined to be read only need not be unshared since copies of the values will have been distributed at the time the environment was constructed.