F.G.H.
7/11/85

LispCourse #41: The Compiler and MASTERSCOPE

The Interlisp Compiler

What is the Interlisp Compiler?

An important goal of any program is that it run FAST. *Ceteris paribus*, the faster it runs the more work it can do.

Running fast, usually means doing as little work as possible while the program is running.

There are two ways to accomplish this:

1) Minimize the amount of work to do

2) Do some work ahead of time so there is less work to do while the program is running.

Recall from LispCourse #34 (and Homework #34) that the Lisp evaluator (i.e., EVAL/APPLY) does lots and lots of work whenever it evaluates a function call.

Because of this, evaluating a function call is relatively slow.

And because programs are made up of evaluating function calls, (interpreted) Lisp programs tend to be relatively slow.

Moreover, in the evaluator outlined in LispCourse #34 (& 35), all of the work is done each and every time the function is called, at the time function is called.

Much of this work is in fact redundant and need only be done once, e.g., when the function is called for the first time.

Thus the Lisp evaluator in LispCourse #34 ignores both of the speed-up techniques described above.

The goal of the ***compiler*** is to make these two speed-up techniques available in the Interlisp evaluation process.

The compiler is a program that takes a Lisp function definition in source code form (i.e., in the form that you write it in) and does as much of the "evaluation work" as it can.

It then rewrites the function definition in a new form (i.e., compiled code form) that captures all of the work it has done.

When the Lisp evaluation process encounters a function definition in compiled code form, it can take advantage of the work the compiler has already done and therefore APPLYing the function is much faster than if the function were in source code form.

Since the compiler is run once, before the function is ever evaluated, it does two things:

1) it minimizes work by getting rid of the redundancy in multiple evaluations

2) it moves some of the cost of function call evaluation from the time of evaluation to some earlier time (i.e., to the time when compilation is done).

The result is that compiled functions can be evaluated much, much faster than the equivalent interpreted (i.e. source code) functions. This is an obvious advantage.

The disadvantage of the compiler scheme is that you have to take the time to compile your functions before you run them.

This can be a big disadvantage when you are debugging and testing your program.

In these cases, you make frequent changes to your function definitions.

If you have to recompile your functions each time you make a change, the compile time can easily outweigh the time wasted by the slower evaluation of interpreted functions.

Other disadvantages of compiled code are the following:

1) Compiled code can be read only by real Lisp wizards. To the rest of us, it looks like gibberish.

2) Compiled code to some extent brings back the data/program distinction into Lisp.

In source code form, function definitions are just list structures that can easily be treated as data by another function.

In compiled code form, function definitions are special objects that are not easily accessible using the standard Interlisp data manipulation mechanisms.

The bottom-line is that compiled code is good for helping finished programs run fast while interpreted code is good for testing and debugging, and when you need to blur the distinction between data and program.

An important feature of Interlisp (in fact of most Lisps) is that compiled code and interpreted code can be freely intermixed without any special considerations on the part of the programmer.

An example of what the compiler does

The compiler is a very complex program that does lots and lots of fancy things to speed up the evaluation of Lisp code.

Compiler research and the problem of compiled code optimization are important research areas in computer science.

None of this will be covered here (since I don’t know anything about it!).

However, the following is an example of what the compiler does:

Consider the following abstract function definition:

(LAMBDA (A B) (LET (D E)(LET (F G) (LIST A B D E F G))))

In the ***interpreter*** outlined in LispCourse #34, evaluation of the LIST function call would involve calling the **LookUpValue** function 6 times to look up the values of the atoms A, B, ...

Each time, the **LookUpValue** function would search up the stack looking for a binding of the given variable. For A & B, it would find the binding on the third stack frame from the bottom. D & E would be found on the second stack frame, etc.

All of this stack searching would take lots of time.

The compiler, however, can do some of the interpretation work ahead of time. For example:

The compiler can predict (based on the structure of the Lisp source code) that any reference to A within the second embedded LET will refer to the A bound in the third stack frame from the bottom.

Furthermore, since A is the first item in the parameter list, the compiler can figure out that A will be the first bound variable in its stack frame.

The compiler knows about the format of the stack. It can therefore generate code that directly fetches the value of the first bound variable in the third stack frame without any search of the stack.

Thus, the compiler would replace all the references to the value of A within the second embedded LET with compiled code that just looks up the value of the first bound variable in the third stack frame.

Then, when the function is later evaluated, the expensive stack lookup operation for the value of A would be skipped.

The compiler could do the analogous thing for all of the bound variables within this function defintion.

If the function were compiled, the evaluation of the (LIST ...) statement would involve no stack lookup operations, resulting in a much faster evaluation.

Note that the compiler *cannot* do the same thing for free variable references. This is because the stack frame binding refered to by a free variable is determined at *run time* by what functions and bindings are currently on the stack.

The compiler has no way of predicting what the stack will look like at run time, and can therefore not replace the stack lookup by a direct reference.

A good compiler, however, could also do some optimizations in this function.

In particular, the two LETs could be collapsed into a single LET, eliminating the need to create an additional stack frame.

This is possible because the compiler can tell that the embedded LET does not rebind any of the variables used in the outer LET.

Using the compiler

Compiler Questions

All of the functions that invoke the compiler start by asking the user the following series of questions. Each question should be answered "Yes" (or "Y") or "No" (or "N"), followed by a Carriage Return.

Listing? ž Asks whether you want a detailed listing of the compiled code being generated. Always answer this question with "No".

REDFINE? ž Asks whether the compiled code should replace the source code as the function definition for the functions being compiled. In general, this question should be answered "Yes".

Occasionally, you may just want to create a file of compiled code without altering the definitions in the current virtual memory. In this case, answer this question with "No".

SAVE EXPRS? ž Asks whether to save the original source code whenever a function is redefined using its compiled code.

If "No", then the original code is lost when the function is redefined with the compiled code.

If "Yes", then the original source code is placed on the property list of the atom that is the function’s name using the property EXPR.

The editor, the compiler, the file package, etc. all know about the EXPR property and handle in appropriately.

For example, if you call DEdit on a function whose definition is compiled code, DEdit will instead edit the source code stored in the EXPR property (if there is any).

If the function definition stored in EXPR is changed during the DEdit, then DEdit automatically redefines the function to be the *new* source code and saves the old compiled definition on the property list under the property CODE.

In general, answer this question "Yes". because you will often want to edit the source code and recompile the function.

If you answer "No", you will have lreoad the source code from a file (if you even bothered to save it) when you want to change the function.

OUTPUT FILE? ž Asks whether to write the compiled code to a file that can be LOADed at a later time or in a new sysout, etc.

"No" means no file will be created.

"Yes" will cause the compiler to prompt you for a file name.

Anything else will be interpreted as a file name, in which case "Yes" will be assumed and that file will be used.

Note: as a shorthand you can answer the **LISTING?** question can be answered using the following:

S ž use the same answers to all questions as given for the last compile.

F ž just compile to a file without redefining the functions in the virtual memory.

ST ž answer REDFINE? and SAVE EXPRS? with "Yes" but ask about the output file.

STF ž answer REDFINE? with "Yes" and SAVE EXPRS? with "No" and ask about the output file.

Functions that invoke the compiler

The following function invoke the compiler:

(COMPILE *Functions*) ž Compile the current source code definitions for each of the functions in the list *Functions*. If *Functions* is an atom, (LIST *Functions*) is used.

The current source code definition is either the function definition or the source code stored under EXPR on the property list.

(TCOMPL *Files*) ž Used to compile source code files created by MAKEFILE. *Files* is a list of source code files to be compiled one-by-one in order. If *Files* is atomic, (LIST *Files*) will be used.

Compiling a MAKEFILE file involves compiling all of the functions on that file, writing the compiled code to a new file of the same name (but with the extension .DCOM), and then copying all of the non-function items (e.g., VARS, RECORDS, etc) from the source file to the new compiled file.

The resulting .DCOM file is a LOADable replacement of the original MAKEFILE source file, except that the function definitions contaion compiled rather than source code.

TCOMPL returns a list of DCOM file produced.

Note: Since TCOMPL automatically produces a file, it does not ask the OUTPUT FILE? question.

(RECOMPILE *File*) ž Used to recompile a single source code file *File* after one or more of its functions have been edited using DEdit.

RECOMPILE works like TCOMPL, except that it does not compile all functions on *File*. Instead the following scheme is used:

If the function definition in the virtual memory is an EXPR (i.e., is not compiled code), then RECOMPILE compiles that definition and writes it to the output DCOM file. [As indicated above, functions are redefined to be their EXPR version (source code) whenever they are edited using DEdit.]

If the function definition in the virtual memory is NOT an EXPR, then RECOMPILE simply copies the previous compiled definition from the previous version of the DCOM corresponding to *File*.

RECOMPILE is considerably faster than TCOMPL when only one or a few function definitions have been changed because it doesn't recompile functions that haven't changed.

Example

```
32_(DEFINEQ (AAA (LAMBDA (A B C) (PLUS A B C))))
(AAA)
33_(DEFINEQ (BBB (LAMBDA (A B C) (LIST A B C))))
(BBB)
34_(SETQ EXAMPLECOMS '((FNS AAA BBB)(VARS (XYZ 44))))
((FNS AAA BBB) (VARS (XYZ 44)))
35_(MAKEFILE 'EXAMPLE)
```

```
{PHYLUM}<HALASZ>EXAMPLE.;1
36_(TCOMPL 'Example]
listing? no
redefine? yes
save exprs? yes
(dwimifying AAA)
(AAA (A B C))
(AAA redefined)
(dwimifying BBB)
(BBB (A B C))
(BBB redefined)
({PHYLUM}<HALASZ>EXAMPLE.DCOM;1)
37_DF[AAA]
prop unsaved
AAA
38_(MAKEFILE 'EXAMPLE)
{PHYLUM}<HALASZ>EXAMPLE.;2
39_(RECOMPILE 'EXAMPLE]
listing? N
redefine? Y
save exprs? Y
(dwimifying AAA)
(AAA (A B C))
(AAA redefined)
BBB,
{PHYLUM}<HALASZ>EXAMPLE.DCOM;2
```

Special Considerations when Writing Code to be Compiled

Interlisp takes ever effort to make compiled and interpreted code totally interchangeable. However, there are some ways in which this simply cannot be done. The following are some special considerations involved in writing code that will be compiled.

All of these considerations are optional. They are simply ways of taking advantage of feature available in compiled but not interpreted code.

GLOBALVARS

As described above, the compiler writes special code to handle many of the variable references more efficiently than the standard stack search mechanism.

As we discussed in LispCourse #34, free variable reference in interpreeted code is always done through a stack search unless you use the GETTOPVAL/SETTOPVAL functions. In the later case, the value cell of the atom is used directly without any stack search.

You have a little more control over this process in code produced by the compiler. In particular, you can declare any variable to be a **GLOBALVAR**.

Declaring a variable to be a GLOBALVAR tells the compiler that whenever that variable is used freely in a function, code should be generated to directly access the value of the atom, skipping the stack search. Declaring a variable to be a GLOBALVAR is essentially telling the compiler to generate code to do a SETTOPVAL or GETTOPVAL whenever the variable is used freely.

If you don't declare a variable to be a GLOBALVAR, then the compiler will generate code to do the normal stack search when it encounters that variable used freely.

There are several ways to declare a variable as a GLOBALVAR:

1) Put a clause in the COMS list that contains the functions that you want to use that variable as a GLOBALVAR. The clause should be of the form **(GLOBALVARS *Var1 Var2* ...)**.

When any of the functions on the file are compiled, free variable references for any variable in a GLOBALVARS clause will be compiled as global variables.

2) Put a property GLOBALVAR with value T on the property list of the atom. Anytime the compiler runs across this atom used as a free variable, it will compile it as a global variable.

3) Add the atom to the global list GLOBALVARS. Anytime the compiler runs across this atom used as a free variable, it will compile it as a global variable.

Macros

A macro is a Lisp form that is evaluated at compile time to produce a Lisp form that is in turn compiled.

The evaluation of a macro to produce the form to be compiled is called ***expanding*** the macro.

For example: *(LIST (CAR ’(PLUS DIFFERENCE)) A B)* might be a macro that when expanded returns the Lisp form (PLUS 33 C) given that the value of A is 33 and B is C when the macro is expanded.

Note that the form that gets entered into the compiled function is (PLUS 33 C). When this compiled function later gets evaluated, the variables A and B have no effect whatsoever, only the variable C (which didn’t appear in the macro definition at all) is releveant to the evaluation.

Contrast the concept of a macro with the following Lisp construction:

(EVAL (LIST (CAR ’(PLUS DIFFERENCE)) A B))

When this form is evaluated, the inner LIST function returns (PLUS 5 6) given that the value of A is 5 and B is 6. This form is then evaluated by EVAL.

Note that if this form were compiled, it would be compiled exactly as is. The value of A and B would not enter into the compiling process but would appear as variables in the compiled definition (as they are in the source definition). The value of A and B are then used at evaluation time, i.e., when the compiled form is being evaluated.

Compiler macros are an important part of most Lisp dialects. Unfortunately, in Interlisp they are relatively poorly implemented and very clumsy to use. Therefore, macros are not used with high frequency in Interlisp.

Using Macros

Macros are used very much like functions ž in fact many of the Interlisp "functions" we have talked about are in fact implemented as macros.

Example: If TestList is a macro name, then (TestList A B C) would be a form that calls that macro.

The name TestList cues the interpreter or the compiler that the TestList macro should be expanded using A B C as arguments to the expansion. (Expansion is described below.)

The form that actually gets evaluated or compiled is the form that results from this expansion.

The original form (TestList A B C) is basically ignored.

The compiler and the interpreter treat macros slightly differently:

Whenever he compiler encounters a form, it first checks to see if the CAR of the form is an atom with a macro definition. If so, it expands the macro (as described below) and compiles the result of this expansion instead of the original form.

If there is no macro definition, then the complier looks to see if the CAR has a function definition.

In contrast, the interpreter looks first for a function definition and only if one is not found does it look for a macro definition for the CAR of a form. If it needs and finds a macro definition, it expands the macro (as described below) and evaluates the result of this expansion in place of the original form.

Defining macros and macro expansion

To define a macro, you need to put a macro definition onto the property list of the macro’s name under the property MACRO (or DMACRO).

The definition should be an Lisp form with one of the following formats:

(*List SExpression*) ž this is called a substitution macro. When this macro is expanded, each time an atom appearing in *List* appears in *SExpression*, the corresponding argument from the macro call (e.g., the A B C in *(TestList A B C)*) is substituted in its place. The result is *SExpression* with these substitutions.

Example:

If *(PUTPROP ’ADD2 ’MACRO ’((X) (PLUS X 2)))*

Then *(ADD2 (CAR Z))* would expand to *(PLUS (CAR Z) 2)*

(*LitAtom SExpression*) ž When this macro is expanded, *LitAtom* is bound to the CDR of the calling form. *SExpression* is then evaluated. The result of the expansion is the result of this evaluation.

This format allows you to compute the SExpression to be compiled (or evaluated).

Example:

```
If (PUTPROP ’LIST ’MACRO ’(Args
        (LET ((ConsList (CONS NIL NIL)))
                (FOR Item IN (REVERSE Args)
        DO (SETQ ConsList
              (CONS
                 (CONS (QUOTE CONS)
                     (CONS Item
                     ConsList))
              NIL)))
   (CAR ConsList))
```

Then: *(LIST 1 2 3)* would expand to *(CONS 1 (CONS 2 (CONS 3 NIL)))*.

(LAMBDA *ParamList FunctionDefinition*) ž When this macro is expanded, it simply returns the function call form generated by using the LAMBDA expresion as the CAR of the form and the arguments to the macro call as the CDR of the form.

Example:

If *(PUTPROP ’ABS ’MACRO ’(LAMBDA (X) (COND ((GREATERP X 0) X)(T (MINUS X)))))*

Then (ABS (CAR (LIST 1 3))) would expand to (*(LAMBDA (X) (COND ((GREATERP X 0) X)(T (MINUS X))))) (CAR (LIST 1 3)))*

The purpose of this is to avoid the expense of a function call in the compiled code, but to make the source code look and function just like a function call.

The macro expansion takes care of turning the (ABS ..) form into an equivalent form that does not require a function call when compiled.

The bottom line on Interlisp macros

In several years of Interlisp programming, I have written only two macros.

On the other hand, there are others, especially those who were brought up in other Lisp dialects, who use macros quite a bit.

MASTERSCOPE

MASTERSCOPE is a very handy Interlisp package that comes loaded with every standard Interlisp sysout.

MASTERSCOPE will analyze functions for you, store these analyses in a database, and then allow you to ask questions about the analyzed functions and their relationships.

The following will be a very brief description of some of the commands available in MASTERSCOPE. The whole package is fairly complex and will not be covered in detail here (but see Chapter 13 of the IRM).

MASTERSCOPE has "English-like" commands implemented as a single NLAMBDA-NoSpread function whose name is ".". Thus all MASTERSCOPE commands consist of a period followed by a command or "question".

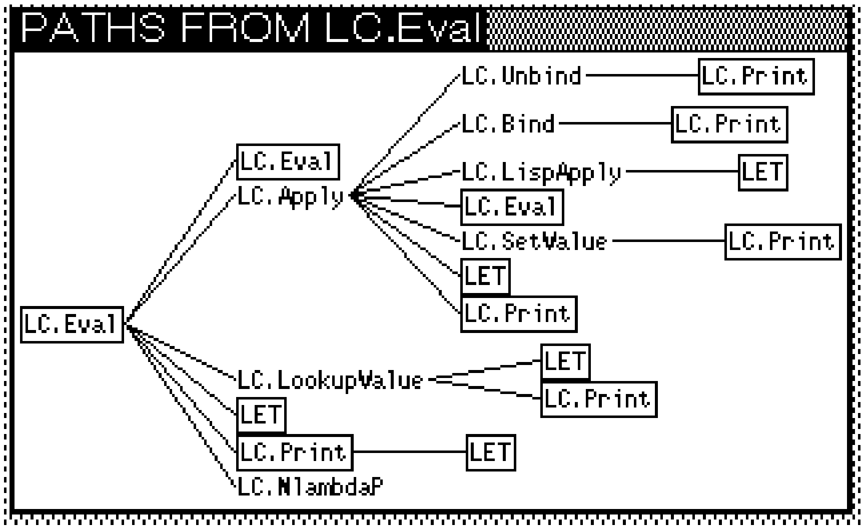
The following are some of the useful commands:

(. ANALYZE ALL ON *File*) ž analyzes all of the functions on the source file *File*. This is usually the first command in a MASTERSCOPE analysis of a set of functions.

Example: (. ANALYZE ALL ON HOMEWORK34)

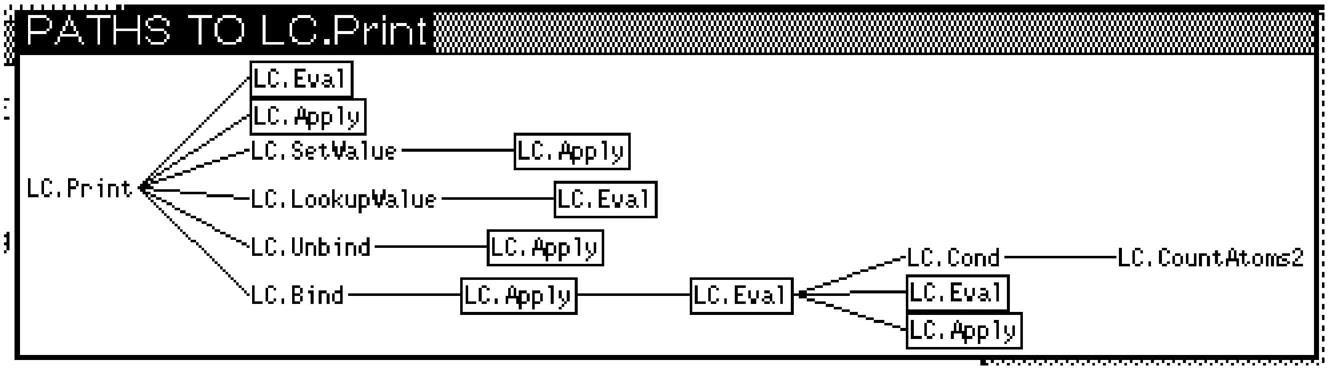
(. SHOW PATHS FROM *Function*) ž opens a window that shows a graph of all the (analyzed) functions that are called directly or indirectly from *Function*.

Example: (. SHOW PATHS FROM LC.Eval)



(. SHOW PATHS TO *Function*) ž opens a window that shows a graph of all the (analyzed) functions that directly or indirectly call *Function*.

Example: (. SHOW PATHS TO LC.Print)



(. WHO CALLS *Function*) ž returns a list of functions that directly call *Function*.

Example: (. WHO CALLS LC.Eval) returns *(LC.Apply LC.Eval LC.Cond)*

(. WHO USES *Variable*) ž returns a list of functions that use the variable *Variable*.

Example: (. WHO USES Stack) retruns *(LC.Bind LC.Unbind LC.LookupValue LC.SetValue LC.Apply LC.Cond LC.Eval LC.LispApply)*

(. WHO USES AS A RECORD *RecordName*) ž returns a list of functions that use the record *RecordName*.

(. EDIT WHERE ANY CALLS *Function*) or **(. EDIT WHERE ANY USES *Variable*)** or **(. EDIT WHERE ANY USES AS A RECORD *RecordName*)** ž invokes DEdit on every function that directly calls the function *Function* (or uses the variable *Variable*). The exact form in each function that does the calling (or using) is selected and centered in the DEdit window when DEdit starts up.

The (. EDIT WHERE ..) command can be followed by a ž and one or more commands from the TTY Editor. In this case, these commands will be carried out on the functions instead of invoking DEdit on the functions.

Example: The following will change all varaibles named *Stack* to *MyStack* within the HOMEWORK34 functions.

```
(. EDIT WHERE ANY USES Stack ž (R Stack MyStack))
```

MASTERSCOPE actually has many much more specific questions you can ask (e.g., free variable usage, who fetches from a record, etc.). Consult the IRM for details.

There are some strategies, however, that make MASTERSCOPE work better.

For example: If you have many functions each with a bound variable that has the same "semantics" in each of the functions, then call that bound variable the same thing in each function. Naming it the same in each function will not effect the operation of the functions because its locally bound, but it will make it easy to change all of the functions at once if you want to change the semantics of the variable.

In NoteCards. many (maybe 100 in total) functions have a bound variable called NoteCardID. It was initially called just ID, but at one point a LinkID was added to many of these functions and we decided to rename all ID variables to NoteCardID. Luckily we had named them all alike so that this could be accomplished in a single MASTERSCOPE command.

The bottom line: learn to use the basics of MASTERSCOPE. Its one of the handiest tools around for writing large programs in Interlisp.

References

The Interlisp Compiler is the subject of Chapter 12 of the IRM.

Macros are the subject of Section 5.5 of the IRM.

MASTERSCOPE is the subject of Chapter 13 of the IRM.