Recursive Compiling and Programming Environments
SUMMARY

Leslie Lamport

25 July 1984

1 Introduction

While there are many programming environments, to my knowledge only Interlisp has the following feature: A user can write a program with Interlisp and add that program to his version of Interlisp, using it to help write and debug other programs. While there are many screen editors, to my knowledge only EMACS has the following feature: A user can write a program with EMACS and add that program to his version of EMACS, using it to help him edit other files.

This self-modifying feature of Interlisp and EMACS adds enormously to their versatility, making it possible to build very powerful systems one piece at a time. It is probably one of the major reasons why Interlisp and EMACS are so popular where they are available. For example, I added a command to my EMACS that performs nontrivial syntax checking for input to a typesetting program. This was fairly easy because I could do it entirely within EMACS. Adding such a command to any other editor I know of would require modifying the editor's source code and recompiling it—a formidable task.

The ability to create such self-modifying systems derives from the following property of the languages in which they are implemented: a program written in the language can manipulate the text of a program as data and then execute it. This is possible in Lisp, the language underlying Interlisp, because of the *eval* function. It is possible in TECO, the language used to implement EMACS, because a TECO program can manipulate a string of characters as data, then store it in a "Q-register" and execute it as a program. Not surprisingly, a version of EMACS has also been written in Lisp. (There is also a program called EMACS that is written C, but it is a hollow shell of EMACS because it lacks the self-modifying property.)

Unfortunately, Lisp and TECO lack many features found in more modern programming languages—features like block structuring and typing. In this paper, I describe a method for extending any imperative language to permit the writing of such self-modifying programs. The basic idea is to add statements that allow a compiled program to compile and execute a procedure—hence, the name *recursive compiling.*

I will describe recursive compiling in terms of Pascal, since Pascal is both well known and simple. Pascal's simplicity means that it lacks certain features needed for the proposed extension that are present in more sophisticated languages like Ada. This will force me to describe explicitly exactly what language features are required. However, recursive compiling can be added to any imperative language.

Recursive compiling requires the following extensions to Pascal:

- String variables. (They are already present in many extensions to Pascal.)
- Variables of type *procedure* whose values are executable procedures, with a mechanism for executing the contents of a procedure variable.
- A *compile* procedure to compile a string (the program text) into a procedure.

Certain other extensions are proposed for reasons of efficiency and elegance. To simplify the implementation of interactive programs, I also propose extensions to Pascal's input/output commands.

2 String Variables

My first extension to Pascal is a built-in *string* data type. A string is a sequence of ASCII characters; the declaration of a string variable places no limit on the length of string it may hold. A string variable must be able to hold an entire text file—perhaps as long as 500,000 characters.

With modern operating systems, implementing the storage management to handle such long strings should present no problem. However, compiling efficient code requires restricting the operations that may be performed on long strings. The basic string operations that are needed are insertion and deletion. There should also be *sticky pointers*—pointers specifying a particular character position within a string. For example, a sticky pointer that points to the fifth character of a string is changed to point to the seventh character when two characters are inserted at the beginning of the string.

It must also be possible to read a file into a string variable, and to write the contents of a string variable onto a file. The complete paper will describe a set of simple functions and procedures for performing these operations.

3 Procedures

A new *procedure* data type must be added to Pascal, where the value of a variable of type *procedure* is an executable procedure. An ordinary procedure, compiled at the same time as the main program, is a constant of type *procedure*. Thus, a Pascal procedure declaration is a special case of constant declaration. Writing

procedure *foo* ...

is completely analogous to writing

const *foo* = ...

Procedure values are executed by the command

execute *proc*(*args*) **exception** *excep_proc*

where *proc* and *excep_proc* are values of type procedure, and *args* is an argument list. This statement executes procedure *proc*, executing *excep_proc* only if there is a run-time error in *proc*. Procedure *excep_proc* must take two arguments that are used to specify the type of error, as explained in the complete paper. A run-time error in the execution of *excep_proc* is a run-time error in the program containing the **execute** command.

A procedure value includes the number and types of its parameters, so run-time type checking can be performed. Compiling an **execute** command generates code to compare the type of each argument in the argument list (which, in Pascal, is known at compile time) with the types of the procedure's parameters, which are part of the procedure value. A type mismatch produces a run-time error, invoking the exception procedure. As will become evident below, the actual object used to represent a type is a pointer to a symbol-table entry.

Without some restriction on the assignment of values to procedure variables, one can execute a procedure outside the scope in which it should be executed. For example, suppose *foo* is a procedure declared in block B, and *foo* accesses a variable v local to block B. Code in B could assign the procedure *foo* to a procedure variable p declared outside B. Code external to block B could then execute p, accessing v where it is no longer defined.

To prevent this, I introduce the concept of a level number. A declaration is at level n if it occurs in a block nested within $n-1$ other blocks. Every object that is referred to by name—a variable, a user-defined type, a declared procedure, etc.—has a level number—namely, the level number of its declaration. (Predefined objects, such as built-in types and functions, are assigned level number -1.) Proper nesting means that it is impossible for a statement in a level-n block to access an object with a level number greater than n.

To every value of type *procedure*, we attach an *execution-level number* that is the smallest (outermost) level at which the procedure may be executed. Its value equals the largest level number of any object it references, excluding those objects that are local to the procedure itself. For example, a procedure that mentions only its parameters, its own local variables, and built-in types, can be executed anywhere. The execution-level number of a procedure is easily determined at compile time, since the level number of every declared object is known.

The code to implement an assignment to a procedure variable compares the level number of that variable (known at compile time) with the execution-level number of the value being assigned to it. A run-time error occurs if the execution-level number is less than the variable's level number.

Everything I have done for procedures can also be done for functions. However, it is not clear whether it should be done, since exception handling in function calls is rather awkward. A run-time error in a function call should invoke an exception function that returns a value of the same type as the original function. This means that, in Pascal, one cannot write a single exception handler that works for all functions.

4 The Compile Procedure

Pascal is next extended by adding a built-in **compile** procedure, which is called as follows:

$$\textbf{compile}(\mathit{proc},\ \mathit{str},\ \mathit{var}_1,\ \mathit{var}_2)$$

where *proc* is a procedure variable and *str* is a value of type *string*. (The variables var_1 and var_2 are used to return error information and will be described in the complete paper.)

The **compile** procedure compiles its string argument exactly as if that string had appeared as a procedure declaration in the current block. For example, when applied to the string

```
procedure  foo ( arg : integer )
   var  x :  user_defined_type ;
 begin   y := x + arg ;
          ...
 end
```

the identifiers *user_defined_type* and *y* have the same meanings as in the context where the **compile** procedure appears, while *x* and *arg* have the obvious local definitions.

Note that all names declared in this **procedure** declaration are invisible outside the scope of the compiled procedure. In fact, a value of type *procedure* has no name. The only reason for giving a procedure a name, as we did above, is for defining recursive procedures.

To implement recursive compiling, a compiled program must keep a symbol table that is updated at run time whenever a procedure or function is invoked or returns—i.e., every time the context changes. This is exactly the kind of symbol-table manipulation that the compiler must perform, so there is no real implementation problem. However, it does take time, which may make it unacceptable if execution speed is important. To improve efficiency, we need two classes of declarations: *visible* and *invisible* ones. Only those objects—constants, types, variables, procedures, and functions—declared to be visible are known to the **compile** procedure, and they are the only ones that must be added to and removed from the symbol table during invocation and exit.

The cost of symbol-table manipulation is incurred only during function and procedure calls, and only for variables visible to the **compile** procedure. Functions and procedures that are invoked frequently will make almost all of their declared objects invisible, so they will incur little overhead. Only high-level procedures that are infrequently invoked because their execution takes a long time—procedures such as a text editor, whose execution lasts for an entire editing session—will use significant numbers of visible objects. In this case, the only cost is the space occupied by the symbol table, which is negligible

The complete paper will provide a syntax for indicating in its declaration whether an object is to be visible or invisible. As we shall soon see, when debugging a procedure, one wants all its objects to be visible to the **compile** function. The language should therefore contain a statement that forces all declarations to be visible.

There are good reasons for constraining the scope of declared variables even without recursive compiling. The scope rules of Pascal imply that a variable declared within a procedure *proc* may be accessed by any procedure nested within *proc*. This makes it hard to keep track of data dependencies in large programs, so it would be nice to require an explicit "export" declaration for a variable to be used outside the current nesting level. The introduction of visible and invisible objects is just one instance of the general idea of controlling the export of declarations, and should be considered in that more general context. This will be discussed in the complete paper.

5 Interactive Display

The line-oriented input/output of Pascal is not well-suited to implementing modern, screen-oriented user interaction. Pascal's output commands need to be augmented with ones for creating "windows" on a terminal screen and displaying data inside them.

The complete paper will describe commands to create windows. To display text within a window, I propose a command that assigns a string variable to a window in such a way that the screen always displays the current contents of the variable. More precisely, the command specifies a string variable together with the portion of the variable that is to appear in the window—for example, by specifying a sticky pointer to the first character to appear on the screen. The screen always displays the selected part of the string variable's current value. I will not discuss the display of graphical data.

Pascal's repertoire of input commands needs to be extended to handle character-by-character input. An obvious extension, which exists on some versions of Pascal, is the addition of a boolean-valued function to test if a character has been typed, and a function that returns the typed character. It would also be useful to have built-in functions for reading the position of a mouse-controlled cursor. These functions would return values indicating which character within which window the cursor is pointing to.

6 Symbol Tables

One difficulty with implementing an extensible system in this way is that Pascal does not provide any extensible data structures. For example, a system should allow the user to define named commands which he can later invoke by name. Using standard Pascal data structures, this can be done only by declaring an array of procedures with enough elements to hold one procedure for every command that the user will ever want to define.

A more elegant approach is to introduce a new data structure called a *symbol table*. The statement

$$foo : \textbf{symbol table} \ \ \textbf{of} \ \ type$$

is similar to the statement

$$foo : \textbf{array} \ \ [1 \,..\, n] \ \ \textbf{of} \ \ type$$

However, instead of being indexed by integers, the elements of *foo* are indexed by strings. One can write expressions like *foo*["John"] to represent the value of the element associated with the name "John". There must also be some way to test whether a symbol table entry has been defined.

Note that this symbol table mechanism is already used by the compiler, so it should not be difficult to implement in object code. Symbol tables will be discussed at greater length in the complete paper.

7 Applications

I will illustrate the utility of recursive compiling with two simple examples: text editing and interactive debugging.

7.1 Text Editing

Using the display functions described above, one can write a simple procedure that acts as an EMACS-style text editor. The text is kept in a string variable that is displayed in a window on the screen. An array *char_cmd* of procedures is declared by

$$\mathit{char\_cmd} : \textbf{array}\ \ [0 \mathinner{\ldotp\ldotp} 256]\ \ \textbf{of}\ \ \textbf{procedure}$$

and the main loop of the editor is as follows, where *Read*(*Keyboard*) returns the next typed character and *ord*(*InChar*) is the ASCII code for character *InChar*:

$$\begin{array}{l} \mathit{InChar}\ := \ \mathit{Read}(\mathit{Keyboard})\ ; \\ \textbf{execute}\ \ \mathit{char\_cmd}[\mathit{ord}(\mathit{InChar})]\ \ \textbf{exception} \ldots \end{array}$$

One customizes the editor, á là EMACS, by executing commands that compile the contents of a buffer and assign the resulting procedures to a chosen element of *char_cmd*.

The complete paper will discuss this example in more detail, indicating how the editor provides an environment for writing new commands. Use will be made of the symbol table data structure to store named commands.

Such an editor can do more than just edit text. Commands can easily be created to assist in writing Pascal programs, and to check for syntax errors. Editor commands can also be added that compile and run programs.

7.2 Interactive Debugging

One can define a procedure *Debug* that invokes an editor, allowing the user to enter (and edit) Pascal statements that are then executed in the context in which the *Debug* procedure was invoked. The user can put a call of the *Debug* procedure anywhere in his program. When the procedure is executed, it allows him to interrogate and change the status of his program by executing Pascal commands. For example, executing $write(x)$ prints the current value of x.

One can also write an interactive interpreter. I will quickly outline here how it is implemented; the complete paper will contain a more detailed description. The heart of the interpreter is a recursive procedure with a single string argument, which is the Pascal text for a sequence of statements. This procedure presents the user with a buffer containing the Pascal text, and gives him commands to edit the buffer, pop back to a higher level, or execute a specified statement in the buffer. The latter is the interesting option. It is handled by compiling the statement (in its current context) and executing it, unless the statement is a procedure or function call. In the case of a procedure call, the interpreter compiles a call to a new procedure having the same declarations as the one being called, but whose body contains only a recursive call to the interpreter with an argument consisting of the procedure body. A function call is handled similarly.

8 Conclusion

Recursive compiling allows a complex system to be bootstrapped from a simple one. I showed how, from a simple skeleton, one can gradually create a very sophisticated editor. Moreover, since the commands one writes are compiled, just like ordinary Pascal programs, the resulting editor can be as efficient as one written and compiled in a traditional fashion. One can create a Pascal-based EMACS.

The example of the interpretor indicates how recursive compiling permits one to implement sophisticated programming environments in a similar incremental fashion.

The additions to an ordinary Pascal compiler required for recursive compiling are really quite modest. Except for those commands introduced to implement interactive display, every extension to Pascal I have described involves giving to the programmer capabilities that the compiler itself possesses. I would estimate that these additions would add about ten percent to the cost of implementing a Pascal compiler—assuming that it is designed right from the beginning to be a recursive compiler. Unfortunately, modifying an existing compiler is not so simple. I hope to initiate a project at SRI to build a recursive Pascal compiler for a personal computer, but no such project is yet under way.

Bibliography

The complete paper will include citations of relevant literature, including references to Interlisp, EMACS, and the various extensions to Pascal that are mentioned.