An Icon Subsystem for UNIX[†] Emacs*

William H. Mitchell

TR 84-8

May 1984

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

†UNIX is a trademark of Bell Laboratories.

An Icon Subsystem for UNIX* Emacs

1. Introduction

This report describes a version of the UNIX Emacs[1] text editor that contains the Icon programming language[2] as a subsystem. A general familiarity with both Icon and Emacs on the part of the reader is assumed.

Emacs contains the language MLisp as a subsystem. MLisp is an embedded language: MLisp programs can perform editor operations, and access both buffer data and information about the editor state. The combined system is quite powerful; it is possible to perform programming tasks with MLisp that are quite difficult to perform with conventional languages. The tasks that MLisp often is used for can be divided into several groups:

simple text transformation
assistance with text composition
assistance with editor control
interactive systems

Simple text transformation tasks include changing text in certain contexts, modifying each line in a file in a similar way (for example, inserting line numbers), and moving text from one location to another. Often, these sorts of operations can just as easily be performed by a conventional programming language.

MLisp can also assist with text composition. For example, when composing program text for languages that have syntactic elements that must be balanced, such as parentheses and brackets, MLisp functions can help the programmer locate matching elements and note imbalance during the course of text composition. As another example, consider a function that accepts a list of variable names and constructs an output statement that prints the name of each variable and its associated value.

In addition to helping with text composition, MLisp functions can assist the user by controlling the editor state in various ways. Examples are a function that allows a list of files being edited to be manipulated as a stack or a queue, and a function that allows screen states to be saved and manipulated as objects.

At the upper end of the MLisp application spectrum are fairly complex interactive systems. Part of the standard Emacs system is the MLisp *info* package that allows the programmer to access a hierarchical database of information about editor commands. There also are various MLisp systems to handle electronic messages.

The power and convenience of the Emacs environment arises from a combination of elements: text buffers, which serve as variable length strings with a large repertoire of associated operations, on-screen display of multiple text buffers with automatic update upon change, binding of operations to sequences of keystrokes, and a sophisticated process control system. Together, these elements form a "critical mass" with which it is easy to accomplish textually-oriented programming tasks.

As mentioned, however, the programming environment presented by Emacs is weak in one area: MLisp. MLisp is very primitive; it has only integers and strings as primary data types, and the control structures are quite limited. The lack of sophisticated control structures is often manifested in the form of overly complex and convoluted code, but the lack of data structures makes it impractical to implement packages requiring complex data structures.

Because the Emacs/MLisp environment often provides convenient solutions for applications where the shortcomings of MLisp are not crippling, it is logical to assume that a language that is much more powerful than MLisp would allow the utility of Emacs to be applied to more elaborate problems. To this end, the Icon

* UNIX is a trademark of Bell Laboratories

programming language has been incorporated in Emacs as a subsystem. Icon was selected because it is a modern, high-level language with dynamic typing that supports a variety of higher-level data types and powerful control operations that include generators and goal-directed evaluation. This extended version of Emacs is completely downward compatible with standard Emacs and also provides the programmer with the option of using Icon to formulate solutions to problems. A description of the implementation of the system is included as an appendix to this report.

2. Programming with Icon in Emacs

Programmable editing is, as the name implies, editing performed under program control. Editing is essentially making modifications to a buffer. Not surprisingly, the fundamental paradigm of programmable editing is to accomplish a task by changing the contents of a buffer. The following examples illustrate programmable editing and the techniques involved therein.

Line Numbering

A simple illustration of the contrast between conventional programming and programmable editing is shown by a program that numbers the lines of a file. This can be formulated as an Icon procedure that reads lines from standard input and writes numbered lines to standard output:

```
procedure number_lines()
        local lineno, line
        lineno := 1
        while line := read() do {
                write(right(lineno, 5), "    ", line)
                lineno +:= 1
                }
end
```

An alternative formulation, when using Icon in Emacs, is to express the task in terms of modifications to the current buffer:

```
procedure number_lines()
        local lineno
        lineno := 1
        beginning_of_file()
        while eobp() = 0 do {
                beginning_of_line()
                insert_string(right(lineno, 5) || "    ")
                lineno +:= 1
                next_line()
                }
end
```

Note the differences between the two procedures: The first is a filter that produces new versions of the input lines as its output. The second makes an in-place transformation by moving the cursor to each line in turn and inserting the line number at the beginning of the line. While the first procedure employs the sequential nature of the read function to serialize the input, the second solution uses cursor movements and positional checks to maneuver through the file in a rather clumsy fashion.

In practice, performing an operation on a sequence of lines is common. Consider a function to generate the lines in a buffer. The two obvious alternatives are to generate strings containing the text of the lines or to generate buffer positions. Because a paradigm of programmable editing is in-place modification of text, generation of buffer positions is more idiomatic. The function

```
procedure gen_lines()
        beginning_of_file()
        while eobp() = 0 do {
                beginning_of_line()
                suspend
                next_line()
                }
end
```

is a generator that suspends with the cursor positioned at the start of each line in the buffer in turn. Using gen_lines, number_lines can be written in a form that is comparable to the Icon solution:

```
procedure number_lines()
        local lineno
        lineno := 1
        every gen_lines() do {
                insert_string(right(lineno, 5), "    ")
                lineno +:= 1
                }
end
```

gen_lines is an unconventional generator because it modifies the state of the editor instead of returning a value. There is an implied protocol because gen_lines assumes that the caller leaves the cursor positioned on the same line that the last resumption of gen_lines moved it to. Because gen_lines is a generator, calculation of the first result includes positioning the cursor at the first line.

For the preceding example, the solution provided by programmable editing does not provide any significant advantage over the conventional solution, but a slight modification of the problem makes a conventional solution awkward: Number the lines in decreasing, rather than increasing order. A conventional solution might be to read the file once to determine the number of lines, and then make a second pass over the file to produce the numbered lines. An obvious idea for an editing solution is to get the number of lines in the file from the buffer management system, but Emacs does not maintain this information. Two immediate alternatives are present: write a function like gen_lines that generates lines in reverse order or use gen_lines to count lines:

```
lineno := 0
every gen_lines() do lineno +:= 1
```

and then make a second pass with gen_lines to add line numbers in decreasing order.

The example of number_lines illustrates the closest approach of conventional programming to programmable editing: the direct mapping of sequential filtering into an editing framework. In this case, editor primitives are used, but they do not really apply any leverage to the problem. A related class of programmable editing tasks is composed of problems that are suited for conventional solutions, but where editor operations can be used to gain leverage and simplify the solution. Problems in this class are difficult to illustrate with medium-sized examples; instead, they tend to be either trivial solutions to problems (typically involving regular-expression searches and transformations) or elements of larger solutions, as shown below.

Augmented Text Composition

A second task that programmable editing can be applied to is partially mechanized text composition. During the course of text composition with an editor, the user often is required to enter repetitive or tedious text sequences, usually for the benefit of another program. An example is in-line font changes for *troff*[3]. To direct *troff* to italicize the word italics, the user must type \fIitalics\fP. This is quite tedious manually, but with a programmable editor, it can be made much more convenient. Consider a function font_change to insert font changes. font_change is to be invoked by the user after completing a word that is to be presented in a particular font. An implementation of font_change is:

```
procedure font_change()
        local font, state
        message("Font? ")
        font := map(char(get_tty_character()), &lcase, &ucase)
        insert_string("\\fP")
        state := savestate()
        re_search_reverse("[ \t\n]")
        forward_character()
        insert_string("\\f" || font)
        restorestate(state)
end
```

The user is prompted for a single letter that represents the desired font. The resulting letter is capitalized using the map function. (The char function converts the integer returned by get_tty_character into an Icon string.) The trailing font reversion is appended and the buffer state is saved so that subsequent restoration leaves the cursor positioned after the trailing "P". There is some question as to the extent of the text to include in the font change, but in practice a good choice has proven to be text bounded by blanks, tabs, or the start of a line. The desired boundary is located with a regular-expression search, thus using the power of the editor to apply leverage in the solution. Finally, the leading font change is inserted and the previous buffer state is restored, leaving the user positioned to enter more text.

Augmented text composition provides the user with an editing environment. The user easily can build tools and then apply them to text composition tasks. To a limited extent, similar effects can be achieved with editors that allow portions of buffers to be filtered through external programs, but using a programmable editor is much more convenient.

Function Location

A notion similar to augmented text composition is to assist the user at editing levels above that of text manipulation. The UNIX *ctags*[4] program accepts C, Fortran, or Pascal source files as input and produces a file of "tags" as output. The tag file associates function names with source file locations by producing three-column output, where the first column is the function name, the second column is the name of the source file containing the function, and the third column is a regular-expression that can be used to locate the function. For example, the line

```
join<tab>listfuncs.c<tab>?^join(a, b)$?
```

indicates that the function join is in listfuncs.c and can be located using the pattern ^join(a, b)$. The editor *vi*[5] supports a tag *function* command that examines the tag file and edits the file containing the named function, positioning the cursor at the start of the function.

Emacs does not have this capability, but it is easy to write a procedure to do the job. The steps that need to be taken are:

```
visit the tags file
locate the tag for the specified function (if it exists)
extract the file name and pattern from the tag
visit the file and search for the pattern
```

The following function performs these steps:

```
procedure find_tag()
        local fcn_name, tag
        fcn_name := get_tty_string("Function? ")
        visit_file("tags")
        beginning_of_file()
        if not re_search_forward("^" || fcn_name) then
                error_message("No such function in tag file")
        tag := string_to_list(current_line(), '\t')
        visit_file(tag[2])
        beginning_of_file()
        re_search_forward(tag[3][2:-1])
        line_to_top_of_window()
        delete_other_windows()
end
```

The tag file is visited and is searched for a line beginning with fcn_name, which is the name supplied by the user. If the function cannot be found, this is noted as an error. The cursor is now positioned on the line containing the tag for the desired function.

There are several ways to dissect the tag line. A naive way to do this is to work through the line using word and character movements interspersed with region_to_string operations to extract the file and pattern fields. A more sophisticated approach is to use regular-expression matching and the region_around_match function to extract the fields. A third method is to let Icon do the work.

The Icon procedure string_to_list(s, c) breaks the string s into pieces delimited by members of the cset c. For example, string_to_list("a b c", ' ') returns ["a", "b", "c"]. current_line() is an Icon procedure that returns the text of the line the cursor is positioned on. Thus,

```
tag := string_to_list(current_line(), '\t')
```

breaks the tag line into a three-element list that contains the function name, the file name, and the pattern.

Given the file name and pattern, it is a simple matter to visit the file and locate the function by searching for the pattern. Note that using pattern[2:-1] as the search string removes the question mark on each end of the pattern. Finally, the function is presented at the top of a one-window screen.

Window Configuration

It often is desirable to present several windows on the screen at the same time. Often the windows should be in a certain order and be of a certain size. Consider a function configure_windows that lays out the windows on the screen according to a specification. configure_windows accepts a list argument that describes the desired window configuration. The list has the form:

[[*name*, *size*], ..., [*name*, *size*]]

where *name* is the name of the buffer to associate with the window and *size* is the height in lines of the window.

The algorithm is simple: The screen is split into the appropriate number of windows, each window is bound to the named buffer, and the height is adjusted to the appropriate size.

If window_list is the argument to configure_windows, the screen must be split into as many windows as there are elements in window_list. Emacs provides no primitive for splitting the screen into n windows, but it does provide split_current_window, which splits the current window into two windows. The procedure make_windows splits the screen into n windows:

```
procedure make_windows(n)
        delete_other_windows()
        every 1 to n - 1 do {
                split_current_window()
                next_window()
                }
end
```

To get to a known starting point, delete_other_windows is performed, leaving one window on the screen. After each split, next_window is performed to avoid splitting the same window repeatedly.

A problem with standard Emacs is that there is no primitive to determine what window the cursor is in. Two primitives were added to Emacs to help the programmer manipulate windows on the screen. The function first_window returns a unique integer corresponding to the first (the top) window on the screen and this_window returns a unique integer corresponding to the window the cursor is presently in. Given these functions, a procedure to position the cursor is:

```
procedure go_to_window(n)
        while this_window() ~= first_window() do
                next_window()
        every 2 to n do
                next_window()
end
```

Note that an alternative approach would be to provide go_to_window as a primitive and also to provide window_position, which would return the serial position of the window on the screen.

Another component for configure_windows is a function to make a window contain a certain number of lines. As with window splitting, Emacs does not provide a direct means to do this, but it does provide sufficient primitives to formulate the operation. window_size can be written as:

```
procedure window_size(n)
        local adj, adjfunc
        adj := window_height() - n
        if adj < 0 then
                adjfunc := enlarge_window
        else
                adjfunc := shrink_window
        return_prefix_argument(adj)
        adjfunc()
end
```

The method is to determine the discrepancy between the actual size and the desired size and then supply enlarge_window or shrink_window with a prefix argument to effect the necessary change. It is unfortunate that enlarge_window and shrink_window do not allow the adjustment count to be transmitted as an argument.

configure_windows can now be written:

```
procedure configure_windows(window_list)
        local wspec
        make_windows(*window_list)
        go_to_window(1)
        every wspec := !window_list do {
                switch_to_buffer(wspec[1])
                window_size(wspec[2])
                next_window()
                }
end
```

Note that if the size component of a specification is omitted, the expression wspec[2] fails and window_size is not called. This permits some window sizes to "float" while constraining others. For example,

```
configure_windows([["Directory", 6], ["Text"]])
```

makes the first window six lines high and associates it with the buffer Directory. The second window is associated with the buffer Text and occupies the rest of the screen.

3. Programmable Editor Design

Programming with Icon in Emacs has lead to an interesting conclusion: The set of primitives provided by Emacs is poorly suited for programmable editing. A review of the MLisp primitives reveals that while a few of them obviously are intended for use by programs, the majority are for use by a person manually editing a body of text.

The orientation of the primitive set towards manual editing is reasonable, but in many cases the programmer has been neglected. For example, consider the cursor motion commands. Prefix arguments are used to indicate repetition counts for the commands. That is, to move forward three words, a prefix argument of 3 is supplied to forward–word. This specification is convenient from the keyboard, but the programmer must use the awkward construction:

```
(provide–prefix–argument 3 (forward–word))
```

For programming purposes

```
(forward–iword 3)
```

would be much better. Using prefix arguments rather than actual arguments is pervasive, and while this mechanism is interesting, the treatment of prefix arguments is certainly not ideal. It would be much simpler if a prefix argument were treated as the *last* argument to a function.

The prefix argument situation is annoying, but it is more of a misfeature rather than a design problem. A more serious problem is the lack of programmer-oriented analogues for various user-oriented functions. Consider the functions beginning–of–line and end–of–line. From the programmer's standpoint, it might be better to have a single function that subsumes the actions of these two functions. For example, line–position(1) would move the cursor to the start of the line, while line–position(0) would move the cursor to the end of the line. Similarly, other values would indicate intraline positions.

There are two basic approaches to the design of a set of primitives for programmable editor. The first is to base the design on manual editing activities. The Emacs command set is an example of this approach. Operations have been factored out so that frequently performed tasks are separate commands. While the user benefits from this approach, the programmer must either simulate a user's activities or design a virtual interface.

The second approach is to design the set of primitives with the programmer in mind. This approach does not mean that the user must suffer. Rather, a set of primitives well-suited for users can be built on the set of primitives for the programmer. This method appears to provide dual advantages: the programmer has a concise, orthogonal set of operations with which to formulate programs, and the user is unaffected.

The design of the primitives supplied by Emacs has obviously been influenced by the limitations of MLisp. By incorporating Icon into Emacs, a more powerful language is present, but the primitive set is unchanged. It is interesting to consider how the language features of Icon could be entwined with a programmable editor.

Because Icon provides data structures, it might be worthwhile to maintain data structures that correspond to internal objects. Two such objects are buffers and windows. A variety of information about buffers is available via primitives, but it might be better to supply this information in the form of a data structure. For example, buffers might be represented using

```
record buffer(name, size, file, mode, modified)
```

The collection of active buffers could then be represented by a list of these records. Similarly, windows have various sorts of information associated with them, and a window record could be provided:

```
record window(buffer, height, width, dot)
```

As with buffers, a list of these window records would represent the windows visible on the screen. These lists of buffer and window records could then be associated with new keywords, &buffers and &windows. Using these new structures, consider a function that switches to each buffer associated with a file whose name ends in a specified character and does a recursive–edit.

```
procedure edit_buffers(suffix)
        local buf

        every buf := !&buffers do
                if buf.name[−1] == suffix then {
                        switch_to_buffer(buf)
                        recursive_edit()
                        }
end
```

Other keywords easily can be imagined: &text, to represent the text of the current buffer; &line, for the text of the current line; &word, for the word (if any) that the cursor is positioned on; &cursor, as the position of the cursor.

4. Conclusions

Programmable editing is a curious activity. It allows the programmer to compose solutions that are combinations of editing activities and conventional programming. With a system such as Icon in Emacs, the programmer has available a spectrum of solutions, which range from those that mainly use editor operations to those that mainly use Icon operations. It is often difficult to determine what the best combination of methods is, and as in conventional programming, there are various styles of solutions that can be formulated. Obviously, much of the confusion over methods results from the fact that two complete programming systems are present and the programmer is free to draw from either at any time. An obvious solution to this problem is to confine the programmer to a single system, but that is exactly what is provided by conventional editors and conventional languages. The power of programmable editing lies in the combination of two paradigms and thhe power of this combinnation is much greater than the sum of its parts.

Acknowledgements

Ralph Griswold graciously served as advisor for this project and made numerous suggestions regarding the form and content of this report. Chris Fraser and David Hanson provided a number of helpful comments on a late draft of this report. This project benefited greatly from a number of discussions with Roger Hayes. Several readers of the Unix-Emacs@CMU-CS-C and Editor-People@SU-SCORE mailing lists contributed to this work in various ways.

References

[1] *Unix Emacs Reference Manual*, UniPress Software Inc., Highland Park, New Jersey, 1983.

[2] Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1983.

[3] J. F. Ossanna, *NROFF/TROFF User's Manual*, Computing Science Technical Report 54, Bell Laboratories, Murray Hill, New Jersey, 1976.

[4] Ken Arnold, *Ctags(1) Manual Page*, Unix Programmer's Manual, Volume 1.

[5] William N. Joy, *Ex/Vi Reference Manual*, Unix Programmer's Manual, Volume 2C.

[6] R. E. Griswold, W. H. Mitchell, and S. B. Wampler, *The C Implementation of Icon; a Tour Through Version 5*, Technical Report 83-11a, Department of Computer Science, University of Arizona, 1983.

Appendix

The Incorporation of Icon in Emacs

5. Modifications to Icon

In standard Icon, the linker combines ucode files produced by the translator and produces an icode file that is composed of all procedure and data declarations that are to be used during program execution. This mode of operation is suitable when it is known in advance what modules will be required during the course of execution, but Emacs users have a wide variety of programming packages at their disposal and it is not practical to determine, *a priori*, the complete set of "programs" that may be needed during an Emacs session. The approach taken by Emacs is to allow users to load packages of functions as they are required. Furthermore, when a package is being developed, it can be modified and reloaded without needing to reload modules that were not affected.

In order to accommodate the Emacs model of usage and development, a variant of Icon, referred to as Icon$_i$, has been developed for incorporation into Emacs. Icon$_i$ eliminates the need for the Icon linker to produce a complete executable unit; rather, Icon$_i$ allows executable units to be dynamically incorporated into the system. Icon$_i$ has an additional built-in function, load(filename), that loads the named Icon executable file into the Icon interpreter. For example, suppose there are two files, dohello.icn:

```
procedure main()
        load("hello")
        say_hello()
end
```

and hello.icn:

```
procedure say_hello()
        write("Hello world!")
end
```

Translating and running these files via

```
icont dohello.icn
icont hello.icn
dohello
```

produces the output: "Hello world!".

While the load function is very simple outwardly, it conflicts in many ways with the internals of standard Icon. The Icon interpreter evolved from the Icon compiler[6] and has a number of static constraints that are not expected in an interpreter. There are two major classes of static constraints in Icon: the allocation of global and static variables, and the allocation of code and identifier regions.

One of the functions that the Icon linker performs is the resolution of variable references. The expression x + y generates the ucode:

```
var   x
var   y
plus
```

which instructs the interpreter to push pointers to the variables x and y onto the stack and perform the plus operation. The var instruction is generic — the linker translates it into one of three icode instructions depending on whether the variable is global, local, or static. In icode, references to variables are in terms of offsets from an address. Local variables are contained in the procedure's stack frame, while global and static variables are contained in arrays that are allocated when the run-time system is initialized. Consider the program:

```
global x

procedure main()
        write(x)
end
```

The linker identifies three global variables in the program: x is a global variable because is it declared as such; main, because it is the name of an Icon procedure; and write, because it is the name of a built-in procedure. As global variables are encountered, they are assigned to consecutive positions in the global variable array. In this example, main, x, and write occupy positions 0, 1, and 2 of the global array. (The global variable main is special-cased by the linker and always resides at position 0 in the global array.) The write expression generates the icode:

```
global      2
global      1
invoke      1
```

This causes globals 2 (write) and 1 (x) to be pushed on the stack and the procedure write to be invoked with one argument.

The link-time binding of global variables to memory locations is incompatible with dynamic loading because it is not possible to determine during linking what global variables will be present at run-time. Icon$_i$ solves this problem by delaying the binding of global variables to memory locations until the first run-time reference to the variable. For global variable references, the Icon$_i$ linker outputs icode instructions that contain the names of global variables. The example above produces:

```
global_s    "write"
global_s    "x"
invoke      1
```

At run-time, a global variable reference is accomplished by looking up the operand of global_s in a hash table and obtaining the index of the named variable in the global variable array. The global_s instruction is replaced by a global instruction that has the index of the variable as its operand. Execution then proceeds as it does for a global instruction.

In standard Icon, the linker collects scope information about variables from various modules and pools the information to determine the scope of each variable. For example, if a file containing a procedure p is linked with other files that reference p, but do not declare it, p is referenced as a global variable.

This method of scope resolution is unsatisfactory for Icon$_i$ because the linker cannot determine which variables are global.. Standard Icon uses aan implicit local scope for variables that have no other scope information. Icon$_i$ uses an implicit scope of *undefined.* When the linker encounters a var instruction for a variable with no scope information, it generates a uglobal_s instruction rather than a global_s instruction. At run-time, when a uglobal_s instruction is encountered, the global hash table is searched for the named variable. If it is found, the uglobal_s instruction is replaced with a global_s instruction and the operation is restarted. If the named variable can not be found, it is taken to be a run-time error. This is similar to the treatment of unbound variables in Lisp.

The primary ramification of this scheme is that local variables in Icon$_i$ procedures must be declared; in practice, this approach has proven to be quite satisfactory.

Operation of the Load Function

One of the tasks of the load function is to merge the globals defined or referenced in the file that is being loaded with the existing globals. An array of global variables is maintained, as in standard Icon, but the binding of variables to locations in the array is maintained by a hash table. For efficiency and simplicity, the hash table is represented by a static array of entries that associate global variable names with the locations of the variables in the global array.

The load function reads an icode file into memory and analyzes it. An Icon$_i$ icode file has several regions:

header
procedures
records
global variables
global variable names
strings

The header section contains the total size of the other regions and identifies the location of each of the other regions.

The procedure section is composed of zero or more procedures. Each procedure has a procedure block, a code section, and a string section. The string section contains strings and other non-integer literals that are referenced in the procedure.

The record section is composed of zero or more record constructor procedure blocks.

The global variable section is an array of variable descriptors. For the global variables that are procedures, the descriptors hold the locations of the associated procedure blocks in the procedure section.

The global variable name section is an array of string descriptors holding the names of the global variables. The descriptors point into the string region of the icode file.

The ultimate result of loading a file is the addition (or possible replacement) of global variables and associated values. load scans the icode file's global variable and global name arrays and takes appropriate actions.

The first step is scanning of the global variable array to put procedures and records into allocated memory. For procedure variables, the procedure block is copied into the heap, and the code and string regions of the procedure are copied into static memory. The variable descriptor is updated to reflect the location of the procedure block. The block has pointers to the code and string data for the procedure. Records are simplified versions of procedures and do not have code and string regions.

The second step is merging the globals in the icode file with the existing globals. The global name list in the icode file is scanned and an attempt is made to enter each name into the hash table. If the name is not already present, the next available element in the global array is associated with the variable. If the named global variable already exists and the value of the new global is non-null, it replaces the value of the existing global. This causes procedure definitions in an executable file being loaded to replace existing definitions, but prevents redundant global declarations (with no associated value) from overwriting existing values.

Static variables also present problems for dynamic loading. Standard Icon maintains an array of static variables just as for globals, and icode references to static variables are in the form of offsets into the array. Icon$_i$ treats static variables much like global variables, but composes names for them. In the linker, when a ucode var instruction referencing a static variable is encountered, the icode instruction

static_s *"procedure—name$variable—name"*

is generated. For example, in:

```
procedure main()
        static count
             .
             .
             .
```

references to count generate static_s instructions naming main$count. In the Icon$_i$ interpreter, static names are hashed just as global names are hashed, and references to static variables are handled as references to global variables.

Icon does not place restrictions on uniqueness of names or positions of fields in records. Consequently, field references must be resolved at run-time. The standard Icon linker constructs a matrix with record names on one axis and field names on the other axis. The entry values are the positions of the fields in the records. This matrix is passed on to the interpreter. This technique is obviously unsuitable for Icon$_i$, since it relies on the linker to bind field names to the positions of fields in each record.

The basic scheme used in Icon$_i$ is to pass field names through to the interpreter as strings, and also to pass information about the positions of fields in each record. As an example, consider the program:

```
record a(x, y)

global r

procedure main()
        r := a(1, 2)
        r.x
end
```

The expression r.x generates the icode

```
global_s    "r"
field_s     "x"
```

Furthermore, two global variables, a.x and a.y, are created and given the integer values 1 and 2 to denote that x and y are the first and second fields in a. When the interpreter executes the field_s instruction, the global variable for the record r (already on the stack) is examined and the record constructor block associated with it is found. The name of the record constructor, a, is concatenated with a dot and the named field to produce a.x. The global hash table is searched for a.x and the value associated with the entry, 1, is the desired field in the record block of r.

Storage management issues

Standard Icon has several data regions that are built by the linker and that conflict with dynamic loading of executable files. The linker gathers the data blocks and icode for all procedures and places them in a contiguous region. Data for non-integer literals and strings that represent the names of objects are gathered and placed in a contiguous area known as the identifier region. Also, data blocks for records reside in their own region of memory. As described above, the arrays of global and static variables and the array of global names are also allocated at link time. Furthermore, icode contains references that are relative to the start of the code and identifier regions. The various regions are laid out by the linker and allocated by the iinterpreter when it begins execution.

In standard Icon, memory regions are arranged in the following order:

code region
record region
global variable array
global variable name array
static variable array
identifier region
co-expression stacks
string space
heap

The code, record, identifier, and co-expression stack regions are not relocatable. However, with dynamic loading, data may need to be added to all of the first six regions. However, the data above each region is not relocatable and thus it is not feasible to expand the regions. Also, although the heap and string spaces are relocatable, they must be contiguous, and thus the various regions that need expansion cannot be placed above the heap.

In Icon$_i$, allocated memory consists of

global hash table
global variable array
global variable name array
static memory
string space
heap space

In the design of Icon$_i$, it was decided to fix the maximum number of global variables. This number can specified via an environment variable, but once Icon$_i$ starts, the limit cannot be changed. The global hash table, and global variable and name arrays all have the same number of elements. Note that there is no region for static variables, since Icon$_i$ treats them as a special case of global variables.

Icon$_i$ divides dynamically allocatable memory into two regions: a static region and a relocatable region. The relocatable region is further divided into string and heap spaces. The static region contains memory objects that cannot be relocated. If the static region needs to grow, the heap and string spaces are moved up in memory. A slightly modified version of the UNIX *malloc* routines is used to manage the static memory. The *malloc* routines used differ from the standard ones in that they request memory from the Icon$_i$ garbage collector rather than obtaining it via the *sbrk* system call.

In Icon$_i$, rather than grouping the code and identifier regions for all procedures in a contiguous region, code and identifier regions are associated with each procedure. Procedure blocks are allocated in the heap and contain pointers to the code and identifier blocks for the procedure, which reside in static memory. A record block is treated as a special type of procedure block that has no associated code or identifier blocks. Code blocks are placed in static memory because expression frame markers contain absolute icode addresses. Similarly, the data for string and cset literals used by a procedure is contained in the identifier region.

The run-time system has a pair of variables that point to the data block and the identifier/code region for the current procedure. These variables are updated whenever control passes from one procedure to another.

6. Incorporation of Icon$_i$ in Emacs

The fundamental design issues for incorporating Icon$_i$ in Emacs were how to interface the Icon$_i$ subsystem to Emacs and how to deal with MLisp. The MLisp subsystem is fairly well isolated from the rest of the editor. This led to a "black-box" approach for incorporating the Icon$_i$ subsystem in Emacs. That is, the MLisp subsystem is treated like a black-box with a small number of connections to the editor. Icon$_i$ also is treated like a black-box and appropriate modifications were made to both systems to match the interface architecture of MLisp. This approach yields a system with full MLisp compatibility, but with Icon and MLisp disjoint in several ways.

At the conceptual level, the incorporation of Icon in Emacs was primarily a matter of resolving a number of conflicts between the Emacs/MLisp and Icon environments.

Data Type Systems

The most basic conflict between Icon and MLisp lies in the set of data types supported by the systems. Icon supports a number of data types, but MLisp supports only three: integers, strings, and *markers*, the former two intersecting with Icon's data types.

MLisp integers serve their conventional purpose and certain character constants are represented as integers. Icon and MLisp integers are completely compatible; when an Icon or MLisp value needs to be converted to the other value domain, the conversion is made by simple assignment.

MLisp strings are represented as pointers with associated lengths, but operations involving strings are inconsistent as to whether the length field determines the length of the string or if a null character denotes the end of a string. For example, if a `region-to-string` operation is performed, the length of the string is the length of the region even if it includes a null character, but if that string is inserted with `insert-string`, the insertion stops when the null character is reached. Icon strings are variable in length and may contain any character, including the null character. When an Icon string is converted to an MLisp string, the length of the Icon string is used as the length of the MLisp string and the string is copied from Icon's string space into static memory allocated by Emacs. The string cannot remain in Icon's string space, since a garbage collection may occur and the string may be relocated. This would invalidate any pointers to the string that Emacs had established. To avoid this problem, when an MLisp string is converted into an Icon string, the string is copied into the Icon string space and is given the length specified in the MLisp value.

Markers are objects that specify positions in a buffer. Whenever a change is made to the buffer, the markers associated with the buffer are updated so that their relative position in the text remains constant. Icon_i stores marker data using unstructured data (*udata*) blocks. These blocks are variable in size and reside in the heap. A *subtype* field is associated with each udata block. Udata blocks can be used to experiment with new data types without having to go to the trouble of making the system modifications required to add a new block type.

Variables

The co-existence of MLisp and Icon presents a number of problems dealing with variables. Icon is statically scoped, while MLisp is dynamically scoped. MLisp also has *buffer specific* variables, which have an instantiation associated with each buffer. An additional complication is that the name spaces for MLisp variables and functions are not disjoint — an MLisp function can have the same name as an MLisp variable.

Emacs maintains distinct variable spaces for MLisp and Icon variables. As described above, in lieu of scope information for a variable, Icon_i generates a uglobal_s instruction for the reference. The Icon_i system embedded in Emacs performs a three-step search to resolve uglobal_s references. First, the Icon_i global hash table is examined. Second, the table of Emacs macro names is examined. Third, the table of Emacs variable names is examined. If the variable is either an Emacs macro or an Emacs variable, the uglobal_s instruction is replaced with a eglobal_s instruction. If the variable is not found in any of these three places, a run-time error is produced.

When an Icon program accesses an MLisp variable, it uses the instantiation of the variable deemed to be current by the MLisp subsystem.

MLisp functions that use names of variables as arguments are called from Icon with the desired variable names given as strings. For example,

```
declare_buffer_specific("first", "last")
```

declares first and last as buffer-specific variables.

Identifier names

Another conflict between Icon and MLisp is caused by identifier names. Icon identifiers start with a letter or an underscore and are followed by any number of alphanumeric and underscore characters. MLisp identifiers can contain a variety of characters — many more than are valid in Icon. In practice, most MLisp identifiers are valid Icon identifiers with the exception of the extensive use of dashes. MLisp names of this form are mapped into valid Icon names by using underscores instead of dashes. For example, the MLisp function switch-to-buffer is referred to as switch_to_buffer in an Icon program. This simple transformation is adequate in most cases.

Argument evaluation

MLisp uses a rather peculiar scheme for argument passing. It has no notion of formal parameters for a function; instead, it has a function

(arg *n prompt*)

whose value is the nth argument to the current function. If less than n actual arguments are supplied, the user is prompted for a value with the string prompt. Furthermore, the arguments are not evaluated until they are demanded. Consequently, control structures can be written in MLisp. (This argument handling regime is similar to FEXPRs in Lisp.)

When a call is made from MLisp to Icon, all the MLisp arguments are converted into values and an Icon argument list is built from the values. When a call is made from Icon to MLisp, calls to the arg function cause the appropriate value in the Icon argument list to be converted to a MLisp value and returned.

Control Structures

In addition to control structures for altering the flow of control, MLisp has several control structures that are used to save state information temporarily. For example,

```
(save–excursion expr1  exprn)
```

saves information about the current buffer and then executes each $expr_i$. When all the expressions have been executed, the information about the buffer is restored. This allows an MLisp function to leave the state of the current buffer undisturbed.

save–excursion presents a major problem because it is used frequently and it cannot work within the call-by-value framework of Icon. To counter this deficiency, Icon_i has two functions: savestate and restorestate. savestate packages the information saved by save–excursion and returns it as a udata block. restorestate accepts a udata block created by savestate and makes the appropriate restorations. For example,

```
(save–excursion (f))
```

in MLisp is equivalent to

```
state := savestate()
f()
restorestate(state)
```

in Icon_i. While this construction is not as elegant as that provided by MLisp, it has more power because the saved states are data objects and can be manipulated.

MLisp has two other functions of the same nature as save–excursion: save–window–excursion and save–restriction. Neither of these have been implemented in Icon_i.

Initiation and Termination of Icon_i

The modifications described constitute Icon_i as a programming system, but further modifications were necessary to make Icon_i suitable for incorporation in Emacs.

Icon_i operates as a subsystem of Emacs. When Emacs determines that an Icon procedure is to be executed, it passes control to Icon to execute the procedure and expects to eventually regain control. In standard Icon, when a run-time error is encountered, the system prints an appropriate error message and terminates execution. Because Emacs is a programming environment, it is not appropriate for a programming error to cause termination of Emacs. Rather, an Icon error should be handled like an error detected by a subroutine. The variant of Icon_i incorporated in Emacs contains modifications that allow the Icon system to operate much like a subroutine.

There are two C subroutines in this variant system that Emacs uses as an interface to Icon. An Emacs start-up routine does one-time initialization of the Icon subsystem. A second routine, called iconstart, is used to actually execute an Icon procedure. iconstart has the form

```
iconstart(number–of–arguments, argument–list–location, result–location)
```

argument–list–location is the address of the low word in the Icon argument list that is to be used to invoke the Icon procedure. The procedure to invoke is determined by the contents of the zero'th argument in the list. *result–location* is the address of a descriptor that is to hold the return value (if any) of the procedure. In addition, iconstart returns a character string representing the *image* of the value returned by the proceduure.

An Icon procedure can return a result, fail, or produce a run-time error. In Icon, a result returned by a procedure is taken to be the value of the procedure invocation, failure is transmitted to the enclosing expression, and a run-time error causes termination of the program.

When an Icon procedure is called from MLisp, an argument list is built from the MLisp-supplied values and iconstart is called, passing control to the Icon subsystem. The simplest case is when the Icon procedure calls other Icon procedures and ultimately returns a value. In this case, the descriptor pointed to by *result–location* in the call to iconstart is made to reference the return value. The value is then converted to an MLisp value. If the value is of a type that does not appear in MLisp, the string image of the value is used instead.

If an Icon procedure fails, iconstart returns a value of −1 rather than a pointer to a character string image of the result. This value is then mapped to the string "failure".

In the event of an Icon run-time error, the stack is cleared back to the call of iconstart and iconstart returns −2. In addition, the appropriate error message is presented to the user.

Note that MLisp and Icon functions can call each other in a completely transparent fashion (with the exceptions noted above). Thus, the stack may contain interleaved segments of Icon and MLisp frames, which presents an interesting garbage collection problem.

The Icon garbage collector splits the stack into two logical segments, one that contains Icon stack frames and another that contains C stack frames. The portion with Icon stack frames begins with the frame for the main procedure and grows until a system operation, such as the call of a built-in function, occurs. When a built-in function is called, a global variable known as the boundary is set to point to the frame of the C function corresponding to the built-in function. The execution of the C code for the function may cause a garbage collection. If this happens, the Icon portion of the stack, which lies between the boundary and the frame for the main procedure, is *swept* to mark descriptors on the stack. The portion of the stack containing C stack frames is ignored during garbage collection.

Consider a sequence of function calls:

MLisp → Icon → MLisp → Icon → Icon-built-in

where the Icon built-in function causes a garbage collection. The stack contains two Icon segments. Without any modifications to the garbage collector, only the last Icon segment would be swept and this is obviously unsatisfactory. Icon$_i$ solves this problem by maintaining a linked stack. If an Icon procedure is started with iconstart and the stack already contains an Icon segment, a link is made to the earlier stack segment. Subsequent recursive calls to iconstart merely add elements to this chain of Icon stack segments. The garbage collector starts with the most recent stack segment, sweeps it, and then follows the link to the previous stack segment, continuing until all segments have been swept.

Using Icon Procedures in Emacs

While the MLisp translator is embedded in Emacs, the Icon translator and linker remain disjoint, with only the Icon run-time system being part of Emacs. This yields a two-step process; the Icon translator and linker produce an icode file and this file is subsequently loaded into Emacs. As an example, consider a procedure to insert the string "Hello World!" at the end of the current text buffer:

```
procedure say_hello()
        end_of_file()
        insert_string("Hello World!")
end
```

Assuming that this procedure is in the file hello.icn, the command:

```
icont −e hello.icn
```

translates and links the source code and produces the icode file hello. (The −e switch directs the linker to produce an icode file that is suitable to be iloaded into Emacs.) This file can subsequently be loaded into Emacs using

```
(iload "hello")
```

after loading, say_hello can be executed as an extended command.

The Icon procedure iload_current_file translates and links the Icon source file corresponding to the current buffer and iloads the resulting icode file.