Programming on an Already Full Brain

Christopher Fry

Bugs happen because people create them when they can't remember the details needed to write correct programs. Why waste the user's scarce biological cache memory when plentiful computer memory performs better?

PROGRAMMERS OPTIMIZE THE INNER LOOPS OF their code because that's where the biggest performance gains are possible. Programming environments should do the same. The inner loop involves figuring out what to type and then typing it. Is it possible to create a programming environment that speeds both the figuring out and the typing?

Getting correct all the details of code demands lots of biological memory, frequently more than is available. But at any point in the typing of code, options are usually constrained by the legal constructs of the language, the functions and data available, and the context. Since the programmer's brain is already full of the details of the task and the algorithm, let's use the computer's brain instead. Emacs Menus is an adjunct to the Emacs text editor presenting the programmer with context-sensitive menus containing options that are at least syntactically and typewise legal for the code under the mouse. Emacs Menus, not a commercial product, was written by the author during the past few years in Boston.

Most debugging tools concentrate on finding and fixing relatively infrequent deeper bugs, such as errors in algorithms. But in practice, too much time goes to edit-compile-debug cycles for minor bugs, such as typos and "brainos," use of "kill" instead of "delete," or reversal of the order of two arguments to a function. By catching these problems in the editing stage, Emacs Menus speeds programming's inner loops.

The Limits of Biological Memory

Of the things humans do, there is, perhaps, no task more demanding of biological memory than programming. A common myth in our society is that the human brain has unlimited capacity for knowledge acquisition and storage. I don't buy it. We can store only so much at a particular time, and our ability to change the contents of that finite memory over time is, to be polite, limited. Overcoming that limitation is the strong suit of the computer. Humans are good compared to their silicon peripherals at abstract thought and higher-level knowledge organization. Computers are good at storing and searching large amounts of data. Emacs Menus helps each member of the programming team—a computer and a person—optimize what it, he, or she does best and interface smoothly with the other half of the team.

Programming involves design as well as writing. The continuum between concept and code can be organized into four major stages:

- Conceptual design
- Picking the right components to assemble
- Assembling the components
- Testing

Conceptual design takes place mainly in the programmer's head. Picking the components and understanding how to assemble them requires answering such questions as What components are there to choose from? and How do I put them together? Answering these questions often overwhelms the programmer's biological memory.

A modern programming language may have choices numbering in the hundreds or thousands. For humans fortunate enough to have occupations that allow them to program full time or blessed with an infallible memory for detail, answers to the related questions may pop to mind immediately. The rest of us tend to stick to a small set of familiar commands we are likely to use correctly and give up on the rest, regardless of their utility.

Understanding what parts are available and how to put them together is crucial. Since the need for this information is greatest just as the user is about to type, Emacs Menus builds an agent into the text editor that dynamically computes the available choices using knowledge of the programming language and knowledge of the interactive context. Help is available not in a Help window or in Help mode but through a pop-up menu at the point of insertion, exactly where the programmer needs it. Further, unlike most Help systems, Emacs Menus doesn't just tell users what they should do, it spares them the possibility of mistyping it.

Earlier Solutions

Until now, there have been three approaches to the problem of inputting programs. The classic one has been to type programs in an unrestricted text editor, just as the programmer might type a natural language document. The second is to have the system provide a text-based structure editor. The third is to use a special kind of structure editor that uses icons instead of text to represent the program source.

Text Editors (the nonsolution). Text editors give programmers so much rope it's difficult to use them to specify so simple a task as hanging yourself, let alone writing a valid piece of software. Programmers create a new document and are presented with a large blank space in which to enter just about any amount of garbage consisting of characters. They are free to create illegal syntax and nonsensical vocabulary and to add comments to the code with whatever they like, regardless of validity.

After a document is created in the text editor, the programmer then submits it to a compiler or interpreter, which checks the validity of the syntax and reports errors. If the program makes it through this step, it is executed and any runtime errors that arise are reported. Forced by errors back to the text editor, the programmer is faced with the cognitive task of associating error messages with the place in the code that caused them. Several errors may be reported at once, possibly due to either a single underlying cause or a series of independent causes, further confusing the matter. While modern programming environments try to minimize the overhead of context-switching in the edit-compile-debug cycle, this overhead is nonetheless significant even in the best of them and violates the principle of immediacy (see the article by Ungar et al. in this issue.)

Structure Editors and Forms-Based Languages (the bureaucratic solution). The structure-editing approach tries to retain the flexibility of text editing while adding knowledge about the programming language structure [10]. This way, the unambiguous structure of function calls or data-structure description is neatly delineated by a form that permits the programmer to enter text into a named slot. Free text editing is allowed only within the fields.

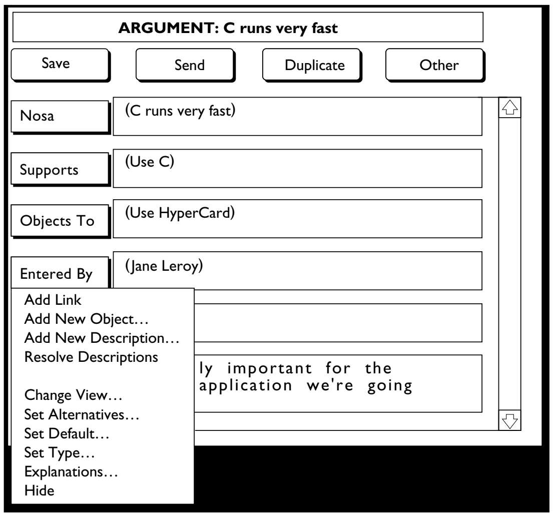
Although structure editors reduce the possibility of syntactic error, they lose the flexibility of free text editing. No longer can the text motion, searching, and editing commands be used indiscriminately on any visible text; needed instead are specific commands to move up and down the template structure. Because illegal syntax can't be entered, you can't change your code by passing through several illegal states (often necessary when you want to make something more than a trivial modification).

Structure editors also typically take advantage of only the static knowledge of the programming language syntax. As demonstrated by Emacs Menus, much more can be done by taking into account the dynamic context—exactly where the user is in typing the program, what has already been typed, what variables are available at the moment, and more.

There are two kinds of structure editors: One is embedded inside a text editor, and the templates are inserted as text; the other consists of forms-based programming, where the template is graphical.

While at MIT's Center for Coordination Science, I worked on a forms-based programming environment called Objects, Views, Agents, and Links (OVAL) [5], which provided clear online help for both writing and reading code (see Figure 1). Parsing for the compiler is trivial, so some checking can be performed on arguments and data-slot values by rather simple programs.

But forms-based programming suffers from severe space inefficiencies (as do the iconic languages discussed in the next section). The designer of the form can't predict the optimal size for an argument to take up. Dynamically varying the size of fields via scrolling or resizing the form consumes more operations in the user interface, more space for controls, or both. Embedding forms within forms is the logical strategy for depicting nesting, but the borders tend to take up an ever-larger amount of screen real estate.


ARGUMENT: C runs very fast

Nosa	(C runs very fast)
Supports	(Use C)
Objects To	(Use HyperCard)
Entered By	(Jane Leroy)

ly important for the application we're going

Figure 1. An OVAL form for specifying a data structure

Iconic Languages (the straightjacket solution). Iconic programming environments offer many benefits to memory-challenged humans [8], providing palettes of functions so programmers need not remember what functions exist or how each is spelled. They provide constrained ways to wire up functions so syntactically incorrect code can't be entered. Furthermore, the operations used to edit code can be as intuitive as dragging a wire between an output and an input node using the mouse. But for large complex programs, the iconic cure is often worse than the freedom-of-text disease.

As much as I love graphics, words are more spatially efficient. One word is worth a thousand pixels of icon, yet takes up less screen real estate. Iconic programs can't display as large a section of the program at once. The smaller the piece of a program the programmer can see at once, the more difficult it is to mentally piece the whole thing together, causing yet another biological memory problem. In addition, trying to remember the meanings of a large set of small icons, humans run into even more memory problems. Marian Petre [7] discovered that a graphic representation was so difficult to understand, people tended to use the textual representation to help them understand the graphical one rather than the other way around.

Visual programming also confronts the programmer with having to spatially lay out the icons. Unfortunately, programming involves constant revision. Sometimes the programmer wants to add 10 times as much code in a particular area than the original layout left room for but must spread everything out to make more space. The original layout work is wasted. Automatic layout programs are available but are tricky to get right and often don't convey the "semantic proximity" the programmer intended. Are we doomed to use tools no better than a text editor?

Emacs Menus

It should come as no surprise that I chose rather conventional text as the medium for displaying code. Text is the most flexible and the most dense medium. I'm all for careful pretty printing of the text, but that function is already provided in decent text editors.

Hackers familiar with text editors who switch to different kinds of programming environments often lament the lack of their favorite features in their favorite text editor. The first characteristic of a programming tool I wanted to achieve was to be no worse than an existing text editor. So I implemented Emacs Menus on top of an Emacs-like text editor. All commands in the underlying text editor are present. Also, I did not want to waste permanent screen real estate on displaying palettes of commands, as in many iconic environments. In fact, unless you are aware of the enhancements, you may be unable to detect that a powerful tool awaits under your mouse.

Whenever the user is in the text editor, holding down the mouse brings up a pop-up menu after a short delay. The delay is provided so a quick click simply repositions the text cursor without bringing up the menu. Holding down the mouse requests help from the system in the form of the question What might I do next? It is important that the menu appear right at the point of insertion—where the user's attention is focused—not elsewhere on the screen.

The contents of the menu (see Figure 2) appearing at any moment is highly dependent on the textual context. Emacs Menus analyzes the surrounding code and displays only the choices that make sense for the current context. Since the user is faced with a large number of possible choices, the pop-up menu presents a large number of items, but the categorization of items and special 4D menu navigation features make these large menus less scary than one might initially expect.

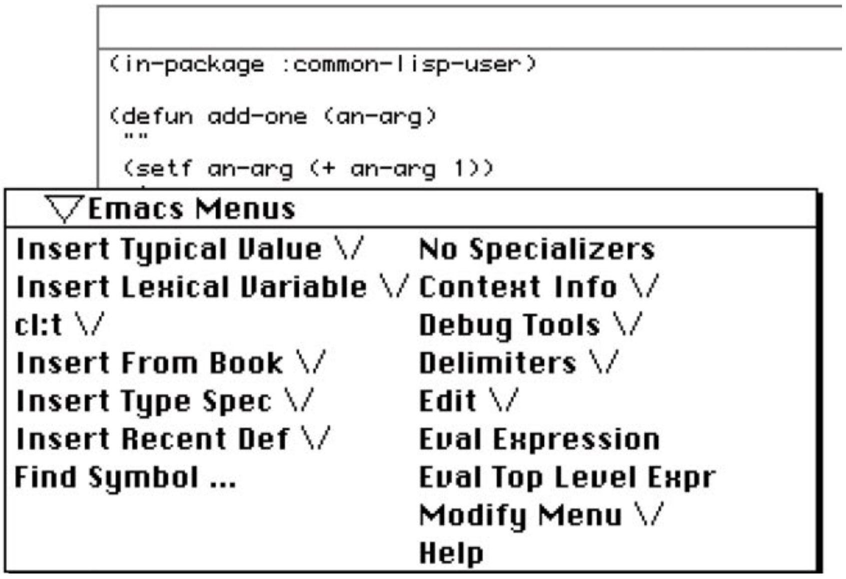
Just-In-Time Help

The menu that pops up contains items and submenus supplying:

- Knowledge about what was just clicked on
- Code to insert
- Other editing and debugging operations

Emacs Menus and Common Lisp Code. I built a database of information on the global variables and functions in Common Lisp. Each variable is associated with the type of values it can hold as well as with several typical values, one designated as the default value. The database entry for each function contains the type of the result, as well as the name, type, typical values, and a default value for each parameter. (I use *parameter* to mean the description of the kind of value that can be passed to a function, and *argument* to mean a particular value passed in a particular function call.) All parameters to a Lisp function are not necessarily required. The Emacs Menus database also stores the kind of parameter, whether optional, keyword, or "&rest" (meaning any number of arguments can be passed).

A typical mouse click is on an argument to a function. Emacs Menus' inside-out parser figures out its corresponding function and parameter, looks it up in



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
 ""
 (setf an-arg (+ an-arg 1))
```

Figure 2. Emacs Menus top-level menu

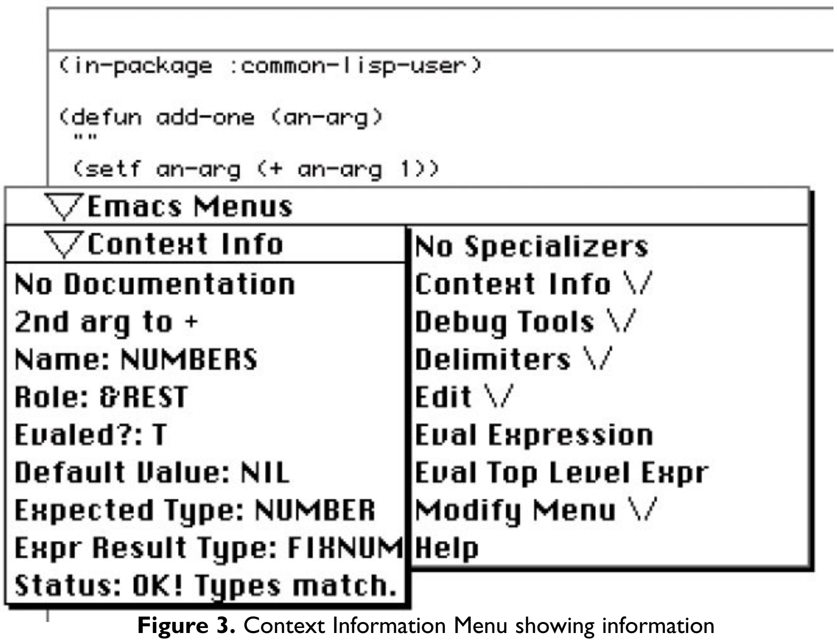
the Emacs Menus Common Lisp database, and constructs, on the fly, a submenu containing items for information purposes only. The items are:

- The name of the enclosing function call and rank of the argument, such as `2nd arg to +`
- The kind of the parameter, such as required or optional
- The type of the parameter, such as number and sequence
- The type of the actual argument being passed, such as integer or string
- Whether or not the type of the argument matches (is the same as or a subtype of) the parameter type

Emacs Menus can infer the argument type by knowing the result type of arguments that are function calls, the type of Common Lisp global variables, or the type of literals (see Figure 3). For lexical variables and some other constructs, Emacs Menus can't infer the type. Such type-checking helps catch a lot of errors that in a weakly typed language, such as Lisp, might not be caught until runtime. In a strongly typed language, such as Java, the Emacs Menus' early-warning system would catch the errors before the compiler is invoked.

Inside-Out Parsing

To escape the straightjacket of iconic programming and the regimentation of forms-based programming, the user must be able to create illegal code fragments. Maintaining an accurate model of what all the text in a buffer represents is therefore difficult at best. Emacs Menus doesn't do it. However, Emacs Menus does include detailed information about the code under the mouse. Every time the programmer clicks and holds on some code, a parse of the underlying text is started. Most parsers start at the beginning and move sequentially through the text. Theoretically, Emacs Menus could start at the beginning of the buffer, but pausing at the beginning of the buffer would take too long, especially in the face of illegal code between the beginning of the buffer and the mouse position. The Emacs Menus user interface requires that a fraction of a second after the user clicks, a context-specific menu of information and commands pops up. The parser starts where the user clicked and parses outward. It stops when it has captured just enough code to identify the current statement and its enclosing form, usually which argument of which function call.



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
 ""
 (setf an-arg (+ an-arg 1))
```

Figure 3. Context Information Menu showing information about the `2nd argument to +`

Coding via Mouse

The sooner the programmer catches errors in the coding process the better. One reason this rule is true is that the programmer has already built up a mental model of the current code, and creating that model takes time. The Information menu in Emacs Menus helps catch such errors as "wrong number of arguments to a function" or "wrong type of an argument" before compiling or runtime but after they've been entered.

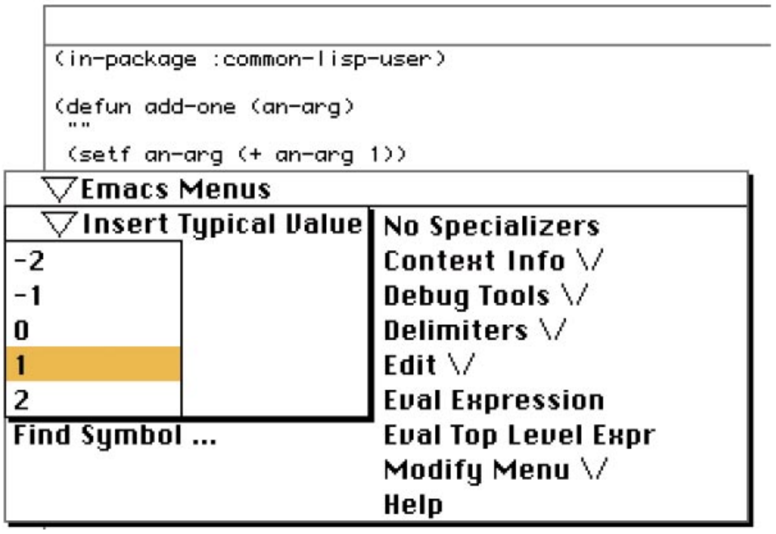
Can we catch errors even sooner? The Information menu (see Figure 3) delivers just-in-time help on an argument before the programmer enters it. Say you type in a function call to the Common Lisp function `elt`, which returns the *n*th element of a list, but you wonder whether it's the first or the second parameter that accepts the sequence in question. You can use the Information menu to tell the name and type of the first argument to `elt`. You'll find the first argument is a sequence and that you can type in your sequence. That's a lot faster than looking up the function in the documentation.

Can we do better? Another submenu of the main Emacs Menus menu contains typical values for the place under the mouse. In the case of the `2nd argument to +` where you must pass a number, you'd see menu elements for `-2`, `-1`, `0`, `1`, and `2`, giving you examples of the kinds of values that can go there (see Figure 4). A correct set of examples not only makes a strong hint as to the type of value that can be used but tells you the syntax of those values.

Can we do even better? In the case of such functions as the `2nd argument to +`, certain values are common. Several small integers qualify. If you select an item from the Typical Values menu, that item is inserted into the program text at the place of the mouse. This automatic insertion spares the programmer having to type in the code, but more important, it eliminates the risk of making a mistake that forces you to go through a compile-debug-edit cycle. For parameters of type `sequence`, the menu of typical values might contain the empty string " ", the empty list ( ), and the empty array # [ ]. A user can pick a value to insert, then edit the value to contain exactly what is wanted. Emacs Menus can't read a programmer's mind, but at least it helps programmers get the basic syntax right.

Typing in unbalanced delimiters is a common bug in

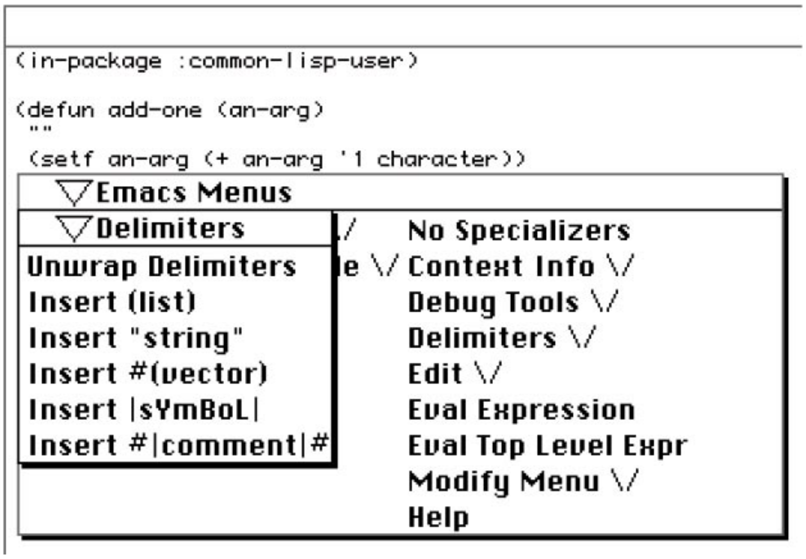
Lisp coding that need never occur if you use Emacs Menus (see Figure 5). I found eliminating this source of bugs so compelling, I added a special menu containing just delimiters. Common Lisp multiline-comment delimiters are particularly error prone. Inserting delimiters on white space inserts just an empty pair. But if the mouse is over an expression and the programmer chooses `insert`, the delimiters are automatically wrapped around the text in question.



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
 ""
 (setf an-arg (+ an-arg 1))
```

Figure 4. Typical Values menu for an `argument to +`



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
 ""
 (setf an-arg (+ an-arg '1 character))
```

Figure 5. Delimiters menu

Inserting Lexical Variables. Sometimes the code wanted for an argument is not a literal but the name of a lexical variable (also known as a local variable, such as the names of function parameters). Another submenu off the main Emacs Menus menu contains a list of all the legal lexical variables, or all the variables "in scope," at that point in the program. Thus, users need type in only the name of each variable once (to declare it). From then on, they can pick it from a menu, preventing inconsistent spellings. Also, since the menu contains only variables legal at that point, they never get an "undeclared variable" error. Unfortunately, Emacs Menus doesn't yet infer the types of lexical variables, so users can still insert a typewise incorrect variable. Fixing this limitation is a natural extension to Emacs Menus.

Inserting Function Calls. Because complex code frequently involves deep nesting of function calls, programmers need easy ways of inserting function calls. When a programmer picks a function call from an Emacs Menus Insert menu, a call to the function is inserted, including both open and closed parentheses. The cursor is moved to the position of the first argument so the user is ready to pick it from a menu as well.

The real problem with functions is that there are so many to choose from. Common Lisp contains about a thousand. It's nice having all that functionality in the language, but wading through the functions to find what is wanted is a challenge even for seasoned hackers. Emacs Menus to the rescue.

The first trick for Emacs Menus is to figure out the functions the user is likely to want to call, depending on context, then stick them on a menu. A few special cases can go a long way; one common context is at top level within a file. When the user clicks down on white space, not in another construct, the typical values for that context are forms for defining functions, macros, and global variables. Emacs Menus' typical values for the top-level context menu also contain the package declaration form and a call to load a file (see Figure 6). As a group, these comprise the vast majority of top-level forms in most programs.

As it turns out, function calls at the top level *within* a function definition also fall into a relatively constrained pattern. Declaring and setting local variables and a few control structures, such as `if`, `cond`, and `do` loops, make up a significant fraction of many top-level calls in a definition. Starting off with these two contexts, programmers can use Emacs Menus' typical values for a fair fraction of code. Typical values may include function calls themselves, indicating common idioms.

However, these contexts don't cover many important cases. For example, adding the thousand or so Common Lisp functions to the functions for a window system and a few other utility packages yields several thousand possibilities. All on one menu?

4D Menus

A linear 1D menu with more than 20 or 30 items is cumbersome. We can create a second dimension using hierarchical menus. That helps for a few hundred items at best. Do we have to remember what to type relying on our already full brain? This problem prompted me to invent a new kind of menu with two more dimensions—the horizontal and "pages."

I've found three or four columns of 20 items each is a reasonable amount of information for programmers to navigate. "Pages" of these multi-columned menus represent a fourth dimension. Moving the mouse outside the menu, then jiggling it slightly to the right or left allows the user to quickly flip the 2D pages.

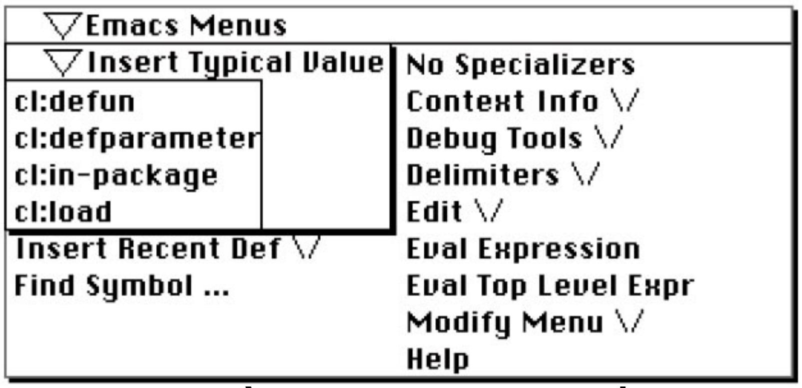


Figure 6. Top-level Insert Function Call menu

Just as humans with good spatial memory can remember, say, some blob in the lower right-hand corner of a page in a book and flip the pages to find that page, Emacs Menus users can relocate a menu item they've been to before. I used color as an additional clue to help users recognize a page and as an indicator of how deep they are in a page stack.

Using pages with, say, 50 items each and page stacks five deep gives users 250 items without scrolling. Adding hierarchical submenus makes it easy to get up to several thousand items on one 4D menu, all without scrolling.

Menu Organization. Anything we can do to help the user select the correct function the first time saves debugging time down the road. Being able to put thousands of items on a menu is nice, but without proper organization, finding what is wanted can be time consuming. I concluded that programmers needed more than one organization of the Common Lisp functions, because they want to look them up in different indexes.

Alphabetical ordering is an obvious organization, although it is not part of the current implementation. Having the front page of the menu contain all the starting characters of functions (with each item being hierarchical) is an easy way to take advantage of 4D menus.

My primary organization is by returned type (see Figure 7). All the functions that return strings are under one subtree, all the functions that return numbers under another, and so on. This organization is especially convenient when users are about to insert an argument to a function call for which they know the type of the argument and the general functionality they are after but have no idea how to spell the function name.

I found this organization wasn't always what I needed. Sometimes I didn't really care about the returned type of the function but just needed to scan the functions that manipulate strings. Some of these return non-strings, so users can't simply look under the String subtype menu to find them. I built a menu organized according to the chapter titles of the book *Common Lisp: The Language* [9] in which each function is documented. This organization proved to be a valuable way to capture semantic neighborhood.

Even with such structuring, some of the submenus in Emacs Menus still include a confusing number of items. The horrible thing about large dictionaries and large programming languages is that looking for a common word takes much longer than looking for a common word in a smaller language. So I've split large categories into common and uncommon parts. Users who first scan the common list and fail to find what they want can delve into a submenu of the less frequently used functions. This organization is especially useful for beginners, since it eliminates the primary disadvantage of large languages.

Commands

Programming is not just inserting text. Text has to be edited, even when syntactically correct. The programmer would also like to invoke a number of debugging tools that take code fragments as arguments, such as steppers [4]. Emacs Menus has an extensive Tools menu (see Figure 8) that simplifies the invocation of tools. I briefly describe just one—Find Symbols—which is so important I moved it to the top-level menu.

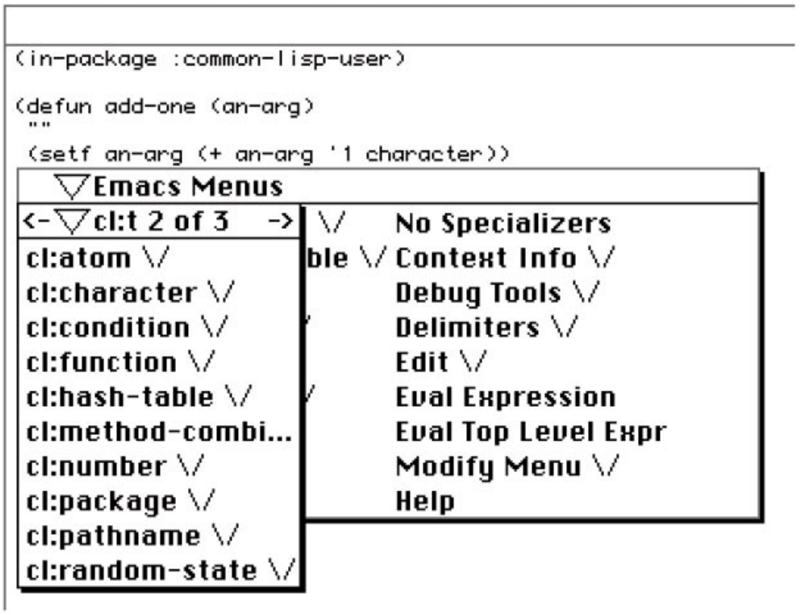
Lisp has long had a function called Apropos, which takes an argument of a string and prints out all the Lisp functions and global variables whose names contain that substring. The Find Symbols dialog I developed for Emacs Menus gives many more ways to filter the possible functions than traditional Apropos (see Figure 9). Consider it an interactive way of constructing custom menus of functions in case the organizations using 4D menus are not convenient enough. With a few mouse clicks, the Find Symbols dialog enables users to filter down the set of interesting functions to an easily scannable 10–30, then allows them to get documentation on each, and, after finding the right one, click on it to insert it into a program. Since Find Symbols can be invoked on some selected text already in an editor buffer, the programmer doesn't need to type the initial substring as required to invoke conventional Apropos.

Few things are more frustrating than trying to fix a bug and inadvertently introducing a second bug by misusing a debugging tool. If Emacs Menus fails to prevent the first kind of bug, at least it prevents the second.

Language Design and Hardware

Although the problems and the solutions I provide for them are applicable to just about any textual programming language, the particulars of the syntax make a difference. The more inconsistent and complex a language, the more the programmer needs help and the more difficult it is to write the code to provide that help. Elegant syntaxes, such as Lisp, are easiest for logical humans and programs to parse. Languages, such as C and Java, that combine infix and prefix syntax make life more difficult for both the user and the tool-builder. Tools like Emacs Menus can reduce but not eliminate the problems caused by unnecessarily complex syntax.

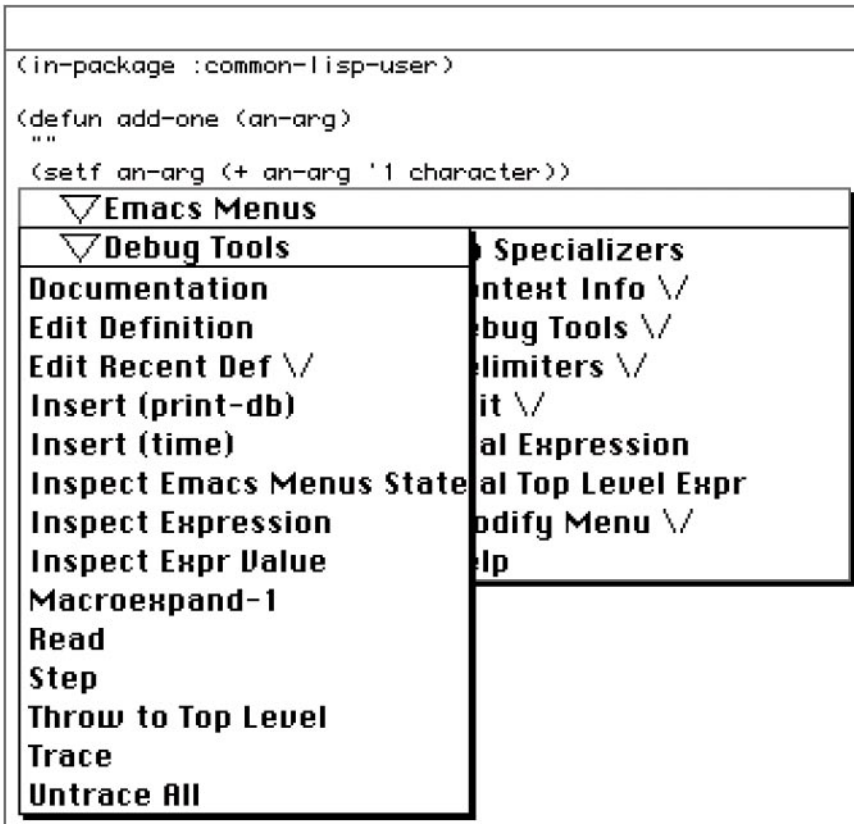
Emacs Menus is software that can benefit from input interface hardware designed with humans in mind.



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
 ""
 (setf an-arg (+ an-arg '1 character))
```

Figure 7. Users may know the type of the value they want, but not how to spell the name of the function. Items in this menu are types that contain submenus of functions that return a value of that type.



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
 ""
 (setf an-arg (+ an-arg '1 character))
```

Figure 8. Debug Tools menu

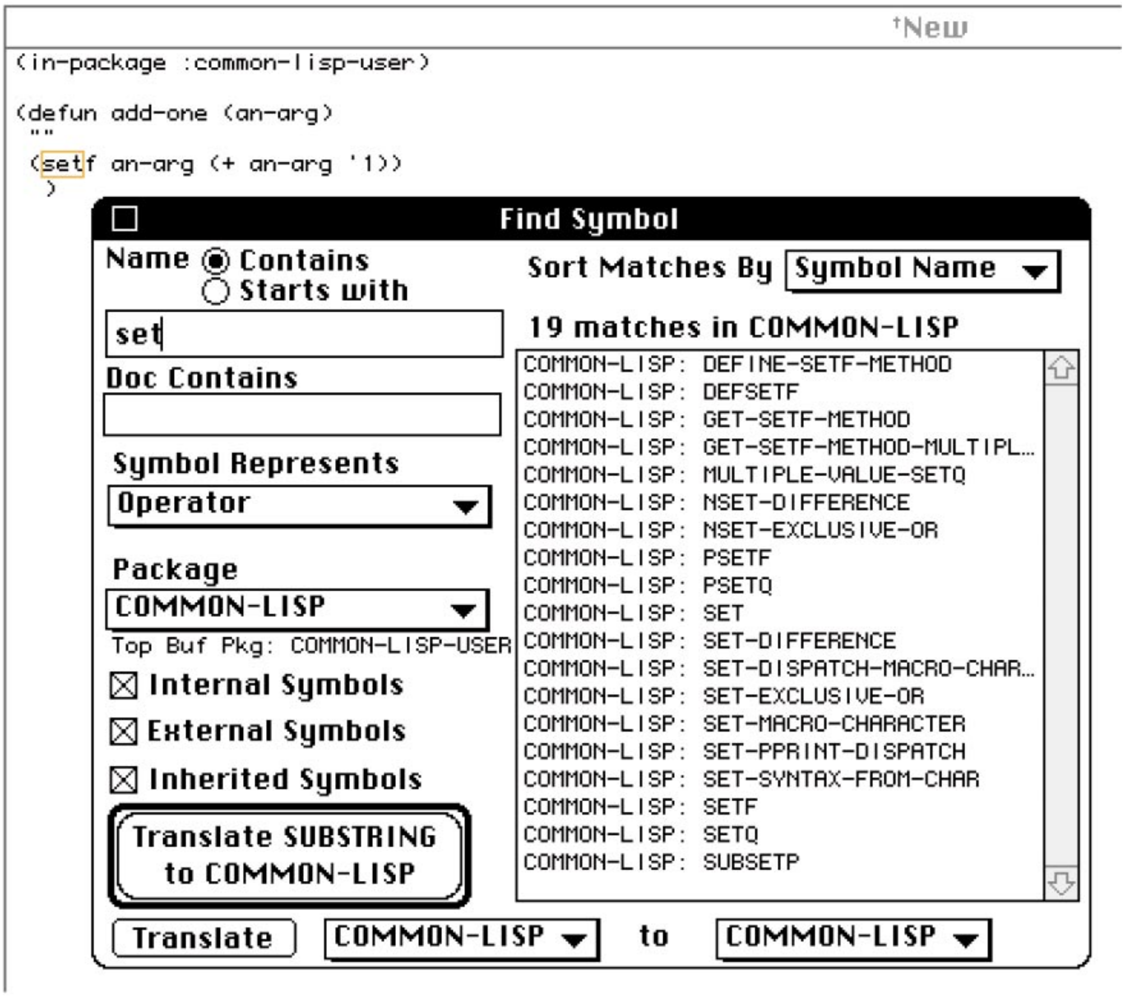
Switching between mouse and keyboard is bad. Most hackers I know think in terms of keyboard commands that perform equivalent mouse operations, so they don't have to switch to and from the mouse. I think in the opposite direction, so the programmer doesn't have to switch to and from the keyboard. Emacs Menus provides most operations via mouse input, although each new identifier must be entered at least once.

Humans would benefit greatly from a mouse with a button for each finger. Having to hold down a keyboard key in combination with a mouse-click should convince any user how badly the designers of one-, two-, and three-button mice blew it. For example, allowing scrolling via mouse would eliminate the need for scroll bars taking up lots of screen real estate. Putting the default action on a button eliminates the coordination needed for double-clicking. Having Help and an Additional Operations menu each on its own mouse button would spare users from moving up to the menu bar for operations.

The pen is mightier than the mouse and can be even more convenient for entering small amounts of text [3]. Character recognition is becoming quite reliable, especially with stylized characters, such as those used in the Graffiti software product [6]. Onscreen keyboards are potentially even faster than Graffiti, especially if well laid out (as shown by the Instant Text software product [11]).

When designing Emacs Menus, I wanted a pen-based computer. A tablet is inadequate; better is a screen you can write on. Getting five buttons on a pen is difficult, but a pen would give not only higher spatial resolution but the additional dimensions of pen angle and pressure that make up for the one or two buttons that fit comfortably on a pen. Speech recognition would be another advantage; so would two pointing devices, one for each hand [1].

Throwing out the keyboard and using input devices designed to fit the human body would finally permit the smooth integration of hardware and software. Under such an arrangement, tools like Emacs Menus could really shine, because the hardware wouldn't get between users and their bugs.



```
(in-package :common-lisp-user)

(defun add-one (an-arg)
  ""
  (setf an-arg (+ an-arg '1))
  )
```

Name: [✓ Contains] Starts with
set
Sort Matches By: Symbol Name
Symbol Represents: Operator
Package: COMMON-LISP
Top Buf Pkg: COMMON-LISP-USER
☒ Internal Symbols
☒ External Symbols
☒ Inherited Symbols
Translate COMMON-LISP to COMMON-LISP

```
19 matches in COMMON-LISP
COMMON-LISP: DEFINE-SETF-METHOD
COMMON-LISP: DEFSETF
COMMON-LISP: GET-SETF-METHOD
COMMON-LISP: GET-SETF-METHOD-MULTIPL…
COMMON-LISP: MULTIPLE-VALUE-SETQ
COMMON-LISP: NSET-DIFFERENCE
COMMON-LISP: NSET-EXCLUSIVE-OR
COMMON-LISP: PSETF
COMMON-LISP: PSETQ
COMMON-LISP: SET
COMMON-LISP: SET-DIFFERENCE
COMMON-LISP: SET-DISPATCH-MACRO-CHAR…
COMMON-LISP: SET-EXCLUSIVE-OR
COMMON-LISP: SET-MACRO-CHARACTER
COMMON-LISP: SET-PPRINT-DISPATCH
COMMON-LISP: SET-SYNTAX-FROM-CHAR
COMMON-LISP: SETF
COMMON-LISP: SETQ
COMMON-LISP: SUBSETP
```

Figure 9. Find Symbols dialog

Programming Pain

For many hackers, debugging involves both mental and physical pain. Hackers hack long hours. Repetitive strain injury (RSI)—the hacker's disease—is an occupational hazard. Emacs Menus allows users to perform most operations via mouse. Since it is trivial to move between keyboard and mouse, users can pick whichever input device is less painful at the moment, making their movements less, well, repetitive. Furthermore, the mouse, particularly the one-button variety on Macintoshes, has many variations and can be used with either hand. The mouse can also be augmented with head trackers, eye-trackers, trackballs, trackpads, the IBM Trackpoint, or devices operated via feet or voice. Emacs Menus allows programmers to distribute their physical stresses across a variety of input mechanisms.

No less dangerous than RSI of the hand is RSI of the brain. Programming environments requiring repetitive, low-level, mind-numbing operations to get anything done are a prime cause of programmer burnout.

Status and Future Modifications

Emacs Menus was implemented on the Macintosh using Macintosh Common Lisp [2] by the author with lots of assistance from its own earlier versions. What good is a tool that can't help make itself better?

The Common Lisp database to support Emacs Menus is not fully implemented. Although support is included for enough functions to test the basic ideas, support for many more Lisp functions is needed before Emacs Menus lives up to its promise for coding Common Lisp. However, it is not yet used beyond developing itself and could benefit greatly from user studies.

Conclusions

All programming environments involve trade-offs. The main one with Emacs Menus was to design a system that didn't make it easy to make mistakes (like a text editor) yet made it easy to do whatever the programmer wanted (like a text editor). The proper balance was achieved by augmenting a text editor with a giant context-sensitive 4D menu that gives users just-in-time help, along with the ability to insert textual code without typing and access to appropriate tools—all with minimal cognitive overhead.

Unlike traditional hierarchical menus, 4D menus have the power to provide thousands of operations without requiring the biological memory that would otherwise be filled with thousands of words. Regardless of the capacity of a user's human memory system, when it's full, it needs help. C

ACKNOWLEDGMENTS

These ideas were not developed in a vacuum but in Cambridge, Mass., with the help of:

- The people at Coral Software I worked with to develop the first version of Macintosh Common Lisp
- David Levitt, who roped me into cofounding Hip Software, a creative 1980s startup doing iconic programming environments that went nowhere financially
- Tom Malone of MIT's Center for Coordination Science for the vision of a generalized Information Lens employing forms-based programming
- Jo Marks of Harlequin Inc. for wisdom seldom found among software developers
- Mike Plusch of PowerScout Corp. for creativity broad and deep
- Henry Lieberman of MIT's Media Lab for being the best comrade in arms I can imagine during our long battle for innovation in programming environments

REFERENCES

1. Buxton, B. The natural language of interaction. In *The Art of Human-Computer Interface Design*, B. Laurel, ed. Addison-Wesley, Reading, Mass., 1990.
2. Digitool. *Macintosh Common Lisp*. User manual and language. Digitool, Cambridge, Mass., 1996.
3. Goldberg, D., and Richardson, C. Touch typing with a stylus. In *Proceedings of the Conference on Human Factors in Computing Systems (INTERCHI)* (Amsterdam, Apr.). ACM Press, New York, 1993, pp. 80–87.
4. Lieberman, H., and Fry, C. ZStep 95, a reversible, animated source code stepper. In *Software Visualization: Programming as a Multimedia Experience*, J. Domingue, J. Stasko, M. Brown, and B. Price, eds. MIT Press, Cambridge, Mass., 1997.
5 Malone, T., Lai, K.-Y., and Fry, C. Experiments with OVAL: A radically tailorable tool for cooperative work. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*. (Toronto, Canada, Oct. 31–Nov. 4). ACM Press, New York, 1992, pp. 289–297.
6. Palm Computing Inc. *Graffiti*. Software product. Los Altos, Calif. 1995.
7. Petre, M., Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM 38*, 6 (June 1995), 33–44.
8. Shu, N. *Visual Programming*. Van Nostrand Reinhold Co., New York, 1988.
9. Steele, G.L., Jr. *Common Lisp: The Language*. Digital Press, Maynard, Mass., 1990.
10. Szwillus, G., and Neal, L. *Structure-Based Editors and Environments*. Academic Press, 1996.
11. Textware Solutions. *Instant Text*. Software product, Burlington, Mass, 1995.

CHRISTOPHER FRY (cfry@shore.net) is the chief technical officer of PowerScout Corp., a developer of agent-based browsers for the Web, in Boston.