GNU Emacs Lisp Bytecode Reference Manual

Collated and edited by Rocky Bernstein with contibutions from Pipcet, Chris Wellons, Stefan Monnier, Hallvard Breien Furuseth, Vladimir Kazanov, Andrea Corallo, and the Emacs community.

Short Contents

Table of Contents

1 Introduction to Emacs Lisp Byte Code and LAP

1.1 Why is Emacs Lisp Bytecode Important and How is Emacs as a Program Different?

If we were to compare two similar complex programs in around 2018, Firefox 53.0.3 and Emacs 25.3, we would see that the Firefox tarball is 5 times bigger than the Emacs tarball. How are these made up, and what languages are they comprised of?

For Firefox whose core is written in C++ we have:

```
$ cloc --match-f='\.(js|c|cpp|html|py|css)$' firefox-53.0.3
   89156 text files.
   86240 unique files.
    1512 files ignored.

cloc v 1.60  T=244.20 s (353.2 files/s, 56012.8 lines/s)
-------------------------------------------------------------
Language                     files     comment           code
-------------------------------------------------------------
C++                           7267      418019        3057110
Javascript                   25855      532629        2859451
HTML                         45311      120520        2209067
C                             3482      400594        1664666
```

And for Emacs whose core is written in C we have:

```
$ cloc emacs-25.3.tar.xz
    3346 text files.
    3251 unique files.
    1130 files ignored.

cloc 1.60  T=13.85 s (160.1 files/s, 154670.7 lines/s)
-------------------------------------------------------------------
Language                     files     comment           code
-------------------------------------------------------------------
Lisp                          1616      200820        1270511
C                              255       66169         256314
C/C++ Header                   176       11505          34891
```

If we look at the relative ratio of C++ to Javascript code in Firefox, and the ratio of C versus Lisp code in Emacs, we see that much more of Emacs is written in Lisp than Firefox is written in Javascript. (And a lot of C code for Emacs looks like Lisp written using C syntax).

My take is that Emacs a lot more orthogonal in its basic concepts and construction. Just as Leibniz was amazed that such diversity could come out of such simple rules of mathematics and physics, so it is remarkable that something as complex as Emacs can come out of the relatively simple language, Lisp.

1.2 Emacs Lisp Bytecode and LAP

However pervasive Emacs Lisp is in the Emacs ecosystem, Emacs Lisp is not and never has been a speedy language compared to say, C, C++, Go, Rust or Java. And that's where LAP and bytecode come in.

As stated in a commment in `byte-opt.el` added from Lucid Emacs circa 1992:[1]

> No matter how hard you try, you can't make a racehorse out of a pig.
>
> You can, however, make a faster pig.
>
> —*Jamie Zawinski*

Emacs Lisp bytecode is the custom lower-level language used by the Emacs bytecode interpreter. As with all bytecode, its instructions are compact. For display purposes, there is a `disassemble` command that unpacks the fields of the instruction. With this and the constants vector, bytecode can be printed in an assembly language-like format.

I'll often use an Emacs Lisp bytecode instruction to refer to an assembly representation of the instruction.

LAP stands for Lisp Assembly Program. It is an internal representation of the bytecode instructions in a more symbolic form. It is used behind the scenes to make bytecode more amenable to optimization, since the instructions are in a structure which is easier to operate on.

If we want to write the instruction sequence in this symbolic form rather than give a byte-encoded form, we can do that using the function `byte-compile-lapcode`.

Example showing use of `byte-compile-lapcode`

```
(defalias 'get-foo
  (make-byte code
    #x000                   ;; lexical parameter counts
    (byte-compile-lapcode
      '((byte-varref . 0)
        (byte-return)))     ;; instruction sequence
    [foo]                   ;; constants vector
    1))                     ;; max stack usage
```

Silly Loop Example (`https://www.gnu.org/software/emacs/manual/html_node/elisp/Speed-of-Byte_002dCode.html`) in the Emacs Lisp Manual gives a program to time running in some code in the bytecode interpreter versus running the code in the Lisp interpreter. When I ran this program, bytecode ran 2.5 times faster[2]. The Emacs Lisp manual gets a speed improvement of about 3 times.

---

[1] This likely an adaptation of dialogue from "East of Eden by John Steinbeck" (`https://www.goodreads.com/quotes/7745830-you-can-t-make-a-race-horse-of-a-pig-no`). Jamie Zawinski (jwz) is responsible for its addition (and the bit about turbocharged VW bugs below it) but makes no claims to its origin.

[2] Code was compiled to use dynamic binding for variable access, as was probably the case in the Emacs Lisp manual. We should note that byte-compiling with lexical binding for variable access gives code that runs a bit faster than when dynamic binding is used.

1.3 Emacs Lisp Virtual Machine

The Emacs Lisp bytecode interpreter, like many bytecode interpreters such as Smalltalk, CPython, Forth, or PostScript, has an evaluation stack and a code stack. Emacs Lisp bytecode instructions use reverse Polish notation: operands appear prior to the operator. This is how many other bytecode interpreters work. It is the opposite of the way Lisp works. To add the values of two variables we might write `(+ a b)`. However in bytecode it is the other way around: the operator or function comes last. So the corresponding bytecode is:

```
0       varref    a
1       varref    b
2       plus
```

As in most language-specific virtual machines, but in contrast to a typical general-purpose virtual machine, the things that are on the evaluation stack are the same objects that are found in the system that they model. Here, these objects can include Emacs buffers, or font faces, Lisp objects like hashes or vectors, or simply (30-bit) Lisp integers. Compare this with, say, LLVM IR, or JVM instructions where the underlying objects on the stack are registers which can act as pointers, and the internal memory layout of objects is exposed.

Control flow in Lisp bytecode is similar to a conventional assembly language: there are unconditional and conditional jumps. More complex control structures are simply built out of these.

Although it may be obvious, one last thing to point out is that the Emacs Lisp bytecode instruction set is custom to Emacs. In addition to primitives that we would expect for Lisp such `car` and `cdr`, there are primitive bytecodes for more-complex Emacs editor-specific concepts such as "save-excursion"[1].

The interpreter is largely backward compatible, but not forward compatible (although eventually old Emacs Lisp bytecode instructions do die). So old versions of Emacs cannot necessarily run new bytecode. Each instruction is between 1 and 3 bytes. The first byte is the opcode and the second and third bytes are either a single operand or a single immediate value. Some operands are packed into the opcode byte.

[1] The semantic level difference between Emacs Lisp and its bytecode is not great, so writing a decompiler for it more feasible than if the bytecode language were of a general nature such as, say, LLVM IR.

1.4 Wither Bytecode - Its Future

Emacs's bytecode is pretty old—about as old as Emacs itself. And although there have been some changes to it, there has always been lurking in the background the question of whether it might be totally ditched, either as a by-product of switching out the underlying Lisp implementation for something else, or as a result of using JIT technology.

Let's take these two situations where Emacs Lisp Bytecode might become obsolete separately. Both ideas have been floating around for a long time.[1]

With respect to alternate programming-language implementations, there have been many languages that been proposed and experimented with. The big obstacle in totally replacing Emacs Lisp is in rewriting the huge current Emacs Lisp code base. (The counts given in the last section for Emacs 25.3 are 1.5K files and 100K lines of code.)

I think that if such an approach were to work, the language would have to be available as an additional language until the current code base was replaced. At present (circa 2018), alternate programming languages haven't gained much of a foothold; they are not in the current Emacs distribution or in any of its branches.

An obvious alternative language proposed is Common Lisp. Over time, an Emacs Lisp package implementing Common Lisp has been providing more and more Common Lisp functionality; names, however, are prefaced with `cl-`.

The addition of features in Common Lisp has been somewhat reflected in changes in the run-time systems, such as the addition of lexical scoping. And this approach partially solves the large code-base migration problem. But it also reduces the need to jump cold turkey from Emacs Lisp Bytecode to something else.

And what about the other possibility where Emacs incorporates JIT technology? The motivation for this is to speed up Emacs. There is widespread belief among the development community that there could be big performance wins if this were done right. After all, it is not uncommon for some people to live inside a single GNU Emacs session.

This idea of using a JIT to speed performance goes back over a decade, at least back to 2006. Of the JITs that have been proposed, at least four of them use Emacs Lisp Bytecode as the basis from which to JIT from. I think that is because Emacs Lisp Bytecode is a reasonable target to JIT: it is sufficiently low level, while also easy to hook a JIT into.

Two alternatives to Emacs Lisp Bytecode which have sophisticated JIT technology are LLVM IR and JVM IR. For each, the surrounding run-time environment would have to be replicated. Another IR possibility might be JavaScript IRs: specifically, the ones for V8 and Spidermonkey.

Pipcet's work that allows SpiderMonkey's garbage collector to be used in Emacs, allows for a real possibility of using SpiderMonkey's JIT with either JavaScript, Emacs Lisp bytecode, or Emacs Lisp bypassing Emacs Lisp bytecode. That last route I think might be harder. JIT'ing from Emacs Lisp bytecode to via SpiderMonkey (if it is possible) would allow for dual Emacs Lisp and JavaScript scripting while the other options don't.

In late 2019, Andrea Corallo proposed a prototype for adding native compilation to Emacs Lisp. The work is described in `https://zenodo.org/record/3736363#`.

[1] Tom Tromey says about the same thing in his FOSDEM 2020 talk `https://fosdem.org/2020/schedule/event/emacsthoughts/` or at least he seems to come up with similar conclusions.

`X_LeOFNKjMU`. The idea behind this is to use LAP as main input for the compilation, this is lowered into LIMPLE, an SSA based intermediate representation, where several compiler passes are used to perform a number of Lisp specific optimizations. Eventually LIMPLE is translated into libgccjit IR and libgccjit is used to plug into the GCC infrastructure and achieve native code generation.

The approach of a dedicated SSA based Intermediate Representation allow for advanced value/type inference within the code enabling effective optimizations as type check removal. Also Elisp is extended for allowing the programmer to provide compiler hints.

The so called Emacs Lisp native compiler (AKA gccemacs[2]) is capable of AOT as well as parallel asynchronous JIT compilation. In the later case byte compiled functions are used till the native compiled version is available and function hot swap is performed.

As of January 2021 this work is being integrated in Emacs and is planned to released as part of Emacs 28.

Needless to say, such a lot of work is involved in adding any sort of JIT technology; it will not totally replace Emacs Lisp Bytecode in the near future.

[2] Andrea Corallo gccemacs development blog. `http://akrl.sdf.org/gccemacs.html`

2 Emacs Lisp Bytecode Environment

In this chapter we discuss the ways Emacs creates, modifies and uses bytecode in order to run code. We describe a little of the two kinds of interpreters Emacs has, what goes into a bytecode file, and the inter-operability of bytecode between versions.

2.1 Emacs Lisp Bytecode Objects

This section is expanded and edited from Chris Wellons' blog on "Emacs byte code Internals" and from the Emacs Lisp Reference manual. See references at the end of this doc.

Emacs Lisp bytecode is an encoded form of a low-level assembly format that is suited to running Emacs Lisp primitives and functions.

Emacs Lisp bytecode is not a low-level sequence of octets (bytes) that requires a lot of additional special-purpose machinery to run. There is a custom C code interpreter to handle each of the instruction primitives, and that is basically it. And even here, many of the instructions are simply a bytecode form of some existing Emacs primitive function like "car" or "point".

Emacs Lisp bytecode is a built-in Emacs Lisp type (the same as a Lisp "cons" node, or a Lisp symbol).

Functions `aref` and `mapcar` can be used to extract the components of bytecode once it is built, The bytecode object is made up of other normal Emacs Lisp objects described next. Bytecode is created using the `make-byte-code` function.

One important component of the bytecode object is the "constants vector." It is a Emacs Lisp vector. The `constant` instruction refers to one of these objects.

An Emacs Lisp object of a bytecode type is analogous to an Emacs Lisp vector. As with a vector, elements are accessed in constant time.

The print syntax of this type is similar to vector syntax, except `#[...]` is displayed to display a bytecode literal instead of `[...]` as in a vector.

A bytecode object is one of the several kinds of functions that Emacs understands. See see [symbol-function], page 70, for other objects that act like a function.

Valid bytecode objects have 4 to 6 elements and each element has a particular structure elaborated on below.

There are two ways to create a bytecode object: using a bytecode object literal or with `make-byte-code` (see Section 2.6.18 [make-byte-code], page 25). Like vector literals, bytecode functions don't need to be quoted.

The elements of a bytecode function literal are:

1. Function Parameter (lambda) List
2. Bytecode Unibyte String
3. Constants Vector
4. Maximum Stack Usage
5. Docstring
6. "Interactive" Specification

2.1.1 Function Parameter (lambda) List

The first element of a bytecode-function literal is the parameter list for the `lambda`. The object takes on two different forms depending on whether the function is lexically or dynamically scoped. If the function is dynamically scoped, the argument list is a list and is exactly what appears in Lisp code. In this case, the arguments will be dynamically bound before executing the bytecode.

Example showing how a parameter list is transformed:

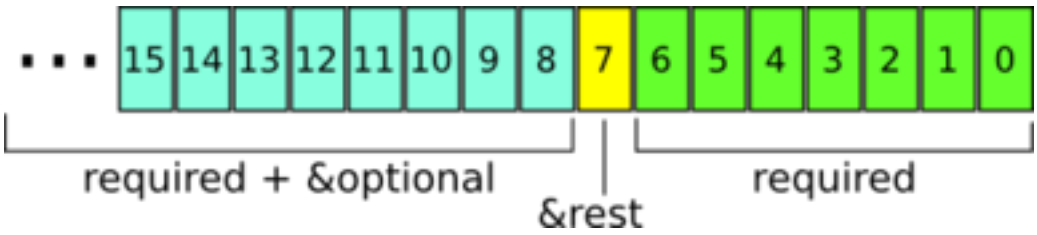
```
ELISP> (setq lexical-binding nil) ; force lexical binding
ELISP> (byte-compile
  (lambda (a b &optional c) 5))

#[(a b &optional c) "\300\207" [5] 1]
```

Above we show raw bytecode data. Emacs after version 25 makes an effort to hide the data.

There is really no shorter way to represent the parameter list because preserving the argument names is critical. With dynamic scoping, while the function body is being evaluated these variables are globally bound (eww!) to the function's arguments.

On the other hand, when the function is lexically scoped, the parameter list is packed into an Emacs Lisp integer, indicating the counts of the different kinds of parameters: required, `&optional`, and `&rest`. No variable names are needed. In contrast to dynamically-bound variables, the arguments are on the stack of the byte-code interpreter before executing the code

The following shows how parameter counts and flags are encoded:



The least significant 7 bits indicate the number of required arguments. This limits compiled, lexically-scoped functions to 127 required arguments. The 8th bit is the number of **&rest** arguments (up to 1). The remaining bits indicate the total number of optional and required arguments (not counting **&rest**). It's really easy to parse these in your head when viewed as hexadecimal because each portion almost always fits inside its own "digit."

Examples showing how lexical parameters are encoded:

```
ELISP> (byte-compile-make-args-desc '())

#x000  ;; (0 args, 0 rest, 0 required)

ELISP> (byte-compile-make-args-desc '(a b))

#x202  ;; (2 args, 0 rest, 2 required)

ELISP> (byte-compile-make-args-desc '(a b &optional c))

#x302  ;; (3 args, 0 rest, 2 required)

ELISP> (byte-compile-make-args-desc '(a b &optional c &rest d))
```

```
#x382  ;; (3 args, 1 rest, 2 required)
```

The names of the arguments do not matter in lexical scope; they're purely positional. This tighter argument specification is one of the reasons lexical scope is sometimes faster: the byte-code interpreter doesn't need to parse the entire lambda list and assign all of the variables on each function invocation; furthermore, variable access is via a compact index located usually in the operand value rather than an index into the constants vector followed by a lookup of the variable.

2.1.2 Bytecode Unibyte String

The second element of a bytecode-function literal is either

- a unibyte string, or
- a pointer to a unibyte string,
- An autoload function

A unibyte string is a sequence of bytes or octets. Despite the type name, it is not interpreted with any sort of Unicode encoding. These sequences should be created with `unibyte-string()` because strings can get transformed into longer sequences of bytes when encoded. To disambiguate the string type to the Lisp reader when higher values are present (> 127), the strings are printed in an escaped octal notation, keeping the string literal inside the ASCII character set.

Examples unibyte strings:

Bytecode for `(defun double-eg(n) (+ n n))` is:

```
PC  Byte  Instruction
 0    8   varref[0] n
 1  137   dup
 2   92   plus
 3  135   return

Constants Vector: [n]
```

To encode the byte sequence then for this we could use:

```
ELISP> (unibyte-string 8 137 92 135)

"^H^?\\\207"
```

It is unusual to see a bytecode string that doesn't end with 135 (#o207, return).

We describe how to decode the bytecode string in Section 3.1 [Instruction-Description Format], page 39.

However when a function has been defined as a result of reading a bytecode file, the unibyte string is a pointer into that file. This pointer is represented by a `cons` node where the `car` is the filename and the `cdr` is the bytecode offset from the beginning of the file

```
ELISP> (aref
        (symbol-function 'cl-gcd)
        1) ;; 1 is the bytecode string field
```

```
"("/tmp/emacs/lisp/emacs-lisp/cl-extra.elc" . 7352)
ELISP> (aref
        (symbol-function 'ediff-buffers)
        1)

(autoload "ediff" 975154 t nil)
```

2.1.3 Constants Vector

The third object in a bytecode-function literal is the "constants vector". It is a normal Emacs Lisp vector and can be created with `(vector ...)` or using a vector literal.

There is a possibility for confusion by the name "constants vector". The vector size and its values are indeed constant. Also, only the `constant` bytecode instructions (see Section 3.3 [Constants-Vector Retrieval Instructions], page 50) refers to one of these objects. However, in addition to values string and integer values that do not change, values in this vector also can be function and variable names. So although a variable or function *name* stored in the constants vector doesn't change, the *binding* of that particular variable or function can change, even in the course of running the bytecode.

By using a constants vector, operand sizes in the bytecode instructions are fixed and small. Also, operand values can be shared, reducing the size of the constant vector.

Since the constants vector is a true Emacs Lisp vector, the overall bytecode interpreter is simpler: all Lisp objects are handled in a unified way: the representation of a integers, vectors, lists, strings, and other Lisp objects is no different from the representation in the Emacs Lisp interpreter.

Example Showing a Constants Vector:

```
ELISP> (aref
        (byte-compile
          (lambda (a b)
           (my-func '("hi" "there") a nil 5)))
         2) ;; 2 is the bytecode constants field

[a my-func
 ("hi" "there")
  nil 5]
```

The above assumes that dynamic binding is in effect.

The constants vector in the above example contains 5 elements:

- `a` — the symbol `a` which refers to a variable
- `myfunc` —the symbol `myfunc` which likely refers to an external function
- `("hi" "there")` — a list constant containing two strings
- `nil` — the nil constant
- `5` — the integer constant 5

The properties of symbol `a` and symbol `myfunc` are consulted at run time, so there is no knowledge in the bytecode representing the fact that `a` is a dynamically-bound parameter while `my-func` is probably an external function.

If the lambda were lexically scoped, the constants vector would not have the variable symbol `a` listed, but instead there would be a stack entry.

Note that although the symbol `b` is a parameter of the lambda, it does not appear in the constants vector, since it is not used in the body of the function.

2.1.4 Maximum Stack Usage

The fourth object in a bytecode-function literal is an integer which gives the maximum stack space used by this bytecode. This value can be derived from the bytecode itself, but it is pre-computed so that the byte-code interpreter can quickly check for stack overflow. Under-reporting this value is probably another way to crash Emacs.

In our example above, the maximum-stack value is five since function `myfunc` is called with four parameters which are pushed onto the stack, and there is an additional stack entry pushed, the `myfunc` symbol itself. All of this needs to be in place on the stack just before a `call` instruction runs to perform the `myfunc` call.

2.1.5 Docstring

The fifth object in a bytecode-function literal. It is optional. As with the bytecode unibyte string, this value is either a string literal or a pointer to a string in a bytecode file.

Examples showing DocStrings:

```
ELISP> (aref
        (byte-compile
         (defun double(a)
          "double parameter A"
          (+ a a)))
        4) ;; 4 is the bytecode docstring field

"double parameter A"


ELISP> (aref
        (symbol-function 'cl-gcd)
         4)

("/tmp/emacs/lisp/emacs-lisp/cl-extra.elc" . 7251)
```

2.1.6 "Interactive" Specification

When there is a sixth field in the bytecode function, the function is a command, i.e., an "interactive" function. Otherwise the function is not a command. This parameter holds the exact contents of the argument to `interactive` in the uncompiled function definition. Note that `(interactive)` causes the sixth field to be nil, which is distinct from there not being a sixth field.

Examples showing the "interactive" specification

```
ELISP> (aref
         (byte-compile
           (lambda (n)
            (interactive "nNumber: ") n)
            )
         5) ;; 5 is the bytcode interactive specification field

"nNumber: "

ELISP> (aref
         (byte-compile
          (lambda (n)
           (interactive (list (read))) n))
         5)

(list
  (read))
```

The interactive expression is usually interpreted, which is fine because, by definition, this code is going to be waiting on user input, but it slows down keyboard macro playback.

2.2 Emacs Lisp Bytecode Compiler

The bytecode compiler is an ahead-of-time compiler that accepts Emacs Lisp input and produces bytecode that can be run by Emacs. The compiler itself is written in Emacs Lisp[1], and is a comparatively compact program contained in the files `bytecomp.el` and `byte-opt.el`.

Internally, the compiler first produces an intermediate Lisp structure in LAP code, then performs various optimizations on that, and finally translates the LAP code into bytecode. LAP code is used during compilation, but not kept in memory or used when running bytecode.

It is possible to go back to LAP code from bytecode. This is done in order to inline functions and when bytecode disassembly is requested.

[1] Usually the compiler itself is compiled into bytecode, which avoids overflow problems

2.3 Emacs Lisp Bytecode Interpreter

Note: the **bytecode** *interpreter that is described here should not be confused with the Emacs* **Lisp** *Interpreter.*

When a function is called and the function is represented as bytecode, control passes to the bytecode interpreter. The interpreter is written in C and is written more for speed than readability.

The bytecode interpreter operates on a single function at a time. For a function call, the bytecode interpreter calls other parts of Emacs, which might call the bytecode interpreter again, recursively. Thus, in contrast to languages like FORTH, there is no code stack per se, just the C stack.

The bytecode interpreter implements a stack machine utilizing a fixed-size evaluation stack, which is usually allocated as a block on the C stack. Instructions can access either this stack or a constants vector, which is produced at compile time and made part of the bytecode object.

The evaluation stack, as well as the constants vector, contains Lisp values, usually 64-bit words containing an integer (Emacs integers are limited to 62 bits on 64-bit machines), symbol index, or a tagged pointer to one of various Emacs structures such as markers, buffers, floating-point numbers, vectors, or cons cells.

Values on the evaluation stack are created at run time. Values in the constants vector are created when the byte-compiled file is read and converted into bytecode objects. The underlying bit representation of values in the constants vector can vary between Emacs instance; they are constants in the sense that they do not vary within a single Emacs instance.

Bytecode objects contain a number safely estimating the maximum stack size the evaluation stack can grow to.

2.4 Emacs Lisp Bytecode Bytes

The bytecode interpreter, once it has set up the evaluation stack and constants vector, executes the instructions that make up the bytecode byte sequence. Each instruction is between one and three bytes long, containing an opcode in the first byte and sometimes an eight- or 16-bit integer in the following bytes. Those integers are usually unsigned, and 16-bit integers are stored in little-endian byte order, regardless of whether that is the natural byte order for the machine Emacs runs on.

Some opcodes, allocated in blocks, encode an integer as part of the opcode byte.

Bytecode instructions operate on the evaluation stack. For example, `plus`, the addition function, removes two values from the top of the stack and pushes a single value, the sum of the first two values, back onto the stack.

Since the arguments for a function call need to be on the stack before the function can operate on them, bytecode instructions use reverse Polish notation: first the arguments are pushed onto the stack, then the function or operation is called. For example, the Lisp expression `(+ a b)` turns into this bytecode:

```
PC  Byte  Instruction
 0    8   varref a
 1    9   varref b
 2   92   plus
```

First `a` and `b` are dereferenced and their values pushed onto the evaluation stack; then `plus` is executed, leaving only a single value, the sum of `a` and `b`, on the stack.

2.5 Emacs Lisp Bytecode Files

When Emacs is build from source code, there is C code for some primitive or built-in functions. These include Lisp functions like `car`, or primitive Emacs functions like `point`. Other equally important functions are implemented in Emacs Lisp. These are byte compiled and then loaded into Emacs. On many systems there is the ability to dump Emacs in some kind of image format after these basic functions have been loaded, but even if that does not happen, a file called `loaddefs.el` is created which contains many of the important basic primitive functions as bytecode.

When we invoke Emacs then, it has a number of functions already loaded and these are either coded in C or have been byte compiled and loaded. Before running a function, Emacs queries the type of code that is associated with the function symbol and calls either its lambda S-expression interpreter or its bytecode interpreter.

When we run `load`, which reads and evaluates Lisp code from a file, at the top-level it does not matter whether the file contains bytecode or Emacs Lisp source code. Either way the only thing done is to open the file and read its contents using the normal Lisp reader.

The difference between the two kinds of files is more about convention than about their contents, and specifically two things: First the bytecode file will have a comment header in it that starts `;ELC^W^@^@^@` while the source code probably does not (although nothing to stop us from adding in that line if we feel like it). And, in addition to this comment header, a bytecode file will have other meta-comments such as which version of Emacs was used to compile the file and whether optimization was used. In earlier versions, there was information about the program that was used to compile the program, such its version number, and the source code path used to be in there as well. (I think these things should still be in there but that's a different story.) See Chapter 4 [Instruction Changes Between Emacs Releases], page 166, where we give examples of the headers to show how they have changed.

The second thing that is typically different between source code files and bytecode files is that bytecode files contain the `bytecode` calls used in the file and lack of any `defun`, `defmacro`, or `lambda` calls. But again there is presumably nothing stopping anyone from using these in their source code.

In fact, we can take a file with the `.elc` extension, rename it with an `.el` extension and `load` that, and it will run exactly the same if it had been loaded as a bytecode file[1].

Similarly, just as we can concatenate any number of independent Emacs Lisp source code files into one file, and this is sometimes done as a poor-man's way to create a package, we can also concatenate any numbers of Emacs Lisp bytecode files.

Of course, there are probably certain programs that are fooled when the extension is changed. In particular, the `byte-recompile-directory` function will think that the bytecode file does not exist because it has the wrong extension. So even though Emacs is permissive about such matters, it is best to stick with the normal Emacs conventions.

The final thing that should be mentioned when talking about bytecode files is interoperability between Emacs versions.

[1] If we go the other way and rename a Lisp file as a bytecode file, Emacs will notice the discrepancy because at the top of the file is a header that Emacs checks. But if we add a reasonable-looking header we can go that direction as well.

Even though a bytecode header has a meta comment indicating the version of Emacs that was used to compile it, that information is not used in determining whether the bytecode file can be run or not. This has the benefit of being able to run bytecode compiled in a different Emacs version than the version currently running. Since Emacs bytecode instructions do not change often, this largely works. The scary part, though, is that opcode meanings have changed over the 30 years, and the interpreter sometimes lacks checks. (In the past the interpreter aborted when running an invalid bytecode.) So Emacs does not even know when we are running bytecode from a different interpreter, and we might run off a cliff running older or newer bytecode without a check.

Emacs developers maintain that, in practice, problems have not been reported very much. Also, they try to keep backward compatibility between versions so that bytecode generated in an older version of Emacs will often still be interpreted in a recent newer version. While this is a worthwhile intention, my experience is that this does not always work, especially going back more than one version, and it is unrealistic to expect for a program that is 30 years old.

Because there is no up-front checking, bytecode generated from a newer version of Emacs will run silently on an older version until there is opcode that the older version cannot handle. In some cases it will complete. See Chapter 4 [Instruction Changes Between Emacs Releases], page 166, for when this is likely to work and when it won't. Although running newer bytecode in an older version of Emacs is not explicitly considered, since bytecode does not change very often, this can sometimes work out.

Note the sharp contrast with other bytecode interpreters, such as Python, where the magic used in compiling has to be the same as the value of the running interpreter or it will refuse to run.

It would be nice to have an Emacs Lisp bytecode checker, perhaps a `safer-load` function that looks at the bytecode. Its meta-comments would glean when there is something that is known to cause problems. Any volunteers?

2.6 Functions and Commands for working with LAP and Bytecode

You can byte-compile an individual function or macro definition with the `byte-compile` function. To extract individual components of that array use `aref`. To recover human-readable LAP code from a byte-compiled file use `dissasemble`. Perhaps in the future there will be a decompiler which reconstructs higher-level Lisp from LAP.

You can see if a symbol's value holds one of the function types or an alias to a function with `functionp`. To retrieve the definition of the function use `symbol-function`.

You can compile a buffer with `emacs-lisp-byte-compile`, or a whole file with `byte-compile-file`.

Several can be compiled with `byte-recompile-directory` or `batch-byte-compile`.

Sometimes, the byte compiler produces warning and/or error messages (see Section "Compiler Errors" in *GNU Emacs Lisp Reference Manual*, for details). These messages are normally recorded in a buffer called `*Compile-Log*`, which uses compilation mode. See Section "Compilation Mode" in *The GNU Emacs Manual*. However, if the variable `byte-compile-debug` is non-nil, error message will be signaled as Lisp errors instead (see Section "Errors" in *GNU Emacs Lisp Reference Manual*).

Be careful when writing macro calls in files that you intend to byte-compile. Since macro calls are expanded when they are compiled, the macros need to be loaded into Emacs or the byte compiler will not do the right thing. The usual way to handle this is with `require` forms which specify the files containing the needed macro definitions (see Section "Named Features" in *GNU Emacs Lisp Reference Manual*). Normally, the byte compiler does not evaluate the code that it is compiling, but it handles `require` forms specially, by loading the specified libraries. To avoid loading the macro definition files when someone runs the compiled program, write `eval-when-compile` around the `require` calls (see Section "Eval During Compile" in *GNU Emacs Lisp Reference Manual*). See Section "Compiling Macros" in *The GNU Emacs Lisp Reference Manual* for more details.

Inline (`defsubst`) functions are less troublesome. If you compile a call to such a function before its definition is known, the call will still work right; it will just run slower.

In the list below, some of the functions are somewhat general and are not specific to bytecode. however they are mentioned because they are specifically have an interesting use in bytecode and their connection might be readily appearent.

2.6.1 `aref`

`aref` *arry idx* [Function]

Return the element of *array* at index *idx*.

Use this to extract the individual components of a byte-code object. See Section 2.1 [Emacs Lisp Bytecode Objects], page 8, for numerous examples using `aref`.

```
ELISP> (aref
        (symbol-function 'cl-gcd)
        1) ;; 1 is the bytecode string field

"("/tmp/emacs/lisp/emacs-lisp/cl-extra.elc" . 7352)
```

2.6.2 batch-byte-compile

`batch-byte-compile` **&optional** *noforce* [Function]

This function runs `byte-compile-file` on files specified on the command line. This function must be used only in a batch execution of Emacs, as it kills Emacs on completion. An error in one file does not prevent processing of subsequent files, but no output file will be generated for it, and the Emacs process will terminate with a nonzero status code.

If *noforce* is non-`nil`, this function does not recompile files that have an up-to-date '`.elc`' file.

```
$ emacs -batch -f batch-byte-compile *.el
```

2.6.3 batch-byte-recompile-directory

`batch-byte-recompile-directory` *directory* **&optional** *arg* [Function]

Run `byte-recompile-directory` on the dirs remaining on the command line. Must be used only with `-batch`, and kills Emacs on completion. For example, invoke `emacs -batch -f batch-byte-recompile-directory .`.

Optional argument *arg* is passed as second argument *arg* to `byte-recompile-directory`; see there for its possible values and corresponding effects.

2.6.4 byte-code

`byte-code` *bytestr vector maxdepth* [Function]

This function is executes byte code and is used internally in byte-compiled code. The first argument, *bytestr*, is a string of byte code; the second, *vector*, a vector of constants; the third, *maxdepth*, the maximum stack depth used in this function. If the third argument is incorrect, Emacs may crash.

```
ELISP> (setq msg-string "hi")
"hi"
ELISP> (byte-code "\301Ф!\207" [msg-string message] 2)
"hi"
```

2.6.5 byte-compile

`byte-compile` *form* [Command]

If *form* is a symbol, byte-compile its function definition.

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial
```

```
(byte-compile 'factorial)
⇒
#[(integer)
  "^H\301U\203^H^@\301\207\302^H\303^HS!\"\207"
  [integer 1 * factorial]
  4 "Compute factorial of INTEGER."]
```

If *form* is a lambda or a macro, byte-compile it as a function.

```
(byte-compile
  (lambda (a) (* a a)))
⇒
  #[(a) "^H\211\207" [a] 2]
```

If *symbol*'s definition is a bytecode function object, `byte-compile` does nothing and returns `nil`. It does not compile the symbol's definition again, since the original (non-compiled) code has already been replaced in the symbol's function cell by the byte-compiled code.

2.6.6 byte-compile-file

`byte-compile-file` *filename* **&optional** *load* [Command]

This function compiles a file of Lisp code named *filename* into a file of bytecode. The output file's name is made by changing the '`.el`' suffix into '`.elc`'. If *filename* does not end in '`.el`', it adds '`.elc`' to the end of *filename*.

Compilation works by reading the input file one form at a time. If it is a definition of a function or macro, the compiled function or macro definition is written out. Other forms are batched, then each batch is compiled, and written so that its compiled code will be executed when the file is read. All comments are discarded when the input file is read.

This command returns `t` if there are no errors and `nil` otherwise. When called interactively, it prompts for the file name.

If *load* is non-`nil`, this command loads the compiled file after compiling it. Interactively, *load* is the prefix argument.

```
$ ls -l push*
-rw-r--r-- 1 lewis lewis 791 Oct  5 20:31 push.el

(byte-compile-file "~/emacs/push.el")
     ⇒ t

$ ls -l push*
-rw-r--r-- 1 lewis lewis 791 Oct  5 20:31 push.el
-rw-rw-rw- 1 lewis lewis 638 Oct  8 20:25 push.elc
```

2.6.7 byte-compile-sexp

`byte-compile-sexp` *sexp* [Function]

Bytecode compile and return *sexp*.

This can be useful for seeing what the byte compile does, especially when combined with `disassemble`.

```
ELISP> (disassemble
         (byte-compile-sexp '(1+ fill-column)))
byte code:
  args: nil
0 varref   fill-column
1 add1
2 return
ELISP> (disassemble
         (byte-compile-sexp
           '(unwind-protect (1+ fill-column) (ding))))

byte code:
  args: nil
0 constant  <compiled-function>
      args: nil
    0     constant  ding
    1     call      0
    2     return

1 unwind-protect
2 varref   fill-column
3 add1
4 unbind   1
5 return
```

2.6.8 `byte-recalc-examples`

`byte-recalc-examples` *begin end* [Command]

This command is what we use in this document to format our examples. It is not part of Emacs lisp but in `byte-pretty.el` inside the repository where this document lives.

Since we want to show values of various kinds — offsets, opcodes, operand, and constant-vector values — this output is a more verbose than the format you get from the `disassemble` command.

2.6.9 `byte-recompile-directory`

`byte-recompile-directory` *directory* **&optional** *flag force* [Command]

This command recompiles every '`.el`' file in *directory* (or its subdirectories) that needs recompilation. A file needs recompilation if an '`.elc`' file exists but is older than the '`.el`' file.

When a '`.el`' file has no corresponding '`.elc`' file, *flag* says what to do. If it is `nil`, this command ignores these files. If *flag* is 0, it compiles them. If it is neither `nil`

nor 0, it asks the user whether to compile each file, and asks about each subdirectory as well.

Interactively, `byte-recompile-directory` prompts for *directory* and *flag* is the prefix argument.

If *force* is non-`nil`, this command recompiles every '`.el`' file that has a '`.elc`' file.

The returned value is unpredictable.

2.6.10 `byte-recompile-file`

`byte-recompile-file` *filename* **&optional** *force arg load* [Command]

Recompile *filename* file if it needs recompilation. This happens when its '`.elc`' file is older than itself.

If the '`.elc`' file exists and is up-to-date, normally this function does not compile *filename*. If the prefix argument *force* is non-nil, however, it compiles *filename* even if the destination already exists and is up-to-date.

If the '`.elc`' file does not exist, normally this function *does not* compile *filename*. If optional argument ARG is 0, it compiles the input file even if the '`.elc`' file does not exist. Any other non-nil value of *arg* means to ask the user.

If optional argument *load* is non-nil, loads the file after compiling.

If compilation is needed, this functions returns the result of `byte-compile-file`; otherwise it returns *no-byte-compile*.

2.6.11 `compile-defun`

`compile-defun` **&optional** *arg* [Command]

This command reads the defun containing point, compiles it, and evaluates the result. If you use this on a defun that is actually a function definition, the effect is to install a compiled version of that function.

`compile-defun` normally displays the result of evaluation in the echo area, but if *arg* is non-`nil`, it inserts the result in the current buffer after the form it compiled.

2.6.12 `disassemble`

`disassemble` *object* **&optional** *buffer-or-name* [Command]

This command displays the disassembled code for *object*. In interactive use, or if *buffer-or-name* is `nil` or omitted, the output goes in a buffer named `*Disassemble*`. If *buffer-or-name* is non-`nil`, it must be a buffer or the name of an existing buffer. Then the output goes there, at point, and point is left before the output.

The argument *object* can be a function name, a lambda expression, or a byte-code object (see Section 2.1 [Emacs Lisp Bytecode Objects], page 8). If it is a lambda expression, `disassemble` compiles it and disassembles the resulting compiled code.

There are a couple of variables that control how disassembly is displayed:

Variable Name	Default Value
`disassemble-column-1-indent`	8
`disassemble-column-2-indent`	10
`disassemble-recursive-indent`	3

2.6.13 `disassemble-file`

`disassemble-file` *filename* [Command]

The command is not part of GNU Emacs, but is included in an experimental decompiler. It disassembles the entire contents of a bytecode file using the `disassemble-full` for each function.

2.6.14 `disassemble-full`

`disassemble` *object* **&optional** *buffer-or-name indent* [Command]

The command is not part of GNU Emacs, but is included in an experimental decompiler. In contrast to the standard `disassemble`, the format is slightly modified to make it easier to decompile the code. For example, the full text of docstring is preserved and is preceded by a length code of the string.

This functions prints disassembled code for *object* in *buffer-or-name*. *object* can be a symbol defined as a function, or a function itself (a lambda expression or a compiled-function object). If *object* is not already compiled, we compile it, but do not redefine *object* if it is a symbol."

2.6.15 `display-call-tree`

Even though this is a command, it only has an effect when *byte-compile-generate-call-tree* is set to non-nil; it is `nil` by default. In this case, it is called when a file is byte compiled, such as from `byte-compile-file`.

`display-call-tree` **&optional** *filename* [Command]

Display a call graph of a specified file. This lists which functions have been called, what functions called them, and what functions they call. The list includes all functions whose definitions have been compiled in this Emacs session, as well as all functions called by those functions.

The call graph does not include macros, inline functions, or primitives that the bytecode interpreter knows about directly, e.g. `eq`, `cons`.

The call tree also lists those functions which are not known to be called (that is, to which no calls have been compiled), and which cannot be invoked interactively.

2.6.16 `emacs-lisp-byte-compile`

Byte compile the file containing the current buffer. If you want to do that and also load the file, use `emacs-lisp-byte-compile-and-load`.

2.6.17 `functionp`

This is a general function, regarding functions in general.

`functionp` *object* [Function]

Non-nil if *object* is a function.

Use this to see if a symbol is a function, that is something that can be called. In most cases though `symbol-function` is more useful as it not only distinguishes functions from non-functions, but can it returns more information in those situations where *object* is a function.

2.6.18 `make-byte-code`

`make-byte-code` *arglist byte-code constants depth* **&optional** *docstring interactive-spec* **&rest** *elements* [Function]

Create a byte-code object with specified arguments as elements. The arguments should be the *arglist*, bytecode-string *byte-code*, constant vector *constants*, maximum stack size *depth*, (optional) *docstring*, and (optional) *interactive-spec*.

We briefly describe parameters below. For a more detailed discussion of the parameters, see Section 2.1 [Emacs Lisp Bytecode Objects], page 8.

The first four arguments are required; at most six have any significance. The *arglist* can be either like the one of 'lambda', in which case the arguments will be dynamically bound before executing the bytecode, or it can be an integer of the form *NNNNNNNRMMMMMMM* where the 7bit *MMMMMMM* specifies the minimum number of arguments, the 7-bit *NNNNNNN* specifies the maximum number of arguments (ignoring **&rest**) and the *R* bit specifies whether there is a **&rest** argument to catch the left-over arguments. If such an integer is used, the arguments will not be dynamically bound but will be instead pushed on the stack before executing the byte-code.

There very little checking of the validity of the elements either at creation time or at run time. If a parameter is are invalid or inconsistent, Emacs may crash when you call the function.

Examples of calling `make-byte-code`:

```
;; Null bytecode: no args, no bytecode, no stack needed
ELISP> (make-byte-code nil "" [] 0)

#[nil ""
    []
    0]

;; This byte-code for: '(lambda(a) (* a a ))
ELISP> (make-byte-code '(a) "^H211_\207" [a] 2)

#[(a)
  "^H211_\207"
  [a]
  2]

ELISP> (make-byte-code 1 2 3 4)

#[1 2 3 4]  ;; Doesn't even do type checking!
```

2.6.19 `symbol-function`

This is a general function, but it has an interesting use in conjunction with bytecode.

`symbol-function` *symbol* [Function]

Return *symbol*'s function definition, or nil if that is void.

The value returned from `symbol-function` for a function will when non-nil can be a number of things including:

- its Lisp expression value (type cons node)
- its bytecode value (type compiled-function)
- its C function value (type subr)
- its Rust function value, if remacs
- an autoload function call.

For example if we take a function that is autoloaded when Emacs starts up:

```
ELISP> (symbol-function 'insert-file)

#[257 "\300^A301\"\207"
      [insert-file-1 insert-file-contents]
      4 2029839 "*fInsert file: "]
```

However if you load a file redefining the function, by loading in emacs source, you get the last definition:

```
ELISP> (load-file "/usr/share/emacs/25.2/lisp/files.el.gz")

t
ELISP> (symbol-function 'insert-file)

(closure
  (backup-extract-version-start t)
  (filename)
  "Insert contents of file FILENAME into buffer after point.\nSet mark after the
  (interactive "*fInsert file: ")
  (insert-file-1 filename #'insert-file-contents))
```

Consider a function that hasn't been is set to be autoloaded:

```
ELISP> (symbol-function 'ediff-buffers)

(autoload "ediff" 975154 t nil)
```

Finally, consider an interal function like `eolp`

```
ELISP> (type-of (symbol-function 'eolp))
subr
```

2.7 Source and Bytecode Optimization

The bytecode optimizer works on two levels: source-to-source optimizing transformations and LAP-level peephole optimizations. It's possible to control optimization level by setting the *byte-optimize* variable (defined in `bytecomp.el`) to `nil`, `t`, `source` or `byte`.

The optimizing framework itself resides in `byte-opt.el`.

The source-to-source optimizer entry point is the `byte-optimize-form` function. It works on macro-expanded forms, recursively looking for function calls and special forms that have optimising functions defined for them.

The peephole optimizer (see `byte-optimize-lapcode`) function works on LAP directly. It goes through the instruction sequence using a window of 3 instructions to look for subsequences that can be dropped or replaced with more efficient versions.

Both optimizers work until a fixpoint is reached, i.e. all the transformations are applied until there are not further changes possible.

2.7.1 Source-to-source Transformations

All source-to-source transformations are performed by the `byte-optimize-form` function. It's possible to run `byte-optimize-form` on any well-formed form. E.g., `(byte-optimize-form '(if t 1 2))` returns `1`.

Given a form the `byte-optimize-form` function performs transformations in two ways: first it runs the `byte-optimize-form-code-walker` function; then it checks if the current form is both a function application and the function symbol has a specialized optimizing function defined for it in the `byte-optimizer` property.

`byte-optimize-form-code-walker` recognizes special forms (`if`, `let`, `let*`, `lambda`, `closure`, ...) and, having performed some sanity checks, recursively runs `byte-optimize-form` on subforms. Here's an example transformation of the `if` special form:

```
;; sanity checks
(when (< (length form) 3)
     (byte-compile-warn "too few arguments for `if'"))

;; simplify the form by optimizing subforms
(cons fn
  (cons (byte-optimize-form (nth 1 form) nil)
    (cons
      (byte-optimize-form (nth 2 form) for-effect)
      (byte-optimize-body (nthcdr 3 form) for-effect))))
```

The `byte-optimizer` symbol property is defined for most of the built-in side-effect-free functions. E.g. it is defined for some of the basic math functions in `byte-opt.el`:

```
(put '+   'byte-optimizer 'byte-optimize-plus)
(put '*   'byte-optimizer 'byte-optimize-multiply)
(put '-   'byte-optimizer 'byte-optimize-minus)
(put '/   'byte-optimizer 'byte-optimize-divide)
(put 'max 'byte-optimizer 'byte-optimize-associative-math)
(put 'min 'byte-optimizer 'byte-optimize-associative-math)
```

This is how constant folding (see Section 2.7.4 [Constant Folding], page 31) works in the compiler. The same approach is used for dead code elimination (see Section 2.7.5 [Dead Code Elimination], page 32) in built-in forms:

```
(put 'and   'byte-optimizer 'byte-optimize-and)
(put 'or    'byte-optimizer 'byte-optimize-or)
(put 'cond  'byte-optimizer 'byte-optimize-cond)
(put 'if    'byte-optimizer 'byte-optimize-if)
(put 'while 'byte-optimizer 'byte-optimize-while)
```

2.7.2 Peephole Optimization

Peephole optimization is implemented by the `byte-optimize-lapcode` function. Patterns to be replaced are represented as `cond` condition cases. The instruction window is 3 instructions long.

There are two groups of replacements: transformations applied until no further work can be done and transformations that should only be applied once.

Example transformation dropping side-effect-free instructions followed by the `discard` (drop the top of the stack) instruction:

```
;; <side-effect-free> pop -->  <deleted>
;;  ...including:
;; const-X pop   -->  <deleted>
;; varref-X pop  -->  <deleted>
;; dup pop       -->  <deleted>
((and ;; drop the top of the stack in the 2nd op?
      (eq 'byte-discard (car lap1))
      ;; and the 1st op has no side effects?
      (memq (car lap0) side-effect-free))
            ;; safely discard both instructions

            ;; make sure there will be one more pass
            (setq keep-going t)

            ;; we'll need to correct stack alignment info
            (setq tmp (aref byte-stack+-info (symbol-value (car lap0))))

            ;; next pass will start from the next instruction
            (setq rest (cdr rest))

            ;; drop instructions depending on what should be on the stack
            (cond ((= tmp 1)
                   (byte-compile-log-lap
                     "  %s discard\t-->\t<deleted>" lap0)
                   (setq lap (delq lap0 (delq lap1 lap))))
                  ((= tmp 0)
                   (byte-compile-log-lap
                    "  %s discard\t-->\t<deleted> discard" lap0)
```

```
                        (setq lap (delq lap0 lap)))
                       ((= tmp -1)
                        (byte-compile-log-lap
                         "  %s discard\t-->\tdiscard discard" lap0)
                        (setcar lap0 'byte-discard)
                        (setcdr lap0 0))
                       ((error "Optimizer error: too much on the
                                    stack"))))
    ;; more transformations...
```

2.7.3 Adding an Optimizing Transformation

TODO

2.7.4 Constant Folding

In cases were constants can be evaluated at compile time to come up with simpler results, that is done.

`(defun constant-fold-eg() (+ 1 2))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  193   constant[1] 2
 2   92   plus
 3  135   return

Constants Vector: [1 2]
```

while with optimization we get:

```
PC  Byte  Instruction
 0  192   constant[0] 3
 1  135   return

Constants Vector: [3]
```

In Floating-point constant folding in Emacs byte compile (`https://lists.gnu.org/archive/html/emacs-devel/2018-04/msg00018.html`) the notion was put forth that optimization has to be portable over improving code. (The issue here was compiling Emacs with larger integers allowed for larger possibiles of constant folding).

Although Emacs can be compiled with different for integers and floats depending the setting of `--with-wide-int`, for portability, Emacs will assume in bytecode the smaller value of integers and will skip opportunities that would assume larger integers.

2.7.5 Dead Code Elimination

If there is no way code can be reached, it is removed. This optimization interacts with the previous optimization: constant folding.

With bytecode optimization and lexicals scoping off:

```
(defun dead-code-eg(a)
   (or t a))
```

generates:

```
PC  Byte  Instruction
 0  193   constant[1] t
 1  134   goto-if-not-nil-else-pop [5]
           5
           0
 4    8   varref[0] a
 5  135   return

Constants Vector: [a t]
```

On the other hand, with bytecode-optimization we get:

```
PC  Byte  Instruction
 0  192   constant[0] t
 1  135   return

Constants Vector: [t]
```

2.7.6 Strength Reduction

The optimizer can recognize when there is primative instructions that implements an equivalent longer set of instructions.

For example without optimization:

`(defun strength-reduce-eg(a) (+ a 1))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1  193   constant[1] 1
 2   92   plus
 3  135   return

Constants Vector: [a 1]
```

However with optimizaion `(defun strength-reduce-opt-eg(a) (+ a 1))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1   84   add1
 2  135   return

Constants Vector: [a]
```

Notice that the optimizer took advantage of the commutative property of addition and treated `(+ a 1)` as the same thing as `(+ 1 a)`.

2.8 LAP Decompiler

Let's face it — most of us would rather work with Emacs Lisp, a higher-level language than bytecode or its more human-friendly LAP disassembly. There is a project in its early stages that can often reconstruct Emacs Lisp from bytecode generated from Emacs Lisp.

See the Elisp Decompiler Project (`https://github.com/rocky/elisp-decompile`) for more details.

Although there is much work that needs to be done to make this work more of the time, quite frankly, it blows my mind that I've been able to get this far. And it was a lot of work to get this far. First there is setup just to have some sort of decompilation framework, albeit in Python. And then, I needed to first start writing *this* document since there wasn't anything nearly as complete when I started. There were various tables around for bytecode instructions, but I needed not just cumulative evaluation stack effects for a bytecode operation, but how many entries on the stack are read and then how many written. This was the motivation for adding that to this document. Many of the mistakes that I've found here are a direct result of working on the decompiler. And having the semantics of each operation was helpful in writing the decompiler.

A number of people have opined that you really don't need a decompiler.[1]

Below are the arguments offered and why I think they are inadequate.

[1] Whenever something new or novel comes along, in addition to the group that says "Why not?!" there is the "Why bother?" group. In my (rocky) lifetime, I have experienced this attitude when undo and regular expressions were added to GNU Emacs, adding a debugger to languages where debugging support

2.8.1 It's GNU Emacs, so of course I have the source code!

There is a difference between being able to find the source code and having it *accessible* when you need it, such as in an interactive session or at runtime from a stack trace.

When many people hear the word "decompile" they think reverse engineering or hacking code where source has deliberately been withheld. But there are other situations where a decompiler is useful.

A common case is where you wrote the code, but have accidentally deleted the source code and just have the bytecode file.

But, I know, you always use version control and GNU Emacs provides it's tilde backup file.

So that leads us to the situation where there are *several* possible source-code versions around, e.g. a development version and a stable version, or one of the various versions in your version-control system, and you'd like to know which one of those is stored in a bytecode file, that you have loaded.

And then we come to situation where there *is* no source-code file. One can create functions on the fly and change them on the fly. Lisp is known for its culture in having programs create programs; it is possible such a program doesn't have a "debug" switch (that you know about) to either save or show the result before it was byte-compiled. Perhaps you'd like to reconstruct the source code for a function that you worked on interactively.

2.8.2 Isn't it simpler to just disassemble?

Interestingly, a number of people suggested that programmers who want to understand bytecode should use LAP and decompile mentally.

Most people don't know LAP. In fact, before I started writing this document, there really *wasn't* any good documentation on LAP, short of reading source code. So a side benefit is that the process of writing a decompiler has made the documentation of what's there in bytecode more easily accessible.

But from writing this decompiler and noting all of the subtleties and intricacies, I am pretty convinced that those who say they understand LAP have to spend a bit of time-consuming tedious work to decipher things. I sure do.

This isn't is what computers were invented for? They do this kind of thing very fast compared to humans.

Here are some simple examples:

macros

I would find it tedious enough just to descramble something that has been macro expanded. And I am sure people may find that unsatisfying right now with our results. Now imagine how you'd feel to add another layer on that in going from LAP to Elisp for the expanded macros.

The LAP instructions for:

```
(define-minor-mode testing-minor-mode "Testing")
```

---

didn't exist or was weak, using decomplation to give more precise error location information, and here. Convincing the crowd that is happy with the status quo is hard. The next section is more for those who have an open mind and want to see a *better* world.

expand to 60 or so LAP instructions; a lot more when various parameters have been filled in.

And then you have things like `dolist` which are just long and boring template kinds of things. Because it's long it is very easy to lose sight of what it is.

Stacked values

Keeping track of values pushed on a stack is also tedious. Again, there can be some non-locality in when a value is pushed with when it is used and popped. As an example, consider the LAP instructions in Figure Figure 2.1. It is similar to a very well-known mathematical function.

```
constant  fn
dup
varref    n
sub1
call      1
constant  fn
varref    n
constant  2
diff
call      1
plus
call      1
```

Figure 2.1

At what point is that duplicated function the second instruction used? And what are the arguments to this function?

How long did it take you to figure this out? It takes a computer hundredths of a second to reconstruct the Lisp code.

Keyboard bindings

This is yet another piece of tedium for the few that know how to do.

As before see how fast you can come up with the key names for:

```
[134217761]
[134217854]
[134217820]
```

Again this is done in hundredths of a second by a computer.

Pedagogy

Another aspect about a decompiler, especially this one, is that it can help you learn LAP and Emacs bytecode, and how the existing compiler and optimizer work.

The program creates a parse tree from (abstracted) LAP instructions, where the the higher levels of the tree consist of grammar nodes that should be familiar in higher-level Emacs Lisp terms.

With this it is possible to see how the individual instructions combine to form the higher-level constructs.

Figure Figure 2.2 has is decompilation using of the LAP instructions in Figure Figure 2.1.

```
$ lap-decompile --tree=after tmp/foo.lap
fn_body (3)
  0. body
    exprs
      expr_stmt (2)
        0. expr
          call_exprn (3)
            0. expr
              name_expr
                    0 CONSTANT   fn
            1. expr
              binary_expr (3)
                0. expr
                  call_exprn (3)
                    0. expr
                          1 DUP
                    1. expr
                      unary_expr (2)
                        0. expr
                              2 VARREF     n
                        1. unary_op
                              3 SUB1
                    2.    4 CALL_1     1
                1. expr
                  call_exprn (3)
                    0. expr
                      name_expr
                            5 CONSTANT   fn
                    1. expr
                      binary_expr (3)
                        0. expr
                              6 VARREF     n
                        1. expr
                          name_expr
                                7 CONSTANT   2
                        2. binary_op
                              8 DIFF
                    2.    9 CALL_1     1
                2. binary_op
                     10 PLUS
            2.   11 CALL_1     1
        1. opt_discard
  1. opt_label
  2. opt_return
(defun foo(n)
  (fn (+ (fn (1- n)) (fn (- n 2)))))
```

Figure 2.2

Looking at Figure Figure 2.2 we see that:

```
1 DUP
2 VARREF      n
3 SUB1
4 CALL_1      1
```

contains a function call and its parameters; One of the parameters is the unary expression:

```
2 VARREF      n
3 SUB1
```

And if you want to know which operation the stack value of first instruction `CONSTANT fn` is used in, the nesting makes it easy to see it is the very last instruction `CALL_1` at offset 11.

As I mentioned before, personally, I find matching this stuff up a bit tedious.

You can also get control-flow graphs from the decompiler. See the github repository for more details.

3 Emacs Lisp Bytecode Instructions

Although we document bytecode instructions here, the implementation of the bytecode interpreter and its instructions appear in `src/bytecode.c`. If there is any question, that should be consulted as the primary reference.

3.1 Instruction-Description Format

In this chapter we'll document instructions over the course of the entire history of Emacs. Or at least we aim to.

For the opcode names, we will prefer canonicalized names from the Emacs C source `bytecode.c` (under directory `src/`) when those differ from the names in `bytecomp.el` (under directory `lisp/emacs-lisp`). Most of the time they are the same under the transformation described below.

We use names from `bytecode.c` because that is a larger set of instruction names. Specifically, obsolete instructions names (both those that can be interpreted even though they are no longer generated, and some that are no longer interpreted) are defined in that file, whereas that is not the case in `bytecomp.el`.

Names in `bytecode.c` must follow C conventions and must be adjusted to harmonize with other C names used. But this aspect isn't of use here, so we canonicalize those aspects away.

For example, in `bytecode.c` there is an opcode whose name is `Bbuffer_substring`. We will drop the initial `B` and replace all underscores (`_`) with dashes (`-`). Therefore we use `buffer-substring`.

The corresponding name for that opcode in `bytecomp.el` is `byte-buffer-substring`. For the most part, if you drop the initial `byte-` prefix in the `bytecomp.el` name you will often get the canonic name from `bytecode.c`.

However this isn't always true. The instruction that the Emacs Lisp `save-current-buffer` function generates nowadays has opcode value 114. In the C code, this value is listed as `B_save_current_buffer_1`; `bytecomp.el` uses the name `byte-save-current-buffer`. We report the instruction name for opcode 114 as `save-current-buffer-1`.

To shorten and regularize instruction descriptions, each instruction is described a standard format. We will also require a small amount of jargon. This jargon are explained below.

3.1.1 Instruction Jargon

- `OPERAND` The value of operand of the instruction. Many times this is done by taking the value of the opcode and subtracting from it the value of the first opcode in the series of opcodes it is part of. For example, the `call` opcodes span opcode values 32 to 39. The `OPERAND` value for opcode 32 then is $0 = 32 - 32$; for opcode 33, the value is $1 = 33 - 32$.
- `TOS` The value of top of the evaluation stack. Many instructions either read or push onto this.

- `S` This is an array of evaluation stack items. `S[0]` is the top of the stack, or `TOS`.

  Note that indexing starts from the top of the stack and increases as we move down the stack.

- `top` A pointer to the the top of the evaluation stack. In C this would be `&TOS`. When we want the stack to increase in size, we add to top. For example, to makes space to store a new single new value, we can use `top++` and then assign to `TOS`.

  Note that in changing `top`, the value accessed by `TOS` or `S` values all change.

- ϕ This is used in describing stack effects for branching instructions where the stack effect is different on one branch versus the other. This is a function of two arguments. The first argument gives the stack effect on the non-nil branch and the second argument gives the stack effect for the nil branch. So $\phi(0, -1)$ which is seen in `goto-if-not-nil-else-pop` means that if the jump is taken, the stack effect is 0, otherwise the effect removes or pops an evaluation-stack entry.

- instruction-name subscripting (`[]`) In many instructions such as `constant`, `varref`, you will find an index after the instruction name. What's going on is that instruction name is one of a number of opcodes in a class encodes an index into the instruction. We generally call this an "Argument-encoding" instruction. In the display of the opcode in assembly listings and in the opcode table chapter where we list each opcode, we will include that particular instruction variant in subscripts.

  For example consider `constant[0]` versus `constant[1]`. The former has opcode 192 while the latter has opcode 193. In terms of semantics, the former is the first or zeroth-index entry in a function's constant vector while the latter is the second or 1-index entry.

- eval stack types Just as in Emacs Lisp, many functions expect their arguments to be of a certain type, so it is true in Bytecode. Although the information here mirrors what is described in the Emacs Lisp Reference Manual (see Section "Lisp Data Types" in *The GNU Emacs Lisp Reference Manual*), for convenience we include that here as well.

3.1.2 Instruction Description Fields

The description of fields use for describing each instruction is as follows

Implements:
A description of what the instruction does.

Generated via:
These give some Emacs Lisp constructs that may generate the instruction. Of course there may be many constructs and there may be limiting situations within that construct. We'll only give one or a few of the constructs, and we'll try to indicate a limiting condition where possible.

Operand: When an instruction has an operand, this describes the type of the operand. Note that the size of the operand (or in some cases the operand value) will determine the instruction size.

Instruction size:
The number of bytes in the instruction. This is 1 to 3 bytes.

Stack effect:
This describes how many stack entries are read and popped and how many entries stack entries are pushed. Although this is logically a tuple, we'll list this a tuple like $(-3, 2)$ as a single scalar $-3 + 2$. In this example, we read/remove three stack entries and add two. The reason we give this as $-3 + 2$ rather than the tuple format is so that the overall effect (removing a stack entry) can be seen by evaluating the expression.

Added in: This is optional. When it is given this gives which version of Emacs the opcode was added. It may also give when the opcode became obsolete or was no longer implemented.

Example: Some Emacs Lisp code to show how the instruction is used. For example the for the `goto` instruction we give:

```
(defun goto-eg(n)
  (while (n) 1300))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] n
 1   32   call[0]
 2  133   goto-if-nil-else-pop [8]
           8
           0
 5  130   goto [0]
           0
           0
 8  135   return

Constants Vector: [n]
```

From the above we see that the `goto` instruction at program counter (PC) 5, has decimal opcode 130. The instruction is three bytes long: a one-byte opcode followed by a two-byte operand.

The instruction name at PC 0 with opcode 192, `constant[0]`, looks like it is indexing, but it is a just name, where the brackets and number are part of the name. We use this kind of name because it is suggestive of how it works: it indexes the first element into the constants vector and pushes that value onto the evaluation stack. `constant[1]` with opcode 193 pushes the second element of the constants vector onto the stack. We could have also used instruction names like `constant0` and `constant1` for opcodes 192 and 193 instead.

Unless otherwise stated, all code examples were compiled in Emacs 25 with optimization turned off.

3.2 Argument-Packing Instructions

These instructions from opcode 1 to 47 encode an operand value from 0 to 7 encoded into the first byte. If the encoded value is 6, the actual operand value is the byte following the opcode. If the encoded value is 7, the actual operand value is the two-byte number following the opcode, in Little-Endian byte order.

`stack-ref` (1–7)

Reference a value from the evaluation stack.

Implements:
`top++; TOS <- S[OPERAND+1]`

Generated via:
`let`, `let*` and lambda arguments.

Operand: A stack index

Instruction size:
1 byte for `stack-ref[0]` .. `stack-ref[4]`; 2 bytes for `stack-ref[5]`, 8-bit operand; 3 bytes for `stack-ref[6]`, 16-bit operand.

Stack effect:
$-0+1$.

Type Information:
after: `TOS: Object`

Added in: Emacs 24.1. See Section 4.7 [Emacs 24], page 172.

Example: When lexical binding and optimization are in effect,

```
(defun stack-ref-eg()
  (let ((a 5) (_b 6) (c 7))
    (+ a c)))}
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] 5 ;; top++; TOS <- 5
 1  193   constant[1] 6 ;; top++; TOS <- 6
 2  194   constant[2] 7 ;; top++; TOS <- 7
 3    2   stack-ref[2]  ;; top++; TOS <- S[3] (5)
 4    1   stack-ref[1]  ;; top++; TOS <- S[2] (7)
 5   92   plus
 6  135   return

Constants Vector: [5 6 7]
```

Warning Running an instruction with opcode 0 (logically this would be called `stack-ref[0]`), will cause an immediate abort of Emacs in versions after version 20 and before version 25! The abort of the opcode was in place before this instruction was added.

Zero is typically an invalid in bytecode and in machine code, since zero values are commonly found data, e.g. the end of C strings, or data that has been initialized to value but

represents data that hasn't been written to yet. By having it be an invalid instruction, it is more likely to catch situations where random sections of memory are run such as by setting the PC incorrectly.

varref (8–15)

varref (8–15)

Pushes the value of the symbol in the constants vector onto the evaluation stack.

Implements:
`top++; TOS <- (eval constants[OPERAND])`

Generated via:
dynamic variable access

Operand: A constants vector index. The constants vector item should be a variable symbol.

Instruction size:
1 byte for `varref[0]` .. `varref[4]`; 2 bytes for `varref[5]`, 8-bit operand; 3 bytes for `varref[6]`, 16-bit operand.

Stack effect:
$-0+1$.

Type Information:
after: `TOS: Object`

Example: When dynamic binding is in effect,

```
(defun varref-eg(n)
  n)
```

generates:

```
PC  Byte  Instruction
 0    8   varref[0] n
 1  135   return

Constants Vector: [n]
```

varset (16–23)

`varset` (16–23)

Sets a variable listed in the constants vector to the TOS value of the stack.

Implements:

```
constants[OPERAND] <- TOS; top--
```

Operand: A constants vector index. The constants vector item should be a variable symbol.

Instruction size:

1 byte for `varset[0]` .. `varset[4]`; 2 bytes for `varset[5]`, 8-bit operand; 3 bytes for `varset[6]`, 16-bit operand.

Stack effect:

$-1 + 0$.

Type Information:

before: `TOS: Object`

Example: When dynamic binding is in effect,

```
defun varset-eg(n)
  (setq n 5))
```

generates:

```
PC  Byte  Instruction
 0  193   constant[1] 5
 1  137   dup
 2   16   varset[0] n ;; sets variable n
 3  135   return

Constants Vector: [n 5]
```

varbind (24–31)

`varbind` (24–31)

Binds a variable to a symbol in the constants vector, and adds the symbol to a special-bindings stack.

Implements:

```
*constants[OPERAND]<-TOS; top--
```

Instruction size:

1 byte for `varbind[0]` .. `varbind[4]`; 2 bytes for `varbind[5]`, 8-bit operand; 3 bytes for `varbind[6]`, 16-bit operand.

Stack effect:

$-1 + 0$.

Type Information:

before: `TOS: Object`

Example: When dynamic binding is in effect,

```
defun varbind-eg()
  (let ((c 1))
    (1+ c)))
```

generates:

```
PC  Byte  Instruction
 0  193   constant[1] 1
 1  137   dup
 2   24   varbind[0] c ;; creates variable c
 3   84   add1
 4   41   unbind[1]    ;; removes variable c
 5  135   return

Constants Vector: [c 1]
```

call (32–39)

`call` (32–39)

Calls a function. The instruction argument specifies the number of arguments to pass to the function from the stack, excluding the function itself.

Implements:

```
arg[0..OPERAND-1] = S[OPERAND-1..0]
fn <- S[OPERAND];
top -= OPERAND;
TOS <- fn(*arg) # replace fn by the fn's return value
```

Before the call, the top of the stack has the final call parameter, if there is one. If there are no parameters, the top of the stack has the function to be called. The function to be called is always the least recent evaluation stack entry followed the function parameters in the order they were given.

For example, for `gcd(3, 5)` the operator is `call[2]` and the value of `OPERAND` is 2. The evaluation stack before the function call for this example would be:

```
+------------+
| S[0] : 5   |
+------------+
| S[1] : 3   |
+------------+
| S[2] : gcd |
+------------+
```

After the call completes, the evaluation stack will be:

```
+-------------------------+
| S[0] : gcd-return-value |
+-------------------------+
```

Instruction size:

1 byte for `call[0]` .. `call[4]`; 2 bytes for `call[5]`, 8-bit operand; 3 bytes for `call[6]`, 16-bit operand.

Stack effect:

$-(\texttt{OPERAND} + 1) + 1$.

Example:

```
(defun call-eg()
  (exchange-point-and-mark)
  (next-line 2))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] exchange-point-and-mark
 1   32   call[0]
 2  136   discard
```

```
 3  193   constant[1] next-line
 4  194   constant[2] 2
 5   33   call[1]
 6  135   return

Constants Vector: [exchange-point-and-mark next-line 2]
```

unbind (40–47)

unbind (40–47)

Remove the binding of a variable to symbol and from the special stack. This is done when the variable is no longer needed.

Implements:
undo's a `let`, `unwind-protect`s, and `save-excursion`s

Generated via:
`let` in dynamic binding. Balancing the end of `save-excursion`.

Instruction size:
1 byte for `unbind[0]` .. `unbind[4]`; 2 bytes for `unbind[5]`, 8-bit operand; 3 bytes for `unbind[6]`, 16-bit operand.

Stack effect:
$-0+0$.

Example: When dynamic binding is in effect,

```
defun varbind-eg()
  (let ((c 1))
    (1+ c)))
```

generates:

```
PC  Byte  Instruction
 0  193   constant[1] 1
 1  137   dup
 2   24   varbind[0] c ;; creates variable c
 3   84   add1
 4   41   unbind[1]    ;; removes variable c
 5  135   return

Constants Vector: [c 1]
```

3.3 Constants-Vector Retrieval Instructions

The instructions from opcode 192 to 255 push a value from the Constants Vector. See Section 2.1.3 [Constants Vector], page 11. Opcode 192 pushes the first entry, opcode 193, the second and so on. If there are more than 64 constants, opcode `constant2` (opcode 129) is used instead.

`constant` (192–255)

Pushes a value from the constants vector on the evaluation stack. There are special instructions to push any one of the first 64 entries in the constants stack.

Implements:

```
top++; TOS <- constants[OPERAND]
```

Instruction size:

1 byte

Stack effect:

$-0+1$.

Type Information:

after: `TOS: Object`

Example:

```
defun n3(n)
  (+ n 10 11 12))
```

generates:

```
PC  Byte  Instruction
 0  193   constant[1] +
 1    8   varref[0] n
 2  194   constant[2] 10
 3  195   constant[3] 11
 4  196   constant[4] 12
 5   36   call[4]
 6  135   return

Constants Vector: [n + 10 11 12]
```

constant2 (129)

`constant2` (129)

Pushes a value from the constants vector on the evaluation stack. Although there are special instructions to push any one of the first 64 entries in the constants stack, this instruction is needed to push a value beyond one the first 64 entries.

Implements:

`top++; TOS <- constants[OPERAND]`

Operand: a 16-bit index into the constants vector.

Instruction size:

3 bytes

Stack effect:

$-0+1$.

Example:

```
(defun n64(n)
 (+ n 0 1 2 3 .. 64))
```

generates:

```
PC  Byte  Instruction
 0  193   constant[1] +
 1    8   varref[0] n
 2  194   constant[2] 0
 3  195   constant[3] 1
 4  196   constant[4] 2
[...]
63  255   constant[63] 61
64  129   constant2 [64] 62
          64
           0
67  129   constant2 [65] 63
          65
           0
70  129   constant2 [66] 64
          66
           0
73   38   call [66]
          66
75  135   return

Constants Vector: [n + 0 1 2 .. 61 62 63 64]
```

3.4 Exception-Handling Instructions

`pophandler` (48)

Implements:
Removes last condition pushed by `pushconditioncase`

Generated via:
`condition-case`

Instruction size:
1 byte

Stack effect:
$-0+0$.

Added in: Emacs 24.4. See Section 4.7 [Emacs 24], page 172.

Example:

```
(defun pushconditioncase-eg()
  (condition-case nil
    5
    (one-error 6)
    (another-error 7)))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] (another-error)
 1   49   pushconditioncase [16]
          16
           0
 4  193   constant[1] (one-error)
 5   49   pushconditioncase [12]
          12
           0
 8  194   constant[2] 5
 9   48   pophandler
10   48   pophandler
11  135   return
12   48   pophandler
13  136   discard
14  195   constant[3] 6
15  135   return
16  136   discard
17  196   constant[4] 7
18  135   return

Constants Vector: [(another-error) (one-error) 5 6 7]
```

pushconditioncase (49)

pushconditioncase (49)

Implements:
Pops the TOS which is some sort of condition to test on and registers that. If any of the instructions errors with that condition, a jump to the operand occurs.

Operand: 16-bit PC address

Instruction size:
3 bytes

Stack effect:
$-1 + 0$.

Added in: Emacs 24.4. See Section 4.7 [Emacs 24], page 172.

Example:

```
(defun pushconditioncase-eg()
  (condition-case nil
    5
    (one-error 6)
    (another-error 7)))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] (another-error)
 1   49   pushconditioncase [16]
          16
           0
 4  193   constant[1] (one-error)
 5   49   pushconditioncase [12]
          12
           0
 8  194   constant[2] 5
 9   48   pophandler
10   48   pophandler
11  135   return
12   48   pophandler
13  136   discard
14  195   constant[3] 6
15  135   return
16  136   discard
17  196   constant[4] 7
18  135   return

Constants Vector: [(another-error) (one-error) 5 6 7]
```

pushcatch (50)

`pushcatch` **(50)**
?

3.5 Control-Flow Instructions

`goto` (130)

Implements:
Jump to label given in the 16-bit operand

Generated via:
`while` and various control-flow constructs

Operand: 16-bit PC address

Instruction size:
3 bytes

Stack effect:
$-1+0$

Example: `(defun goto-eg(n) (while (n) 1300))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] n
 1   32   call[0]
 2  133   goto-if-nil-else-pop [8]
           8
           0
 5  130   goto [0]
           0
           0
 8  135   return

Constants Vector: [n]
```

goto-if-nil (131)

`goto-if-nil` (131)

Implements:
Jump to label given in the 16-bit operand if TOS is nil. In contrast to `goto-if-nil-else-pop`, the test expression, TOS, is always popped.

Generated via:
`if` with "else" clause and various control-flow constructs

Operand: 16-bit PC address

Instruction size:
3 bytes

Stack effect:
$-1+0$

Example: `(defun goto-if-nil-eg(n) (if (n) 1310 1311))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] n
 1   32   call[0]
 2  131   goto-if-nil [9]
           9
           0
 5  193   constant[1] 1310
 6  130   goto [10]
           10
           0
 9  194   constant[2] 1311
10  135   return

Constants Vector: [n 1310 1311]
```

goto-if-not-nil (132)

`goto-if-not-nil` (132)

Implements:
Jump to label given in the 16-bit operand if TOS is not nil. In contrast to `goto-if-not-nil-else-pop`, the test expression, TOS, is always popped.

Generated via:
`or` inside an `if` with optimization and various control-flow constructs

Operand: 16-bit PC address

Instruction size:
3 bytes

Stack effect:
$-1+0$

Example: With bytecode optimization, `(defun goto-if-not-nil-eg(n) (if (or (n) (n)) 1320))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] n
 1   32   call[0]
 2  132   goto-if-not-nil [10]
           10
            0
 5  192   constant[0] n
 6   32   call[0]
 7  133   goto-if-nil-else-pop [11]
           11
            0
10  193   constant[1] 1320
11  135   return

Constants Vector: [n 1320]
```

Note the change in opcode when bytecode optimization is not performed.

goto-if-nil-else-pop (133)

`goto-if-nil-else-pop` (133)

Implements:
Jump to label given in the 16-bit operand if TOS is nil; otherwise pop the TOS, the tested condition. This allows the test expression, nil, to be used again on the branch as the TOS.

Generated via:
`cond`, `if` and various control-flow constructs

Operand: 16-bit PC address

Instruction size:
3 bytes

Stack effect:
$\phi(0, -1) + 0$

Example: `(defun goto-if-nil-else-pop-eg(n) (cond ((n) 1330)))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] n
 1   32   call[0]
 2  133   goto-if-nil-else-pop [6]
           6
           0
 5  193   constant[1] 1330
 6  135   return

Constants Vector: [n 1330]
```

goto-if-not-nil-else-pop (134)

`goto-if-not-nil-else-pop` (134)

Implements:

Jump to label given in the 16-bit operand if TOS is not nil; otherwise pop TOS, the tested condition. This allows the tested expression on TOS to be used again when the jump is taken.

Generated via:

`cond`, `if` and various control-flow constructs

Operand: 16-bit PC address

Instruction size:

3 bytes

Stack effect:

$\phi(0, -1) + 0$

Example:

```
(defun goto-if-not-nil-else-pop-eg(n)
  (if (or (n) (n))
    1340))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] n
 1   32   call[0]
 2  134   goto-if-not-nil-else-pop [7]
           7
           0
 5  192   constant[0] n
 6   32   call[0]
 7  133   goto-if-nil-else-pop [11]
           11
           0
10  193   constant[1] 1340
11  135   return

Constants Vector: [n 1340]
```

Note the change in opcode when bytecode optimization is performed.

return (135)

`return` (135)

Implements:

Return from function. This is the last instruction in a function's bytecode sequence. The top value on the evaluation stack is the return value.

Generated via:

`lambda`

Instruction size:

1 byte

Stack effect:

$-1+0$

Example: `(defun return-eg(n) 1350)` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1350
 1  135   return

Constants Vector: [1350]
```

switch (183)

`switch` (183)

Jumps to entry in a jump table.

Implements:

switch-like jump table. Top of stack is a variable reference. Below that is a hash table mapping compared values to instruction addresses. `v, ht <- TOS, S[1]; top -=2; pc = ht.lookup(v)`

Note: for efficiency, the bytecode interpreter tends to use actually addresses rather than offsets off of bytecode start address. This is seen here in the values stored in the lookup table.

Generated via:

`cond` with several clauses that use the same test function and variable.

Instruction size:

1 byte

Stack effect:

$-2 + 0$

Added in: Emacs 26.1

Example:

```
(defun switch-eg(n)
   (cond ((equal n 1) 1)
         ((equal n 2) 2)
         ((equal n 3) 3)))
```

generates:

```
PC  Byte  Instruction
 0    8   varref[0] n
 1  193   constant[1] #s(hash-table size 3 test equal rehash-size 1.5 rehash-thre
 2  183   switch
 3  130   goto [12]
          12
           0
 6  194   constant[2] 1
 7  135   return
 8  195   constant[3] 2
 9  135   return
10  196   constant[4] 3
11  135   return
12  197   constant[5] nil
13  135   return

Constants Vector: [n #s(hash-table size 2 test equal rehash-size 1.5 rehash-thres
```

3.6 Function-Call Instructions

These instructions use up one byte, and are followed by the next instruction directly. They are equivalent to calling an Emacs Lisp function with a fixed number of arguments: the arguments are popped from the stack, and a single return value is pushed back onto the stack.

3.6.1 Lisp Function Instructions

These instructions correspond to general functions which are not specific to Emacs; common cases are usually inlined for speed by the bytecode interpreter.

`symbolp` (57)

Call `symbolp`. For a description of this Emacs Lisp function, See Section "Symbols" in *The GNU Emacs Lisp Reference Manual*, .

Implements:
`TOS <- (symbolp TOS)`.

Generated via:
`symbolp`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Example: When lexical binding is in effect, `(defun symbolp-eg(n) (symbolp n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   57   symbolp
 2  135   return
```

consp (58)

consp (58)

Call `consp`. For a description of this Emacs Lisp function, See Section "Predicates on Lists" in `The GNU Emacs Lisp Reference Manual`.

Implements:
`TOS <- (consp TOS)`.

Generated via:
`consp`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Example: When lexical binding is in effect, `(defun consp-eg(n) (consp n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   58   consp
 2  135   return
```

stringp (59)

`stringp` (59)

Call `stringp`. For a description of this Emacs Lisp string predicate, See Section "Predicates for Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (stringp TOS)`.

Generated via:
unary `stringp`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Example: When lexical binding is in effect, `(defun stringp-eg(n) (stringp n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   59   stringp
 2  135   return
```

listp (60)

`listp` (60)

Call `listp`. For a description of this Emacs Lisp list predicate, See Section "Predicates on Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (listp TOS)`.

Generated via:
unary `listp`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Example: When lexical binding is in effect, `(defun listp-eg(n) (listp n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   60   listp
 2  135   return
```

eq (61)

`eq` (61)

Call `eq`. For a description of this Emacs Lisp equality predicate, See Section "Equality Predicates" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (eq S[1] TOS); top--`.

Generated via:
binary `eq`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Object; S[1]: Object`

after: `TOS: Bool`

Example: When lexical binding is in effect, `(defun eq-eg(a b) (eq a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   61   eq
 3  135   return
```

memq (62)

memq (62)

Call `memq`. For a description of this Emacs Lisp list function, See Section "Using Lists as Sets" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (memq S[1] TOS); top--`.

Generated via:
binary `memq`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: List; S[1]: Object`

after: `TOS: Object`

Example: When lexical binding is in effect, `(defun memq-eg(a b) (memq a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   62   memq
 3  135   return
```

not (63)

`not` (63)

Call `not`. For a description of this Emacs Lisp combining condition, See Section "Constructs for Combining Conditions" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (not TOS)`.

Generated via:
unary `not`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Example: When lexical binding is in effect, `(defun not-eg(a) (not a))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   63   not
 2  135   return
```

symbol-value (74)

`symbol-value` (74)

Call `symbol-value`. For a description of this Emacs Lisp function, See Section "Accessing Variable Values" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (symbol-value TOS)`.

Generated via:
`symbol-value`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Example: When lexical binding is in effect, `(defun symbol-value-eg(a) (symbol-value a))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   74   symbol-value
 2  135   return
```

symbol-function (75)

`symbol-function` (75)

Call `symbol-function`. For a description of this Emacs Lisp function, See Section "Accessing Function Cell Contents" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (symbol-function TOS)`.

Generated via:
`symbol-function`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Example: When lexical binding is in effect, `(defun symbol-function-eg(a) (symbol-function a))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   75   symbol-function
 2  135   return
```

set (76)

`set` (76)

Call `set`. For a description of this Emacs Lisp function, See Section "Setting Variable Values" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (set S[1] TOS); top--`.

Generated via:
`set`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Example: When lexical binding is in effect, `(defun set-eg(a b) (set a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   76   set
 3  135   return
```

fset (77)

`fset` (77)

Call `fset`. For a description of this Emacs Lisp function, See Section "Accessing Function Cell Contents" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (fset S[1] TOS); top--`.

Generated via:
binary `fset`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Example: When lexical binding is in effect, `(defun fset-eg(a b) (fset a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   77   fset
 3  135   return
```

get (78)

`get` (78)

Call `get`. For a description of this Emacs Lisp function, See Section "Accessing Symbol Properties" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (get S[1] TOS); top--`.

Generated via:
binary `get`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Example: When lexical binding is in effect, `(defun get-eg(a b) (get a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   78   get
 3  135   return
```

equal (154)

`equal` (154)

Call `equal`. For a description of this Emacs Lisp function, See Section "Equality Predicates" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (equal S[1] TOS); top--`.

Generated via:
binary `equal`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Object; S[1]: Object`

after: `TOS: Bool`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: When lexical binding is in effect, `(defun equal-eg(a b) (equal a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2  154   equal
 3  135   return
```

member (157)

`member` (157)

Call `member`. For a description of this Emacs Lisp function, See Section "Using Lists as Sets" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (member S[1] TOS); top--`.

Generated via:
`member`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Example: When lexical binding is in effect, `(defun member-eg(a b) (member a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2  157   member
 3  135   return
```

assq (158)

assq (158)

Call `assq`. For a description of this Emacs Lisp function, See Section "Association Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (assq S[1] TOS); top--`.

Generated via:
binary `assq`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Example: When lexical binding is in effect, `(defun assq-eg(a b) (assq a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2  158   assq
 3  135   return
```

numberp (167)

numberp (167)

Call `numberp`. For a description of this Emacs Lisp function, See Section "Type Predicates for Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (numberp TOS)`.

Generated via:
`numberp`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: When lexical binding is in effect, `(defun numberp-eg(a) (numberp a))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1  167   numberp
 2  135   return
```

integerp (168)

integerp (168)

Call `integerp`. For a description of this Emacs Lisp function, See Section "Type Predicates for Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (integerp TOS)`.

Generated via:
`integerp`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: Bool`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: When lexical binding is in effect, `(defun integerp-eg(a) (integerp a))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1  168   integerp
 2  135   return
```

3.6.2 List Function Instructions

These instructions correspond to general functions which are not specific to Emacs; common cases are usually inlined for speed by the bytecode interpreter.

`nth` (56)

Call `nth` with two stack arguments. For a description of this Emacs Lisp function, See Section "Accessing Elements of Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (nth S[1] TOS); top--`.

Generated via:
`nth`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: List; S[1]: Number`

after: `TOS: Object`

Note the that the order of the stack values is opposite `nthcdr` even though in Emacs Lisp the order of the parameters is the same for both functions.

Example: When lexical binding is in effect, `(defun nth-eg(l) (nth 560 l))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 560
 1    1   stack-ref[1]
 2   56   nth
 3  135   return

Constants Vector: [560]
```

car (64)

`car` (64)

Call `car` with one stack argument. For a description of this Emacs Lisp function, See Section "Accessing Elements of Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (car TOS)`.

Generated via:
`car`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Cons-cell`

after: `TOS: Object`

Example: When lexical binding is in effect, `(defun car-eg(l) (car l))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   64   car
 2  135   return
```

cdr (65)

`cdr` (65)

Call `cdr` with one stack argument. For a description of this Emacs Lisp function, See Section "Accessing Elements of Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (cdr TOS)`.

Generated via:
`cdr`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Cons-cell`

after: `TOS: Object`

Example: When lexical binding is in effect, `(defun cdr-eg(l) (cdr l))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   65   cdr
 2  135   return
```

cons (66)

cons (66)

Call `cons` with two stack arguments. For a description of this Emacs Lisp function, See Section "Building Cons Cells and Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (cons S[1] TOS); top--`.

Generated via:
`cons`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Object; S[1]: Object`

after: `TOS: Cons`

Example: `(defun cons-eg() (cons 'a 'b))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] a
 1  193   constant[1] b
 2   66   cons
 3  135   return

Constants Vector: [a b]
```

list1 (67)

`list1` (67)

Call `list` with TOS. For a description of this Emacs Lisp function, See Section "Building Cons Cells and Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (list TOS)`.

Generated via:
`list`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Object`

after: `TOS: List`

Example: `(defun list1-eg() (list 'a))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] a
 1   67   list1
 2  135   return

Constants Vector: [a]
```

Call `list` with TOS.

list2 (68)

`list2` (68)

Call `list` with two stack items. For a description of this Emacs Lisp function, See Section "Building Cons Cells and Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (list S[1] TOS); top--`.

Generated via:
`list`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Object; S[1]: Object`

after: `TOS: List`

Example: `(defun list2-eg() (list 'a 'b))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] a
 1  193   constant[1] b
 2   68   list2
 3  135   return

Constants Vector: [a b]
```

list3 (69)

`list3` (69)

Call `list` with three stack items. For a description of this Emacs Lisp function, See Section "Building Cons Cells and Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[2] <- (list S[2] S[1] TOS); top -= 2`.

Generated via:
`list`

Instruction size:
1 byte

Stack effect:
$-3+1$.

Type Information:
before: `TOS: Object; S[1]: Object; S[2]: Object`
after: `TOS: List`

Example: `(defun list3-eg() (list 'a 'b 'c))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] a
 1  193   constant[1] b
 2  194   constant[2] c
 3   69   list3
 4  135   return

Constants Vector: [a b c]
```

list4 (70)

`list4` (70)

Call `list` with four stack items. For a description of this Emacs Lisp function, See Section "Building Cons Cells and Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[3] <- (list S[3] S[2] S[1] TOS); top -= 2`.

Generated via:
`list`.

Instruction size:
1 byte

Stack effect:
$-4 + 1$.

Type Information:
before: `TOS: Object; S[1]: Object; S[3]: Object; S[2]: Object`
after: `TOS: List`

Example: `(defun list4-eg() (list 'a 'b 'c 'd))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] a
 1  193   constant[1] b
 2  194   constant[2] c
 3  195   constant[3] d
 4   70   list4
 5  135   return

Constants Vector: [a b c d]
```

listN (175)

`listN` (175)

Call `list` on up to 255 items. For a description of this Emacs Lisp function, See Section "Building Cons Cells and Lists" in *The GNU Emacs Lisp Reference Manual*.

Note that there are special instruction for the case where there are 1 to 4 items in the list.

Implements:
`S[OPERAND-1] <- (list S[OPERAND-1] S[OPERAND-2] ... TOS); top -= (OPERAND-1)`

Generated via:
`list`.

Operand: 8-bit number of items in list

Instruction size:
2 bytes

Stack effect:
$-$`OPERAND` $+ 1$

Type Information:
before: `S[OPERAND-1]: Object; ... S[1]: Object; TOS: Object`
after: `TOS: List`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun listN-eg() (list 'a 'b 'c 'd 'e))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] a
 1  193   constant[1] b
 2  194   constant[2] c
 3  195   constant[3] d
 4  196   constant[4] e
 5  175   listN[5] 5
 7  135   return

Constants Vector: [a b c d e]
```

length (71)

`length` (71)

Call `length` with one stack argument. For a description of this Emacs Lisp function, See Section "Sequences" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (length TOS)`.

Generated via:
`length`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Sequence`

after: `TOS: Number`

Example: `(defun length-eg() (length '(a b)))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] (a b)
 1   71   length
 2  135   return

Constants Vector: [(a b)]
```

aref (72)

`aref` (72)

Call `aref` with two stack arguments. For a description of this Emacs Lisp function, See Section "Functions that Operate on Arrays" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (aref S[1] TOS); top--`.

Generated via:
`aref`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Example: `(defun aref-eg() (aref '[720 721 722] 0))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] [720 721 722]
 1  193   constant[1] 0
 2   72   aref
 3  135   return

Constants Vector: [[720 721 722] 0]
```

aset (73)

`aset` (73)

Call `aset` with three stack arguments. For a description of this Emacs Lisp function, See Section "Functions that Operate on Arrays" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`S[2] <- (aset S[2] S[1] TOS); top-=2`.

Generated via:

`aset`.

Instruction size:

1 byte

Stack effect:

$-3+1$.

Type Information:

before: `TOS: Object: S[1]: Number; S[2]: Array`

after: `TOS: Object`

Example: `(defun aset-eg() (aset array-var 0 730))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] array-var
 1  193   constant[1] 0
 2  194   constant[2] 730
 3   73   aset
 4  135   return

Constants Vector: [array-var 0 730]
```

nthcdr (155)

nthcdr (155)

Call `nthcdr` with two stack arguments. For a description of this Emacs Lisp function, See Section "Accessing Elements of Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (nthcdr S[1] TOS); top --`.

Generated via:
`nthcdr`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: List`

after: `TOS: Object`

Note the that the order of the stack values is opposite `nth` even though in Emacs Lisp the order of the parameters is the same for both functions.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun nthcdr-eg() (nthcdr '(1550 1551 1552) 2))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] (1550 1551 1552)
 1  193   constant[1] 2
 2  155   nthcdr
 3  135   return

Constants Vector: [(1550 1551 1552) 2]
```

elt (156)

`elt` (156)

Call `elt` with two stack arguments. For a description of this Emacs Lisp sequence function, See Section "Sequences" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (elt S[1] TOS); top --`.

Generated via:
`elt`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number: S[1]: Sequence`

after: `TOS: Object`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun elt-eg() (elt '(1560 1561 1562) 2))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] (1560 1561 1562)
 1  193   constant[1] 2
 2  156   elt
 3  135   return

Constants Vector: [(1560 1561 1562) 2]
```

nreverse (159)

nreverse (159)

Call `nreverse` with one stack argument. For a description of this Emacs Lisp function, See Section "Sequences" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (elt TOS)`.

Generated via:
`nreverse`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Sequence`

after: `TOS: Sequence`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun nreverse-eg() (nreverse '(1590 1591)))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] (1590 1591)
 1  159   nreverse
 2  135   return

Constants Vector: [(1590 1591)]
```

setcar (160)

`setcar` (160)

Call `setcar` with two stack arguments. For a description of this Emacs Lisp function, See Section "Altering List Elements with `setcar`" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (setcar S[1] TOS); top--`.

Generated via:
`setcar`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Object; S[1]: cons`

after: `TOS: Object`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect, `(defun setcar-eg(l) (setcar l 1600))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1  192   constant[0] 1600
 2  160   setcar
 3  135   return

Constants Vector: [1600]
```

setcdr (161)

`setcdr` (161)

Call `setcdr` with two stack arguments. For a description of this Emacs Lisp function, See Section "Altering the CDR of a List" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (setcdr S[1] TOS); top--`.

Generated via:
`setcdr`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Object; S[1]: Cons-cell`

after: `TOS: Object`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect, `(defun setcdr-eg(l) (setcdr l 1610))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1  192   constant[0] 1610
 2  161   setcdr
 3  135   return

Constants Vector: [1610]
```

car-safe (162)

`car-safe` (162)

Call `car-safe` with one argument. For a description of this Emacs Lisp function, See Section "Accessing Elements of Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (car-safe TOS)`.

Generated via:
`car-safe`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Cons-cell`

after: `TOS: Object`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect, `(defun car-safe-eg(l) (car-safe l))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1  162   car-safe
 2  135   return
```

cdr-safe (163)

`cdr-safe` (163)

Call `cdr-safe` with one argument. For a description of this Emacs Lisp function, See Section "Accessing Elements of Lists" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (cdr-safe TOS)`.

Generated via:
`cdr-safe`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Cons-cell`

after: `TOS: Object`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect, `(defun cdr-safe-eg(l) (cdr-safe l))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1  163   cdr-safe
 2  135   return
```

nconc (164)

nconc (164)

Call `nconc` with two stack arguments. For a description of this Emacs Lisp function, See Section "Altering the CDR of a List" in *The GNU Emacs Lisp Reference Manual*.

The Emacs Lisp function `nconc` can take an arbitrary number of arguments. The bytecode compiler breaks these up into repeated binary operations.

Implements:
`S[1] <- (nconc S[1] TOS); top--`.

Generated via:
`nconc`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: List; S[1]: List`

after: `TOS: List`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect, `(defun nconc-eg(l1 l2 l3) (nconc l1 l2 l3))` generates:

```
PC  Byte  Instruction
 0    2   stack-ref[2]
 1    2   stack-ref[2]
 2  164   nconc
 3    1   stack-ref[1]
 4  164   nconc
 5  135   return
```

3.6.3 Arithmetic Function Instructions

These instructions correspond to general functions which are not specific to Emacs; common cases are usually inlined for speed by the bytecode interpreter.

`sub1` (83)

Call `1-`. For a description of this Emacs Lisp arithmetic operation, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (1- TOS)`.

Generated via:
`1-`.

Instruction size:
1 byte

Stack effect:
$-1 + 1$.

Type Information:
before: `TOS: Number`

after: `TOS: Number`

Example: When lexical binding is in effect, `(defun sub1-eg(n) (1- n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   83   sub1
 2  135   return
```

add1 (84)

`add1` (84)

Call `1+`. For a description of this Emacs Lisp arithmetic operation, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (1+ TOS)`.

Generated via:
unary `-`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Number`

after: `TOS: Number`

Example: When lexical binding is in effect, `(defun add1-eg(n) (1+ n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   84   add1
 2  135   return
```

eqlsign (85)

eqlsign (85)

Call `=`. For a description of this Emacs Lisp comparision function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (= S[1] TOS); top--`.

Generated via:
binary `=`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When dynamic binding is in effect, `(defun eqlsign-eg(a b) (= a b))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1    9   varref[1] b
 2   85   eqlsign
 3  135   return

Constants Vector: [a b]
```

gtr (86)

gtr (86)

Call `>`. For a description of this Emacs Lisp function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`S[1] <- (> S[1] TOS); top--`.

Generated via:

`>`.

Instruction size:

1 byte

Stack effect:

$-2+1$.

Type Information:

before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When lexical binding is in effect, `(defun gtr-eg(a b) (> a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   86   gtr
 3  135   return
```

lss (87)

`lss` (87)

Call `<`. For a description of this Emacs Lisp function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (< S[1] TOS); top--`.

Generated via:
`<`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When dynamic binding is in effect, `(defun lss-eg(a b) (< a b))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1    9   varref[1] b
 2   87   lss
 3  135   return

Constants Vector: [a b]
```

leq (88)

`leq` (88)

Call `<=`. For a description of this Emacs Lisp function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (<= S[1] TOS); top--`.

Instruction size:
1 byte

Generated via:
`<=`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When dynamic binding is in effect, `(defun leq-eg(a b) (<= a b))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1    9   varref[1] b
 2   88   leq
 3  135   return

Constants Vector: [a b]
```

geq (89)

geq (89)

Call `>=`. For a description of this Emacs Lisp function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (>= S[1] TOS); top--`.

Instruction size:
1 byte

Generated via:
`>=`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When lexical binding is in effect, `(defun geq-eg(a b) (>= a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   89   geq
 3  135   return
```

diff (90)

`diff` (90)

Call binary `-`. For a description of this Emacs Lisp function, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

When the argument list is more than two, a `call` instruction is used instead.

Implements:
`S[1] <- (- S[1] TOS); top--`.

Generated via:
binary `-`.

Instruction size:
1 byte

Instruction size:
1 byte

Stack effect:
$-2 + 1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When lexical binding is in effect, `(defun diff-eg(a b) (- a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2   90   diff
 3  135   return
```

negate (91)

`negate` (91)

Call unary `-`. For a description of this Emacs Lisp function, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (- TOS)`.

Generated via:
unary `-`.

Instruction size:
1 byte

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `TOS: Number`

after: `TOS: Number`

Example: When lexical binding is in effect, `(defun negate-eg(a) (- a))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1   91   negate
 2  135   return

Constants Vector: [a]
```

plus (92)

`plus` (92)

Call binary `+`. For a description of this Emacs Lisp arithmetic operation, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

When adding more than two numbers, a `call` instruction is used instead.

Implements:
`S[1] <- (+ S[1] TOS); top--`.

Generated via:
`+`.

Instruction size:
1 byte

Stack effect:
$-2 + 1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

Example: When dynamic binding is in effect, `(defun plus-eg(n) (+ n n))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] n
 1  137   dup
 2   92   plus
 3  135   return

Constants Vector: [n]
```

mult (95)

`mult` (95)

Call binary `*`. For a description of this Emacs Lisp arithmetic operation, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

When multiplying more than two numbers, a `call` instruction is used instead.

Implements:
`S[1] <- (* S[1] TOS); top--`.

Generated via:
`*`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: String; S[1]: String`
after: `TOS: Bool`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: When dynamic binding is in effect, `(defun mult-eg(n) (* n n))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] n
 1  137   dup
 2   95   mult
 3  135   return

Constants Vector: [n]
```

max (93)

max (93)

Call binary `max`. For a description of this Emacs Lisp comparision function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

The Emacs Lisp function `max` can take an arbitrary number of arguments. The bytecode compiler breaks these up into repeated binary operations.

Implements:
`S[1] <- (max S[1] TOS); top--`.

Generated via:
`max`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`
after: `TOS: Number`

Example: When dynamic binding is in effect, `(defun max-eg(a b c) (max a b c))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1    9   varref[1] b
 2   93   max
 3   10   varref[2] c
 4   93   max
 5  135   return

Constants Vector: [a b c]
```

min (94)

`min` (94)

Call binary `min`. For a description of this Emacs Lisp comparison function, See Section "Comparison of Numbers" in *The GNU Emacs Lisp Reference Manual*.

The Emacs Lisp function `min` can take an arbitrary number of arguments. The bytecode compiler breaks these up into repeated binary operations.

Implements:
`S[1] <- (+ S[1] TOS); top--`.

Generated via:
binary `min`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`
after: `TOS: Number`

Example: When dynamic binding is in effect, `(defun min-eg(a b c) (min a b c))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1    9   varref[1] b
 2   94   min
 3   10   varref[2] c
 4   94   min
 5  135   return

Constants Vector: [a b c]
```

quo (165)

quo (165)

Call `/`. For a description of this Emacs Lisp arithmetic operator, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (/ S[1] TOS); top--`.

Generated via:
`/`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

A zero value for `TOS` will raise an arithmetic error.

Example: When dynamic binding is in effect, `(defun min-quo(a b) (/ a b))` generates:

```
PC  Byte  Instruction
 0    8   varref[0] a
 1    9   varref[1] b
 2  165   quo
 3  135   return

Constants Vector: [a b]
```

rem (166)

`rem` (166)

Call `%`. For a description of this Emacs Lisp arithmetic operator, See Section "Arithmetic Operations" in *The GNU Emacs Lisp Reference Manual*.

implements:
`S[1] <- (% S[1] TOS); top--`.

generated via:
`%`

Instruction size:
1 byte

Stack effect:
$-2+1$

Type Information:
before: `TOS: Number; S[1]: Number`

after: `TOS: Number`

A zero value for `TOS` will raise an arithmetic error.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: When lexical binding is in effect, `(defun rem-eg(a b) (% a b))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2  166   rem
 3  135   return
```

3.6.4 String Function Instructions

These instructions correspond to general functions which are not specific to Emacs; the bytecode interpreter calls the corresponding C function for them.

`substring` (79)

Call `substring` with three stack arguments. For a description of this Emacs Lisp string function, See Section "Creating Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[2] <- (substring S[2] S[1] TOS); top-=2`.

Generated via:
`substring`.

Instruction size:
1 byte

Stack effect:
$-3+1$.

Type Information:
before: `TOS: Number: S[1]: Number; S[2]: String`
after: `TOS: String`

Example: `(defun substring-eg() (substring "abc" 0 2))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "abc"
 1  193   constant[1] 0
 2  194   constant[2] 2
 3   79   substring
 4  135   return

Constants Vector: ["abc" 0 2]
```

concat2 (80)

concat2 (80)

Call `concat` with two stack arguments. For a description of this Emacs Lisp function string function, See Section "Creating Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (concat S[1] TOS); top--`.

Generated via:
`concat`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: String: S[1]: String`

after: `TOS: String`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun concat2-eg() (concat "a" "b"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "a"
 1  193   constant[1] "b"
 2   80   concat2
 3  135   return

Constants Vector: ["a" "b"]
```

concat3 (81)

concat3 (81)

Call `concat` with three stack arguments. For a description of this Emacs Lisp function string function, See Section "Creating Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`S[2] <- (concat S[2] S[1] TOS); top-=2`.

Generated via:

`concat`.

Instruction size:

1 byte

Stack effect:

$-3+1$.

Type Information:

before: `TOS: String; S[1]: String; S[2]: String`

after: `TOS: String`

Example: `(defun concat3-eg() (concat "a" "b" "c"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "a"
 1  193   constant[1] "b"
 2  194   constant[2] "c"
 3   81   concat3
 4  135   return

Constants Vector: ["a" "b" "c"]
```

concat4 (82)

concat4 (82)

Call `concat` with four stack stack arguments. For a description of this Emacs Lisp function string function, See Section "Creating Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[3] <- (concat S[3] S[2] S[1] TOS); top -= 3`.

Generated via:
`concat`.

Instruction size:
1 byte

Stack effect:
$-4 + 1$.

Type Information:
before: `TOS: String; S[1]: String; S[2]: String; S[3]: String`
after: `TOS: String`

Example: `(defun concat4-eg() (concat "a" "b" "c" "d"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "a"
 1  193   constant[1] "b"
 2  194   constant[2] "c"
 3  195   constant[3] "d"
 4   82   concat4
 5  135   return

Constants Vector: ["a" "b" "c" "d"]
```

concatN (174)

`concatN` (174)

Call `concat` on up to 255 stack arguments. Note there are special instructions for the case where there are 2 to 4 items to concatenate. For a description of this Emacs Lisp function string function, See Section "Creating Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[OPERAND-1] <- (concat S[OPERAND-1] S[OPERAND-2] ... TOS); top -= (OPERAND-1)`.

Generated via:
`concat`.

Operand: 8-bit number of items in concat

Instruction size:
2 bytes

Stack effect:
$-n+1$ where n is the value of the instruction operand.

Type Information:
before: `S[OPERAND-1]: String; ... S[1]: String; TOS: String`
after: `TOS: String`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun concatN-eg() (concat "a" "b" "c" "d" "e"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "a"
 1  193   constant[1] "b"
 2  194   constant[2] "c"
 3  195   constant[3] "d"
 4  196   constant[4] "e"
 5  176   concatN [5]  5
 7  135   return

Constants Vector: ["a" "b" "c" "d" "e"]
```

upcase (150)

upcase (150)

Call `upcase` with one stack argument. For a description of this Emacs Lisp string function, See Section "Case Conversion in Lisp" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (upcase TOS)`.

Generated via:
`upcase`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `S[TOS]: Character or String`

after: `TOS: same type as before operation`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun upcase-eg() (upcase "abc"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "abc"
 1  150   upcase
 2  135   return

Constants Vector: ["abc"]
```

downcase (151)

`downcase` (151)

Call `downcase` with one argument. For a description of this Emacs Lisp string function, See Section "Case Conversion in Lisp" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (downcase TOS)`.

Generated via:
`downcase`.

Instruction size:
1 byte

Stack effect:
$-1+1$.

Type Information:
before: `S[TOS]: Character or String`

after: `TOS: same type as before operation`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun downcase-eg(l) (downcase "ABC"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "ABC"
 1  151   downcase
 2  135   return

Constants Vector: ["ABC"]
```

stringeqlsign (152)

stringeqlsign (152)

Call `string=` with two stack arguments, comparing two strings for equality. For a description of this Emacs Lisp string function, See Section "Comparison of Characters and Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (string= S[1] TOS); top--`.

Generated via:
`string=`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: Number; S[1]: Number`
after: `TOS: Number`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect,

```
(defun stringeqlsign-eg(a)
  (string= a "b"))
```

generates:

```
PC  Byte  Instruction
 0  137   dup
 1  192   constant[0] "b"
 2  152   string=
 3  135   return

Constants Vector: ["b"]
```

stringlss (153)

`stringlss` (153)

Call `string<` with two stack arguments, comparing two strings. For a description of this Emacs Lisp string function, See Section "Comparison of Characters and Strings" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (string< S[1] TOS); top--`.

Generated via:
`string<`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Type Information:
before: `TOS: String; S[1]: String`
after: `TOS: Bool`

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect,

```
(defun stringlss-eg(a)
  (string< a "b"))
```

generates:

```
PC  Byte  Instruction
 0  137   dup
 1  192   constant[0] "b"
 2  153   string<
 3  135   return

Constants Vector: ["b"]
```

3.6.5 Emacs Buffer Instructions

`current-buffer` (112)

Call `current-buffer`. For a description of this Emacs Lisp buffer function, See Section "The Current Buffer" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
top++; TOS <- (current_buffer)
```

Generated via:

`current-buffer`

Instruction size:

1 byte

Stack effect:

$-0+1$.

Type Information:

after: `TOS: Buffer`

Example: `(defun current-buffer-eg() (current-buffer))` generates:

```
PC  Byte  Instruction
 0  112   current-buffer
 1  135   return
```

set-buffer (113)

`set-buffer` (113)

Call `set-buffer` with TOS. For a description of this Emacs Lisp buffer function, See Section "The Current Buffer" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`TOS <- (set-buffer TOS)`

Generated via:

`set-buffer`

Instruction size:

1 byte

Stack effect:

$-1+1$.

Type Information:

before: `TOS: String`

after: `TOS: Buffer`

Example: `(defun set-buffer-eg() (set-buffer "*scratch*"))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "*scratch*"
 1  113   set-buffer
 2  135   return

Constants Vector: ["*scratch*"]
```

save-current-buffer-1 (114)

`save-current-buffer-1` (114)

Call `save-current-buffer`. For a description of this Emacs Lisp buffer function, See Section "The Current Buffer" in *The GNU Emacs Lisp Reference Manual*.

Replaces older `save-current-buffer`. See [save-current-buffer], page 162.

Implements:

`to be filled in...`

Generated via:

`save-current-buffer`

Instruction size:

1 byte

Stack effect:

0 + 1?

Added in: Emacs 22. See Section 4.5 [Emacs 22], page 171.

Example:

```
(defun save-current-buffer-1-eg()
   (save-current-buffer (prog 5)))}
```

generates:

```
PC  Byte  Instruction
 0  114   save-current-buffer
 1  192   constant[0] prog
 2  193   constant[1] 5
 3   33   call[1]
 4   41   unbind[1]
 5  135   return

Constants Vector: [prog 5]
```

buffer-substring (123)

`buffer-substring` (123)

Call `buffer-substring` with two stack arguments. For a description of this Emacs Lisp buffer function, See Section "Examining Buffer Contents" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (buffer-substring S[1] TOS); top--`.

Generated via:
`buffer-substring`.

Instruction size:
1 byte

Stack effect:
$-2+1$. Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun buffer-substring-eg() (buffer-substring 1230 1231))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1230
 1  193   constant[1] 1231
 2  123   buffer-substring
 3  135   return

Constants Vector: [1230 1231]
```

3.6.6 Emacs Position Instructions

These instructions correspond to Emacs-specific position functions that are found in the "Positions" chapter of the Emacs Lisp Reference Manual. They are not inlined by the bytecode interpreter, but simply call the corresponding C function.

`point` (96)

Call `point`. For a description of this Emacs Lisp position function, See Section "Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (point)`

Generated via:
`point`

Instruction size:
1 byte

Stack effect:
$-0+1$.

Added in: Emacs 18.31, renamed from `dot`. See Section 4.1 [Emacs 18], page 166.

Example: `(defun point-eg() (point))` generates:

```
PC  Byte  Instruction
 0   96   point
 1  135   return
```

goto-char (98)

`goto-char` (98)

Call `goto-char` with one stack argument. For a description of this Emacs Lisp motion function, See Section "Motion by Characters" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`TOS <- (goto-char TOS)`

Generated via:

`goto-char`

Instruction size:

1 byte

Stack effect:

$-1 + 1$.

Example: With lexical binding in effect, `(defun goto-char-eg(n) (goto-char n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   98   goto-char
 2  135   return
```

point-max (100)

`point-max` (100)

Call `point-max`. For a description of this Emacs Lisp position function, See Section "Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (point-max)
```

Generated via:

`point-max`

Instruction size:

1 byte

Stack effect:

$-0+1$.

Added in: Emacs 18.31, renamed from `dot-max`. See Section 4.1 [Emacs 18], page 166.

Example: `(defun point-max-eg() (point-max))` generates:

```
PC  Byte  Instruction
 0  100   point-max
 1  135   return
```

point-min (101)

`point-min` (101)

Call `point-min`. For a description of this Emacs Lisp position function, See Section "Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (point-min)`

Generated via:
`point-min`

Instruction size:
1 byte

Stack effect:
$-0+1$.

Added in: Emacs 18.31, renamed from `dot-min`. See Section 4.1 [Emacs 18], page 166.

Example: `(defun point-min-eg() (point-min))` generates:

```
PC  Byte  Instruction
 0  101   point-min
 1  135   return
```

forward-char (117)

`forward-char` (117)

Call `forward-char` with one stack argument. For a description of this Emacs Lisp motion function, See Section "Motion by Characters" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (forward-char TOS)
```

Generated via:

`forward-char`

Instruction size:

1 byte

Stack effect:

$-1 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun forward-char-eg() (forward-char 1))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  117   forward-char
 2  135   return

Constants Vector: [1]
```

forward-word (118)

`forward-word` (118)

Call `forward-word` with one stack argument. For a description of this Emacs Lisp motion function, See Section "Motion by Words" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (forward-word TOS)
```

Generated via:

`forward-word`

Instruction size:

1 byte

Stack effect:

$-1 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun forward-word-eg() (forward-word 1))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  118   forward-word
 2  135   return

Constants Vector: [1]
```

forward-line (121)

`forward-line` (121)

Call `forward-line` with one stack argument. For a description of this Emacs Lisp motion function, See Section "Motion by Text Lines" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (forward-line TOS)
```

Generated via:

`forward-line`

Instruction size:

1 byte

Stack effect:

$-1 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun forward-line-eg() (forward-line 1))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  121   forward-line
 2  135   return

Constants Vector: [1]
```

skip-chars-forward (119)

`skip-chars-forward` (119)

Call `skip-chars-forward` with two stack arguments. For a description of this Emacs Lisp motion function, See Section "Skipping Characters" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (skip-chars-forward S[1] TOS); top--`.

Generated via:
`skip-chars-forward`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun skip-chars-forward-eg() (skip-chars-forward "aeiou" 3))`
generates:

```
PC  Byte  Instruction
 0  192   constant[0] "aeiou"
 1  193   constant[1] 3
 2  119   skip-chars-forward
 3  135   return

Constants Vector: ["aeiou" 3]
```

skip-chars-backward (120)

`skip-chars-backward` (120)

Call `skip-chars-backward` with two stack arguments. For a description of this Emacs Lisp motion function, See Section "Skipping Characters" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (skip-chars-backward S[1] TOS); top--`.

Generated via:
`skip-chars-backward`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun skip-chars-backward-eg() (skip-chars-backward "aeiou" 3))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "aeiou"
 1  193   constant[1] 3
 2  120   skip-chars-backward
 3  135   return

Constants Vector: ["aeiou" 3]
```

narrow-to-region (125)

narrow-to-region (125)

Call `narrow-to-region` with two stack arguments. For a description of this Emacs Lisp narrowing function, See Section "Narrowing" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (narrow-to-region S[1] TOS); top--`.

Generated via:
`narrow-to-region`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun narrow-to-region-eg() (narrow-to-region 1250 1251))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1250
 1  193   constant[1] 1251
 2  125   narrow-to-region
 3  135   return

Constants Vector: [1250 1251]
```

widen (126)

`widen` (126)

Call `widen`. For a description of this Emacs Lisp function, See Section "Narrowing" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (widen)`

Generated via:
`widen`

Instruction size:
1 byte

Stack effect:
$-0+1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun widen-eg() (widen))` generates:

```
PC  Byte  Instruction
 0  126   widen
 1  135   return
```

3.6.7 Emacs Text Instructions

These instructions correspond to Emacs-specific text manipulation functions found in the "Text" chapter of the Emacs Lisp Reference Manual. They are not inlined by the bytecode interpreter, but simply call the corresponding C function.

`insert` (99)

Call `insert` with one stack argument. For a description of this Emacs Lisp text function, See Section "Inserting Text" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (insert TOS)
```

Generated via:

```
insert
```

Instruction size:

1 byte

Stack effect:

$-1 + 1$.

Example: With lexical binding in effect, `(defun insert-eg(n) (insert n))` generates:

```
PC  Byte  Instruction
 0  137   dup
 1   99   insert
 2  135   return
```

insertN (99)

`insertN` (99)

Call `insert` on up to 255 stack arguments. Note there is a special instruction when there is only one stack argument. For a description of this Emacs Lisp text function, See Section "Inserting Text" in *The GNU Emacs Lisp Reference Manual*.w

Implements:
`S[n-1] <- (insert S[n-1] S[n-2] ... TOS); top -= (n-1)`.

Generated via:
`insert`

Operand: 8-bit number of items in concat

Instruction size:
2 bytes

Stack effect:
$-n+1$ where n is the value of the instruction operand.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: With lexical binding in effect, `(defun insertN-eg(a b c) (insert a b c))` generates:

```
PC  Byte  Instruction
 0    2   stack-ref[2]
 1    2   stack-ref[2]
 2    2   stack-ref[2]
 3  177   insertN [3]
           3
 5  135   return
```

char-after (102)

`char-after` (102)

Call `char-after` with one stack argument. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`TOS <- (char-after TOS)`

Generated via:

`char-after`

Instruction size:

1 byte

Stack effect:

$-1+1$.

Example: `(defun char-after-eg() (char-after))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] nil
 1  102   char-after
 2  135   return

Constants Vector: [nil]
```

following-char (103)

`following-char` (103)

Call `following-char`. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (following-char TOS)
```

Generated via:

`following-char`

Instruction size:

1 byte

Stack effect:

$-0+1$.

Example: `(defun following-char-eg() (following-char))` generates:

```
PC  Byte  Instruction
 0  103   following-char
 1  135   return
```

preceding-char (104)

`preceding-char` (104)

Call `preceding-char`. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (preceding-char TOS)
```

Generated via:

`preceding-char`

Instruction size:

1 byte

Stack effect:

$0 + 1$.

Example: `(defun preceding-char-eg() (preceding-char))` generates:

```
PC  Byte  Instruction
 0  104   preceding-char
 1  135   return
```

current-column (105)

`current-column` (105)

Call `current-column`. For a description of this Emacs Lisp function, See Section "Counting Columns" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (current-column)
```

Generated via:

`current-column`

Instruction size:

1 byte

Stack effect:

$-0+1$.

Example: `(defun current-column-eg() (current-column))` generates:

```
PC  Byte  Instruction
 0  105   current-column
 1  135   return
```

indent-to (106)

`indent-to` (106)

Call `indent-to`. For a description of this Emacs Lisp indentation function, See Section "Indentation Primitives" in *The GNU Emacs Lisp Reference Manual*.

Implements:

`TOS <- (indent-to TOS)`

Generated via:

`indent-to`

Instruction size:

1 byte

Stack effect:

$-1+1$.

Example: `(defun indent-to-eg() (indent-to 5))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 5
 1  106   indent-to
 2  135   return

Constants Vector: [5]
```

eolp (108)

`eolp` (108)

Call `eolp`. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (eolp)
```

Generated via:

```
eolp
```

Instruction size:

1 byte

Stack effect:

$-0+1$.

Example: `(defun eolp-eg() (eolp))` generates:

```
PC  Byte  Instruction
 0  108   eolp
 1  135   return
```

eobp (109)

`eobp` (109)

Call `eobp`. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (eobp)
```

Generated via:

```
eobp
```

Instruction size:

1 byte

Stack effect:

$-0+1$.

Example: `(defun eobp-eg() (eobp))` generates:

```
PC  Byte  Instruction
 0  109   eobp
 1  135   return
```

bolp (110)

`bolp` (110)

Call `bolp`. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`TOS <- (bolp)`

Generated via:
`bolp`

Instruction size:
1 byte

Stack effect:
$-0+1$.

Example: `(defun bolp-eg() (bolp))` generates:

```
PC  Byte  Instruction
 0  110   bolp
 1  135   return
```

bobp (111)

bobp (111)

Call `bobp`. For a description of this Emacs Lisp text function, See Section "Examining Text Near Point" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
TOS <- (bobp)
```

Generated via:

```
bobp
```

Instruction size:

1 byte

Stack effect:

$-0+1$.

Example: `(defun bobp-eg() (bobp))` generates:

```
PC  Byte  Instruction
 0  111   bobp
 1  135   return
```

delete-region (124)

`delete-region` (124)

Call `delete-region` with two stack arguments. For a description of this Emacs Lisp deletion function, See Section "Deleting Text" in *The GNU Emacs Lisp Reference Manual*.

Implements:
`S[1] <- (delete-region S[1] TOS); top--`.

Generated via:
`delete-region`.

Instruction size:
1 byte

Stack effect:
$-2+1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun delete-region-eg() (delete-region 1240 1241))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1240
 1  193   constant[1] 1241
 2  124   delete-region
 3  135   return

Constants Vector: [1240 1241]
```

end-of-line (127)

`end-of-line` (127)

Call `end-of-line` with one stack argument. For a description of this Emacs Lisp motion function, See Section "Motion by Text Lines" in *The GNU Emacs Lisp Reference Manual*.

Implements:

```
(end-of-line TOS; top--
```

Generated via:

`delete-region`.

Instruction size:

1 byte

Stack effect:

$-1+0-$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun end-of-line-eg() (end-of-line))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] nil
 1  127   end-of-line
 2  135   return

Constants Vector: [nil]
```

3.6.8 Emacs Misc Function Instructions

These instructions correspond to miscellaneous Emacs-specific functions. They are not inlined by the bytecode interpreter, but simply call the corresponding C function.

`char-syntax` (122)

Call `char-syntax` with one stack argument.

Implements:
`TOS <- (char-syntax TOS)`

Generated via:
`char-syntax`

Instruction size:
1 byte

Stack effect:
$-1 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun char-syntax-eg() (char-syntax ?a))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 97
 1  122   char-syntax
 2  135   return

Constants Vector: [97]
```

unwind-protect (142)

`unwind-protect` (142)

To be completed...

Implements:
`(unwind-protect)`.

Generated via:
`unwind-protect`

Instruction size:
1 byte

Stack effect:
$-1+0$.

Example: When lexical binding is in effect, `(defun unwind-protect-eg() (unwind-protect (toggle-read-only)))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] #[0 "\300\207" [nil] 1]
 1  142   unwind-protect
 2  193   constant[1] toggle-read-only
 3   32   call[0]
 4   41   unbind[1]
 5  135   return

Constants Vector: [#[0 "\300\207" [nil] 1] toggle-read-only]
```

In the above, `constant[0]` is a nilary function that returns `nil`. The disassembly of `\300\207` is:

```
PC  Byte  Instruction
 0  constant  nil
 1  return
```

which is used when no unwind form was given.

save-excursion (138)

`save-excursion` (138)

Make a binding recording buffer, point, and mark.

This instruction manipulates the special-bindings stack by creating a new binding when executed. It needs to be balanced with `unbind` instructions.

Implements:
`(save-excursion)`.

Generated via:
`save-excursion`

Instruction size:
1 byte

Stack effect:
$-0+0$.

Example: When lexical binding is in effect, `(defun save-excursion-eg() (save-excursion 1380))` generates:

```
PC  Byte  Instruction
 0  138   save-excursion
 1  192   constant[0] 1380
 2   41   unbind[1]
 3  135   return

Constants Vector: [1380]
```

set-marker (147)

`set-marker` (147)

Call `set-marker` with three stack arguments.

Implements:
`S[2] <- (set-marker S[2] S[1] TOS); top -= 2`.

Generated via:
`set-marker`

Instruction size:
1 byte

Stack effect:
$-3 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: When lexical binding is in effect, `(defun set-marker-eg(marker position) (set-marker marker position))` generates:

```
PC  Byte  Instruction
 0    1   stack-ref[1]
 1    1   stack-ref[1]
 2  192   constant[0] nil
 3  147   set-marker
 4  135   return

Constants Vector: [nil]
```

match-beginning (148)

`match-beginning` (148)

Call `match-beginning` with one stack argument.

Implements:
`TOS <- (match-beginning TOS)`

Generated via:
`match-beginning`

Instruction size:
1 byte

Stack effect:
$-1 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun match-beginning-eg() (match-beginning 1))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  148   match-beginning
 2  135   return

Constants Vector: [1]
```

match-end (149)

`match-end` (149)

Call `match-end` with one stack argument.

Implements:
`TOS <- (match-end TOS)`

Generated via:
`match-end`

Instruction size:
1 byte

Stack effect:
$-1 + 1$.

Added in: Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

Example: `(defun match-end-eg() (match-end 1))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  148   match-end
 2  135   return

Constants Vector: [1]
```

3.7 Stack-Manipulation Instructions

`discard` (136)

Discard one value from the stack.

Implements:
`top--`

Instruction size:
1 byte

Generated via:
Function calls that do not use the returned value; the end of `let` forms in lexical binding to remove locally-bound variables.

Stack effect:
$-1 + 0$.

Example: `(defun discard-eg() (exchange-point-and-mark) (point))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] exchange-point-and-mark
 1   32   call[0]
 2  136   discard
 3   96   point
 4  135   return

Constants Vector: [exchange-point-and-mark]
```

discardN (180)

`discardN` (180)

Discards up to 127 arguments from the stack. Note there is a special instruction when there is only one argument.

Implements:
`S[OPERAND] <- TOS; top -= OPERAND;`

operand: 7-bit number of items to discard. The top 8th bit when set indicates to keep the old TOS value after discarding.

Instruction size:
2 bytes

Generated via:
Function calls that do not use the returned value; the end of `let` forms in lexical binding with optimization to remove locally-bound variables.

Stack effect:
$-OPERAND + 0$

Type Information:
before: `TOS: Object; S[1]: Object; ... S[OPERAND-1]: Object`

Added in: Emacs 24.1. See Section 4.7 [Emacs 24], page 172.

Example: When lexical binding is in effect and optimization are in effect,

```
(1+ (let ((a 1) (_b) (_c)) a)))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] 1
 1  193   constant[1] nil
 2  137   dup
 3    2   stack-ref[2]
 4  182   discardN [131]
         131
 6   84   add1
 7  135   return

Constants Vector: [1 nil]
```

dup (137)

dup (137)

Make a copy of the top-of-stack value and push that onto the top of the evaluation stack.

Implements:
`top++; TOS <- S[1]`

Generated via:
`setq` in dynamic bindings to set a value and then use it. In lexical binding, to use the first argument parameter.

Instruction size:
1 byte

Stack effect:
$-1 + 2$.

Type Information:
before: `TOS: Object` **after:** `TOS: Object; S[1]: Object`

Example: When lexical binding is in effect,

```
(defun dup-eg(n) n)
```

generates:

```
PC  Byte  Instruction
 0  137   dup  ;; duplicates top of stack, argument n
 1  135   return
```

stack-set (178)

`stack-set` (178)

Sets a value on the evaluation stack to TOS.

Implements:

`S[i] <- TOS; top--` where `i` is the value of the instruction operand.

Note that `stack-set[0]` has the same effect as `discard`, but does a little more work to do this. `stack-set[1]` has the same effect as `discardN 1` with the top bit of `discardN` set to preserve `TOS`.

Generated via:

`let`, `let*` and lambda arguments.

Operand: A 8-bit integer stack index

Instruction size:

2 bytes

Stack effect:

$-1 + 0$.

Added in: Emacs 24.1. See Section 4.7 [Emacs 24], page 172.

Example: When lexical binding is in effect and optimization

```
defun stack-set-eg()
(let ((a 5) a)))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] 5
 1  193   constant[1] nil
 2  193   constant[1] nil
 3  178   stack-set [2]
           2
 5  136   discard
 6  135   return

Constants Vector: [5 nil]
```

stack-set2 (179)

`stack-set2` (179)

Implements:

`S[i] <- TOS; top--` where `i` is the value of the instruction operand.

Note that `stack-set2[0]` has the same effect as `discard`, but does a little more work to do this. `stack-set2[1]` has the same effect as `discardN 1` with the top bit of `discardN` set to preserve `TOS`.

Generated via:

`let`, `let*` and lambda arguments.

Operand: A 16-bit integer stack index

Instruction size:

3 bytes

Stack effect:

$-1 + 0$.

Added in: Emacs 24.1. See Section 4.7 [Emacs 24], page 172.

Example: ??

3.8 Obsolete or Unused Instructions

These instructions are not generated by Emacs Lisp bytecode generation. In some cases, they were generated in a older version of Emacs. In some cases the instructions may have been planned to being used but never were. In some cases, the instructions are still handled if they appear (such as from older bytecode), but in other cases they are no longer accepted by the interpreter.

It is also possible that code outside of the Emacs Lisp distribution generates these instructions.

`save-current-buffer` (97)

Replaced by `save-current-buffer-1`. See [save-current-buffer-1], page 125.

`mark` (97)

Used in V 17; obsolete in Emacs 18.31. See Section 4.1 [Emacs 18], page 166.

`scan-buffer` (107)

Obsolete in Emacs 18.31. See Section 4.1 [Emacs 18], page 166.

`read-char` (114)

`set-mark` (115)

Obsolete in Emacs 18.31. See Section 4.1 [Emacs 18], page 166.

`interactive-p` (116)

`save-window-excursion` (139)

Call `save-window-excursion`.

Implements:

`(save-window-excursion BLOCK)`

Generated via:

`save-window-excursion`

Instruction size:

1 byte

Stack effect:

$-1 + 0$.

Obsolete since:

Emacs 24.1. See Section 4.7 [Emacs 24], page 172. Now generates a sequence of bytecode that includes calls to `current-window-configuration` and `set-window-configuration`

Example:

```
(defun save-window-excursion()
   (save-window-excursion 1390))
```

generates:

```
PC  Byte  Instruction
 0  192   constant[0] (1390)
 1  139   save-window-excursion
 2  135   return

Constants Vector: [(1390)]
```

condition-case (143)

Replaced by `pushconditioncase`. See [pushconditioncase], page 53,

Implements:
?

Generated via:
?

Instruction size:
1 byte

Stack effect:
$-2+1$.

Obsolete since:
Emacs 24.4. See Section 4.7 [Emacs 24], page 172.

Example: `(defun condition-case-eg() (?))` generates:

temp-output-buffer-setup (144)

Implements:
Setup for `with-output-to-temp-buffer`.

Generated via:
`with-output-to-temp-buffer`

Instruction size:
1 byte

Stack effect:
$-1+0$.

Obsolete since:
Emacs 24.1. See Section 4.7 [Emacs 24], page 172.

Example: `(defun wottb-eg () (with-output-to-temp-buffer "wottb" 5))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "wottb"
 1  144   temp-output-buffer-setup
 2  193   constant[1] 5
 3  145   temp-output-buffer-show
 4  135   return

Constants Vector: ["wottb" 5]
```

`temp-output-buffer-show` (145)

Implements:
Finishing code of `with-output-to-temp-buffer`.

Generated via:
`with-output-to-temp-buffer`

Instruction size:
1 byte

Stack effect:
$-0+0$.

Obsolete in:
Emacs 24.1. See Section 4.7 [Emacs 24], page 172.

Example: `(defun wottb-eg () (with-output-to-temp-buffer "wottb" 5))` generates:

```
PC  Byte  Instruction
 0  192   constant[0] "wottb"
 1  144   temp-output-buffer-setup
 2  193   constant[1] 5
 3  145   temp-output-buffer-show
 4  135   return

Constants Vector: ["wottb" 5]
```

`unbind-all` (146)

Introduced in Emacs 19.34 for tail-recursion elimination by jwz, but never used. See Section 4.2 [Emacs 19], page 167.

3.8.12 Relative Goto Instructions

In Emacs 19.34, Hallvard Furuseth introduced relative goto instructions. However, they have rarely have been generated in bytecode, and currently are not.

From Hallvard:

> Relative jump instructions: There's an apparently unanswered mail in my mailbox about them being buggy and asking how they worked. I expect they got disabled for that reason rather than someone trying to debug. I don't remember why I introduced them. Maybe just a space optimization and not worth the effort. I hadn't quite learned that there are times to not bother optimizing:-)

There have been reports however that others have used these instructions in alternate languages that generate bytecode.

`Rgoto` (170)

Relative jump version of see [goto], page 55.

Introduced but unused in Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

`Rgotoifnil` (171)

Relative jump version of see [goto-if-nil], page 56.

Introduced but unused in Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

`Rgotoifnonnil` (172)

Relative-jump version of see [goto-if-not-nil], page 57.

Introduced but unused in Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

`Rgotoifnilelsepop` (173)

Relative-jump version of see [goto-if-nil-else-pop], page 58.

Introduced but unused in Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

`Rgotoifnonnilelsepop` (174)

Relative-jump version of see [goto-if-not-nil-else-pop], page 59.

Introduced but unused in Emacs 19.34. See Section 4.2 [Emacs 19], page 167.

4 Instruction Changes Between Emacs Releases

The infomation in this chapter may not be as precise or granular as it could be. I invite those who want more detail to look at Lars Brinkhoff's Emacs History (`https://github.com/larsbrinkhoff/emacs-history`) project.

Feel free suggest changes, as github pull requests, to make this chapter more detailed.

4.1 After 16 and Starting in 18.31

The following instructions were renamed:

- `dot` becomes `mark` (97). See [mark], page 162.
- `dot-min` becomes `point-min` (100). See [point-min], page 130.
- `dot-max` becomes `point-max` (101). See [point-max], page 129.

The following instructions became obsolete:

- `mark` (97). See [mark], page 162.
- `scan-buffer` (107). See [scan-buffer], page 162.
- `set-mark` (115). See [set-mark], page 162.

Version 18 Release History

- Emacs 18.31 was released Nov 23, 1986
- Emacs 18.32 was released Dec 6, 1986
- Emacs 18.33 was released Dec 12, 1986
- Emacs 18.35 was released Jan 5, 1987
- Emacs 18.36 was released Jan 21, 1987
- Emacs 18.37 was released Feb 11, 1987
- Emacs 18.38 was released Mar 3, 1987
- Emacs 18.39 was released May 14, 1987
- Emacs 18.40 was released Mar 18, 1987
- Emacs 18.41 was released Mar 22, 1987
- Emacs 18.44 was released Apr 15, 1987
- Emacs 18.46 was released Jun 8, 1987
- Emacs 18.47 was released Jun 15, 1987
- Emacs 18.48 was released Aug 30, 1987
- Emacs 18.49 was released Sep 17, 1987
- Emacs 18.50 was released Feb 13, 1988
- Emacs 18.51 was released May 6, 1988
- Emacs 18.52 was released Aug 31, 1988
- Emacs 18.59 was released Oct 30, 1988

4.2 After 18.59 and Starting 19.34

Jamie Zawinski and Hallvard Breien Furuseth made major changes and additions to the bytecode interpreter.

From Hallvard:

> Originally I just generalized some stuff, made bytecomp output byte-code at file level, added code to skip compiling very small code snippets when introducing the byte-code call would be a pessimization (looks like this has been partly reverted now that there are #[function objects]), and I made some other simple optimizations.
>
> Bytecomp compiled directly to bytecode. Jamie Zawinski invented the intermediate stage lapcode which made more thorough optimization possible (byte-opt), and we got together about optimizing more.

The following instructions were added:

- `mult` (97). See [mult], page 109.
- `forward-char` (117). See [forward-char], page 131.
- `forward-word` (118). See [forward-word], page 132.
- `skip-chars-forward` (119). See [skip-chars-forward], page 134.
- `skip-chars-backward` (120). See [skip-chars-backward], page 135.
- `forward-line` (121). See [forward-line], page 133.
- `char-syntax` (122). See [char-syntax], page 151.
- `buffer-substring` (123). See [buffer-substring], page 126.
- `delete-region` (124). See [delete-region], page 149.
- `narrow-to-region` (125). See [narrow-to-region], page 136.
- `widen` (126). See [widen], page 137.
- `end-of-line` (127). See [end-of-line], page 150.
- `unbind-all` (146). See [unbind-all], page 164.
- `set-marker` (147). See [set-marker], page 154.
- `match-beginning` (148). See [match-beginning], page 155.
- `match-end` (149). See [match-end], page 156.
- `upcase` (150). See [upcase], page 119.
- `downcase` (151). See [downcase], page 120.
- `stringeqlsign` (152). See [stringeqlsign], page 121.
- `stringlss` (153). See [stringlss], page 122.
- `equal` (154). See [equal], page 74.
- `nthcdr` (155). See [nthcdr], page 91.
- `elt` (156). See [elt], page 92.
- `member` (157). See [member], page 75.
- `assq` (158). See [assq], page 76.
- `nreverse` (159). See [nreverse], page 93.

- `setcar` (160). See [setcar], page 94.
- `setcdr` (161). See [setcdr], page 95.
- `car-safe` (162). See [car-safe], page 96.
- `cdr-safe` (163. See [cdr-safe], page 97.
- `nconc` (164). See [nconc], page 98.
- `quo` (165). See [quo], page 112.
- `rem` (166). See [rem], page 113.
- `numberp` (167). See [numberp], page 77.
- `integerp` (162). See [integerp], page 78.
- `Rgoto` (170). See [Rgoto], page 165.
- `Rgotoifnil` (171). See [Rgotoifnil], page 165.
- `Rgotoifnonnil` (172). See [Rgotoifnonnil], page 165.
- `Rgotoifnilelsepop` (173). See [Rgotoifnilelsepop], page 165.
- `Rgotoifnonnilelsepop` (174). See [Rgotoifnonnilelsepop], page 165.
- `listN` (175). See [listN], page 87.
- `concatN` (176). See [concatN], page 118.
- `insertN` (177). See [insertN], page 139.

Instruction `unbind-all` was added to support tail-recursion removal. However this was never subsequently implemented; so this intruction was never generated.

Starting in this version, unless C prepocessor variable `BYTE_CODE_SAFE` (off by default) is defined, the obsolete instructions listed in 18.59 are not implemented.

The following obsolete instructions throw an error when `BYTE_CODE_SAFE` is defined:

- `mark` (97)
- `scan-buffer` (107)
- `set-mark` (115)

Bytecode meta-comments look like this:

```
;;; compiled by rms@psilocin.gnu.ai.mit.edu on Mon Jun 10 17:37:37 1996
;;; from file /home/fsf/rms/e19/lisp/bytecomp.el
;;; emacs version 19.31.2.
;;; bytecomp version FSF 2.10
;;; optimization is on.
;;; this file uses opcodes which do not exist in Emacs 18.
```

Version 19 Release History

- Emacs 19.7 was released May 22 1993
- Emacs 19.8 was released May 25 1993
- Emacs 19.9 was released May 27 1993
- Emacs 19.10 was released May 30 1993
- Emacs 19.11 was released Jun 1, 1993

- Emacs 19.12 was released Jun 1, 1993
- Emacs 19.13 was released Jun 8, 1993
- Emacs 19.14 was released Jun 17, 1993
- Emacs 19.15 was released Jun 19, 1993
- Emacs 19.16 was released Jul 6, 1993
- Emacs 19.17 was released Jul 7, 1993
- Emacs 19.18 was released Aug 8, 1993
- Emacs 19.19 was released Aug 14, 1993
- Emacs 19.20 was released Nov 11, 1993
- Emacs 19.21 was released Nov 16, 1993
- Emacs 19.22 was released Nov 27, 1993
- Emacs 19.23 was released May 17, 1994
- Emacs 19.24 was released May 23, 1994
- Emacs 19.25 was released May 30, 1994
- Emacs 19.26 was released Sep 7, 1994
- Emacs 19.27 was released Sep 11, 1994
- Emacs 19.29 was released Jun 19, 1995
- Emacs 19.30 was released Nov 24, 1995
- Emacs 19.31 was released May 25, 1996
- Emacs 19.31 was released May 25, 1996
- Emacs 19.32 was released Aug 7, 1996
- Emacs 19.33 was released Sept 11, 1996

The Emacs Lisp tarball for 19.2 is Aug, 1992. (The tarball date for 19.2 is much later; and even after the date on the 20.1 tarball.)

4.3 After 19.34 and Starting in 20.1

`save-current-buffer` (97). See [save-current-buffer], page 162, and `save-current-buffer-1` (114) do the same thing, but the former is deprecated. The latter opcode replaces `read-char` which was not generated since v19.

I am not sure why the change; changing this opcode number however put it next to other buffer-related opcodes.

Bytecode meta-comments look like this:

```
;;; Compiled by rms@psilocin.gnu.ai.mit.edu on Sun Aug 31 13:07:37 1997
;;; from file /home/fsf/rms/e19/lisp/emacs-lisp/bytecomp.el
;;; in Emacs version 20.0.97.1
;;; with bytecomp version 2.33
;;; with all optimizations.
;;; This file uses opcodes which do not exist in Emacs 18.
```

Version 20 Release History

- Emacs 20.1 was released Sep 15, 1997
- Emacs 20.2 was released Sep 19, 1997
- Emacs 20.3 was released Aug 19, 1998
- Emacs 20.4 was released Jul 14, 1999

4.4 After 20.1 and Starting in 21.1

There were no instruction changes. However there were major changes in the bytecode interpreter.

An instruction with opcode 0 causes an abort.

Bytecode meta-comments look like this:

```
;;; Compiled by pot@pot.cnuce.cnr.it on Tue Mar 18 15:36:26 2003
;;; from file /home/pot/gnu/emacs-pretest.new/lisp/emacs-lisp/bytecomp.el
;;; in Emacs version 21.3
;;; with bytecomp version 2.85.4.1
;;; with all optimizations.
```

Version 21 Release History

- Emacs 21.1 was released Oct 20, 2001
- Emacs 21.2 was released Mar 16, 2002
- Emacs 21.3 was released Mar 18, 2003
- Emacs 21.4 was released Feb 6, 2005

4.5 After 21.4 and Starting in 22.1

There were no instruction changes.

The bytecode meta-comment no longer includess the bytecomp version used.

Bytecode meta-comments look like this:

```
;;; Compiled by cyd@localhost on Sat Jun  2 00:54:30 2007
;;; from file /home/cyd/emacs/lisp/emacs-lisp/bytecomp.el
;;; in Emacs version 22.1
;;; with all optimizations.

;;; This file uses dynamic docstrings, first added in Emacs 19.29.
```

Version 22 Release History

- Emacs 22.1 was released Jun 02, 2007
- The Emacs 22.2 tarball is dated Mar 26 2008
- The Emacs 22.3 tarball is dated Sep 05 2008

4.6 After 22.3 and Starting in 23.1

There were no instruction changes.

Bytecode meta-comments look like this:

```
;;; Compiled by cyd@furry on Wed Jul 29 11:15:02 2009
;;; from file /home/cyd/emacs/lisp/emacs-lisp/bytecomp.el
;;; in Emacs version 23.1
;;; with all optimizations.

;;; This file uses dynamic docstrings, first added in Emacs 19.29.
```

Version 23 Release History

- Emacs 23.1 was released Jul 29, 2009
- Emacs 23.2 was released May 7, 2010
- Emacs 23.3 was released Mar 7, 2011
- The Emacs 23.4 tarball is dated Jan 28, 2012

4.7 After 23.4 and Starting in 24.1

An error is thrown for unknown bytecodes rather than aborting.

The following instructions were added:

- `stack-set` (178). See [stack-set], page 160.
- `stack-set2`, (179). See [stack-set2], page 161.
- `discardN`, (180). See [discardN], page 158.

Unless C preprocessor variable `BYTE_CODE_SAFE` (off by default) is defined, obsolete instructions below and from earlier versions are not implemented.

- `temp-output-buffer-setup` (144). See [temp-output-buffer-setup], page 163.
- `temp-output-buffer-show` (145). See [temp-output-buffer-show], page 164.
- `save-window-excursion` (139). See [save-window-excursion], page 162.

Instruction `unbind-all`, which never was generated, was marked obsolete in this version.

The bytecode meta-comment no longer who user/hostname compiled and at what time. A message indicating whether utf-8 non-ASCII characters is used is included.

The following instructions were added in 24.4:

- `pophandler` (48). See [pophandler], page 52.
- `pushconditioncase` (49). See [pushconditioncase], page 53.
- `pushcatch` (50). See [pushcatch], page 54.

Bytecode meta-comments look like this:

```
;;; from file /misc/emacs/bzr/emacs24-merge/lisp/emacs-lisp/bytecomp.el
;;; in Emacs version 24.3
;;; with all optimizations.

;;; This file uses dynamic docstrings, first added in Emacs 19.29.

;;; This file does not contain utf-8 non-ASCII characters,
;;; and so can be loaded in Emacs versions earlier than 23.
```

Version 24 Release History

- The Emacs 24.1 tarball is dated Jun 10, 2012
- The Emacs 24.2 tarball is dated Aug 27, 2012
- Emacs 24.3 was released Mar 11, 2013
- Emacs 24.4 was released Oct 20, 2014
- Emacs 24.5 was released Apr 10, 2015

4.8 After 24.5 and Starting in 25.1

Instruction 0 becomes an error rather than aborting emacs.

A number of changes were made to `bytecode.c`.

The bytecode meta-comment no longer includes the source-code path.

Bytecode meta-comments look like this:

```
;;; Compiled
;;; in Emacs version 25.2
;;; with all optimizations.

;;; This file uses dynamic docstrings, first added in Emacs 19.29.

;;; This file does not contain utf-8 non-ASCII characters,
;;; and so can be loaded in Emacs versions earlier than 23.
```

Version 25 Release History

- Emacs 25.1 was released Sep 16, 2016
- The Emacs 25.2 tarball is dated Apr 21, 2017
- Emacs 25.3 was released Sep 11, 2017

4.9 After 25.3 and Starting in 26.1

- Emacs 26.1 tarball is dated May 28, 2018
- Emacs 26.2 tarball is dated Apr 12, 2019
- Emacs 26.3 tarball is dated Aug 28, 2019

The following instruction was added:

- `switch` (183) See commit 88549ec38e9bb30e338a9985d0de4e6263b40fb7.

4.10 After 26.3 and Starting in 27.1

Version 27 Release History

- Emacs 27.1 tarball is dated Aug 10, 2020

No changes yet.

5 Opcode Table

In the table below, a * before the intruction name indicates an obsolete instruction, or instruction that is no longer generated by the bytecode compiler. On the other hand, ! indicates not just an obsolete instruction, but one that no longer is interpreted. See Section 3.1 [Instruction-Description Format], page 39, for abbreviations used here, a description of how to interpret an opcode when it contains an index, and for a description of how to interpret the stack-effect field.

5.1 Opcodes (0000-0077)

Oct	Dec	Instruction	Size	Description	Stack
00	0			An error. Before 25.1 it is an immediate program abort! Logically `stack-ref[0]` but `dup` should be used instead.	
01	1	`stack-ref[1]`	1	See [stack-ref], page 42.	+1
02	2	`stack-ref[2]`	1	See [stack-ref], page 42.	+1
03	3	`stack-ref[3]`	1	See [stack-ref], page 42.	+1
04	4	`stack-ref[4]`	1	See [stack-ref], page 42.	+1
05	5	`stack-ref[5]`	1	See [stack-ref], page 42.	+1
06	6	`stack-ref[6]`	2	See [stack-ref], page 42.	+1
07	7	`stack-ref[7]`	3	See [stack-ref], page 42.	+1
010	8	`varref[0]`	1	See [varref], page 44.	+1
011	9	`varref[1]`	1	See [varref], page 44.	+1
012	10	`varref[2]`	1	See [varref], page 44.	+1
013	11	`varref[3]`	1	See [varref], page 44.	+1
014	12	`varref[4]`	1	See [varref], page 44.	+1
015	13	`varref[5]`	1	See [varref], page 44.	+1
016	14	`varref[6]`	2	See [varref], page 44.	+1
017	15	`varref[7]`	3	See [varref], page 44.	+1
020	16	`varset[0]`	1	See [varset], page 45.	−1
021	17	`varset[1]`	1	See [varset], page 45.	−1
022	18	`varset[2]`	1	See [varset], page 45.	−1
023	19	`varset[3]`	1	See [varset], page 45.	−1
024	20	`varset[4]`	1	See [varset], page 45.	−1
025	21	`varset[5]`	1	See [varset], page 45.	−1
026	22	`varset[6]`	2	See [varset], page 45.	−1
027	23	`varset[7]`	3	See [varset], page 45.	−1
030	24	`varbind[0]`	1	See [varbind], page 46.	−1 + 0

031	25	varbind[1]	1	See [varbind], page 46.	$-1+0$
032	26	varbind[2]	1	See [varbind], page 46.	$-1+0$
033	27	varbind[3]	1	See [varbind], page 46.	$-1+0$
034	28	varbind[4]	1	See [varbind], page 46.	$-1+0$
035	29	varbind[5]	1	See [varbind], page 46.	$-1+0$
036	30	varbind[6]	2	See [varbind], page 46.	$-1+0$
037	31	varbind[7]	3	See [varbind], page 46.	-1
040	32	call0	1	See [call], page 47.	$-1+1$
041	33	call1	1	See [call], page 47.	$-2+1$
042	34	call2	1	See [call], page 47.	$-3+1$
043	35	call3	1	See [call], page 47.	$-4+1$
044	36	call4	1	See [call], page 47.	$-5+1$
045	37	call5	1	See [call], page 47.	$-6+1$
046	38	call6	2	See [call], page 47.	$-n-1+1$
047	39	call7	3	See [call], page 47.	$-n-1+1$
050	40	unbind0	1	See [unbind], page 49.	-0
051	41	unbind1	1	See [unbind], page 49.	-0
052	42	unbind2	1	See [unbind], page 49.	-0
053	43	unbind3	1	See [unbind], page 49.	-0
054	44	unbind4	1	See [unbind], page 49.	-0
055	45	unbind5	1	See [unbind], page 49.	-0
056	46	unbind6	2	See [unbind], page 49.	-0
057	47	unbind7	3	See [unbind], page 49.	-0
060	48	pophandler	1		-0
061	49	conditioncase	3		$-1+\phi(0,+1)$
062	50	pushconditioncase	1		-0
063	51			Unused	
064	52			Unused	
065	53			Unused	
066	54			Unused	
067	55			Unused	
070	56	nth	1	See [nth], page 79.	$-2+1$
071	57	symbolp	1	See [symbolp], page 62.	$-1+1$
072	58	consp	1	See [consp], page 63.	$-1+1$
073	59	stringp	1	See [stringp], page 64.	$-1+1$
074	60	listp	1	See [listp], page 65.	$-1+1$
075	61	eq	1	See [eq], page 66.	$-2+1$
076	62	memq	1	See [memq], page 67.	$-2+1$
077	63	not	1	See [not], page 68.	$-1+1$

5.2 Opcodes (0100-0177)

Oct	Dec	Instruction	Size	Description	Stack
0100	64	car	1	See [car], page 80.	−1 + 1
0101	65	cdr	1	See [cdr], page 81.	−1 + 1
0102	66	cons	1	See [cons], page 82.	−2 + 1
0103	67	list1	1	See [list1], page 83.	−1 + 1
0104	68	list2	1	See [list2], page 84.	−2 + 1
0105	69	list3	1	See [list3], page 85.	−3 + 1
0106	70	list4	1	See [list4], page 86.	−4 + 1
0107	71	length	1	See [length], page 88.	−1 + 1
0110	72	aref	1	See [aref], page 89.	−2 + 1
0111	73	aset	1	See [aset], page 90.	−3 + 1
0112	74	symbol-value	1	See [symbol-value], page 69.	−1 + 1
0113	75	symbol-function	1	See [symbol-function], page 70.	−1 + 1
0114	76	set	1	See [set], page 71.	−2 + 1
0115	77	fset	1	See [fset], page 72.	−2 + 1
0116	78	get	1	See [get], page 73.	−2 + 1
0117	79	substring	1	See [substring], page 114.	−3 + 1
0120	80	concat2	1	See [concat2], page 115.	−2 + 1
0121	81	concat3	1	See [concat3], page 116.	−3 + 1
0122	82	concat4	1	See [concat4], page 117.	−4 + 1
0123	83	sub1	1	See [sub1], page 99.	−1 + 1
0124	84	add1	1	See [add1], page 100.	−1 + 1
0125	85	eqlsign	1	See [eqlsign], page 101.	−2 + 1
0126	86	gtr	1	See [gtr], page 102.	−2 + 1
0127	87	lss	1	See [lss], page 103.	−2 + 1
0130	88	leq	1	See [leq], page 104.	−2 + 1
0131	89	geq	1	See [geq], page 105.	−2 + 1
0132	90	diff	1	See [diff], page 106.	−2 + 1
0133	91	negate	1	See [negate], page 107.	−1 + 1
0134	92	plus	1	See [plus], page 108.	−2 + 1
0135	93	max	1	See [max], page 110.	−2 + 1
0136	94	min	1	See [min], page 111.	
0135	95	mult	1	See [mult], page 109.	−2 + 1
0140	96	point	1	See [point], page 127.	
0141	*97	*mark	1	See [mark], page 162,	−0 + 1
0142	98	goto-char	1	See [goto-char], page 128.	−1 + 1
0143	99	insert	1	See [insert], page 138.	−1 + 1
0145	100	point-max	1	See [point-max], page 129.	−0 + 1
0146	101	point-min	1	See [point-min], page 130.	−0 + 1
0144	102	char-after	1	See [char-after], page 140.	−1 + 1

0147	103	following-char	1	See [following-char], page 141.	$-0+1$
0150	104	preceding-char	1	See [preceding-char], page 142.	$-0+1$
0151	105	current-column	1	See [current-column], page 143.	$-0+1$
0152	106	indent-to	1	See [indent-to], page 144.	$-1+1$
0153	*107	*scan-buffer		See [scan-buffer], page 162.	
0154	108	eolp	1	See [eolp], page 145.	$-0+1$
0155	109	eobp	1	See [eobp], page 146.	$-0+1$
0156	110	bolp	1	See [bolp], page 147.	$-0+1$
0157	111	bobp	1	See [bobp], page 148.	$-0+1$
0160	112	current-buffer	1	See [current-buffer], page 123.	$-0+1$
0161	113	set-buffer	1	See [set-buffer], page 124.	$-1+1$
0162	114	save-current-buffer-1	1	See [save-current-buffer-1], page 125.	$-0+1$?
0162	*114	*read-char	1	See [read-char], page 162.	$+1$
0163	*115	*set-mark	1	See [set-mark], page 162.	-0
0164	*116	*interactive-p	1	See [interactive-p], page 162.	$+1$
0165	117	forward-char	1	See [forward-char], page 131.	$-1+1$
0166	118	forward-word	1	See [forward-word], page 132.	$-1+1$
0167	119	skip-chars-forward	1	See [skip-chars-forward], page 134.	$-2+1$
0170	120	skip-chars-backward	1	See [skip-chars-backward], page 135.	$-2+1$
0171	121	forward-line	1	See [forward-line], page 133.	$-1+1$
0172	122	char-syntax	1	See [char-syntax], page 151.	$-1+1$
0173	123	buffer-substring	1	See [buffer-substring], page 126.	$-2+1$
0174	124	delete-region	1	See [delete-region], page 149.	$-2+1$
0175	125	narrow-to-region	1	See [narrow-to-region], page 136.	$-2+1$
0176	126	widen	1	See [widen], page 137.	$-0+1$
0177	127	end-of-line	1	See [end-of-line], page 150.	$-1+1$

5.3 Opcodes (0200-0277)

Oct	Dec	Instruction	Size	Description	Stack
0201	129	constant2	1	See [constant2], page 51.	+1
0202	130	goto	1	See [goto], page 55.	$-1+0$
0203	131	goto-if-nil	1	See [goto-if-nil], page 56.	$-1+0$
0204	132	goto-if-not-nil	1	See [goto-if-not-nil], page 57.	$-1+0$
0205	133	goto-if-nil-else-pop	1	See [goto-if-nil-else-pop], page 58.	$\phi(-1,0)+0$
0206	134	goto-if-not-nil-else-pop	1	See [goto-if-not-nil-else-pop], page 59.	$\phi(-1,0)+0$
0207	135	return	1	See [return], page 60.	$-1+0$
0210	136	discard	1	See [discard], page 157.	$-n+0$
0211	137	dup	1	See [dup], page 159.	$-1+2$
0212	138	save-excursion	1	See [save-excursion], page 153.	$-0+1$
0213	*139	*save-window-excursion	1	See [save-window-excursion], page 162.	$-1+0$
0214	140			Unused	
0215	141			Unused	
0216	142	unwind-protect	See [unwind-protect], page 152.		$-1+0$
0217	*143	*condition-case	1	See [condition-case], page 163.	$-1+1$
0220	144	temp-output-buffer-setup	1	See [temp-output-buffer-setup], page 163.	$-1+0$
0221	145	temp-output-buffer-show	1	See [temp-output-buffer-show], page 164.	$-0+0$
0222	146			Unused	
0223	147			Unused	
0256	174			Unused	
0257	175	listN	2	See [listN], page 87.	$-n+1$
0260	176	concatN	2	See [concatN], page 118.	$-n+1$
0261	177	insertN	2	See [insertN], page 139.	$-n+1$
0262	178	stack-set	1	See [stack-set], page 160.	-1
0263	179	stack-set2	1	See [stack-set2], page 161.	-1
0222	*146	*unbind-all	1	See [unbind-all], page 164.	-0
0223	147	set-marker	1	See [set-marker], page 154.	$-3+1$

0224	148	match-beginning	1	See [match-beginning], page 155.	$-1+1$
0225	149	match-end	1	See [match-end], page 156.	$-1+1$
0226	150	upcase	1	See [upcase], page 119.	$-1+1$
0227	151	downcase	1	See [downcase], page 120.	$-1+1$
0230	152	stringeqlsign	1	See [stringeqlsign], page 121.	$-2+1$
0231	153	stringlss	1	See [stringlss], page 122.	$-2+1$
0232	154	equal	1	See [equal], page 74.	$-2+1$
0233	155	nthcdr	1	See [nthcdr], page 91.	$-2+1$
0234	156	elt	1	See [elt], page 92.	$-2+1$
0235	157	member	1	See [member], page 75.	$-2+1$
0236	158	assq	1	See [assq], page 76.	$-2+1$
0237	159	nreverse	1	See [nreverse], page 93.	$-1+1$
0240	160	setcar	1	See [setcar], page 94.	$-2+1$
0241	161	setcdr	1	See [setcdr], page 95.	$-2+1$
0242	162	car-safe	1	See [car-safe], page 96.	$-1+1$
0243	163	cdr-safe	1	See [cdr-safe], page 97.	$-1+1$
0244	164	nconc	1	See [nconc], page 98.	$-2+1$
0245	165	quo	1	See [quo], page 112.	$-2+1$
0246	166	rem	1	See [rem], page 113.	$-2+1$
0247	167	numberp	1	See [numberp], page 77.	$-1+1$
0250	168	integerp	1	See [integerp], page 78.	$-1+1$
0251	169			Unused	
0252	*170	*Rgoto	1	See [Rgoto], page 165.	$-1+0$
0253	*171	*Rgotoifnil	1	See [Rgotoifnil], page 165.	
0254	*172	*Rgotoifnonnil	1	See [Rgotoifnonnil], page 165.	$-1+0$
0255	*173	*Rgotoifnilelsepop	1	See [Rgotoifnilelsepop], page 165.	$\phi(-1,0)+0$
0256	*174	*Rgotoifnonnilelsepop	1	See [Rgotoifnonnilelsepop], page 165,	$\phi(-1,0)+0$
0257	175	listN	2	See [listN], page 87.	$-n+1$
0260	176	concatN	2	See [concatN], page 118.	
0261	177	insertN	1	See [insertN], page 139.	$-n+1$
0262	178	stack-set	1	See [stack-set], page 160.	$-0+0$
0263	179	stack-set2	2	See [stack-set2], page 161.	$-0+0$
0264	180			Unused	
0265	181			Unused	
0266	182	discardN	1	See [discardN], page 158.	$-n+0$
0267	183	switch	1	See [switch], page 61.	$-2+0$
0270	184			Unused	
0271	185			Unused	
0272	186			Unused	

0273	187	Unused
0274	188	Unused
0275	189	Unused
0276	190	Unused
0277	191	Unused

5.4 Opcodes (0300-3277) Constants

Oct	Dec	Instruction	Size	Description	Stack
0300	192	`constant[0]`	1	See [constant], page 50.	+1
0301	193	`constant[1]`	1	See [constant], page 50.	+1
0302	194	`constant[2]`	1	See [constant], page 50.	+1
0303	195	`constant[3]`	1	See [constant], page 50.	+1
0304	196	`constant[4]`	1	See [constant], page 50.	+1
0305	197	`constant[5]`	1	See [constant], page 50.	+1
0306	198	`constant[6]`	1	See [constant], page 50.	+1
0307	199	`constant[7]`	1	See [constant], page 50.	+1
0310	200	`constant[8]`	1	See [constant], page 50.	+1
0311	201	`constant[9]`	1	See [constant], page 50.	+1
0312	202	`constant[10]`	1	See [constant], page 50.	+1
0313	203	`constant[11]`	1	See [constant], page 50.	+1
0314	204	`constant[12]`	1	See [constant], page 50.	+1
0315	205	`constant[13]`	1	See [constant], page 50.	+1
0316	206	`constant[14]`	1	See [constant], page 50.	+1
0317	207	`constant[15]`	1	See [constant], page 50.	+1
0320	208	`constant[16]`	1	See [constant], page 50.	+1
0321	209	`constant[17]`	1	See [constant], page 50.	+1
0322	210	`constant[18]`	1	See [constant], page 50.	+1
0323	211	`constant[19]`	1	See [constant], page 50.	+1
0324	212	`constant[20]`	1	See [constant], page 50.	+1
0325	213	`constant[21]`	1	See [constant], page 50.	+1
0326	214	`constant[22]`	1	See [constant], page 50.	+1
0327	215	`constant[23]`	1	See [constant], page 50.	+1
0330	216	`constant[24]`	1	See [constant], page 50.	+1
0331	217	`constant[25]`	1	See [constant], page 50.	+1
0332	218	`constant[26]`	1	See [constant], page 50.	+1
0333	219	`constant[27]`	1	See [constant], page 50.	+1
0334	220	`constant[28]`	1	See [constant], page 50.	+1
0335	221	`constant[29]`	1	See [constant], page 50.	+1
0336	222	`constant[30]`	1	See [constant], page 50.	+1
0337	223	`constant[31]`	1	See [constant], page 50.	+1
0340	224	`constant[32]`	1	See [constant], page 50.	+1
0341	225	`constant[33]`	1	See [constant], page 50.	+1
0342	226	`constant[34]`	1	See [constant], page 50.	+1
0343	227	`constant[35]`	1	See [constant], page 50.	+1
0344	228	`constant[36]`	1	See [constant], page 50.	+1
0345	229	`constant[37]`	1	See [constant], page 50.	+1
0346	230	`constant[38]`	1	See [constant], page 50.	+1
0347	231	`constant[39]`	1	See [constant], page 50.	+1
0350	232	`constant[40]`	1	See [constant], page 50.	+1

0351	233	constant[41]	1	See [constant], page 50.	+1
0352	234	constant[42]	1	See [constant], page 50.	+1
0353	235	constant[43]	1	See [constant], page 50.	+1
0354	236	constant[44]	1	See [constant], page 50.	+1
0355	237	constant[45]	1	See [constant], page 50.	+1
0356	238	constant[46]	1	See [constant], page 50.	+1
0357	239	constant[47]	1	See [constant], page 50.	+1
0360	240	constant[48]	1	See [constant], page 50.	+1
0361	241	constant[49]	1	See [constant], page 50.	+1
0362	242	constant[50]	1	See [constant], page 50.	+1
0363	243	constant[51]	1	See [constant], page 50.	+1
0364	244	constant[52]	1	See [constant], page 50.	+1
0365	245	constant[53]	1	See [constant], page 50.	+1
0366	246	constant[54]	1	See [constant], page 50.	+1
0367	247	constant[55]	1	See [constant], page 50.	+1
0370	248	constant[56]	1	See [constant], page 50.	+1
0371	249	constant[57]	1	See [constant], page 50.	+1
0372	250	constant[58]	1	See [constant], page 50.	+1
0373	251	constant[59]	1	See [constant], page 50.	+1
0374	252	constant[60]	1	See [constant], page 50.	+1
0375	253	constant[61]	1	See [constant], page 50.	+1
0376	254	constant[62]	1	See [constant], page 50.	+1
0377	255	constant[63]	1	See [constant], page 50.	+1

Appendix A

A.1 References

- Execution of byte code produced by bytecomp.el (`http://git.savannah.gnu.org/cgit/emacs.git/tree/src/bytecode.c`)
- bytecomp.el — compilation of Lisp code into byte code (`http://git.savannah.gnu.org/cgit/emacs.git/tree/lisp/emacs-lisp/bytecomp.el`)
- data.c — Primitive operations on Lisp data types (`http://git.savannah.gnu.org/cgit/emacs.git/tree/src/data.c`)
- Emacs Byte-code Internals (`http://nullprogram.com/blog/2014/01/04/`)
- Emacs Wiki ByteCodeEngineering (`https://www.emacswiki.org/emacs/ByteCodeEngineering`)
- Assembler for Emacs' bytecode interpreter (`https://groups.google.com/forum/#!topic/gnu.emacs.sources/oMfZT_4Oxrc` `easm.el`)
- Emacs Lisp Decompiler (`https://github.com/rocky/elisp-decompile`)
- GNU Emacs Lisp Reference Manual (`https://ftp.gnu.org/pub/gnu/emacs`)
- GNU Emacs source for version 18.59 (`https://ftp.gnu.org/pub/old-gnu/emacs/emacs-18.59.tar.gz`)
- GNU Emacs source for version 19.34 (`https://ftp.gnu.org/pub/old-gnu/emacs/emacs-19.34b.tar.gz`)
- GNU Emacs source for version 20.1 (`https://ftp.gnu.org/pub/old-gnu/emacs/emacs-20.1.tar.gz`)
- GNU Emacs source code version 21.4 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-21.4a.tar.gz`)
- GNU Emacs source code for version 22.1 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-22.1.tar.gz`)
- GNU Emacs source code for version 22.1 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-23.1.tar.gz`)
- GNU Emacs source code for version 23.1 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-23.2.tar.gz`)
- GNU Emacs source code for version 24.1 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-24.1.tar.gz`)
- GNU Emacs source code for version 24.1 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-24.1.tar.gz`)
- GNU Emacs source code for version 25.3 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-25.3.tar.gz`)
- GNU Emacs source code for version 25.3 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-25.3.tar.gz`)
- GNU Emacs source code for version 26.3 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-26.3.tar.gz`)
- GNU Emacs source code for version 27.0.90 (`https://ftp.gnu.org/pub/gnu/emacs/emacs-27.0.90.tar.gz`)
- Lars Brinkhoff's Emacs History (`https://github.com/larsbrinkhoff/emacs-history`)

- Github Elisp Decompiler Project (`https://github.com/rocky/elisp-decompile`)
- NYC Emacs Lisp Meetup talk: Bytecode and miscellaneous thoughts on the Emacs Runtime (`https://rocky.github.io/NYC-Emacs-April-2018`)
- gccemacs (`http://akrl.sdf.org/gccemacs.html`)
- Bringing GNU Emacs to Native Code (`https://zenodo.org/record/3736363#.X_LeOFNKjMU`)

A.2 Instruction Index

N

P

Q

R

S

T

U

V

W

A.3 Bytecode Function Index

A.4 Concept Index