# Rapid Case Dispatch in Scheme

William D Clinger
Northeastern University
`will@ccs.neu.edu`

## Abstract

The `case` expressions of Scheme can and should be implemented efficiently. A three-level dispatch performs well, even when dispatching on symbols, and scales to large `case` expressions.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—compilers, optimization

***General Terms*** Algorithms, Languages, Performance

***Keywords*** case expressions, symbols, Scheme

## 1. Introduction

Programming languages should be implemented not by piling hack upon hack, but by removing the inefficiencies and restrictions that make additional hacks appear necessary.

The `case` expressions of Scheme are a convenient syntax for rapid selection between actions determined by a computed value that is expected to lie within a known finite set of symbols, numbers, characters, and booleans [5].

Although Scheme's `case` expressions are fast by design, too many systems still implement them inefficiently. These inefficient implementations have led some programmers to write contorted and inefficient code for case dispatch when `case` expressions would have been more natural *and* more efficient.

In particular, some Scheme programmers believe the evaluation of a `case` expression requires time proportional to the number of literals mentioned in its clauses. Others believe the efficiency of multi-way case dispatch on characters depends upon the size of the character set. Some understand that multi-way case dispatch on numbers and characters is efficient, but believe that multi-way case dispatch on symbols is inherently inefficient. These incorrect beliefs have led some programmers to eschew the use of symbols as enumerated values, to fear Unicode, or to avoid `case` expressions altogether.

The contributions of this paper are:

1. To show that Scheme's `case` expressions are efficient when implemented properly.
2. To describe an efficient triple-dispatch technique for implementing general case dispatch.

The techniques used to implement fast case dispatch in languages like Pascal, C, and Java are well-known, so the primary focus of this paper is on more Scheme-specific issues: fast dispatch for symbols and for dispatch on values of mixed types.

## 2. Implementation

This section describes the implementation of `case` expressions in Larceny v0.92, which uses the Twobit compiler [1, 4].

The basic idea can be seen in figure 1:

1. Dispatch on the type.
2. Use some type-specific dispatch to map the value to the index of its associated clause.
3. Use binary search on the clause index to select the expressions to be evaluated for that clause.

What remains to be explained are the details. Following Orbit's example [7], Twobit's first pass macro-expands `case` expressions into more primitive expressions, as described in R5RS 7.3 [5].

When control optimization is enabled, Twobit's second pass recognizes `if` expressions whose test is a call

```
(let ((n (cond ((char? var0)
                <dispatch-on-char>)
               ((symbol? var0)
                <dispatch-on-symbol>)
               ; miscellaneous constants
               ((eq? var0 '#f) ...)
               ...
               ((fixnum? var0)
                <dispatch-on-fixnum>)
               (else 0))))
  <dispatch-on-n>)
```

**Figure 1.** General form of triple dispatch

to eq?, eqv?, memq, or memv whose first argument is a variable and whose second argument is a literal constant. When such an expression is found, Twobit looks for nested if expressions of the same form whose test compares the same variable against one or more literal constants. Twobit analyzes these nested if expressions to reconstruct the equivalent of a set of case clauses, each consisting of a set of constants paired with the expressions to be evaluated if the variable's value is one of those constants. This analysis removes duplicate constants, so the sets of constants are disjoint.

Twobit then decides between one of two strategies:

- brute-force sequential search
- the triple dispatch of figure 1

Sequential search is used if the total number of constants is less than some threshold, typically 12, for which benchmarks have shown the triple-dispatch technique to be faster than a simple sequential search.

If Twobit decides to use triple dispatch, then it numbers the clauses sequentially (reserving 0 for the else clause, if any) and generates code of the form shown in figure 1. If there are no miscellaneous constants, then the corresponding cond clauses will not appear. If there are no character constants, then the character clause is unnecessary, and similarly for the symbol and fixnum clauses.

(A *fixnum* is a small exact integer. Twobit's idea of the fixnum range may be smaller than the fixnum range that is actually defined by some of Larceny's back ends, so Twobit may misclassify a large fixnum as a miscellaneous constant. That misclassification is safe because the miscellaneous constants come before the fixnum? test in figure 1.)

```
(lambda (x)
  (case x
   ((#\a #\e #\i #\o #\u #\A #\E #\I #\O #\U
     a e i o u)
    (f-vowel x))
   ((#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m
     #\n #\p #\q #\r #\s #\t #\v #\w #\x #\y #\z
     #\B #\C #\D #\F #\G #\H #\J #\K #\L #\M
     #\N #\P #\Q #\R #\S #\T #\V #\W #\X #\Y #\Z
     b c d f g h j k l m n p q r s t v w x y z)
    (f-consonant x))
   (else
    (f-other x))))
```

**Figure 2.** Example: source code

The three type-specific dispatches are independent, and can be implemented in completely different ways.

To map a fixnum to a clause index, Twobit chooses one of these techniques:

- sequential search
- binary search
- table lookup

Sequential search is best when there are only a few fixnum constants, with gaps between them. The cost of a binary search depends on the number of intervals, not on the number of constants; for example, the cost of testing for membership in $[1, 127]$ is the same as the cost of testing for membership in $[81, 82]$. The choice between binary search and table lookup is made on the basis of code size: a binary search costs about 5 machine instructions per interval, while a table lookup costs about $hi - lo$ words, where $lo$ and $hi$ are the least and greatest fixnums to be recognized. Binary search followed by table lookup would be an excellent general strategy, but Twobit does not yet combine binary search with table lookup.

To map a character to a clause index, Twobit converts the character to a fixnum and performs a fixnum dispatch.

To map a symbol to a clause index, Twobit can use either sequential search or a hash lookup. In Larceny, every symbol's hash code is computed when the symbol is created and is stored explicitly as part of the symbol structure, so hashing on a symbol is very fast. Twobit uses a closed hash table, represented by a vector of symbols (or #f) alternating with the corresponding clause index (or 0 for the else clause). As this vector is

```
(lambda (x)
  (let* ((temp x)
         (n (if (char? temp)
                (let ((cp (char->integer:chr temp)))
                  (if (<:fix:fix cp 65)
                      0
                      (if (<:fix:fix cp 124)
                          (vector-ref:trusted
                            '#(1 2 2 2 1 2 2 2 1 2 2 2 2 2 1 2
                               2 2 2 2 1 2 2 2 2 2 0 0 0 0 0 0
                               1 2 2 2 1 2 2 2 1 2 2 2 2 2 1 2
                               2 2 2 2 1 2 2 2 2 2 0)
                            (-:idx:idx cp 65))
                          0)))
                (if (symbol? temp)
                    (let ((symtable
                           '#(#f 0 #f 0 #f 0 #f 0 w 2 x 2 y 2 z 2
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              c 2 d 2 e 1 f 2 #f 0 #f 0 a 1 b 2
                              k 2 l 2 m 2 n 2 g 2 h 2 i 1 j 2
                              s 2 t 2 u 1 v 2 o 1 p 2 q 2 r 2))
                          (i (fixnum-arithmetic-shift-left:fix:fix
                              (fixnum-and:fix:fix 63 (symbol-hash:trusted temp))
                              1)))
                      (if (eq? temp (vector-ref:trusted symtable i))
                          (vector-ref:trusted symtable (+:idx:idx i 1))
                          0))
                    0))))
    (if (<:fix:fix n 1)
        (f-other x)
        (if (<:fix:fix n 2)
            (f-vowel x)
            (f-consonant x)))))
```

**Figure 3.** Example: partially optimized intermediate code

generated, Twobit computes the maximum distance between the vector index computed from a symbol's hash code and the vector index at which the symbol is actually found. This bound on the closed hash search allows Twobit to generate straight-line code, without loops.

All of the fixnum, character, symbol, and vector operations that implement these strategies will operate on values that are known to be of the correct type and in range, so most of those operations will compile into a single machine instruction.

Figures 2 and 3 show the complete code for an artificial example. (For this example, all of the symbols are found at the vector index computed from their hash code, so no further search is necessary. The intermediate code has been edited to improve its readability.)

Twobit's optimization of `case` expressions could have been performed by implementing `case` as a low-level macro. This would be slightly less effective than what Twobit actually does, because Twobit will optimize nested `if` expressions that are equivalent to a `case` expression, even if no `case` expression was present in the original source code. The macro approach may nonetheless be the easiest way to add efficient case dispatch to simple compilers or interpreters.

## 3. Survey

An incomplete survey of about 140,000 lines of Scheme code distributed with Larceny v0.92 located about 330 `case` expressions [4]. Of the 180 that were examined in detail, the largest and most performance-critical were found within various assemblers and peephole optimizers, written by at least three different programmers. The largest `case` expression is part of Common Larceny's new in-memory code generator (for which the fashionable term would be "JIT compiler"), and translates symbolic names of IL instructions to the canonical strings expected by Microsoft's `System.Reflection.Emit` namespace. This `case` expression contains 217 clauses with 363 symbols. The next largest contains 17 clauses with 102 symbols. Four `case` expressions contain 32 to 67 fixnum literals, and another dozen or so contain 16 to 31 symbols.

Only seven of the 180 `case` expressions contain literals of mixed type. One is the 217-clause monster, which contains 214 lists as literals in addition to its 363 symbols, but those list literals are useless and derive from an otherwise benign bug in a local macro; the 363 symbols should have been the only literals. (Had the list literals slowed this case dispatch, loading a source file into Common Larceny would be even slower than it is.) The mixed types in three other `case` expressions were caused by that same bug. The three purposeful examples of mixed-type dispatch contain 7, 10, or 11 literals, mixing symbols with booleans or fixnums, and their performance is unimportant. Mixed-case dispatch appears to be more common in the less performance-critical code whose case expressions were not examined in detail.

## 4. Benchmarks

Source code for the benchmarks described in this section is available online [2].

A six-part `case` micro-benchmark was written to test the performance of case dispatch on fixnums and on symbols, for `case` expressions with 10, 100, or 1000 clauses that match one fixnum or symbol each. Figure 4 shows the 10-clause `case` expression for symbols, from which the other five parts of the micro-benchmark can be inferred. Each of the six parts performs one million case dispatches, so any differences in timing between the six parts must be attributed to the number of clauses in each case dispatch, and to the difference between dispatching on a fixnum and dispatching on a symbol.

```
(define (s10 x)
  (define (f x sum)
    (case x
      ((one) (f 'two (- sum 1)))
      ((two) (f 'three (+ sum 2)))
      ((three) (f 'four (- sum 3)))
      ((four) (f 'five (+ sum 4)))
      ((five) (f 'six (- sum 5)))
      ((six) (f 'seven (+ sum 6)))
      ((seven) (f 'eight (- sum 7)))
      ((eight) (f 'nine (+ sum 8)))
      ((nine) (f 'onezero (- sum 9)))
      ((onezero) (f 'oneone (+ sum 10)))
      (else (+ sum 9))))
  (f x 0))
```

**Figure 4.** One part of the `case` micro-benchmarks

The `monster` micro-benchmark is a mixed-type case dispatch that uses the 217-clause, 577-literal `case` expression of Common Larceny v0.92 to translate one million symbols to strings. (That many translations might actually occur when a moderately large program is loaded into Common Larceny.)

A set of four benchmarks was written to measure performance of Scheme systems on components of a realistic parsing task [2]. The `parsing` benchmark reads a file of Scheme code, converts it to a string, and then parses that string repeatedly, creating the same data structures the `read` procedure would create. The timed portion of the `parsing` benchmark begins after the input file has been read into a string, and does not include any i/o. The `lexing` and `casing` benchmarks are simplifications of the `parsing` benchmark, and measure the time spent in lexical analysis and in case dispatch, respectively. (The `lexing` benchmark computes the same sequence of lexical tokens that are computed by the `parsing` benchmark, but does not perform any other parsing. The main differences between the `lexing` benchmark and the `casing` benchmark are that the `casing` benchmark does not copy the characters of each token to a token buffer and does not keep track of source code locations. The `casing` benchmark still includes all other string operations that are performed on the input string during lexical analysis, so it is not a pure case dispatch benchmark.) The `reading` benchmark performs the same task as the `parsing` benchmark, but uses the built-in `read` procedure to read from a string port (SRFI 6 [3]). The main purpose of the `reading` benchmark is to show that the `parsing` benchmark's computer-generated lexical analyzer and parser are not outrageously inefficient.

Both the state machine of the lexical analyzer and the recursive descent parser were generated by the author's LexGen and ParseGen, which can generate lexical analyzers and parsers written in Scheme, Java, or C [2]. This made it fairly easy to translate the `parsing` benchmark into Java. As it was not obvious whether the strings of the Scheme benchmark should be translated into arrays of `char` or into instances of the `StringBuilder` class, two versions of the Java code were written; a third version, just for grins, uses the thread-safe `StringBuffer` class.

The timings reported in the next section for the `casing`, `lexing`, `parsing`, and `reading` benchmarks were obtained by casing, lexing, parsing, or reading the `nboyer` benchmark one thousand times [2].

## 5. Benchmark Results

Tables 1 and 2 show execution times for the benchmarks, in seconds, as measured for several implementations on an otherwise unloaded SunBlade 1500 (64-bit, 1.5-GHz UltraSPARC IIIi). Most of the timings represent elapsed time, but a few of the slower timings represent CPU time. For the compiled systems and the fastest interpreters, the timings were obtained by averaging at least three runs. For the slower interpreters, the reported timing is for a single run.

For the two largest `case` micro-benchmarks, three of the Scheme compilers generated C code that was too large or complex for `gcc` to handle.

From table 1, it appears that compilers C and D use sequential search for all `case` expressions. Compilers B, E, and F generate efficient code when dispatching on fixnums, but appear to use sequential search for symbols.

Compiler A (Larceny v0.92) has the best overall performance on the micro-benchmarks, and Compiler B (Larceny v0.91) is next best. The difference between them is that Larceny v0.92 implements `case` expressions as described in this paper.

Table 2 shows that, for the `parsing` benchmark, most of these implementations of Scheme spend roughly half their time in case dispatch. The two that spend the least time in case dispatch, compilers F and B, perform well on the fixnum `case` micro-benchmarks and appear to be doing well on the `parsing` benchmark's character dispatch also. Compiler C's performance may mean sequential search is fast enough for this benchmark, or it may mean that compiler C recognizes `case` clauses that match sets of consecutive characters (such as `#\a` through `#\z`, `#\A` through `#\Z`, and `#\0` through `#\9`) and tests for them using a range check instead of testing individually for each character.

The difference between Larceny v0.92 and v0.91 (compilers A and B) does not matter for the `parsing` benchmark, because v0.91 was already generating efficient code for case dispatch on characters.

## 6. Related Work

Compilers for mainstream languages typically implement `case/switch` statements using sequential search, binary search, or jump tables [6].

A binary search usually concludes with a jump to code for the selected case. In the subset of Scheme that serves as Twobit's main intermediate language, jumps are best implemented as tail calls. Those calls would interfere with many of Twobit's intraprocedural optimizations, so Twobit does not use a single-level binary search.

Jump tables are hard to express in portable Scheme without using `case` expressions, which are not part of Twobit's intermediate language. Adding even a restricted form of `case` expressions to Twobit's intermediate language is unattractive, because it would complicate most of Twobit's other optimizations.

Jump tables can be implemented portably using a vector of closures, but it would cost too much to create those closures and to store them into a vector every time the scope containing a `case` expression is entered. A vector of lambda-lifted closures could be created once and for all, but would entail the costs of passing extra arguments and of making an indirect jump. With either form of jump table, calling a closure that cannot be identified at compile time would interfere with many of Twobit's intraprocedural optimizations.

The Orbit compiler demonstrated that it is practical to macro-expand `case` expressions into `if` expressions, and for control optimization to recognize and to generate efficient code from those `if` expressions [7].

## Acknowledgments

The author is grateful to the anonymous reviewers, not only for remarks that improved this paper, but also for suggestions that improved the implementation of `case` expressions in Larceny v0.92 [4].

|  | `case` |  |  |  |  |  | `monster` |
|---|---|---|---|---|---|---|---|
|  | 10 literals |  | 100 literals |  | 1000 literals |  | 577 literals |
|  | fixnum | symbol | fixnum | symbol | fixnum | symbol | mixed |
| Compiler A | .04 | .05 | .04 | .08 | .11 | .13 | .16 |
| Compiler B | .04 | .05 | .07 | .21 | .14 | 3.94 | 3.04 |
| Compiler C | .04 | .04 | .18 | .17 | 3.80 | 4.61 | 8.33 |
| Compiler D | .09 | .09 | .24 | .22 | — | — | 15.94 |
| Compiler E | .06 | .12 | .02 | .50 | — | — | 15.11 |
| Compiler F | .05 | .70 | .04 | 6.03 | — | — | 26.95 |
| Interpreter G | 1.52 | 1.30 | 10.00 | 7.80 | 96.81 | 78.03 | 26.21 |
| Interpreter H | 1.79 | 1.76 | 10.65 | 10.91 | 115.52 | 119.67 |  |
| Interpreter I | 3.48 | 3.48 | 15.62 | 15.38 | 188.12 | 186.38 | 33.50 |
| Interpreter J | 6.00 | 6.33 | 20.99 | 21.63 | 193.17 | 196.21 | 60.26 |
| Interpreter K | 5.00 | 5.00 | 21.00 | 24.00 | 211.00 | 256.00 | 59.00 |
| Interpreter L | 5.36 | 5.38 | 29.09 | 28.30 | 280.22 | 289.58 | 147.43 |
| Interpreter M | 6.12 | 4.48 | 49.48 | 30.42 | 447.08 | 301.53 | 338.78 |
| Interpreter N | 13.82 | 13.88 | 77.68 | 78.18 | 757.16 | 776.75 | 459.51 |

**Table 1.** Timings in seconds for the `case` and `monster` micro-benchmarks

|  | `casing` | `lexing` | `parsing` | `reading` |
|---|---|---|---|---|
| HotSpot (array of `char`) |  |  | 11.05 |  |
| HotSpot (`StringBuilder`) |  |  | 12.21 |  |
| Compiler C | 7.36 | 10.67 | 13.27 | 2.23 |
| Compiler F | 2.83 | 5.39 | 14.48 | 2.60 |
| Compiler B | 6.93 | 12.84 | 21.17 | 14.67 |
| HotSpot (`StringBuffer`) |  |  | 24.95 |  |
| Compiler D | 13.53 | 22.65 | 27.20 | 17.78 |
| Compiler E | 45.67 | 63.88 | 84.46 | 72.53 |
| Interpreter G | 79.93 | 108.44 | 128.95 | 13.88 |
| Interpreter H | 82.80 | 116.12 | 214.82 | 18.98 |
| Interpreter I | 180.64 | 237.37 | 297.96 |  |
| Interpreter L | 257.13 | 383.77 | 432.13 |  |
| Interpreter J | 436.19 | 566.83 | 645.31 |  |
| Interpreter M | 479.36 | 589.17 | 701.70 |  |
| Interpreter K | 468.00 | 628.00 | 745.00 |  |
| Interpreter N | 1341.93 | 1572.89 | 1793.64 |  |

**Table 2.** Timings in seconds for `parsing` and related benchmarks

## References

[1] Clinger, William D, and Hansen, Lars Thomas. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, 1994, pages 128–139.

[2] Clinger, William D. Source code for the benchmarks described in this paper are online at `www.ccs.neu.edu/home/will/Research/SW2006/`

[3] Clinger, William D. SRFI 6: Basic String Ports. `http://srfi.schemers.org/srfi-6/`.

[4] Clinger, William D., et cetera. The Larceny Project. `http://www.ccs.neu.edu/home/will/Larceny/`

[5] Kelsey, Richard, Clinger, William, and Rees, Jonathan (editors). Revised$^5$ *report on the algorithmic language Scheme.* ACM SIGPLAN Notices 33(9), September 1998, pages 26–76.

[6] Fischer, Charles N., and LeBlanc Jr, Richard J. *Crafting a Compiler with C.* Benjamin/Cummings, 1991.

[7] Kranz, David, Adams, Norman, Kelsey, Richard, Rees, Jonathan, Hudak, Paul, and Philbin, James. ORBIT: an optimizing compiler for Scheme. In *Proc. 1986 SIGPLAN Symposium on Compiler Construction*, 1986, pages 219–233.

## A. Notes on Benchmarking

The benchmarked systems:

**HotSpot** is the Java HotSpot(TM) Client VM of Sun Microsystems (build 1.5.0_01-b08, mixed mode, sharing).

**A** is Larceny v0.92.

**B** is Larceny v0.91.

**C** is Chez Scheme v6.1.

**D** is Gambit 4.0b17.

**E** is Chicken Version 1, Build 89.

**F** is Bigloo 2.7a.

**G** is MzScheme v352.

**H** is MzScheme v301.

**I** is the Larceny v0.92 interpreter.

**J** is the Gambit 4.0b17 interpreter.

**K** is the Bigloo 2.7a interpreter.

**L** is the MIT Scheme 7.3.1 interpreter.

**M** is the Scheme 48 1.3 interpreter.

**N** is the Chicken 1,89 interpreter.

Except for MzScheme, the interpreters were benchmarked with no declarations and with the default settings. The compilers and MzScheme were benchmarked as in Chez Scheme's `(optimize-level 2)`: safe code, generic arithmetic, inlining the usual procedures. Specifically:

**A** was compiled with
`(compiler-switches 'fast-safe)` and
`(benchmark-mode #f)`.

**B** was compiled the same as A.

**C** was compiled with `(optimize-level 2)`.

**D** was compiled with
`-prelude "(declare (extended-bindings))"`
`-cc-options "-O2" -dynamic`.

**E** was compiled with `-no-trace`
`-optimize-level 2 -block -lambda-lift`.

**F** was compiled as a module with `-O6 -copt -O2`.

**G** was interpreted as a module.

**H** was interpreted the same as G.