

Implementing an efficient string= case in Common Lisp

Paul Khuong

September 1, 2007

1 Introduction

In <http://neverfriday.com/blog/?p=10>, OMouse asks how best to implement a `string=` case (in Scheme). I noted that naively iterating through the cases with `string=` at runtime is suboptimal. Seeing the problem as a simplistic pattern matching one makes an efficient solution obvious. Note that, unlike Haskell, both Scheme and CL have random-access on strings in $O(1)$, something which I exploit to generate better code.

This is also a `pbook.el` file (the pdf can be found at <http://www.discontinuity.info/~pkhuong/string-case.pdf>). I'm new at this not-quite-illiterate programming thing, so please bear with me (: I'm also looking for comments on the formatting. I'm particularly iffy with the way keywords look like. It just looks really fuzzy when you're not really zoomed in (or reading it on paper).

I usually don't use packages for throw-away code, but this looks like it could be useful to someone.

```
1 (cl:deffpackage #:string-case
2   (:use      #:cl)
3   (:export  #:string-case))

5 (cl:in-package #:string-case)
```

2 Some utility code

```
1 (defun SPLIT (list &key (test 'eql) (key 'identity))
2   "Splits input list into sublists of elements
3   whose elements are all such that (key element)
4   are all test.
5   It's assumed that test and key form an equality class.
6   (This is similar to groupBy)"
7   (when list
8     (let* ((lists ())
9            (cur-list (list (first list)))
10           (cur-key  (funcall key (first list))))
11       (dolist (elt (rest list) (nreverse (cons (nreverse cur-list)
12                                                lists)))
13         (let ((new-key (funcall key elt)))
```

```

14         (if (funcall test cur-key new-key)
15             (push elt cur-list)
16             (progn
17               (push (nreverse cur-list) lists)
18               (setf cur-list (list elt)
19                       cur-key new-key)))))))))

21 (defun IOTA (n)
22   (loop for i below n collect i))

24 (defun HASH-TABLE->LIST (table &key (keep-keys t) (keep-values t))
25   "Saves the keys and/or values in table to a list.
26   As with hash table iterating functions, there is no
27   implicit ordering."
28   (let ((list ()))
29     (maphash (cond ((and keep-keys
30                     keep-values)
31                 (lambda (k v)
32                   (push (cons k v) list)))
33             (keep-keys
34                 (lambda (k v)
35                   (declare (ignore v))
36                   (push k list)))
37             (keep-values
38                 (lambda (k v)
39                   (declare (ignore k))
40                   (push v list))))
41     table)
42   list))

44 (defun ALL-EQUAL (list &key (key 'identity) (test 'eql))
45   (if (or (null list)
46           (null (rest list)))
47       t
48       (let ((first-key (funcall key (first list))))
49         (every (lambda (element)
50                 (funcall test first-key
51                         (funcall key element)))
52               (rest list))))))

```

3 The string matching compiler per se

I use special variables here because I find that preferable to introducing noise everywhere to thread these values through all the calls, especially when `*no-match-form*` is only used at the very end.

```
1 (defparameter *INPUT-STRING* nil
```

```

2  "Symbol of the variable holding the input string")
4  (defparameter *NO-MATCH-FORM* nil
5  "Form to insert when no match is found.")

```

The basic idea of the pattern matching process here is to first discriminate with the input string's length; once that is done, it is very easy to safely use random access until only one candidate string (pattern) remains. However, even if we determine that only one case might be a candidate, it might still be possible for another string (not in the set of cases) to match the criteria. So we also have to make sure that **all** the indices match. A simple way to do this would be to emit the remaining checks at the every end, when only one candidate is left. However, that can result in a lot of duplicate code, and some useless work on mismatches. Instead, the code generator always tries to find (new) indices for which all the candidates left in the branch share the same character, and then emits a guard, checking the character at that index as soon as possible.

In my experience, there are two main problems when writing pattern matchers: how to decide what to test for at each fork, and how to ensure the code won't explode exponentially. Luckily, for our rather restricted pattern language (equality on strings), patterns can't overlap, and it's possible to guarantee that no candidate will ever be possible in both branches of a fork.

Due to the the latter guarantee, we have a simple fitness measure for tests: simply maximising the number of candidates in the smallest branch will make our search tree as balanced as possible. Of course, we don't know whether the subtrees will be balanced too, but I don't think it'll be much of an issue.

Note that, if we had access, whether via annotations or profiling, to the probability of each case, the situation would be very different. In fact, on a pipelined machine where branch mispredictions are expensive, an unbalanced tree will yield better expected runtimes. There was a very interesting and rather sophisticated Google lecture on that topic on Google video (the speaker used markov chains to model dynamic predictors, for example), but I can't seem to find the URL.

TODO: Find bounds on the size of the code!

```

1  (defun FIND-BEST-SPLIT (strings to-check)
2  "Iterate over all the indices left to check to find
3  which index (and which character) to test for equality
4  with, keeping the ones which result in the most balanced
5  split."
6  (flet ((evaluate-split (i char)
7  "Just count all the matches and mismatches"
8  (let ((= 0)
9  (/= 0))
10  (dolist (string strings (min = /=))
11  (if (eql (aref string i) char)
12  (incf =)
13  (incf /=))))))
14  (uniquify-chars (chars)
15  "Only keep one copy of each char in the list"
16  (mapcar 'first (split (sort chars '<:key 'char-code))))))
17  (let ((best-split 0) ; maximise size of smallest branch
18  (best-posn nil)

```

```

19     (best-char nil))
20 (dolist (i to-check (values best-posn best-char))
21   (dolist (char (uniquify-chars (mapcar (lambda (string)
22     (aref string i))
23     strings)))
24     (let ((Z (evaluate-split i char)))
25       (when (> Z best-split)
26         (setf best-split Z
27               best-posn i
28               best-char char)))))))))

```

At each step, we may be able to find positions for which there can only be one character. If we emit the check for these positions as soon as possible, we avoid duplicating potentially a lot of code.

```

1 (defun EMIT-COMMON-CHECKS (strings to-check)
2   (flet ((emit-checking-form (common-chars)
3     (when common-chars
4       (let ((common-chars (sort common-chars '< :key 'car)))
5         (if (null (rest common-chars))
6           (destructuring-bind (posn . char)
7             (first common-chars)
8             '(eql ,char (aref ,*input-string* ,posn)))
9           #+ (or) '(and ,@(mapcar (lambda (pair)
10             (destructuring-bind (posn . char)
11               pair
12               '(eql ,char (aref ,*input-string* ,posn))))
13             common-chars))
14           #+ (and) '(zerop (logior
15             ,@(mapcar
16               (lambda (pair)
17                 (destructuring-bind (posn . char)
18                   pair
19                   '(logxor (char-code ,char)
20                     (char-code
21                       (aref ,*input-string* ,posn))))))
22             common-chars)))))))))
23   (let ((common-chars ())
24         (left-to-check ()))
25     (dolist (posn to-check (values (emit-checking-form common-chars)
26                                   (nreverse left-to-check)))
27       (if (all-equal strings :key (lambda (string)
28         (aref string posn)))
29         (push (cons posn (aref (first strings) posn))
30               common-chars)
31         (push posn left-to-check))))))

```

The driving function: First, emit any test that is common to all the candidates. If there's only one candidate, then we just have to execute₄ the body; if not, we look for the best test and

emit the corresponding code: execute the test, and recurse on the candidates that match the test and on those that don't.

```

1 (defun MAKE-SEARCH-TREE (strings bodies to-check)
2   (multiple-value-bind (guard to-check)
3     (emit-common-checks strings to-check)
4     (if (null (rest strings))
5       (progn
6         (assert (null to-check)) ; there shouldn't be anything left to check
7         (if guard
8           '(if ,guard
9             (progn ,@(first bodies))
10            ,*no-match-form*)
11          '(progn ,@(first bodies))))
12       (multiple-value-bind (posn char)
13         (find-best-split strings to-check)
14         (assert posn) ; this can only happen if all strings are equal
15         (let ((=strings ())
16              (=bodies ())
17              (/=strings ())
18              (/=bodies ()))
19           (loop
20             for string in strings
21             for body in bodies
22             do (if (eql char (aref string posn))
23                 (progn
24                   (push string =strings)
25                   (push body =bodies))
26                 (progn
27                   (push string /=strings)
28                   (push body /=bodies))))
29           (let ((tree '(if (eql ,char (aref ,*input-string* ,posn))
30                          ,(make-search-tree =strings =bodies
31                                               (remove posn to-check))
32                          ,(make-search-tree /=strings /=bodies
33                                               to-check))))
34             (if guard
35               '(if ,guard
36                 ,tree
37                 ,*no-match-form*)
38               tree))))))

```

Finally, we can glue it all together. To recapitulate, first, dispatch on string length, then execute a search tree for the few candidates left, and finally make sure the input string actually matches the one candidate left at the leaf.

```

1 (defun EMIT-STRING-CASE (cases input-var no-match)
2   (flet ((case-string-length (x)
3         (length (first x))))

```

```

4   (let ((*input-string* input-var)
5       (*no-match-form* no-match)
6       (cases-lists (split (sort cases '<
7                           :key #'case-string-length)
8                           :key #'case-string-length)))
9   '(case (length ,input-var)
10      ,@(loop for cases in cases-lists
11             for length = (case-string-length (first cases))
12             collect '(,length
13                     (locally (declare (type (array * (,length)) ,*input-string*))
14                             ,(make-search-tree (mapcar 'first cases)
15                                                  (mapcar 'rest cases)
16                                                  (iota length))))))
17      (t ,no-match))))))

```

Just wrapping the previous function in a macro, and adding some error checking (the rest of the code just assumes there won't be duplicate patterns). Note how we use a local function instead of passing the default form directly. This can save a lot on code size, especially when the default form is large.

```

1 (defmacro STRING-CASE ((string &key (default '(error "No match"))
2                          &body cases)
3   "string-case (string &key default)
4   case*
5   case ::= string form*
6   — t form*
7   Where t is the default case."
8 (let ((cases-table (make-hash-table :test 'equal)))
9   "Error checking cruft"
10  (dolist (case cases)
11    (assert (typep case '(cons (or string (eql t)))))
12    (let ((other-case (gethash (first case) cases-table)))
13      (if other-case
14          (warn "Duplicate string-case cases: ~A -> ~A or ~A~%"
15              (first case)
16              (rest other-case)
17              (rest case))
18          (setf (gethash (first case) cases-table) case))))
19  (let ((input-var (gensym "INPUT"))
20        (default-fn (gensym "ON-ERROR"))
21        (default-body (rest (gethash t cases-table (list t default)))))
22    '(let ((,input-var ,string)
23          (flet ((,default-fn ()
24                  ,@default-body))
25              ,(emit-string-case (progn
26                                  (remhash t cases-table)
27                                  (mapcar 'rest (hash-table->list cases-table)))
28                                  input-var
29                                  '(,default-fn))))))

```

Demo output (downcased):

```
1 #+bad #+reader #+hack

3 (macroexpand-1 '(string-case ("foobar")
4     (" " 'empty)
5     ("foo" 'foo)
6     ("fob" 'fob)
7     ("foobar" 'hit)
8     (t 'default)))
9 =>
10 (let ((#:input2037 "foobar"))
11   (flet ((#:on-error2038 ()
12         'default))
13     (case (length #:input2037)
14       ((0) (locally (declare (type (array * (0)) #:input2037))
15                 (progn 'empty)))
16       ((3) (locally (declare (type (array * (3)) #:input2037))
17                 (if (zerop (logior (logxor (char-code #\f)
18                                           (char-code (aref #:input2037 0)))
19                                           (logxor (char-code #\o)
20                                           (char-code (aref #:input2037 1))))))
21                 (if (eql #\b (aref #:input2037 2))
22                     (progn 'fob)
23                     (if (eql #\o (aref #:input2037 2))
24                         (progn 'foo)
25                         (#:on-error2038))))
26                 (#:on-error2038))))))
27       ((6) (locally (declare (type (array * (6)) #:input2037))
28                 (if (zerop (logior (logxor (char-code #\f)
29                                           (char-code (aref #:input2037 0)))
30                                           (logxor (char-code #\o)
31                                           (char-code (aref #:input2037 1)))
32                                           (logxor (char-code #\o)
33                                           (char-code (aref #:input2037 2)))
34                                           (logxor (char-code #\b)
35                                           (char-code (aref #:input2037 3)))
36                                           (logxor (char-code #\a)
37                                           (char-code (aref #:input2037 4)))
38                                           (logxor (char-code #\r)
39                                           (char-code (aref #:input2037 5))))))
40                 (progn 'hit)
41                 (#:on-error2038))))))
42       (t (#:on-error2038))))))
```

A disassembly of the output on SBCL/x86-64:

```
(disassemble (lambda (x)
  (declare (optimize speed)
    (type simple-base-string x))
  (string-case (x)
    ("" 'empty)
    ("foo" 'foo)
    ("fob" 'fob)
    ("foobar" 'hit)
    (t 'default))))

; 034A5D70:      488B4AF9      MOV RCX, [RDX-7]          ; no-arg-parsing entry point
;               D74:      4885C9      TEST RCX, RCX
;               D77:      7513      JNE L1
;               D79:      488B1560FFFFFF MOV RDX, [RIP-160]       ; 'EMPTY
;               D80: L0:  488D65F0      LEA RSP, [RBP-16]
;               D84:      F8      CLC
;               D85:      488B6DF8      MOV RBP, [RBP-8]
;               D89:      C20800      RET 8
;               D8C: L1:  4883F918      CMP RCX, 24
;               D90:      0F848B000000      JEQ L4
;               D96:      4883F930      CMP RCX, 48
;               D9A:      7405      JEQ L2
;               D9C:      E9DA000000      JMP L7
;               DA1: L2:  480FB64201      MOVZX RAX, BYTE PTR [RDX+1]
;               DA6:      48C1E003      SHL RAX, 3
;               DAA:      488BC8      MOV RCX, RAX
;               DAD:      4881F130030000      XOR RCX, 816
;               DB4:      480FB64202      MOVZX RAX, BYTE PTR [RDX+2]
;               DB9:      48C1E003      SHL RAX, 3
;               DBD:      483578030000      XOR RAX, 888
;               DC3:      4809C1      OR RCX, RAX
;               DC6:      480FB64203      MOVZX RAX, BYTE PTR [RDX+3]
;               DCB:      48C1E003      SHL RAX, 3
;               DCF:      483578030000      XOR RAX, 888
;               DD5:      4809C1      OR RCX, RAX
;               DD8:      480FB64204      MOVZX RAX, BYTE PTR [RDX+4]
;               DDD:      48C1E003      SHL RAX, 3
;               DE1:      483510030000      XOR RAX, 784
;               DE7:      4809C1      OR RCX, RAX
;               DEA:      480FB64205      MOVZX RAX, BYTE PTR [RDX+5]
;               DEF:      48C1E003      SHL RAX, 3
;               DF3:      483508030000      XOR RAX, 776
;               DF9:      4809C1      OR RCX, RAX
;               DFC:      480FB64206      MOVZX RAX, BYTE PTR [RDX+6]
;               E01:      48C1E003      SHL RAX, 3
;               E05:      483590030000      XOR RAX, 912
;               E0B:      4809C1      OR RCX, RAX
```

```

; E0E: 4885C9 TEST RCX, RCX
; E11: 7402 JEQ L3
; E13: EB66 JMP L7
; E15: L3: 488B15CCFEFFFF MOV RDX, [RIP-308] ; 'HIT
; E1C: E95FFFFFFFFF JMP L0
; E21: L4: 480FB64201 MOVZX RAX, BYTE PTR [RDX+1]
; E26: 48C1E003 SHL RAX, 3
; E2A: 488BC8 MOV RCX, RAX
; E2D: 4881F130030000 XOR RCX, 816
; E34: 480FB64202 MOVZX RAX, BYTE PTR [RDX+2]
; E39: 48C1E003 SHL RAX, 3
; E3D: 483578030000 XOR RAX, 888
; E43: 4809C1 OR CX, RAX
; E46: 4885C9 TEST RCX, RCX
; E49: 7402 JEQ L5
; E4B: EB2E JMP L7
; E4D: L5: 480FB64203 MOVZX RAX, BYTE PTR [RDX+3]
; E52: 4883F862 CMP RAX, 98
; E56: 750C JNE L6
; E58: 488B1591FEFFFF MOV RDX, [RIP-367] ; 'FOB
; E5F: E91CFFFFFFFFF JMP L0
; E64: L6: 480FB64203 MOVZX RAX, BYTE PTR [RDX+3]
; E69: 4883F86F CMP RAX, 111
; E6D: 750C JNE L7
; E6F: 488B1582FEFFFF MOV RDX, [RIP-382] ; 'FOO
; E76: E905FFFFFFFFF JMP L0
; E7B: L7: 488B157EFEFFFF MOV RDX, [RIP-386] ; 'DEFAULT
; ; no-arg-parsing entry point
; E82: 488D65F0 LEA RSP, [RBP-16]
; E86: F8 CLC
; E87: 488B6DF8 MOV RBP, [RBP-8]
; E8B: C20800 RET 8
; E8E: 90 NOP
; E8F: 90 NOP
; E90: 0F0B0A BREAK 10 ; error trap
; E93: 02 BYTE #X02
; E94: 18 BYTE #X18 ; INVALID-ARG-COUNT-ERROR
; E95: 4E BYTE #X4E ; RCX
; E96: 0F0B0A BREAK 10 ; error trap
; E99: 02 BYTE #X02
; E9A: 34 BYTE #X34 ; OBJECT-NOT-SIMPLE-BASE-STRING-ERR
; E9B: 8F BYTE #X8F ; RDX
;

```