Published: 12/07/67

## Identification

An example of the use of EPL adjustable-length and varying-length strings.
D. B. Wagner

## Epigraph

One pill makes you larger,
One pill makes you small,
And the ones that Mother gives you
Don't do anything at all ...

- The Jefferson Airplane

## Purpose

This Section continues in the spirit of BN.10.01. We take a simple program, go through different ways of organizing it, and show how these source-level decisions affect object code efficiency. Then we give a set of notes which should allow a highly motivated system programmer to read the object code for the various versions. (Source and object listings are given in BN.10.02A.)

## The Program

Our sample program maintains a symbol-table. The calls are:

```
call table (name,value);
call table$lookup(name,value,no);
```

Here <u>name</u> is a character-string, <u>value</u> is the fixed-point value associated with this name, and <u>no</u> is a label. <u>Table</u> enters <u>name</u> and <u>value</u> into the table. <u>Table$lookup</u> looks up <u>name</u> and returns <u>value</u> or goes to <u>no</u> if <u>name</u> is not in the table.

## The numbers

The various versions of the program will be found in BN.10.02A. The following table gives the sizes and timings of these versions.

| | size of text and link, words. | time in ms. | | | |
|---|---|---|---|---|---|
| | | first call to table | average after first | first call to lookup | average after first |
| 1. A single threaded list of adjustable structures | 463 | 194.4 | 1.2 | 8.8 | 2.1 |
| 2. Same with non-adjustable structures | 305 | 141.1 | 0.7 | 6.5 | 1.0 |
| 3. Hashcoded with a list for each bucket, adjustable structures | 599 | 195.2 | 1.3 | 5.6 | 0.9 |
| 4. Some with non-adjustable structure | 453 | 141.9 | 0.8 | 5.3 | 0.5 |
| 5. Two parallel arrays, varying string | 387 | 59.3 | 0.5 | 5.1 | 1.2 |
| 6. Same with non-varying strings | 265 | 35.3 | 0.4 | 5.2 | 1.2 |

## The Versions

Our first version just keeps a threaded list of table entries, and the static pointer list points to the head of this list. Each element of the list contains a pointer to the next, except the last which contains a null pointer.

Note the two structure declarations: entry is the self-referencing declaration for the table entry. A self-referencing structure cannot, of course, be allocated, so we have in addition the structure declaration al-entry which is adjustable but not self-referencing.

In the second version we make two simplifying assumptions: first, that no name is longer than 32 characters; second, that time is more important than space. So we remove all our machinery and make the structure non-adjustable. The results might be called dramatic: we almost double the speed of entering into the table, and better than double the speed of a lookup. Ah, the price of generality.

The next two versions add a simple hashcoding scheme to the first two. Now instead of a single list we have 21 lists, and a name goes into one of these lists depending on its hashcode. Note that in both cases we get a factor of two improvement in the speed of table$lookup and only a small degradation in the speed of table.

The hashcode is very simple: the binary value of the first character of the name, taken mod 21. Anything fancier would be too hard to write in EPL. The internal procedure hasher evaluates this hashcode using a mismatched declaration. We never even tried not using the mismatch, because we were sure the non-mismatched way would be so slow it would drown all the gains of hashcoding.

The fifth version makes the simplifying assumptions that there will be no more than 100 entries in the table, and that space is unimportant. We have two parallel arrays, one containing the names of varying strings and the other containing the corresponding values. Note that the time for entering into the table has improved quite a bit, while the time for looking something up is disappointing.

The final version wastes a terrible amount of space and gives a rather disappointing account of itself. The time for entering in the table has improved a little more, but lookup time is still mediocre.

## Notes on the object code

We take up here where we left off in BN.10.01. Below are notes which will help in the task of reading the object listings in BN.10.02A

Page 3, lines 21-18.

This is the code for the declaration,

    dcl list ptr static init (null);

Lines 21-22 are the null pointer. They occur in every program which uses the built-in function null. The null pointer always has the alias xx0000.

Isspc is the name of one of the location counters used for the code sequence which handles the static initial attribute. Lines 24-26 initialize the variable list. We will discuss later how this code sequence gets invoked.

Pages 3-4, lines 32-92.

This is the code compiled for the declaration of the based adjustable structure entry. It is not really necessary to understand this code completely, but it is worthwhile to note the sheer mass of it. This code, and more in subroutines, is executed on every reference to entry. A better discussion of this code than is given here will be found in BN.6.02 and BN.7.01. Lines 33-51 are a subroutine which calculates the expression,

    ep → entry.ln

In general one subroutine like this would appear for each adjustable element of the structure.

Lines 61-69 are the dope vector template. This is the proper dope vector for the structure with all known extents filled in. Note that on line 69 the dope for the string entry.line has the proper id code but no length filled in.

Lines 71-77 are prologue code. They copy the dope template into the stack and make up half a specifier.

Lines 79-92 are an internal procedure which creates dope for the structure whenever it is needed. Lines 296-298 show a typical call to it. It fills in the minimum information necessary in the dope vector, then goes to .dp0 which calls tdope_ to complete the dope vector.

Pages 5-6, lines 144-153.

This is the code for the statement

```
ln = length (name);
```

There is a bit of fat here: in particular the call to .of0 is quite unnecessary.

Page 6, lines 155-190.

This is the utterly unnecessary code to initialize the free_ segment. It is discussed in BN.10.01. This time it appears because of the allocate statement.

Page 7, lines 215-218, 220-224, and 226-228.

These are, respectively, the three statements initializing the first three elements of the structure entry. Note that the code is really quite good, because of J. F. Gimpel's work. This good code is possible for everything which comes before any adjustable item in a structure.

Page 7, lines 230-255.

This is the code for the assignment,

```
ep ->al_entry.name = name;
```

The code is the same as it would be for,

```
call stgop_$cscs_(name, ep->al_entry.name);
```

Lines 233-236 put the first argument pointer into the argument list.

Lines 237-252 put in the second argument pointer. This code breaks down into several parts:

237-241 Set base pair bb<-bp to point to the structure and ab<-ap to point to the structure's dope.

| | |
|---|---|
| 242-244 | Set bb←bp to point to the string within the structure and ab←ap to point to the string dope within the structure dope. |
| 249-251 | Create specifier and dope for the string in stack locations 70-75. |
| 252 | Store a pointer to this specifier into the argument list. |

Page 9, lines 334-340.

This is the statement,

```
go to no;
```

Since no is not local, this is compiled essentially as,

```
call unwinder (no);
```

So that any necessary epilogues can be performed.

Page 10, lines 359-373.

This code sets up for the growing and initialization of the block of storage used for internal static storage. See BN.7.08 for an overview of what is happening.

Lines 362-363 are the beginning of the internal procedure which initializes the internal static storage. This internal procedure has been built up in two location counters, specc and isspc, because of problems related to getting specifiers set up before they are needed.

Look back to lines 276-277 to see how internal static storage is accessed. The instruction,

```
eapbp lp|.is,*
```

causes the storage to be set up if it hasn't already. In any case, by the time the instruction is finished, bb←bp is set to point to the fully initialized block of static storage.

Page 18, lines 102-115.

This is the call to stgop_$cscs_ to perform the assignment,

```
ep->entry.name = name;
```

Compare with page 7, lines 250-255. This time ep- entry.name is not adjustable. Consequently in the preparation of the second argument for the call, we find:

| | |
|---|---|
| 106-108 | Make bb<-bp point to the string |
| 109-111 | Create specifier and dope for the string in stack locations 46-51. |
| 112 | Store a pointer to this specifier into argument list. |

Page 28, lines 66-71.

This is the code for,

```
     list (j) = null;
```

It is, of course, rather unfortunate that this code multiplies by 2 by multiplying by 72 and dividing by 36. Code equivalent to lines 67-69 would be:

```
             eax6      thing,*7

             ...

   thing: arg          0,7
```

In addition, just to be cute, one might replace lines 68-71 with,

```
             eax6      thing2,*7

             staq      1p|.is,*6

             ...

  thing 2: arg         xx0028,7
```

Page 36, lines 458-470.

This code picks up the argument misdeclared,

```
     dcl name char(1);
```

Note the comment "idiotic". It means we are referencing a short string parameter. In general this comment comes out whenever a short string may or may not cross a word boundary: in this case lines 464-466 are unnecessary, but EPL is not set up to take advantage of the fact that by convention single characters never cross word boundaries.

Page 38, lines 544-555.

The subroutine .of0 computes a word offset and a shift from string dope. Once upon a time this subroutine included a test for whether the string was packed or aligned. At some point in the past year someone took the test out of this compiled code, and left us open to an interesting bug. EPL does occasionally pass aligned strings: for example the call on page 33, lines 306-328, passes an aligned string. We have not been able to find an example of such a case in which the "offset" in the dope is non-zero (which is the only case which causes trouble), so it is possible that this is not a bug so far as EPL is concerned. But this is an incompatibility between EPL and any other PL/I-like compiler for Multics, since the PL/I standards permit both packed and aligned strings with arbitrary offsets. The compiler must be changed.

Page 59, lines 56-74.

This is the code for the static array of varying strings list_name. Line 57-63 are the dope, and lines 65-74, in the internal static initialization code sequence, sets up the specifier and initializes using varst_$zero.

Page 59, line 77.

This is all that is needed for the array of fixed-point numbers list_value. No dope or specifier is needed, so pass 1.5 has eliminated these.

Page 60, lines 95-128.

This is the code for the call to do the assignment,

```
     list_name(list_top) = name;
```

The code breaks down as follows:

| | |
|---|---|
| 98-100 | copy list_top into an automatic temporary. |
| 101-104 | Make bb←bp point to the array data and ab←ap point to the array dope. |

| | |
|---|---|
| 105-107 | Make bb<-bp point to the particular element of the array. |
| 109-110 | Save bb<-bp and ab<-ap for later. |
| 114-117 | Put in argument pointer for <u>name</u> (the easy one). |
| 118-119 | Restore saved bb<-bp and ab<-ap. |
| 120-124 | Create varying-string dope and specifier for the array element in stack locations 44-51. |
| 125 | Store a pointer to this specifier into argument list. |
| 126-128 | Make the call. |

Page 69, line 21.

Note that now list_name does not need dope and specifier any more.

Page 69-70, lines 41-66.

This is the call to do the assignment,

list_name (list_top) = name;

Compare with page 60, lines 95-128. Now list_name is an array of ordinary non-varying strings. The code breaks down into:

| | |
|---|---|
| 44-46 | Copy list_top into an automatic temporary. |
| 50-53 | Make argument pointer for <u>name</u> (the easy one). |
| 54-56 | Make specifier and dope for the array element in stack locations 40-45. |
| 63 | Store a pointer to this specifier into the argument list. |
| 64-66 | Make the call. |