The use of character sets and character mappings in Icon*

R. E. Griswold

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, USA

This paper describes the properties and use of character sets and character mappings in the Icon programming language. Examples of programming techniques based on these features are given to illustrate paradigms for solving a variety of nonnumerical problems. Such solutions are characterised by efficiency and compact data representations.

(Received April 1979)

SNOBOL4 (Griswold, Poage and Polonsky, 1971) has an operation for mapping strings according to correspondences between sets of characters and it has pattern matching operations that deal implicitly with sets of characters. SNOBOL4, however, has no character set type *per se*, although typical implementations of SNOBOL4 support character sets internally (Griswold, 1972; Gimpel, 1974). The emergence of character sets in the Icon programming language (Griswold, Hanson and Korb, 1979) represents the linguistic elevation of an implementation mechanism to full status as a source language feature. The consequences of this elevation have exceeded the mechanisms for which they were originally developed, however.

The character set and character mapping facilities in Icon, used in conjunction with its string processing facilities, support a number of unusual programming techniques that can be used to advantage in a variety of non-numerical programming problems.

This paper describes the features of Icon that are important to these techniques and characterises their usage. Examples are given to illustrate the major paradigms.

1. An overview of Icon

Icon is a programming language that is intended primarily for non-numerical applications with an emphasis on string processing. While Icon resembles SNOBOL4 in the level of its language features and in its support for ease of programming, it has a modern syntax and traditional control structures. An example is

if *e*1 **then** *e*2 **else** *e*3

where *e*1, *e*2, and *e*3 are arbitrary expressions. Control structures such as **if-then-else** and **while-do** are driven by conditional expressions that signal success or failure as in SNOBOL4. Relational expressions are typical:

if $x \Leftarrow y$ **then** $z := x$ **else** $z := y$

As a result of executing this construction, z is assigned the value of x or the value of y depending on whether or not x is less than or equal to y.

One iterative control structure is

every *e*1 **do** *e*2

where *e*1 is a *generator* that produces a sequence of values. One commonly used generator is

i **to** *j* **by** *k*

which generates the integer values from i to j in increments of k. This generator, used in the **every-do** control structure, produces the effect of the familiar **for** statement of ALGOL:

every $n := i$ **to** j **by** k **do** *e*2

For other examples of generators, see Griswold, Hanson and Korb (1979).

Icon has a large repertoire of string operations. Some representative operations are:

size (*s*)	number of characters in *s*
*s*1 || *s*2	concatenation of *s*1 and *s*2
repl (*s*, *i*)	concatenation of *i* copies of *s*
s[*i*]	*i*th character of *s*
substr(*s*,*i*,*j*)	substring of *s* starting at *i* of length *j*

String-valued keywords provide some useful constants:

&lcase	string of lower case letters
&ucase	string of upper case letters

An Icon program is composed of a sequence of procedures. An example of a procedure is

```
procedure max (i, j)
  if i > j then return i else return j
end
```

The **return** expression returns the value of the procedure call, as indicated.

Procedures may have local identifiers. Local identifiers are ordinarily dynamic (automatic) and exist only during an invocation of the procedure. Local identifiers may be declared to be static, however, in which case they survive from one invocation of the procedure to the next. Procedures may also have an initial clause that specifies expressions to be executed on the first invocation of the procedure. Uses of these features of procedures are illustrated in subsequent examples.

2. Character sets

There are a variety of character sets in use on different kinds of computers. They differ in size, in the relationship between the internal representations of characters for control functions and external graphics, and (hence) in collating sequence. The most commonly used character sets are ASCII (American National Standards Institute, 1977), EBCDIC (IBM Corporation, 1976) and various forms of BCD (Control Cata Corporation, 1971). Internally, a character is simply an integer in the range from 0 to one less than the size of the character set. Thus in ASCII there are 128 characters with internal representations from 0 to 127 (decimal), inclusive.

Most of the programming techniques described in this paper depend on the use of characters within a program, rather than their input or output. Where graphic representations are important, upper and lower case letters are useful, but not essential. Any of the common collating sequences suffice. The size of the character set is significant, however, since in a number of applications individual characters are used to represent or label other objects.

The internal character set of Icon has 256 members. This character set is independent of the size of the character set for the host computer on which Icon runs. The internal character set and the host character set are interfaced only by input and output routines. Although its character set has 256 members,

*This work was supported by the National Science Foundation under Grants MCS75-01307 and MCS79-03890.

Icon is ASCII based and the first 128 characters have ASCII interpretations. The usefulness of the remaining characters is illustrated in subsequent examples. It is assumed for ease of presentation that both upper and lower case letters are available on the host machine. This assumption is not essential, however, since Icon provides escape conventions for the literal representation of any internal character, regardless of input limitations that may be imposed by the host computer (Griswold and Hanson, 1979).

Character sets in Icon, called csets for short, may have from 0 to 256 members. The value of the keyword &cset is a cset containing all 256 characters.

Csets are constructed from strings using the built-in function cset(s), which produces a cset consisting of the characters in the string s. While a string may contain duplicate characters, a cset cannot, of course. Similarly, the order of characters in s is irrelevant to the resulting cset. Thus

cset ('armada')
cset ('ramada')
cset ('drama')
cset ('dram')

all produce equivalent csets.

Aside from type conversions, there are four built-in operations defined on csets:

$\sim c$	complement with respect to &cset
$c_1 ++ c_2$	union
c_1 ** c_2	intersection
$c_1 -- c_2$	difference

The creation of a cset from a string may be considered to be type conversion. Conversely, a cset may be converted to a string using the built-in function string(c). In this operation, the resulting string is alphabetised, that is, the characters of c are placed in the string according to their relative position in the collating sequence. For example,

alpha := string (&cset)

assigns to alpha a string consisting of all the available characters in order of their collating sequence. This string has a number of computational uses and is referred to from place to place throughout this paper.

As a consequence of the properties of these conversions, the result of

s_2 := string (cset(s_1))

is a string s_2 that contains every distinct character of s_1 arranged in alphabetical order. This transformation can be used to advantage, as is described in later sections.

Icon also supports implicit type conversions, coercing arguments to expected type as the context demands. For example, if c_1 and c_2 are csets, c_1 || c_2 produces a string that is the concatenation of the results of converting c_1 and c_2 to strings.

3. Character mappings

Icon has a built-in function for mapping the characters in a string, map (s_1, s_2, s_3). This function produces a result in which every character of s_1 that appears in s_2 is replaced by the corresponding character in s_3. For example, the result of

map ('retroactive', 'aeiou', '-----')

is 'r-tr--ct-v-'. Different characters can also be mapped differently. The result of

map ('retroactive', 'aeiou', 'AEIOU')

is 'rEtrOActIvE'.

3.1 *Properties of character mappings*

The description of the map function given above is superficial. In order to use the full capabilities of this function, a more precise description is necessary. In the discussion that follows, the form of the operation is

s_4 := map (s_1, s_2, s_3)

1. The length of s_4 is the same as the length of s_1, regardless of the values of s_2 and s_3. In Icon terms, this is stated as

   size (s_4) = size (s_1)

   To remain in the domain of Icon as much as possible, this terminology is used subsequently.
2. The relative order of characters of s_2 and s_3 is significant, since it establishes the correspondence used in the mapping. Thus the two expressions

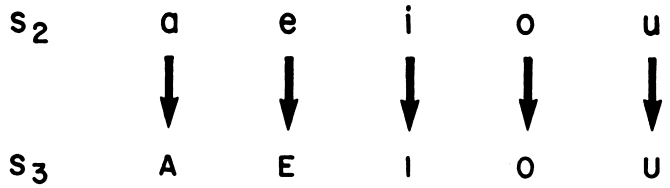
   map (s_1, 'aeiou', 'AEIUO')
   map (s_1, 'uoiea', 'UOIEA')

   produce the same result regardless of the value of s_1, but the two expressions

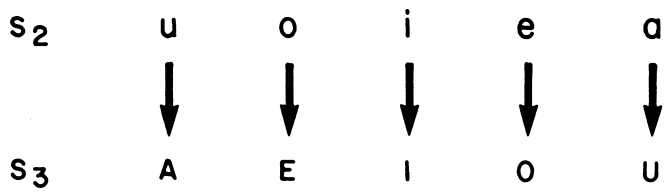
   map (s_1, 'aeiou', 'AEIOU') (1)
   map (s_1, 'uoiea', 'AEIOU') (2)

   produce quite different results, in general.

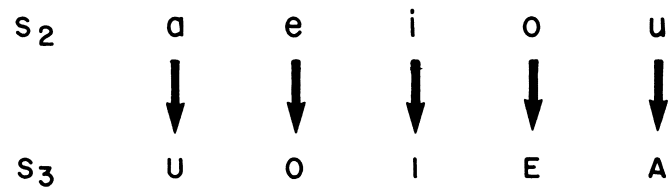
   Maps may be visualised as correspondences between characters in s_2 and s_3. The map for expression (1) is



   Expression (2) has the map



   Note that only the relative order is important. Thus the map



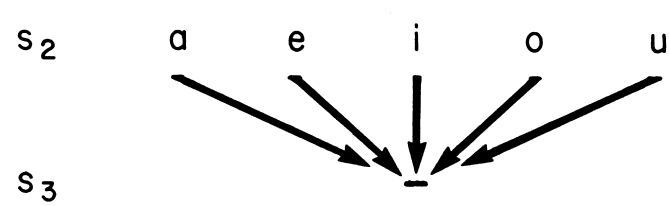
   is equivalent to the previous map. The expression

   map (s_1, 'aeiou', 'UOIEA')

   is also equivalent to expression (2).
3. As illustrated by

   map ('retroactive', 'aeiou', '-----')

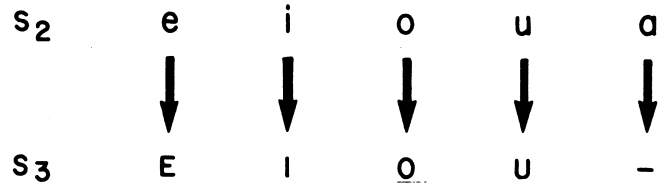
   s_3 may contain duplicate characters. This results in a mapping that is illustrated as follows



4. Duplicate characters in s_2 are permitted. In this case the last (rightmost) correspondence between s_2 and s_3 holds. For example, the map for

   map (s_1, 'aeioua', 'AEIOU-')

   is



For the purposes of discussion, it is convenient to deal with the *reduced forms* of s_2 and s_3, in which there are no duplicate characters in s_2. This leads to the concept of *canonical forms* in which s_2 is in reduced form and in alphabetical order and s_3 is rearranged accordingly. The expression above in canonical form is

map (s_1, 'aeiou', '-EIOU')

The symbols $\hat{s}_2$ and $\hat{s}_3$ are used for the canonical forms of s_2 and s_3, respectively. See the end of Section 3.4 for a method of computing canonical forms. In practice, it is often more convenient or efficient to use values of s_2 and s_3 that are not canonical or reduced. The map function can be thought of as performing the necessary canonicalisation.

5. Characters of s_1 that do not occur in s_2 appear unchanged in their respective positions in s_4. The map function can be thought of as establishing automatic correspondences between such characters and themselves, but such detail is cumbersome and is omitted from maps shown in this paper. It is worth noting that

map (s_1, s_2, s_3)

and

map (s_1, alpha || s_2, alpha || s_3)

are equivalent.

6. s_1, s_2, and s_3 may be of any size, provided that the sizes of s_2 and s_3 are the same. Hence size ($\hat{s}_2$) = size ($\hat{s}_3$) ⇐ size (alpha). Furthermore, as noted above, size (s_4 = size (s_1).

3.2 *Substitutions*

The use of map (s_1, s_2, s_3) in which s_2 and s_3 are fixed and s_1 varies is called a *substitution* for s_1. As a consequence of the properties listed in Section 3.1, the following condition holds:

Substitution inverse condition:
For fixed s_2 and s_3 and varying s_1, the substitution

s_4 := map (s_1, s_2, s_3)

has an inverse if and only if $\hat{s}_3$ is equal to $\pi(\hat{s}_2)$ for some permutation π. An inverse is

s_1 := map (s_4, $\hat{s}_3$, $\hat{s}_2$)

The classical use for this kind of mapping occurs in cryptography. Substitution ciphers, which by definition must have inverses, are used to substitute for characters of a message. The form of substitution given above is directly applicable to monoliteral substitutions. See Griswold (1975) for an extended discussion and for programming techniques in SNOBOL4 that can be directly employed in Icon.

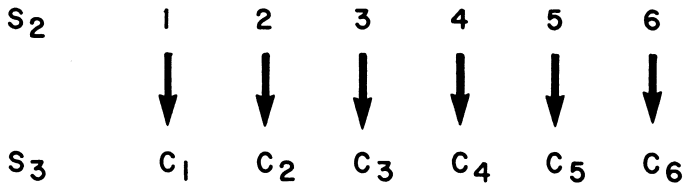
3.3 *Permutations*

The map function was originally designed to perform substitutions and its use for this purpose is obvious. Its use to effect permutations (rearrangements) is less obvious.

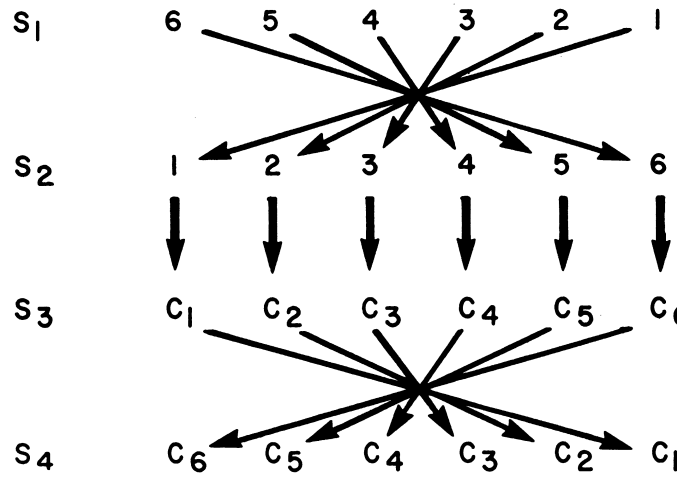
A simple example illustrates the technique. Suppose that the order of the characters of a string is to be reversed, end-for-end. As a specific case, suppose size (s_3) = 6. Then

s_2 := '123456'
s_1 := '654321'
s_4 := map (s_1, s_2, s_3)

produces the desired result. In this expression, the mapping between s_2 and s_3 depends on the particular characters in s_3. If s_3 consists of characters $C_1C_2C_3C_4C_5C_6$, then the map is



The desired permutation is accomplished, since the characters of s_1 are mapped through s_2, by relative position, into those of s_3, as illustrated by the following diagram.



From the example above, it is clear that the technique can be used to perform any permutation, provided s_3 is not longer than size (alpha). Specifically:

Permutation property:
If π is a permutation on a string of size n ⇐ size (alpha) and s_2 is a string of n distinct characters, then the result of

s_4 := map ($\pi(s_2)$, s_2, s_3)

is $s_4 = \pi(s_3)$. Furthermore, an inverse to the permutation is

s_3 := map (s_2, $\pi(s_2)$, s_4)

Note that for constant values s_2 and $\pi(s_2)$, the first expression above applies the permutation π to all strings s_3 of size n.

An application of fixed permutations applied to a set of strings occurs again in classical cryptography, where various transposition ciphers (route transposition, columnar transposition, and so forth) can all be seen as instances of this paradigm (Griswold, 1975).

3.4 *Positional transformations*

Permutations are a restricted case of more general *positional transformations* (Gimpel, 1976). A positional transformation $\rho(s)$ of a string s is a rearrangement of the characters of s in which

1. Any character in a specific position in s may appear in zero or more fixed positions in $\rho(s)$.
2. Additional constant characters, independent of the characters in s may appear in $\rho(s)$ at other fixed positions. These characters are called *nulls*.

For example, (*abc*) (*cba*) is a positional transformation of *abc*. The same positional transformation applied to *xxy* produces (*xxy*) (*yxx*). In this example, the parentheses are nulls.

Positional transformation property:
If $\rho(s)$ is a positional transformation, then the result of

s_4 := map ($\rho(s_2)$, s_2, s_3)

is s_4 := $\rho(s_3)$.

Obviously not all positional transformations have inverses.

For example

$$s_4 := \text{map ('fl', 'fllllll', } s_3)$$

produces a two-character string consisting of the first and last characters of a seven-character string s_3.

One form of positional transformation that always has an inverse is the permutation, as described in Section 3.3. The class of positional transformations with inverses is more general, however.

Positional transformation inverse property:
Given a positional transformation ρ, the mapping

$$s_4 := \text{map } (\rho(s_2), s_2, s_3)$$

has an inverse if and only if

1. All the characters in s_2 are distinct.
2. All characters in s_2 appear at least once in $\rho\ (s_2)$.

If these conditions hold, the inverse is

$$s_3 := \text{map } (s_2, \rho\ (s_2), s_4)$$

In the first place, if there is a duplicate character in s_2, only the last correspondence with s_3 will hold, and a character of s_3 will be deleted in the transformation and hence cannot be restored, in general, by any mapping.

Similarly, it is easy to see that if $\rho\ (s_2)$ does not contain some character in s_2, then the corresponding character in s_3 will not appear in s_4 and hence cannot be restored by any mapping. It is also easy to show that characters of s_2 can occur more than once in $\rho\ (s_2)$ and that nulls in $\rho\ (s_2)$ do not affect the inverse mapping (Griswold, 1978).

Note
The canonical forms in the substitution paradigm can be obtained as follows:

$$\hat{s}_2 := \text{string (cset } (s_2))$$
$$\hat{s}_3 := \text{map } (\hat{s}_2, s_2, s_3)$$

This mapping is the inverse of the positional transformation that maps $\hat{s}_2$ into s_2.

Positional transformations with inverses appear in classical transposition ciphers, such as grilles (Griswold, 1975), in which null characters are added to the cipher to obscure the transposed message. It is interesting to note that message characters can be duplicated in the cipher without interfering with the inverse deciphering process.

4. Applications and examples

As mentioned above, many of the models for substitution and positional transformation are found in classical enciphering techniques. While classical enciphering is no longer of practical interest, there are a number of important related applications that are of interest. The examples that follow illustrate techniques that may be useful in such cases. For brevity, program solutions are reduced to their essentials. Tests for the validity of arguments and so forth are deliberately omitted; these components can easily be added.

4.1 *Substitutions*

Example 1: *Case folding*
One of the common uses for substitution is to establish equivalences between characters by mapping one set into another. For example, it is often convenient to consider upper and lower case letters to be equivalent. Instances of this situation arise in command processors that are insensitive to case. To simplify processing, therefore, the input is 'folded' into a single case. The following procedure maps upper case letters into lower case ones:

```
procedure fold(s)
  return map(s, &ucase, &lcase)
end
```

Example 2: *Displaying card decks*
Another application of substitution is illustrated by the problem of manipulating and displaying a deck of playing cards. Here a standard deck of playing cards can be represented by 52 distinct characters. Although any 52 distinct characters can be used, it is convenient to use the upper and lower case letters, since their graphic representations facilitate program development and debugging. Therefore

```
deck := &ucase || &lcase
```

provides a 'fresh' deck. Since individual characters are used to represent the cards, shuffling can be done easily by character exchanges (Knuth, 1969):

```
procedure shuffle (deck)
  local m
  every m := size (deck) to 2 by -1 do
    deck [random(m)] :=: deck[m]
  return deck
end
```

(The operator :=: exchanges the subscripted characters.)

In order to display a shuffled deck, it is necessary to determine the suit and denomination of each card. The suits can be determined by a substitution in which the first 13 characters of an image of the deck are mapped into the character *C* (for clubs), the second 13 into *D* (for diamonds), and so on. The third argument to map in this case is

```
suits := repl ('C', 13) || repl ('D', 13) || repl ('H', 13)
  || repl ('S', 13)
```

Similarly, the denominations can be identified by associating the first character of each 13-character group of an image of the deck with *A* (for ace), the second character in each group by 2, and so on. The third argument of map in this case is

```
denoms := repl ('A23456789TJQK', 4)
```

A simple display of a deck of cards is then provided by the following procedure. The necessary constants are assigned to static identifiers in the initialisation that occurs during the first call to the procedure.

```
procedure display (deck)
  local static deckimage, suits, denoms
  initial {
    deckimage := &lcase || &ucase
    suits := repl ('C', 13) || repl ('D', 13) || repl ('H', 13)
      || repl ('S', 13)
    denoms := repl ('A23456789TJQK', 4)
    }
  write (map (deck, deckimage, suits))
  write (map (deck, deckimage, denoms))
  return
end
```

This procedure displays the deck with the suits on the first line and the denominations directly below. For example, if the shuffled deck begins with the 3 of clubs, the ace of hearts, and the 8 of spades, and so on, the display has the following form:

```
CHS . . .
3A8 . . .
```

A refinement to this display is given in Section 4.2.

Note that the technique used above is independent of the character set of the host computer on which Icon runs. Even if the host character set is BCD, the procedures above will work properly, since Icon supports a larger character set internally. Thus it is not necessary to change deckimage if the host character set does not contain lower case letters. The interface between the internal character set only occurs when the (upper case) results are written out.

Example 3: *Masking characters*

In order to isolate characters of interest from those that are not of interest, it is useful to map all 'uninteresting' characters into a single 'null' that is not in the set of interest. The following procedure substitutes the one-character string s_3 for all characters in s_1 that are not contained in s_2.

```
procedure mask (s1, s2, s3)
  return map (s1, ~s2, repl (s3, size (~s2)))
end
```

For example,

```
mask ('Watch for spooks', 'aeiou', '-')
```

produces -a-----o----oo--.

An alternate form of coding that uses duplicate characters rather than character-set complementation is

```
procedure mask (s1, s2, s3)
  return map (s1, alpha || s2, repl (s3, size (alpha)) || s2)
end
```

Here a correspondence between each character of alpha (the string of all characters) and s_3 is first established and then the correspondences of characters in s_2 with themselves are appended to override their earlier correspondences with s_3.

Example 4: *Extracting and displaying suits*

In card games like bridge, it is customary to sort hands into suits and to order the suits by denomination. All the cards in the same suit can be 'extracted' by substituting some null for all characters that are not in the desired suit. Standard 'templates' for the suits can be set up as follows:

```
blanker := repl (' ', 13)
denom := 'abcdefghijklm'
clubs := denom || repl (blanker, 3)
diamonds := blanker || denom || repl (blanker, 2)
hearts := repl (blanker, 2) || denom || blanker
spades := repl (blanker, 3) || denom
```

The mapping to get the clubs, for example, is

```
suit := map (hand, deckimage, clubs)
```

The identifier denom is used to associate the cards of each suit with the same denominations, regardless of suit. For example, the 2 of clubs and the 2 of hearts are both mapped into *b*. In each case, all characters that do not correspond to a given suit are mapped into a blank. Note that it is essential to select a null that is not among the characters used to represent the cards.

If the suit above is converted to a cset and back to a string, the result is an (alphabetised) version of the suit with a single instance of the null. A further substitition can be performed to get the correct visual representation of each card:

```
map (cset (suit), denom, 'AKQJT98765432')
```

If the hand contains the ace, queen, ten and two of clubs, the result would be AQT2.

Note that the null used here is 'invisible' in printed output, although it is actually the first character in the string produced above (because of the ASCII collating sequence). It can be removed easily, if desired. The final mapping to get the desired visual representation is done after the formation of the cset, since the visual representations are not in alphabetical order according to suit rank.

Other substitution applications

A number of other interesting uses of substititon are given in Gimpel (1976). Two examples are the translation of Roman numerals to a higher 'octave' in the conversion of Arabic numerals, and the use of ten's-complement arithmetic to effect symbolic subtraction by addition. Bit-string operations can also easily be simulated by mappings on strings composed of zeroes and ones (Griswold, 1978).

4.2 *Positional transformations*

Example 5: *Reversal*

The reversal of the order of characters in a string, as described in Section 3.3, is not of interest in itself, since there is a built-in function in Icon for performing this operation. The solution of the problem, however, serves as a model for a number of other positional transformations.

The approach is to provide, by conventional means, general templates for the transformation. The second argument of map serves as a labelling for the third argument, while the first argument is the desired permutation. The terms image and object are used to refer to these two strings, respectively. The following procedure uses a pair of strings chosen for visual clarity.

```
procedure reverse(s)
  local static image, object, rvsize
  initial {
    image := 'abcdefghijklmnopqrstuvwxyz'
    object := 'zyxwvutsrqponmlkjihgfedcba'
    rvsize := size (image)
    }
  if size(s) <= rvsize then
    return map (substr (object, rvsize -size(s) + 1, sizes(s)),
      substr (image, 1, size(s)), s)
  else
    return reverse (substr (s, rvsize + 1, size(s) -rvsize))
      || map (object, image, substr (s, 1, rvsize))
end
```

If s is not longer than the image template, the reversal is done in one mapping. In this case, specific templates of the correct length are selected from the general ones. Note that the first part of image is used, while the last part of object is used. If s is too long, it is divided into two portions. One portion is reversed by a recursive call, while the other is reversed using the full templates. This process can also be done iteratively at the expense of some complication of the code.

Note that the templates can be chosen in any convenient fashion, as long as the object is the reversal of the image. For maximum efficiency in reversing long strings, the templates should be as long as possible: alpha and its reversal. These strings can be formed as follows:

```
image := ''
every i := 1 to size (alpha) do
  image := alpha[i] || image
```

Alternatively, these strings can be obtained by bootstrapping, replacing the initialisation section of the procedure by

```
initial {
  image := 'ab'
  object := 'ba'
  rvsize := size (image)
  object := reverse (alpha)
  image := alpha
  rvsize := size (image)
  }
```

This technique has the advantage of using the most elementary characterisation of the positional transformation as well as avoiding possible errors in constructing the two long strings by other methods.

It is reasonable to question the usefulness of map to effect this permutation, since it can be more easily coded by concatenation as illustrated above. Both the concatenation method and the mapping method are approximately time linear in size(s) if secondary effects such as storage management anomalies are ignored. The conventional method is clearly linear. The map function itself is time linear in the sizes of its first and second arguments (see Section 6.2). In the procedure above,

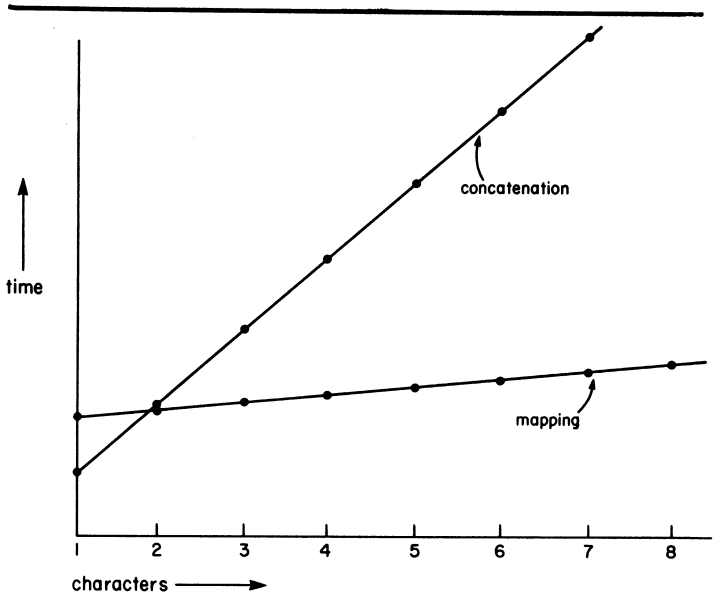


Fig. 1 Timings of string reversal methods

these two sizes are the same. Hence the mapping method is also time linear in size(s). Results of actual timings are shown in **Fig. 1.**

The interesting fact is that the (measured) constant of proportionality for the iterative method is nearly 8.5 times that of the mapping method. Furthermore, the crossover point is at two characters! That is, the two methods take about the same amount of time for two-character strings, and the relative performance of the mapping method improves rapidly thereafter. Although the space requirements in terms of transient storage are dependent on the details of internal storage management, the mapping method has the clear advantage of creating fewer intermediate strings. Part of the cost of transient allocation is shown in the relative constants of proportionality, but part is deferred in the form of garbage collection that may occur at unpredictable times to the detriment of the conventional method. These observations apply in general to the relative efficiency of effecting positional transformations by conventional means versus mapping.

Example 6: *Displaying a card deck*

The display of the deck of cards as described in Example 2 produces an unattractive result. A more attractive display is obtained if the suit and denomination of each card are adjacent and there are separators (say blanks) between each card. Here there are 104 objects to be labelled (52 suit characters and 52 denomination characters) and some 156 characters in the result if one separating character is placed after the representation of each card. While the result can be obtained with a single map using long image and object strings, for purposes of exposition it is more reasonable to divide the result into sections, say four sections of 13 cards each. Convenient image and object strings are

```
image  := 'ABCDEFGHIJKLMabcdefghijklm'
object := 'Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm'
```

The upper case letters label the suits and the lower case letters label the denominations. The suit and denomination strings are then concatenated before mapping. A procedure is

```
procedure display (deck)
  local i
  local static image, object, deckimage, suits, denoms
  initial {
    image := 'ABCDEFGHIJKLMabcdefghijklm'
    object := 'Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm'
    deckimage := &lcase || &ucase
    suits := repl ('C', 13) || repl ('D', 13) || repl ('H', 13)
     || repl ('S', 13)
    denoms := repl ('A23456789TJQK', 4)
    }
  every i := 1 to 52 by 13 do
    write (map (object, image, map (substr (deck, i, 13),
     deckimage, suits)))
end
```

Example 7: *Directed graphs*

While it is customary to represent directed graphs by list structures or adjacency matrices, they can also be represented by character strings in which a distinct character is associated with each node and in which the arcs are represented as character pairs. Consider the graph shown in **Fig. 2.**

This graph has arcs AB, AC, CD, BD, and DD. As a single string, this graph is represented by

$$g := \text{'ABACCDBDDD'}$$

(If nodes without connecting arcs are allowed, a string containing all the nodes may be kept separately). This representation is very compact and with the string processing operations of Icon, many graph operations can be performed economically. For example, a procedure to compute the number of arcs in a graph is simply

```
procedure arccount(g)
  return size(g)/2
end
```

An example of the use of this representation is given by a procedure to determine the transitive closure of a node in a graph:

```
procedure closure (n ,g)
  local st, sn
  sn := n
  while (tn := sn ++ successors (sn, g)) ~=== sn do
    sn := tn
  return tn
end
```

The procedure successors (*sn*, *g*) returns all direct successors in *g* of nodes in the cset *sn*. Definition of this procedure is left as an exercise. The operation $x \sim=== y$ succeeds if x and y are different csets, so the loop continues until nothing new is added to the cset. It should be noted that *all* direct successors of the nodes in the evolving cset are added at each step.

Although the representation used above is compact and easy to manipulate, it is not suitable for display purposes. A positional transformation can be used to produce a much more attractive display. Using an image and object of the form

```
image  := '12'
object := '1 → 2;'
```

the display of the graph becomes

A → B; A → C; C → D; B → D; D → D;

It is a straightforward matter to generate longer image and object strings and to write a general purpose procedure for producing the display. Translation between various formats for

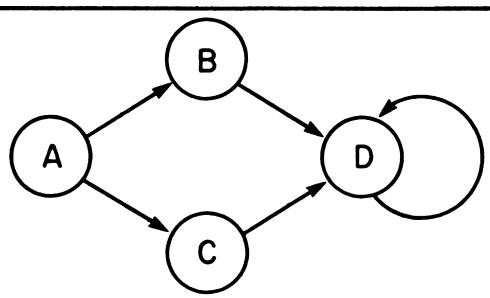


Fig. 2 A simple directed graph

input, output, display and internal manipulation are easily derived in this manner.

Other applications of positional transformations

Gimpel (1976) and Griswold (1975) provide numerous examples of positional transformations ranging from the reformatting of dates to the generation of pig latin. Conversion among binary, octal and hexadecimal representations of characters can also be accomplished with such techniques (Griswold, 1978).

5. Implementation

The techniques used to implement string and cset operations are of interest here only to the extent that they affect the efficiency of the programming techniques that have been described. See Hanson (1978) for a description of dynamic storage management in Icon and the details of data layout.

5.1 *Character sets*

Character sets are represented as bit strings, with the bit in the position of the character in collating sequence, set to 1 if the character is in the character set and set to 0 otherwise. The amount of space required for a cset depends on the size of the internal character set (256 in Icon), not on the particular characters the cset contains. In any event, csets require comparatively little storage space.

The construction of a cset from a string involves processing the characters of the string in sequence, setting the corresponding bit in the cset. This process is time linear in the size of the string. Constructing a string from a cset involves the converse process and is also time linear in the number of characters in the cset.

Complementing a character set is time linear in the number of characters not contained in the character set, but is a comparatively fast operation compared to those that involve accessing characters. The other built-in character set operations are also time linear and correspondingly efficient.

5.2 *Mapping*

Map (s_1, s_2, s_3) is performed by first building a table of correspondences between the characters of s_2 and those of s_3. This table contains one entry for each possible character (256 in Icon) and it is initialised by having each character correspond to itself. Then the entry for each character is s_2 is replaced by the corresponding character in s_3, working from left to right. Thus if there are duplicate characters in s_2, the last (rightmost) correspondence results naturally. Once the table is built, the result is constructed by indexing the table with successive characters of s_1.

The amount of time required to build the table of correspondences is proportional to the size of s_2 and the amount of time required to do the actual mapping is proportional to the size of s_1. Thus the total time required for the mapping is approximately

$$a \times \text{size}(s_1) + b \times \text{size}(s_2) + c$$

where c is constant overhead that includes the initialisation of the table of correspondences.

The table of correspondences is static. The only storage allocation required for mapping is for the resulting string. Furthermore, if map is called successively with the same values of s_2 and s_3, the previous table of correspondences is used without reinitialisation.

6. Conclusions

The character set and string processing facilities of Icon make programming techniques feasible that otherwise would require entirely different approaches. The main advantages of these techniques are the compactness of the data representations and the comparative efficiency of the operations.

This efficiency is largely obtained by the internalisation of processes that would ordinarily involve loops at the source language level. Specific examples are identifying distinct characters, sorting them using cset(s), and the substitution and positional transformation of long strings using a single mapping operation. Given appropriate computer architecture, character sets can be manipulated as bit vectors, with the potential improvement in efficiency that can be obtained from parallel operations (Aho, Hopcroft and Ullman, 1976).

The main limitation on the programming techniques described in this paper are imposed by the limited size of the character set. In positional transformations, this is usually more of an annoyance than an actual limitation, since most positional transformations (such as the reversal of a long string) can be decomposed into a sequence of shorter transpositions. However, if the scope of the transposition requires more labels than there are characters in the character set, a different technique has to be used. The really serious limitation occurs in the use of characters to label objects. The representation of a deck of playing cards in this way works nicely, but that is merely a convenient coincidence. In the case of graphs, the representation used clearly limits the cases that can be handled. Furthermore, since the methods specifically rely on character operations, there is no way to extend the techniques if the size of the character set is inadequate.

It is interesting to note that csets are useful in their role as sets independent of their relationships to specific characters, despite the limitation on the number of objects that can be represented. At the same time, csets provide an economical facility, largely because they *are* limited in size.

There is no inherent reason why a language character set should be restricted to the character set of the host machine. Indeed, in the CYBER 175 implementation of Icon, the language character set is four times the size of the (standard) host character set and on the DEC-10 it is twice the size of the host character set. Character sets larger than those normally supported by any computer could easily be implemented, increasing the scope of the string processing facilities.

The problem of supporting a language character set that is different from the host character set is not as difficult as it might appear. In Icon, the size of a character (and hence of character sets) is an implementation parameter. Icon was originally designed for 128 characters and later changed to 256 characters to allow more flexibility. The change was easily accomplished in less than an hour. Furthermore, an internal character set that is independent of the host character set is an advantage, especially for enhancing portability, since the bulk of the system is written in machine independent form with known collating sequence (ASCII in the case of Icon). For example, the lexical analyser is machine independent, whereas if the internal character set varied according to the host character set of the target computer, there would be many complications.

The penalty for a larger character set is primarily in the space required for representing csets and strings. Doubling the size of the character set approximately doubles the amount of space required for storing a cset proper, although there is some storage overhead that is independent of the size of the character set. Similarly, the larger the character set, the more space is required for each character of every string. The time for some operations is increased also. The larger size of strings may require more time in data movement and the number of items that have to be processed is increased for cset operations and the correspondences established in map.

On the other hand, such 'super character sets' would extend the domain of applicability of the techniques described in this paper. Although it is beyond the scope of this paper, a potentially more important advantage of very large character sets

lies in their capacity to provide internal representations for larger sets of graphics than are supported by the host character set and hence in the processing of data for devices like phototypesetters.

Acknowledgement

Morris Seigel first called my attention to the use of mapping in SNOBOL4 to reverse strings and to the corresponding use of the translate instruction on the IBM 360 (Seigel, 1969; Computer Usage Company, 1966). I am indebted to Jim Gimpel for introducing me to the full range of transformations that can be performed by mapping. Students in my classes have served in an exemplary manner as guinea pigs for various experiments with the use of character sets and mappings. In addition, David R. Hanson and John T. Korb have provided helpful suggestions on the presentation of the material in this paper.

References

AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. (1976). *The Design and Analysis of Computer Algorithms*, Addison Wesley.
AMERICAN NATIONAL STANDARDS INSTITUTE (1977). *USA Standard Code for Information Interchange*, X3.4-1977.
COMPUTER USAGE COMPANY (1966). *Programming the IBM/360*, John Wiley & Sons, p. 208f.
CONTROL DATA CORPORATION (1971). *SCOPE Reference Manual*, Publication Number 60307200.
GIMPEL, J. F. (1974). The minimization of spatially multiplexed character sets, *CACM*, Vol. 17 No. 6, pp. 315-318.
GIMPEL, J. F. (1976). *Algorithms in SNOBOL4*, John Wiley & Sons, pp. 46-51.
GRISWOLD, R. E. (1972). *The Macro Implementation of SNOBOL4; A Case Study in Machine-Independent Software Development*, W. H. Freeman.
GRISWOLD, R. E. (1975). *String and List Processing in SNOBOL4, Techniques and Applications*, Prentice-Hall.
GRISWOLD, R. E. (1978). *Programming Techniques Using Character Sets and Character Set Mappings in Icon*, Technical Report TR 78-15a, Department of Computer Science, The University of Arizona.
GRISWOLD, R. E. and HANSON, D. R. (1979). *Reference Manual for the Icon Programming Language*, Technical Report TR 79-1, Department of Computer Science, The University of Arizona.
GRISWOLD, R. E., HANSON, D. R. and KORB, J. T. (1979). The Icon programming language; an overview, *SIGPLAN Notices*, Vol. 14 No. 4, pp. 18-31.
GRISWOLD, R. E., POAGE, J. F. and POLONSKY, I. P. (1971). *The SNOBOL4 Programming Language*, 2nd ed, Prentice-Hall.
HANSON, D. R. (1978). *A Portable Storage Management System for the Icon Programming Language*, Technical Report TR 78-16, Department of Computer Science, The University of Arizona.
IBM CORPORATION (1976). *System/370 Reference Summary*, Form GX20-1850-3.
KNUTH, DONALD, E. (1969). *The Art of Computer Programming*, Vol. 2, Addison-Wesley, p. 125.
SEIGEL, M. M. (1969). Letter to author.

Book reviews

The Phenomenon of Science; A Cybernetic Approach to Human Evolution by V. F. Turchin, 1977; 348 pages. (*Columbia University Press*, £17·50)

Cybernetics is the science of control and communication. Systems having internal communications which form feedback loops may display self-organising properties, and nowhere more so than in the phenomenon we call 'life'. Dr Turchin considers evolution in this cybernetic sense, from its beginnings, through the emergence of mankind, to the intellectual structures which have evolved at an increasing rate during recent centuries up to the present decade. A central theme of his discussion is the notion of a metasystem. This is essentially the same idea as iteration which can lead to exponentially increasing rates of evolution; as for example when tools are used to make more powerful tools, or when common language is used to make the powerful languages of symbolism, including many branches of mathematics.

Thus natural selection does not merely favour survival of the genotype itself, but selects for the appearance of selectable features; and so iteratively in a hierarchy of loops. The molecular biology of how this may happen has been outlined in a recent article by Professor Darlington (1979) who, approaching this question as a biologist, follows arguments that are generally consonant with those of Turchin from his general systems standpoint.

Turchin's conclusions are rationalistic and optimistic. He sees science and reason as the the true path for mankind, and believes that aesthetic and moral values have in the long run the formidable support of the mechanisms of evolution. His book is a product of deep thought, and therefore a provoker of it, well worth its reading and study.

P. B. FELLGETT (Reading)

Reference

DARLINGTON, C. D. (1979). Feedback and Evolution, *Kybernetes*, Vol. 8 No. 4, pp. 275-284.

Business Systems Handbook by Robert W. Gilmour, 1979; 229 pages. (*Prentice-Hall*, £14·55)

In the author's words: this is a 'working' handbook. It is intended to provide a basis for an organisation to develop their own documentation and procedures. It is not claimed for the book that it helps with the techniques of analysis and design and where these are discussed, it is at a very superficial level. Sections dealing with forms design, writing style and presentations are treated in great detail.

The advice given by Mr Gilmour on documentation and procedures is well illustrated and is basically sound, although a certain naivety shows through. The suggestion that a systems analyst should arm himself with a timepiece and indulge in DIY time study, albeit informing supervisors and employees before doing so, might well see the end of systems work in an organisation for a very long time. This contrasts oddly with the author's stringent standards for those called upon to give presentations to management. Although, in this context, the recommendation that the presenter should break up the monotony of his delivery by making gestures, amongst other activities, makes one wonder what the author has in mind.

Although useful, I would not regard this book as being amongst the frontrunners as far as value for money is concerned. An organisation on the brink of adopting standards would do better, in my view, to look to some other text. Preferably the one that originated not a million miles from Eindhoven.

C. D. EASTEAL (London)