28 May 1991

A Subsequence Algebra: First Class Values for Substrings

Wilfred J. Hansen
Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract. Strings are a basic data type in most programming languages, but substrings are seldom accorded first class status on a par with, say, integers. Substrings are important as the result of search and parsing algorithms since the calling routine may need to access both the matched substring and adjacent text or punctuation. To promote substrings, this paper describes a new algebra for subsequences which, when specialized to substrings, yields appropriately first class values. The key idea is that the basic data type is not sequences or references to positions in sequences, but rather references to subsequences. Primitive operations for the algebra are constants, concatenation, and four new functions--*base*, *start*, *next*, and *extent*--which map subsequence references to subsequence references.

This paper informally presents the algebra, shows that it is sufficient to define search functions, and then contrasts it with other models of substring values. Later sections of the paper contrast various models of substring values, show how the subsequence algebra can be concisely implemented, and touch on the many other aspects and advantages of the algebra. Examples are given in Ness, a language incorporating the algebra which is implemented as part of the Andrew Toolkit.

Keywords: strings, substrings, sequences, programming language design, applicative programming, string searching, rich text, document processing, desktop publishing, Andrew Toolkit, ATK, Ness

Despite the importance of text and strings, programming languages have offered no innovations in string data types or operations since the introduction of pattern matching and *substr* which happened at least as early as COMIT [Yngve, 1963] and PL/I [IBM, 1965], respectively. The most recent innovations, in Icon [Griswold, 1983], retain the standard string data types and augment them with carefully designed control structures.

This paper defines and demonstrates a new data type for strings by introducing a subsequence algebra and specializing it to strings. In the algebra each value is a reference to a subsequence of a base sequence, so each single string value represents an entire substring. With other string models multiple variables are required to represent a substring, leading to more complexity and errors. Please note that although this paper discusses an algebra, the presentation is informal and not algebraic. See [Hansen, 1989a] for a formal definition.

That there is a need for a new string data type is a consequence of three emerging trends:

Desktop publishing is accustomizing users to text with typographic formatting, multiple character sets, and even embedded objects: rasters, drawings, equations, footnotes, references, and so on. Such text can be dealt with in existing programming languages by the addition of various library packages and augmentation of the compiler and runtime system to accept non-ASCII characters in string values. However, since much effort is required to make these enhancements it is appropriate to introduce a new data type at the same time.

Applicative programming can be characterized as programming without side effects; an expression as written can be examined purely for its value so the reader need not keep the many details of possible side-effects in mind. The psychological advantages of this approach have not been explored in depth, but are related to the notions of modularity reviewed in section 3.4.2 of [Shneiderman, 1980]. As section 4 will show, traditional string value architectures encourage the use of side effects, at least to the extent of requiring two separate statements to record the position of a substring in one variable and its length in another.

"Professional non-programmers" denote professionals who are not programmers, but who happen to program computers as a tool in their work. Recent interactive systems such as Hypercard [Atkinson, 1987] have introduced programming languages intended for these people. While it is true that these languages do not permit control over every CPU cycle, they compensate by allowing clear and concise programs.

To satisfy these three emerging trends, the string facility of a language should be something that can be described as follows.

Simple. Programs are short and straightforward. A minimal number of data types and concepts are required to read and write programs.

First Class. Strings and substrings are as well supported as numeric values. Syntactic forms are offered for concatenation, string constants, and declaration of substring variables. Semantically, substring values can be passed as arguments to functions and returned as values. Comparison and assignment apply to substrings.

Unbounded. String values are not declared with size bounds and string expressions have no such bounds. The implementation manages the space for all strings.

Rich. String values support typographic styling, a large, potentially infinite character set, and the inclusion of embedded objects such as images, equations, and tables.

No widely used programming languages combine all four characteristics. Most lack Simplicity and First Class substrings, as will be discussed in Section 4. A few language provide Unbounded string values; for instance Icon, LISP [Steele, 1984], Awk [Aho, 1979], and REXX [IBM, 1987]. The latest version of C [ANSI, 1990] has reached toward Rich strings to the extent of offering a data type for "wide" characters and library functions for conversions between wide and multi-byte representations. Despite this added complexity, C cannot be said to support typographic styles or embedded objects. That no language currently offers Rich strings is not surprising given how recently desktop publishing has become practical. Most languages with First Class strings could be extended to provide Rich strings as well by redefining the syntax of string constants, providing a suitable implementation and defining functions to operate on typography and embedded objects.

The only two languages satisfying all four criteria, Ness [Hansen, 1989b; Hansen, 1990] and cT [CDEC, 1989], both base their string data type on the subsequence algebra presented here. Both were originally implemented under the Andrew Toolkit (ATK) [Morris, 1986; Palay, 1988], although cT has recently been re-implemented. The capability range of ATK is illustrated by this paper: a single file with various embedded objects created using ATK’s ez text editor. Examples below are given in Ness, the full power of which can be noted from the fact that under 2600 lines of Ness code are needed for a translator to ATK format from Microsoft’s Rich Text Format (RTF) [Microsoft, 1990]. Typographic styling is permitted in Ness programs; the programs below were compiled and executed without removing the styles.

In this paper, the subsequence algebra is introduced in the first section followed by descriptions of non-primitive and searching functions in the next two sections. The fourth section contrasts the algebra with alternative models of strings. The comparison is continued in the fifth section with an in-depth examination of solutions to a practical problem. The sixth section discusses implementation showing that the algebra cannot be comfortably implemented as a subroutine package but can be readily compiled. Many additional aspects of the algebra are summarized briefly in the seventh section.

1. An Algebra for Subsequences

We first define a subsequence reference algebra and then specialize it to Rich strings.

A subsequence reference algebra is a four-tuple [**E**, **S**, **R**, **O**] where **E** is a arbitrary set of elements and **O** a set of operations defined over a domain **R** derived from the set **S** of sequences over **E**. Specifically we define these terms:

> A *sequence* in **S** is a finite, ordered collection of elements from **E**. Before each element and after the last is a "*position*".[1]
>
> Each element of **R** is a triple [**b**, **l**, **r**] where **b** is a sequence in **S** and **l** and **r** are positions in **b**.
>
> An element of **R** is called a *subsequence reference*, or *subseq*. A subseq [**b**, **l**, **r**] is said to *refer* to the elements of **b** between the two positions **l** and **r**. If **l** and **r** are the same, the subseq is said to be *empty*.

For discourse, the elements in a sequence are imagined to be in a horizontal line with earlier elements to the "left" of later elements. Note that empty sequences are not all equivalent; they may differ as to their locations within their bases. Usually the operators **O** will include a nullary operator for constants which maps the denotation of a sequence of elements into a subseq with that sequence as a base and positions at its opposite ends. Other operators map subseqs to subseqs.

When a subsequence algebra is incorporated as a data type in a programming language, there is no need for values which are elements of **E** or **S** since both can be represented by elements of **R**. For a single element from **E**, the subseq in **R** has a base string containing the element and the two positions on each side of it. For a sequence from **S**, the subseq in **R** has that sequence as its base and the two positions at opposite ends.

For common programming languages the specialization of a subsequence reference algebra would define the set **E** as set of ASCII characters. Several languages implement the operator set **O** with mixed domain functions like *substr(r, i, j)* where the first argument is a string and the others are integers. Usually however, the resulting reference is to a new base sequence rather than the original *r*. The specialization in this paper employs a richer set of elements, a set of operations defined solely over the domain of substrings, and result values that refer to the base strings of their arguments.

When the elements referred to by a subseq are characters it will be common below to talk of it as a *substring*. Figure 1 shows three substring or subseq values, *m*, *s*, and *p*, defined on the base sequence "**’Twas brillig and the slithy toves**" and referring respectively to "**s bril**", an empty subsequence, and **toves**.

---

[1] In a more formal presentation, **S** is a mapping from a range of integers to elements and positions are denoted by integers. This precision is not needed here.

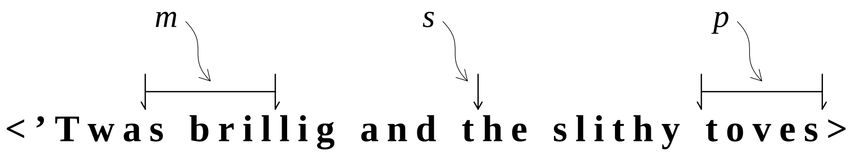


Figure 1. Three subseq values on a base sequence. The base sequence is shown between < and >. The end positions of subseq value are shown as arrows pointing between elements of the base. For an empty subseq the value is shown as a vertical arrow; for a non-empty subseq the value is shown as two half arrows joined with a horizontal line.

In the rest of this paper we specialize to "the" subsequence algebra [**E***, **S***, **R***, **O***] where:

> An element in **E*** is a character from an arbitrary character set with arbitrary typographic styling or an object implemented in an object-oriented programming system.
>
> **S*** and **R*** are the sets of sequences and references to subsequences analogous to **S** and **R**.
>
> The operators **O*** are the set
>
> {"...", *start*, *base*, *next*, *extent*, ~}
>
> as defined below.

The operators *base*, *start*, *next*, and *extent* are illustrated in Figure 2. These and the others are defined thus:

> **"..."** - denotes the set of nullary **constants** with a member of **S*** substituted for the ellipses. The value produced is a reference to the base sequence composed of the elements of that member of **S*** with positions at the opposite ends.

For example, we might have the constant:

> "碁を打ちますか is Japanese for *Do you play go?*"

which illustrates Rich strings whether the Japanese characters are in a font or in a raster image as they are here.

> **start(x)** - Returns the empty subseq at the beginning of its argument. Specifically the value is on the same base as *x* and has both positions the same as the leftmost of the positions in *x*.
>
> **base(x)** - Returns a subseq for the entire base of x. Specifically, the return value is on the same base as *x* and has the two positions at opposite ends of that base.

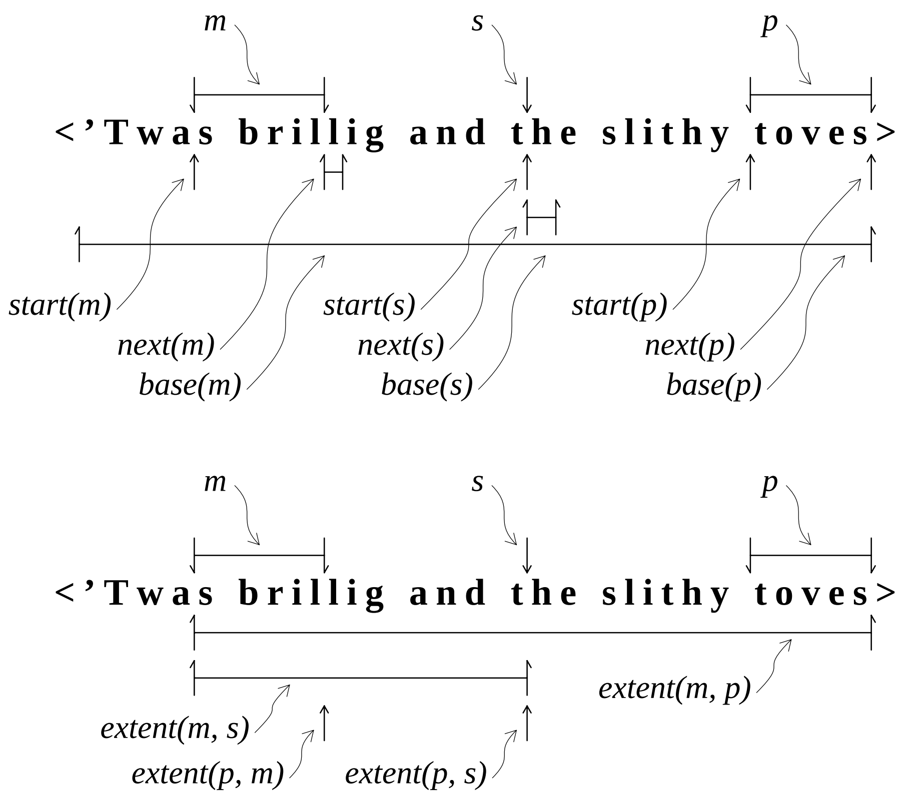


Figure 2. Four primitive functions. The subseq values below the base show the result of applying the primitive functions to *m*, *s*, and *p*. Values for *extent(s, m)* and *extent(s, p)* are given in the text.

In the Figure, *start(m)* is the empty subseq between **a** and **s**; and the values of *base(m)*, *base(p)*, and *base(s)* are each the entire sequence between < and >. To get an empty subseq at the beginning of *x*'s base sequence, we can write *start(base(x))*. The opposite composition, *base(start(x))*, returns exactly the same value as *base(x)* because *x* and *start(x)* are both on the same base sequence.

> **next(x)** - This function returns a subseq for the element following *x*. Specifically, the base of the result is the same as that of *x*, one position is at the rightmost position of x, and the other position is one element further to the right in the base. If the argument *x* extends all the way to the end of its base string, then *next(x)* returns an empty subseq for the position at the end of the base.

Next(m) and *next(s)* in the figure are both single elements, while *next(p)* is empty. The element just after start of *x* is *next(start(x))*, while the empty subseq at the end of *x* is *start(next(x))*. *Next(base(x))* is the empty subseq at the end of *base(x)*.

> **extent(x, y)** - In general, returns a subseq for everything from the beginning of *x* to the end of *y*. Specifically, when *x* and *y* are on the same base, the result subseq is also on that base and has one position at the end of *start(next(y))*; the other position is either *start(x)* or *start(next(y))*,

whichever is further left in the base. If *x* and *y* are on different bases, the result is an empty subseq on a unique empty base.

One non-empty result in Figure 2 is *extent(s, p)* which extends from *s* to the end of the base; conversely, *extent(s, m)* gives an empty subseq at the same position as *extent(p,m)*. All of the base before *m* is *extent(base(m), start(m))* and all of the base after *m* is *extent(next(m), base(m))*; observe that for both the result is shorter than *base(m)* even though that is one of the arguments.

Given two subseqs *x* and *y*, it is possible to determine whether both are on the same base sequence as long as both are not on empty bases. They are on the same base if *(base(x) = base(y) and extent(base(x), base(y)) = base(x))*.

x ~ y - Tilde denotes **concatenation** and generates a new base string composed of copies of the subsequences referred to by *x* and *y*. The value returned is a reference to the new base string with positions at its opposite ends.

In terms of Figure 2, *m ~ p* is a subseq whose base is the new value "**s briltoves**" and whose positions are at the extremes of that base.

In examples below, subseq variables are declared[2] with the form:

subseq m, p, s

Function arguments and values are subseq values by default. Assignment of subseq values does not copy the string referenced; instead it copies only the reference. Comparison, however, compares the strings referenced and does not distinguish between strings on different bases.

Given a subseq value we can write simple operations to scan through the base string. If *m* refers to a blank, we can advance it to point to the nearest following non-blank with:

while m = " " *do* m := next(m) *end while*

Note that this loop will terminate with *m* referring to either a non-blank or to the empty subseq at the end of the base. Of course, if *m* originally referred to a non-blank, it would remain unchanged. Denoting the initial value of *m* as m_0, the loop invariant before the predicate is that *extent*(m_0, *start(m))* is all blanks.

Suppose *m* refers to a word, that is, consecutive non-blank characters with adjacent blanks on both ends. To find the next word we must first skip the blanks following *m* and then build a subseq referring to everything prior to the next blank. In the Ness implementation of the algebra, this is written as:

---

[2] In the Ness and cT implementations of the algebra, the declarator is "marker" instead of "subseq."

```
function nextword(m)
        while next(m) = " " do m := next(m) end while
        m := next(m)            -- first letter of word
        while next(m) /= "" and next(m) /= " " do
                -- another non-blank: include it in m
                m := extent(m, next(m))
        end while
        return m
end function
```

The first *while* loop scans across all blanks after *m* and the second scans across all subsequent non-blanks to accumulate the word. The test *next(m) /= ""* checks to see if *m* ends at the end of its base, in which case it is deemed to be at the end of a word. When there is no word, *nextword* returns an empty string. For a shorter version of this function see Section 3.

Even this brief example can illustrate how values in the algebra are First Class and aid in applicative programming: the result of one function can be directly passed as an argument to another. For instance the statement

if m = "function" *then* addToTable(nextword(m), functiontable) *end if*

will pass to *addToTable* the entire word returned from *nextword*. No global variables or side effects are required; the computation within *nextword* or *addToTable* can be arbitrarily complex and the same functions can be utilized in other contexts. As sections 4 and 5 will discuss, coding this statement is more awkward with other popular models of string values.

The operators *start* and *next* are asymmetric with respect to text order in that one moves from left-to-right and the other returns the leftmost position in its argument, while the corresponding operators for the reverse direction are non-primitive. This asymmetry reflects a decision to engineer the primitives for the most common order of examining text. Indeed, in some implementations utilizing multi-byte character encoding there may be a performance penalty for right-to-left traversal.

2. Non-primitive Functions

With the primitive functions as a foundation we can write expressions for interesting substrings relative to a given substring. Commonly used functions include those identified in Figure 3 and defined in Table 1.

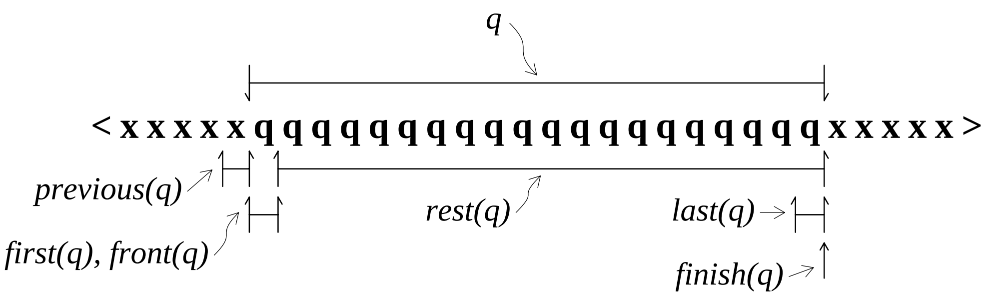


Figure 3. Non-primitive functions. For the subtle difference between *first* and *front*, see Table 1.

Function	Definition	Expression
finish(x)	the empty string at the point where *x* ends	*start(next(x))*
front(x)	the element which starts where *x* starts (even if *x* is empty)	*next(start(x))*
rest(x)	all of *x* past its first element (empty if *x* is empty)	*extent(next(front(x)), x)*
first(x)	first element of *x*, but empty if *x* is empty	*extent(x, start(rest(x)))*
last(x)	the last element in *x*, or *x* itself, if it has no elements	{see text}
previous(x)	the element preceding *x*	*last(extent(base(x), start(x)))*

Table 1. Non-Primitive Functions. The function named in the first column and defined in the second can be implemented with the expression given in the third.

Finish is analogous to *start* and also produces an empty subseq, but at the other end of its argument. Functions *front*, *first*, *last*, and *previous* all produce subseqs for single elements analogously with *next*. *Rest(x)* returns a subseq one element shorter than *x*.

Figure 4 illustrates further the non-primitive functions of Table 1. Here variable *m* has the same properties as *q* in Figure 3 and variables *s* and *p* show the results for empty and one element values, respectively. *First* and *front* differ in their values only for the empty subseq *s*; in this case, *first(s)* is *s* itself and *front(s)* is the element *next(s)*. Note that for *s* and *p* the functions *first* and *last* both return their arguments as their values.

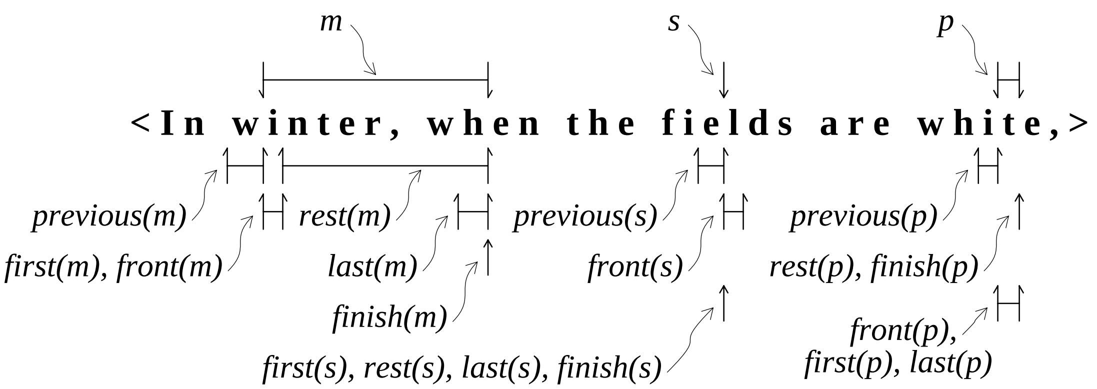


Figure 4. Examples of non-primitive functions.

The expressions for *rest* and *first* exploit the definition of *extent*. When *x* has one or more elements, the value of *next(front(x))* begins just after the first element, so the expression for *rest* produces a value extending from just after that first element to the end of *x*. When *x* is empty, its end precedes the start of *next(front(x))*, so the result for *rest* is the empty subsequence at the end of *x*, which is the same value as *x* itself, as per the definition of *rest*. The same trick applies in the definition of *first*, which gives the single element preceding *rest(x)* if *x* is non-empty, but otherwise *x* itself.

One way to implement *last(x)* is with a loop examining each of the subseqs

> *start(x)*,
> *next(start(x))*,
> *next(next(start(x)))*,
> ...

looking for one, say *t*, for which *extent(finish(t), finish(x))* yields the empty string. Then *t* is the desired last element. A more elegant recursive definition relies on *rest*:

> *function* **last**(x)
>   *if* rest(x) = "" *then return* x
>   *else return* last(rest(x)) *end if*
> *end function*

The *then* case applies when *x* is initially empty or when the recursion has descended to the last element; otherwise the *else* case recurs to compute the last element in the rest of *x*, which will be the same as the last element of *x*. In practice, of course, *last* is implemented about the same way as *next*.

For more insight into the algebra, the reader is invited to define functions of subseq *m* which produce the second element of *m* or the subsequence from the beginning of *m* to the end of its base. It is also amusing to find the middle of *m* without doing arithmetic. With the latter as a function named *middle*, there is a simple solution to one typical formal language theory problem: the expression extent(s, start(middle(s))) = extent(middle(s), s)

has the value **True** when the two halves of *s* are identical.

3. Searching Strings

It is common in string algorithms to scan a string looking for a substring satisfying some property described with a regular expression, a context free grammar, or some more general scheme. Such advanced pattern matching is beyond the scope of this paper, but a few simple search operations will serve as valuable examples and tools for later Algorithms.

The search operations below each have two arguments, a *subject* and a *specification*. By convention in Ness, the subject both bounds the range in which the search is conducted and specifies the value returned if no satisfactory substring is found. If the subject argument is non-empty, the range is that substring; but if empty, the range extends from the location of the subject argument to the end of its base. Since a successful search always yields a non-empty substring, failure is indicated by returning an empty string, usually the one at the end of the subject argument. These conventions are related since the end of the subject argument appears in both. Although this relationship has not proven awkward in the many programs written so far, there is certainly room for argument as to exactly what conventions are best.

In the descriptions below, when the second argument--the specification--is *obj*, the match must be an exact match, character-for-character; but when it is *set*, the string is treated as a set of characters. A typical set value is "**0123456789**" for the set of all digits. These functions are illustrated in Figure 5.

search(subj, obj) - Scans the range from left to right looking for an instance of *obj* and returns a subseq referring to the first such instance encountered. If none is found, the function returns *finish(subj)*.

match(subj, obj) - If the range has *obj* as its initial elements, a subseq for those elements is returned; otherwise the function returns *finish(subj).*

span(subj, set) - Returns a subseq for *start(subj)* and all contiguous succeeding elements of the range which are characters from *set*. If *front(subj)* is not in *set*, the function returns *start(subj)*.

token(subj, set) - Returns a subseq for the leftmost contiguous subsequence in the range which is composed of characters from *set*. If none is found, the function returns *finish(subj)*.

trim(subj, set) - Returns a subseq for all of the range except for any trailing characters which are in *set*. If all elements of the range are in *set*, the value *start(subj)* is returned.

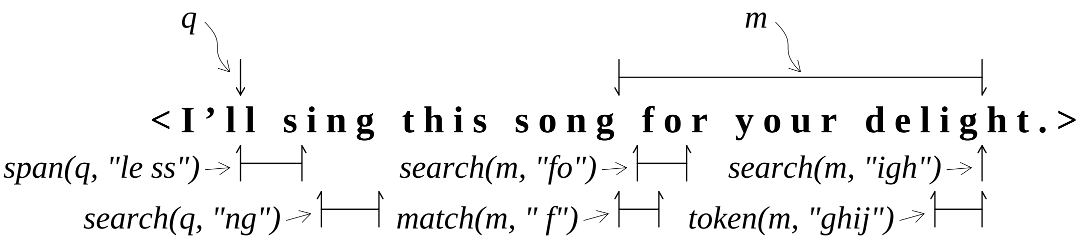


Figure 5. Examples of search functions. Note that when *m* is the search subject, the value of *token* does not extend beyond *finish(m)* and "**igh**" is not found.

Utilizing the search operations we can now rewrite *nextword* of Section 1 more briefly as

```
function nextword(m)
        m := finish(span(finish(m, " ")))
        return extent(m, start(search(m, " ")))
end function
```

Of course, if we had a variable *wordCharacters* giving a complete set of characters allowed in words, *nextword* would be simply

```
token(finish(m), wordCharacters)
```

This latter approach suffices for English or the European languages, but perhaps not for the universe of all letters of all languages.

The search functions can be defined entirely in terms of the primitive operations of the algebra. We begin with a simple support function which searches a string *src* looking for a single character *c*. If found, a subseq for it is returned, otherwise the function returns an empty subseq at the end of *subj*:

```
function findchar(src, c)
        while src /= "" and c /= first(src) do
                src := rest(src)
        end while
        return first(src)
end function
```

This function illustrates one form of loop; at each cycle around the while loop, *src* is one character shorter by virtue of the call on *rest(src)*. The loop ends if either *src* becomes empty or its first character matches *c*. If *c* should happen to be empty, this version of *findchar* will also return an empty subseq, although not without first searching all of *src*. The loop invariant before the predicate is that *c* is not equal to any character in *extent*(src_0, *start(src))*, where src_0 denotes the initial value of *src*.

Algorithm 1 expresses *span* as a function in terms of *findchar* and the primitive operations. The first *if-then* sets *s* to be the range of the search as defined by the search conventions: if *subj* is non-empty, the search is limited to its length, but if empty the search extends from the beginning of *subj* to the end of the base, as computed by *extent(s, base(s))*. The loop has the same paradigm as the one in *findchar*, calling *rest(s)* at each step to shorten *s* by one element.

```
function span(subj, set)
        marker s        -- the search range
        s := subj
        if s = "" then  s := extent(s, base(s))  end if
        while findchar(set, first(s)) /= "" do
                s := rest(s)
        end while
        return extent(subj, start(s))
end function
```

Algorithm 1. Span. Returns a subseq for all elements following *start(subj)* which are in *set*. The loop invariant before the predicate is that all elements in *extent(subj, start(s))* are characters in *set*.

Sometimes a loop advances through a string in steps longer than one character at a time, as illustrated by the *search* function in Algorithm 2. Each cycle of the *while* loop calls *findchar* to find the first character of *obj* and then calls *match* to determine if all of *obj* has been found. If so the appropriate value is returned, but if not, there is no point to re-checking *extent(s, f)*, so *s* is set to everything after *f*. If *s* becomes empty, *findchar* returns an empty subseq and the loop exits via the test of *f = ""* .

```
function search(subj, obj)
        marker s        -- the search range
        marker f        -- a location in subj of first(obj)
        marker m        -- the result of matching obj at f
        s := subj
        if s = "" then  s := extent(s, base(s))  end if
        while True do
                f := findchar(s, first(obj))
                if f = "" then return finish(subj) end if          -- fail
                m := match(start(f), obj)
                if m /= "" then return m end if                    -- succeed
                s := extent(finish(f), s)
        end while
end function
```

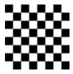
Algorithm 2. Search. Find the leftmost occurrence of *obj* in the search range. An invariant of the loop is that a substring matching *obj* does not begin in *extent(subj, start(s))*.

In the Ness implementation of the subsequence algebra, search is coded with a far faster non-linear search [Sunday, 1990]. The details of *match*, *token*, and *trim* are left as exercises to the reader.

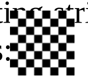
One common programming technique in Ness-- which has neither arrays nor structures-- is to store data tables in strings and look up values in them with *search*. The result of the search is a position in the table and associated data can be found at adjacent locations. Algorithm 3 utilizes this technique to scan a text and replace names of games with icons. A sample execution might convert

Go is older and more interesting than chess and checkers.

to

碁 is older and more interesting than ♜ and  .

Note that the program builds the result by concatenating strings.

subseq **GameNames** := " chess:♜ checkers: Go:碁 "

function **IconifyNames**(text)
    subseq word, icon
    subseq result := ""
    while True do
        word := token(text, letters)
        result := result ~ extent(text, start(word))
        if word = "" *then return* result *end if*
        text := extent(finish(word), text)
        icon := next(search(GameNames, " "~word~":"))
        if icon /= "" *then*
            result := result ~ icon
        else
            result := result ~ word
        end if
    end while
end function

Algorithm 3. Converting names to icons. *IconifyNames(text)* goes through the *text* and produces a copy having the names of certain games replaced with their iconic representations. Denoting the initial value of *text* as $text_0$, an invariant of the loop just before the first *if* is that *result* is a copy of *extent*($text_0$, *start(word))* with all game name replaced with icons and *extent(word, text)* is that portion of $text_0$ which has not been copied.

At each step in the loop the *token* operation gets a reference to the next word and then characters preceding it are concatenated with the result. If no word was found, *word* will be the empty subseq at the end of *text*, so the result is complete and can be returned. The call on *search* determines if the word is in the *GameNames* table and, if not, returns an empty at the end of the table so that *next* will also return an empty value. The final *if-then-else* inserts either the icon or the word into the result. Note that a space and colon are appended to *word* as the second argument of *search*; this prevents partial word matches so, for instance, **check** will not be converted to its following character, **e**.

4. Models of string values

With the subsequence algebra as one model we can now consider the models found in existing languages with respect to whether they are Simple and First Class as required in the Introduction. The discussion will show that while string values in the other models can be First Class, they do not represent substrings; no models other than the subsequence algebra offer First Class substrings.

Unitary. In the Unitary model of strings, also sometimes called the "free monoid over the character set," each string value is a distinct, atomic object. Operations and string functions return values that are effectively new strings with no relation to other existing strings. Unitary strings can certainly be First Class, given the right implementation, however, they are not always Simple to use. Problems arise for parsing and searching operations because the result of a search must report not only the string which matched, but also its position. For instance, it may be desirable to test adjacent punctuation.

Although there are no major languages with a pure Unitary model of strings, the possibility has been demonstrated by Eli [Glickstein, 1990]. In this Lisp-like language, search functions return a list of three strings which could be concatenated to recreate the original subject string. The middle element of the list satisfies the search specification and the other elements are (copies of) the preceding and following portions of the subject.

Starting at least as early as XPL [McKeeman, 1970], implementations of Unitary string values have not actually copied strings to produce new values. Instead each new substring value is produced by creating a reference to it within its base. Although this is much like the values posited in the subsequence algebra, these languages do not expose this machinery, so it is impossible to define the primitives *next*, *base*, and *extent*.

Positional. In the Positional model, a string value is a pointer to a string or an integer index to an element of a string (usually the latter is in the Unitary model). Such values can easily be First Class since integers and pointers are themselves First Class, but the Positional model is not quite so Simple. Complexity is increased if there are both positions and unitary values; it is also increased by having recourse to a domain--integers or pointers--outside the domain of strings. Simplicity suffers also because a position by itself cannot select a substring without conjoining a length or another position. These extra values entail extra variables and more assignments, thus increasing program size and decreasing its comprehensibility. Confusion is compounded by the potential for off-by one errors: does a position value refer to an element or the gap between elements? With integer positions, is the first zero or one?

The quintessential example of the Positional model with pointers is C, wherein a string value is a pointer to a string or a tail of a string. To illustrate the awkwardness of returning substrings in the positional model, consider the function *token* which finds a substring and returns a reference to it. The function must be defined with an additional argument pointing to where to store the length:

```
char *token(x, len)  char *x;  int *len;  {
        ... compute position and length ...
        *len = length;
        return position;
}
```

If *token(x, &n)* is evaluated, the substring it locates cannot be directly passed as an argument to another function as in

```
g(token(x, &n), n)              /* WRONG */
```

because the value of *n* passed to *g* would not necessarily be the value assigned by *token* to **len*. To be correct, the computation needs two steps

```
p = token(x, &n);
g(p, n);
```

Such extra steps and variables can increase the opportunity for error in a large program.

Integer positions are found in many languages, including PL/I and Fortran [ANSI, 1978]. Pointer and integer positions are not entirely equivalent. If a string is copied, integer values referring to positions in the original will refer to the same positions in both copy and original. With pointers, pointer arithmetic is needed to compute from a pointer into the original a corresponding pointer into the copy.

Overlays. There are a few languages, like Lisp, PostScript [Adobe, 1985], and even COBOL [ANSI, 1985], which permit one string to be defined as an overlay on top of another, thus making the defined string a subsequence reference. None of these languages, however, offer functions sufficient to enable implementation of *next*, *base*, or *extent*; the overlaid string behaves as though it were itself a Unitary value.

APL algorithms sometimes employ a very general form of substring; the algorithm associates a bit vector with a base string and the one bits in the vector select elements of the string to be in the substring. For instance, this expression

```
(b⩝≠\b←t='''')/t
```

removes from `t` all substrings quoted with paired apostrophes [Bernecky, 1991]. Execution begins with the subexpression `b ← t = ''''` which assigns to `b` a bit vector with ones in the positions of the apostrophes, the ≠\ converts to a vector for the characters inside the apostrophes, and finally ⩝ computes a vector indicating the positions of characters to be retained in the result, which is produced by the final `/t`. Bit vectors are a very general tool since they can select non-contiguous subsets of the characters in a string; however, to pass arguments to general functions both the bit vector and the base string have to be passed. Moreover, functions cannot return both a string and a bit vector as a single value.

Icon. To reduce program size and the need for extra variables, Icon provides a special operator, `?`, for pattern scanning. The expression

```
s ? operations
```

first makes string value s be the value of the global variable &subject and then executes the operations. The current position in the subject is given by another global variable, &pos, which initially has the value which indicates the position before the first character. String operations upto, many, find, any, match, and bal examine &subject starting at &pos and return a new position, depending on their nature. Two other functions, tab and move, change the value of &pos and return the string portion between the old and new values of &pos. Thus it is common to write statements like

```
t := tab(many(letters))
```

which advances &pos across a sequence of letters and assigns to t a copy of the string value passed over, in this case the next sequence composed entirely of characters from the set letters. Since functions like many may not find any instances of the set, the concept of failure is employed in Icon. If no values are found, the expression "fails," which initiates backtracking or, if that is not possible, terminates the scanning operation and all other operations on the stack up to a conditional statement.

Not only does Icon have the additional complexity of offering both the Unitary and Positional models, but there are separate sets of functions for each. There is a potential for confusion between tab and move, the Unitary model functions, and many, upto, and the other Positional model functions. Indeed, it would be interesting to know if omission of required tab and move operations is a common error in Icon programs.

Subsequence references incorporate both a string value and the position of that value within its base. Thus it is a First Class value for substrings suitable to return from parsing or other searching/scanning operations. It is common in programs written with the subsequence algebra to utilize a single variable both for its value and its position: In Algorithm 1, variable *s* is utilized for its first character with *first(s)*, its position with *start(s)*, and its extent with *rest(s)*. In Algorithm 2, the results of *findchar* and *match* are stored in *f* and *m* respectively. Since both are on the same base as *s*, the new value of *s* can be set to begin after *f* and the value of *m* can be returned, both retaining the position in the original base string. In Algorithm 3, the variable *word* appears both for its value and its position in its base. In all these cases more variables and assignment statements would be required with other string models.

Although internally more complex than integers, subsequence references are Simple in that they reduce the required number of concepts, even beyond the fact that the single concept obviates the need for both Positional and Unitary values. With a subsequence reference model of strings, neither character nor string values are required, as shown in the beginning of Section 1. Moreover, instead of requiring the semantics of multiple domains, subsequence references require only those described in Section 1. This can be valuable when presenting string processing to users who are uncomfortable with numbers.

5. Approaches to a string processing problem

The models of string values presented in the previous section are illustrated here with functions written in three different languages to solve a practical problem: normalization of C preprocessor statements. Some, but not all, C compilers have allowed whitespace--blanks and tabs--within preprocessor statements and some compilers allow arbitrary text after #else and #endif while others allow only comments. Thus in converting software to be more portable, a program is needed to scan for preprocessor lines and reformat them appropriately. As part of such a program, we require a function *NormalizeLine(line)* which returns a possibly modified copy of its argument. A line is modified if it has the form:

<WS> **#** <WS> <*key*> <WS> <*text*> <WS> <newline>

where <WS> is whitespace, <*key*> is one of five words, and <*text*> is arbitrary text. When <*key*> is **if**, **ifdef**, or **ifndef** the output is

<*key*> <space> <*text*> <newline>

but when <*key*> is **else** or **endif** the output when <text> is non-empty is

<*key*> <space> **/*** <space> <*text*> <space> ***/** <newline>

and when <*text*> is empty is

<*key*> <newline>

If the input line does not have one of the expected forms, it is returned unchanged.

Subsequence Algebra and Ness

In the subsequence algebra, as instantiated in Ness, the problem can be coded as in Algorithm 4. Note that *keytable* is defined in 4a with a "long constant" construct delimited by // and \\ . The text between the delimiters is the exact constant, newlines and all. If there were tabs, other control characters, and escape sequences, they too would remain untranslated. This form of constant is a valuable addition to the syntax of string languages because it allows the programmer to encode strings exactly as they are to appear. In addition to *keytable*, 4a defines *whitespace* and *letters*, which are later used as arguments to *span* and *trim*. The function *skipwhite* defines a particular scanning function suited to the problem; it could easily be written in a language with a Positional model since it effectively returns the position following any whitespace after its argument. The other function, *nextfield*, is an excellent illustration of subsequence references; its argument is a reference to a semi-colon-delimited field in *keytable* and it returns the following such field.

NormalizeLine itself, in Algorithm 4b, utilizes variable *t* as the current position in the line. The algorithm begins by skipping whitespace, checking for a "**#**", and skipping more whitespace. Next, *t* is set to <*key*> by spanning subsequent letters and <*key*> is sought in *keytable* with the result being assigned to *fix*. The last assignment to *t* gives it a value extending from the first non-whitespace character after <*key*> to the last non-whitespace character on the line. This value together with the fields following *fix* are used to build the final result value.

||| Algorithm 4 about HERE |||

The usage of *nextfield* in Algorithm 4b deserves emphasis. Since its value is an entire field, that value is suitable for concatenation to construct the result as in the final *return* statement. But since its value is a subsequence reference, it can also serve as argument to a function (in this case, itself) to locate and return the next following field, as in *nextfield(nextfield(fix))*. The reader is invited to try to code this expression in other languages.

C

The C programming language is very close to hardware level, as befits a language intended for writing a compiler and operating system. String constants are provided, but all other string operations are deferred to library functions. The usual C paradigm is to perform string operations with in-line loops. For instance, to skip across whitespace the code might be:

```
while (isspace(*t)) t++;
```

The need for these loops is reduced with functions like *strspn,* but they still occur as in the loop to trim the text.

The C version of *NormalizeLine* in Algorithm 5 begins by skipping whitespace, checking for a "#" and skipping more whitespace. The next lines set *key* and *t* to opposite ends of the key and then search for the key in *keytable*. Subsequent lines find the <*text*> by skipping whitespace after the key and then scanning backward to find the last non-space character. The last section of the algorithm builds the output by copying the prefix from the table, the text from the input, and the postfix from the table.

||| Algorithm 5 about HERE |||

It may be obvious to the reader that *textlen* can be negative, but was not so to the author. Consequently, this was the only one of the versions that had a bug beyond syntax errors. This sort of error is more common in Positional model programs because there are more variables to keep in mind while coding.

In the actual C program from which the problem was taken, *NormalizeLine* modified its argument rather than returning a copy. This requirement is crucial because without it, as in the Algorithm, the program is seriously flawed: The first three return statements return the initial value of *line*, the fourth returns a pointer to a constant, and the last returns newly allocated memory. Since the calling routine cannot effectively distinguish the storage class of the result, it cannot deallocate the allocated memory and the program will consume memory proportional to the input. This problem of managing storage is one of the chief defects of C for string processing.

Icon

Icon is a large language featuring many innovative constructs beyond those of Ness and C, so two versions of *NormalizeLine* are presented, one utilizing a subset and the other the full language. The subset is limited to approximately the control constructs and data structures which are available in Ness and is utilized in Algorithm 6. Here the function many(cset, string, i) scans string starting at position i and returns the position of the first charactrer encountered which is not in cset; if none is found, it fails. Subscript expressions of the form [i:j] select the substring extending between positions i and j. The find operator not only returns the location of the beginning of what it finds, but also succeeds or fails. Thus it is essential in the Algorithm that the first assignment to tblloc be inside the if predicate to determine whether key was found. In the next line the expression t+*key computes the position after the key as the starting point for skipping more whitespace. In constructing the result value, fieldend computes the end of fields whose beginnings must then be held in other variables; in this case tblloc and postloc.

<u>||| Algorithm 6 about HERE |||</u>

The full Icon language is exploited by Algorithm 7, which employs the string scanning operator ? described in Section 4. Each of the calls on tab or the = operator advance &pos so it serves as the start for subsequent operations. The '| &pos' clauses provide a default value for tab in case the many function fails to find any of the characters it is looking for. The \keytable[...] expression searches the keytable for the key; if found, fix is set to the record found, but otherwise keytable has the value null which is converted by the backslash operator into a failure which propagates out to the if (after some futile backtracking).

<u>||| Algorithm 7 about HERE |||</u>

Comparison

Cursory examination shows the Ness and full Icon versions of *NormalizeLine* in Algorithms 4b and 7b to be briefer than the others, to use fewer variables, and to avoid explicit recourse to arithmetic. The flow of the logic matches the number of statements, without extra assignments to save separately the two opposite ends of substrings. For more detailed comparison we can consider the algorithms to have four phases: skip hash and white space, find and process <key>, isolate <text>, and build output. The first phase is roughly comparable in all the languages since it is simply a matter of moving a position past unwanted material, but the other phases differ considerably.

In the <key> phase, the Ness version utilizes a single variable, *t*, to represent the key, whereas C and the subset Icon code each need two (*key* and *t* for C; t and *key for subset Icon). The string scanning operation hides this in the full Icon version: tab(many(&lcase)) advances &pos across the key and returns a copy of its string value.

For the <text> phase, the Ness and subset Icon versions both set a single variable to the text. The C code is more complex only because the trim function is performed explicitly with a loop. The full Icon code illustrates Icon's potential for convoluted code: the function name trim is considerably separated from its arguments, tab(0) and whitespace++'\n'; and surrounding these is an if statement semantically linked to the = and \ operators inside trim's first argument.

In the final output phase, the Ness and full Icon versions simply concatenate the appropriate values. The other two versions illustrate the deficiencies of the Positional Model by requiring several variables to keep track of the ends of the pieces to be concatenated.

The four versions of *NormalizeLine* reflect markedly different programming languages. We can say qualitatively that Ness is wordy and uses few concepts, C is low level, and Icon achieves brevity through special operator characters and a variety of data structures. These notions can be informally quantified by counting the number of operator instances, as shown in Table 2. The operators counted are listed in the body of Table 3.

		# unique operators		
	# operators	words	symbols	# concepts
Ness	49	11	4	6
C	74	12	13	12
Icon subset	58	8	9	10
Full Icon	49	7	10	14

Table 2. Number of operators and concepts in the versions of NormalizeLine. "Words" are those operators encoded with letters alone and "symbols" are those coded with non-alphabetic characters. Declarations are not counted as operations, but operations within initializations are counted.

In Table 2 we see that C uses the most operators, subset Icon fewer, and Ness and full Icon the least. Note, however, that Ness uses sixteen fewer operations to initialize keytable and nine more to extract fields from it; for this example the Icon language could benefit from constants for table initialization and Ness could benefit from some form of table construct. From the relative use of words versus symbols for operators, it is apparent that Ness de-emphasizes syntax, despite the disadvantage of longer program texts. With infix operators for the primitives and shorter function names, the Ness definition of the body of *NormalizeLine* would be about the same size as the full Icon version.

||| Table 3 about HERE |||

The real difference in complexity can be seen by counting the "concepts" required to understand each version. One way to do this is to classify each operator according to what concepts it employs, as shown in Table 3. Clearly the reader can argue with at least one of the assignments of operator to concept, but it seems clear that the Icon versions use about twice as many concepts as the Ness version. There are two ways to consider this: if language complexity is to be minimized, Icon is far more complex; but if complexity is acceptable, the subsequence algebra can be incorporated in a complex language permitting *NormalizeLine* to be expressed even more succinctly. With the algebra, the extra complexity of additional data types is not required in order to attain short programs.

6. Implementation

Before considering implementation of the subsequence algebra as part of a programming language, we must consider whether it can be provided instead as a library package. This is problematical in C, and will be challenging even in a language designed to support packages.

The biggest problem is management of string storage: to conserve space, a base sequence can and should be deleted when no subseq refers to it. In a low-level language like C which does not provide storage management, the string manager must keep track of all subseqs to manage their space and that of the base sequences. This may be impossible, however. Consider expressions where a string value returned from one function is passed as argument to another as in this fragment from Algorithm 4:

```
extent(skipwhite(t), line)
```

Skipwhite allocates a subseq and returns either it or a pointer to it. But then, which routine deallocates that memory and when? The simplest option is to never free the memory occupied with subseqs and base sequences, but this profligacy may lead to excessive paging or program termination. Subseqs and base strings constitute a non-circular structure and can be managed with references counts instead of more general and costly tools. However, if the algebra is implemented in a higher level language with storage reorganization, the more general tools are likely to be used--with consequent excess cost.

When successfully implemented as a package, the algebra must be defined in terms of some more primitive notion of strings provided in the language itself. This means two separate string mechanisms with an attendant loss in efficiency and increase in the number of concepts a program reader may encounter.

Finally, a string facility requires lexical support for string constants and a primitive operation--preferably syntactic--for concatenation. Few languages are flexible enough to be extended in these directions. Moreover, if strings are to support typographic styles and embedded objects there is the further requirement that the program development support system also support editing of programs containing such constants. Thus program editing must move closer to word processing and programs can begin to have typographic styles themselves.

Given the deficiencies of a library implementation, it is reasonable to assume that we are implementing the subsequence algebra as the native tool for strings in a high level programming language. Although this paper has claimed no interest in efficiency, it turns out that the subsequence algebra can be implemented without undue overhead. Should the algebra become widely used, appropriate compiler optimizations will ensure acceptable execution speed. With even wider use, computer hardware will adapt to the algebra as it has for floating point and programming languages, as in [Hester, 1990].

Central to an implementation are data structure definitions for subseq values and base strings. Additional data structures are needed for typographic styles and embedded objects, to whatever extent they are supported by the implementation. The Ness implementation of the algebra relies on the Andrew ToolKit (ATK) implementations of these data structures, so the language development entailed little new data structure design.

In ATK, base strings are stored as a physical sequence of characters in a space larger than the string itself. The unused space is retained as a gap within the base at the position of the most recent modification (since ATK and Ness allow modification of base sequences). If consecutive modifications tend to happen at nearby locations, the overhead of copying characters to make changes is imperceptible [Hansen, 1987].

One alternative to physically sequential storage is a list of characters. Insertions and deletions are far faster, but space requirements mushroom and performance degrades with increased paging as the list gets fragmented among many pages. Experience has demonstrated that sequential text storage reduces paging sufficiently to offset the cost of copying strings when changes are made. Of course there are numerous intermediate data structure designs with linked lists of elements each having a physically sequential block of text. We have not tried this approach because the physically sequential approach has been satisfactory.

The minimal implementation of an immutable subseq is three words: a pointer to the base, and representations of the leftmost and rightmost positions. When the elements are stored consecutively, integers suffice to indicate positions, so a subseq value can be described in C as:

```
struct subseq {
        struct basestring *b;   /* the base */
        long l, r;              /* the leftmost and rightmost positions */
};
```

A fourth word is required if the implementation chooses to implement "reference counting" by linking together all subseqs on each base. The algorithms below assume that a *struct basestring* has at least got fields *l* and *r* to indicate the two positions at its ends.

Operations in the algebra are most succinctly implemented by modifying their argument, so, in an applicative environment, the code compiled to pass a subseq as an argument copies its value to the stack. Suppose these stack elements are struct subseqs as above with *x* pointing to the first argument on the stack and *y* to the second. Then the four primitive functions can be compiled as if they were:

```
start:   x->r = x->l;

base:    x->l = x->b->l;   x->r = x->b->r;

next:    x->l = x->r;   if (x->r < x->b->r)  x->r++;

extent: if (x->b != y->b) *x = <UniqueEmptyString>;
         else {x->r = y->r;   if (x->l > y->r)  x->l = y->r;}
         <pop y from stack>;
```

In two places integer comparison determines the ordering of positions; this would have to be changed if base strings were not stored consecutively. Similarly, the ++ in *next* must be modified if elements occupy more than one byte. Conceptually, these are the only places in the implementation that require knowledge of the implementation of base sequences.

For a simple assignment like

```
m := start(s)
```

the stack is inefficient. An optimizing compiler could copy *s* into *m* and then set m->r= m->l. If we are compiling for the IBM RISC architecture, the entire statement could take as few as three instructions: a load-multiple and a store-multiple to transfer the subseq representation and an additional store to m->r. For assignment of an expression with more operators, the overhead of copying the initial value is distributed among them all and is thus proportionately lower, so the average number of instructions per primitive is low.

7. Other Advantages of the Algebra

Examples above have primarily operated on ASCII strings, but the implementation clearly permits more general sequences. These deserve discussion, but there is not room here for more than a brief mention of additional aspects of the subsequence algebra.

Invariants. In writing invariants, it is often desirable to express attributes of subsequences. As shown in various of the examples above, invariants can be succinctly expressed in the subsequence algebra.

Integer positions. While the algebra permits ignorance of a Positional model of strings, it does not preclude it. For instance, the primitives are sufficient to implement *nextn(s, n)* as the result of applying *next* a total of *n* times starting with the initial value *s*. Then the traditional function *substr(s, i, j)*--which returns the substring of *s* starting with the *i*'th character and extending for *j* characters--is

extent(nextn(start(s), i), nextn(start(s), i+j)).

By defining *substr* in terms of *next*, there is no question that the indices refer to elements rather than byte positions in an array.

Arrays. With *nextn* and a simple trick, sequences can subsume one-dimensional arrays. The trick is to address the array by a reference to the empty element at its start. Thus if *A* is an array, *nextn(A, i)* returns the *i*'th element of the array. A compiler could optimize such applications so the user could deal with arrays without the introduction of special syntax or the complexity of understanding a new data organization.

Formatted text. In addition to permitting text with arbitrarily styled segments, it is valuable to have operators to manipulate styles. These operations are convenient to define with the subsequence algebra because a style naturally applies to a subsequence of the text. In the Ness implementation, functions are provided to add styles, remove them, interrogate the style of text, and traverse the text in sections delimited by style change boundaries.

Embedded Objects. When sequences permit embedded objects the language needs operations to convert an object into an element of a sequence and to extract an object from a sequence. It is also useful to be able to interrogate an element to see if it is an object, and determine its type.

Multi-media. This term sometimes means just voice and video, but is more generally applicable to text or applications with embedded objects of all sorts. A voice recording, video clip, or other multi-media component can be incorporated in a document as a single embedded object and manipulated with the object operators of the preceding paragraph. An exciting possibility is to represent the time sequence of the voice or video object as a subseq. Arbitrary reference to and rearrangement of such sequences would then be possible under program control.

Mutability. Much literature has been devoted to applicative versus imperative programming. The subsequence algebra has been presented in a purely applicative form above, but a sequence modification operator can easily be defined (and not quite so easily implemented). *Replace(s, r)* modifies the base sequence of *s* so the portion that was referred to by *s* contains a copy of *r*. In some applications--for instance text editors--where the base may have numerous subseqs referring to its parts, such modification can be a useful tool because the other subseqs remain attached to the base whereas if a new base were constructed it would have no subseqs on it. *Replace* is also useful when it can be employed to avoid creating new strings; the overhead of copying strings is not too bad, but the overhead of allocating memory and paging-in non-adjacent strings can be large.

General sequences. In an abstract view, there is little to choose between sequences and Lisp lists. The algebra can subsume lists if subseq values can be objects embedded in sequences. Of course, with this extension simple reference counting will no longer work and garbage collection will be necessary.

Unbounded sequences. The most general sequences are potentially infinite by virtue of being defined with a function to generate successive elements. The subsequence algebra is entirely adequate to deal with such sequences because only the *next* operator need access further along in the sequence--it would call the generator if necessary. The *base* operator would return a general sequence including a reference to the generator function.

8. Conclusion

The introduction described three trends that will benefit from introduction of a new data type for substrings; and the remainder of the paper described such a data type. In it data values are references to subsequences of underlying base sequences. As a consequence, a search or parsing algorithm can return a single value which incorporates both the string that matched and its location in the base. The function caller can then examine both the string that matched and surrounding punctuation or text. The operators of the algebra are the set

{*"..."*, *~*, *next*, *start*, *base*, *extent*} ,

all of which are closed over the domain of references to subsequences. This paper introduced these operations in the first section and showed in the second and third sections that these operators are sufficient to access other relevant substrings of a base string and to define various searching functions.

Also introduced in the Introduction were four characteristics that should be found in any substring data type; it should be Simple, First Class, Unbounded, and Rich.

> That the algebra is **Simple** is discussed at length in Section 4 and demonstrated graphically in Table 3 of section 5. That table shows that programs written with the algebra can be as short as they could be with a far richer language and also that they can be understood with half as many concepts.
>
> That substring values with the algebra are **First Class** is evident from the definition; subsequence references are arguments and return values for the primitive functions; and other functions can share this property. That substring values are not First Class in other models of substring values is shown in Section 4.
>
> **Unbounded** string values require automatic storage management as shown in section 6. Any language with storage management can have unbounded strings. But with the subsequence algebra, management of the string space can be more efficient.

> **Rich** strings are supported by the algebra in that its abstraction completely hides the physical representations of strings and sequences. A substring reference is precisely the correct value to serve as an argument to setting typographic styles or to return from a search for text in a given style. Objects can be elements in base sequences, and the description of an object can be a sequence itself.

Despite their advantages, it is difficult to implement substring references as a library package in most existing programming languages, as shown in Section 6. However that section goes on to show they can be implemented quite efficiently if incorporated in the language.

Subsequence reference values raise many interesting practical issues, several of which are discussed in Section 7. Some of the most interesting are:

1. Syntax: Are there good graphic symbols for *next*, *start*, *base*, and *extent*? What language constructs best support pattern matching?

2. Semantics: Is there a better set of primitive operations? What are the best search conventions?

3. Implementation: What optimizations are possible and desirable in compiling subsequence expressions? What are the best data structures to support the algebra? Can modification of base sequences be implemented efficiently?

4. Generality: Can the subsequence algebra subsume arrays and list processing, thus reducing further the number of programming concepts? Can unbounded lists be handled satisfactorily with lazy evaluation?

While we have much to learn about subsequence references, they already can be a valuable tool in simplifying programming by professional non-programmers for desktop publishing and other applications.

Acknowledgments. Bruce Sherwood was a bountiful source of enthusiasm and encouragement; I am indebted to him, Judy Sherwood, David Andersen and others at the Center for Design of Educational Computing, Carnegie-Mellon University, who implemented and utilized the algebra as part of cT. This work began with the support of the Science and Engineering Research Council (Britain, grant number GR/D89028) and the Department of Computer Science at the University of Glasgow, under the stimulating direction of Malcolm Atkinson. Support for the Ness implementation in ATK was provided by the IBM Corporation and the Information Technology Center, Carnegie Mellon University under then director Alfred Z. Spector.

References

[Adobe, 1985] Adobe Systems, Inc., *Postscript Language: Reference Manual*, Addison-Wesley, (Reading, Mass., 1985).

[Aho, 1979] Aho, A., B. W. Kernighan, P. Weinberger, *Awk - A Pattern Scanning and Processing Language*, Bell Telephone Laboratories, Murray Hill, New Jersey, 1979.

[ANSI, 1978] ANSI, *American National Standard - Programming Language - FORTRAN*, ANSI X3.9-1978, ANSI (NY, 1978).

[ANSI, 1985] ANSI, *American National Standard for Information Systems - Programming Language - COBOL*, ANSI X3.23-1985, ANSI (NY, 1985).

[ANSI, 1990] ANSI, *American National Standard for Information Systems - Programming Language - C*, ANSI X3.159-1989, ANSI (NY, 1990).

[Atkinson, 1987] Atkinson, B., *HyperCard*, Version 1.0.1, M0556 / 690-5181-A, Apple Computer (Cupertino, CA, 1987).

[Bernecky, 1991] Bernecky, R., "Fortran 90 Arrays," *ACM SIGPLAN Not. 26*, 2 (Feb, 1991) 83-98.

[CDEC, 1989] Center for Design of Educational Computing, CMU, *The cT™ Programming Language*, Falcon Software (Wentworth, NH, 1989).

[Glickstein, 1990] R. Glickstein, "Lisp Primitives in ELI, the Embedded Lisp Interpreter", available on X-windows distribution tape as .../contrib/andrew/overhead/eli/doc/procs.doc, Information Technology Center, Carnegie-Mellon Univ., 1990.

[Griswold, 1983] Griswold, R. E. and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall (Englewood Cliffs, 1983).

[Hansen, 1987] Hansen, W. J., Data Structures in a Bit-Mapped Text-Editor, *Byte* (January, 1987), 183-190.

[Hansen, 1989a] Wilfred J. Hansen, "The Computational Power of an Algebra for Subsequences", CMU-ITC-083, Information Technology Center, Carnegie-Mellon Univ., 1989.

[Hansen, 1989b] Wilfred J. Hansen, "Ness Language Reference Manual", available on X-windows distribution tape as .../contrib/andrew/atk/ness/doc/nessman.d, Information Technology Center, Carnegie-Mellon Univ., 1989.

[Hansen, 1990] Hansen, Wilfred J., "Enhancing documents with embedded programs: How Ness extends insets in the Andrew ToolKit," *Proceedings of 1990 International Conference on Computer Languages*, March 12-15, 1990, New Orleans, IEEE Computer Society Press (Los Alamitos, CA, 1990), 23-32.

[Hester, 1990] Hester, P. D., "RISC System/6000 Hardware Background and Philosophies," *IBM RISC System/6000 Technology*, SA23-2619, IBM Corp., 1990, 2-7.

[IBM, 1965] IBM Corporation, PL/I Language Specifications, C28-6571-0, IBM Corp., Data Processing Division, (White Plains, NY, 1965).

[IBM, 1987] IBM Corporation, *Common Programming Interface Procedures Language Reference*, SC26-4358-0, IBM (Endicott, NY, 1987).

[McKeeman, 1970] McKeeman, W. M., J. J. Horning, and D. B. Wortman, *A Compiler Generator,* Prentice-Hall, Inc. (Englewood Cliffs, 1970).

[Microsoft, 1990] Microsoft Corp., *Microsoft WORD for Windows and OS/2 Technical Reference*, Microsoft Press, (Redmond, WA, 1990)

[Morris, 1986] Morris, J., M. Satyarayanan, M. H.Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. "Andrew: A distributed Personal Computing Environment," *Comm. ACM,* V. 29, 3 (March, 1986), 184-201.

[Palay, 1988] Palay, A. J., W. J. Hansen, M. Sherman, M. Wadlow, T. Neuendorffer, Z. Stern, M. Bader. and T. Peters, "The Andrew Toolkit - An Overview", *Proceedings of the USENIX Winter Conference*, Dallas, (February, 1988), 9-21.

[Shneiderman, 1980] Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers (Cambridge, MA, 1980).

[Steele, 1984] Steele, G. L., Jr., *Common Lisp: The Language*, Digital Press (Bedford, MA, 1984).

[Sunday, 1990] Sunday, D. M., "A Very Fast Substring Search Algorithm," *Comm. ACM 33*, 8 (Aug, 1990) 132-142.

[Yngve, 1963] Yngve, V. H., "COMIT," Comm. ACM 6, 10 (Mar, 1963) 83-84.

```
-- keytable:
--      Each <key> is followed by three semicolon-separated fields:
--              the output if <text> is empty, the prefix, and the postfix.
--      If <text> is non-empty, the output is
--              prefix    <text>    postfix
subseq keytable  :=
//
<if>;#if
;#if ;
;
<ifdef>;#ifdef
;#ifdef ;
;
<ifndef>;#ifndef
;#ifndef ;
;
<endif>;#endif
;#endif /* ; */
;
<else>;#else
;#else /* ; */
;
\\
subseq whitespace := " \t\b\?"          -- space, tab, backspace, del
subseq letters :=
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

-- Return the empty subseq following t and any adjacent whitespace
function skipwhite(t)
        return finish(span(finish(t), whitespace))
end function

-- Return field after f.  Fields are separated with semicolons
function nextfield(f)
        f := finish(next(f))      -- after the semicolon
        return extent(f, start(search(f, ";")))
end function
```

Algorithm 4a. Declarations for Ness version of NormalizeLine.

```
function NormalizeLine(line)
        subseq t, fix
        t := skipwhite(start(line))
        if next(t) /= "#" then return line end if
        t := skipwhite(next(t))
        t := span(t, letters)                    -- get <key>
        fix := search(keytable, "<" ~ t ~ ">")
        if fix = "" then return line end if
        t := trim(extent(skipwhite(t), line), whitespace ~ "\n")
        -- build the result from 'fix' and 't'
        if t = "" then  return nextfield(fix)  end if
        fix := nextfield(nextfield(fix)) -- skip to prefix field
        return fix ~ t ~ nextfield(fix)
end function
```

Algorithm 4b. Ness version of NormalizeLine.

```
const struct namepair {
        char *name, *notext, *pre, *post;
        int namelen, prelen, postlen;
} keytable[6] = {
        "if", "#if\n", "#if ", "\n", 2, 4, 1,
        "ifdef", "#ifdef\n", "#ifdef ", "\n", 5, 7, 1,
        "ifndef", "#ifndef\n", "#ifndef ", "\n", 6, 8, 1,
        "endif", "#endif\n", "#endif /* ", " */\n", 5, 10, 4,
        "else", "#else\n", "#else /* ", " */\n", 4, 9, 4,
        0,0,0,0,0,0,
};
const char letters[] =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
const char whitespace[] = " \t\b";
char *strnput(t, str, len)   char *t, *str;
        { strncpy(t, str, len);   return t+len; }
```

Algorithm 5a. Declarations for C version of NormalizeLine

```
	char *
NormalizeLine(line)
	char *line;
{
	char *t, *key, *text;
	int textlen;
	struct namepair *kx;

	t = line + strspn(line, whitespace);
	if (*t != '#') return line;		/* no # */
	t += strspn(t+1, whitespace);
	key = t;			/* location of key */
	t += strspn(t, letters);
	for (kx = keytable; kx->name; kx++)
		if (kx->namelen == t-key &&
				strncmp(kx->name, key, t-key) == 0)
			break;
	if ( ! kx->name) return line;	/* not a key we process */
	t += strspn(t, whitespace);
	text = t;
	t = text+strlen(text);
	while (isspace(*(t-1))) t--;	/* trim whitespace */
	textlen = t - text;		/* NOTE: textlen may be negative */

	/* now build line from		kx->pre  text...(textlen)  kx->post */
	if (textlen <= 0)   return kx->notext;
	line = (char *)malloc(textlen + kx->prelen + kx->postlen + 1);
	t = strnput(line, kx->pre, kx->prelen);
	t = strnput(t, text, textlen);
	t = strnput(t, kx->post, kx->postlen);
	*t = '\0';
	return line;
}
```

Algorithm 5b. C version of NormalizeLine.

```
global keytable
procedure fieldend(index)
        return upto(';', keytable, index)
end
procedure NormalizeLine(line)
        static whitepace, letters
        local t, key, text, tblloc, postloc
        initial {
                whitespace := ' \b\t'
                keytable := "<if>;#if\n;#if ;\n;"
                        || "<ifdef>;#ifdef\n;#ifdef ;\n;"
                        || "<ifndef>;#ifndef\n;#ifndef ;\n;"
                        || "<endif>;#endif\n;#endif /* ; */\n;"
                        || "<else>;#else\n;#else /* ; */\n;"
                letters := &lcase ++ &ucase
        }
        <<body of procedure>>
end
```

Algorithm 6a. Declarations for Icon subset version of NormalizeLine.

```
        t := many(whitespace, line, 1)
        if line[t:t+1] ~== "#" then return line
        t := many(whitespace, line, t+1)
        key := line[t:many(letters, line, t)]
        if not (tblloc := find("<"||key||">", keytable)) then
                return line
        t := many(whitespace, line, t+*key)
        text := trim(line[t:0], whitespace ++ '\n')
        # found a preprocessor line,  create new version
        tblloc := fieldend(tblloc) + 1
        if text == "" then
                return keytable[tblloc:fieldend(tblloc)]
        tblloc := fieldend(tblloc) + 1
        postloc := fieldend(tblloc) + 1
        return keytable[tblloc:postloc-1] || text ||
                keytable[postloc:fieldend(postloc)]
```

Algorithm 6b. <<Body>> of procedure for Icon subset version of NormalizeLine.

```
procedure NormalizeLine(line)
        static whitepace, keytable
        local text, fix
        record affixes(notext, pre, post)
        initial {
                whitespace := ' \b\t'
                keytable := table()
                keytable["if"] := affixes("#if\n", "#if ", "\n")
                keytable["ifdef"] := affixes("#ifdef\n", "#ifdef ", "\n")
                keytable["ifndef"] := affixes("#ifndef\n", "#ifndef ", "\n")
                keytable["endif"] := affixes("#endif\n", "#endif /* ", " */\n")
                keytable["else"] := affixes("#else\n", "#else /* ", " */\n")
        }
        << body of procedure >>
end
```

Algorithm 7a. Declarations for full Icon version of NormalizeLine.

```
        if text := trim (line ? {
                        tab(many(whitespace) | &pos)
                        = "#"
                        tab(many(whitespace) | &pos)
                        fix := \ keytable[tab(many(&lcase))]
                        tab(many(whitespace) | &pos)
                        tab(0)
                }, whitespace ++ '\n')
        then
                # found a preprocessor line,  create new version
                if text == "" then return fix.notext
                return fix.pre || text || fix.post
        else
                #fail to match, return unmodified line
                return line
```

Algorithm 7b. <<Body>> of full Icon version of NormalizeLine.

Concept	Ness	C	Subset Icon	Icon
assignment	:=	= += ++ --	:=	:=
function call	return	return	return	return
condition	= /=	!= ! <= == && strncmp isspace strspn	== ˜== not	==
if	if	if	if	if
string references	˜ next start extent finish nextfield			
search conventions	span search skipwhite trim	strncmp strspn	find many trim upto fieldend	many trim
loops		while break for		
arithmetic		+ - += ++ --	+ -	
string values			|| [	tab ||
string positions		* strspn strlen strncpy strnput strncmp	* find many trim upto fieldend	many trim tab
pointer		*		
memory		malloc		
character sets		isspace	++	++ many
record		->		affixes .
failure			if not	if many | = ==
generator				|
alternation				|
string scanning				? =
table				table [

Table 3. Classification of operators according to concept. This table shows all the pieces of syntax that were counted as operatos in Table 2. It also shows one way to classify them into concepts.