University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

A Practical Algebra for Substring Expressions

Wilfred J. Hansen

Persistent Programming
Research Report 42
June 1987

Alan Dearle.

A Practical Algebra for Substring Expressions

Wilfred J. Hansen*

Computing Science Department
University of Glasgow

May 1987

*** Author's present address:**
Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract. This paper introduces an algebra for sequences which has a number of desirable properties as a basis for string processing. In the algebra, string values are modelled as *markers*, each of which represents a selected substring of some underlying base string. Six primitive functions on markers are described: next(), start(), extent(), base(), newbase(), and replace(). Together with appropriate definitions for comparison and assignment, these are shown to be sufficient for all string processing and, moreover, conducive to the construction of readable, error-free programs. Examples are given in Ness, a small language incorporating the algebra.

1. Introduction

An algebra is a set of functions closed over a domain of values. A "practical" algebra is one suited to provide a component of a programming language, for example the algebras of integer and Boolean values. A practical algebra must be convenient, expressive, consonant with the remainder of the language, representation independent, and efficiently implementable. It should be based on a minimal core of functions which are sufficient to compute all values over which the algebra is defined. Practicality is not a formal property which can be proven; it is subjective and must be demonstrated by considering its applications.

None of the algebras for string values implemented in major programming languages has been optimally practical; all are at too low or high a level. Most assume that characters are represented by a single eight-bit byte, which does not provide a sufficient variety of characters for modern text processing. Many define a string as an array of characters, an organization which is prone to off-by-one errors with the use of integer subscripts. Others require reorganization of the entire structure of the language. In desperation we designed a new string processing facility based on the substring algebra presented below. In this algebra, string values are "markers", each of which refers to a substring of some underlying base string. Since the functions of the algebra are closed over substrings, there is no need for integer indices. Although the algebra is sufficiently general to describe algorithms over *any* sequence of objects, this paper does not explore that aspect. The algebra has been implemented in a research language called Ness, which will be used for examples [Hansen, 1987a].

The environment for this work is the Andrew system developed at Carnegie-Mellon University as a joint project with the IBM Corporation [Morris, 1986]. In conjunction with this effort, Carnegie-Mellon has been developing a version of the TUTOR language [Sherwood, 1977] used in the PLATO system [Alpert & Bitzer, 1970]. The new language is called CMU-Tutor [Sherwood and Sherwood, 1986]. This language is semi-interpretive with an operator-operands syntax similar to BASIC. The language has always provided extensive capabilities for generating graphics and analyzing student responses to questions, but has lacked string processing.

The earliest programming languages had few or no capabilities for strings. Fortran could only print strings and the syntax was the notorious "Hollerith constant" which required the programmer to count the number of characters in the string: 3HUGH.

Later versions of Fortran and other major languages like COBOL have generally treated strings as arrays of characters with all the limititations of arrays: storage must be allocated at the time of writing the program and characters must be selected with integer subscripts. One perrenial programming problem with integer subscripts is the question of whether the value refers to a character or the space between two characters; confusion over this point has led to numerous off-by-one errors.

PL/I, although it treated strings as arrays of characters, did attempt to provide some amenities. It offered concatenation, but the result had to be assigned to a variable, and all variables had a maximum size. Parameters were call-by-reference, but if a string variable was passed to a function, there was no way of determining its maximum size. There was provision for selecting a substring, but the position and length of the substring had to be given as integers. A substring could be replaced, but only with another piece of text of exactly the same size.

The XPL environment [McKeeman, 1970] may be the closest antecedent to the algebra presented here. String variables in that language refer to a substring of some other string. Each string expression produces another such substring value. The only string operations, however, are the integer based operations of INDEX and SUBSTR inherited from PL/I. Substrings could not be replaced; a new string value had to be created with concatenation. An unfortunate feature was that the implementation restricted strings to a maximum length of 255 characters, which is far too small for many applications.

Even PostScript [Adobe, 1985], which has been designed specifically for complex typography, treats strings as arrays of characters which are accessed with integer subscripts. styles are not an attribute of a string but are described by the program containing the string. Strings are not variable-length encoded, boundless, or elastic. They are delimitable in that the search operator divides a string into three parts, but there is no other means of delimitation.

SNOBOL [Farber, 1964] provided a language entirely oriented to string processing. Strings are so dominant that a portion of every statement is devoted to a pattern to be matched against a string. The mechanism for flow-of-control is supplanted by goto's based on "failure" to accomodate the fact that string pattern matching must return both the location of the match and an indication of success.* There is no way to refer to a substring so there is much copying of strings and there are even two forms of assignment which may be embedded in patterns. SNOBOL has many clever ways to shorten algorithms, but this results in a vast vocabulary that impedes learning the language. There is no set of fundamental concepts with which the language can be understood and to which the programmer has direct access.

* "Failure" is an elegant control mechanism which should be considered for any language design. I did not use it in Ness because I wanted to concentrate on the substring algebra and eliminate anything else superfluous or extraordinary.

The recent language Icon [Griswold, 1979] has resolved many of these difficulties, but strings are still indexed with integers and provisions for pattern matching have still a large influence on the entire design of the language.

None of the problems with the above languages is sufficient by itself to require a new approach, but a new problem has arisen: programs for bit-mapped workstations must process text in a far wider range of character representations. For efficiency it is still desirable to represent most characters with a single byte but some characters may require more. The complexities of text are such that they should be dealt with by system programming rather than as a part of every application program.

Examining the problems with existing languages, we can say that a string processing tool should offer:

> Convenience and Expressivity. The notation should easily express common string processing tasks and should fit comfortably within a traditional programming language. The programmer should not have to predeclare a finite length for the string and it should be possible to replace a portion of the string with a piece of a different length.
>
> Representation Independence. There should be no restriction on the size or representation of each character.
>
> Necessity and Sufficiency. The core of the notation should be a few simple functions which are sufficient for all string processing.
>
> Efficiency. Common characters should be represented in a single byte and there should not be unnecessary copying of string values.

In short, the algebra presented here did not arise as an abstract mathematical entity, but has been designed as a solution to human requirements. In that design process, when requirements conflicted the solution chosen usually was the one that best satisfied Convenience and

Expressivity.

The solution eventually reached was a notation based on the algebra presented below and implemented in an otherwise minimal language called Ness. In addition to the functions of the algebra, Ness offers marker declarations, assignment, if-then-else, while-do, predicates, compound statements, function definition, call (with call-by-value parameters), and return. Spare as it may be, Ness was sufficient to write a preprocessor from itself into C.

The substring algebra is an example of an Abstract Data Type (ADT): the actual implementation of strings is hidden from the application programmer. Much recent research has studied general mechanisms for incorporation of ADT's into programs from libraries. Unfortunately, as Section 7 will show, the substring algebra cannot readily be implemented as a library of subroutines.

Section 2 defines the most fundamental primitives, applications of which are shown in Section 3 where they are used to define other functions to access substrings and Section 4 where they are used to define pattern matching. Section 5 introduces additional primitives for modification of substring values and Section 6 uses all the primitives to show that the algebra implements universal computability. Sections 7 and 8 discuss the implementations of the Algebra and experience with it, while Section 9 describes a few open problems.

2. Substring Primitives

Consider the problem of finding words in a text. If we have one word and want to find the next, we first must find the leading character of the next word. For simplicity we asume that any non-blank character starts a word and write the algorithm in Ness as

```
-- NextNonBlank(w) assumes w is a marker for a word
--      in a base string and returns a marker for the first
--      non-blank character after the end of that word.
--      If there is no such character, the function returns
--      an empty marker at the end of the base string.
--
function NextNonBlank(w) == {
        marker c;      -- will refer to a character of the text
        c := next(w); -- start with the character after w
        while c = " " do  -- skip blanks
                c := next(c);  -- advance c to next character
        return c;      -- return the present value of c
}
```

In NextNonBlank(), *w* and *c* are *marker* values which encode not only a particular string, but also the position of that string within some larger string. Each call on next(*c*) returns a reference to the character at the position one past the substring represented by *c*; this reference is then assigned to *c*. In the Icon language, the underlying string would be a global value; in the substring algebra, it is implicit in the value of the variable. Each variable in the program may refer to a portion of a different base string.

A base string is a sequence of characters. A *character* is an understood, but undefined, primitive object. Examples include: A, a, 1, : , $, fi, α, ∞, $\rightarrow$, ☮. A character may be unrelated to others, or be part of an alphabet (say, latin letters) or an extended alphabet (say, ASCII); the decision is made when the character is entered into the system, either as part of a program or as part of a data file. It is immaterial to the algebra whether a bold letter is a different character than the corresponding plain letter, although for most applications it may be preferable to have them be the same character and to have the programming language provide other functions for manipulating the styles of strings. Such functions are beyond the scope of this paper.

To formally define the values used in this algebra, we will introduce the notion of a *substring marker value*, which represents a piece of a string. (Values will not be written this way in programs.) In the notation, a pair of angle brackets delimits a sequence of characters:

< s1 c >

The identifiers appearing within the angle brackets are meta-variables which refer to individual characters or to sequences of zero or more characters. More commonly a marker value will also contain square brackets:

Definition: A *substring marker value* (or "*marker*") is written as

< s1 [s2] s3 >

where each s_i is a sequence of zero or more characters. The portion s2 is the *marked* or delimited portion of the string. The entire string < s1 s2 s3 > is the *base* of the marked value. If s2 is of length zero, the marker is an *empty* marker.

It may appear that the definitions below require copying strings; the intended interpretation—and the implementation of Ness—does not require copying strings for any of these functions other than concatenation.

Traditional programming constructs have the following behavior with respect to substring marker values:

"*some text*" => < ["*some text*"] >

A string constant returns a marker for the entire string.

< s1 [s2] s3 > ~ < s4 [s5] s6 > => < [s2 s5] >

Concatenation is written with "~" and constructs a new string composed of the juxtaposition of the arguments, returning a marker for the entire value. (Concatentation is not necessarily primitive. A redefinition is given when the algebra is extended in Section 5.)

< s1 [s2] s3 > *relop* < s4 [s5] s6 > => s2 *relop* s5

Comparison of marker values is defined to be string comparison of the delimited portion.

print(< s1 [s2] s3 >) => s2 is printed

The printed value for a marker value is taken to be the delimited portion.

marker var1, var2, ...

Declares var1, var2, . . . to be marker variables.

f(< s1 [s2] s3 >) passes the entire substring marker value to the actual parameter.

In the examples all function arguments are markers and are not declared. Parameters are call-by-value.

The algebra of substring expressions has the following primitive operations; their arguments and values are all substring marker values.

start(< s1 [s2] s3 >) => < s1 [] s2 s3 >

Start() returns a marker both of whose limits are at the beginning of the argument.

next(< s1 [s2] c s3 >) => < s1 s2 [c] s3 >
next(< s1 [s2] >) => < s1 s2 [] >

Next() returns a marker for the character following the argument. If the argument was at the end of its base, next() returns the end of the base.

base(< s1 [s2] s3 >) => < [s1 s2 s3] >

Base() returns a marker for the entire string surrounding the argument.

extent(< s1 [s2] s3 >, < s4 [s5] s6 >)

=> < s1 [s7] s6 >, if < s1 s2 s3 > ≡ < s4 s5 s6 >
& < s2 s3 > ≡ < s7 s6 >

=> < s4 s5 [ ] s6 >, if < s1 s2 s3 > ≡ < s4 s5 s6 >
& < s1 > ≡ <s4 s5 s8 >

=> < ["ERROR"] >, otherwise

Extent(a, b) produces a marker extending from the beginning of its first argument to the end of its second argument. If the second argument ends before the start of the first, the result is an empty marker at the end of the *second* argument. The constant string "ERROR" is produced if the arguments are not equivalent. (The precise meaning of ≡ will be made clear below. For now, we remark that the implementation of extent() does *not* require testing the equality of the argument strings.)

Figure 1 illustrates the effect of start(), next(), base(), and extent().

For illustrative purposes we prove in some detail that NextNonBlank()

returns a value meeting its specification. The proof is by induction on the number of blanks following the marked portion of *w*.

Theorem: NextNonBlank(<s1 [w] b1 ... b*n* x s2 >) returns the value <s1 w b1 ... b*n* [x] s2 > and NextNonBlank(< s1 [w] b1 ... b*n* >) returns < s1 w b1 ... b*n* [] >, where w is the text of *w*, the b*i* are blanks, $n \geq 0$, x is a non-blank, and the s*i* are arbitrary sequences of characters.

Proof: The invariant of the loop just before testing the predicate is

(i) the marked portion of *c* is a single character or an empty mark at the end of the base string, and
(ii) the marked portion of extent(next(*w*), start(*c*)) is all blanks.

Part (i) is preserved throughout because c is always assigned the result of next().

The meaning of the expression in part (ii) is "all characters after *w* and before *c*". This can be verified with reference to the definition by supposing that *c* is

< s1 w b_1 ... [b*i*] ... b*n* x s2 >

With this c, the extent() expression in (ii) evaluates to

extent(next(*w*), start(*c*))
= extent(next(< s1 [w] b_1 ... b*i* ... b*n* x s2 >),
start(< s1 w b_1 ... [b*i*] ... b*n* x s2 >))
= extent(< s1 w [b_1] ... b*i* ... b*n* x s2 >,
< s1 w b_1 ... [] b*i* ... b*n* x s2 >)
= < s1 w [b_1 ...] b*i* ... b*n* x s2 >

The meaning is similar if *c* is x or if x and s2 are absent and *c* is an empty marker after b*n*.

If the invariant is preserved and the function returns c, then c must be the first non-blank after the end of w or the end of the base: part (i) guarantees that c is a single character or the end of the base, part (ii) guarantees that the only characters between w and c are blanks, and the loop predicate guarantees that c is not a blank.

It remains to verify part (ii) of the invariant. Initially, *i* is zero and the predicate is trivially true because the extent() yields an empty marker. If we enter the loop body, *c* must be one of

< s1 w b$_1$ ... [bi] ... bn x s2 > ,
< s1 w b$_1$ ... bi ... [bn] x s2 > , or
< s1 w b$_1$ ... bi ... [bn] > .

These are converted by the loop body to, respectively,

< s1 w b$_1$ ... bi [b$_{i+1}$] ... bn x s2 > ,
< s1 w b$_1$ ... bi ... bn [x] s2 > , and
< s1 w b$_1$ ... bi ... bn [] > ,

all of which again satisfy part (ii). Thus the invariant is preserved and the theorem is proved. ■

With the aid of NextNonBlank() we now define NextWord(). In Ness the convention for unsuccessful searches is that they return an empty marker at the beginning of the original argument. Our definition of NextWord() adheres to this convention:

```
-- NextWord(w) assumes w is a word in a string and returns
--      a marker for the next word. A "word" is any text
--      delimited by blanks or the end of the base string.
--      If there is no such word, the function returns start(w).
function NextWord(w) == {
        marker a;
        a := NextNonBlank(w);
        if a = "" then          -- reached end of base string
                return start(w);
        w := a;
        while a /= "" & a /= " " do   -- scan for next blank
                a := next(a);
        -- now a points to the first blank after a non-blank after w
        --      or it is an empty at the end of its base
        return extent(w, start(a));
}
```

Theorem: NextWord() correctly meets its initial comment.

Proof: The argument w to NextWord() may be initially

w = < s1 [w] sb >

where sb is a sequence of blanks. In this case, the specification of NextNonBlank gives us that

a = < s1 w sb [] >

so the if predicate is satisfied and the function returns start(w), as specified. The alternative and more interesting values for w are

< s1 [w1] sb x w2 bl s3 > or
< s1 [w1] sb x w2 >

where w2 is a sequence of non-blank characters, bl is a blank, and s3 is some sequence of characters. In this case, the specification of NextNonBlank() provides that a is

< s1 w1 sb [x] w2 bl s3 > or
< s1 w1 sb [x] w2 >

and one of these is subsequently assigned to w. By an argument similar to that for NextNonBlank(), the while loop gives a the value

< s1 w1 sb x w2 [bl] s3 > or
< s1 w1 sb x w2 [] >.

In each case, the extent() constructs a correct return value for the function:

< s1 w1 sb [x w2] bl s3 > or
< s1 w1 sb [x w2] > . ■

In order for a set of functions to be teachable and memorable, they should be based on an underlying set of functions that are as small as possible. The next section will present a number of additional substring functions all of which can be defined in terms of the primitives above. To show that the primitive set cannot be reduced we have:

Theorem(Necessity): The four primitive functions—start(), next(), base(), and extent()—are all necessary for computation with the algebra. That is, none can be expressed as a functional composition of the others.

Proof: We wish to demonstrate for each of the primitives, P, that there is no definition "function P(m) == E" which behaves as defined above for P and yet does not contain P in the expression E. We do this by exhibiting a particular value for m and argue for each primitive that no such expression E exists that converts this value appropriately, and therefore no expression E exists which implements P for all arguments. The particular value v is < "ab" ["cd"] "ef" >. Note first that

next(v) = < "abcd" ["e"] "f" >,
start(v) = < "ab" [] "cdef" >, and
base(v) = < ["abcdef"] >.

For extent() no E can exist because the value returned may have to be of any length and the other functions return only markers of zero, one, or all the characters of a base.

For base() no E can exist because none of the other functions can otherwise generate a marker beginning before the left bracket of v.

For next() no E can exist because no other function creates a marker that starts at the end of v.

For start() there can be no E whose return value is generated by base() or next() because they generate values with end brackets after the end of v. If the return value is generated by extent(), its second argument must end at the start of v, but it must ultimately have gotten this value from base() or next(), which it cannot have done.

Since none of the functions can be expressed in terms of the others, all are necessary. ■

We have experimented briefly with various alternative sets of primitive functions. For instance, the symmetry of the algebra permits previous() instead of next() or end() instead of start(); but either would emphasize right-to-left processing instead of the more natural left-to-right processing. We could replace base() with startofbase() which gives a zero-length marker at the start of the base; the present base() function could then be defined with a next() loop to find the end of the base and an extent() to build the value. Next() could return an empty marker after the following character. Experimentation showed that these and other alternatives were not as convenient for programming.

3. Substring Functions

Many substrings of the base for marker m can easily be computed from m with the substring algebra. The ones that follow are illustrated in Figure 2.

end(m) - Returns a marker for the empty substring just after m.

function end(m) == return start(next(m));

rest(m) - Returns a marker for all characters of m other than the first. If m is empty, so is rest(m).

function rest(m) == return extent(next(next(start(m))), m);

{Note that the definition of rest() exploits the clause in the definition of extent() which specifies that an empty result is at the end of its second argument.}

first(m) - Returns the first character of m. If m is empty, so is first(m).

function first(m) == return extent(m, start(rest(m)));

second(m) - Returns a marker for the character following first(m), if there is one.

function second(m) == return first(rest(m));

advance(m) - Returns a marker extending from just after the first character of m to just after next(m). In general advance(m) will have the same number of characters as m, but it may be shorter if the end of m is at the end of the base string.

function advance(m) == return extent(end(front(m)), next(m));

front(m) - Returns a marker for the first character after start(m), if there is one, otherwise m must be empty at the end of its base and this value is returned. Unlike first(), front() returns a character for all cases other than the very end of the base.

function front(m) == return next(start(m));

allprevious(m) - Returns a marker for the substring of the base of m that precedes the start of m.

function allprevious(m) == return extent(base(m), start(m));

allnext(m) - Returns a marker for the substring of the base of m that follow the end of m.

function allnext(m) == return extent(end(m), base(m));

last(m) - Returns the last character of m. If m is empty, so is last(m).

```
function last(m) == {
        if rest(m) = "" then return m;
        else return last(rest(m));
}
```

{The above implementation of *last()* would be grossly inefficient in practice. Fortunately it can be implemented by scanning backward in the base string. This can safely be done if there is a maximum length to the encoding of each character and the byte value that introduces a multi-byte encoding cannot appear within an encoding.}

previous(m) - Returns a marker for the character preceding *m*. If *m* is at the beginning of its base string, previous(*m*) returns the value start(*m*).

```
function previous(m) == last(extent(base(m), start(m)));
```

For the purposes of later theorems we prove the correctness of some of these functions:

Lemma (Rest()): The function rest(), defined as above, has the value

(i) rest(< s1 [c s2] s3 >) => < s1 c [s2] s3> and
(ii) rest(< s1 [] s3 >) => < s1 [] s3 >.

Proof: The proof of (i) has three cases depending on the lengths of s2 and s3. When s2 has at least one character we write s2 as < c2 s4 > and the proof proceeds thus:

rest(< s1 [c c2 s4] s3 >)
= extent(next(next(start(< s1 [c c2 s4] s3 >))),
  < s1 [c c2 s4] s3 >)
= extent(next(next(< s1 [] c c2 s4 s3 >)), < s1 [c c2 s4] s3 >)
= extent(next(< s1 [c] c2 s4 s3 >), < s1 [c c2 s4] s3 >)
= extent(< s1 c [c2] s4 s3 >), < s1 [c c2 s4] s3 >)
= < s1 c [c2 s4] s3 >

The second case is where s2 is empty and s3 has one or more characters. In the third case s2 and s3 are both empty. In each of these cases the marker is reduced to a empty substring at its former end. These cases can be verified by an argument similar to that of the first.

For part (ii) we observe that the first argument to extent() is next(next(start(*m*))), which cannot yield a marker starting before the beginning of *m* and the second argument is *m*, which ends at the end of *m*. Since *m* is empty, the extent() must yield a marker equivalent to *m*. ■

Lemma (Last()): The function last() defined as above has the value

(i) last(< s1 [s2 c] s3 >) => < s1 s2 [c] s3 > and
(ii) last(< s1 [] s3 >) => < s1 [] s3 >.

{That is, last(*s*) produces a marker for the last character in *s* if *s* has any characters and otherwise returns *s*.}

Proof: Part (i) when s2 is of length zero and part (ii) both follow directly from the Rest() Lemma. For part (i) where s2 is of length greater than zero we argue by induction. Suppose s2 is < c s4 >. In this case the predicate fails (by the Rest() Lemma) and so the value is

last(rest(< s1 [c s4] s3 >)
= last(< s1 c [s4] s3 >)

Since s4 is a shorter sequence than s2, the inductive hypothesis applies and the theorem is proved. ■

Lemma (Previous()): The function previous() has the value

(i) previous(< s1 c [s2] s3 >) => < s1 [c] s2 s3 > and
(ii) previous(< [s2] s3 >) => < [] s2 s3 >

Proof: By the definitions of base(), start(), extent(), and last():

previous(< s1 c [s2] s3 >)
= last(< [s1 c] s2 s3 >)
= < s1 [c] s2 s3 >

and

previous(< [s2] s3 >)
= last(< [] s2 s3 >)
= < [] s2 s3 >

In both cases the desired value is computed. ■

With the aid of the substring functions, it is possible to write an expression for any substring of a string. To demonstrate this, consider the set of *all* substrings of a string. This set consists of each instance of a substring starting at one position in the string and continuing to the same or another, later postion. Here are functions to print all the substrings of a string:

```
function printsubstrings(s) == {
        printsubsub(s, s);
        if s /= "" then
                printsubstrings(rest(s));
}
function printsubsub(t, s) == {
        print extent(s, start(t));
        if t /= "" then
```

```
                printsubsub(rest(t), s);
    }
```

It is not difficult to show that printsubstrings(s) prints all substrings of s. We begin with a lemma.

Lemma: (Tail Recursion) If P is an operation and Q is a sequence of zero or more variables each preceded by a comma, then the function f() defined by

```
    function f(x Q) == {
            P(x Q);
            if x /= "" then
                    f(rest(x) Q);
    }
```

performs P once for each *tail* of x, including the final empty substring. That is, if x is < s1 [c1 c2 . . . cn] s3 > then P is executed for each of

< s1 [c1 c2 . . . cn] s3 >,
< s1 c1 [c2 . . . cn] s3 >,
. . .,
< s1 c1 c2 . . . [cn] s3 >, and
< s1 c1 c2 . . . cn [] s3 >.

Proof: If x is < s1 [] s3 >, then n is zero and P is executed once; the then clause is not executed because x = "" is True. When $n > 0$ we argue by induction. A call to f() evaluates P once for the current value of x and then calls f recursively for rest(x). By the Rest() Lemma, rest(x) is a tail of x so n is one less and the general case holds by induction. ■

Lemma (printsubsub): The call printsubsub(s, s) prints all substrings of s that begin at start(s).

{**Proof:** By the Tail Recursion Lemma, printsubsub(s, s) executes print(extent(s, start(t))) for t being each tail of s. This is exactly the subsets of s that begin at the beginning of s. ■

Lemma (printsubstrings): The call printsubstrings(s) print all substrings of s.

Proof: By the Tail Recursion Lemma, printsubstrings(s) executes printsubsub(s, s) for s being each tail of the initial s. By the preceding lemma, this call prints the substrings beginning at each position within s, which is the entire set of substrings. ■

Theorem (Sufficiency): The substring algebra is sufficient to generate all substrings of the base of a string.

Proof: By the preceding Lemma, if s is a string, the call printsubstrings(base(s)) will print all substrings of the base of s. Since they are printed, they must have been generated. Since only the functions of the algebra have been used to operate on string values, those functions must be sufficient. ■

Note that the Sufficiency Theorem proves that all substrings can be generated, but not that any specified substring or set of substrings can be generated. This more general result will be established in Section 6.

4. Pattern Matching

Pattern matching is a fundamental operation on string values. SNOBOL, as pointed out in [Griswold, 1980], even has two distinct linguistic components: one for traditional computation and one for pattern matching. The algebra as described so far has no provision for pattern matching at all, however, it is not difficult to express pattern match primitives within the algebra. In a practical implementation, these would be hand coded for speed.

One of the difficulties in defining pattern match functions is that a pattern match needs to return two facts: whether the match succeeded and, if so, the location of the matching substring. This problem has usually been solved by introduction of *failure* as a control flow mechanism. It is not the purpose of this paper to consider control-flow architectures; failure is an elegant tool and can be used in conjunction with the substring algebra. However, failure is not necessary because marker values are already returning two facts: the two ends of a substring.

By convention, a pattern matching operation in the algebra returns the matched string for success. For failure the operation returns an *empty string at the beginning of the argument susbstring.* This convention has turned out to be the most useful of the various possibilities; it allows nested searches to have suitable behaviors.

With this convention, the Ness implementation of the algebra offers five pattern match functions. Each searches the substring extent(m, end(base(m))) for some substring matching a criterion:

search(m, target) - If successful, returns a marker surrounding the first substring of the source that is equal to *target.*

match(m, target) - Determines whether the source begins with a substring identical to *target*; if so, it returns a marker for that substring.

span(m, clist) - Returns a marker for all characters of the source from its beginning to just before the first character not in *clist*. That is, span matches the longest initial substring of the source that is composed of characters from *clist*.

anyof(m, clist) - Finds the first character in the source that is one of the characters in *clist* and returns a marker for the character.

token(m, clist) - Searches the source for the first substring consisting of characters from *clist* and returns a marker for the substring.

```
        function token(m, pat) == return span(anyof(m, pat), pat);
```

Here are the definitions of two of the pattern matching functions in terms of the primitives. The others are similar. The most fundamental pattern match is search(*m*, *pat*), which searches for an exact match of its second argument:

```
function search(m, pat) == {
        marker pf, startm, tm, tp;
        startm := start(m);
        m := front(m);
        if pat = "" then return startm;
        pat := pat ~ "";        -- copy pat
        pf := front(pat);
        while m /= "" do {
                -- search for first char of pat
                while m /= pf do {
                        m := next(m);
                        if m = "" then
                                return startm;
                }
                -- compare pat at m
                -- if (pat is at m)
                --      return pat at m;
                tm := m;
                tp := front(pat);
                while tm /= "" & tm = tp do {
                        tm := next(tm);
                        tp := next(tp);
                }
                if tp = "" then
                        return extent (m, start(tm));
                else m := next(m);
        }
```



```
                return startm;        -- failure
        }
```

In this definition, *m* is always a single character. The inner while loop advances *m* until it is equal to the first character of *pat*. When it is, subsequent characters are checked against the rest of *pat*. The outer while loop continues this process until a match is found or the end of base(*m*) is reached.

The span function clearly illustrates the utility and implementation of the convention of returning an empty string for failure of a search:

```
        function span(s, clist) == {
                marker m;
                m := front(s);
                clist := "" ~ clist;    -- copy clist to limit the search
                while search(clist, m) /= "" do
                        m := next(m);
                return extent (s, start(m));
        }
```

In this function, *m* is always a single character. It is checked by the search() call to see if it is in *clist* and if so, *m* advances. If not, the desired string extends from the start of *s* to the start of *m*. The reader is invited to study what happens if the first character of *s* is not in *clist* or if all characters of *s* are in *clist*.

The pattern functions above can easily be incorporated sequentially in a program to check for a pattern which is a sequence of items. However, it is not as clear that complex patterns can be implemented simply. To understand this problem, the author examined a collection of clever SNOBOL algorithms [Gimpel, 1976]. The measure of a complex pattern was taken to be the appearance of the operator "|" which is the most general way in which back tracking is required by a pattern. Of the 148 algorithms in the collection only 36 utilize the "|" and only an average of twice per algorithm. Moreover, about half the instances are concerned with end conditions* rather than real alternatives. In the substring algebra, end conditions are more easily dealt with because the ends of the base string can be referred to directly.

---

* Of the 73 patterns that contained "|", 30 used it only for " | REM", " | NULL", or " | RPOS(0)".

In addition to complex patterns, SNOBOL is an interpretive systemand offers many clever and unusual facilities. However the disadvantages of their complexity and diversity are many, as summed up in [Icon]:

(1) An excessively large vocabulary.
(2) Complexity of the pattern matching algorithm.
(3) Unnecessary backtracking and lack of control over the pattern-matching algorithm.
(4) Confusion between pattern construction and pattern matching.
(5) Difficulties with program structuring, especially the necessity of using side-effects.
(6) Inefficiency inherent in run-time construction of patterns.
(7) Dichotomy of languages, with a further increase in total vocabulary and a linguistic schism.
(8) Lack of a mechanism for defining matching procedures.

The solution proposed in Icon is to develop new control constructs based on failure such that pattern matching is not a facility of the language but can be expressed very cleanly. There is no conflict between the constructs developed in Icon and those here; marked substrings could easily be the basic string representation in Icon. However, with marked substrings we have found few tasks where pattern matching was an issue. It appears that it is sufficient to be able to refer to substrings of a base and access adjacent strings through functions calls.

One of the examples often cited for complex pattern matching is that of recognizing arithmetic expressions. Simple recognizers for such expressions can easily be expressed in SNOBOL. For instance the following set of SNOBOL patterns read as though they were a grammar themselves.

```
PRIM = "x" | "y" | "z"  |  "(" *EXP ")"
ADDOP = "+" | "-"
MULOP = "*" | "/"
TERM = PRIM  |  *TERM MULOP PRIM
EXP = TERM  |  *EXP ADDOP TERM
```

(The prefix asterisk operators are peculiar to SNOBOL; they defer evaluation of forward references. The grammar here is an adaptation of one in [Griswold, 1980], which does not give an Icon recognizer for the grammar.) Using the subexpression algebra, a recognizer for the grammar can be written as:

```
function acceptPrim(s) == {
        if search("xyz", front(s)) /= "" then
                return front(s);
```



```
        if front(s) = "(" then
                return extent(s, next(acceptExp(next(front(s)))));
        return start(s);
}

function acceptTerm(s) == {
        marker t;
        t := acceptPrim(s);
        while search("*/", next(t)) /= "" do
                t := acceptPrim(next(next(t)));
        return extent (s, t);
}

function acceptExp(s) == {
        marker t;
        t := acceptTerm(s);
        while search("+-", next(t)) /= "" do
                t := acceptTerm(next(next(t)));
        return extent (s, t);
}
```

Note that each function adheres to the convention of returning the string it matches or else an empty string at the beginning of its initial argument.

The acceptXxx functions are undeniably longer than the SNOBOL patterns, but they have many advantages. It is easier to see how to extend the algebraic version to handle white-space, identifiers of indefintie length, integer constants, syntax errors, and generation of code. Indeed a version that handles all these and interprets the resulting pseudo-code was coded and debugged in four hours. Traditionally it is challenging to permit a compiler to accept identifiers of any length, but this was accomplished trivially with the span() function.

5. String Modification

Functional programming has been an important recent research area emphasizing the desirability of side-effect-free computation. In such systems, values once created never change. The algebra presented so far is completely functional: values are created as constants or with concatenation and thereafter remain inviolate. Efficiency is gained because substring values need not be copies: they are references to pieces of existing strings.

Some programmers, for some applications, nonetheless desire to modify values rather than create new ones. The simplest way to accomodate this desire is to augment the algebra with an operation which appends a string to the end of an existing string. We will use the notation $\tilde{}:=$ for this operation, even though it is not an exact analog of concatenation. Since it will modify an existing string, we will represent this string by variable m with initial value $<$ s1 [s2] s3 $>$. The operation is then defined as

$$
\begin{aligned}
&\mathbf{m}\ \tilde{}:= \mathbf{<\ s4\ [\ s5\ ]\ s6\ >}\\
&\qquad \mathbf{=>\quad m := <\ s1\ [\ s2\ s5\ ]\ >,\quad s3\ is\ empty}\\
&\qquad \mathbf{=>\quad <\ [\ "ERROR"\ ]\ >,\ otherwise}
\end{aligned}
$$

> The marked value of the second operand is inserted at the end of the first operand. The variable which is the first operand is altered to refer to both the old and new text. The operation sets m to the value "ERROR" if there are any characters after the right bracket in the first operand.

With append, it is possible to efficiently produce an altered copy of an existing string. The existing string is processed from left to right and pieces are appended to the result string as they become known.

For general string modification algorithms, $\tilde{}:=$ may not satisfy some people. Therefore a third version of the algebra can be defined by introducing two additional primitives, replace() and newbase(). To keep the underlying algebra as small as possible, these new functions are then used to redefine append and concatenation.

The replace() function is defined formally by:

$$
\begin{aligned}
&\mathbf{replace(<\ s1\ [\ s2\ ]\ s3\ >,\ <\ s4\ [\ s5\ ]\ s6\ >)}\\
&\qquad \mathbf{=>\ <\ s1\ [\ s5\ ]\ s3\ >,}\\
&\qquad\qquad \mathbf{if\ <\ s1\ [\ s2\ ]\ s3\ >\ is\ not\ a\ constant}\\
&\qquad \mathbf{=>\ <\ [\ "ERROR"\ ]\ >,\quad otherwise}
\end{aligned}
$$

> Replace(m, s) modifies the string marker value m so its marked portion now contains the marked portion of string s instead of its former value. The value returned is a marker delimiting the new copy of the replacement value. All strings "$\equiv$" to $<$ s1 [s2] s3 $>$ are also modified (see below).

This function replaces one non-empty substring with another, but can also perform an insertion or deletion. For insertion, an empty marker is replaced with non-empty text; for deletion, a non-empty substring is replaced with empty text.

Since the purpose of replace() is to modify existing values, rather than

create new ones, the definition of $\equiv$ used in the definitions of extent() and replace() must indicate that the two markers are not only the same value, but they are the same instance of that value. We model this situation formally by representing the state of memory as a sequence of base strings:

$$\{<\ ...\ [\ s2\ ]\ ...\ >,\ <\ ...\ [\ s5\ ]\ ...\ >,\ ...\ \}$$

To indicate that markers share the same base, we label the brackets of each marker. Thus two markers on the same base could look like:

$$<\ s1\ [_\alpha\ s2\ [_\beta\ s3\ ]_\alpha\ s4\ ]_\beta\ s5\ >$$

where the marked value of α is $<$ s2 s3 $>$ and that of β is $<$ s3 s4 $>$. We require that the labels on all pairs of brackets be unique, so a label identifies both a base and a marked substring within that base. Two base strings have the relation base(α) $\equiv$ base(β) when both α and β are labels for markers within the same $<\ldots>$ pair. The value returned by replace(α, β) is a marker label for the copy of the marked portion of β that has replaced the former marked portion of α.

Replace(α, β) must affect any marker formerly on the base of α and ending after the start of α. Any marker bracket $[_\gamma$ or $]_\gamma$ that formerly appeared after $]_\alpha$ must remain between the characters it originally separated. Brackets formerly within $[_\alpha \ldots]_\alpha$ and brackets formerly at the same place as one of these may have to be adjusted. The simplest rule for this adjustment can be described as inserting the new text just before all brackets at the position of the $]\alpha$ bracket and then deleting the text formerly labelled by α. Formally, this is described by the rules below, where rule (1) simply makes a copy, ω, of the replacement text in case α and β are on the same base. Then rules (2) and (3) move the brackets, perform the replacement, and discard ω. In these rules ω is a unique marker label, *si* is a sequence of zero or more characters, c is a character, *bi* is a sequence of zero or more brackets, and *sbi* is a mixed sequence of characters and brackets whose brackets taken separately are denoted by *bi* and whose characters taken separately are denoted by *si*.

replace(α, β):

$$
\begin{aligned}
(1)\quad &\{\ldots <\ \mathbf{sb1}\ [_\alpha\ \mathbf{sb3}\ ]_\alpha\ \mathbf{sb4}\ >\\
&\qquad \ldots <\ \mathbf{sb5}\ [_\beta\ \mathbf{sb6}\ ]_\beta\ \mathbf{sb7}\ > \ldots\},\\
&\ =>\{\ldots <\ \mathbf{sb1}\ [_\alpha\ \mathbf{sb3}\ ]_\alpha\ \mathbf{sb4}\ >\\
&\qquad \ldots <\ \mathbf{sb5}\ [_\beta\ \mathbf{sb6}\ ]_\beta\ \mathbf{sb7}\ > \ldots <\ [_\omega\ \mathbf{s6}\ ]_\omega\ >\}
\end{aligned}
$$

$$
\begin{aligned}
(2)\quad &\{\ldots <\ \mathbf{sb1}\ [_\alpha\ \mathbf{sb2}\ \mathbf{c}\ \mathbf{b3}\ ]_\alpha\ \mathbf{sb4}\ > \ldots <\ [_\omega\ \mathbf{s6}\ ]_\omega\ >\}\\
&\ =>\{\ldots <\ \mathbf{sb1}\ [_\alpha\ \mathbf{b2}\ \mathbf{s6}\ \mathbf{b3}\ ]_\alpha\ \mathbf{sb4}\ > \ldots\}
\end{aligned}
$$

(3a) $\{\ldots < \mathrm{sb1}\ \ \mathrm{c}\ \ \mathrm{b2}\ \ [_\alpha\ \ \mathrm{b3}\ \ ]_\alpha\ \ \mathrm{sb4} > \ldots < [_\omega\ \ \mathrm{s6}\ \ ]_\omega > \}$
$\Rightarrow \{\ldots < \mathrm{sb1}\ \ \mathrm{c}\ \ [_\alpha\ \ \mathrm{s6}\ \ \mathrm{b2}\ \ \mathrm{b3}\ \ ]_\alpha\ \mathrm{sb4} > \ldots \}$

(3b) $\{\ldots < \mathrm{b2}\ \ [_\alpha\ \ \mathrm{b3}\ \ ]_\alpha\ \ \mathrm{sb4} > \ldots < [_\omega\ \ \mathrm{s6}\ \ ]_\omega > \}$
$\Rightarrow \{\ldots < [_\alpha\ \ \mathrm{s6}\ \ \mathrm{b2}\ \ \mathrm{b3}\ \ ]_\alpha\ \mathrm{sb4} > \ldots \}$

These rules are illustrated in Figure 3. However, to create robust algorithms, it is probably best to assume that these rules are unknown and write the algorithm so it will work no matter what the rules are.

As an example of replace(), here is a procedure to expand the tabs in string *m* by replacing them with enough spaces so the text after a tab is moved to the next available position among 9, 17, 25, 33, . . . :

```
function ReplaceTabs(m) == {
        marker tab, eight;
        eight := "        ";  -- 8 spaces
        tab := eight;          -- initial distance to tab
        while m /= "" do {
                if first(m) = "\t" then {
                        -- replace tab with spaces
                        replace (first(m), tab);
                        tab := eight;
                }
                else if tab = "" then
                        -- non-tab at tab stop, set for next tab stop
                        tab := eight;
                else
                        -- non-tab: shorten distance to tab stop
                        tab := rest(tab);
                m := rest(m);
        }
}
```

The replace() changes a tab character into just enough spaces to fill to the next tab stop. Note the absence of arithmetic for determining the current output position. As an exercise the reader is invited to write a version of ReplaceTabs() that keeps track of tab position with integers.

Observe that the formal parameter *m* in ReplaceTabs() is modified in two distinct fashions: it is advanced thorugh the base string and the base string is modified with replace(). Since parameters in Ness are call-by-value, only the second of these will affect variables in a routine that calls ReplaceTabs().

In an implementation of the substring algebra, it is desirable that program constants not be modified. If an attempt is made to replace a portion of a constant string, the program is aborted. It may be desirable to create an empty string into which text can be inserted with replace(). For this we provide a primitive read-write constant:

newbase() => < [] >

Creates a new, modifiable and empty, base string.

With newbase() we can now redefine concatentation and introduce an append operation:

s ~ t == base(replace(end(replace(newbase(), s)), t)).

The result contains the concatentation of the marked segments from *s* and *t*.

v ~:= t == v := extent(v, replace(end(v), t)) ,
(where v is a variable).

The value of variable *v* is appended with *t* and *v* is given the entire result as its new value.

With these definitions, one common coding sequence can be

```
t := newbase();
while . . . do {
        . . .
        t ~:= expression;
        . . .
}
```

The variable *t* is initialized to a modifiable empty string and then text is appended to it within the loop.

6. Completeness

One way to show that the substring algebra is "complete" is to show it provides universal computability as defined by a Turing machine:

Theorem (Universal Computability): The substring algebra is sufficient to simulate a Turing machine having a tape with symbols of "0" and "1", a read head which is examining one symbol of the tape, a state machine based on the symbol under the read head, and five operations: move head left, move head right, write a "0", write a "1", and halt.

Proof Outline: The non-empty portion of the tape is represented by a substring marker value with a single character marked; this is the current position of the read head. Suppose that in state S a "0" under the read head will cause operation P and transitition to state U while a "1" under the read head will cause operation Q and a transition to state V. Then state machine implementation is a collection of procedures, one for each state, having the form

```
function S(t) == {
        if t = "0" then { P;  U(t); }
        else { Q;  V(t); }
}
```

The five operations are defined as converting the initial tape t into a new tape value:

```
move right:  if next(t) = "" then
                   t := last(base(t) ~ " ");
             else t := next(t);

move left:   if previous(t) = "" then
                   t := next(start(" " ~ t));
             else t := previous(t);

write a "0": t := replace(t, "0");

write a "1": t := replace(t, "1");

halt:        print(t); exit();
```

These operations are incorporated in place of P and Q in the procedures implementing the finite state machine. The functions do not terminate. If a halt is reached it prints the tape value and exits.

All the substring functions utilized in the state machine are either primitive or have been proven to have the appropriate behavior. Therefore the collection of procedures in the substring algebra implements the Turing machine, so all functions are computable. ■

Note that this version of the Turing machine simulation is not functional; it utilizes the replace() function. A purely functional simulation is not difficult and may amuse the reader; however, it would add bulk without enlightenment to this paper.

The Universal Computability Theorem does demonstrate the

computational power of the substring algebra, but contributes little to an understanding of its practicality. It is preferable to observe that the computable functions include all substring functions and that the Sufficiency Theorem shows that all substrings can be generated. We thus have the

Theorem (Completeness): The substring algebra can compute any function over the set of all substrings of a base. ■

7. Implementation

To demonstrate the practicality of computing with the substring algebra, here is a quick sketch of an implementation.

Each separate base string is represented by a control block which contains pointers to the ends of the stored base text and a pointer to a list of all markers on that base. The list of markers is used to find markers that must be modified for replace(); moreover, when the list is empty the base itself can be discarded.

Each marker is a compound of four values:

- a reference to the beginning of the marked substring,
- a reference to the end of the marked substring,
- a pointer to the control block for the base, and
- a pointer to the next marker in the list from that base.

The definitions of next(), base(), and start() in this storage structure are quite trivial, requiring only copying the argument marker and revising one or both references to the string ends. The extent operation is slightly more complex only because it must check for several conditions.

Although the substring algebra is an Abstract Data Type, it poses a severe test of mechanisms for incorporating ADT's into programs from a library. The difficulty is that the lists of marker values must be maintained so every assignment and procedure call must have special handling for markers. In particular, if a temporary marker value has been passed as a parameter to a procedure, it must be discarded when the procedure exits. A real compiler can directly compile the appropriate code because it has complete control over the stack and heap.

When implementing the algebra as a subroutine package, the most convenient approach is to implement a separate stack for marker values. All marker functions are then parameterless procedures which take their operands from the stack. This makes the simplest marker functions even simpler because they need only modify the top marker on the stack.

In a paging environment, the best organization of the actual string texts is as a single long string of characters as described in [Hansen, 1987c]. To reduce the amount of character movement for replace(), gaps are allowed within the strings.

These techniques have been incorporated in a subroutine package for C [Hansen, 1987b]. The compiler for Ness converts its input into an equivalent C program which utilizes the subroutine package.

To illustrate various aspects of the implementation, here is the function from the Ness compiler which converts a function call into an appropriate C function call.

```
-- DoCall(id, p, v) processes a call to a function whose name
--      is in the source text at the location given by id, whose
--      left parenthesis is at p, and using local variables given by v.
--      Returns the character immediately after the closing ")".
--      Converts   " func ( arg1 , . . . argn ) "
--      into   " ( arg1* , . . . argn* , func* () ) " ,
--      where the *'s indicate translation.
--      If func is not predefined, func* is func;  otherwise
--      it is the value found in FuncNames, where the
--      form of each entry is
--              ,predefinedname(),replacementname()!
--
function DoCall(id, p, v) == {
        marker t, theCall;
        t := "," ~ id ~ "()";
        -- see if the function name is pre-defined
        theCall := search(FuncNames, t);
        if theCall = "" then
                theCall := t;
        else    -- extract the replacement name from the table
                theCall := extent(end(theCall),
                        start(search(end(theCall), "!")));
        while p /= ")" do
                -- p is "," or "("
                -- preprocess an argument expression
                p := DoExpr(Deblank(end(p)), v);
        -- now p is the ")"
        -- remove the function name from the program text
        replace(id, EMPTY);
        -- insert the function call before ")"
        return next(next(replace(start(p), theCall)));
}
```

In practice, the while loop contains an additional test for *p*=";", which

indicates that a right parenthesis has been omitted. This is fixed by inserting a right parenthesis and printing an error message.

For strings of ASCII text, execution with the present implementation of the algebra is not as fast as with routines coded directly in C. The Ness preprocessor (567 lines) processes itself in under 35 seconds on a Sun 2/140. About a third of this time is taken because each marker variable is a pointer to a marker value rather than being a marker value itself. About ten percent of the time is occupied in moving strings around, which can be reduced by introducing an algroithm to apportion gap space among multiple gaps. Another ten percent of the time can be saved by recoding anyof() and span() to test bit vectors rather than call search() for every character.

8. Experience

A number of working programs have been implemented utilizing the substring algebra in Ness and CMU-Tutor. No major difficulties have been encountered and users have been universally enthusiastic about the ease of programming. Only time will tell whether this ease will carry over to a simplicity of maintenance.

The largest program is the Ness compiler of which a sample was given in the previous Section. This program modifies the input file in memory to produce a corresponding C file which is output as a whole after the translation. To illustrate the ease of programming, the entire program had only three bugs. Modification and extension have continued with no new bugs being introduced.

Another large program was written to study the feasibility of more general pattern matching in Ness. The program parses pattern expressions, converts them to a tree form, and interprets them by using them to find substrings in a text. (The tree form is generated with embedded C code.) The grammar implemented includes exact match, star for multiple instances, match any of a set of characters, and alternation of patterns.

Other programs written in Ness include an arithmetic expression parser and interpreter, conversions in both directions between roman numerals and decimal, permutations of characters and words, and counting the words in a file. Programs written in CMU-Tutor include

Reverse a List
Alphabetize a List
Count Vowels in a Text
Translate to Pig-Latin

Plot Two User Functions Simultaneously
Plot Parametric Equations

The last two of these exploit the CMU-Tutor run-time capability for compiling and executing mathematical statements. Thus the functions to be processed are typed in by the user, modified with the substring algebra, compiled, and plotted.

9. Extensions and Open Questions

The substring algebra has so far been shown necessary and sufficient in a narrow sense, but many additional facilities can be proposed which are not easy to define with the primitives:

inalphabet(< s1 [c s2] s3 >) => < a1 [c] a2 >,
where < a1 c a2 > is one of the known alphabets.

> inalphabet(*m*) returns a marker for a copy of the first letter of *m*, but a copy within a string which is the entire alphabet for that character.

With inalphabet() as a primitive, all functions over alphabets can be computed. To give the "name" of an alphabet within a program it is sufficient to give a string containing a character from that alphabet. (This assumes that the editor used to create programs is capable of typing characters in any alphabet.)

bound(< s1 [s2] s3 >) => < [s2] >.

> Bound(*m*) appears to have the same value as *m*~ "", but the implementation of bound() is such that the value is not copied and remains a piece of its original base string.

Bound() provides a means to restrict a pattern match to a substring of a base.

readonly(< s1 [s2] s3 >) => < s1 [s2] s3 >.

> The return value is a marker on the same base as the original argument, but marked as read only so it has the same behavior as a constant.

Readonly() permits passing a substring to a function and guaranteeing that the function does not modify the substring or its base.

Other functions have been defined to manipulate the typographic styles

of strings, but these are beyond the scope of this paper.

An open question is whether a flag should be associated with each marker that would be set when the marked text was modified with a replace(). No use for this facility has occurred in examples to date, but it has been used within a text editor.

Another open question is whether there should be some syntax for the substring functions other than named functions. Perhaps ^, *, +, and - for start(), base(), next(), and extent()? Considerable experimentation is needed before we are ready for this step. In the time being we must answer the question of whether the function names used above are acceptable. Should they be delimited in some way to distinguish them from other, client-defined functions?

The definition of replace() specifies a particular behavior for other markers on the same base as the destination. This specification has some nice properties, such as a simple formal description, a simple textual description, and a simple implementation. In none of the examples considered so far has any other definition been found to be essential, but it is not known if this is the case for all applications.

On a practical level, it is a common error to assign a constant to a variable and then append values to the variable with ~:=. This fails because ~:= fails when replace()ing the end of the constant. Since the program code looks innocuous, some means should be found to make it work.

10. Summary

We began with the notion of a "practical" algebra as one on which to base a programming language data type and then discussed the problems of existing notations for strings (none of which is a one-sorted algebra). A new algebra was then introduced in Sections 2 and 5, based on the primitive operations of next() start(), base(), extent(), newbase(), and replace(). A number of theorems were proven demonstrating the power of the algebra and a number of algorithms were exhibited written in Ness, a research language incorporating the algebra.

There are certainly some disadvantages to the algebra. String manipulations will be slower than with unembellished C and systems programmer will not have detailed control over the character representations of data.

Numerous advantages outweigh these disadvantages, however:

Convenience and Expressivity. The best measure of the felicity of this algebra for programming is the enthusiasm of the programmers who have used it for substantial tasks. They like the facts that string sizes need not be estimated, poritions of strings can be replaced, and integers are not used as subscripts. They found the pattern matching functions sufficient without resorting to non-standard flow-of-control constructions.

Representation Independence. The algebra is suitable no matter whether characters are encoded with one, two, or more bytes. These details are hidden from the application programmer. Indeed, the algebra is not specific to character strings and can be used with any sequence of objects.

Necessity and Sufficiency. Theorems above have demonstrated that the primitive functions are Necessary and Sufficient for substring computations, and moreover, the algebra provides Universal Computability, so *any* function can be computed with substrings alone. Of more practical importance is that the primitives are a small set which can be easily taught and thoroughly understood.

Efficiency. The algebra contributes to efficiency because strings do not have to be copied for substring computations and because the representation independence prevents the doubling in size of all character strings if the domain is to include more character values than can be encoded with a single byte. The algebra also contributes to efficiency in reduced programming and debugging time.

The algebra described above was originally motivated by the desire to deal with modern string values, possibly containing unusual characters or typography. After using it for some time, however, it seems to meet this goal and go far beyond, providing an excellent tool for general string processing.

Acknowledgements. Bruce Sherwood has been an unending source of enthusiasm and encouragement as well as the one who wanted the algebra as a tool for the CMU-Tutor system. I am indebted to him, Judy Sherwood, David Anderson and others at the Center for Design of Educational Computing, Carnegie-Mellon University, who have been implementing and exploring the algebra. This work would not have been possible without the generous support of the Department of Computer Science at the University of Glasgow, and the Science and Engineering Research Council (grant number GR/D89028), both under the energetic and stimulating direction of Malcolm Atkinson. The work

has benefitted from conversations with Kieran Clenaghan, David Harper, Joe Morris, and John Launchberry.

References

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

[Alpert & Bitzer, 1970] Alpert, D., & D. L. Bitzer, *Science* 167, 1582 (1970).

[Farber, 1964] Farber, D. J., R. E. Griswold, I. P. Polonsky, "Snobol, a String Manipulation Language," *J. ACM 11*, 1 (1964) pp. 21-30.

[Gimpel, 1976] Gimpel, J. F., *Algorithms in SNOBOL4*, John Wiley & Sons (New York, 1976).

[Griswold, 1979] Griswold, R. E., D. R. Hanson, and J. T. Korb, "The Icon Prgoramming Language: An Overview," *SIGPLAN Notices 14*, 4 (April, 1979) 18-31.

[Griswold, 1980] Griswold, R. E., and D. R. Hanson, "An Alternative to the Use of Patterns in String Processing," *ACM TOPLAS*, 2, 2 (April, 1980) 153-172.

[Hansen, 1987a] Hansen, W. J., *Ness - Reference Manual*, Computer Science Dept., Univ. of Glasgow, 1987.

[Hansen, 1987b] Hansen, W. J., *Em - Reference Manual*, Computer Science Dept., Univ. of Glasgow, 1987.

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

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

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

[Sherwood, 1977] Sherwood, B. A. *The TUTOR Language*, Control Data Education Co. (Minneapolis, 1977).

[Sherwood, 1986a] Sherwood, B. A., and J, N, Sherwood, *The CMU-Tutor Language, Preliminary Edition*, Stipes Publishing Company (10 Chester Street, Champaign, Ill., 1986).

[Sherwood, 1986b] Sherwood, J. N. *CMU Tutor Reference Manual.* Center for Design of Educational Computing, Carnegie-Mellon University (Pittsburgh, 1986). (This is a printed version of the on-line reference manual.)

(a)

(b)

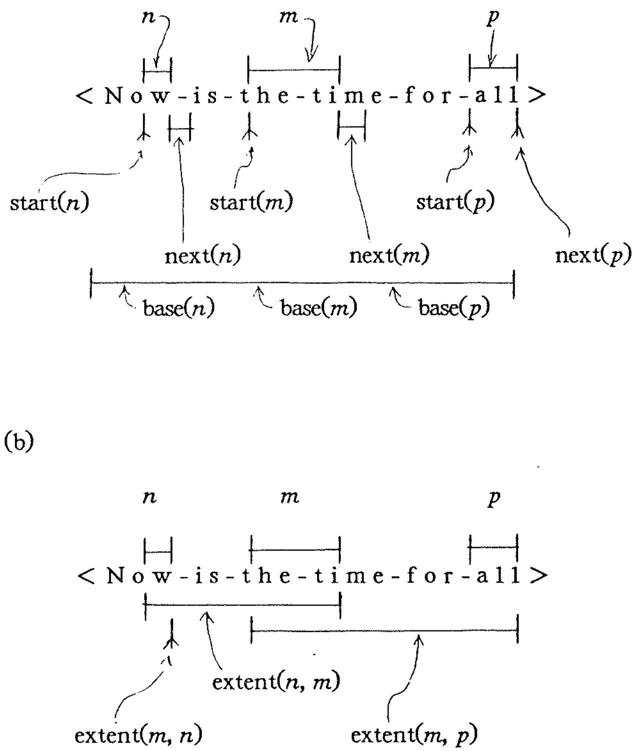

Figure 1. The four primitive functions. The base string for all examples is the non-blank characters between < and >. The marked portion of non-empty markers is shown as an elongated H and that of empty markers as a headless up-arrow.

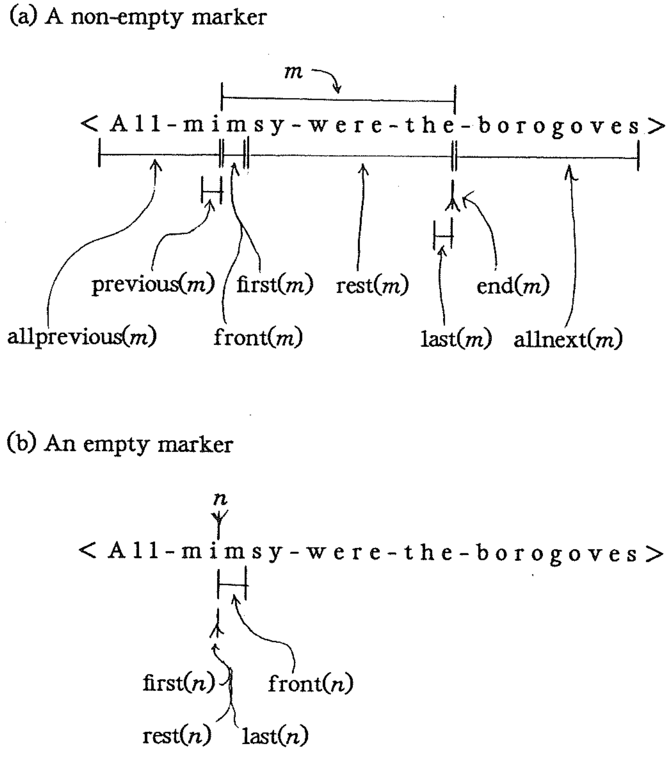

Figure 2. Some derived substring functions.

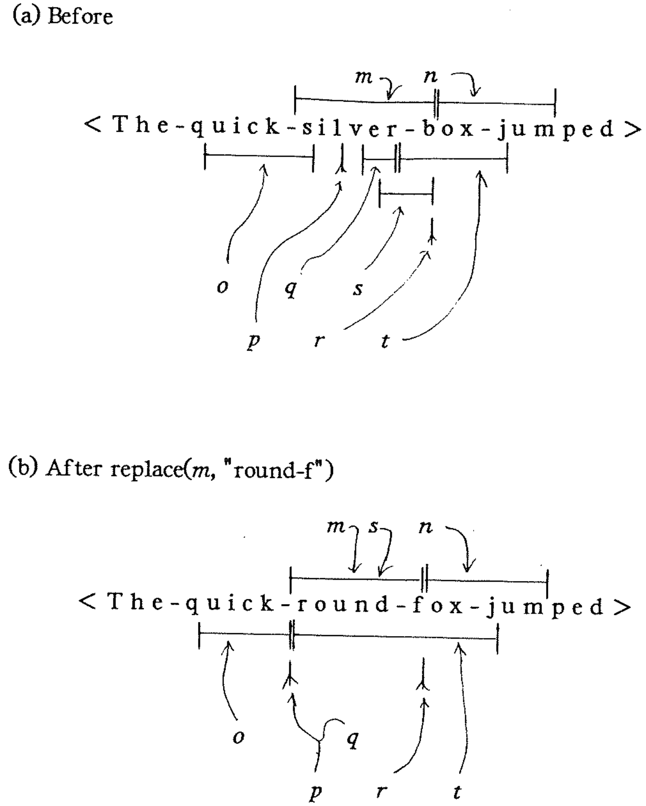

Figure 3. The effect of a replace() on other markers.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 (March 1987)

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 9th July 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-87	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00

PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00