LUCID PROGRAMMING[†]

by

E.A. Ashcroft*

Research Report CS-77-03

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

* Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

† This paper was presented at the International Symposium on Methodologies for the Design and Construction of Software and Hardware Systems, Rio de Janeiro, June 28-July 2, 1976. The reader should be warned that some of the notation has subsequently been changed, in particular in section 4. The 'modern' notation is given in "Lucid Scope Structures and Defined Functions", by E.A. Ashcroft and W.W. Wadge, CS-76-22.

LUCID PROGRAMMING

E.A. Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada

Abstract

The title of this paper is a deliberate double entendre, because Lucid is a programming language. However, the pun is not too outrageous, since, as claimed in this paper, Lucid programming is lucid programming.

Lucid is a very unconventional programming language, and this paper attempts to give a "feel" for the language and justify, a posteriori, the programming style it imposes.

1. Introduction

This paper is aimed at the casual reader. More complete summaries of Lucid can be found in [2,3]. The formal definition of Lucid is given in [1].

2. Elementary Lucid

The primary aim in Lucid is that the statements in a Lucid program should be mathematical assertions about values computed by the program, and these assertions should form the basic premises from which properties of the program can be deduced by more or less conventional mathematical reasoning. Elementary Lucid programs should essentially be sets of axioms. This implies that the order of statements in an elementary program should be irrelevant.

Achieving this aim is made especially difficult by the almost equally important requirement that the programmer should be able to write programs in a reasonably conventional manner, using assignment statements, tests and iteration. (Recursion in programming languages has been around for a long time, and is very useful in certain elegant algorithms, but, as an everyday programming method, it is generally neglected, even avoided. Certain languages with recursion, like LISP, have devoted adherents, but it can be argued that this is because recursion is particularly useful when computing with list structures. Recursion also bestows on the language a certain mathematical elegance, and in fact 'pure' LISP programs do achieve the basic aim mentioned above.) Assignment and iteration are at the basis of the great majority of successful programming languages, and, whether by nature or nurture, are considered by programmers to provide the most natural way of expressing algorithms.

Lucid throws these two requirements into the greatest conflict, by attempting to treat assignment statements as equations. Surprisingly, Lucid solves the seemingly insurmountable problem that this causes.

The problem clearly arises on reassignment to variables; for example, X = 3 and X = 2 can not both be true. Reassignment is inextricably linked with looping; with no loops in a program there need be no reassignment (new variables can be introduced), and a loop requires reassignment if a situation is to be reached where the loop will terminate. The reassignment problem can be solved if we make this correspondence between reassignments and loops quite rigorous.

A variable X that is changing within a loop, and whose value on any given iteration depends on variable values at earlier iterations (usually the previous one), will be called a loop variable. We will require that loop

variables be updated (reassigned) exactly once on each iteration of the loop. Then we can distinguish between the initial value of X, denoted first X, the current value of X, denoted simply X, and the value of X after updating for the next iteration, denoted next X. Then if, for example, X starts off at 0 and is incremented by one each time around the loop, this is expressed by the two statements first X = 0 and next X = X+1. We now make the observation that the correspondence between loops and assignments has become so close that these two statements are sufficient to indicate the **existence of the loop, and explicit control statements are unnecessary!**

Suppose we have another variable Y that is varying within the loop, which uses X. For example, Y starts off at 1, and on each iteration we add twice the (non-updated) value of X plus 3. Well, we simply write first Y = 1, next Y = Y+2X+3. How do we put the four statements together? In any order we please! The meaning of "X" in the program is the same whether it occurs before or after the statement next X = X+1. The language has what is called "referential transparency".

We still need to be able to terminate the loop. Essentially, this is the effect of the function as soon as: if we write

result = X as soon as Y > N,

the value of result is the value that X has on the iteration where Y is, for the first time, greater than N. (N itself could be a varying loop variable, but later we will use this program in a context in which N is a constant.) We still need impose no ordering on the five statements. It is clear that we can choose to write the statements in an order that suggests the usual 'flow of control', but this just helps to make the program look more conventional. In fact, the 'driving force' behind the evaluation of the program is the flow of data - the dependencies of the values of variables on the values of other variables.

We can easily write programs that are difficult to interpret in 'flow of control' terms. For example, we could add to our five statements the statement Z = Y as soon as X < 10Y. We then have two exits from the loop. In general we can think of the as soon as as determining a value to be plucked out of the loop. In the expanded program we pluck two values from the loop. (We do not stop the loop when we get the first of these values.) More difficult to interpret would be the addition to our five statements of the statement N = X as soon as Y > 100. Now the value plucked out by this statement is used to determine the value picked out by the original as soon as statement. Notice that we determine N to be 10, when X gets to 10 and Y gets to 121. But then we already have that Y > N - we should have plucked the value for result out of the loop many iterations previously! The only conventional operational interpretation is that the loop is duplicated, and, after determining N, the values of X and Y are recomputed from the beginning.

The purpose of the last paragraph is not to confuse but rather to warn the reader that the simplicity of Lucid can be deceptive. However, in future we will avoid programs with obscure operational interpretations.

The simple program we started with has a single loop. We can also write programs consisting of concatenated loops and nested loops. Such programs will be introduced later, when necessary. This paper is not intended to be a complete exposition of Lucid. More details can be found in [1,2].

3. Lucid reasoning

Since elementary Lucid programs are simply sets of equations,

it is not surprising that proofs of program properties tend to be simple and direct. We will also see that the construction of programs benefits from the freedom from 'control flow'.

Take the simple program considered in Section 2:

first X = 0

first Y = 1

next Y = Y+2X+3

next X = X+1

result = X as soon as Y>N

This is actually a program to compute the integer square roof of N. We can prove that $Y = (X+1)^2$, i.e. the current value of Y is always equal to the current value of $(X+1)^2$. We prove this as follows - by 'Lucid induction':

(i) **Base step:**

$$\begin{aligned} \text{first } Y &= 1 \\ &= (0+1)^2 \\ &= (\text{first } X+1)^2 \end{aligned}$$

(ii) Induction step:

Assume $Y = (X+1)^2$ **(the Induction Hypothesis).**

$$\begin{aligned} \text{next } Y &= Y+2X+3 \\ &= (X+1)^2+2X+3 \quad \text{(by Induction Hypothesis)} \\ &= (X+1+1)^2 \\ &= (\text{next } X+1)^2 \end{aligned}$$

$$\therefore \quad Y = (X+1)^2 \rightarrow \text{next}(Y=(X+1)^2)$$

Therefore, by induction, $Y = (X+1)^2$.

Notice that we just use the statements in the program as properties to be called upon when required. The reasoning is not complicated by having to consider at which point in the loop something is true; and the property proved, $Y = (X+1)^2$, is a property of the whole program, not something true at a particular point. And yet this whole proof is very analogous to a proof of the same property using 'inductive assertions', in which one has to be very clear about exactly where in the loop one is talking about.

As well as the induction rule there are rules that allow reasoning about the function as soon as, which essentially involve proving program termination. For details, see [1,2].

It is interesting to note that, having proved $Y = (X+1)^2$, simple substitution 'massages' the program into the following equivalent form:

first X = 0

next X = X+1

result = X as soon as $(X+1)^2 > N$

We see that the 'mathematical' nature of the program makes it very easy to transform the program without changing its meaning. Such transformations may be useful for optimization as well as for proving correctness. They also suggest the possibility of program synthesis techniques in which we start with the desired property of the program, and then massage it so that it looks more and more like a Lucid program. The fact that there is a smooth continuum from straightforward mathematics to Lucid programs makes this look like a promising technique.

Even if such a technique is not used, the construction of Lucid programs is made relatively straightforward by the lack of control flow. For example, suppose we are setting out to write an integer square root program, without using squaring. The idea may occur to us to generate

successive squares until we find one larger than N. So we first generate all the non-negative integers, first I = 0, next I = I+1, and then decide how to get their squares, represented by a variable J. Well clearly first J = 0. Now if we have the square of I, we can get the square of I+1 by adding 2I+1, so we can say next J = J+2I+1. We are now sure that J = I^2. Now we note that we are to find the first I^2 that exceeds N, and return the previous value of I, which is clearly I-1.provided N ≥ 0. (If N < 0 there will be no previous value of I, but in that case √N is imaginary and we do not expect our program to deal with it anyway.) So the complete **program is**

```
first I = 0
next I  = I+1
first J = 0
next J  = J+2I+1
result  = I-1 as soon as J>N.
```

The thing to note is that the construction was quite 'modular'. The sequence of values to be taken on by I is independent of the values of the other variables, and the complete specification of I can be made before producing program for the other variables. Assembling the program fragments is trivial - they are just put together as a set. There is no worry about where the statement that breaks the loop should be placed, or which variables should be updated before the others, etc. It is a very different style of programming, and one which has been found to be simpler and more reliable than that for conventional languages. In fact, even when programming in a conventional language, it is often easier to write the program first in Lucid, and then translate it.

4. More Lucid

Here we will consider extensions to Lucid that are more completely described in [3]. One feature of programming languages that is missing in elementary Lucid is scope of variables. An elementary Lucid program is a set of equations specifying some variables, and each variable has one specification.

Local variables can be introduced by using the 'parentheses' produce ... end. For example, in

```
first I = 1
produce next I using I
    first J = 1
    next J = J+1
    result = I+J
end
```

the variables J and result are local to the produce "clause", and are logically distinct from any variables J and result that might appear outside the clause. Variables appearing in the list following "using" are global variables. Locals like J (not result) can be consistently renamed to anything that does not conflict with any global or other local. If the local variables of a produce clause (except result) do not have the same names as any local or global variablesin the enclosing clause or program, the parentheses produce ... end can be removed, replacing the local variable result by the expression following the word produce. Thus, the above piece of program is equivalent to

```
first I = 1
first J = 1
next J  = J+1
next I  = I+J.
```

We can use a similar construct for nested loops. Suppose we wanted to sum the integer square roots of the integers 1 through 100. We write

```
first N = 1
next N  = N+1
first SUM = 0
compute next SUM,using N, SUM
   first I = 0
   next I  = I+1
   first J = 0
   next J  = J+2I+1
   result  = SUM+(I-1 as soon as J>N)
end
result = SUM as soon as N>100
```

The parentheses compute ... end delimit the inner loop, which is evaluated for all values of the global variables SUM and N. Inside the inner loop, the global variables are constant, even though they vary in the outer loop.

These parenthesising notations 'structure' the equation sets, and corresponding structuring of program proofs is necessary. We have to associate with properties the clauses within which they are valid. This gets back a little towards localising assertions, but happily doesn't go all the way to considering particular points in the program.

Similar notations can be used to specify user-defined **'functions' and 'mappings'. A mapping is a "pointwise" function, one whose value** depends only on the values of its arguments at the time it is called. Functions need not work pointwise. For example, we can write a function

STAT(X) which, when called at any time, returns the mean and standard deviation of all the values of X up to that time.

5. Implementations

The reader may agree that programming in Lucid might have some advantages, but wonder how practical the language is. Are there any implementations of Lucid, for example?

The formal proof theory for proving properties of Lucid programs is an intrinsic part of Lucid. This proof theory is based on a formal specification of the semantics of Lucid, given in a quite abstract mathematical way. If the proof theory is to apply to 'real' Lucid programs, the implementations must agree with this formal semantics. Thus, perhaps more than for other languages, interpreters and compilers for Lucid must be proved to be correct.

We have seen that evaluation of Lucid programs must be 'data driven'. This complicates the job of compiling the language, and in fact the full language seems not to be compilable. A large subset can be compiled however. The only compiler currently operational, in fact the only compiler even contemplated, is that of Christoff Hoffman [5]. It is written in the language B, and runs under the timesharing system of the Waterloo Honeywell 6060. A discussion of the new, and complicated, compiling techniques that were necessary is beyond the scope of this paper.

For a completely correct implementation of the full language, an interpreter is required. The construction of an interpreter is relatively straightforward, by basing it directly on the formal semantics. In the formal semantics, for elementary programs, the variables in a program denote infinite sequences of values. For example, in the square root program constructed in Section 3, I is <0,1,2,3,...> and J is <0,1,4,9,...>. We

write I_t to denote the t-th value in the sequence for I. The formal semantics shows that, for example $(A+B)_t = A_t + B_t$, and, for the Lucid functions, $(\underset{\sim}{\text{first}}\ A)_t = A_0$ and $(\underset{\sim}{\text{next}}\ A)_t = A_{t+1}$. $(A\ \underset{\sim}{\text{as soon as}}\ B)_t$ is A_k where B_k is the first true value in the sequence of truth values B. A sequence whose values are independent of t, such as $\underset{\sim}{\text{first}}$ A or A $\underset{\sim}{\text{as soon as}}$ B, is said to be constant. All numerals denote constant sequences. If, say, $result_0$ is the value desired, we can use the formal semantics to determine it, which will involve finding the values of other variables at other 'times'. This is essentially how Lucid interpreters work.

There are two interpreters currently working, one at Warwick (David May) and one at Waterloo (Tom Cargill [4]).

6. Programming Style

Lucid programs are well-structured, being formed from nested and concatenated loops. (There is no conditional statement but there are conditional expressions.) Morever, the use of variables corresponds to good programming practice. Each variable is used for one purpose, and has one 'meaning' (naturally considering local variables in different clauses as different variables). This style is not imposed because it is thought to be beneficial. Rather, this discipline of programming, which is becoming generally approved, is the natural result of designing a language which can treat assignment statements as equations.

It is very satisfying, even significant, that the studies of good programming practice and of ways of unifying mathematics and programming should lead to such similar results.

7. References

[1] Ashcroft, E.A., and Wadge, W.W., "Lucid, a Non Procedural Language with Iteration", CACM (to appear).

[2] Ashcroft, E.A., and Wadge, W.W., "Lucid, a Formal System for Writing and Proving Programs", SIAM J. Comput., 5, No. 4.

[3] Ashcroft, E.A., and Wadge, W.W., "Lucid: Scope Structures and Defined Functions". CS-76-22, Computer Science Dept., University of Waterloo.

[4] Cargill, T., "Deterministic Operational Semantics for Lucid", CS-76-19, Computer Science Dept., University of Waterloo.

[5] Hoffmann, C.N., "Design and Correctness of a Compiler for Lucid", CS-76-20, Computer Science Dept., University of Waterloo.

SUBSEQUENCES IN STACK SORTABLE PERMUTATIONS AND THEIR RELATION TO BINARY TREES

by

D. Rotem

Department of Computer Science

University of Waterloo

Waterloo, Ontario, Canada

ABSTRACT

The class SS_n of stack sortable permutations is known to be in 1-1 correspondence with n-noded binary trees. Expressions are derived for the average length of several types of monotonic subsequences in members of SS_n. The relations between these subsequences and properties of the corresponding tree are demonstrated. It is also shown that the permutation graph of a member of SS_n is an interval graph of a special type.

1. Introduction

Given a permutation $\Pi = <p_1,p_2,\ldots,p_n>$ and an empty stack, the elements of Π can be passed through the stack using two elementary operations coded 'S' and 'X'. The operation 'S' denotes 'put the next element of Π on top of stack' and 'X' stands for 'transfer the element on top of stack to the output'. A sequence L of the above mentioned operations, is called a valid operation sequence (or simply an operation sequence) if and only if (1) all elements of Π are transferred to the output and (2) the operation 'X' is never specified when the stack is empty. Conditions (1) and (2) imply that an operation sequence must consist of $2n$ operations, n of each kind, the number of 'X' operations may never exceed the number of 'S' operations when L is scanned from left to right.

We denote by $L(\Pi)$ the output permutation which results from passing Π through a stack. For example if $\Pi = <1,3,2,4>$ and $L = <S,X,S,X,S,S,X,X>$ then $L(\Pi) = <1,3,4,2>$. A permutation Π is sortable with a stack if and only if there exists an operation sequence $\overline{L}$ such that $\overline{L}(\Pi) = <1,2,\ldots,n>$, it is realizable with a stack if and only if an operation sequence $\overline{R}$ exists such that $\overline{R}(<1,2,\ldots,n>) = \Pi$.

Given a permutation Π, let $\overline{L}$ be the sequence of operations which sorts Π with a stack. Scanning $\overline{L}$ from left to right, we call each sequence of consecutive 'S' operations an S-group and such a sequence of 'X' operations an X-group. Clearly, the number of X-groups is equal to the number of S-groups, two S-groups are seperated by an X-group and vice versa. The S-specification and the X-specification of $\overline{L}$ are

vectors $< s_1, s_2, \ldots, s_\ell >$ and $< x_1, x_2, \ldots, x_\ell >$ respectively, where for $1 \leq i \leq \ell$ s_i denotes the size of the ith S-group and x_i the size of the ith X-group.

We denote by SS_n the class of permutations of order n which are sortable with a stack, and by SR_n the class of permutations of the same order which are realizable with a stack. Those two classes are related as follows,

$$\Pi \in SS_n \quad \text{if and only if} \quad \Pi^{-1} \in SR_n. \tag{1}$$

The class SR_n is characterized by Knuth [3, p. 239] by the following theorem,

Theorem 1: The permutation $\Pi = < p_1, p_2, \ldots, p_n >$ is a member of SR_n if and only if it does not contain a subsequence

$$< p_i, p_j, p_k > \text{ such that } p_i > p_k > p_j. \tag{2}$$

From this theorem and the relation (1) we obtain a characterization of SS_n as follows,

Theorem 1*: $\Pi \in SS_n$ if and only if it does not contain a subsequence

$$< p_i, p_j, p_k > \text{ such that } p_j > p_i > p_k. \tag{3}$$

Two binary trees T and T' are similar $(T = T')$ if they have the same 'shape', formally, they both have the same number of nodes, with the left subtree of T similar to the left subtree of T' and the same holds true for right subtrees. For a node j in T, we denote by $L_T(j)$ and $R_T(j)$ the left and right subtrees of j respectively.

A permutation can be mapped into a labelled binary tree T_π using the following well-known construction.

Construction - T

Given $\Pi = < p_1, p_2, \ldots, p_n >$ and an empty tree T, assign p_1 to the root of the tree; for each p_k, $k = 2,3,\ldots,n$ apply the rule

-- if p_k is inserted into a non-empty subtree rooted by p_i, it is inserted into $L_T(p_i)$ if $p_k < p_i$ otherwise p_k is inserted into $R_T(p_i)$ --

until an empty subtree is reached and then a root labeled p_k is created to that subtree.

Construction-T establishes a 1-1 correspondence between the set SS_n and the set of n-noded binary trees [3;6.2.2]. Given a labeled tree T, its corresponding member of SS_n can be obtained by reading the labels of T in symmetric order (root, left subtree and right subtree).

The class SS_n was studied in Knuth [3] and its relation to the classical ballot problem is shown in [4] and [7]. The correspondence between SS_n and the set of binary trees is used in [8] to generate and rank all 'shapes' of n-noded binary trees. The cardinality of SS_n is $C_n = (n+1)^{-1}\binom{2n}{n}$ (the n'th catalan number).

In this paper we study in detail some of the combinatorial properties of the class SS_n. In sections 2 and 3 expressions are derived for the expected length of some types of monotonic subsequences and the average number of inversions. The set $SS_n \cap SR_n$ is characterized and enumerated in section 4. In the last section the permutation graph associated with $\Pi \in SS_n$ is shown to be an interval graph of a special type.

2. Monotonic subsequences in SS_n and their relation to binary trees.

Let $\Pi = \langle p_1, p_2, \ldots, p_n \rangle$ be a permutation on the set $N = \{1,2,\ldots,n\}$. A descending subsequence of length k in Π satisfies,

$$p_{i_1} > p_{i_2} > \ldots > p_{i_k} \quad \text{and} \quad i_1 < i_2 < \ldots < i_k.$$

A descending subsequence is maximal in Π if no element of Π can be added to it without violating its monotonicity. A longest descending subsequence in Π (LDS) contains the maximum number of elements among all descending subsequences in Π. We get the corresponding definitions for ascending subsequences by replacing '>' with '<' in the above, where LAS stands for 'longest ascending subsequence'. For $j \in N$, we denote by $R_\pi(j)$ the set of elements to the right of j in Π, and by $L_\pi(j)$ the set of elements to the left of j in Π. Two elements p_i and p_j form an inversion in Π if $(p_i - p_j)(i-j) < 0$.

A descending run in Π is a sequence of successive elements $p_i, p_{i+1}, \ldots, p_{i+k}$ such that

(a) $p_{i-1} < p_i$

(b) $p_{i+k} < p_{i+k+1}$

(c) $p_i > p_{i+1} > \ldots > p_{i+k}$

(We assume that $p_1 > p_0$ and $p_n < p_{n+1}$.

The inversion-table of Π, is a vector $\langle b_1, b_2, \ldots, b_n \rangle$ such that for $1 \le i \le n$ b_i counts the number of elements in $R_\pi(i)$ which are smaller than i. It is well-known, that an inversion-table uniquely determines its corresponding permutation. We denote by Π^{-1} the inverse permutation of Π, if $\Pi = \Pi^{-1}$ it is called an involution.

Example:

Let $\Pi = <3,6,4,5,2,1>$. Then $<3,2,1>$ is a maximal descending subsequence in Π, $<6,4,2,1>$ and $<3,4,5>$ are an LDS and an LAS respectively in Π, $R_{\pi}(4) = <5,2,1>$ and $L_{\pi}(6) = <3>$. The inversion-table of Π is $<0,1,2,2,2,4>$. The runs of Π are $<3>$, $<6,4>$ and $<5,2,1>$. □

Theorem 3: The expected length of an LDS in a random permutation in SS_n is asymptotically

$$\sqrt{\pi n} - 1.5 + \frac{11}{24}\sqrt{\frac{\pi}{n}} + O(n^{-3/2}) \qquad (4)$$

Proof: We show that the length of an LDS in $\Pi \in SS_n$ is equal to the depth of stack which is needed to traverse T_{Π} in symmetric order. Equation (4) is Knuth's result for the average depth of stack [3,Ex. 2.3,11].

We observe that the sequence of insertions and removals from stack made during the symmetric traversal of T_{Π} is equivalent to the sequence of operations required to sort Π with a stack.

Let $D = <d_{i_1},d_{i_2},\ldots,d_{i_\ell}>$ be an LDS in Π. While sorting Π, no member of D can leave the stack before d_{i_ℓ} so the stack must have at least ℓ entries.

Conversely, assume that the stack contains m elements during the sorting process and $m > \ell$. Let $B = <b_{i_1},b_{i_2},\ldots,b_{i_m}>$ be the elements in the stack, then B must be a descending subsequence in Π, a contradiction to the definition of D. □

Remark: The problem of finding the expected length of an LDS (or an LAS) in a random permutation is still unsolved analytically. Experimental results show good agreement with $2\sqrt{n}$ [2].

We need the following definitions to prove the corresponding result on the LAS.

A composition of a whole number n into m parts is a vector $C = \langle c_1, c_2, \ldots, c_m \rangle$ such that $c_i > 0$ for $1 \le i \le m$ and $\sum_{i=1}^{m} c_i = n$. A composition C of n can be represented as a zig-zag graph, this graph contains m rows with c_i dots in the i-th row, for $i > 1$ the first dot in the i-th row is written under the last dot in row i-1. Given a composition C, we obtain its conjugate composition $\overline{C} = \langle \overline{c}_1, \overline{c}_2, \ldots, \overline{c}_{n+1-m} \rangle$ such that for $1 \le i \le n+1-m$, $\overline{c}_i$ is equal to the number of dots in the i-th column (from left) of the zig-zag graph of C. For example let $C = \langle 3,2,4,1 \rangle$ be a composition of the integer 10. The zig-zag graph of C is

```
...
 ..
  ....
    .
```

therefore $\overline{C} = \langle 1,1,2,2,1,1,2 \rangle$.

Let Π and Π_{RF} be two members of SS_n (not necessarily distinct) such that their corresponding trees T_π and $T_{\pi_{RF}}$ are reflections of each other about the vertical axis.

Lemma 1: Let $X = < x_1, x_2, \ldots, x_k >$ and $X_{RF} = < x'_1, x'_2, \ldots, x'_m >$ be the X-specifications of $\overline{L}$ and $\overline{L}_{RF}$ respectively. Then the vectors $X^R = < x_k, x_{k-1}, \ldots, x_1 >$ (the reverse of X) and X_{RF} are conjugate compositions of n.

Illustration: Consider the permutations $\Pi = < 6,3,2,1,4,5,8,7 >$ and $\Pi_{RF} = < 3,1,2,6,5,4,7,8 >$. The corresponding binary trees are shown in Figure 1 (a) and (b) respectively.

The X-specification of $\overline{L}$ is $X = < 3,1,2,2 >$ and $X^R = < 2,2,1,3 >$. The zig-zag graph of X^R is .. and therefore its conjugate is $\overline{X}^R = < 1,2,3,1,1 >$, we then have $\overline{X}^R = X_{RF}$.

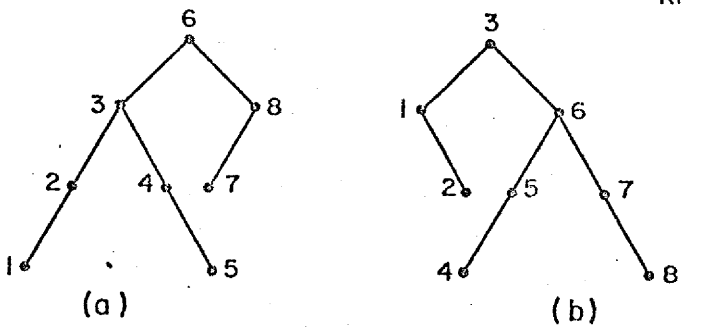

Figure 1

Proof: A binary tree is traversed in reverse symmetric order if a root and its two subtrees are visited in the order (1) right subtree (2) root (3) left subtree. We observe that the operations which are required in order to traverse T_π in reverse symmetric order are equivalent to those necessary for traversing $T_{\pi_{RF}}$ in symmetric order. Therefore $\overline{L}$ and $\overline{L}_{RF}$ specify the stack operations for traversing T_π in symmetric and reverse symmetric order respectively. For two consecutive labels i and $i-1$ we can have

(a) $i \in R_{T_\pi}(i\text{-}1)$ or (b) $i\text{-}1 \in L_{T_\pi}(i)$. While traversing T_π in symmetric order, (a) implies that i must be stacked after $i\text{-}1$ is written on output and therefore $X(i\text{-}1)$ and $X(i)$ are in different X-groups, (b) implies that i is present in stack when $i\text{-}1$ is written, hence $X(i)$ and $X(i\text{-}1)$ are in the same X-group. It is easy to see that in the reverse symmetric order traversal of T_π we have exactly the converse, i.e. the labels i and $i\text{-}1$ are written on output by the same X-group in $\overline{L}_{RF}$ in case (a) and by different X-groups in case (b).

We can represent X as a zig-zag graph in which the i-th row contains the elements written by the i-th X-group in $\overline{L}$. By the above argument, it follows that the i-th X-group in $\overline{L}_{RF}$ will write out elements of the i-th column in this graph, where counting starts from the rightmost column. For example in the above illustration the graph is

123
4
56
78

and the X-group of $\overline{L}_{RF}$ write out $<8>, <7,6>, <5,4,3>, <2>, <1>$, where brackets enclose elements of the same X-group. Therefore X^R and X_{RF} are conjugate compositions and $k = n+1-m$. □

Lemma 2: The length of the LAS in $\Pi \in SS_n$ is equal to the number of components in the S-specification (X-specification) of its sorting sequence.

Proof: Let $\overline{L}$ be a sorting sequence for Π with S-specification

$< s_1, s_2, \ldots, s_\ell >$. Then clearly Π must have exactly ℓ descending runs where the size of the i-th run is s_i. Let an LAS in Π be of length k. Then $k \leq \ell$ since no two elements in an LAS are in the same descending run. To show that $\ell \leq k$, we construct a sequence $D = < d_1, d_2, \ldots, d_\ell >$ where d_i is the last element in the i-th descending run in Π. We show that D is an ascending subsequence in Π by deriving a contradiction. Suppose that for some i $d_i > d_{i+1}$, then there must be an element d in the $i+1^{st}$ run such that $d > d_i$ and $d > d_{i+1}$ and Π must contain a forbidden subsequence $< d_i, d, d_{i+1} >$. □

Theorem 4: The expected length of the LAS in a random permutation is SS_n is $\frac{n+1}{2}$.

Proof: We define a mapping $RF: SS_n \rightarrow SS_n$ such that $\Pi \in SS_n$ is mapped into Π_{RF} by RF. Suppose that the length of the LAS in Π is equal to k. By Lemma 2 this is also the number of components in the S-specification and X-specification of the sorting sequence $\bar{L}$. From Lemma 1, the length of the LAS in Π_{RF} is $n+1-k$. Since RF is a one-to-one correspondence our result follows. □

Another subsequence, which was studied in permutations is the sequence of left to right maxima which is also called the distinguished subsequence by Brock & Baer [1]. For example the distinguished subsequence in $< 1,3,2,5,4,6 >$ is $< 1,3,5,6 >$. It is shown by Knuth [3], [4] and in [1] that the expected length of this subsequence in a random permutation is H_n (the n-th harmonic number). The next theorem gives the corresponding

result for a random permutation in SS_n.

Theorem 5: The expected length of the distinguished subsequence in a random permutation of SS_n is $3 - \frac{6}{n+2}$.

Proof: Given $\Pi = \langle p_1, p_2, \ldots, p_n \rangle \in SS_n$ let $\langle p_{i_1}, p_{i_2}, \ldots, p_{i_n} \rangle$ be the distinguished subsequence in Π. We can form $k+1$ permutations $\Pi_1, \Pi_2, \ldots, \Pi_{k+1}$ of length $n+1$ from Π by inserting the number $n+1$ in each of the positions immediately to the left of p_{i_j} in Π for $1 \le j \le k$ or placing $n+1$ as the last element in Π. For example, if $\Pi = \langle 1,3,2,4 \rangle$ then $\langle 5,1,3,2,4 \rangle$ $\langle 1,5,3,2,4 \rangle$ $\langle 1,3,2,5,4 \rangle$ and $\langle 1,3,2,4,5 \rangle$ are formed in this way. We now show that for $1 \le i \le k+1$ $\Pi_i \in SS_{n+1}$. If not, then for some $2 \le j \le k$ Π_j must contain a forbidden subsequence of the form $\langle p_i, n+1, p_\ell \rangle$ and

$$p_i > p_\ell \qquad i < \ell \tag{5}$$

But this implies that Π must contain a subsequence

$$\langle p_i, p_{i_j}, p_\ell \rangle \tag{6}$$

Now p_{i_j} is in the distinguished subsequence, and therefore sequence (6) is of type (3) thus contradicting $\Pi \in SS_n$. It is easy to see that inserting $n+1$ in any other position of Π will create a permutation Π' such that $\Pi' \notin SS_{n+1}$. On the other hand all the members of SS_{n+1} can be generated from the members of SS_n in this way. Let a_π be the length of the distinguished subsequence in Π. Then

$$|SS_{n+1}| = C_{n+1} = \sum_{\Pi \in SS_n} (a_\pi + 1). \tag{7}$$

$$C_{n+1} = \Sigma a_\pi + C_n \tag{8}$$

$$\frac{\Sigma a_\pi}{C_n} = \frac{C_{n+1}}{C_n} - 1 = \frac{\frac{1}{n+2}\binom{2n+2}{n+1}}{\frac{1}{n+1}\binom{2n}{n}} - 1 \tag{9}$$

which gives

$$\frac{\Sigma a_\pi}{C_n} = 3 - \frac{6}{n+2} \qquad \square \tag{10}$$

Remark: This result is directly related to a theorem by Munro [5] which shows that the average length of a random walk on a binary tree is $2 - \frac{6}{n+2}$.

Corollary: The expected length of the first descending run in a random permutation of SS_n is $3 - \frac{6}{n+2}$.

Proof: Given $\Pi \in SS_n$, the elements of the LAS in Π form the rightmost path in T_π. By symmetry, the average length of the leftmost path over all n-noded binary trees is also $3 - \frac{6}{n+2}$. This path in T_π is formed by the members of the first descending run in Π, since under Construction -T this path is completed before any other part of the tree is constructed. $\square$

3. The Number of Involutions in SS_n

It is well known [4] that a permutation is an involution if and only if it does not contain a cycle with more than two elements. Using this fact, we prove in Lemma 3 that the set of involutions in SS_n is equal to $SS_n \cap SR_n$. A simple expression for the cardinality of this set is then calculated in Theorem 6.

Lemma 3: Let $\Pi \in SS_n$, then Π is an involution if and only if $\Pi \in SS_n \cap SR_n$.

Proof: The 'only if' part follows directly from the definitions. We prove the 'if' part by showing that a permutation which is a non-involution must contain at least one of the subsequences (2) or (3), therefore it is not a member of $SS_n \cap SR_n$.

Let Π be a non-involution, then Π contains a cycle of length $k \geq 3$. Let this cycle be $[a_1, a_2, \ldots, a_k]$ where a_1 is the smallest element in this cycle. We can arrange the elements of the cycle according to their original order in Π in the following way. First we sort the cycle into ascending order, then write under each element its right successor in the cycle, the second line thus obtained forms a subsequence of Π. For example, if Π contains the cycle $[1,4,3,6,5]$ then the above operations will give $\begin{bmatrix} 1 & 3 & 4 & 5 & 6 \\ 4 & 6 & 3 & 1 & 5 \end{bmatrix}$ and $<4,6,3,1,5>$ is a subsequence of Π (a_1 is considered to be the right successor of a_k). We distinguish between two cases:

Case 1: $a_2 < a_3$. Let $k = 3$, then after sorting the cycle we get $\begin{bmatrix} a_1 & a_2 & a_3 \\ a_2 & a_3 & a_1 \end{bmatrix}$ and $< a_2, a_3, a_1 >$ forms a subsequence (3) in II. Assume that $k > 3$. We sort the cycle by placing a_2 on the right of a_1, then inserting the elements $a_k, a_{k-1}, \ldots, a_3$ one by one into their correct positions. We write under each element its right successor when it is inserted. If $a_k > a_2$ then a_k is inserted on the right of a_2 and we get the same result as in the case $k = 3$, a_k playing the role of a_3. Assume that $a_k < a_2$. We insert $a_{k-1}, a_{k-2}, \ldots$ into their positions until an element a_{k-i} is found such that $a_{k-i} > a_2$, the existence of such an element is guaranteed since $a_3 > a_2$. The element a_{k-i+1} is smaller than a_2, hence after inserting a_{k-i} we have the following configuration

$$\begin{bmatrix} a_1 \cdots\cdots a_{k-i+1} \cdots\cdots a_2 & a_{k-i} \\ a_2 \cdots\cdots a_{k-i+2} \cdots\cdots a_3 & a_{k-i+1} \end{bmatrix} \tag{11}$$

and $< a_2, a_3, a_{k-i+1} >$ forms a subsequence (3) in II.

Case 2: $a_2 > a_3$. If $k = 3$ we have the configuration $\begin{matrix} a_1 & a_3 & a_2 \\ a_2 & a_1 & a_3 \end{matrix}$ after sorting the cycle, and $< a_2, a_1, a_3 >$ forms a subsequence (2) in II. Assume $k > 3$. If $a_k < a_2$ we obtain the same subsequence, we therefore consider the case $a_k > a_2$. We use the same procedure as in Case 1, this time we search for the first element a_{k-i} such that $a_{k-i+1} > a_2$ and

$a_{k-i} < a_2$. We then have the configuration

$$\begin{bmatrix} a_1 & a_{k-i} & a_2 & \cdots & a_{k-i+1} & \cdots \\ a_2 & a_{k-i+1} & a_3 & \cdots & a_{k-i+2} & \cdots \end{bmatrix} \tag{12}$$

and $< a_2, a_{k-i+1}, a_3 >$ is a subsequence (3) in Π. □

Theorem 6: The number of involutions in SS_n is equal to 2^{n-1}.

Proof: By Lemma 3, we have to show that there are 2^{n-1} permutations of length n which do not contain subsequences (2) or (3). A permutation $\Pi \in SS_n \cap SR_n$ can be characterized by the following property of its maximal descending subsequences.

Let $D = < d_{i_1}, d_{i_2}, \ldots, d_{i_k} >$ be a maximal descending subsequence in a permutation Π of order n, then $\Pi \in SS_n \cap SR_n$ if and only if for $i \leq j \leq k-1$,

$$d_{i_j} = d_{i_{j+1}+1} \quad \text{(elements of } D \text{ appear in reverse natural order).} \tag{13}$$

Proof: Clearly every permutation which satisfies condition (13) is a member of $SS_n \cap SR_n$, since each of the forbidden subsequences (2) and (3) have at least one pair of elements which belong to a descending subsequence and are not in reverse natural order. We now show that if any violations of condition (13) occur in Π then $\Pi \notin SS_n \cap SR_n$.

Suppose that for some index m, $(1 \leq m \leq k-1)$ $d_{i_m} \neq d_{i_{m+1}+1}$. Let $d_{i_{m+1}+1} = \ell$. Then ℓ cannot appear between d_{i_m} and $d_{i_{m+1}}$ in Π,

since it is not a member of D. Therefore one of the two subsequences $< \ell, d_{i_m}, d_{i_{m+1}} >$ or $< d_{i_m}, d_{i_{m+1}}, \ell >$ must appear in Π, thus contradicting $\Pi \in SS_n \cap SR_n$.

For each permutation $\Pi \in SS_n \cap SR_n$, we can generate two permutations Π_1 and Π_2 of order $n+1$ as follows;

(a) generate Π_1 by inserting $n+1$ one position to the left of n in Π,

(b) generate Π_2 by putting $n+1$ after the rightmost element in Π.

Clearly, condition (13) is not violated in Π_1 and Π_2 thus generated. Furthermore, inserting $n+1$ in any other position of Π, generates a maximal descending subsequence (with $n+1$ as its first element) which does not satisfy condition (13). Therefore Π_1 and Π_2 belong to $SS_{n+1} \cap SR_{n+1}$. Since all the elements of $SS_{n+1} \cap SR_{n+1}$ are generated in this way, we have

$$|SS_{n+1} \cap SR_{n+1}| = 2|SS_n \cap SR_n|. \qquad (14)$$

Our result follows from the fact that $SS_3 \cap SR_3$ contains 4 elements, namely, $< 1,2,3 >$, $< 1,3,2 >$, $< 2,1,3 >$, $< 3,2,1 >$. □

4. The Average Number of Inversions in SS_n

Lemma 4: Let $< b_1, b_2, \ldots, b_n >$ be the inversion-table of $\Pi \in SS_n$, then for node labelled k in T_π, $|L_{T_\pi}(k)| = b_k$.

Proof: We show that the elements which are counted by b_k are exactly the ones which are inserted into $L_{T_\pi}(k)$ by Construction-T. Clearly, only an element j such that $j < k$ and $j \in L_\pi(k)$ can be inserted into $L_{T_\pi}(k)$. If no such element exists in Π then $b_k = 0$ and the subtree $L_{T_\pi}(k)$ is empty. Assume $b_k > 0$. Since $\Pi \in SS_n$, all elements in $L_\pi(k)$ are either bigger or smaller than both k and j, any other possibility will create a subsequence (3) in Π. Therefore, application of the rule of Construction-T will force j to be inserted into the same subtrees as k, finally j must be compared with k and since $j < k$ it follows that $j \in L_{T_\pi}(k)$. □

Theorem 7: The average number of inversions in a random permutation of SS_n is

$$\frac{1}{2}\left(\frac{4^n}{C_n} - 3n - 1\right). \qquad (15)$$

Proof: Let $i(\Pi)$ denote the number of inversions in a permutation Π and $int(T)$ the internal path length of the tree T. The sum of sizes of all subtrees in a binary tree (or any other tree) is equal to $int(T)$. This follows from the fact that in a tree T, the distance of vertex i

from the root is equal to the number of subtrees in which i participates.

Let $< b_1, b_2, \ldots, b_n >$ be the inversion-table of a permutation $\Pi \in SS_n$, then by definition

$$\sum_{i=1}^{n} b_i = i(\Pi). \tag{16}$$

By lemma 4, $i(\Pi)$ is the sum of sizes of all left subtrees in T_π. Hence, by the symmetry of left and right subtrees

$$\sum_{\Pi \in SS_n} int(T_\pi) = 2 \sum_{\Pi \in SS_n} i(\Pi). \tag{17}$$

The value of the left member of (17) is given in [3,p. 404] as

$$\sum_{\Pi \in SS_n} int(T_\pi) = 4^n - (3n+1)C_n, \tag{18}$$

from which (15) follows.

It is interesting to note that on the average a random permutation of SS_n contains $O(n^{1.5})$ inversions, where as the corresponding value for a random permutation of order n is $O(n^2)$.

5. Graphs Associated with SS_n

We give some definitions and notations from graph theory which are required in this section.

A graph $G(V,E)$, consists of a vertex set V and an edge set E, such that each edge in E is associated with two vertices in V called its end points. We consider here only graphs which have no two edges with the same two end points (parallel edges), and no edge for which its two end points are the same (self loop). Two vertices are adjacent if they are the end points of the same edge, this is denoted by $v_i \overset{}{\underset{G}{---}} v_j$, otherwise they are non-adjacent denoted by $v_i \overset{/}{\underset{G}{---}} v_j$. The complement of G, denoted by G^C, has the same vertex set as G, two vertices are adjacent in G^C if and only if they are non-adjacent in G.

A direction can be assigned to the edge $v_i \overset{}{\underset{G}{---}} v_j$, this is denoted by $v_i \to v_j$. If all edges of G are assigned a direction, it is called a digraph (directed graph). A digraph is transitive if for $v_i, v_j, v_k \in V$, the existence of $v_i \to v_j$ and $v_j \to v_k$ implies $v_i \to v_k$. A graph G is transitively orientable (TRO), if it is possible to orient all its edges such that its directed image is transitive.

Let $G(N)$ be a graph which has its vertices labeled by the set $N = \{1,2,\ldots,n\}$. Then $G(N)$ has a defining permutation with respect to its labeling, if there is a permutation Π on N such that;

$i \overset{}{\underset{G(N)}{---}} j$ (vertices are called by their labels) if and only if i and j form an inversion in Π.

A graph G is a permutation graph, if at least one of the possible

labelings of its vertices with N, gives rise to a defining permutation.

Example: A permutation graph G, with two labelings and their respective defining permutations, is shown in Figure 2

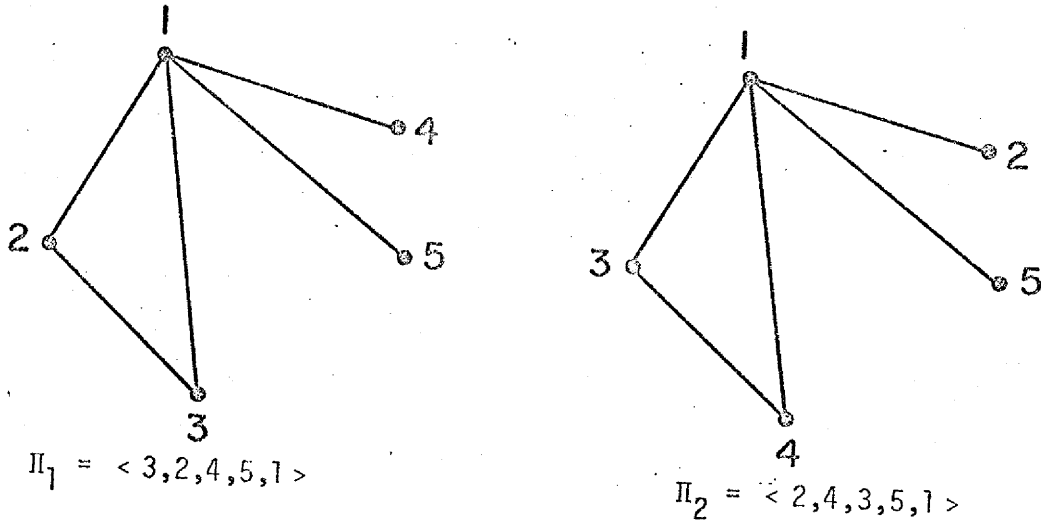

Figure 2

The next theorem of [6] demonstrates the connection between permutation graphs and transitive graphs.

Theorem : A graph G is a permutation graph if and only if both G and G^C are TRO graphs.

A graph G with vertex set V($|V| = n$), is an interval graph if there exists a family of intervals on the line $I = (I_1, I_2, \ldots, I_n)$ such that

$v_i \in V$ corresponds to an interval I_i, and $v_i \underset{G}{\text{---}} v_j$ if and only if $I_i \cap I_j \neq \phi$. A nested interval graph is an interval graph which has a representing family I such that for each pair of intervals I_i and I_j, if $I_i \cap I_j \neq \phi$ then either $I_i \subset I_j$ or $I_j \subset I_i$ holds.

Theorem 8: The following conditions are equivalent:

(1) G is a permutation graph, with a defining permutation $\Pi \in SS_n$.

(2) G is a nested interval graph.

Proof: (2) Consider the sorting sequence of Π, where a line is drawn from each S operation to its corresponding X operation which removes from stack the element stacked by S. For example, for $\Pi = <3,1,2>$ the following sorting sequence and lines are drawn S S X S X X. Let I_i be the line drawn between the S and X which stack and unstack i in Π. For a pair of intervals I_i and I_j assume that I_i has its left end to the left of I ($i \in L_\pi(j)$). Then two cases are possible:

(a) $i < j$, i leaves the stack before j is stacked and $I_i \cap I_j = \phi$

(b) $i > j$, i leaves the stack only after j is unstacked and $I_i \supset I_j$.

In the permutation graph G labeled with Π, vertices labeled i and j are adjacent only in case (b) where i and j form an inversion in Π hence G is a nested interval graph. Conversely, let I be a family of n intervals which is represented by a nested interval graph G. Then, I can be mapped into a sequence of S's and X's by reversing the above procedure. By reading this sequence of S's and X's from left to right we obtain a sorting sequence of some $\Pi \in SS_n$ and Π is a defining permutation for G. □

Conclusions

In this paper we studied some of the combinatorial properties of members of SS_n, and the relations of these properties to the corresponding binary tree. It was observed that members of SS_n tend to be more 'ordered' than ordinary permutations in the sense that on the average they contain less inversions, longer maximum ascending subsequences and shorter maximum descending subsequences.