1 Introduction

In this paper we present a model introduced in the 1960's by Kristen Nygaard of the Norwegian Computing Center (NCC), Oslo, to explain the mechanics of Algol 60 and Simula. The model was used extensively in the first text on Simula (Birtwistle, et al, 1973) and on many courses given by NCC.

When a high level program is compiled it is translated into an equivalent program in machine code. Because of this equivalence, program execution can be pictured in terms of the source text rather than in machine oriented terms. This is important because we can then understand how programs run and argue about correctness in source code terms without getting bogged down in the details of a specific implementation.

Many high level languages utilise block structure as the basic paradigm for defining the language for the programmer and as a basis for execution control at run time. A program under execution may be viewed as a stack of *block instances*. Johnston's contour model is commonly used to show dependency relations between block instances by drawing the main program instance with smaller blocks nested appropriately inside. Explanations using these diagrams then attempt to identify the scope of variables within different blocks. While such models can be used effectively to explain relevant concepts in languages like BASIC, FORTRAN, and COBOL, they fall short in explaining critical concepts in languages like Pascal or C because the dynamic nature of program execution is not shown. The crucial difference between Nygaard's model and that of Johnston's contour model is the distinction between a *block* which is viewed as a piece of source text containing declarations and actions, and a *block instance* which is the agent created for carrying out those actions in an environment enriched by the declarations when the program is running. A block instance is created only when control passes through the initial **begin** of the block and is deleted when control passes through its final **end**.

Nygaard's model turns out to be ideal for the hand simulation of source programs since it gives an understanding of how programs actually run. In addition, other hard concepts such as passing arguments by value and by reference and recursion can be visualised directly.

The paper develops and presents key ideas incrementally, and always by example. To introduce the dynamic nature of block structured programs and the binding rule which applies, blocks, block instances, and templates are defined.

A block instance models the user supplied data and actions of a block of source text. In addition, it has two system links: a static link to its environment enabling the look up of global quantities, and a dynamic link (return address) to indicate the position where execution is resumed when a block instance is deleted. Because block instances are anonymous, they can only be entered by passing through their initial *begin*, that is, from one specific program point. Procedures generalise this concept in two ways. First they are named which means that they can be called from many program points. Second they are allowed parameters which serve to initialise certain local quantities (call by value), or widen their environment in a controlled fashion (call by reference). Procedure instances have similar formats as block instances, they merely extend the declaration part to include parameters. Parameters may be considered to be merely initialised local quantities. In section three, the binding rule for associating definitions to quantities is enunciated and discussed, along with the passing of value and reference parameters.

Functions are similar to procedures except that they return a single value. Section four details how space is made available in function templates for the result to be returned and how the 'result slot' may be accessed.

Section five explains recursion. Recursion is treated as ordinary decomposition, the only new wrinkle being that the sub-problem has the same as form as the original but will be easier to solve in that its parameter values are 'simpler'. In practice this means that several instances of the same block may exist at the same time and each sub-problem is modelled by its own instance.

In section six, the passing of more complex types of parameters such as arrays and procedures is explained.

2 Basic Block Structure Within the Run-Time System

When Pascal program execution begins, an instance of the Pascal run-time system is entered. Such and instance may be depicted with the same block structure and basic relationships as the block instances created by the user program. This basic pattern will first be defined, then applied to the run-time system, and then successively refined.

A *block* is a piece of source text

```
begin
  declarations;
  statements;
end;
```

which describes an action sequence and the local "scratchpad" definitions needed to carry it out. When a program containing the block is executed, an *instance* of that block is created every time program control passes through its initial **begin.** That instance of the block is deleted when program control passes through its final **end.** Several instances of the same block may be created during program execution. However, each instance of the same block has the same template which we visualize with three levels as shown in Figure 1.

The three sections are:

a header which furnishes the block with a unique name. It also links it to its environment with a static link (*SL*) by pointing to the instance in which this one was defined and a dynamic link (*DL*) holding its return address (a program point).

a declaration which has slots for each locally defined quantity (variable, label, procedure, etc) and its current value (if applicable).We make no assumptions as to the values of declared quantities prior to their first assignment.

a body which contains the sequence of statements for the program block.

The following program which reads two numbers and computes the sum can be used to show the program block instance and the extension to the run-time system. d

```
program add2 (input, output);
var x, y, sum : real;
begin
   read (x, y);
   sum := x + y;
```

```
    writeln ('Sum = ', sum);
  end.
```

When program execution starts, the first thing that happens is that a Pascal run-time system instance is created and entered. It is here that links to the program library can be established for standard routines like *writeln* as well as some routines invisible to the user such as parameter passing routines and processes which are necessary for entering and leaving subroutines. The run-time system instance also contains certain initializing and clean-up operations which are common to all Pascal programs. The run-time system instance can be shown using our standard model, as in Figure 2.

The run-time system provides the link between the execution and the user. In addition, a marker which will be referred to as **sequence control** (SC) indicates the current instruction. When execution begins, sequence control points to 'initialize' in the run-time system block instance. Initializing actions include some accounting, the allocation of memory for the program, and the opening of input, output and library files. Sequence control is then moved to the next instruction, 'execute add2'.

At this point, space for the *incarnation* of the program block is allocated. The block instance corresponding to the source program is created as shown in Figure 3(a).

When a block instance is produced, two system variables, **DL** and **SL** are initialized. These are shown in the heading of the block instance along with an identifying name. The DL contains the information which directs the return of sequence control to the run-time system following the completion of the instructions in instance. The SL indicates the path which must be followed if a non-local quantity required for the execution of a particular instruction is not contained in the current instance. Every instance of a block, procedure or function in Pascal possesses such links. In this case, whereas definitions of *x, y* and *sum* can be found locally, there is no local definition for either *read* or *writeln.*

SC, the sequence control indicator is now positioned opposite the first instruction found in the body of the program which is a *read* statement. The meaning of the read statement is not in the current block and the SL pointer is used to locate the next instance in which to search for a meaning for read. In this case, the SL points to the run-time system instance which contains the Pascal library. Having then found a match for the instruction, it is executed as the library dictates. If the user supplies the values of 2.0 and 3.0, these are then mapped to the declarations for x and y in the program instance.

The process of finding a match for the instruction therefore is one in which the declarations are searched to see if the quantity is defined. If a match is found, the occurrence is mapped onto that quantity. The current instance is searched first. If this search is unsuccessful, the SL pointer is followed to the enclosing instance. This process is repeated until a match is found. The process is one of binding the instruction to a declaration and can be stated as the following **binding rule**:

> Look for a match for the quantity in the current instance. If a match is found, accept the match. If no match is found, follow the SL pointer and look for a match in that instance. This step is repeated until a match is found.

At least one occurrence of the identifier must be found. If the indentifier cannot be found, then the program would not have compiled since this is the same look-up mechanism used by the Pascal compiler. The first possible match is taken and other quantities with the same identifier further back along the SL chain are rendered inaccessible to the current block. The compiler will also ensure via typing that the identifier is meaningful.

In the example, question marks (?) initially appear for the values of x, y and *sum* since no values have yet been assigned to these quantities. At this point they have unpredictable values which are dependent on how the machine represents values which are not yet explicitly assigned. Matches for *read* and *writeln* are found in the run-time system instance while matches for *sum*, x, and y are found in the program instance. When the first instruction is complete, and assuming that the values 2.0 and 3.0 were provided as input, these values would have been mapped onto the declaration portion for x and y replacing the '?'. The sequence control then moves to the next statement, (*sum* $:= x + y$;). The current instance contains the values required to assign a value to *sum*. In the next statement, the binding rule will locate *writeln* in the run-time system instance and *sum* in the program instance. Finally, with no further statements, the sequence control follows the DL pointer to the run-time system and is positioned at the next statement (the one textually following the one just completed) as shown in Figure 3(b). Various clean-up operations are then carried out including completion of accounting, closing of files and the release of any other requested resources back to the operating system. The program instance no longer exists when the sequence control has exhausted actions of the body of the instance. All quantities declared within the instance also disappear. When the clean-up is complete, the run-time system instance also disappears.

3 Extension to Procedures

To keep things simple, we first introduce procedures without arguments. Reduction of code repetition and the capability of increasing flexibility of code segments (modularity considerations) through the use of procedures will also be included. Finally, functions will be discussed.

3.1 Procedures with no Parameters

The program below contains a simple procedure with no parameters.

```
program Main (input, output);
var   n : integer;

procedure P;
var   k : integer;
begin
   n := n + 1;
   read (k);
   writeln ('n= ', n,' k= ', k);
end; {of P}

begin {Main}
   n := 0;
   P;
   ...
   P;
end.{of Main}
```

When this program is executed, the run-time system instance does the normal initializations and then establishes and enters the program *Main* as shown in Figure 4.

Two quantities have been declared in *Main, n* and *procedure P.* Initially, the value for *n* is undefined but when the first instruction in the body of the instance named *Main* is executed, *n* is assigned the value of zero. When sequence control moves to the next instruction, the instance representing procedure P is created as shown in Figure 5. The run-time system instance should be well enough understood at this point so it is not shown in full in future diagrams. Instead, it is represented by using the header only.

At the point that the instance for *P* is created, the integer *k* exists although its initial value is undefined. Sequence control then moves to the first statement of procedure P which results in the assignment of 1 to *n.* Since *n* is not found in the instance *P*, the binding rule matches its occurrence to the integer variable in program Main. Note that the value assigned to n also is kept in Main. In the next statement, the binding rule places *k* locally but needs to follow the SL pointer back two steps to locate *read.* Upon completion of the last statement in P, sequence control returns to the statement immediately after that indicated by DL and the instance disappears. Thus since the value of *n* was stored in *Main*, it is retained but the value of *k* is lost when the instance is erased. When the execution of *Main* comes to another invocation of *P*, the initial value for *n* will be 1 and for *k* will be unknown until after the *read* statement. The value of *n* at the completion of the first statement will be 2 but whatever the value assigned to *k*, its value is not retained after the procedure is complete.

Procedures retain access to their declaration environment regardless of the position of the call since SL is a static link. This concept often causes problems in understanding programs. The following example illustrates this concept.

```
program Main (output);
var  x : real;

procedure bumpx;
begin
   x := x + 1.0;
end;

procedure Q;
var  x : real;
begin
   x := 12.0;
   bumpx;
end;

begin {Main}
   x := 2.0;
   Q;
   write ('x= ', x:4:1);
```

```
end.
```

The diagram in Figure 6(a) shows the instances as they exist following the execution of the second statement in *Main*, the call to *Q*. Figure 6(b) shows the instances following the call to *bumpx* in *Q*. Notice that the SL of *Q* is in *Main* as is the SL of *bumpx* since that is were both are defined. However the DL's of the two instances refer to program points in different block instances.

During execution of the program, the program instance, *Main*, is created after the initialization processes. Real x is undefined in the declarations until it is set to 2.0 by the first instruction. When *Q* is called, since the procedure is part of the declarations in *Main*, this is used to create the instance called *Q*. In this instance, there is a real variable x which is undefined initially but which is given a value of 12.0 when the first instruction in *Q* is executed. Note that sequence control moves to the first instruction in the newly created instance each time the creation process is invoked. A call is now made to *bumpx*. A search of the declarations in *Q* does not find a match so the SL pointer is followed to find the procedure in *Main*. The SL pointer always points to the instance in which the match required to create the instance was located. Thus, when *bumpx* increments x, it searches first in the *bumpx* instance and when the match is not found, follows the SL pointer. In this case, the pointer goes back to *Main* and it is this declaration of x which is used. Thus x is incremented to 3.0. As sequence control moves to the next command, two instances are removed in turn since the last command is *end* for both *bumpx* and *Q*. When the value of x is written, the value referenced is the value in the Main program and 3.0 is written.

By using this technique, it is obvious from the diagram produced that there are no declarations for x in *bumpx*. The SL pointer which established the link between *bumpx* and *Main* when the procedure instance was created indicates clearly which x is to be incremented. The same x gets bumped whenever *bumpx* is called regardless of the position within a procedure or the depth of nesting since every call has the same SL.

3.2 Procedures with Value Parameters

Procedures can significantly reduce the repetition of code and can also widen the range of application through the use of appropriate formal value parameters. Suppose that the output of a program has four blank lines interspersed to make the result more readable. Without the use of procedures it is necessary to insert the appropriate code at each point where the four blank lines are required either as four *writeln* commands or as a looping construct. Repetition of code can be substantially reduced by creating a procedure (with no parameters) with the result that a single statement, the procedure call, can be used to insert the blank lines where required as shown below.

```
program Main (input, output);

procedure NL4;
var k : integer;
begin
   for k := 1 to 4 do
      writeln;
   end;
begin
   ....;
   NL4;
   ....;
   ....;
   NL4;
end.
```

Thus the procedure reduces redundant code and increases modularity. However, using a procedure with no parameters does not provide the programmer with any flexibility – four blank lines are always inserted by the procedure call. Altering the procedure to produce 3 or 5 blank lines can be easily accomplished by changing the terminator for the loop but the problem of producing a fixed spacing still exists. Suppose that the program is, instead, modified as follows:

```
program Main (input, output);
var b : integer;
procedure NL;
var k : integer;
begin
   for k := 1 to b do
     writeln;
end;
begin
   ....;
   b := 3;
   NL;
   ....;
   ....;
   b := 5;
   NL;
end.
```

There is now more flexibility in how much space is left but it still requires that the amount of space is part of the code and is fixed for the execution. Some changes can be made by inserting some sort of variable which will define *b* during execution, but substantial problems still exist. A value for *b* must be determined before each call to the procedure and, more importantly, *b* is a global variable which must be accessed through the use of the binding rule. A better option is to allow the production of a local variable when the procedure is created. Then no declaration of this variable need be made in the Main program, its value is used only in the procedure and need not be defined otherwise. Such flexibility is provided by using procedures with value parameters.

The program below provides this capability.

```
program Main (input, output);
var m : integer;

procedure NL (b : integer);
var k : integer;
begin
```

```
    for k := 1 to b do
      writeln;
end;

begin {Main}
    ...
    NL (m);
    ...
    NL (m+3);
end.
```

Figure 7 shows the program when the call *NL (m)* is about to be executed. We assume that the value of m is 5.

In this example, the parameter for *NL* is called by value. Quantities called by value are treated as initialized local variables within the instance. Value parameter binding can be regarded as the assignment of the value in the argument to the local parameter within the instance or

`local_par := argument`

Therefore, when the instance is created, a scan of the heading for parameters indicates a value parameter, *m*. In producing the declarations for the instance, *b* is assigned the value of *m*, thus producing a local value for use in the instance so that the original value of *m* is used only to assign the value to the local variable in the instance. The arguments to a procedure are evaluated in the context of the call.

Thus five lines are inserted in the output, the procedure is erased, and sequence control returns to the next instruction in the program. When *NL* is called again, the argument is a variable but to an expression. In forming the declaration, the value of m+3 is evaluated in the calling instance and the value 8 is assigned to *b*. Arguments called by value may be general expressions. They are always evaluated to a single value on procedure entry.

The important point is that the use of value parameters in the procedure has reduced the amount code and increased flexibility for the programmer. There is no longer a need for a global variable to control the amount of space, or the requirement that the variable be reset specifically within the code. The options for providing a value for the procedure are dramatically increased and the need for spurious variables eliminated. Parameters which are called by value result in an initialized local variable. If *m* has a value of 5 and a call is made to *NL(m)*, the value of 5 is assigned to the local variable *b*. All sorts of changes can be made to *b* but none of these changes will be made to *m*. The value of *b* vanishes when the procedure is terminated but the value of *m* does not.

3.3 Procedures with Reference Parameters

The program below is intended to swap the values of x and y.

```
program Main (output);
var x, y : real;

procedure swap (r, s : real);
var t : real;
```

```
begin
   t := r;
   r := s;
   s := t;
end;

begin {Main}
   x := 2.0;
   y := 3.0;
   swap (x, y);
   writeln (x, y :4:1);
end.
```

On procedure entry, the parameters called by value, r and s, are initialized to 2.0 and 3.0 respectively. Since *t* is a local variable it will be initialized to an undefined value. Thereafter, *r* and *s* are treated just like the local variable *t*. Immediately prior to the output statement, t = 2.0, s = 2.0 and r = 3.0 and the values have been swapped inside of the procedure. However, on procedure exit, all the local values created by the procedure vanish including the swapped pair. Obviously, a different mechanism must be used to enable the values which were swapped in the procedure to also be available when the procedure no longer exists. This is accomplished by calling the values by reference rather than by value. The program below shows the MODIFICATION required.

```
program Main (output);
var x, y : real;

procedure swap (var r, s : real);
var t : real;
begin
   t := r;
   r := s;
   s := t;
end;

begin {Main}
   x := 2.0;
   y := 3.0;
   swap (x, y);
   writeln (x, y :4:1);
end.
```

Figure 8(a) shows the instance structure when the *swap* procedure is created and Figure 8(b) the instance structure after *swap* has completed.

While the effect of calling by reference is radically different from calling by value, the process involved in creating the instance is only slightly different. When a value was called by value, a local variable was created which contained the appropriate value for the called variable. This process was used in the absence of any quantifier before the parameters in the heading. In this case the reserved word *var* before the parameters indicates that these are **called by reference**. In creating the quantities in the declaration, the local variables are given an address to use rather than setting aside space for the storage of the quantity within the instance. Reference parameter binding can then be regarded as assigning the address of the argument to the parameter, or,

```
parameter := address_of_argument
```

Assigning an address rather than a value is shown with the symbol @. Thus, the local variable still has to obtain a value which is done by referencing the address of x which essentially has no effect on x but it does give t a value which is stored within the instance. When new values are assigned to variables within the procedure instance which are referenced by address the values are stored at the address indicated and not within the procedure instance itself. This provides the capability (and protection) of altering global variables in a controlled fashion since quantities are accessible to the procedure which exist outside of the procedure instance. Had these variables been used as global variables within the procedure, the code would be specific to those variables and access would have been available through application of the binding rule. Specifying the values as reference parameters thus provides controlled access to the variables outside of the scope of the binding rule, and provides flexibility which could not be realized using global variables in the code. When the procedure terminates, reference to the address outside of the instance disappears, local values also disappear, but the changes made to values outside the instance remain.

4 Functions

Functions operate much like procedures but require additional qualifications. Resources are specified in a formal parameter list and in this respect are handled in the same way as formal parameters are handled in procedures. Functions differ from procedures in two ways. First, in the declaration of the instance, storage for the value produced by the function must also be allocated since functions do return a single value and this quantity must be available. The type declaration for the function is a signal that such storage is to be allocated. Second, the return address causes sequence control to come back to the instruction which called the function in order to complete the instruction rather than moving to the instruction after the call as is the case in procedures. During implementation of the function, the position of the function identifier in the code affects what is done. If the function name appears on the left-hand side of the assignment statement, then this indicates that the value of the function which has been determined is to be placed in the appropriate storage location. It is good programming practice to assign a value to the function once only and use local variables to hold intermediate values if this is necessary because the computation is complex (and non-recursive). Whenever the function name appears as an expression, it is essentially another call to the function. The following example will illustrate the operation of functions.

```
program Main (output);
var c : integer;

function factorial (n : integer) : integer;
var  count, ans : integer;
begin
   ans := 1;
   for count := 2 to n do
```

```
        ans := ans * count;
        factorial := ans;
    end;

    begin
       c := 3;
       c := factorial (c);
       writeln (c:10);
    end.
```

Figure 9 shows the instance diagram at the point where the value is assigned to the function.

When the function is called, the instance is created. In the function instance, because the parameter is a call-by-value parameter, the local variable n is assigned the value currently held by *c* which is accessed through SL. When the function is assigned the value of *ans*, SL is used to find the position of the declaration of the function and stores the value in the storage allocated in the declaration in *Main.* Sequence control then moves to the *end* statement, the instance is erased, and sequence control positioned back at the instruction which called the function.

5 Recursion

As has been shown, execution of Pascal programs proceeds through a process of creating and disposing of instances as they are needed. It is therefore possible for a procedure (or instance) to call itself since the format for the creation of the instance exists outside of the calling statement. This recursive process is often used (and mis-used) in programs. More important, recursive constructs are often difficult to understand and certainly difficult to debug when they function incorrectly. Using the methodology discussed in this paper can assist in understanding recursion.

As an example, suppose that an algorithm is being designed to accept a base 10 integer and covert it to the equivalent binary representation. A method which is often used is to repeatedly divide by 2 until the quotient is zero while accumulating the remainders as shown below using 13 for the base 10 integer.

```
2 | 13   remainder 1
  ----
2 |  6   remainder 0
  ----
2 |  3   remainder 1
  ----
2 |  1   remainder 1
  ----
     0
```

If the remainders are concatenated in reverse order the binary equivalent is produced. Thus reading the remainders from bottom to top and concatenating them yields 1101 which is the binary equivalent of 13.

The whole process can be viewed as follows. A master routine is written to print out the binary equivalent for 13. Such a routine can be restated so that a routine is called which can print out the binary equivalent for 6 (the quotient) and when this is completed a 1 (the remainder) will be appended. The task which determines the binary equivalent of 6 will initiate a task which will print the binary equivalent of 3 and which will append a 0 when it is complete. Finally, another task can be used to print the binary equivalent of 1 and which will append a 1 when completed. This task just prints out a 1 and then the chain of commands can be unwound.

The program below implements this algorithm recursively to find the binary equivalent of 6.

```
program convert (input, output);
procedure cvb (n : integer);
var q, r : integer;
begin
   q := n div 2;
   r := n mod 2;
   if q > 0 then
        cvb(q);
   write(r : 1);
end;

begin
   cvb (6);
end.
```

The diagram showing the execution at the point where the program has grown to its fullest extent is shown in Figure 10.

The important point is that each slave's task is structured exactly as its master. The problem at each step is simpler in that each time the parameter is halved until eventually the job is so simple that it can be carried out at once.

6 Arrays and Procedures as Parameters

6.1 Arrays

The search for matches, however, requires a specific declaration structure. The search for a match occurs by looking for a listing of the quantities within the appropriate type in a declaration instance. Thus *type* declarations serve to simplify the search. In addition, Pascal requires that any formal parameter be specified as a single type. This has particular relevance for the use of arrays as formal parameters.

Arrays can be defined in a *var* statement such as

```
x : array [1 .. 10]  of integer
y : array [1 .. 10]  of integer
```

Such declarations of structured types are a composite of identifier type and index type, or two types. While such declarations can be used, the usage is limited to global scope or local declarations. In order to use the arrays above as formal parameters in procedures (or functions), they must be redefined as a single new type as shown below:

```
type group = array [1 .. 10] of integer
```

and then declaring *x* and *y* as being of type *group*, the equivalence of the structure of the two arrays is immediately defined as a single type.

The program below creates a new type called *vec* and contains calls on routines *readvec* and *writevec* which read in and write out typical vectors.

```
program norm (input, output);

type vec = array [1..10] of real;
var v, w : vec;

procedure readvec (var x : vec; n : integer);
var k : integer;
begin
   for k := 1 to n do
      read(x[k]);
end;

procedure writevec (var x : vec; n : integer);
var k : integer;
begin
   for k := 1 to n do
      writeln(k, x[k]);
end;

begin
   readvec (v, 10);
   writevec (v, 10;
   readvec (w, 10);
   writevec (w, 10);
end.
```

Figure 11 shows the instance structure at the first call on *writevec*, when the loop has been entered and the values of *k* and *x[k]* are about to be printed. In this case, *v* is passed by reference and *n* is passed by value making local *n* 10. This means that the formal parameter, *x* is intialized to the same value as that in the appropriate value for the vector *v* which is accessed by address from the operative procedure so *x* aliases global *v* during execution of the procedure body. Whenever *x[k]* is written, it is the value of global *v[k]* that is output. On the second call on *writevec*, *x* will point to the other vector *w* and its contents will be output.

If the procedure heading for *writevec* had called *x* by value rather than by reference, then the call to print *v* would have resulted in a local copy of the vector within the procedure instance. While the results would appear the same to the user, the impact on the machine is a duplicate copy of the vector. Modifying the *readvec* procedure so that the *x* is called by value would not however provide the same results for the user since the information would have been read into the local storage for the vector and when the procedure terminated the values would be lost.

6.2 Procedures

Thus far, we have dealt only with two types of parameters normally used by most Pascal programmers – value and reference parameters. A third type, those called by name also form a part Pascal (and some other high level languages) and the model can assist in understanding and tracing the execution when these parameters are used.

When a procedures is used in a parameter list, what we essentially do is pass a process to the current procedure. Any procedure which exists as a parameter is considered as a reference parameter so that it inherits the environment of the procedure which includes the parameter. This is necessary since the desired outcome is to use the procedure as an additional process within the procedure which calls it. Since no procedure exists until it is actually needed, what we do by using a procedure as a parameter is to create an environment which includes all of the information that is needed to execute (and thus produce the procedure instance). This is then used when the instance is actually created.

Consider the following program which uses procedures as parameters. (This is a modification from van Eijk (1986)). The program builds a binary tree and then prints out all of the paths through the tree. For brevity, text is given omitting the code required to build the tree.

```
program Main (input, output);
type tree = ^node;
   node = record
     item : integer;
     left, right : tree;
  end;
var root : tree;
procedure paths (t : tree; procedure path);
procedure localpath;
begin {localpath}
   path;
   writeln(t^.item:3);
end; {localpath}

begin {paths}
   if t <> nil then
   begin {then}
      if (t^.left = nil) and (t^.right = nil) then localpath;
      paths(t^.left, localpath);
      paths(t^.right, localpath);
   end; {then}
end;

procedure dummy;
begin
   writeln('  ');
end;
```

```
begin {Main}
   {set up tree}
   paths(root, dummy);
end.
```

The tree used in the execution is the left-heavy binary structure with 5 nodes shown in each of Figures 12 through 16.

Figure 12 shows program execution when the tree has been established and the first call to *paths* is about to be made. Once the instance of Main has been initialised and its DL and SL values set, program execution starts. In our case, we execute the informal actions *set up tree*, which we assume creates the tree shown and assign *root* to reference it. Figure 12 shows *root* pointing to the root of the tree (the node numbered 1) with a dotted line.

The next action carried out by *Main* is a call to *paths.* This call results in the creation of a second instance, named *paths{1}*, as shown in Figure 13. For textual clarity, we have simplified the body (Pascal code) of paths{1} by eliminating the conditional code and leaving only the actions that will be executed. DL of paths{1} points to its return position, the instruction in Main following the call to *path.* The template for paths lies within Main, hence the SL of paths{1} is directed there. The declaration portion of paths{1} is constructed from its template's parameter list and local definitions. The first argument, *root*, is a value parameter and is assigned to *t* by a copy operation. From now on, *t* is treated as though it were just another local variable.

The second argument *dummy* is a procedure passed by reference. Here the parameter path receives a descriptor which tells it how to call dummy correctly. This is essentially the same as treating the procedure as a reference parameter and therefore it is assigned the address of the procedure which appears as the argument.

A reference type parameter will be shown as a dotted line leading to its argument in the same way as reference parameters were shown previously. When we call a procedure passed as parameter, we will need to know its entry point and its operating environment (its SL) in order to make the call correctly.

paths{1} also contains a local procedure, *localpath{#1}* and some code. The process of assigning the contents to the local declarations uses the binding rule and follows SL to locate the information required to do this assignment. Therefore, *dummy* retains the environment of *Main* and all other accesses are via SL to *Main* which contains the definition of the procedure required to build the instance. Note that no instance of a procedure is created until the procedure is called, only the definitional information exists. Placing the procedure in the parameter list is not a call to that procedure, it is a process that identifies the environment and location of the task.

Before we carry out the given instructions in the body of paths1, we insert an extra call on *localpath* in order to give a feel for the purpose of this routine, and to cast some light on how we give meanings to quantities. This call would result in an instance *localpath{#0}* being created (#0 being used to distiguish this digression from the original) and its body being entered. Its code is

```
path;
writeln(t^.item : 3);
```

path results in a call on the parameter to paths{1} being called, an instance *dummy* is created, at which time the situation is as depicted in Figure 14. Notice the SL for localpath{0} is paths{1} for that is where localpaths{0} is both called and defined. The SL for dummy{0} is to Main (that is where the template for dummy is defined). We now have enough variety in our display to enable us to give the binding rule for quantities in its full generality.

> Given the operating instance I and the occurrence of a name x. Look in the declaration part of I, if you find a quantity x defined, then accept that definition. If no match is found, repeat from the SL of I.

Figure 15 shows the next situation when dummy has been called, done its thing (nothing), and been deleted. We return to localpaths0. Previously, *paths* was found by following one SL and matching with the parameter local to paths{1}. From here the binding rule locates *writeln* in the RTS, and prints out the value of *t.item* which is found via the binding rule in this instance.

Our digression is now over and we return to the execution of the program proper inside paths1. Since the first node is not *nil* and not a leaf, the actions in the body collapse to

```
paths(t^.left, localpath);
paths(t^.right, localpath);
```

so the next instruction is executed which is another call to paths and the instance *paths{2}* for node #2 is created. The DL refers back to the calling statement, and its SL points back to Main. The arguments to paths{2} are evaluated in the context of the call, that is, inside paths{1}. Thus when we establish paths{2}, its t-parameter points to node #2, the left link of the t of path{1}, and its path-parameter to the definition of *localpath{1}* belonging to paths{1}. paths{2} is now entered and again the if-then infrastructure drops away leaving us with two calls on paths to execute. So instance *paths{3}* is created, its links established, and its body entered. This instance focusses via its parameter t on node #3 which is a leaf node. Both the left and right pointers of its t are nil, thus here we carry out three actions.

```
localpath;
paths(t^.left, localpath);
paths(t^.right, localpath);
```

The total extent of the calls required to process the first three nodes is shown in Figure 16, where each call to *localpath* is numbered in order to facilitate explanation.

The binding rule maps the call onto localpath{3}, local to the operating instance. When instantiated, localpaths{3}, will have its DL and its SL referring back to paths{3}. The body of localpath{3} first makes a call upon path which is non-local. The binding rule tells us to look first in paths{3}, where we find the match. It is to localpaths{2} in paths{2}. So another instance, *localpath{2}*, is created. Its DL is to localpath{3}, its SL is to paths{2}, and its first action is to call path. Again no match can be found locally. The non-local match this time is found in paths{2} and it is to *localpath{1}* in paths. Continuing in this way, we eventually arrive at the most stretched out situation which is shown in Figure 17.

The *dummy* instance is executed, reaches its *end* statement, is then removed and sequence control returns to the instruction designated by DL. The contents of the root node, #1, are written, the instance removed, and then the contents of node #2 written. This continues until all the nodes on the path from the root to node #3 are written.

We have thus achieved what we set out to do which is print the ancestors of leaf node #3 in order of their longevity. This is accomplished by constructing a chain of localpath instances, each one focusing on a specific ancestor node in the tree. Each localpath instance prints its ancestors first before detailing its own information through *writeln(t^.item).* The initialisation of localpath in paths{1} to dummy is a way of limiting the depth of the recursion adopted by the programmer.

Finally, control returns to the third node and execution continues.

7 Summary and Future Directions

We have presented a model which uses the creation and deletion of block instances as a means of visualizing the execution of a program. Beginning with the basic components of the model, these have been expanded to show critical features such as scope of variables, parameter passing, recursion , and the execution of more complex procedures which use procedures as parameters. By using the concepts of static and dynamic links and the binding rule, the model not only provides insights into the reasons for certain occurrences of specific coutcomes in programs, it also makes it possible to trace the execution of more abstract features such as recursion and complex parameters in procedures. The model allows the user to understand the execution in terms of source code and blocks instances which mimic what happens at a much lower level within the machine.

The model has immediate utility as a paper and pencil model since it provides a means of visually describing difficult concepts such as scope, parameter passing (call by reference, value, and name), and recursion. These can be difficult concepts to understand and the subtleties easily lost in verbal explanation (see Slonneger (1987), Krumme (1987)). In addition, it serves as a good introduction to activation records, level, scoping, and binding rules which are required for compilers and linkers.

The model also offers a paradigm for building a program visualization system. Building an initial prototype of such a system is currently under way. This system displays the code and the block instances and allows the use to step through the execution continuously or on command. In addition, it allows the user to step backward through the execution. Such a system is considered to be most useful in helping novice programmers with Pascal. However, monitoring program execution and debugging programs is a constant process for all programmers. A decade ago, Plattner and Nievergelt (1981:91) stated that:

> Monitoring program execution takes up a considerable fraction of a programmer's time; it is unlilkely to decrease in importance, since the insight it can provide is complementary to, and cannot be replaced by, that obtained from static analysis of program texts.

There has been substantial interest recently in providing these graphic interfaces (Meyers, 1989)) and future components of the the current prototype will focus on the experienced programmer and attempt to use the system as a viable debugging tool with editor and compiler interface.

Finally, the model also has the potential for application to other imperative languages such as Modula-2 and C and can be extended further to include coroutines and garbage collection issues.

8 References

Birtwistle, G., Dahl, O-J., Myrhang, B., and Nygaard, K. (1982) *Simula Begin.* Sweden : Student Litteraur.

Krumme, David W. (1987) Comments on an Example for Procedure Parameters. *SIGPLAN*

Notices, 22(6), 109-111.

Myers, B.A. (1989) The state of the art in visual programming and program visualization. In A. Kilgour & R. Earnshaw (Eds.), *Graphics Tools for Software Engineers.* Cambridge : Cambridge University Press.

Plattner, B. & Nievergelt, J. (1981). Monitoring Program Execution: A Survey. *IEEE Computer, 14(11),* 76-93.

Slonneger, Ken (1987) Pitfalls with Procedure Parameters. *SIGPLAN Notices, 22(2)*, 95-99.

van Eijk, Peter (1986) A Useful Application of Formal Procedure Parameters. *SIGPLAN Notices, 21(9).*

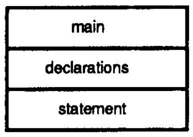


Figure 1

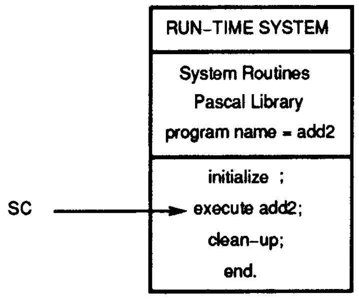
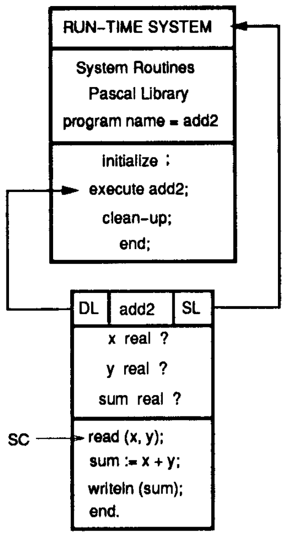
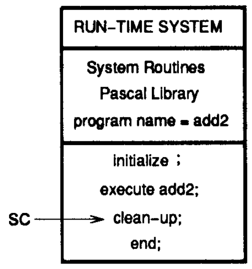


Figure 2



(a)



(b)

Figure 3

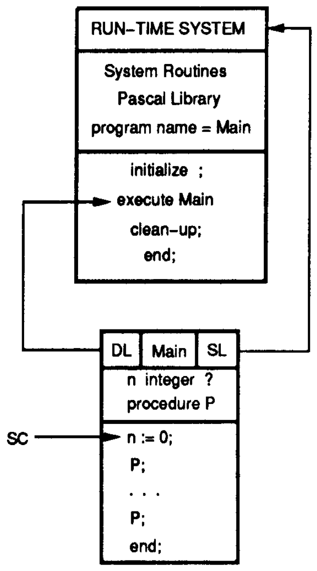


Figure 4

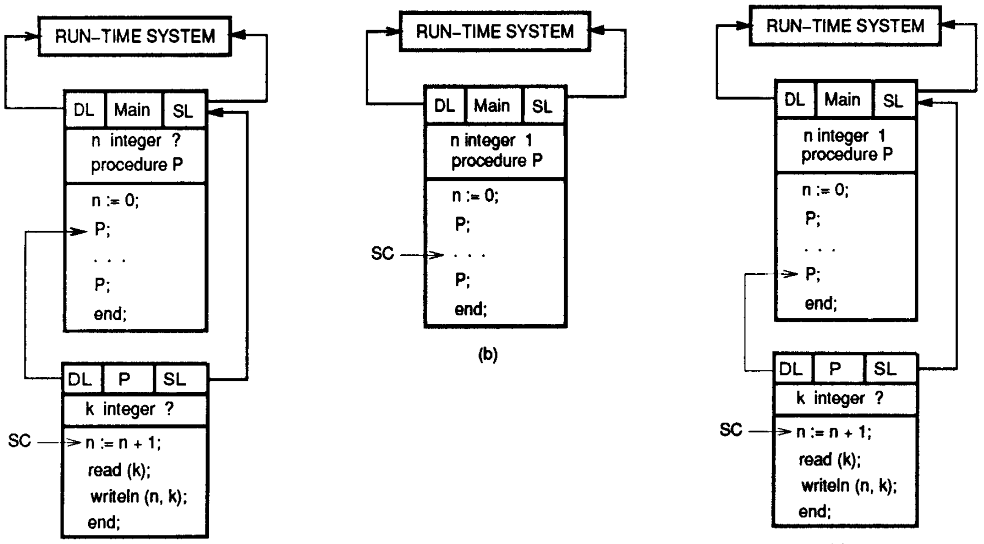
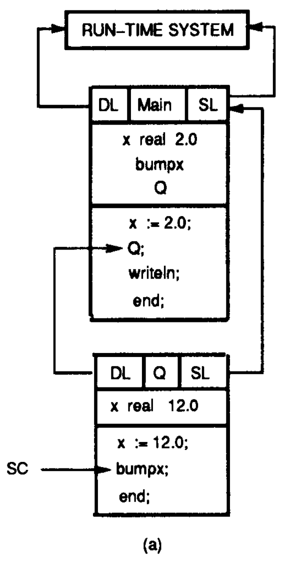


Figure 5



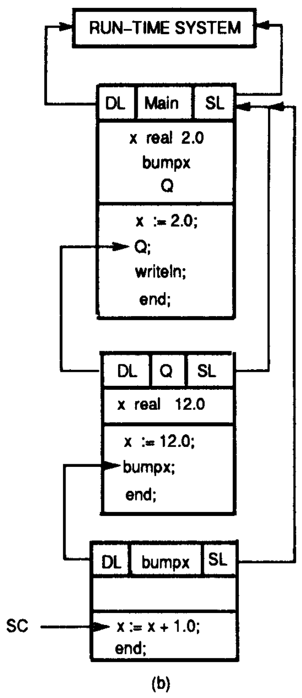


Figure 6

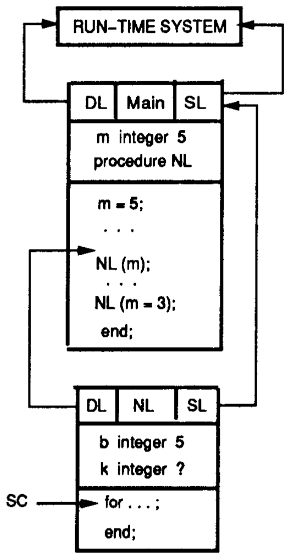


Figure 7

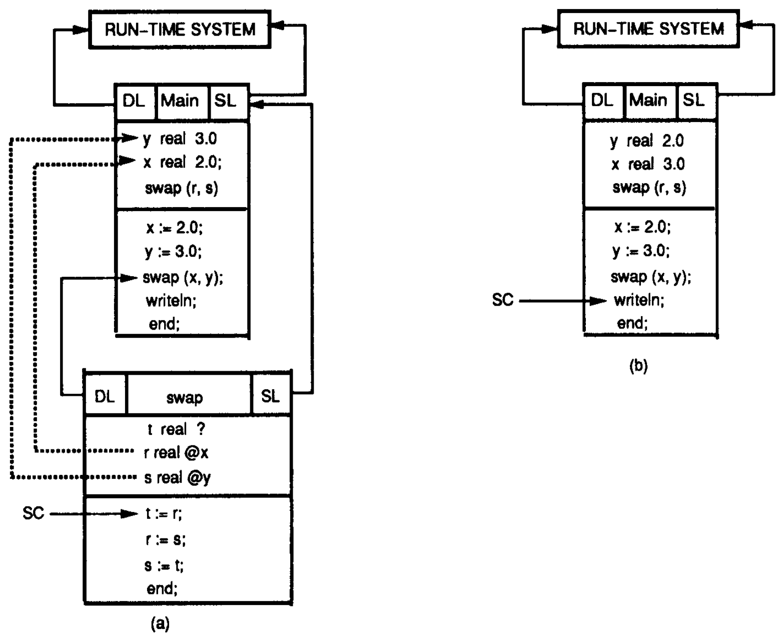


Figure 8

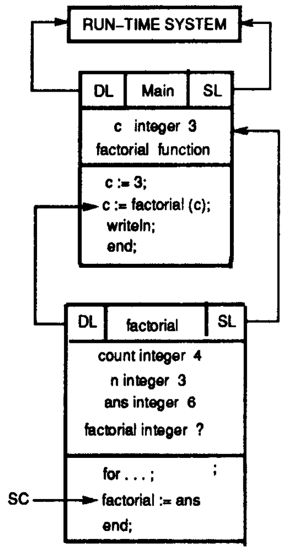
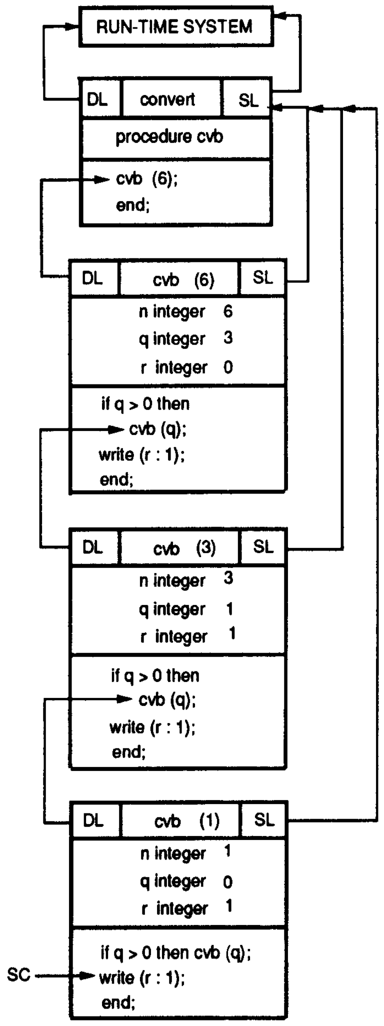


Figure 9



RUN-TIME SYSTEM
DL
convert
SL
procedure cvb
cvb (6);
end;
DL
cvb (6)
SL
n integer 6
q integer 3
r integer 0
if q > 0 then
cvb (q);
write (r : 1);
end;
DL
cvb (3)
SL
n integer 3
q integer 1
r integer 1
if q > 0 then
cvb (q);
write (r : 1);
end;
DL
cvb (1)
SL
n integer 1
q integer 0
r integer 1
if q > 0 then cvb (q);
SC
write (r : 1);
end;

Figure 10

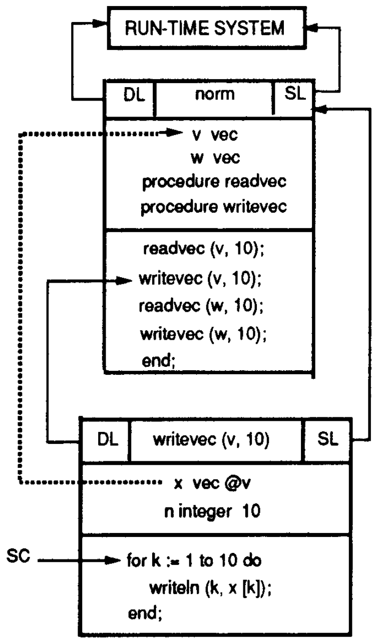


Figure 11

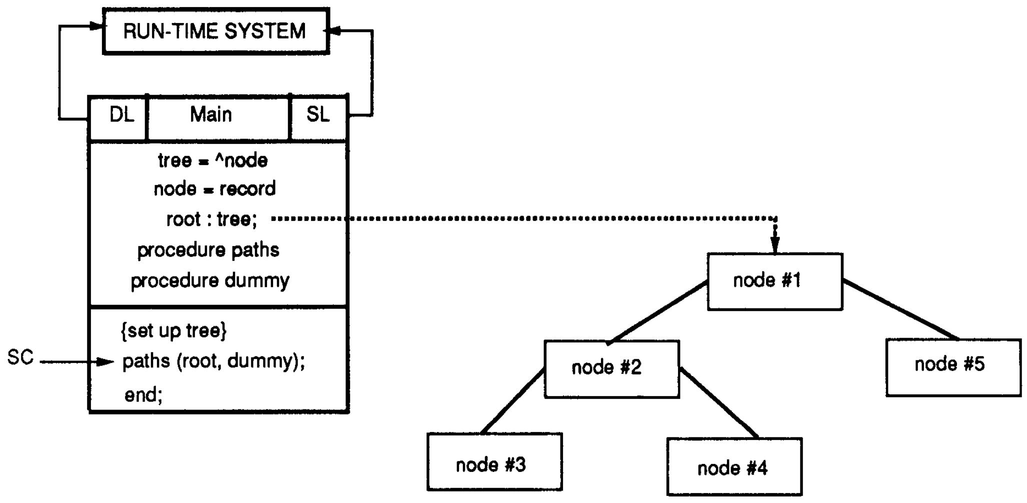


Figure 12

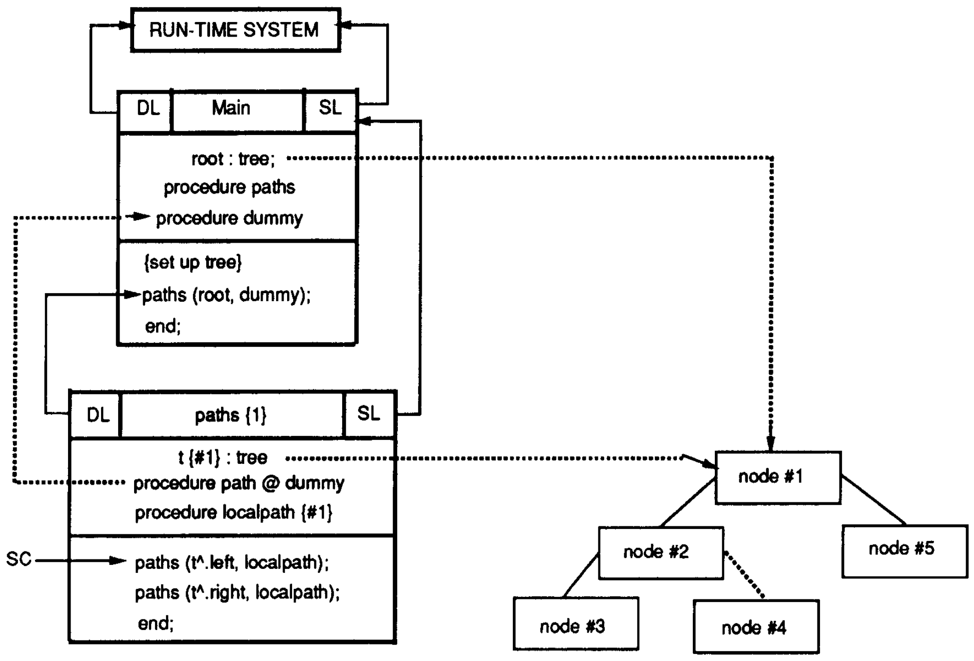

RUN-TIME SYSTEM
DL
Main
SL
root : tree;
procedure paths
procedure dummy
{set up tree}
paths (root, dummy);
end;
DL
paths {1}
SL
t {#1} : tree
procedure path @ dummy
procedure localpath {#1}
SC
paths (t^.left, localpath);
paths (t^.right, localpath);
end;
node #1
node #2
node #5
node #3
node #4

Figure 13

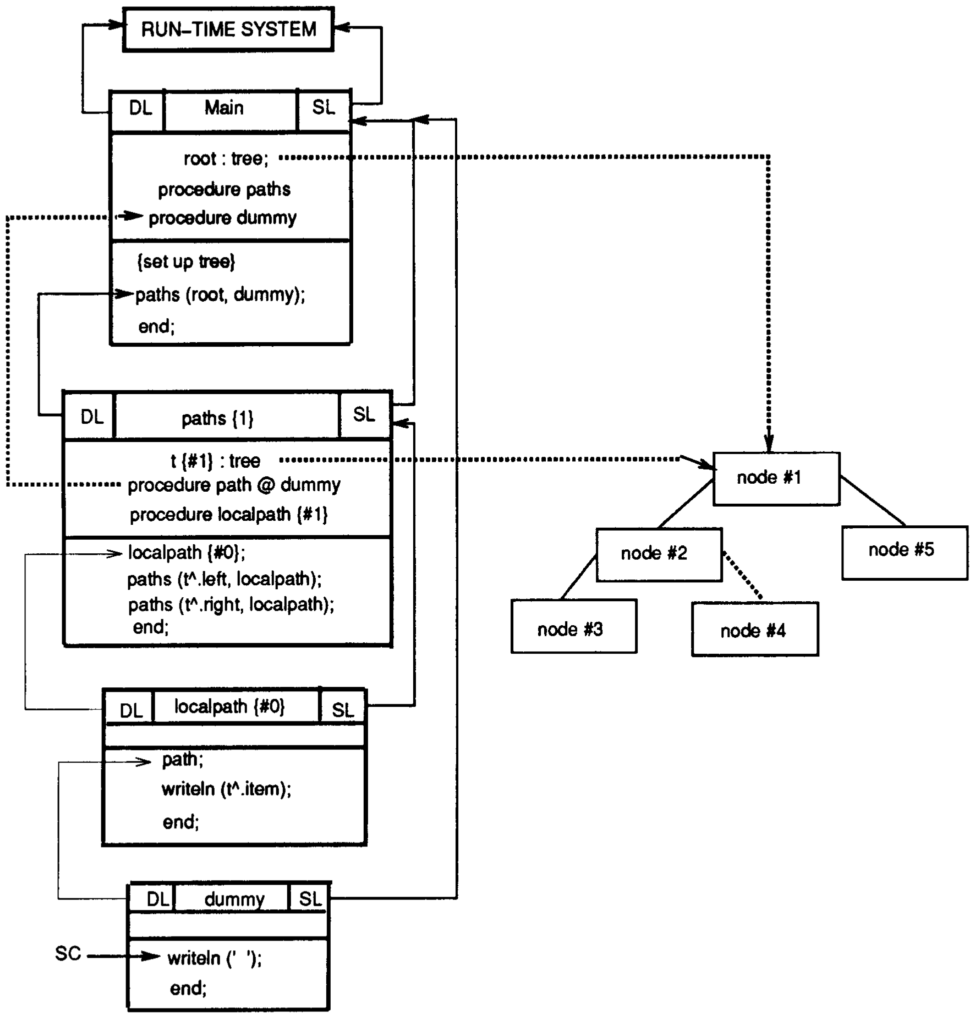

RUN–TIME SYSTEM
DL
Main
SL
root : tree;
procedure paths
procedure dummy
{set up tree}
paths (root, dummy);
end;
DL
paths {1}
SL
t {#1} : tree
procedure path @ dummy
procedure localpath {#1}
localpath {#0};
paths (t^.left, localpath);
paths (t^.right, localpath);
end;
DL
localpath {#0}
SL
path;
writeln (t^.item);
end;
DL
dummy
SL
SC
writeln (' ');
end;
node #1
node #2
node #5
node #3
node #4

Figure 14

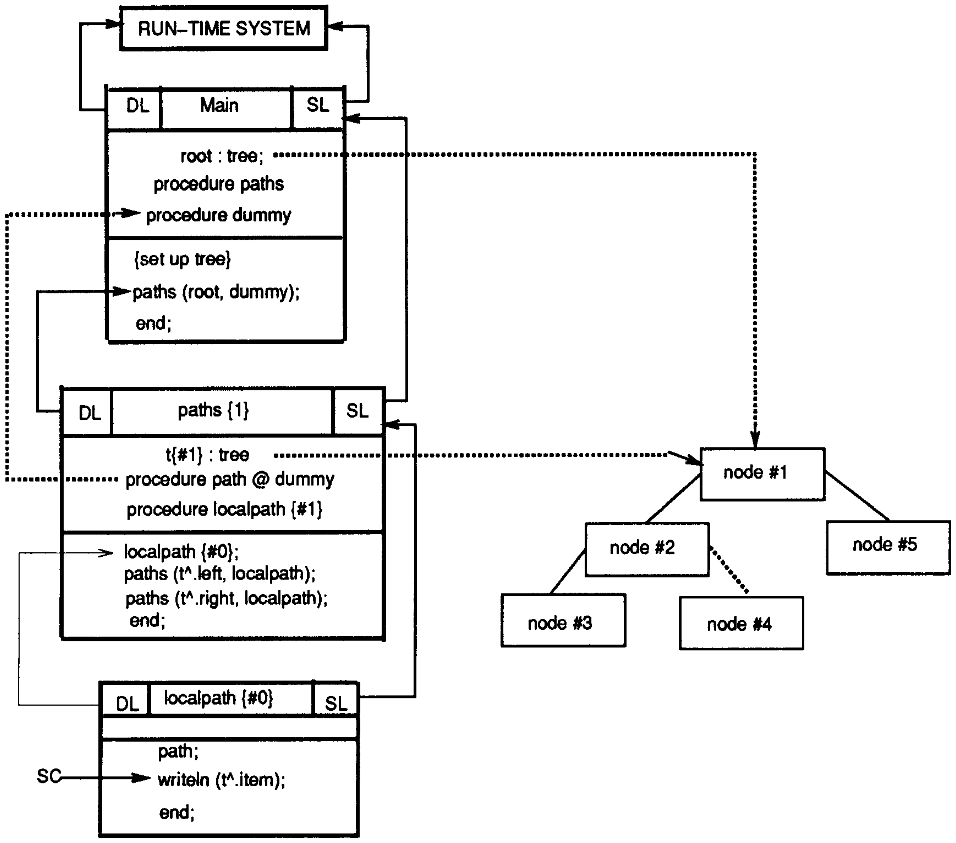

RUN-TIME SYSTEM
DL
Main
SL
root : tree;
procedure paths
procedure dummy
{set up tree}
paths (root, dummy);
end;
DL
paths {1}
SL
t{#1} : tree
procedure path @ dummy
procedure localpath {#1}
localpath {#0};
paths (t^.left, localpath);
paths (t^.right, localpath);
end;
DL
localpath {#0}
SL
path;
SC
writeln (t^.item);
end;
node #1
node #2
node #5
node #3
node #4

Figure 15

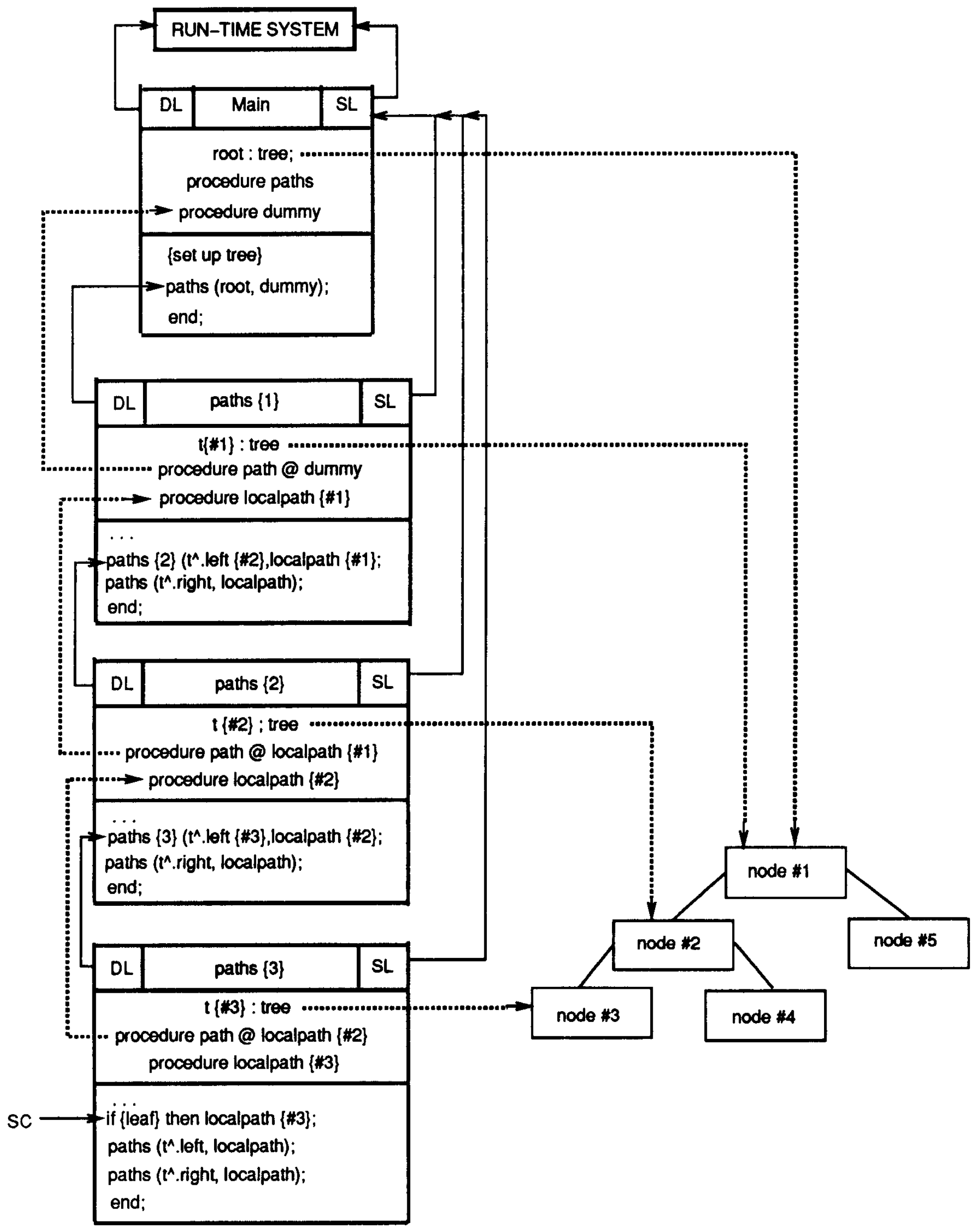
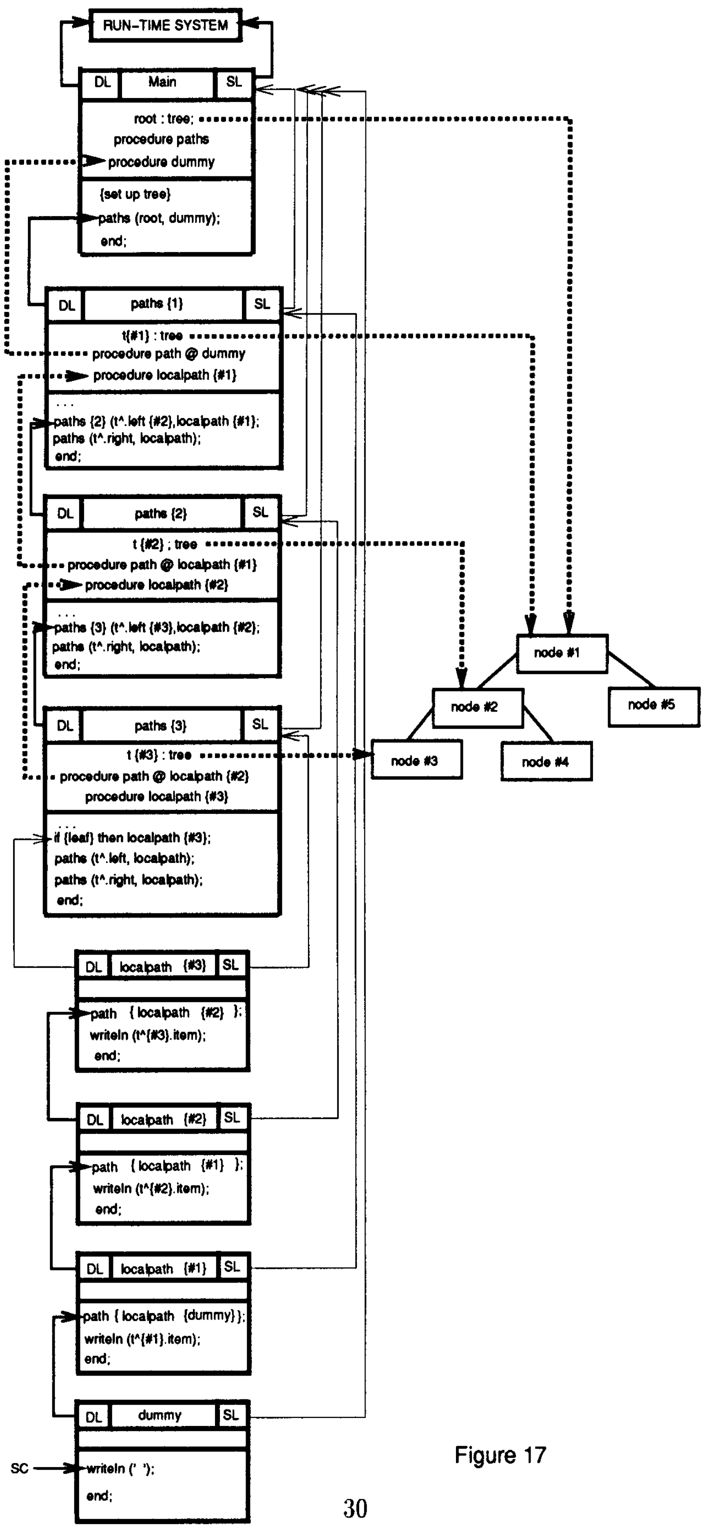

RUN-TIME SYSTEM
DL
Main
SL
root : tree;
procedure paths
procedure dummy
{set up tree}
paths (root, dummy);
end;
DL
paths {1}
SL
t{#1} : tree
procedure path @ dummy
procedure localpath {#1}
. . .
paths {2} (t^.left {#2},localpath {#1};
paths (t^.right, localpath);
end;
DL
paths {2}
SL
t {#2} ; tree
procedure path @ localpath {#1}
procedure localpath {#2}
. . .
paths {3} (t^.left {#3},localpath {#2};
paths (t^.right, localpath);
end;
DL
paths {3}
SL
t {#3} : tree
procedure path @ localpath {#2}
procedure localpath {#3}
. . .
SC
if {leaf} then localpath {#3};
paths (t^.left, localpath);
paths (t^.right, localpath);
end;
node #1
node #2
node #5
node #3
node #4

Figure 16



```
Main
    root : tree;
    procedure paths
    procedure dummy
    {set up tree}
    paths (root, dummy);
    end;

paths {1}
    t{#1} : tree
    procedure path @ dummy
    procedure localpath {#1}
    . . .
    paths {2} (t^.left {#2},localpath {#1};
    paths (t^.right, localpath);
    end;

paths {2}
    t {#2} ; tree
    procedure path @ localpath {#1}
    procedure localpath {#2}
    . . .
    paths {3} (t^.left {#3},localpath {#2};
    paths (t^.right, localpath);
    end;

paths {3}
    t {#3} : tree
    procedure path @ localpath {#2}
    procedure localpath {#3}
    . . .
    if {leaf} then localpath {#3};
    paths (t^.left, localpath);
    paths (t^.right, localpath);
    end;

localpath {#3}
    path { localpath {#2} };
    writeln (t^{#3}.item);
    end;

localpath {#2}
    path { localpath {#1} };
    writeln (t^{#2}.item);
    end;

localpath {#1}
    path { localpath {dummy}};
    writeln (t^{#1}.item);
    end;

dummy
    writeln (' ');
    end;
```

Figure 17