Traversing HyperPascal’s semantic Tree

Introduction
This document contains some suggestions about how the HyperPascal interpreter should, or could work. The basic model is that the interpreter is a recursive subroutine that traverses a tree representation of the HyperPascal program. The nodes in the tree correspond to the various elements of the program: actions (assignment and subroutine invocation actions), control icons (choices and loops), subroutine headers, and so on. At each node in the tree, the interpreter determines the type of the node, and acts accordingly. At choice nodes, for example, it evaluates Boolean expressions - these are stored in subtrees of the choice node - and, according to the result returned by the Boolean expression, either does or does not, evaluate another subtree, corresponding to the “then” clause. This tree is called the Semantic Tree, because it specifies the semantic meaning of the program icons. Each subroutine in the HyperPascal program has its own Semantic Tree, and all the Semantic Trees are linked into another, higher-level structure called the Scope Tree that has one node for each subroutine in the program, linked into a tree representation of the program’s subroutine hierarchy. Before discussing the Semantic Tree in which the interpreter spends most of its time, we begin with a description of the Scope Tree.

The Scope Tree
The HyperPascal interpreter gets its information about the program from the Scope Tree, as the subroutine nodes in the Scope Tree are organised to represent the program’s static subroutine structure, and each of the subroutine nodes points to a tree of code for that subroutine. Thus the Scope Tree contains a representation of the whole HyperPascal program. The root node of the Scope Tree is the node for the main program of the HyperPascal program, and it therefore points to the the semantic tree for the main program. The interpreter is initialised to point at the root node of the Scope Tree and it follows that pointer and starts interpreting the main program.

Structure of the Scope Tree.
The Semantic Tree contains code for a program unit - a procedure, a function, or the main program. When control is passed from one program unit to another, information about the location of its Semantic Tree, the names and storage locations and so on of its variables must be made available to the interpreter. The Scope Tree contains this information, and each program unit has a node in the Scope Tree. Therefore, before we consider the Semantic Tree in detail, let us deal with the structure of the Scope Tree.

Subroutine nodes have the following structure
so after a program with four top-level subroutines, two second-level subroutines, and one third-level subroutine has been constructed, the overall structure of the Scope Tree could look like this:

This structure (called a Trie) allows a binary tree to represent an n-ary tree, that is a tree with an arbitrary number of nodes at each level.

The top left corner of each of these nodes is used for information that belongs to a particular subroutine. This could be implemented as a record nested inside the subroutine node. It needs to contain static information such as the subroutine name, and also provide a way of accessing dynamic information such as the current execution point in the subroutine.

The record containing subroutine information in a Scope Tree node looks like this:
Subroutine information

<table>
<thead>
<tr>
<th>name</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>called by</td>
<td>stack of calling locations</td>
</tr>
<tr>
<td>code location</td>
<td>semantic tree node</td>
</tr>
<tr>
<td>input parameters</td>
<td>list of storage handle stacks</td>
</tr>
<tr>
<td>output parameters</td>
<td>list of storage handle stacks</td>
</tr>
<tr>
<td>writeback location</td>
<td>list of storage handle stacks</td>
</tr>
<tr>
<td>locals</td>
<td>list of storage handle stacks</td>
</tr>
<tr>
<td>function value</td>
<td>list of storage handle stacks</td>
</tr>
<tr>
<td>fn value writeback location</td>
<td>list of storage handle stacks</td>
</tr>
<tr>
<td>current location</td>
<td>stack of semantic tree nodes</td>
</tr>
</tbody>
</table>

For the moment, let us ignore the fact that the pointers in most of these fields point at lists of stacks, and assume that each field contains a pointer to a single list.

The called-by field and current location fields

Each time a subroutine is called, the location to which control should return after the subroutine has finished executing must be recorded. The called-by field contains this information.

Each subroutine node also contains a current-location field, which references the location in the Semantic Tree where the interpretation has reached. While the interpreter is interpreting the subroutine, it accesses this pointer to determine which part of the Semantic Tree to execute, and it updates the pointer as it traverses the Semantic Tree. *(NOTE: could this be dispensed with? It seems more sensible if the interpreter maintains its own current-location pointer (i.e. Program counter) and saves this into the subroutine information record as the return address when it calls another subroutine)*

So when a subroutine A calls a subroutine B, A does nothing to its current-location pointer, but the interpreter puts the address of A’s subroutine node into B’s called by field. While it is executing B, the interpreter access the Semantic Tree through B’s current-location pointer. Thus A’s current-location pointer is inchanged. When the interpreter returns from B to A, it looks at B’s called-by field, finds the location of A’s subroutine node, looks there for A’s current-location pointer, and start executing A again from that location.

Now let us consider the reason why this information is not stored in simple fields, but in stacks.

Subroutines are recursive. When a subroutine is called, it is necessary to remember the point that the first execution of the subroutine had reached, so that when the second
execution finishes, the first execution can resume from the right place. Thus, each time A calls itself, the interpreter will push the address of A’s subroutine node onto A’s called-by stack, and when it starts the next execution of A, it will push a pointer to the top of A’s Semantic Tree onto the top of the current-location stack. Then the interpreter can execute the n\textsuperscript{th} recursion of A without interfering with the location pointer of the n-1\textsuperscript{th} recursion of A. When the n\textsuperscript{th} recursion terminates, the interpreter cuts the current-location stack back by 1 item, and reveals the old current-location value. This can happen as many times as necessary.

**The code location field**
This is a pointer to the Semantic Tree for the subroutine. It does not need to be a stack, because there is only one copy of the code for the subroutine. Multiple accesses to the code - to enable recursion - are supported by a stack of current-location pointers, as explained above.

**The input parameters, output parameters, locals, and function value fields**
Input parameters, output parameters, and so on are all data items that are local to the subroutine. When the subroutine starts executing, storage has to be allocated for each of the items in these fields. The interpreter will call Delphi to allocate the appropriate amount of storage for each item when it is invoking the subroutine, and insert a handle for that item’s storage into the stack. The data structure is a list of stacks; it is a list to allow for an arbitrary number of input and output parameters; each item in the list points at a stack, because new local storage must be allocated each time the subroutine is invoked, so that the changing data values during the n\textsuperscript{th} iteration do not interfere with the data associated with the n-1\textsuperscript{th} iteration. Note that the function value has to be a stack, but it does not have to be an item in a list - a function can only have one value - but it is simpler to use the same data structure for the function value as for the other local storage.

Note that a consistent way of handling storage allocation and disposal, and accessing data items of arbitrary sizes, is essential for active templates as well as for subroutines.

**The writeback location and fn val writeback location fields**
When the subroutine has finished executing, the values it has calculated need to be transferred back to the calling environment. When the subroutine is invoked, the location of the actual output parameter is stored in the writeback location list. The function value may be used in an expression, so it will not necessarily be stored directly into a variable in the calling routine. In that case, the expression evaluator will have requested storage from Delphi for an operand in the expression, and the storage handle for that location will be writeback location in the called subroutine.

**The Semantic Tree**
The semantic tree for each subprogram (for the purposes of this discussion, the main program is a subprogram) is a data structure corresponding closely to the Action Tree which the user has assembled out of choice icons, loop icons, comment icons, and Action Sequence icons. The interpreter traverses this data structure, performing various actions
according to the type of node that it encounters. Details of the data structure and the actions are given below.

**Semantic Tree Headers**
A semantic tree header is a tool that allows a semantic tree node to reference an arbitrary number of subtrees. This is necessary because the user can add any number of children to the icons in the HyperPascal program. An action sequence icon, for example, can contain any number of assignment actions, each with its own semantic subtree. The subtrees are accessed from a list of semantic tree headers.

So a Semantic Tree Header (STH) is a node with a pointer to a semantic subtree, a pointer to another STH, and a field specifying the subtree type. There are five types of subtree: action, guard Boolean, go Boolean, stop Boolean, and control node.

Semantic tree header (STH)

![Semantic Tree Headers Diagram]

It may be necessary to include other fields in the STH, but this is the minimum required to support the behaviour described below.

**Action Sequences: Data structure**
When the user selects an action sequence icon from the oval menu, HyperPascal links an action sequence node into the semantic tree for the current program unit (subroutine or main program). The Action Sequence Node has a pointer field that can point to a Semantic Tree Header. Each time the user creates a new assignment action, the editor creates a semantic subtree for the assignment action, and a Semantic Tree Header for the subtree, and it links the semantic tree header into the list of semantic tree headers that the Action Tree Node points to.
**Action Sequences: Interpretation**

After the user has created a number of assignment actions, the action sequence node will point to a list of Semantic Tree Headers, each of which will point to a semantic subtree for an individual assignment action. When the interpreter reaches the action sequence node, it simply traverses the list of headers, evaluating the semantic subtrees they point to as it goes.

**Choice icons: Data Structure**

When the user selects a choice icon from the oval menu, HyperPascal links a choice node into the semantic tree for the current program unit (subroutine or main program). Each time the programmer adds a new choice to the choice icon, the editor adds *two* nodes to the list of semantic tree headers that the choice node points to. The first header is of type *guard Boolean*, and it points to a semantic tree for a Boolean expression. The second header can point to an action sequence or to another control node, depending on what type of icon the user selects as the subtree controlled by the guard Boolean.

**Choice icons: Interpretation**

When the interpreter reaches the header for the guard Boolean, it evaluates the associated semantic subtree. If the result is true, the interpreter evaluates the subtree associated with the next header in the list, and then stops traversing the list of Semantic Tree Headers. If the result is false, the interpreter omits the next Semantic Tree Header in the list and continues with the following STH (which should point to another guard Boolean), if there is one. Note that if the user leaves the box for a guard Boolean empty in the program, an STH for it should still be put into the list, and its null subtree should evaluate to True, to give the effect of a default *else* clause. And if the user creates a guard Boolean, but doesn’t hang a subtree off it (i.e., the piece of code that gets executed if the Boolean evaluates to True), the editor should still generate a header for the subtree in the list, so that the interpretation traversal will work correctly when it jumps an item in the list.
Loop icons
When the user selects a loop icon from the oval menu, the situation is a little more complex. The editor inserts a loop node into the semantic tree for the current program unit, as usual, but a loop node has pointers to three sets of linked lists.

One is a list of headers that correspond to the entry code. The interpreter evaluates the items in this list. They will usually be a single action sequence, but could be a more complex subtree.
The second linked list is a list of headers that correspond to the icons attached to the bottom of the loop icon. These include Go-Booleans, Stop-Booleans and subtrees\(^1\). When the interpreter encounters a header for a Go-Boolean, it executes the semantic subtree for the Go-Boolean, and if the result is True, the interpreter continues along the list. If the result is False, the interpreter ignores the rest of the list, and moves on to the third list associated with the loop icon (the exit code list). As with the choice icon, if the programmer has inserted a Go-Boolean icon, but omitted any actual Boolean expression, the result should be True.

When the interpreter encounters a header for a Stop-Boolean, it executes the semantic subtree for the Boolean, and if the result is False, the interpreter continues along the list. If the result is True, the interpreter ignores the remainder of the current list, and moves on to the third list associated with the loop icon (the exit code list). If the programmer has inserted a stopBoolean icon, but omitted any actual Boolean expression, the result should be True (so the loop should just terminate when the interpreter reaches the stop icon).

The third list is a list of headers that correspond to the exit code. The interpreter executes the items in this list.

**Subroutine invocations**

Subroutine invocations involve four things: getting data into the subroutine, getting data out of the subroutine, allocating data storage for the subroutine’s parameters and local variables, and transferring control into and out of the subroutine.

Some aspects of transferring control into and out of the subroutine - specifically, the way in which the location of the invocation is remembered and restored after the subroutine execution has finished. - have been described previously, as it was necessary to describe this activity to justify the structure of the subroutine node.

Transferring data into and out of the parameters is another important part of the subroutine invocation. HyperPascal parameters are passed by value. That is, a copy of the data is made, and the subroutine operates on that copy. So if the subroutine alters the parameter, that alteration is not visible from the outside. Output parameters (or function values) are used to get data values out of the subroutine. Just as the values of input parameters are copied into the subroutine at invocation time, so the values of output parameters are copied back into the calling environment when the subroutine exits.

So the invocation involves the following steps:

---

\(^1\) Note that in the diagram, the list contains STHs for a single goBoolean, a single stopBoolean, and a single controlNode. Of course, each of these items can appear any number of times in the list, and in any order. Headers for Action Sequences can also appear.
• The interpreter reaches the invocation and pushes the location of the current subroutine node onto the called-by stack in the called subroutine’s node.
• The interpreter traverses the input parameter, output parameter, local variable and function value lists, and requests Delphi to supply storage for each item in those lists. It pushes the item’s handle onto the item’s stack.
• The interpreter evaluates the input parameter expressions, and stores their results in the locations found in the previous step.
• The interpreter pushes handles for the actual output parameters onto the stacks in the writeback list.
• The interpreter starts executing the called subroutine by pushing its code-location onto its current-location and starting execution from that point.

Subroutine exit
Exiting from a subroutine is a similar process, in reverse. The interpreter reaches the end of the subroutine’s semantic tree, and
• copies the values of the formal output parameters and the function value, if any, to the locations of the actual output parameters (using the information in the writeback locations field) and function value.
• pops the top item off the stacks for: all local variables, all input parameters, all output parameters, the current location, and the function value
• pops the location of the calling subroutine off the called-by stack, and continues executing that subroutine. Since the current-location pointer in that subroutine still points to the location where the invocation occurred, it will start executing again at the right place.

Evaluating expressions
Expressions have a very well-defined format. They consist of a binary tree of operators with values (constants or variables) as the leaf nodes. Unary operators have one null subtree. Consequently, the expression evaluator can be a recursive function that applies the operator to the values that result from evaluating the left subtree and the right subtree, and returns the result. Delphi will therefore allocate temporary storage to two operands at each recursion.