RPL: A Mathematical Control Language

W.C. Wickes

Introduction

In 1984, a project was started at Hewlett-Packard Corvallis Division to develop a new software operating system to streamline calculator development and support a new generation of hardware and software. Previously, all HP calculators were implemented entirely in assembly language, a process that was becoming increasingly cumbersome and inefficient as the memory sizes of the calculators increased. The objectives for the new operating system were as follows:

- To provide execution control and memory management, including plug-in memory;
- To provide a programming language for rapid prototyping and application development;
- To support a variety of business and technical calculators;
- To execute identically out of RAM and ROM;
- To minimize memory use, especially RAM;
- To be transportable to various CPU’s;
- To be extensible; and
- To support symbolic mathematical operations.

Several existing operating systems and languages were considered, but none could meet all of the design objectives. We therefore proceeded with the development of a new system, which merges the threaded interpretation of Forth with the functional approach of Lisp. The resulting operating system, known unofficially as RPL (for Reverse-Polish Lisp), made its first public appearance in June of 1986 in the HP-18C Business Consultant calculator. Subsequently, RPL has been the basis for the HP-17B, HP-19B, HP-27S, HP-28C and HP-28S calculators, demonstrating that it meets the objective of supporting a variety of calculators. The HP-17B, 18C, and 19B are designed for business applications; they and the HP-27S scientific calculator offer an “algebraic” calculating logic, and the underlying operating system is invisible to the user. The HP-28C and HP-28S scientific calculators use an RPN logic, and many of the facilities of operating system are directly available as calculator commands.

Mathematical Control

The official operating system objectives listed above were blended throughout the RPL development cycle with a less formal objective of creating a mathematical control language that would extend the ease-of-use and interactive nature of a calculator to the realm of symbolic mathematical operations. A calculator is distinguished from a computer in this context by:

- very compact size;
- “instant on” — no warm-up or software loading and/or bootstrapping;
- dedicated keys for common operations rather than qwerky keyboards;
- “instant action” when a function key is pressed.

The HP-28, which was developed by the same team that created the RPL operating system, is the first realization of this background objective.

Much of the design of RPL can be derived from a consideration of the manner in which ordinary mathematical expressions are evaluated. Consider, for example, the expression

\[ 1 + 2 \sin(3x) + 4 \]

As any RPN enthusiast knows, the expression as written here does not correspond in its left-to-right order to the order in which a human or a machine could actually carry out the calculation. For example, the first sum has to be delayed until after several other steps are executed. Rewriting the expression in RPN form, we do obtain a representation that is also executable in its written order:

\[ 1 \ 2 \ 3 \ x \ \* \ \sin \ \* \ + \ 4 \ + \]

To translate this sequence into a control language, we need to formalize several concepts. First, we use the generic term object to refer to each step in the sequence, such as 1, 2, or sin. Even in this simple example, there are three classes of objects:

1. Data objects. Execution of an object such as 1, 2, or 3 in the example just returns the value of the object.
2. Names. The symbol x must be the name of some other object; when x is executed, the named object is substituted for the symbol.
3. Procedures. Objects such as *, sin, and + represent mathematical operations, which are applied, for example, to data objects to create new data objects.

The concept of an object is closely tied to the concept of execution, which can be thought of as the “activation” of an object. An individual object is characterized by its object type, which determines its action when executed, and its value, which distinguishes it from another of the same type.

Expression evaluation in these terms becomes the sequential execution of a series of objects (the objects representing the RPN form of the expression). Two constructs are necessary to make the execution coherent: an object stack and an interpreter pointer. The first construct provides a place from which procedure objects can take their arguments and to which they can return their result objects. A LIFO stack as used in Forth is ideal for this purpose, and such a stack is included in RPL. The interpreter pointer is just a program counter that indicates the next object to be executed. The interpreter pointer should be distinguished from the CPU program counter, which indicates the next CPU instruction.

A mathematical expression considered as a sequence of objects suggests an additional classification of objects as either atomic or composite. An atomic object is an object that cannot be taken apart into stand-alone objects; examples are a simple data object like 1 or 2, or perhaps an object like * or + that is implemented normally in assembly language. A composite object is a collection of other objects. In Forth, a secondary word is an example of a composite object. RPL provides at least three types of composite objects: secondaries, which are procedures defined as unrestricted sequences of objects; symbols, which are sequences of objects that must be logically equivalent to algebraic expressions; and lists, which contain objects collected for any logical purpose other than sequential execution.

The final point in this brief mathematics-to-RPL derivation is the observation that the definition of composite objects leads to the concepts of threaded interpretation and a return stack. That is, in the example it is easy to imagine that the name object x could represent a composite object that in turn represents another expres-
sion. In that case, one would expect execution of \( x \) to cause the interpreter pointer to jump to the sequence of objects referenced by \( x \), while the location of the object following \( x \) in the original is stored so that execution can later return there. This process should be able to be indefinitely repeated, so RPL provides a LIFO stack for the return objects.

The preceding introduction might in some respects also have been an introduction for the derivation of Forth, if questions of floating-point versus integer arithmetic are ignored. In particular, both systems use threaded interpretation and a LIFO data stack for interexchange of objects. However, there are several important differences between Forth and RPL:

- RPL supports both direct and indirect threaded execution in a completely uniform manner.
- RPL supports dynamic allocation of its objects.
- RPL code is, in general, completely relocatable.

Formal Definitions

This section will present the abstract definitions of RPL that are independent of any particular CPU or implementation, with special attention to similarities and differences with Forth.

The fundamental structure in RPL is the object. Any object consists of a pair: the prologue address and the object body. The prologue address is that of a machine-code routine that executes the object; the body is data used by the prologue. Objects are classified by type; each type is associated with a unique prologue. The prologues thus serve a dual purpose of executing an object and identifying its type. The RPL object is analogous to a headerless Forth word, consisting of a code field (containing the prologue code address) and a parameter field (the body). An object is either atomic or composite. A composite object is either null or non-null; a non-null composite has a head which is an object and a tail which is composite.

In addition to being executed, all RPL objects can be copied, compared, embedded in composite objects, and skipped. The latter property implies that the memory length of any object is predetermined or can be computed from the object. For atomic objects such as real numbers, the size is invariant. For a more complicated atomic object such as a numerical array, the size can be computed from the array dimensions that are stored in the body of the array object. (RPL arrays are not composite—the elements do not have individual prologues and hence are not objects.) Composite objects may include a length field or they may end with a marker object.

A pointer is an address in the memory space of the CPU, and may be a location pointer or an object pointer. A location pointer addresses any part of memory, whereas a object pointer points to an object, specifically to the prologue location pointer at the start of an object.

RPL requires, in addition to the CPU program counters, five variables for its fundamental operation:

- The interpreter pointer \( I \).
- The current object pointer \( O \).
- The data stack pointer \( D \).
- The return stack pointer \( R \).
- The amount of free memory \( M \).

These variables are identical in purpose to the corresponding Forth variables, with the exception of \( M \) which has no analog in Forth. In the most general definition of RPL, \( I \) is an object pointer pointing to a composite object that is the top of a stack of composite objects called the runstream. \( R \) points to the rest of the runstream stack. In practical implementations, this definition is streamlined by allowing \( I \) to point to any object embedded in a composite, while \( R \) is a location pointer pointing to the top of a stack of object pointers, each of which points to an embedded object.

It is fundamental to RPL that objects can be executed directly or indirectly with equivalent results. This means that an object can be represented anywhere by a pointer to the object as well as by the object itself. This leads to several significant differences between RPL and Forth, in the nature of their system stacks and in the designs of their inner interpreters.

Execution

RPL object execution consists of the CPU execution of the object’s prologue, where the prologue code can access the object’s body by means of the object pointer \( O \). Object pointer execution is the CPU execution of the pointer’s addressee. This interpretive execution is controlled by the inner interpreter, or inner loop, which determines the sequence of object/object pointer execution.

The procedure concept in RPL is represented by the object type secondary, which is a composite object analogous to the Forth colon definition. Execution of a secondary is sequential execution of the objects and object pointers that comprise the body of the secondary. The execution is threaded in that the objects in a secondary may themselves be secondaries or pointers to secondaries. When encountered by the inner loop, an embedded secondary is executed prior to resumption of execution of the current one.

The RPL inner loop and prologue designs are modified from the Forth models in order to provide for interchangeable direct and indirect object execution (note: a patent application has been filed for the concepts described next). The inner loop consists of the following pseudo-code:

\[
\begin{align*}
O &= [I] \\
I &= I + \Delta \\
PC &= [O] + \Delta
\end{align*}
\]

where \([x]\) means the contents of address \( x \), and \( \Delta \) is the length of a memory address. This loop is the same in Forth, except that the CPU execution jumps to \([O]+\Delta\) instead of \([O]\). This is because all RPL prologues start with their own address, which is the feature that makes possible direct execution as well as indirect. Prologues look like this:

\[
\text{PROLOG} \rightarrow \text{PROLOG} \\
\begin{align*}
\text{IF} & \quad O + \Delta \neq PC \\
\text{THEN} & \quad \text{GOTO} \quad \text{REST} \quad \text{Test for direct execution} \\
O &= I + \Delta \\
I &= I + \alpha \\
\text{Correct} & \quad O \\
\text{Correct} & \quad I \\
\text{REST} \quad \text{(rest of prologue)}
\end{align*}
\]

Here \( \alpha \) is the length of the object body.

When an object is being executed directly, the inner loop does not set \( O \) or \( I \) correctly. However, a prologue knows it is being executed directly by comparing the PC address with \( O \) and can update the variables accordingly. A prologue is also responsible for preserving the threaded interpretation by including a return to the inner loop at its end.

This flexible interpretation is intrinsically slower than the indirect-only execution characteristic of Forth, because of the overhead of making the direct/indirect test. In practical implementations of RPL, it is possible to shift the overhead almost entirely to the direct execution case, so that the execution penalty
for the indirect case is negligible, including primitive assembly language objects that are never executed directly. The trick is to replace the last step of the inner loop with the Forth-like PC = [O], and, for prologues of directly-executable objects, replace the self-address at the start of each prologue with a slice of executable code \( \Delta \) in length. The compiled opcodes of this slice must also be the address of a meta-prologue that handles the direct execution case. In the HP-28, for example, the code slice consists of the instruction \( M=M-1 \), since decrementing available memory is common to virtually all directly-executable object prologues, plus a NOP instruction to fill out the \( \Delta \) length.

The virtue of direct execution that makes it worth this additional complication over Forth-style interpretation is that it enables the straightforward management of nameless objects that are created during execution. During the course of symbolic algebraic manipulations, it is common to create, use, and discard any number of temporary intermediate results; the necessity to compile and store these objects with some form of name for indirect reference, then uncompile them to recover memory, would make the whole process unmanageable. In RPL such objects can be placed on the stack, copied, embedded in composite objects, executed, and deleted. For example, a composite object representing the expression \( x + y \) can be added to a second object representing \( 2z \), returning the result object \( x + y + 2z \); furthermore, any of these objects could be embedded in a secondary object to perform the addition repetitively.

Although RPL, like Forth, is primarily a postfix language in which procedures take their arguments from the stack and return results to the stack, it does provide operations that work on the runstream to provide for prefix operations and for alterations to the normal threaded execution. Foremost among the runstream operations is the quoting operation that takes the next object from the runstream and pushes it on the data stack to postpone its execution. This operation is similar in purpose to the Forth word ' (tick), and is given the same name. RPL also has operations to push and pop objects from the return stack. (DO loop parameters, however, are not stored on the return stack, using a special environment instead.)

RPL’s ability to represent procedures as objects allows it to be even more relentlessly postfix than FORTH. For example, an if/then/else branch is accomplished by the object RPITE (reverse-Polish-if-then-else). RPITE takes three arguments from the stack, of which the third is interpreted as a flag. If the flag is true, the second object is executed and the top object is dropped; if false, the top object is executed and the second dropped.

**Memory Management**

The uniformity of direct and indirect execution means not only that objects as well as object pointers can be embedded in the execution stream, but also that object pointers can logically replace objects. In particular, the RPL data and return stacks explicitly are stacks of object pointers. This means, for example, that an object on the data stack can be copied (e.g. by DUP) at a cost of only \( \Delta \) bytes of memory, regardless of the size of the object. Furthermore, duplication and similar stack operations are very fast.

Of course, the objects referenced on the stacks must exist somewhere in memory. Many, including all of the system objects that provide system management and an application language, are defined in ROM and can be referenced by a pointer with no other housekeeping implications. Objects created in RAM may exist in two places. Those that are unnamed are stored in a temporary object area, where each is maintained as long as it is referenced by a pointer anywhere in the system (this implies that if a temporary object moves, all pointers to it must be updated). Naming an object consists of storing it as a pair with a name field in a linked-list called the user object area, which is analogous to the Forth dictionary. These objects are maintained indefinitely, until they are explicitly purged or replaced. A named objects is accessed by means of an identifier object, which consists of an object with a name field as its body. Executing an identifier causes the user object area to be searched for an object stored with the same name, which is then executed. This run-time resolution is intrinsically slower than the compile-time resolution used for ROM objects (and for Forth dictionary words), but it allows for a much more flexible system where the order in which objects are compiled is immaterial.

RPL also provides for objects that are intermediate between those fixed in ROM and those that are mobile in RAM. Objects in a plug-in ROM can be referenced by a ROM-id that is resolved from an address table included in the plug-in. A particular plug-in can be associated with its own portion of RAM, so that, for example, a plug-in might contain permanent formulas for which the variable values are maintained in RAM.

The process of naming objects by storing them with names in the user object area is augmented by the existence of local environments, in which objects can be bound to names that are local to a currently executing procedure. The binding is abandoned when the procedure completes execution. This feature simplifies complicated stack manipulations by allowing the stack objects to be named and then referenced by name within the scope of a defining procedure.

The rigid Forth dictionary management in which deleting a word requires deleting all chronologically later words was viewed as unacceptable for a system intended for "friendly" calculators. RPL, therefore, takes some pains to insure that any object stored in the user object area can be deleted without corrupting anything in the system:

- When a RAM object is stored in the user object area, a new copy of the object is stored, not a pointer to the object.
- Pointers to RAM objects are not permitted in composite objects. When a composite object is created from stack objects, RAM objects are copied and directly embedded in the composite. When a stored object is represented by name in a composite, it is the identifier object that is embedded, not a location pointer as in Forth.
- If a stored object is referenced by any pointers on the stacks at the time when it is purged, it is copied to the temporary object area and the pointers to it are updated accordingly. This means that the memory associated with an object is not recovered until the last reference to it is deleted.

The use of temporary objects with multiple references results in a memory management problem that never arises in Forth. The possibility of multiple references means that a temporary object cannot necessarily be deleted from memory immediately when a single reference to it is eliminated. In current RPL implementations, no memory recovery is performed until the system runs out of memory \( (M=0) \), at which time all unreferenced objects in the temporary object area are deleted. The process of checking for references can be significantly time-consuming, so that RPL execution does not proceed uniformly.
From the preceding discussion, it will be apparent that RPL is not as fast in general as Forth because of its extra interpretation overhead and greatly elaborated memory management scheme. While maximum execution speed is always desirable, the design of RPL emphasizes its role as an interactive mathematical control language in which flexibility, ease of use, and the ability to manipulate procedural information are paramount. In many cases, these attributes of RPL result in faster problem-solving throughput than Forth, which executes faster but is more difficult to program.

**A Practical RPL Implementation: the HP-28**

Among the various RPL-based HP calculators mentioned in the introduction, the HP-28 stands alone in the degree to which it makes many of the features of the underlying operating system directly available to the user. For example,

- The HP-28 contains a sophisticated parser for converting text entries into various RPL objects.
- The system data stack is directly visible, with the stack objects decompiled into readable forms.
- The user object area is accessible, and the user can create, recall, store, and delete the objects there.
- User programs are written in a variant of the internal RPL language.

The HP-28 parser allows the user to create RPL objects by entering an identifying character called a delimitier, followed by a text stream that specifies the object body. For example, a vector is entered with the square bracket [ followed by a series of floating-point numbers terminated by the closing delimiter ]. When the resulting vector object is displayed back to the user, the same symbols are used. An algebraic expression object representing the expression \( x+y+2\sin z \) is entered between ' ' delimiters: 'X+Y+2*SIN(Z)'. The full list of HP-28 user created objects is:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Identification</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real numbers</td>
<td>digits</td>
<td>-1.234E105</td>
</tr>
<tr>
<td>Complex numbers</td>
<td>(real number, real number)</td>
<td>(3.2, 4.5)</td>
</tr>
<tr>
<td>Strings</td>
<td>&quot;characters&quot;</td>
<td>&quot;ABCDEF&quot;</td>
</tr>
<tr>
<td>Real arrays</td>
<td>[ real numbers ]</td>
<td>[1 2 3]</td>
</tr>
<tr>
<td>Complex arrays</td>
<td>[ complex numbers ]</td>
<td>[(1.2) (3,4)]</td>
</tr>
<tr>
<td>Lists</td>
<td>( objects )</td>
<td>{1 'A+B' (3,2)}</td>
</tr>
<tr>
<td>Global names (identifiers)</td>
<td>characters</td>
<td>GEORGE</td>
</tr>
<tr>
<td>Local names</td>
<td>characters</td>
<td>Sam</td>
</tr>
<tr>
<td>Programs (secondaries)</td>
<td>&lt;&lt; objects &gt;&gt;</td>
<td>&lt;&lt; X Y 2 * + &gt;&gt;</td>
</tr>
<tr>
<td>Algebraic objects (symbolics)</td>
<td>'objects'</td>
<td>'X+Y+2*SIN(Z)'</td>
</tr>
<tr>
<td>Binary integer numbers</td>
<td>#digits</td>
<td>#123FFh</td>
</tr>
</tbody>
</table>

The HP-28 command/programming language is a sanitized version of RPL, in which:

- Runstream commands are available only as macro structures, so that the parser can check the integrity of the structures. The HP-28 provides structures for IF/THEN/ELSE, DO/UNTIL, WHILE/REPEAT, and FOR/NEXT, plus structures that make and abandon local variable bindings.

The calculator flavor of the HP-28 is preserved in its association of RPL operations with keys, so that any operation can be executed by pressing a single key. Common stack and mathematical operations are represented by permanent keyboard keys; those for which a site could not be reserved on the keyboard are available in "softkey menus." The actions of the six unlabeled keys directly below the display are determined by various menus that display softkey labels in the bottom row of the display.

The RPL programming available in the HP-28 makes it unique among calculators in the power and flexibility of its programming language, particularly in its ability to manipulate indefinite collections of objects by means of the list object type, and to pass procedures as arguments. The latter capability is demonstrated by its symbolic calculation capability, with which mathematical operations can be performed on expressions without requiring that the expressions be collapsed to numerical values. This feature is implemented by means of the algebraic object type (RPL symbolic object type), which is a composite object (compiled internally in RPN form) marked so that it is entered and displayed in an algebraic form like Basic or Fortran expressions. Mathematical functions that are applied to these objects return new algebraic objects representing the symbolic function value. For example, executing \( + \) with the objects 'SIN(X)' and 'COS(Y)-Z' on the stack returns the result object 'SIN(X)+COS(Y)-Z'. Note that these objects are executable procedures, not just text; 'COS(Y)-Z' is represented internally by the sequence \( Y \cos Z \). Evaluating an algebraic object (by means of the EVAL command) "runs" the internal program form of the algebraic object, causing execution of the functions in the expression, and substitution of the values of named variables.

**Examples**

1. Simple stack arithmetic. Add 1 + 2:

   ![Example Stack](image)

   Notice that pressing the [1] key starts a "command line" for entry of new objects. The stack (level 1 is the top) is initially empty.

   ![Example Stack](image)
ENTER means to parse and execute the command line. In this case, the only object is the real number 1; execution of that data class object just pushes it onto the stack.

\[ 2 \]

Notice the "calculator" feature that the + operation is bound to the \( \texttt{\[+\]} \) key for immediate execution, and moreover that ENTER, i.e. parse and evaluate the command line, is performed automatically by the \( \texttt{\[=\]} \) key before the addition is actually performed. This immediate execution is provided for keystroke efficiency, and for compatibility with previous HP RPN calculators. On the HP-28, the following sequences are equivalent:

\[
\begin{align*}
1 & \text{ ENTER } 2 + \\
1 & \text{ ENTER } 2 \text{ ENTER } + \\
1 & \text{ SPACE } 2 + \\
1 & \text{ SPACE } 2 \text{ SPACE } 2 + \text{ ENTER }
\end{align*}
\]

In the latter case, pressing the \( \texttt{\[=\]} \) key suppresses immediate execution so that the + command name is accumulated into the command line rather than executed. This mode is used for entering programs or other objects where deferred execution is desired.

The command line is exactly equivalent to a program that is executed immediately. If the program quote \( \texttt{<<} \) is the first item in the command line, the command line program is not executed, but is pushed onto the stack as a program:

\[
\text{<< 1 \text{ SPACE } 2 + \text{ ENTER}}
\]

The \( \texttt{<<} \) symbol in the command line acts as a combination of Forth's tick (\( ' \)) and colon (\( : \)). The closing \( >> \) acts as Forth's semicolon (\( ; \)).

Now press \texttt{EVAL}:

\[
\begin{align*}
& 4: \\
& 3: \\
& 2: \\
& 1: \quad 1
\end{align*}
\]

\[
\begin{align*}
& 2: \quad 1 \\
& 1: \quad 3
\end{align*}
\]

2. Other Object Types. Multiply the matrices \[
\begin{bmatrix}
1 & 2 \\
2 & 1
\end{bmatrix}
\] and \[
\begin{bmatrix}
0 & 2 \\
1 & 0
\end{bmatrix}
\].

Because of the check and dispatch system, the logical steps for performing matrix arithmetic are the same as for scalar arithmetic:

\[
\begin{align*}
& [[1 \ 2] \ [2 \ 1]] \text{ ENTER} \\
& 3: \\
& 2: \\
& 1: \quad 1 \\
& \quad \begin{bmatrix}
1 & 2 \\
2 & 1
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
& \quad \begin{bmatrix}
0 & 2 \\
1 & 0
\end{bmatrix} \text{ ENTER} \\
& 3: \\
& 2: \\
& 1: \quad 2 \\
& \quad \begin{bmatrix}
2 & 2 \\
1 & 4
\end{bmatrix}
\end{align*}
\]


A \texttt{ENTER}

\[
\begin{align*}
& 4: \\
& 3: \\
& 2: \\
& 1: \quad \begin{bmatrix}
2 & 2 \\
1 & 4
\end{bmatrix} \text{ 'A'}
\end{align*}
\]

The variable A is undefined, so executing the name A just returns the quoted name to the stack. Add B:

B \texttt{+}

\[
\begin{align*}
& 4: \\
& 3: \\
& 2: \\
& 1: \quad \begin{bmatrix}
2 & 2 \\
1 & 4
\end{bmatrix} \text{ 'A+B'}
\end{align*}
\]
Notice that the new algebraic object ‘A+B’ has been formed. Square this result:

```
2
```

Now execute EXPAN (expand):

```
4: [ [ 2 2 ] [ 1 4 ] ]
```

This last operation is more than just appending a function to the end of an RPN expression. The HP-28 has analyzed the expression, and performed the symbolic multiplication implied by the quadratic power. This is an example of an identity operation, in which an expression is rearranged to a new but formally equivalent structure. This is fundamentally different from the kind of expression editing that occurs in Basic, for example, where an expression is represented by a character string during editing, and the computer merely replaces the string when the editing is complete.

Assign a value to A: 10 ‘A’ STO. Now press EVAL:

```
4: [ [ 2 2 ] [ 1 4 ] ]
```

A has been replaced in the expression by its newly assigned value. The substitution can also be symbolic: ‘Z+T’ ‘B’ STO assigns the value ‘Z+T’ to B; then EVAL produces:

```
4: [ [ 2 2 ] [ 1 4 ] ]
```


The HP-28 lets a programmer intermix stack- and variable-oriented styles according to his taste and to the requirements of a program. For example, the following program computes the roots of a quadratic equation \( a x^2 + bx + c \), where \( a, b, \) and \( c \) are supplied as stack arguments (in the program listings that follow, comments or stack contents are shown to the right of the program steps):

```
\[
\begin{align*}
&\text{\( a \ b \ c \)} \\
&\text{\( -(b+\sqrt{(b^2-4*a*c)})/(2*a) \)} \text{ EVAL} \\
&\text{\( x_1 \)} \\
&\text{\( -(b-\sqrt{(b^2-4*a*c)})/(2*a) \)} \text{ EVAL} \\
&\text{\( x_2 \)} \\
&\end{align*}
\]
```

In this version, the coefficients are stored in local variables \( a, b, \) and \( c \), and then the roots are computed from explicit expressions in those variables. An alternate version of this program performs all calculations on the stack:

```
<< 3 PICK / \ a b c \\
SNAP ROT 2 * / NEG \ a b/c \\
DUP SQ \ c/a -b/2a \\
ROT - \ b^2/4a -\sqrt{(b^2/4a)^2 -c/a} \\
DUP2 + \ -b/2a \ (b/2a)^2 \ -c/a \ x_1 \ x_2 \\
3 ROLLD - \ x_1 \ x_2 \\
>>
```

The second version is more compact and faster, but is harder to write and modify.

The next program illustrates a variety of RPL programming features, including recursion and the use of lists to hold arguments and initial and final results. Because it is self-recursive, this program must be stored in a variable matching the name that it calls, in this case SORT. SORT (using the variable name to label the program) sorts the elements of a list in increasing numerical order. The algorithm is to take the first element of the list, then divide the remaining elements into two lists, one containing the elements smaller than the first, the other containing larger elements. Then the two lists are sorted using the same method, and then recombined into a single list with the original first element inserted between those of the two lists.

```
<< IF DUP SIZE 1 > >
THEN LIST->
DUP 1 + ROLL 
-> x
\[\text{Put the objects on the stack. Get the first object. Save the first element as } x.\] 
\[\text{Initialize "less" and "greater" lists.} \]
\[\text{Iterate for } n-1 \text{ elements: } \]
\[\text{Get the next element. } \]
\[\text{Sort the first list. } \]
\[\text{Sort the second list. } \]
\[\text{Combine the lists. } \]
ELSE +
END
NEXT
SORT
SWAP SORT
x + SWAP +
END
>>
```

As a simple example, entering the list \( \{ 5 \ -1 \ 0 \ 4 \ 2 \} \), then executing SORT returns the sorted list \( \{-1 \ 0 \ 2 \ 4 \ 5 \} \). It is easy to modify SORT to work for tests other than simple numerical ordering. Better yet, a minor change to SORT would allow a procedure defining an ordering test to be passed as an argument to SORT, so that the same program can be used for any kind of ordering of arbitrary tests.

This discussion has shown a few examples of the properties of the HP-28 as a practical RPL system. The HP-28 has numerous features, including graphics, equation solving, and unit conversions, that may be considered as pre-programmed applications of the RPL system rather than as RPL tools. The existence of those applications, and the ease with which the user can extend the applications or create new ones, is a manifestation of the consistency, uniformity, and flexibility of the RPL language.