ND1 RPL Implementation (MorphEngine v.1.3)

pdf (3/6/11)

Overview

ND1 implements HP's RPL programming language, as implemented in the HP-50g. The language is fully implemented, but not the full set of commands. The available commands are listed in the functions summary.

Both "user functions" and "normal" User RPL programs are supported. (System RPL is not supported.)

All language constructs and language-close commands are fully supported. These includes all stack commands (DUP, PICK3, etc.) and all processing commands (DOSUB, DOLIST, STREAM, SEQ, MAP).
There is, however, one major limitation:
- Interactive commands (other than MSGBOX) are not part of the ND1 command set, currently. (This will come with the 1.4 update.)

And there's one very minor limitation:
- IFT() and IFTE() cannot be used recursively

The implementation does have "transparent extensions", that is, features that extend RPL without that they change any aspect of the specification. Specifically, in ND1's implementation
- any RPL program can be used as "user function". This implies that you can graph a RPL program (it still needs to be *suitable*, that is, return the correct data (a single real number in case of the existing DRAW function)).
- a RPL program can call a JavaScript function
- a RPL program can be called from a JavaScript function
- ND1 features a recording mechanism that will construct a RPL program from a stream of recorded user actions

ND1 automatically translates RPL transcodes (such as \<<, \v/, etc.) upon data import. To use this feature, make sure your datum begins with the characters "\<<" (w/o the quotes).

Example Programs

There're many sources of RPL example programs on the net, as thousands of RPL programs written for HP-28, 48, 49, 50g are in the public domain.

You can get the example programs from the HP-28S User's Manual by downloading the "HP Example Programs" shared folder. You may need the User's Manual (which you can also find with Google) to make sense of the example programs, though there's a short description for each available in a forum post.

To look at them, tap the 2nd-level key and then the respective soft-key (this is short-hand for 'name' RCL EDIT). You can also email the folder to yourself (or set up your computer as server) and peruse them on your computer (open with an editor as UTF-8).

After having set up your computer as server, you can transfer a folder back to your device using a folder's Sharing | Download function. Any transcode character sequences (\<<, \v/, etc.) will be converted automatically.

Many more RPL programs can be found in other shared folders. For example, see Examples and Challenges. Some of the programs require a 50g or ND1. (That is, they use RPL language constructs or commands not available in HP-28 or HP-48 calculators.)

The Language

RPL is one of the easiest computer languages to pick up.

This section provides a minimal crash course. You should be able to find nicer tutorials on the net.

Looking at example programs is a great way to learn the language, as well.

It is assumed here that you read the User Functions tutorial. So, you already know that an RPL program is enclosed in two special delimiting characters, and have some familiarity with it.

The second example program you looked at in the tutorial used a *do-until* construct, and we walked through its operation.

Do-until is one of a handful of *control structures* in RPL, that control the flow of *program instructions*. Instructions are either function names (built-in or user), variable names (built-in or user), *literals* (number, strings, vectors, matrices, other RPL programs; written just like you'd enter them on the edit line) or *keywords* of control structures. Keywords are reserved words (you cannot use them as user variable or function names), which we will list in the following.

To recap, the **do-until** construct looked like this

```
DO
   instructions
UNTIL
   instructions, producing condition code
END
```

Note, "instructions" means 0 or more instructions. (There's no limit to the number of instructions you can put between the keywords, in this control structures, as well the ones below.)

Following this format, here are the other control structures in RPL:

Conditionals

if-then construct:

```
IF
   instructions, producing condition code
THEN
   instructions
END
```

Examples:

```
IF x 0 > THEN "x is greater then zero" END
IF 0 THEN "this will never be printed" END
IF 1 THEN "this will always be printed" END
```

if-then-else construct:

```
IF
   instructions, producing condition code
THEN
   instructions
ELSE
   instructions
END
```

Example:

IF x 0 > THEN “x is greater then zero” ELSE “x is not greater than zero” END

case construct:

```
CASE
   instructions, producing condition code
   THEN
      instructions
   END
   instructions, producing condition code
   THEN
      instructions
   END
   ... (repeat the previous four lines as many times as you have cases)

   default instructions
END
```

Example:

```
CASE
   DUP "A" == THEN "Alpha" END
   DUP "B" == THEN "Beta" END
   DUP "G" == THEN "Gamma" END
   "Unknown letter"
END
```

Conditional Loops

while-repeat construct:

```
WHILE
   instructions, producing condition code
REPEAT
   instructions
END
```

Examples:

WHILE RAND 0.5 > REPEAT “random number is greater than 0.5” END

WHILE @typeof “Number” == REPEAT END (this would delete numbers off the stack until a different kind of object is encountered)

WHILE TYPE 1 == REPEAT END (same; using the more cryptic historical TYPE command)

do-until construct:

```
DO
   instructions
UNTIL
   instructions, producing condition code
END
```

Example:

DO “random number: ” RAND UNTIL 0.5 > END (produces strings and random numbers until random number is greater than 0.5)

Unconditional Loops

start-next construct:

```
startval, endval
START
   instructions
NEXT
```

(where *startval* and *endval* are numbers)

Examples:
  1 5 START “say this five times” END (pushes on stack/prints string five times)

start-step construct:
  startval, endval
  START
    instructions
  incrementVal
  STEP
(where *startval*, *endval*, and *incrementVal* are numbers)
Examples:
  1 5 START “say this five times” .1 STEP (pushes on stack/prints string 50 times)

for-next construct:
  startval, endval
  FOR localVarName
    instructions
  NEXT
(where *startval* and *endval* are numbers; *localVarName* is a name)
Examples:
  1 5 FOR i 2 i * END (pushes 2, 4, 6, 8, 10 on the stack)

for-step construct:
  startval, endval
  FOR localVarName
    instructions
  incrementVal
  STEP
(where *startval*, *endval*, and *incrementVal* are numbers; *localVarName* is a name)
Examples:
  1 5 FOR i i .5 STEP (pushes 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5 on the stack)

comment:
  @ here is a comment; it runs until the end of the line

Since instructions in constructs can contain other constructs, you can *nest* constructs.
For example, here’s an *if-then* construct inside a *for-loop*

```
  1 10 FOR i
       IF i 5 <
       THEN
          doThis
       ELSE
          dothat
       END
  NEXT
```

Local variables in the *for-loop* can be used as normal integers. If you don’t specify a local variable, or other variable, by name, the input to a function call will be taken from the stack. The stack is also where the results of functions, if any, go.

Besides of for loops, you can create local variables through a special instruction which takes elements from the stack, assigns them to local names, and then executes a program or expression. The local variables will only be valid in that *execution context*.

This is the an unusual computer language construct but a very powerful one that is used all the time.

Local context-construct:

→ localVarNames

« »

or

'expr'

where *localVarNames* is one or more names

Examples:

« → x y 'sqrt(x*x + y*y)' » (take two items from the stack and name them "x" and "y", use them in the expression, and return the result)

« → x y « x x * y y * + sqrt » » (same, but using an RPL program instead of an expression)

This construct is so important because it gives you convenient access to variables. The stack can be very convenient, too, but it can get in the way if you both need to take items off it and put them on. If you repeatedly need to refer to the same item, you definitely want to use a local variable for it.

Note, the local variable is only valid inside the RPL program or expression that follows. Note, too, that *local context* being just another language construct, you can, of course, have a second local construct in the RPL program of a first one. And so on. In other words, you can also nest local execution contexts. In doing so, you can reuse variable names without fear of "overwriting" variables in a context higher up. If you have a local variable *i*, for example, it will not interfere with another variable *i* in the context, if any, that embeds your local context.

In combination, local variable and stack manipulations, can produce very powerful, concise programs.
A downside of RPL is that they may be quite hard to read (because they are so compact).
Consider this program

« → n « IF n 1 <= THEN n ELSE 0 1 2 n START DUP ROT + NEXT SWAP DROP END » »

to produce the Fibonacci sequence of numbers. *n* is the local variable here and it's used multiple times. At the same time, intermediate results (and eventually, the final result) are put and taken from the stack.

A big advantage of using RPL is that object types can be used transparently.
Both examples for the *local-context* construct work just as well with real numbers as they do with complex ones, for example.