Stack Parameter Check

Bruce Horrocks, #609

In the last Datafile, my article 'Stack Manipulation' gave two routines that simplified stack manipulation by allowing you either to swap items around in groups or to save them out of the way for later use. Both programs required input arguments to let them know how much of the stack to work on but neither did any argument checking because, as I hinted at the time, doing so is a lot more complicated than it really ought to be and stack checking often takes more code than the program itself. This risks obscuring the purpose of the code, so what would be useful is an RPL command that checks the type and number of arguments and which can be simply and economically called at the start of any program. Before launching into a solution, it is worth taking a quick tour through the existing mechanisms for obtaining user provided input of a known type.

For checking arguments on the stack there are the commands **`TYPE`** and **`VTYPE`** and for prompting for input there are the commands **`INFORM`**, **`INPUT`**, **`PROMPT`**, **`PROMPTSTO`** and **`CHOOSE`**.

`TYPE` **&** **`VTYPE`**	Returns the data type of the top item on the stack. (**`VTYPE`** returns the type of the data that is in the variable named on the top of the stack). This is very efficient for a single argument of a single type but becomes very inefficient when multiple arguments of multiple types require checking. We will see more on this later.
`INFORM`	Displays a form comprising one or more fields that the user fills in. These fields can be forced to accept only specified data types and the user is alerted at the time so they know exactly which field is in error. The primary disadvantage is that it interrupts the execution flow of the program – great for programs that require dynamic input of data, not so great for library routines.
`INPUT`	Prompts for data input into the command line. Access to the stack is disabled but, otherwise, it is the normal command entry method so the user is able to enter any data type. Pressing **`ENTER`** terminates input and program execution continues. The user's input is returned to the program as a string which you then have to do **`OBJ→`** on and deal with any errors. There is an option to have the input verified as being a valid object: this ensures that **`OBJ→`** will succeed but makes no restriction on the number or type of objects that the user may have entered. Typically for use only where the user knows what they are doing and, as with **`INFORM`**, this command is of no use to us because it interrupts the program execution.

PROMPT	Is similar to **INPUT** except that stack operations and calculations are available. When the user executes **CONT** the program continues with the stack as the user leaves it – there is no input string to be parsed.
PROMPTSTO	Takes a single argument – the name of a variable – which is displayed to the user and then accepts input via the command line. It then creates a global variable with the supplied name and stores the user's input in it. A bit of an odd command because its operation is more like **INPUT** than **PROMPT** but without the advantages.
CHOOSE	Allows the user to choose from a list of items. Great, because it means that only known data can be supplied, but limiting, because the program has to supply that data in the first place. As with the others, it interrupts program flow.

It should be clear then that, for checking parameters passed to subroutines, only **TYPE** and **VTYPE** will be of any use. Whilst they are simple and easy to use, **TYPE** and **VTYPE** require a lot of code when what is deemed valid input can be one of several types. Consider the utterly trivial program « **1. +** » which, even if only 'numbers' are allowed, has to check for type 0 (real number), type 1 (complex number), type 10 (binary integer) and type 28 (real integer). More generically still, any of these could be tagged (multiple times) and potentially stored in a variable since the + command will happily accept all of these types of input.

Let's have a go at an 'add one to the value on the stack' program that checks that it has valid input and see what we get.

```
\<<
   @ Too few arguments?
   IF DEPTH 1 \<= THEN #201d DOERR END

   DUP
   DTAG       @ Remove any tags that might be present

   @ If we have a variable then check its content
   IF DUP TYPE 6. == OVER TYPE 7. == OR
   THEN VTYPE ELSE TYPE END

   @ Only now can we check for valid types
   IF
      DUP 0. ==
      OVER 1. == OR
      OVER 10. == OR
      OVER 28. == OR
   THEN
      DROP              @ Input is valid
   ELSE
      DROP #202h DOERR   @ Bad argument type
   END
   1. +   @ Hooray! Finally, we can execute our program
\>>
```

It would be quicker and shorter to just do

```
            \<<  IFERR 1. + THEN #202h DOERR END \>>
```

and let the **+** command take the strain, so to speak. Unfortunately even this won't work because **+** also accepts strings, lists, matrices, vectors, algebraics and numbers with units as inputs, which we don't want in this case. In other words, the list of exclusions is as long as the list of valid types.

In going through the above exercise, we have learnt some things about what our solution should and shouldn't do.

- There is no need to make it efficient for single arguments as **TYPE** can handle this well enough.
- We want some sort of 'lazy' approach so tagged items and variables resolve down to 'what actually gets added'. But, equally, there needs to be a 'strict' approach – a way to specify that we should not resolve those types for the occasions where the input must be a tagged item or must be a variable.
- Numbers with units attached should not be included in the lazy approach. (A program that performs 'multiply by one' would be okay but is probably the exception.)
- There's no real need to allow for combinations e.g. input that must be a tagged string, as this is so rare it is just as easy to invoke the test twice – once for a tag and then, if successful, for a string.
- It should be straightforward to use the test more than once, in logical combination
- Lastly, it is probably worth creating additional types that join two or more of the **TYPE** types together for convenience. The most useful of these is likely to be 'number' to include both real numbers and real integers. Others could be: 'two element vector' for both array vector and complex number; and 'matrix' for normal and symbolic matrices (lists of lists).

We need to devise a convenient way of specifying the allowable items in each stack level while taking these points into account. The obvious approach is to use the **TYPE** numbers in a list, something like $\{\ p_2\ p_3 \dots p_n\ \}$ where each p_i represents the parameter on level *i* of the stack to be checked and can be $t \mid \{\ t_1\ t_2\ t_3 \dots\ \}$ where *t* is one of the valid **TYPE** values, or a list of those valid types. So now a parameter checked 'plus 1' program might look something like:

```
      \<< IF  { { 0. 1. 10. 28. } } PTYPE  THEN 1. + END \>>
```

where `'PTYPE'` is the name we have chosen to give our program, analogously to **VTYPE**.

For a single parameter this is just about bearable but consider the simple program

```
\<< + + \>>
```

which adds together the top three numbers on the stack. With parameter checking this becomes:

```
\<< IF  { { 0. 1. 10. 28. } { 0. 1. 10. 28. } { 0. 1. 10. 28. }
   } PTYPE  THEN + + END \>>
```

and we can see that things have rapidly become inelegant, if not unwieldy, already.

The solution is to ditch the **TYPE** numbers completely and use a string instead. If we assign letters to represent the various types, then we can easily specify the allowed types in a simple and compact notation. This also neatly solves the problem of how to specify additional types such as ‘number’ meaning both real number and real integer, as we can assign a single letter and worry about interpreting it later. Using this approach, our ‘plus 1’ program becomes

```
\<< IF "n" PTYPE THEN 1. + END \>>
```

and our “add 3 numbers” program becomes

```
\<< IF "nnn" PTYPE THEN + + END \>>
```

This is starting to look a lot better. If we add an action indicator to the start of the string to tell the program how to handle errors then it can be made even more compact. For example if ‘X’ is defined as *Exit on error* meaning that **PTYPE** reports any parameter errors to the user so that your code doesn’t have to, then the above programs become

`\<< "Xn" PTYPE 1. + \>>` and `\<< "Xnnn" PTYPE + + \>>`

which are about as compact and neat as possible.

The action codes and letter combinations are:

Level 1 / Argument 1		Level 2 / Item 2	Level 1 / Item 1
"X*ccc…*"	→		
"T*ccc…*"	→	"R*rrr…*"	0./1.

where

Code	*Meaning*
X	**eXit mode**. Action code instructing **PTYPE** to abort and display an error message if any parameter is incorrect. If all parameters are correct then the parameter string is removed from the stack and PTYPE ends, allowing the calling program to continue with a now-validated stack.

Code	*Meaning*

T	**Test mode**
	Action code informing `PTYPE` that it should leave a result code on the stack upon completion for subsequent testing.
	The result code in level 1 is a real number having the following values:
	• `0.` indicates that all the parameters were as expected; while
	• `1.` indicates that one or more parameters were incorrect.
	If the test failed then level 2 is a string starting with the letter 'R' and followed by a series of x's and full stops, one per parameter being tested. An 'x' indicates that the parameter in the corresponding position in the test string failed while a full stop indicates that the parameter passed.
	Example 1:
	`\<< 12.34 "TEST" "Tns" PTYPE \>>`
	would result in a stack of:
	`3:         12.34`
	`2:         "TEST"`
	`1:         0.`
	Example 2:
	`\<< 12.34 "TEST" "Tnn" PTYPE \>>`
	would result in a stack of:
	`4:         12.34`
	`3:         "TEST"`
	`2:         "R.x"`
	`1:         1.`
	Note that PTYPE returns true if the test fails. This facilitates error handling e.g. code similar to
	`\<< 1.2 "Tn" IF PTYPE THEN handle_error END`
	`rest_of_program \>>`
	which avoids a large "else" clause containing the bulk of the program.

c	**Character code** A character indicating the type of object that is expected on the stack level indicated by its position from the start of the string.
r	**Result code** A result character indicating whether the parameter in that position in the string was of the required type or not. A full stop indicates that it was a match, an 'x' indicates that it wasn't.

and the valid character codes are:

Character Code	***Object***		***TYPE Code***
-	Any		The parameter can be any object
n	Number	0 28	Real number Real integer
u	Units	13	Number with units
c	Complex number	1	Complex number
b	Binary	10	Binary integer
a	Array	3 4 29	Real array Complex array Symbolic vector/matrix
2	2-element Vector	1 3 29	Complex number Array (of appropriate dimension) Symbolic vector/matrix (of appropriate dimension)
3	3-element Vector	3 29	Array (of appropriate dimension) Symbolic vector/matrix (of appropriate dimension)
l	List	5	List
s	String	2	Character string
v	Variable	6 7	Global name Local name
t	Tagged object	12	Tagged object
p	Program	8	Program
f	Formula/Function	9	Algebraic
g	Graphic	11	Graphics object

I’m sure you will have noticed that several codes are omitted. This is deliberate:

- the system objects with **`TYPE`** values from 20 upwards are ignored since most userRPL programs will never encounter them
- the others I consider to be so rarely used on their own (and even more rarely used in conjunction with other parameters on the stack) that it is not worth the

overhead of providing for them. After all, **PTYPE** is intended to complement **TYPE**, not replace it entirely.

Taking all of the above into account gives:

PTYPE

```
%%HP: T(3)A(R)F(.);
\<<
  @ PTYPE expects a string on the top of the stack
  @ indicating the types of values to expect on the rest
  @ of the stack. So check that:

  @ 1) there is at least one item
  IF DEPTH 2. < THEN #201h DOERR END  @ Too few args

  @ 2) the first item is a string
  IF DUP TYPE 2. \=/ THEN DROP #202h DOERR END @ Bad arg type

  @ 3) there are at least as many values on the
  @ stack as there are characters in the string.
  IF DEPTH OVER SIZE > THEN #201h DOERR END

  @ Keep the stack as it is and simply write the test
  @ results over the top of the test string.
  @ So, stack usage at the start of the main loop is:
  @
  @ n: param n-1
  @    ...
  @ 3: param 2
  @ 2: param 1
  @ 1: string specifying the parameter types

  @ Go through the string and check each type against
  @ the value on the stack.
  @ (Start from char 2 as the first is special)
  2. OVER SIZE FOR a

    @ Get the a'th parameter
    a PICK

    @ Get its type
    DUP TYPE

    @ Get the a'th code character
    PICK3 a a SUB

    @ And save these three as local variables
    \-> p t c \<<

      @ If the parameter is tagged then remove the tag
      @ (unless the test is explicitly for a tagged value)
      IF t 12. ==  c "t" \=/  AND
```

```
THEN
  p DTAG 'p' STO
  p TYPE 't' STO
END

@ If the parameter is a variable then retrieve its
@ contents
@ (unless the test is explicitly for a variable)
IF t 6. ==  t 7. ==  OR
   c "v" \=/  AND
THEN
  p VTYPE 't' STO
  p RCL 'p' STO

  @ Do the tags test again (because a local var
  @ can hold a tagged value even though a
  @ global can't)
  IF t 12. ==  c "t" \=/  AND
  THEN
    p DTAG 'p' STO
    p TYPE 't' STO
  END

END

@ Finally, we get to the 'actual' tests. Each
@ case statement test checks one of the character
@ code types and leaves a true/false value on
@ the stack.
@ (The case order affects the speed of execution so
@ they are ordered with the most common first)
CASE
  c "n" == THEN    @ Number
    t 0. ==  t 28. ==  OR
  END

  c "f" == THEN    @ Formula/Function
    t 9. ==
  END

  c "a" == THEN    @ Array
    t 3. ==  t 4. ==  t 29. ==  OR OR
  END

  c "l" == THEN    @ List
    t 5. ==
  END

  c "s" == THEN    @ String
    t 2. ==
  END

  c "c" == THEN    @ Complex number
```

```
      t 1. ==
    END

    c "b" == THEN    @ Binary
      t 10. ==
    END

    c "p" == THEN    @ Program
      t 8. ==
    END

    c "2" == THEN    @ 2 element vector
      IF t 3. ==  t 29. ==  OR
      THEN
        p SIZE {2.} ==
      ELSE
        0.
      END
      t 1. == OR
    END

    c "3" == THEN    @ 3 element vector
      IF t 3. ==  t 29. ==  OR
      THEN
        p SIZE {3.} ==
      ELSE
        0.
      END
    END

    c "u" == THEN    @ Number with units
      t 13. ==
    END

    c "t" == THEN    @ Tagged object
      t 12. ==
    END

    c "v" == THEN    @  Variable
      t 6. ==  t 7. ==  OR
    END

    c "g" == THEN    @ Graphic
      t 11. ==
    END

    c "-" == THEN    @ Any type allowed
      1.
    END

    @ Default -> unknown/unexpected so error
    0.
  END
```

```
      @ Update the result string with the outcome
      IF THEN "." ELSE "x" END
      a SWAP REPL

    \>>
  NEXT

  @ Report final error status - "x" marks the spot!
  DUP "x" POS
  IF OVER HEAD "T" ==
  THEN
    @ Turn POS value into 0. or 1.
    NOT NOT

    @ Start the result string with "R"
    SWAP 1. "R" REPL SWAP
  ELSE
    @ Report error
    NIP IF THEN #202h DOERR END
  END
\>>
```

HP50g Checksum: #A67Dh Bytes: 1278

RPL Stack Manipulation Revisited

So, to come full circle, here is a version of the program SWAPab from Datafile V27N3p6 which checks its parameters using PTYPE. The additional code is underlined.

<< <u>"Xnn" PTYPE</u> DUP2 + -> a b c << 1. b START c ROLLD NEXT >> >>

There, that wasn't so bad!