# Recent and Planned Language Extensions in pqR

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

`http://www.cs.utoronto.ca/~radford`

`http://radfordneal.wordpress.com`

`http://pqR-project.org`

# Good Language Features Should...

**Lead to reliable programs.** The simple, easy, obvious way of doing something should produce the correct result. An R design flaw:

```
A <- M[i:j,]
```

doesn't always work. Instead, you need

```
A <- M[i-1+seq_len(j-i+1),,drop=FALSE]
```

**Lead to efficient programs.** The obvious way should be the efficient way. It's not good to make the obvious way be slow, and introduce tricky ways for "experts" that are faster, such as...

```
seq.int, rep.int, .rowSums, anyNA, v[1L] instead of v[1]
```

**Be clear, concise, and beautiful.** With a bit of thought, there should be no need to trade off this for the previous two.

# Language Extensions in pqR

Implemented recently:

- A set of related changes addressing R's design flaws surrounding the ":" operator and dimension dropping.
- New forms of the `for` statement.

Planned:

- Some ways to write code that is shorter, clearer, and less ugly.
- A general syntactical mechanism for adding "flags" to formal and actual arguments, and several uses of this mechanism.

Possible:

- A scheme for handling quoted arguments, including their use in achieving the effect of "pass by reference".
- Automatic differentiation, needed for gradient-based methods.
- Automatic incremental computation, very useful for MCMC.

# Need for a Sequence Operator that Operates Correctly

**Problem:** Using `i:j` to create an increasing sequence does not produce a zero-length sequence when `j` is less than `i`. This is very annoying, and leads to buggy code — including bugs in code maintained by R Core.

Using `seq_len` is clumsy and not sufficiently general.

**Another problem:** `1:n-1` does not start at `1` — poorly-chosen precedence.

**Solution:** A new operator, which produces only increasing sequences, including zero-length ones, and which has lower precedence than the arithmetic operators.

# New Sequence Operator ". ." in pqR

Examples of its use:

```
for (i in 1..n-1) A[i, i..i+1] <- 0

v[1..n] <- A[1..n,i]

if (any (v %in% (i..j))) stop("something not right")
```

**But...** `i..j` is a valid symbol!

Yes. It's necessary to disallow symbols with consecutive dots, except at the beginning or end.

Some ambiguities:

`i..(j+1)` is a call of the function `i..`

`i..-j` subtracts `j` from `i..`

Putting spaces around `..` unambiguously makes it the sequence operator. There would be no ambiguities if multiple dots at the end of a symbol were disallowed, but they do get used...

# Compatibility Issues with `..`

Explicit use of `..` within a symbol seems to be fairly rare, but can be accomodated in old code by setting an option to not parse the `..` operator.

The `make.names` function converts illegal characters to dots, which could produce a symbol with consecutive dots — it now collapses such sequences to one dot.

Similarly, `make.unique` could create a name with consecutive dots, when adding a dot as a separator — it now doesn't add a dot if one is already there.

# Stopping Inadvertent Dimension Dropping

**Problem:** We want to create a sub-array of `A` with all its columns, but only those rows whose indexes are in `v`. We try to do that with `A[v,]`.

It usually works, but we get a vector rather than a matrix if either `v` has length one or `A` has only one column. So there's lots of buggy code. Adding `drop=FALSE` everywhere works, but is very tedious and unreadable.

**Start of a solution:**

First, pqR defines "`_`" to be a special object equivalent to a missing argument. When used as an array subscript it selects all of a dimension, without ever dropping it.

Writing "`_`" for a missing argument is also clearer than writing nothing.

Second, pqR doesn't drop a dimension if the index is a 1D non-logical array, even if it has length one. This probably won't break much existing code.

Result: Now `A[array(v),_]` always produces a matrix.

# Make the New Sequence Operator Produce a 1D Array

Many of the vectors used to index arrays are produced by a sequence operator. The new `..` operator in pqR is defined to produce a 1D array, so we don't have to use `array`.

Now `A[1..n,_]` produces a matrix with one row when `n` is one, and a matrix with zero rows when `n` is zero.

Similarly, `A3[1..n,1,1..m]` delivers a 2D matrix even when `n` and/or `m` is zero or one. Note that adding `drop=FALSE` would not solve the problem here, since it would always produce a 3D array.

# An Unfortunate Impossibility: Zero-Length Vectors Can't Contain Negative Elements

**Problem:** If `ix` is a vector of positive integers, `v[-ix]` gives a vector with all the elements of `v` except those in `ix`.

Well, almost. Unfortunately, it doesn't work when `ix` is of length zero!

**Possible solution (not yet in pqR):**

Define a function `except(ix)` that returns `ix` with some suitable attribute attached that signifies exclusion rather than inclusion.

Now `v[except(ix)]` works correctly when `ix` happens to have length zero. Also, it can now work with indexes that are names.

It's maybe clearer too. Plus, we can now find bugs more easily, if we make zero and negative numbers in `ix` illegal.

# New Forms of the For Statements

pqR now implements several new forms of the `for` statement:

```
for (i along v) S
```
  means `for (i in seq_along(v)) S`

```
for (i,j along M) S
```
  means `for (j in 1..ncol(M)) for (i in 1..nrow(M)) S`

```
for (i down M) S
```
  means `for (i in 1..nrow(M)) S`

```
for (j across M) S
```
  means `for (j in 1..ncol(M)) S`

(Except pqR doesn't handle methods for `length` and `dim` yet, but will soon.)

These are non-essential conveniences, but they have no backwards compatibility issues — `along`, `down`, and `across` (and also `in`) don't need to be reserved words.

# Some Planned Syntactic Sugar and Spice

- Matrix / data frame interchangeability: For any matrix, `X`, make

  `X$fred` equivalent to `X[,"fred"]`

- Non-ugly attribute references: For any non-S4 object, `x`, make

  `x@fred` equivalent to `attr(x,"fred")`

  `x@fred <- v` equivalent to `attr(x,"fred") <- v`

- More convenient way to create lists:

```
L $$ a=1 $$ b=2   # equivalent to c(L,list(a=1,b=2))
$$ ab $$ cd $$ ef # equivalent to list(ab=ab,cd=cd,ef=ef)
```

- More convenient way to tack on attributes:

```
1..6 @@ dim=c(2,3) @@ class="fred"
```

# Vector / Array Constructors

The following possible syntax for constructing vectors or arrays is reminiscent of Python:

`v <- [ 7, 1, 9 ]`

result is like `v <- c(7,1,9)` but has 1D `dim` attribute of 3.

`a <- [ x, y, z ]`

dimensions/length of `x`, `y`, and `z` must match

result may be like `c(x,y,z)` or `rbind(x,y,z)`

`dim(a)[1]` will be always be 3.

`M <- [[x,0,0],[0,y,0],[0,0,z]]`

Creates a 3-by-3 matrix. No ambiguity with `[[`.

`x`, `y`, and `z` must be scalars.

# Flags for Arguments

Several needs could be addressed by allowing specification of certain "flags" on formal or actual arguments of functions.

Here's an example, using a `\notlazy` flag that disables lazy evaluation, and a counterpart `\lazy` that enables it.

```
# Define a function in which argument top defaults to lazy,
# and argument bottom to notlazy
f <- function (top, bottom \notlazy) ...


f (g(), h())              # h() evaluated immediately,
                          # g() only when argument is used
f (g() \notlazy, h())     # g() and h() both evaluated
                          # immediately
f (g(), h() \lazy)        # g() and h() both evaluated
                          # only when argument is used
```

# Argument Flags Relating to Matching and Evaluation

Here are some possible flags regarding argument matching and evaluation:

| | |
|---|---|
| `lazy, notlazy` | Is lazy evaluation used for this argument? |
| `exact, notexact` | Must the argument name match exactly? |
| `quoted, notquoted` | Should references default to the expression? |
| `ignorable` | Can this (actual) argument be ignored? |

If `x` is a quoted argument, `x` would give the expression passed, and `@x` would give its value. Some examples:

```
showmean <- function (x \quoted)
    cat ("The mean of", x, "is", mean(@x))


sym_ave <- function (M \quoted, i, j)
    (@M[i,j] + @M[j,i]) / 2
```

This is implemented, without the new syntax, in the quotedargs package I've put on CRAN.

# Another Use of Quoted Arguments

One could apply the quoted arguments idea to implement call-by-reference (really call-by-name, in Algol 60 terms).

Here's an example:

```
zero_corners <- function (M \quoted) {
    @M[1,1] <- 0
    @M[1,ncol(@M)] <- 0
    @M[nrow(@M),1] <- 0
    @M[nrow(@M),ncol(@M)] <- 0
}
```

A call of `zero_corners(A)` would modify the variable `A` in the caller's environment.

This isn't in the quotedargs package. (It's not possible with the current implementation of subset assignment.)

# Flags for Checking Argument Validity

The flags below could be used in a facility for conveniently and quickly checking whether arguments of functions are valid (all imply `notlazy`):

| | |
|---|---|
| `integer` | Must be convertable without loss to type integer |
| `real` | Must be convertable without loss to type real |
| `character` | Must be convertable without loss to type character |
| `scalar` | Must be a vector of length one |
| `vector` | Must be a vector with no dimensions or one dimension |
| `matrix` | Must be a matrix (two-dimensional array) |
| `positive` | All elements must be greater than zero |
| `noNA` | Must not have NA or NaN values |

Flags could be combined: `f <- function (x \scalar \integer \noNA)`

There'd be no "not" forms, since we don't want callers overriding checks.

Callers could add extra checks — eg, `fun (x \integer)` to give an error if `x` has non-integer elements, even if `fun` doesn't check. And in particular, `A <- B \real \matrix` would check that `B` is a real matrix before assigning.

# More Possible Language Extensions

- Support for automatic differentiation. Example:

```
loglik <- with_gradient (theta) {
   ll <- 0
   for (d in data) ll <- ll + log(model_prob(d,theta))
   ll
}
model_prob <- function (obs,theta)
   if (obs==1) theta else 1-theta
```

  The `loglik` variable will be assigned a value with a `gradient` attribute.

- Support for quickly recomputing a function of a vector `theta` after only one element of `theta` is changed. Would be very useful for MCMC, but I'm not sure how to do it.

- A way to return more than one item without putting them all in a list. Would allow what's returned to be extended without invalidating existing calls, like argument defaults allow what's passed to be extended.