The .External Interface I

- the R side of the .Exernal interface is almost the same as the .C interface, just change .C to .External like so:

```
dyn.load("prog3.so")

doStuff <-
    function (nIters, timeInSecs, propBurnIn)
{
    .External("do_stuff",
              as.integer(nIters),
              as.double(timeInSecs),
              as.double(propBurnIn))
}

doStuff(100, 10, 0.1)
```

- let this code live in a file called prog3.R and dump the C code in in a file called prog3.c

The .External Interface II

- lets look at the C side

```
#include <R.h>
#include <Rinternals.h>

SEXP
do_stuff (SEXP args)
{
        int n_iters;
        double time_in_secs, prop_burn_in;
        MonteCarloSpecs *mcs = NULL;

        args = CDR(args); n_iters = INTEGER(CAR(args))[0];
        args = CDR(args); time_in_secs = REAL(CAR(args))[0];
        args = CDR(args); prop_burn_in = REAL(CAR(args))[0];

        mcs = mcs_new(n_iters,
                      time_in_secs,
                      prop_burn_in);
        mcs_print(mcs);
        mcs_free(&mcs);
        return R_NilValue;
}
```

The `.External` Interface III

- the argument `args` is a `Lisp`-like `cons cell` object
- a quick intro to `Lisp`-like `cons` cell object:
  - `car`: "contents of address register", `cdr`: "contents of decrement register"
  - `> (setq a (cons (cons (cons 1 2) 3) 4))` sets `a` to `(((1 . 2) . 3) . 4)`
  - `> (car a)` gets us `((1 . 2) . 3)`
  - `> (cdr a)` gets us `4`
  - `> (cdr (car a))` gets us `3`
  - `> (cdr (car (car a)))` gets us `2` and so on

The .External Interface IV

- points to note about the `do_stuff` function:
  - note it takes only one SEXP type argument and returns SEXP type variable
  - as mentioned before, `args` is a `cons cell` like object and hence to get the individual components of the object we need to use code like the following:

    ```
    args = CDR(args); n_iters = INTEGER(CAR(args))[0];
    args = CDR(args); time_in_secs = REAL(CAR(args))[0];
    args = CDR(args); prop_burn_in = REAL(CAR(args))[0];
    ```
  - so an overhead of using the `.External` mode is that you need to first extract the arguments one by one as above
  - make sure the order of arguments extraction is same as that of argument passing in the corresponding `doStuff( )` function in R

The .External Interface V

- lets do something *semi*-useful with this interface:
- say in our Monte Carlo iterations we want to do the one of simplest things in the world: create an vector of first `n_iters` natural numbers and return it
- apart from that say we also attach an attribute to our output: a `class` attribute with value `sample` i.e. we want output of the form:

```
> dd <- doStuff(10, 10, 0.1)
> dd
 [1] 0 1 2 3 4 5 6 7 8 9
attr(,"class")
[1] "sample"
```

The .External Interface VI

```
SEXP
do_stuff (SEXP args)
{
        int ii, n_iters, nProtected = 0;
        double time_in_secs, prop_burn_in;
        SEXP retVec, className;

        args = CDR(args); n_iters = INTEGER(CAR(args))[0];
        args = CDR(args); time_in_secs = REAL(CAR(args))[0];
        args = CDR(args); prop_burn_in = REAL(CAR(args))[0];

        PROTECT(retVec = allocVector(REALSXP, n_iters));
        ++nProtected;
        for (ii = 0; ii < n_iters; ++ii)
          REAL(retVec)[ii] = ii;

        PROTECT(className = allocVector(STRSXP, 1));
        ++nProtected;
        SET_STRING_ELT(className, 0, mkChar("sample"));
        setAttrib(retVec, R_ClassSymbol, className);
        UNPROTECT(nProtected);
        return retVec;
}
```

The `.External` Interface VII

- points to note about the `do_stuff` function:
  - first we extract the arguments
  - any memory allocation for `R` objects should always be done within a `PROTECT` macro using `R` macros called `alloc***` e.g. `allocVector` in our case
  - number of `PROTECT` statements should be kept track of using a count: `nProtected`
  - once done working with the allocated objects one should use the `UNPROTECT` statement
  - note the `PROTECT` macro takes an expression whereas the `UNPROTECT` macro takes an integer
  - note we haven't really used the arguments `time_in_secs` and `prop_burn_in` they are just there for illustration purposes, in a real MCMC program you would definitely use all the arguments

The .External Interface VIII

- the need for using PROTECT and UNPROTECT macros is prevent R's in-built garbage collector "cleaning unused objects"
- PROTECT and UNPROTECT is *only* necessary for allocating R objects, i.e., allocating a new SEXP object
- for allocating C objects e.g. a new double * or Vector *, say, you may very well use your good-old malloc( ) or any other vector_new( ) function(s)

Code Files

prog3.c

prog3.R

prog4.c

prog4.R