The simples way to get started with the R/S-Perl interface is to use the *.PerlExpr()* which evaluates one or more Perl expressions given as a string.

```
[]
```

1 Network packages

In the following examples, we will work with some of the Perl modules that can be downloaded from `www.cpan.org`.

We make the contents of a Perl module available to the Perl session using the function *.PerlPackage()*. This is equivalent to the use command in Perl. In the first example (below), we load the Net::Domain package. Next, we call the `domainname()` in that package. There are two ways to do this. The first specifies the method name and the package.

```
[]
.PerlPackage("Net::Domain")
.Perl("Net::Domain::domainname")
 .Perl("domainname", pkg="Net::Domain")
```

Similarly, we can call the `hostname()` subroutine.

```
[]
.Perl("hostname", pkg="Net::Domain")
```

We can also use the Net::Time module to compute the time.

```
[]
.PerlPackage("Net::Time")
.Perl("Net::Time::inet_time", "khronos.ih.lucent.com")
.Perl("Net::Time::inet_daytime", "khronos.ih.lucent.com")
```

2 NNTP

For one project in which we were interested, we needed to be able to read newsgroups and look at the messages and then apply some statistical methodology to derived data. We could of course manually fetch the messages and store them in files and then read them into S. This is time consuming and error prone. We could even write a program to do this for us and then read the results into S. However, there are situations in which the statistical methodology dictates, at "run time", which messages to retrieve. In this case, it is simplest to have the mechanism to read these groups and their message history directly accessible to the statistical software. The Perl module News::NNTPClient provides such functionality.

```
[]
.PerlPackage("News::NNTPClient")
n <- .PerlNew('News::NNTPClient')
n$group("comp.lang.tcl")
```

Rather than retrieving the lines in the body of the message as a list containing strings, we may want to have it as a single string.

```
[]

paste(.Perl(n$body()), collapse="\n")
```

Now we position the "cursor" at the next article in the group. We use the Perl method next. Note that we cannot use the syntax n$next() as next is a reserved word in S!

```
[]
.Perl("next", ref=n)
```

3 Dates and Times

The package Date::Manip provides a number of easy to use functions/subroutines to (you guessed it) manipulate dates.

The point behind this example is not to provide a very fast replacement for `chron`. Instead, it shows the ease with which a Perl module can be used from R/S.

We start by loading the Date::Manip package into the Perl session.

```
[]
 .PerlPackage("Date::Manip")
```

In spite of this being a module, the subroutines are actually installed into the top-level Perl namespace. Thus, we can call the different "functions" easily via the *.Perl()* function. We start by parsing dates in different formats. We use the sub-routine `ParseDate()`. This returns the actual date as a string.

```
[]
> .Perl("ParseDate", "today")
[1] "2000103115:03:48"

> .Perl("ParseDate", "1st thursday in June 1992")
[1] "1992060400:00:00"
```

```
[]
> .Perl("Date_Cmp", .Perl("ParseDate", "today"), .Perl("ParseDate", "8:00pm december tenth"))
[1] -1
> .Perl("DateCalc", .Perl("ParseDate", "today"), .Perl("ParseDate", "8:00pm december tenth"))
[1] "+0:0:5:5:4:52:22"
> .Perl("DateCalc", "today","+ 3hours 12minutes 6 seconds");
[1] "2000103118:20:49"
```

```
> .Perl("DateCalc", "today","+ 3 business days")
[1] "2000110315:08:35"
```

Note that we are looking up the Perl subroutine `ParseDate()` by name in several calls. We can avoid this by retrieving it just once and storing it in R as a reference to a Perl object. We can then pass this to the different *.Perl()* calls and avoid resolving the routine each time.

We retrieve the routine reference via the function *.PerlGetCode()*. This returns an object of class

```
[]
 pc <- .PerlGetCode("ParseDate")

 .Perl(pc,"today")
```

Using this approach, we avoid the expense of repeated lookups but the syntax is still akward due to calling the *.Perl()* function. In some cases, it is more natural to think of the PerlCodeReference as a function and to be able to call it directly from R. For example, we might assign such a functional form to the *S* variable *perlDate* and call it as

```
[]
 perlDate("today")
```

To create such an R function which invokes a particular Perl subroutine we can again use *.PerlGetCode()*, but this time specify T for the value of the argument *asFunction*. Note that this is *R*-specific and does not work in SPlus.

```
[]
 perlData <- .PerlGetCode("ParseDate", asFunction=T)
```

4 Internationalization

The Date::Manip supports foreign languages and international format. To avail of this, one can initialize the package with different options.

```
[]
  .Perl("Date_Init", "Language=French","DateFormat=non-US")
```

Having initialized the package in this way, we can specify dates in French, such as

```
[]
  .Perl("ParseDate", "1er decembre 1990");
```