Chapter 12

hoc −− NEURON's interpreter

Much of the flexibility of NEURON is due to its use of a built−in interpreter for defining the anatomical and biophysical properties of models of neurons and neuronal networks, controlling simulations, and creating a graphical user interface. This interpreter, which is called `hoc` (pronounced "hoak"), is based on a floating point calculator by the same name that was developed by Kernighan and Pike [, 1984 #175]. The original `hoc` has a C−like syntax and is very similar to the `bc` calculator. The latest implementation of `hoc` in NEURON contains many enhancements and extensions beyond its original incarnation, both in added functions and additions to the syntax. Despite these enhancements, for the most part programs written for versions as far back as 2.x will work correctly with the most recent release of NEURON.

One important addition to `hoc` is an object−oriented syntax, which first appeared in version 3 of NEURON. Although it lacks inheritance, `hoc` can be used to implement abstract data types and encapsulation of data (see **Chapter 13**). Other extensions include functions that are specific to the domain of neural simulations, and functions that implement a graphical user interface. Also, the user can build customized `hoc` interpreters that incorporate special functions and variables which can be called and accessed interactively. As a result of these extensions, `hoc` in NEURON has become a powerful language for implementing and exercising models.

NEURON simulations are not subject to the performance penalty often associated with interpreted (as opposed to compiled) languages because computationally intensive tasks are carried out by highly efficient precompiled code. Some of these tasks are related to integration of the cable equation and others are involved in the emulation of biological mechanisms that generate and regulate chemical and electrical signals.

In this context, several important facts and their consequences bear special mention. First, a large part of what constitutes the NEURON simulation environment is actually written in `hoc`. This includes the standard run system, which is an extensive library of functions for initializing and controlling simulations (see **Chapters 7** and **8**), and almost the entire suite of GUI tools (the sole exception being the Print & File Window Manager, which is implemented in C). Second, the GUI tools for building models of cells and networks (which are, of course, all written in `hoc`) actually work by constructing `hoc` programs. Finally, and perhaps most important, all of the `hoc` code that defines the standard run system and GUI tools is provided in plain text files ("`hoc` libraries") that accompany the standard distribution of NEURON and are located in `nrn/share/nrn/lib/hoc/stdrun.hoc` under UNIX/Linux, or `c:\nrn\lib\hoc` in MSWindows. This makes it easy for users to review implementational details, and if necessary modify and replace functions and procedures that are normally defined in the standard libraries. Since `hoc` is an interpreter, it is also easy to replace these standard functions and procedures with user−defined alternatives; the only caveat is to make sure to load the alternatives *after* the standard library. For example, in order to replace the `init()` procedure (see **Chapter 7**), the text of the new procedure should occur sometime after the statement `load_file("nrngui.hoc")`. If the new definition is read prior to loading the library version, the library version will overwrite the user version instead of the other way around.

The interpreter

The `hoc` interpreter has served as the general I/O module in many kinds of applications, and as such is directly executed under many different names, but we will confine our attention to its use in NEURON. The simplest interface between `hoc` and domain−specific problems consists of a set of functions and variables that are callable from `hoc`. This was the level of implementation of the the original CABLE program (NEURON version 1). NEURON version 2 broke from this style by introducing neuron−specific syntax into the interpreter itself. This allowed users to specify cellular properties at a level of discourse more appropriate to neurons, and helped relieve the confusion and reduce the mental energy required to constantly shift between high level neural concepts and their low level representation on the computer. NEURON version 3 added object syntax that allows much better structuring of the conceptual pieces that are assembled in building a simulation.

Installing NEURON under UNIX or Linux results in the construction of several programs, but the principal one that we are concerned with in this book is `nrniv`. Located in `nrn/i686/bin`, this is the main executable, and it contains the `hoc` interpreter with all of its extensions. Under Linux, `nrniv` can add new mechanisms by dynamically loading shared objects that have been compiled from model description files (see **Adding new mechanisms** below; also see **Chapter 9**). For example, the demonstration program that comes with NEURON is started by executing `neurondemo`, which is actually a shell script that starts `nrniv` with a command line that makes it load a shared object that contains additional biophysical mechanisms. Under UNIX, however, shared objects are not used, so `nrniv` is a self−contained file, and `neurondemo` is a complete duplicate of everything in `nrniv` plus the extra mechanisms needed by the demonstration.

Under MSWindows, the program that corresponds to `nrniv` is `nrniv.exe`. There is also a program called `neuron.exe`, which is a short stub that starts a Cygwin terminal window (see `http://cygwin.com/`) and then runs `nrniv.exe` in that window. It is `neuron.exe` that is the target of icons and shortcuts used to start NEURON. As with the Linux version, `nrniv.exe` can load new mechanisms dynamically (see next section).

Adding new mechanisms to the interpreter

The shell script nrnivmodl, located in nrn/i686/bin, is used to add new mechanisms under UNIX and Linux. For example,

```
    nrnivmodl file1 file2 . . .
```

compiles the model descriptions defined in `file1.mod`, `file2.mod`, etc.. Under Linux, the result is a shared object located in a subdirectory `.i686/.libs` of the current working directory, and a shell script called `special` in the `.i686` subdirectory that starts `nrniv` and makes it load the shared object. Under non−Linix UNIX, the result is a complete executable called `special`. to `nrniv`.

If `nrnivmodl` is called with no file arguments, all the files in the current working directory that have the suffix `.mod` are compiled. Regardless of how `nrnivmodl` is invoked, the first step in the process is translation of the model model descriptions into C by the `nocmodl` translator. Models must contain a `NEURON` block that specifies the type of model (distributed mechanism or point process), the names of ions it uses, and which variables are to be treated as range variables.

The MSWindows version of `nrnivmodl` is called `mknrndll`. It compiles and links the models into a dynamically loadable library called `nrnmech.dll`. `neuron.exe` automatically looks in the current working directory for a `nrnmech.dll` file, and if one exists, loads it into memory and makes the mechanisms available to the interpreter. More than one `dll` file can be loaded by listing them after the `−dll` argument to `neuron.exe` when it is run.

The stand−alone interpreter

The rest of this chapter describes the general aspects of the interpreter which are common to all applications that contain it. Although for concreteness we use `nrniv` or `neuron.exe`, all the examples and fragments can be typed to any program that contains the interpreter, such as `oc`.

Starting the interpreter

Under UNIX and Linux, `hoc` is started by typing the program name in a terminal window

```
    nrniv [filenames] [−]
```

where the brackets indicate optional elements. When there are no filename arguments, `hoc` takes its commands from the standard input and prints its results on the standard output. With filename arguments, the files are read in turn and the commands executed. After the last file is executed, `hoc` exits. The `−` signals that commands are to be taken from the standard input until an EOF (`^D`). One can also exit by executing the `quit()` expression.

When starting `hoc` with arguments it is easy to forget the final `−` and be surprised when the program quickly exits, perhaps after putting graphs on the screen. Generally the `−` is left off only when running the interpreter in batch mode under control of a shell script.

The MSWindows version, `neuron.exe`, does not exit if the trailing `−` is left out. This makes it more convenient to attach `neuron.exe` to `hoc` files so one can merely click on the file name in a file manager. Also, `neuron.exe` starts a Cygwin terminal window into which one can type `hoc` commands. Exiting can be done by typing `^D` or quit() at the interpreter's `oc>` prompt, or by selecting File / Quit in the NEURON Main Menu (if the Main Menu is present).

On startup, neuron prints a banner showing the current version and last change date.

```
NEURON --  Version 5.4 2003/04/30
by John W. Moore, Michael Hines, and Ted Carnevale
Duke and Yale University -- Copyright 2001

        1
oc>
```

The `oc>` prompt at the beginning of a line means the interpreter is waiting for you to type a command. This is sometimes called "immediate mode" to signify that commands are evaluated and executed (if valid) immediately, as shown in the following listing (user entries are **`Courier bold`** while the interpreter's output is plain `Courier`).

```
oc>2
        2
oc>1+2
        3
oc>x=2
first instance of x
oc>x
        2
oc>x*x
        4
oc>
```

The interpreter has a "history" function that allows previous lines to be recalled by pressing the up arrow key on the keyboard. Multiple keypresses will recall earlier commands, and if you overshoot the mark, you can advance to later commands by pressing the down arrow key. This allows prior commands to be repeated with or without modification. For example, in

```
oc>proc foo() { print x^3 }
oc>foo()
8
oc>proc foo() { print x^4 }
oc>foo()
16
oc>
```

line 1 defines a new procedure that prints the value of x^3, line 2 calls this procedure, and line 3 shows the numeric result. The fourth line was created by pressing the up arrow key twice, to recall the first line. Then the left arrow key was pressed twice to move the editing cursor (blinking vertical line on the monitor) just to the right of the `3`. At this point, pressing the backspace key deleted the `3`, and pressing the numeral 4 on the keyboard changed the exponent to a `4`. Finally the return ("enter") key was pressed, and the interpreter responded with an `oc>` prompt. Now typing the command `foo()` produced a new numeric result.

In immediate mode, each statement must be contained in a single line. Very long statements can be assembled by using the continuation character \ to terminate all but the last line. Thus in

```
oc>proc foo() { print x^4 \
oc>, x }
oc>foo()
16 2
oc>
```

the interpreterer merges the first and second lines into the single line

```
proc foo() { print x^4 , x }
```

Quoted strings continued in this way have a limit of 256 characters and the newlines appear in the string as though `\n` was used.

Error handling

This is one of many areas where `oc` falls short. `oc` is a good I/O facility but a bad general purpose language. Debugging large programs in `oc` is difficult and it is best to keep things short.

`oc` is implemented as a stack machine. This means that commands are first parsed into a more efficient stack machine representation, and subsequently the stack machine is interpreted.

Errors during compilation are called parse errors and range from invalid syntax

```
oc>1++1
nrniv: parse error near line 3
1++1
   ^
oc>
```

to the use of undefined names

```
oc>print x[5], "hello"
nrniv: x not an array variable near line 9
print x[5], "hello"
           ^
```

These kinds of errors are usually easy to fix since they stop the parser immediately, and the error message, which always refers to a symptom, generally points to the cause. Error messages specify the current line number of the file being interpreted and print the line along with a carat pointing to the location where the parser failed (usually one or two tokens from the mistake).

Errors during interpretation of the stack machine are called run−time errors:

```
oc>sqrt(−1)
sqrt: DOMAIN error
nrniv: sqrt argument out of domain near line 5
sqrt(−1)
         ^
```

These errors usually occur within a function, and the error message prints the call chain

```
oc>proc p() {execute("sqrt(-1)")}
oc>p()
sqrt: DOMAIN error
nrniv: sqrt argument out of domain near line 8
{sqrt(-1)}
           ^
        execute("sqrt(-1)")
      p()
nrniv: execute error: sqrt(-1) near line 8
^
oc>
```

Unfortunately there is no trace facility to help debug run−time errors, and the line number refers to the latest command instead of the location of the offending statement.

Interpretation of a `hoc` program may be interrupted with one or two `^C` typed at the terminal. Generally, one `^C` is preferred because while the interpreter is busy it will wait until it reaches a safe place (won't be in the middle of updating an internal data structure) before it halts and waits for further input. Two situations may necessitate the second `^C`. If the program is waiting inside a system call, e.g. waiting for console input. If the program is executing a compiled function that is taking so long that program control doesn't reach a known safe place in a reasonable time. If the interpreter is in an infinite loop, as in

```
oc>while(1) {}
```

a single `^C` will stop it

```
^Cnrniv: interrupted near line 2
while(1) {}

oc>
```

Syntax

Names

A name is a string that starts with an alpha character and contains fewer than 100 alphanumeric characters or the underscore _. Names must not conflict with keywords or built−in functions. Names are global except when the `local` declaration is used to create a local scalar within a procedure or function, or unless the name is declared within a template (class). A user−created name can be associated with any one of the following:

global scalar (available to all procedures/functions)
local scalar (created/destroyed on procedure entry/exit)
array
string
template (class or type)
object reference

The `hoc` interpreter in the current version of NEURON has many keywords that have been added over the years. It is helpful to have a general idea of what these are useful for, and specific knowledge of where they are declared. This first table presents the most basic keywords and built−in constants and functions of the `hoc` interpreter with object extensions and elementary functionality for neuronal modeling; the authoritative list is in `nrn/src/oc/hoc_init.c`.

Control

return	break	stop	continue
if	else	for	while
iterator_statement			

General declaration

proc	func	local
double	strdef	iterator
eqn	depvar	

Miscellaneous

print	read	delete
em	debug	

Object−oriented

begintemplate	endtemplate	
public	external	
objectvar	objref	new

Built−in constants

PI	E	GAMMA	DEG
PHI	FARADAY	R	

Built−in functions

sin	cos	atan	
log	log10	exp	sqrt
int	abs	erf	erfc
use_mcell_ran4	mcell_ran4	mcell_ran4_init	
variable_domain	units		
prmat	solve	eqinit	
sred	xred		
chdir	getcwd	neuronhome	
ropen	wopen	xopen	
load_proc	load_func	load_template	
load_file	load_java		
getstr	strcmp		
printf	fprint	fscan	
ivoc_style			
save_session	print_session		
xpanel	xcheckbox		
xbutton	xstatebutton	xradiobutton	
xmenu	xlabel	xvarlabel	xslider
xvalue	xpvalue	xfixedvalue	
doEvents	doNotify		
numarg	symbols		

object_id	object_push	object_pop	
allobjectvars	allobjexts	name_declared	
boolean_dialog	continue_dialog	string_dialog	
pwman_place	startsw	stopsw	
execute	execute1		
machine_name	saveaudit	retrieveaudit	
show_errmess_always	coredump_on_error		
checkpoint	system	quit	

Built−in variables

float_epsilon	hoc_ac_	atan

Neuron−specific

create	connect		
access	setpointer		
insert	uninsert		
forall	ifsec	forsec	secname

This next table lists additional functions and variables that are specific to modeling neurons; the authoritative list of these is in `nrn/src/nrnoc/neuron.h`.

Variables

t	dt	secondorder	stoprun
celsius	diam_changed		

Functions

pt3dclear	pt3dadd	p3dconst	
x3d	y3d	z3d	diam3d
n3d	arc3d		
define_shape			
spine3d	setSpineArea	getSpineArea	
initnrn	distance	area	
topology	ri		
issection	ismembrane	sectionname	psection
disconnect	delete_section		
pop_section	push_section		
this_section	this_node		
parent_section	parent_node	parent_connection	
section_orientation			
ion_style	nernst	ghk	
finitialize	fadvance		
batch_run	batch_save		
fit_praxis	attr_praxis		
stop_praxis	pval_praxis		

Mechanism types and variables are defined in `nrn/src/nrnoc` by `capac.c`, `extcelln.c`, `hh.mod`, and `pas.mod`. This directory also contains several `mod` files that define neuron−specific point process classes such as `IClamp`, `SEClamp`, and `AlphaSynapse`.

There are also several other built−in object classes, such as the neuron−specific `SectionList`, `SectionRef`, and `Shape`, and more generic classes such as `List`, `Graph`, `HBox`, `File`, `Random`, and `Vector`. These are described in the Programmer's Reference (see http://www.neuron.yale.edu/neuron/docs/help/contents.html).

Variables

Double precision variables are defined when a name is assigned a value in an assignment expression, e.g.

```
var = 2
```

Such scalars are available to all interpreted procedures and functions. There are several built−in variables that should be treated as constants:

`FARADAY`	coulombs/mole
`R`	molar gas constant, joules/mole/deg−K
`DEG`	180 / PI, i.e. degrees per radian
`E`	base of natural logarithms
`GAMMA`	Euler constant
`PHI`	golden ratio
`PI`	circular transcendental number
`float_epsilon`	resolution for logical comparisons and int()

Arbitrarily dimensioned arrays are declared with the `double` keyword, as in

```
double vector[10], array[5][6], cube[first][second][third]
```

Array elements are initialized to 0. Array indices are truncated to integers and run from 0 to the declared value minus 1. When an array name is used without its indices, indices of 0 are assumed. Arrays can be dynamically re−dimensioned within procedures.

String variables are declared with the `strdef` keyword, e.g.

```
strdef st1, st2
```

Assignments are made to string variables, as in

```
st1 = "this is a string"
```

String variables may be used in any context which requires a string, but no operations, such as addition of strings, are available (but see sprint function).

After a name is defined as a string or array, it cannot be changed to another type. The `double` and `strdef` keywords can appear within a compound statement and are useful for throwing away previous data and reallocating space. However the names must be originally declared at the top level before redefining in a procedure.

Expressions

The arithmetic result of an expression is immediately typed on the standard output unless the expression is embedded in a statement or is an assignment expression. Thus

```
2*5
```

typed at the keyboard prints

```
10
```

and

```
sqrt(4)
```

yields

```
2
```

The operators used in expressions are, in order of precedence from high to low,

`()`	
`^`	exponentiation (right to left precedence)
`- !`	unaryminus, not
`* / %`	multiplication, division, "remainder"
`+ -`	plus, minus
`> >= < <= != ==`	logical operators
`&&`	logical AND
`||`	logical OR
`=`	assignment (right to left precedence)

Logical expressions have value 1.0 (`TRUE`) and 0.0 (`FALSE`). The remainder `a%b` is in the range 0 <= `a%b` < `b` and can be thought of as the value that results from repeatedly subtracting or adding `b` until result is in the range. This differs from the C syntax in which `(-1)%5` = −1. For us, `(-1)%5` = 4.

Logical comparisons of real values are inherently ambiguous due to roundoff error. Roundoff can also be a problem when computing integers from reals and indices for vectors. For this reason the built−in global variable `float_epsilon` is used for logical comparisons and computing vector indices. The constant ϵ in this table denotes `float_epsilon`, which has a default value is 10^{-11} but can be assigned a different value by the user.

hoc	**math or C equivalent**
`x == y`	$-\epsilon \le x - y \le \epsilon$
`x < y`	$x < y - \epsilon$
`x <= y`	$x \le y + \epsilon$
`x != y`	$x < y - \epsilon$ or $x > y + \epsilon$
`x > y`	$x > y + \epsilon$
`x >= y`	$x \ge y - \epsilon$
`int(x)`	(int)(x + ϵ)
`a[x]`	a[(int)(x + ϵ)]

Statements

A statement terminated with a newline is immediately executed. A group of statements separated with newlines or white space and enclosed in curly brackets `{}`

form a compound statement which is not executed until the closing `}` is typed. Statements typed interactively do not produce a value. An assignment is parsed by default as a statement rather than an expression, so assignments typed interactively do not print their values. Note, though, the expression

```
(a = 4)
```

would print the value

```
4
```

An expression is treated as a statement when it is within a compound statement.

Comments

Text between `/*` and `*/` is treated as a comment.

```
/* a single line comment */
/* this comment
   extends over
   several lines */
```

Comments to the end of the line may be started by the double slash, as in

```
print PI    // this is a comment
```

Flow control

In the syntax below, `stmt` stands for either a simple statement or a compound statement.

```
if (expr) stmt
if (expr) stmt1 else stmt2
while (expr) stmt
for (expr1; expr2; expr3) stmt
for var = expr1, expr2, expr3 stmt
for iterator_name( . . . ) stmt
```

In the `if` statement, `stmt` is executed only if `expr` evaluates to a non−zero value. The else form of the `if` statement executes `stmt1` when `expr` evaluates to a non−zero (TRUE) value and `stmt2` otherwise.

The `while` while statement is a looping construct which repeatedly executes `stmt` as long as `expr` is TRUE. The `expr` is evaluated prior to each execution of `stmt` so if `expr` starts as 0, `stmt` will not be executed even once.

The general form of the `for` statement is executed as follows: The first `expr` is evaluated. As long as the second `expr` is true the `stmt` is executed. After each execution of the `stmt`, the third `expr` is evaluated.

The short form of the `for` statement is similar to the DO loop of FORTRAN and is often more convenient to type. It is, however, very restrictive in that the increment can only be unity. If `expr2` is less than `expr1` the `stmt` will not be executed even once. Also the expressions are evaluated once at the beginning of the `for` for loop and not reevaluated.

The iterator form of the `for` statement is an object oriented construct that separates the idea of iteration over a set of items from the idea of what work is to be performed on each item. As such it is most useful when dealing with objects that are collections of other objects. It doesn't add any power to the language (neither does the `for i = 1,10` form), but it is useful whenever iteration over a set of items has a nontrivial mapping to a sequence of numbers and is used many times. As a concrete example consider the definition of an iterator called `case`

```
iterator case() {local i
   for i = $2, numarg()−1 {
      $&1 = $i
      iterator_statement
   }
}
```

Now it is easy to use this iterator to loop over small sets of unrelated integers as in

```
for case(&x, 1, −1, 3, 25, −3) print x
```

(assuming that `x` has already been used as a scalar variable). The alternative would be the relatively tedious

```
double num[5]
num[0] = 1
num[1] = −1
num[2] = 3
num[3] = 25
num[4] = −3
for i = 0, 4 {
   x = num[i]
   print x
}
```

These statements are used to modify the normal flow of control:

`break`	Exit from the enclosing while or for loop.
`continue`	Jump to end of stmt of the enclosing while or for.
`return`	Exit from the enclosing procedure.
`return expr`	Exit from the enclosing function.
`stop`	Exit to the top level of the interpreter.
`quit()`	Exit from the interpreter.

Functions and procedures

The definition syntax is

```
func name() {stmt}
proc name() {stmt}
```

Functions must return a value via a

```
return expr
```

statement. Procedures do not return a value. As a trivial example of a function definition, consider

```
func three() {
   return 3
}
```

This defines the function `three()` which returns a fixed numerical value. Typing the name of this function at the oc> prompt will cause its returned value to be printed.

```
oc>three()
    3
oc>
```

Notice the recommended placement of `{}`. The opening `{` must appear on the same line as the statement to which it is a part. This also applies to conditional statements. The closing `}` is free form, but clarity is best served if it is placed directly under the beginning of the statement it closes and interior statements are indented.

Arguments

Scalars, strings, and objects can be passed as arguments to functions and procedures. Arguments are retrieved positionally, e.g.

```
func quotient() {
  return $1/$2
}
```

defines the function `quotient()` which expects two scalar arguments. The `$1` and `$2` inside the function refer to the first and second arguments, respectively.

Formally, an argument starts with the letter `$` followed by an optional `&` to refer to a scalar pointer, followed by an optional `s` or `o` that signifies a string or object reference, followed by an integer. Thus a string argument in the first position would be known as `$s1`, while an object argument in the third position would be `$o3`.

For example,

```
proc printerr(){
  print "Error ", $1, "-- ", $s2
}
```

defines a procedure that expects a scalar for its first argument and a string for its second argument. If we invoke this procedure with the statement `printerr(29, "too many channels")`, it will print the message `Error 29 : too many channels`.

There is also a "symbolic positional syntax" which uses the variable `i` in place of the positional constant to denote which argument is to be retrieved, e.g. if `i` equals 2 then `$i` and `$2` refer to the same argument. The value of `i` must be in the range [1..`numarg()`], where `numarg()` is a built−in function that returns the number of arguments to a user−written function. This usage literally requires the symbol `$i`; `$` plus any other letter (e.g. `$j` or `$x`) will not work. Furthermore, `i` must be declared `local` to the function or procedure.

The function `numarg()` can be called inside a user−written function or procedure to obtain the number of arguments. Thus if we declare

```
proc countargs(){
  print "Number of arguments is ", numarg()
}
```

and then execute `countargs(x, sin(0.1), 9)`, where `x` is a scalar that we have defined previously, NEURON's interpreter will print `Number of arguments is 3`. Generally `numarg()` is used in procedures and functions that employ symbolic positional syntax, as in

```
proc printargs() { local i
  for i = 1, numarg() print $i
}
```

If we execute `printargs(PI, −4, sqrt(5))`, the interpreter will respond by printing

```
3.1415927
−4
2.236068
```

Similarly, we could define a function

```
proc printstrs() { local i
  for i = 1, numarg() print $si
}
```

and then execute `printstrs("foo", "faugh", "fap")` to get the printed output

```
foo
faugh
fap
```

Call by value vs. call by reference

Scalar arguments use call by value so the variable in the calling statement cannot be changed. If the calling statement has a `&` prepended to the variable, that variable is passed by reference and must be retrieved with the syntax `$&1`, `$&2`, etc..

If the variable passed by reference is a one dimensional array (i.e. a double), then `$&1` refers to its first (0th) element and index `i` is denoted `$&1[i]`. Be warned that there is ***no*** array bounds checking, and the array is treated as being one−dimensional. A scalar or array reference may be passed to another procedure with `&$&1`. To save a scalar reference, use the `Pointer` class.

Arguments of type `strdef` and `objref` use call by reference, so the calling value may be changed. Objects are discussed further in **Chapter 13**.

Local variables

Local variables maintained on a stack can be defined with the `local` statement. The `local` statement must be the first statement in the function and on the same line as the `proc` statement. For example, in

```
proc squares() { local i, j, k /* print squares up to arg */
   for (i=1; i <= $1; i=i+1) print i*i
}
```

making `i`, `j`, and `k` local insures that this procedure does not affect any previously defined global variables with these names.

Recursive functions

User defined functions can be used in any expression. Functions can be called recursively. For example, the factorial function can be defined as

```
func fac() {
   if ($1 == 0) {
      return 1
   } else {
      return fac($1−1)*$1
   }
}
```

and the call

```
fac(3)
```

would produce

```
6
```

It would be a user error to call this function with a negative argument or non−integer argument. Besides the fact that the algorithm is numerical nonsense for those values, in theory the function would never return since the recursive argument would never be 0. Actually, after some time the stack frame list would overflow and an error message would be printed as in

```
oc>fac(−1)
nrnoc: fac call nested too deeply near line 10
fac(−1)
        ^
        fac(−99)
      fac(−98)
    fac(−97)
  fac(−96)
and others
oc>
```

Input and output

The following describes simple text based input and output. User interaction is better performed with the graphical interface, and to deal with multiple files one must use the File class.

Standard `hoc` supplied `read` and `print` statements whose use can best be seen from the example

```
while (read(x)) {
   print "value is ", x
}
```

The return value of `read()` is 1 if a value was read, and 0 if there was an error or end of file. The `print` statement takes a comma separated list of arguments which may be strings or variables. A newline is printed at the end. `read` and `print` use the standard input and output respectively.

For greater flexibility the following builtin functions are available.

`printf("format string", arg1, arg2, . . . )`

`printf` is compatible with the standard C library function, allowing `f`, `g`, `d`, `o`, and `x` formats for scalar arguments, and the `s` format for strings. All the `%` specifications for field width apply.

`fprint("format string", arg1, arg2, . . . )`

`fprint` is the same as the `printf` function except that the output goes to the file opened with the `wopen("filename")` function. Files opened with the `wopen` function are closed with `wopen()` with no arguments or `wopen("")`. When no write file is open, `fprint` defaults to the standard output. `wopen` returns a 0 on failure of the attempted open.

`sprint(strdef, "format string", arg1, . . . )`

This function is very useful in building filenames out of other variables. For example, if data files are names `dthis.1`, `dthis.2`, etc., then the names can be generated with variables in the following fashion.

```
    strdef file, prefix
    prefix = "this"
    num = 1
    sprint(file, "d%s.%d", prefix, num)
```

After execution of these statements the string variable `file` contains `dthis.1`.

`fscan()`

`fscan` returns the value read sequentially from the file opened by `ropen("filename")`. The file is closed with `ropen()` or by a call to `ropen` with another filename. `ropen` returns a 0 if the file could not be opened. If no read file is open, `fscan` takes its input from the standard input.

Read files must consist of whitespace or newline separated numbers in any meaningful format. An EOF will interrupt the program with an error message. The user can avoid this with a sentinel value as the last number in the file or by knowing how many times to call `fscan`.

`getstr(strvar)`

`getstr` reads the next line from the file opened with `ropen` and assigns it to the string variable argument. The trailing newline is part of the string.

`xred("prompt", default, min, max)`

`xred` is a useful function that places a prompt on the standard error device along with the default value and waits for input on the standard input. If a newline is typed `xred` returns the default value. If a number is typed, it is checked to see if it is in the range defined by `min` and `max`. If so, the input value is returned. If the value to be returned is not in the range, the user is prompted again for a number within the proper range.

`xopen("filename")`

The file is read in and executed by `hoc`. This is useful for loading previously written procedures and functions that were left out of the command line during `hoc` invocation.

Editing

The `em` command invokes a public domain emacs editor that is similar, if not identical, to MicroEMACS. Readers who are interested in trying this editor will find a description of it in **Appendix III**. However, most users are already familiar with some other editor, and it is quite easy to transfer text files into `hoc` with the `xopen()` or `load_file()` command.