NESL User's Manual
(For NESL Version 3.1)

Guy E. Blelloch   Jonathan C. Hardwick
Jay Sipelstein   Marco Zagha

August 20, 1995

CMU-CS-95-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This manual is a supplement to the language definition of NESL version 3.1. It describes how to use the NESL system interactively and covers features for accessing on-line help, debugging, profiling, executing programs on remote machines, using NESL with GNU Emacs, and installing and customizing the NESL system.

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330 and contract number F19628-91-C-0168. It was also supported in part by an NSF Young Investigator Award and by Finmeccanica.

Keywords: Parallel programming languages, collection-oriented languages, remote execution, programming environments, supercomputers, nested parallelism

Contents

1 Introduction

This manual is a supplement to the language definition of NESL version 3.1 [1], and assumes that the reader is familiar with the basics of the language. It describes how to use the NESL system interactively and covers features for accessing on-line help, debugging, profiling, executing programs on remote machines, using NESL with GNU Emacs, and installing and customizing the NESL system.

NESL 3.1 is available via anonymous FTP and runs on serial workstations, Connection Machines CM-2 and CM-5, Cray Y-MP (8, C90, J90, EL) vector machines, the MasPar, and most machines supporting MPI such as the IBM SP-1 and SP-2 and the Intel Paragon. The normal mode of operation is for the interactive front end to run on a user's workstation and for the computational back end to run on a remote supercomputer. The NESL 3.1 system has the following features:

- Remote execution
- Profiling
- Tracing
- On-line documentation
- Background execution

1.1 System requirements

NESL assumes the machine on which it is running uses some variant of the Unix operating system. Building NESL requires a C compiler, `lex` and `yacc` and Common Lisp (GNU, Allegro, CMU, or Lucid). For the graphics functions it also requires the X11 library.

There is also a stand-alone version of NESL only requiring Common Lisp. This version implements all of NESL, but has no support for remote evaluation or graphics. See Section 5.4 for more on this version.

1.2 Other sources of information

There are several additional sources of information if you want to learn more about NESL:

- A World Wide Web home page for the SCANDAL project, which always contains the latest information on NESL. The URL is

  ```
  http://www.cs.cmu.edu/~scandal
  ```

  If you have any problems accessing the page send e-mail to `nesl-bugs@cs.cmu.edu`. This page contains links to many NESL related web pages including an on-line demo and tutorial, html manual, and many examples.

- Our FTP site (see Section 5.1). The WWW home page contains a link to this site.
- A mailing list used to discuss NESL and announce new patches and releases. If you want to be added to this list send e-mail to `nesl-request@cs.cmu.edu`.
- Papers on the implementation [2], uses [4], and teaching [3] of NESL. These can be obtained from the FTP site, or viewed directly from the WWW home page.

1.3 Conventions

Within this document, interactions with the NESL system are shown in a `typewriter font`, and command arguments are shown in an *italic font*. Enumerated choices are shown within curly braces {`on`,`off`}, and optional arguments are shown within square brackets [*var*].

2 Using NESL

2.1 Starting NESL

To start NESL, type `runnesl`. This should load a dumped Common Lisp image containing the NESL system. If `runnesl` isn't on your PATH, look for `bin/runnesl` in to the NESL distribution tree at your site. See Section 5 for instructions on installing NESL.

2.2 How NESL evaluates expressions

The NESL system is interactive: the current implementation is built on top of Common Lisp and implements a similar read-eval-print loop. Expressions are typed at the NESL prompt and terminated with a semicolon and a carriage return. For example:

```
<Nesl> 2 + 3;

Compiling..Writing..Loading..Running..
Exiting..Reading..
it = 5 : int
```

Expressions are compiled dynamically into an intermediate language called VCODE, which is then interpreted by a subprocess. The phases of executing an expression are:

- `Compiling`: Compiles the expression and any uncompiled needed to evaluate it into VCODE.
- `Writing`: Writes the compiled VCODE program out to a file.
- `Loading`: Starts up a subprocess for the VCODE interpreter and loads the VCODE program.
- `Running`: Subprocess executes the VCODE program.
- `Exiting`: Subprocess writes the results to a file.
- `Reading`: Reads the results back into the NESL system.

This setup makes it relatively easy to run code on remote machines, since the VCODE interpreter can be run remotely, communicating with the NESL system through a shared file system or through calls to `rsh` and `rcp`. This is how the CM-2, CM-5, Cray, Maspar, and MPI implementations work.

2.3 Top-level expressions and commands

At the NESL prompt you can type a NESL *top-level expression*, as defined by the language, or a *top-level command*, which is used to control or examine various aspects of the environment. The top-level commands are summarized in Figure 1 and most are described in Section 3.

A top-level expression is one of

```
NESL top-level forms:
  function <name> <pattern> [: <typespec>] = <exp>;  -- Function Definition
  datatype <name> <typeexp>;                         -- Record Definition
  <pattern> = <exp>;                                 -- Top level Assignment
  <exp>;                                             -- Any NESL expression

Top-level Commands:
  help; OR ?                  -- Print this message.
  load [<exp>];               -- Load a file.  If no arg, reload last file.
  describe <funcname>;        -- Describe a NESL function.
  apropos <name>;             -- List functions which contain <name>.
  set arg_check {on,off};     -- Set the argument check switch.
  set trace <funcname> <n>;   -- Trace a NESL function.
                                   0=off, 1=fname, 2=args, 3=vars, 4=vals
  set profile <funcname> {on,off}; -- Set profiling for function <funcname>.
  set print_length <n>;       -- Set maximum sequence print length.
  set verbose {on,off};       -- Set the verbose switch.
  set editor <pathname>;      -- Set the editor.
  show status;                -- List settings of current environment.
  show bugs;                  -- List the known bugs.
  show code <funcname>;       -- Show the code for a function.
  dump vcode <filename> <exp>;  -- Dump VCODE for <exp> to file <filename>.
  dump world [<filename>];    -- Dump current NESL environment to a file.
  dump info [<filename>];     -- Dump info for bug reports (default=stdout).
  edit [<filename>];          -- Edit & load a NESL file (default=last file).
  <pattern> |= <exp>;         -- Assign to a file variable.
  exit;                       -- Exit NESL.
  lisp; or ^D                 -- Go to the Common Lisp interpreter.

Commands for running VCODE on remote machines:
  defconfig <name> <args>; -- Define a new configuration.
  set config <config>;     -- Set the current configuration to <config>.
  set memory_size <n>;     -- Set the memory size of the current configuration.
  show config;             -- List the properties of the current configuration.
  show configs;            -- List the available configurations.
  <name> &= <exp> [,mem := <n>] [,max_time := <n>] [,config := <config>];
                           -- Execute exp in the background.
  get <name>;              -- Get a background result.
```

Figure 1: Top-level commands (screendump obtained by typing ?).

toplevel	`::=`	**`function name`** *pattern* `[:` *typedef*`]` `=` *exp* `;`	function definition
		`datatype name` *typedef* `;`	datatype definition
		pattern `=` *exp* `;`	variable binding
		exp `;`	expression

where *exp* is any expression and *pattern* can either be a single variable name or a parenthesized pattern of variable names (the square brackets indicate that the *typedef* in a function definition is optional). A full syntax for each of these is given in Appendix A of the NESL language definition [1]. Some examples of top-level expressions include:

```
function double(a) = 2*a;
function add3(a,b,c) = a + b + c;
datatype complex(float,float);
foo = double(3) + add3(1,2,3);
foo;
```

Expressions that are not assigned to a user defined variable are assigned to the variable `it`.

If you hit Return before an expression is completed, either for readability or by mistake, a "`>`" is printed at the beginning of each new line until the expression is completed. For example:

```
<Nesl> 2
> +
> 3;

Compiling..Writing..Loading..Running..
Exiting..Reading..
it = 5 : int
```

If you get lost, instead of hitting Ctrl-C try typing a few semicolons to end the expression.

For an example NESL session showing many features of the language, see Appendix A.

2.4 Errors

In NESL most errors result in an error message being printed, and the system returns you to the NESL prompt.

```
<Nesl> nosuchfunc(2);
Error at top level.
NOSUCHFUNC is undefined.
<Nesl>
```

Some errors, however, may cause you to abort out of NESL and back to the Common Lisp prompt. The only case where this is supposed to happen is if you hit Ctrl-C. If it happens in any other situation, please report it as a bug (see Section 6.2). When it does happen, you can return to NESL by getting back to the top level of your Common Lisp system and then typing `(nesl)`.

Running out of memory: The VCODE interpreter uses a fixed amount of memory for storing data. The default value depends on the configuration used (see Section 5.5), but is normally at least 1048576 (2^{20}) 64-bit words. It can be changed with the command `set memory_size` n (see Section 3.6).

If your program requires too much memory, you will get the following error,

```
<Nesl> [0:1000000000];

Compiling..Writing..Loading..Running..
compacting vector memory...done
vinterp: ran out of vector memory.  Rerun using larger -m argument.
Reading..
Error at top level.
Error: Error while running VCODE.
```

Before allocating more memory using `set memory_size` (see Section 3.5), think about why you are running out of memory. Your algorithm might require more memory than your machine can possibly supply. Or your algorithm might have a bug and be recursing infinitely. In any case, the memory size probably should not be set to more than your total physical memory. So if you have 16 Megabytes on your workstation, don't set `memory_size` to anything more than 2097152 (2 Megawords). If you want to find out more about the memory system used by NESL and the meaning of the `compacting vector memory` message, see the paper on the implementation of NESL [2].

Parse errors: Common syntax errors include functions with no arguments, mismatched parentheses, and empty vectors without types. Sometimes errors are a bit cryptic, for example:

```
function foo = sqrt(2.0);   ⇒   = is missing its left argument.
```

Most semantic errors (such as type mismatches) produce more informative error messages.

Garbage collection: The NESL system will occasionally pause because of a garbage collection by the underlying Common Lisp. This does not affect the operation of NESL programs, and in particular has no affect on the running times of programs.

2.5 The init file

When NESL is started, it loads the file `.nesl` (if it exists) from your home directory. This file should be in the same format as any NESL file—it can contain definitions as well as top-level commands. It is typically used to modify environment defaults such as the preferred configuration (`set config`), memory size (`set memory_size`), editor (`set editor`), and maximum print length for sequences returned at the top level (`set print_length`). These commands are all described later in this manual.

2.6 Exiting NESL

`lisp;`

Exits NESL to Common Lisp. Ctrl-D can also be used. To get back to NESL type `(nesl)`.

`exit;`

Exits both NESL and Common Lisp.

2.7 Variable and Function Redefinition

In most functional languages, when you define a variable with the same name as an existing variable, the new definition shadows the old definition but will not affect any previous references to the old variable. For example:

```
a = 22;
function foo(b) = a + b;
a = 1.0;
```

Now `a` is redefined to be 1.0, but `foo` would still refer to the value 22.

For the sake of convenience, NESL adds the feature that when you define a variable at the top level and then later redefine it with the same name and type, the system changes all previous references to the variable to the new value (note that function names are variables, so the same is true with function definitions). This allows the user to redefine a variable or function without having to reload everything that depends on it. It is important to realize that previous references to the variable are not redefined if the new value is of a different type, including the redefinition of a function to have a new type (since such a redefinition would lead to type inconsistencies), and that redefining only happens at the top level. The system warns the user when defining a variable with an existing name but a new type. For example:

```
<Nesl> x=2;
x = 2 : int
<Nesl> x=0.0;
Redefining X with a new type.  Old calls will not be modified.
x = 0.0 : float
```

3 Top-level Commands

The *top-level commands* are used for controlling and examining the NESL environment. They are not part of the NESL language and therefore are not found in the language definition. Top-level commands can be used either at the `<Nesl>` prompt, or at the top level within a file—they cannot appear within an expression.

3.1 On-line help

`help;`

Prints a list of all the top-level commands, as shown in Figure 1. The command `?` (with or without a terminating semicolon) has the same effect.

`describe` *funname*`;`

This gives a description of function *funname*, including the documentation from the manual [1].

`apropos` *name*`;`

This prints the names of all the NESL functions and variables that contain the string *name* in either their name or their documentation string.

`show code` *funname*`;`

This displays the NESL code for function *funname*. Code cannot be shown for primitive functions.

3.2 Loading NESL files

`load [`*exp*`];`

This loads a NESL file into the current environment. If present, the expression *exp* must evaluate to a string (sequence of characters) and be a valid filename. If the filename ends with the suffix "`.nesl`", the suffix can be omitted. If *exp* is omitted, it defaults to the last file that was loaded, or edited using `edit` (see Section 4.2). Files are loaded relative to the current directory. NESL files can contain any NESL top-level expressions or top-level commands; a file can therefore load other files, or set various environment variables, such as the memory size (see Section 3.5).

3.3 Customizing output

`set verbose {on,off};`

This turns the verbose mode on or off. When verbose mode is off, the NESL system no longer prints

```
Compiling..Writing..Loading..Running..
Exiting..Reading..
```

when evaluating an expression at the top level. Note that this command is local to a file, so that putting it in your `.nesl` file only turns verbose mode off while that file is being loaded. Verbose mode is often useful when debugging a new configuration (see Section 5.5).

`set print_length` *n*;

This sets the maximum print length for sequences returned at the top level. Only *n* elements of a sequence are printed on the screen, followed by "`...`". The default value for `print_length` is 100. The print length applies to each level of a nested sequence. For example:

```
<Nesl> set print_length 3;
<Nesl> x = [[1:10],[1:10],[1:10],[1:10]];
x = [[1, 2, 3,...], [1, 2, 3,...], [1, 2, 3,...],...] : [[int]]
```

pattern `|=` *exp*;

By typing `a |=` *exp*; at the top level, the expression *exp* is assigned to the *file variable* `a`. The *pattern* can be any variable pattern. This construct is useful for evaluating expressions with a large return value, because the user does not have to wait for the result to be read back into NESL—only the type is returned. You can use file variables in expressions just like any other variables. Here is an example:

```
<Nesl> a |= index(10000);
a : [int]
<Nesl> sum(a);
it = 49995000 : int
<Nesl> function foo(n) = take(a,n);
foo = fn : int -> [int]
<Nesl> foo(10);
it = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] : [int]
```

File variables are stored in the `temp_dir` specified by the configuration (`/tmp/` by default). This means that if you switch your configuration to a new configuration with a different `temp_dir`, the VCODE interpreter won't be able to find the variable and will give a runtime error message. Files created in this process are not removed by NESL.

3.4 Monitoring execution

`set arg_check {on,off};`

This turns argument checking on or off. Argument checking is on by default and includes bounds checking, divide by zero checking, and range checking. Runtime errors detected by argument checking print a message of the form:

```
<Nesl> let a = [2,3,4] in a[5];

Compiling..Writing..Loading..Running..
RUNTIME ERROR: Sequence reference (a[i]) out of bounds.
Exiting..Reading..
```

Argument checking takes time, so it can be turned off to generate faster code.

`set trace` *funname n*;

This sets the tracing level for any non-primitive function. Tracing is used for debugging and prints out a message each time the function is entered and exited. The argument n specifies the level of tracing. The choices are:

1 Print the function name when entering and exiting the function.

2 Print the function name along with the values of its arguments and the result.

3 Print the function name, and the variable names for each binding in the outermost `let` statement when it is assigned. This can be used to help locate a runtime error.

4 A combination of 2 and 3, plus it prints the value of each binding in the outermost `let` statement.

Tracing can be turned off for the function by specifying a trace level of 0. An example use is:

```
<Nesl> function norm(a,b) = sqrt(a^2 + b^2);

norm = fn : (float, float) -> float
<Nesl> set trace norm 2;

<Nesl> set trace sqrt 1;

<Nesl> norm(3.0,4.0);

Compiling..Writing..Loading..Running..
Entering NORM
  A = 3.000000e0
  B = 4.000000e0
Entering SQRT
Leaving SQRT
Leaving NORM
  RESULT = 5.000000e0
Exiting..Reading..
5.0 : float
```

NESL primitives cannot be traced. If you would like to trace a primitive, you can create a stub function that calls the primitive, and then replace calls to the primitive with calls to the stub. For example:

```
<Nesl> set trace sin 4;

SIN is a primitive, you cannot trace it.

<Nesl> function my_sin (x) = sin(x);

<Nesl> set trace my_sin 4;
```

When tracing prints values, it truncates sequences and only prints the first few elements. It also only prints nested sequences down to a fixed depth. The truncation of sequences for tracing is different than the print-length set by `set print_length`, and is controlled by the two variables, `trace_string_length` and `trace_string_depth`. They are ordinary NESL variables and should be set at top level using *var* `=` *val*.

When tracing functions that are called in parallel, NESL will only print an "`entering foo`" message once even though the functions is being entered many times in parallel. It will, however, print any values (arguments, results, or values of `let` statements) for each parallel call. For example:

```
<Nesl> function foo(a) = a + 3;

foo = fn : int -> int
<Nesl> set trace foo 2;

<Nesl> {foo(a): a in [2,3,4]};

Compiling..Writing..Loading..Running..
Entering FOO
  A = 2
  A = 3
  A = 4
Leaving FOO
  RESULT = 5
  RESULT = 6
  RESULT = 7
Exiting..Reading..
it = [5, 6, 7] : [int]
```

`set trace off;`

Turns tracing off for all functions.

`set profile` *funname* `{on,off};`

When profiling is turned on for a function, the time taken for each expression on the right of a `let` binding in that function are printed. The profiling also works on functions that are called in parallel: it prints the total time taken across all parallel calls.

In the example below, suppose we would like to profile a function that scrambles the order of elements in a vector. We could first use the `time` function to measure the total running time:

```
function scramble (vec) =
  let
    n = #vec;
    rands = {rand(j): j in dist(n,n)};
    random_permutation = rank(rands)
  in
    vec->random_permutation;

<Nesl> time(#scramble([100000:200000]));

it = (100000, 0.042053647356) : int, float
```

To find out where the time is going, we turn on profiling as shown below. The timings indicate that the `rank` function is consuming most of the running time.

```
<Nesl> set profile scramble on;

Turning timing on for each let binding of SCRAMBLE.

<Nesl> #scramble([100000:200000]);

5.106242e-5 seconds for expression:
      #vec
5.976757e-3 seconds for expression:
      {rand(j): j in dist(n, n)}
3.046853e-2 seconds for expression:
      rank(rands)
3.965526e-3 seconds for expression:
      vec -> random_permutation
```

To get more accurate profiling, we turn argument checking off. Note that some functions are much faster with argument checking off.

```
<Nesl> set arg_check off;

<Nesl> #scramble([100000:200000]);

5.183748e-5 seconds for expression:
      #vec
3.150965e-3 seconds for expression:
      {rand(j): j in dist(n, n)}
3.034669e-2 seconds for expression:
      rank(rands)
8.495138e-4 seconds for expression:
      vec -> random_permutation
```

It should be noted that timing will give erroneous numbers when nested. This means that if you profile a recursive function, the times for the `let` bindings that do not make recursive calls will be accurate, but the time for the recursive call itself will meaningless. It should also be noted that to get reasonably accurate timings, the function should be profiled a few times, since the time taken by an expression can vary depending on the system load.

Redefining monitored functions: When you redefine a function with a new type, old calls to the function are still traced or profiled but the new version will not be monitored.

`set profile off;`

Turns profiling off for all functions.

3.5 Configurations and remote execution

In NESL it is possible to evaluate any top-level expression on a remote machine. The remote machine can be another workstation, a Cray Y-MP, a CM-2 or CM-5, a Maspar, or one of the supported MPI machines. To run expressions on a remote machine, configurations first need to be set up using `defconfig` (see Section 5.5). Assuming that the configuration files have already been set up, this section describes the top-level commands used for remote execution.

`show configs;`

This displays a list of currently-available machine configurations. Note that the same physical machine might be in multiple configurations, depending on the turnaround time requested for a job, how many processors are used, etc.

```
<Nesl> show configs;
The current machine configurations are:
LOCAL
CRAY
CM5
To use one type: set config <config>;
```

`set config` *config*;

This causes all future NESL expressions (until the next `set config` command) to be evaluated on the machine configuration *config*. (In order to use NESL graphics, you must first give the remote machine access to your X server by executing the command `xhost + remote_machine` in a local shell.)

```
<Nesl> set config cray;
[...]
<Nesl> 2 + 3;

Compiling..Writing..Loading on PSC Cray C90 (mario)..
Running..
Exiting..Reading..
it = 5 : int
```

`show config;`

This displays parameters of the current machine configuration. At a minimum, it displays the name of the configuration, the default memory size, the path to the VCODE interpreter, the path to the X11 graphics interface program `xneslplot` (see Section 5.5), and the directory used by the NESL system for temporary files. Remote configurations may also include the name of the remote machine, the `rsh` command used to start up the VCODE interpreter, and the name of the script used to submit batch jobs.

```
<Nesl> show config;
Configuration Name: "cray"
interp_file:        "/afs/cs.uvwx.edu/user/joeuser/nesl/bin/vinterp.cray"
memory_size:        3600000
temp_dir:           "/afs/cs.uvwx.edu/user/joeuser/tmp/"
plot_file:          "/afs/cs.uvwx.edu/user/joeuser/nesl/bin/xneslplot.cray"
rsh_command:        "rsh -l joeuser mario.psc.edu"
machine_name:       "CRAY C90 at PSC"
background_command: "background-cray"
```

`set memory_size` *n*;

This sets the amount of memory that the VCODE interpreter allocates for data storage in the current configuration. In the standard configurations, it defaults to 1048576 (2^{20}) 64-bit words. See Section 2.4 for what happens when you run out of memory.

3.6 Background execution

NESL allows background execution of expressions. This is most useful when evaluating expressions that might take a long time to complete. It is also useful on supercomputers that allow more machine resources (memory, processors, runtime) to be used for batch jobs than for interactive jobs.

name `&=` *exp* `[,mem :=` *n*`] [,max_time :=` *n*`] [,config :=` *config*`];`

This evaluate the expression *exp* in the background and assigns the result to the background variable *name*. The NESL prompt returns soon as the job has been submitted. For example:

```
<Nesl> a &= sum([0:100000]);

Compiling..Writing..Submitting..
[1] 12782
background a : int
<Nesl>
```

The result is retrieved using the `get` command described below. The `&=` command has three optional arguments:

- `mem`: This specifies the amount of memory the job will need. It uses the same units as used by the `mem_size` field of a configuration, and defaults to the value specified in the current configuration.
- `max_time`: This specifies the maximum amount of time (in seconds) that the job will run. It is normally used when submitting jobs on supercomputers, since it can be used as a safety cap in case the job goes into an infinite loop.
- `config`: This is used to specify a configuration other than the current configuration on which to run the job.

`get` *name*;

This is used to retrieve a background variable. If the job is not completed, the message "`Variable` `waiting for result`" will be printed. If the job is completed, all output generated during execution will be printed, and if there was no error the result will be assigned to the variable *name*. The status of background jobs can be examined with the `show status` command described below.

3.7 Checking interpreter status

`show status;`

This command will report the current `print_length` and whether verbose mode and argument checking are turned on. It will also list all the functions that are being traced or profiled, and all the outstanding background variables and whether the corresponding jobs have completed.

```
<Nesl> show status;
verbose on
arg_check on
print_length = 100
traced functions:
  sort
background variables:
  X (done)
  Y (waiting on PSC Cray C90 (mario))
```

3.8 Saving NESL state

`dump world` [*filename*]`;`

This dumps an executable Common Lisp image containing all of the current NESL environment to *filename*. If *filename* is not specified it defaults to `bin/runnesl` relative to the NESL distribution tree. The NESL image can then be run directly rather than entering Common Lisp and loading the NESL source files. Note that the image will typically be very large (from 3 to 35 Megabytes) and might take a long time to dump. Also, since it contains everything defined in the current session, you might want to start a fresh NESL before executing `dump world`.

`dump vcode` *filename exp*`;`

In normal operation the NESL system deletes the intermediate VCODE files after they have been read. This command writes a permanent copy of the VCODE program that evaluates *exp* to the file *filename*. It is normally used for reporting bugs (see Section 6.2). It can also be used for creating stand-alone VCODE applications, which can be executed by the VCODE interpreter outside of the NESL system. This is useful for improving the startup time and memory usage of applications once they have been debugged in the NESL system.

4 Editor Support

4.1 Using NESL with GNU Emacs

Within the top-level NESL directory there should be a subdirectory named `emacs` which contains the files `nesl-mode.el` and `nesl.el`, written by Tom Sheffler. If you use `M-x load` to load these files into your GNU Emacs, all files whose names end with `.nesl` will from then on be edited in nesl-mode (see the comments at the top of `nesl-mode.el` for how to load these automatically). This mode adds the following functions to GNU Emacs:

`TAB`	Adjust indentation of current line.
`C-M-x`	Evaluate the function containing or after point, and send it as input to the NESL process (nesl-send-defun).
`C-M-a`	Move to the beginning of current or preceding function (beginning-of-nesl-function).
`C-M-e`	Move to the end of current or following function (end-of-nesl-function).
`C-c t`	Insert function type for the function containing or after the point (nesl-insert-function-type).

For nesl-mode to find the end of a function, the function needs to be terminated with a dollar-sign (`$`) sign instead of a semicolon (`;`). In NESL, the dollar-sign and semicolon can be used interchangeably to mark the end of a function definition. `C-M-x` only works if it can find the end of the function.

The following variables can be set by the user:

Variable	Default	Documentation
nesl-process	`"nesl"`	String name of the inferior NESL process.
nesl-mode-hook	nil	Function hook called on entry to nesl-mode.
inferior-nesl-program	`"runnesl"`	Program to execute on `M-x run-nesl`.
nesl-indent-level	4	Indentation to be used inside NESL blocks.

A NESL subprocess can be started with `M-x run-nesl`.

4.2 Using NESL with other editors

`set editor` *command*;

This sets the command line that gets invoked when using `edit`. The argument *command* must be a string. When you run `edit`, it will prepend the string to the filename specified (with a space) and run it as a shell command. The command can therefore include flags. For example, "`set editor "xterm -e vi";`" will set up the editor so that it will invoke `vi` within a new xterm.

`edit [`*filename*`];`

This starts the editor (which must have previously been set with `set editor`) on the NESL file *filename*. If no filename is specified, it defaults to the last file loaded. If the filename ends with the suffix "`.nesl`", the suffix can be omitted. When you exit from the editor, you will be asked if you want to load the file you just edited.

5 Installing NESL

5.1 Getting the files

FTP to `nesl.scandal.cs.cmu.edu` (currently 128.2.198.40), login as `anonymous`, enter your e-mail address as the password, `cd code/nesl`, and `get nesl.tar.Z`[1]. Depending on your FTP client you may need to set the transfer mode to binary first. Finally, uncompress and untar the file.

```
% ftp nesl.scandal.cs.cmu.edu
Name: anonymous
Password: me@my.site.name
ftp> cd /afs/cs/project/scandal/public/code/nesl
```

[1]If you have GNU gzip, nesl.tar.gz is also available, and should be more compact)

```
ftp> binary
ftp> get nesl.tar.Z
ftp> quit
% uncompress nesl.tar
% tar -xf nesl.tar
```

The system requires about 3.5 Megabytes of disk space uncompressed and without binaries. Some of this can be removed if you are not going to be using all of the parallel machines (see the `cvl` directories in the next section), or if you don't need to keep the manuals. The biggest use of space will be for the dumped Common Lisp image you will make after building NESL; this will occupy from 3 to 35 Megabytes, depending on the version of Lisp you use and the machine on which you are running (see the discussion of building the system in Section 5.3).

5.2 Structure of NESL distribution

The nesl distribution unpacks into the following directory tree. The files in *slanted font* will be created during the build process.

`BUILD`	Directions on how to build NESL
`COPYRIGHT`	
`Makefile`	
`README`	
`bin/`	
runnesl	The NESL executable
*vinterp.**	The VCODE interpreter, for various architectures
xneslplot	X11 interface used for NESL graphics
`foreground-*`	Scripts for executing VCODE from NESL in the foreground
`background-*`	Scripts for executing VCODE from NESL in the background
`config.nesl`	Definitions of configurations
`cvl/`	
`cm2/`	Source code for the CM-2 version of CVL
`cm5/`	Source code for the CM-5 version of CVL
`cray/`	Source code for the CRAY version of CVL
`mpi/`	Source code for the MPI version of CVL
`serial/`	Source code for the serial version of CVL
`doc/`	
`cvl.ps`	The CVL manual
`manual.ps`	The NESL manual
`user.ps`	This user's guide
`vcode-ref.ps`	The VCODE manual
`emacs/`	NESL editing mode for GNU Emacs
`examples/`	Collection of NESL examples (see Appendix A)
`include/`	`cvl.h` include file
`lib/`	
libcvl.a	The CVL library
`neslseqsrc/`	Source code for stand-alone NESL
`neslsrc/`	Source code for NESL

`release.notes`	List of changes since the last release
`utils/`	Source code for `xneslplot`
`vcode/`	Source code for the VCODE interpreter

5.3 Building NESL

Once you have unpacked the NESL distribution, the following steps should be sufficient to build a version of NESL to run on your local workstation:

1. Run `make` from the top-level NESL directory. This builds CVL, VCODE, and `xneslplot`, leaving `vinterp.serial` and `xneslplot` in the `bin` directory.

2. Start a Common Lisp (either GNU, Allegro, CMU, or Lucid) in the top-level NESL directory, and enter `(load "neslsrc/build.lisp")`.

3. Follow the instructions for dumping an executable version of NESL. This will create a file `bin/runnesl`, which can be executed directly to start NESL.

4. The simplest test of the system is to enter `1+1;`, which should exercise all the phases of the system as explained in Section 2.2. For a more complete test, try `load "neslsrc/test";` followed by `testall(0);`, which runs through a series of test functions.

The rest of this section discusses what can be changed if the above procedure does not work or if you don't want to create a dumped NESL Lisp image. The next section discusses how to set up configurations for remote execution.

The C compiler: The default C compiler and optimization level is `gcc -O2`. This can be changed by setting the variable `CC` at the top of the `Makefile`.

Making CVL, VCODE and `xneslplot` separately: The top-level `Makefile` recursively calls `make` in the subdirectories `cvl/serial`, `vcode`, and `utils`, and then moves the result to either the `lib` or `bin` directory. These builds can also be done by hand if necessary.

Compiling CVL and VCODE for a Connection Machine, Cray, or Maspar: To build a version of the VCODE interpreter for a CM-2, CM-5, Cray or Maspar, you will need to `make cm2`, `make cm5`, `make cray` or `make maspar`.

Compiling CVL and VCODE for MPI: To build a version of the VCODE interpreter using MPI, you will need to do some customization. Read the `cm5_mpi`, `paragon_mpi` and `sp1_mpi` entries in the top-level `Makefile`, and modify one of them (and the appropriate `mpiCC` and `mpidir` variables) to match your MPI installation. Note that only the MPICH implementation of MPI from Argonne/Mississipi State is directly supported by this release, and that ANSI C is required. See the `Makefile` and `README` files in the `vcode` and `cvl/mpi` subdirectories for further details. A comparison of the CM-2, CM-5 and MPI versions is given in [5].

Compiling for multiple serial architectures: If you want versions of NESL for multiple serial architectures, you will need copies of `vinterp.serial`, `xneslplot` and `runnesl` for each architecture. If you are using a shared file system, before building a version for a new machine, you should run `make clean` from the top-level directory to clean up any old object files. Before dumping the executable image of NESL (`runnesl`), you should make sure that the configuration points to the correct versions of `vinterp.serial` and `xneslplot` (see the next section for an explanation of configurations).

Compiling for Linux To compile NESL for Linux, do a `make linux`. The Linux version assumes you have Gnu Common Lisp, the Gnu C compiler, flex and bison.

Avoiding dumping NESL: Because Common Lisp images can be quite large, the NESL executable (`runnesl`) may require a comparatively large amount of space: between 3 and 35 Megabytes (the exact amount depends on which version of Common Lisp you are using, and can be as big as 35 Megabytes for Lucid Common Lisp). If this is too much space for your liking, then the variable `*nesl-path*` should be set in the file `neslsrc/load.lisp`, and the user should start up Common Lisp and load `neslsrc/load.lisp` each time they want to run NESL. This will load in the compiled files and be much faster than doing a build, but not as fast as starting up a dumped executable.

Portable pseudorandom numbers: For users who want a pseudorandom number generator that is portable across parallel machines, we supply hooks in the MPI and CM-5 code to use the additive lagged-Fibonacci generator described in [6]. This is available to US residents via FTP from `ftp://ftp.super.org/pub/mascagni/lfibrng6a.tar.Z`. See the appropriate CVL `Makefile` for instructions on how to enable the hooks.

What can be deleted?: After building working versions of the three binaries `vinterp.serial`, `xneslplot` and `runnesl`, the only files that are necessary for running NESL are all in the `bin` directory: anything else can be deleted, if so desired. If you run `make clean` in the top-level directory, it will remove all unnecessary object files.

5.4 Building Stand-alone NESL

This release of NESL includes an experimental stand-alone verson of the language that does not use VCODE or CVL. This version runs inside a Common Lisp environment and is limited in several respects:

1. no graphics
2. no remote execution
3. no tracing
4. no profiling
5. no `spawn` function
6. significantly slower on large problems

However, this version is much easier to build and is a good way for a new user to experiment with NESL.

To build the stand-alone NESL, first `cd neslseqsrc` and then start up Common Lisp. Load the NESLfiles into the lisp with `(load "neslseqsrc/load.lisp")`. To start up NESL, type `(nesl)`.

5.5 Machine configurations

The top-level NESL command `defconfig` is used to define new configurations. Machine configurations are mostly used for remote execution, but can also be used to define aspects of the environment for local execution, such as the amount of memory allocated by the VCODE process, or a VCODE interpreter file other than the default. This section describes `defconfig` and outlines how remote execution is implemented. The mechanism for remote execution was designed to be quite flexible; we hope that you will be able to adapt it to the idiosyncrasies of your environment.

Machine configurations for your local site should be defined in the `config.nesl` file in the top-level directory. This file includes several example configurations. Users can also specify their own configurations by using `defconfig` either from the interpreter or within a file (it is common to put `defconfig`s within the `.nesl` init file). The syntax for `defconfig` is:

<u>`defconfig` *name*</u> `[,memory_size :=` *n*`]` `[,interp_file :=` *str*`]` `[,temp_dir :=` *str*`]`
`[,rsh_command :=` *str*`]` `[,plot_file :=` *str*`]` `[,machine_name :=` *str*`]`
`[,foreground_command :=` *str*`]` `[,background_command :=` *str*`]`
`[,max_time :=` *n*`]` `[,arguments :=` *str*`];`

This command takes several optional arguments, described below. These optional arguments can appear in any order. In its simplest form, `defconfig` *name*, it defines a configuration with all the default settings. In the following, *nesl_path* refers to the pathname to the nesl distribution.

- `memory_size`: This specifies the memory size used by the VCODE interpreter. The default is 1048576 (2^{20}) double precision floating point numbers (64 bits each on most machines).

- `interp_file`: The executable file for the interpreter for this configuration. The default is *nesl_path*`/bin/vinterp`.

- `temp_dir`: This is the directory to which the VCODE source file and the output from the VCODE interpreter are written each time an expression is executed. If the remote machine shares a file system with the local machine, this should be a shared directory so that both NESL (running locally) and the VCODE interpreter (running remotely) can access it. The default is `/tmp/`.

- `rsh_command`: This is the command used to initiate remote execution. If both the local and remote machine support `rsh` and the user name is the same locally as remotely, then this can simply be `rsh` *machinename*. If the remote user name is different, then `rsh -l` *username machinename* should work. Of course, the user needs to set the `.rhosts` appropriately at the remote host (see your local manual page for `rsh(1)`, and note that some sites may restrict its use because of security considerations). If your system does not support `rsh`, but supports some other command for remote procedure calls, it may be possible to substitute that. The default for `rsh_command` is the empty string, specifying that the VCODE interpreter should be executed locally.

- `machine_name`: This is used to specify the remote machine's name. It is only used to print messages for the user, so it need not be the "official" name. The default is the empty string.

- `plot_file`: The NESL graphics routines work by starting a subprocess from within the VCODE interpreter using the `spawn` function. This subprocess is then passed commands from the interpreter through a pipe to its standard input, and translates them into X11 calls. The file that is used to invoke the subprocess is specified by `plot_file`. The default is *nesl_path*`/bin/xneslplot`.

- `foreground_command`: The shell script used for foreground execution. The script is first searched for on the user's path, and then in `*nesl_path*/bin`. The default is `foreground-unix`.

- `background_command`: The shell script used for background execution. It is searched for as above. The default is `background-unix`.

- `max_time`: The maximum number of seconds allowed for background jobs. This can be overridden with the `max_time` option of the *name* `&=` command.

- `arguments`: This string is passed directly to the shell scripts specified by `foreground_command` and `background_command`. It can be used for various purposes, such as specifying the number of processors for the CM-2 or CM-5 implementations. The default is the empty string.

Remote execution works as follows: After NESL writes out the vcode file, it starts up a subprocess by executing one of the `background-*` or `foreground-*` scripts from the `bin` directory. The script to be used is specified by the `foreground_command` and `background_command` in the configuration definition. NESL passes these scripts the following 7 arguments:

1. `rsh_command`: This is passed directly from the configuration. The scripts are set up so that if this argument is the empty string, the interpreter will run locally.

2. `interp_file`: Passed directly from the configuration.

3. `memory_size`: Passed directly from the configuration.

4. `temp_dir`: Passed directly from the configuration.

5. `job_ident`: A unique job identifier (used to generate filenames).

6. `max_time`: Passed directly from the configuration.

7. `arguments`: Passed directly from the configuration.

NESL always writes the VCODE program to the file *temp_dir*/*job_ident*`_code`. VCODE, in turn, writes the result to the file *temp_dir*/*job_ident*`_output`, where NESL expects to find it. With background mode, two additional files *temp_dir*/*job_ident*`_err` and *temp_dir*/*job_ident*`_check` are created. The `err` file is used to store all output generated during the execution of the VCODE interpreter, including any errors. The `check` file is written to after the VCODE interpreter has completed, and is used so that NESL can determine when it can read the result. It should be noted that if the job was successful, all these files are deleted after being read.

It might be necessary to create new *background* and *foreground* scripts for your local site. Looking at the existing scripts should help in defining new ones.

6 Bugs

6.1 Current bugs

`show bugs;`

This displays a list of known bugs, and possible workarounds, in the current release of the NESL system.

6.2 How to report bugs

If you find a bug not listed by `show bugs`, please send a bug report to `nesl-bugs@cs.cmu.edu`. You can help us identify and correct the bug by first finding the smallest example demostrating the bug and by including the following information in your bug report. Include the NESL source, and in addition, the VCODE output from the NESL compiler using the `dump vcode` command. Use the `dump info` command to generate a description of your Lisp and hardware platform. If you found the bug on a parallel machine, does your local serial configuration exhibit the same problem?

We will try to respond to your bug report promptly, but can make no guarantees – NESL is a research tool rather than a production system.

If you make any improvements to the system, please send them to us so we can incorporate them in future versions.

Acknowledgements

Margaret Reid-Miller and Girija Narlikar provided useful comments on this manual. Tom Sheffler implemented the GNU Emacs nesl-mode. Martin Santavy suggested several of the features included in the NESL system.

References

[1] Guy E. Blelloch. The NESL language definition (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, July 1995.

[2] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[3] Guy E. Blelloch and Jonathan C. Hardwick. Class notes: Programming parallel algorithms. Technical Report CMU-CS-93-115, School of Computer Science, Carnegie Mellon University, February 1993.

[4] John Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings Sixth Annual Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.

[5] Jonathan Hardwick. Porting a vector library: A comparison of MPI, paris, CMMD and PVM. Technical Report CMU-CS-94-200, School of Computer Science, Carnegie Mellon University, November 1994.

[6] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni, and M.L. Robinson. Implementation of a portable and reproducible parallel pseudorandom number generator. In *Proceedings Supercomputing '94*, pages 311–319, Washington D.C., December 1994. ACM.

A Nesl Examples

A.1 Example code in distribution

The subdirectory `examples` contains the following examples of Nesl code:

adaptive-integration Adaptive integration of single-variable functions.

awerbuch-shiloach Algorithm for finding the connected components of a graph.

convex-hull The QuickHull algorithm for finding convex hulls. Includes a graphics demo.

hash-table An implementation of a parallel hash table.

line-fit Least-squares fit of a line to a set of points.

micro-shell A micro shell that keeps track of current directory and executes commands.

nas-cg The NAS conjugate gradient benchmark.

median Recursively finds the k^{th} largest element of a vector.

primes Work-efficient parallel implementation of the prime sieve algorithm.

separator Geometric separator code. Includes a graphics demo of it running on an airfoil.

sort Various sorts: quicksort, Batcher's bitonic sort and Batcher's odd-even mergesort.

spectral Spectral separator code. Includes a graphics demo using everyone's favorite airfoil.

string-search Fast string search algorithm.

The class notes [3] contain other examples.

A.2 Sample session

This transcript of a Nesl session shows many of the language features.

A.2.1 Scalar operations

```
<Nesl> 2 * (3 + 4);

Compiling..Writing..Loading..Running..
Exiting..Reading..
it = 14 : int
<Nesl> set verbose off;     % turns off verbose compiler messages %

<Nesl> (2.2 + 1.1) / 5.0;

it = 0.66 : float
<Nesl> t or f;

it = T : bool
<Nesl> `a < `d;     % that's a backquote, not a quote %
```

```
it = T : bool
<Nesl> 3;

it = 3 : int
<Nesl> 1.6 + 7;     % these aren't the same type %

Error at top level.
For function + in expression
  1.6 + 7
inferred argument types don't match function specification.
  Argument types: float, int
  Function types: a, a :: (a in number)

<Nesl> 1.6 + float(7);

it = 8.6 : float
<Nesl> sin(.6);

it = 0.564642473395035 : float
<Nesl> a = 4;

a = 4 : int
<Nesl> a + 5;

it = 9 : int
<Nesl> if (4 < 5) then 11 else 12;

it = 11 : int
<Nesl> let a = 3 * 4   % the '>' is a prompt for you to enter more %
>      in a + (a * 5);

it = 72 : int
<Nesl> let a = 3 * 4;
>          b = a + 5
>      in a + b;

it = 29 : int
<Nesl> function fact(i) =    % you can define functions at top level %
>        if (i == 1)
>        then 1
>        else i * fact(i-1);

fact = fn : int -> int
<Nesl> fact(5);

it = 120 : int

<Nesl> function circarea(r) = pi * r * r;     % pi is predefined %

circarea = fn : float -> float
<Nesl> circarea(3.0);

it = 28.2743338823081 : float
```

```
<Nesl> (2, `a);

it = (2, `a) : int, char
<Nesl> function div_rem(a, b) = (a / b, rem(a, b));

div_rem = fn : (int, int) -> (int, int)
<Nesl> div_rem (20, 6);

it = (3, 2) : int, int
```

A.2.2 Vector operations

```
<Nesl>  [2, 5, 1, 3];

it = [2, 5, 1, 3] : [int]
<Nesl>  "this is a vector";

it = "this is a vector" : [char]
<Nesl> [(2, 3.4), (5, 8.9)];    % a vector of tuples %

it = [(2, 3.4), (5, 8.9)] : [(int, float)]
<Nesl>  ["this", "is", "a", "nested", "vector"];

it = ["this", "is", "a", "nested", "vector"] : [[char]]
<Nesl> [2, 3.0, 4];     % vectors must have homogeneous elements %

Error at top level.
For function make_sequence in expression
  [2, 3.0]
inferred argument types don't match function specification.
  Argument types: [int], float
  Function types: [a], a :: (a in any)

<Nesl>  {a + 1: a in [2, 3, 4]};

it = [3, 4, 5] : [int]
<Nesl>  let a = [2, 3, 4] in {a + 1: a};

it = [3, 4, 5] : [int]
<Nesl> {a + b: a in [2, 3, 4]; b in [4, 5, 6]};

it = [6, 8, 10] : [int]
<Nesl> let a = [2, 3, 4]; b = [4, 5, 6] in {a + b: a; b};

it = [6, 8, 10] : [int]
<Nesl> {a == b: a in "this"; b in "that"};

it = [T, T, F, F] : [bool]
<Nesl> {fact(a): a in [1, 2, 3, 4, 5]};

it = [1, 2, 6, 24, 120] : [int]
<Nesl> {div_rem(100, a): a in [5, 6, 7, 8]};
```

```
it = [(20, 0), (16, 4), (14, 2), (12, 4)] : [(int, int)]
<Nesl> sum([2, 3, 4]);

it = 9 : int
<Nesl> dist(5, 10);

it = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5] : [int]
<Nesl> [2:50:3];

it = [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47] : [int]
<Nesl>  "big" ++ " boy";

it = "big boy" : [char]
<Nesl> {x in "wombat" | x <= `m};

it = "mba" : [char]
<Nesl> {sum(a): a in [[2, 3, 4], [1], [7, 8, 9]]};

it = [9, 1, 24] : [int]
<Nesl> bottop("testing");  % split sequence into two parts %

it = ["test", "ing"] : [[char]]
<Nesl> partition("break into words", [5, 5, 6]);

it = ["break", " into", " words"] : [[char]]

<Nesl> function my_sum(a) =
>         if (#a == 1) then a[0]
>         else
>           let res = {my_sum(x): x in bottop(a)}
>           in res[0] + res[1];

my_sum = fn : [a] -> a :: (a in number)
<Nesl> my_sum([7, 2, 6]);

it = 15 : int
```

A.2.3 An example: string searching

The algorithm shown here is explained in [3]. The example illustrates the way in which Nesl functions can be developed "from the inside out", using the interactive system to test each new addition.

```
<Nesl> teststr = "string small strap asop string";

teststr = "string small strap asop string" : [char]
<Nesl> candidates = [0:#teststr-5];

candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23, 24] : [int]
<Nesl> {a == `s: a in teststr -> candidates};

it = [T, F, F, F, F, F, F, T, F, F, F, F, F, T, F, F, F, F, F,
```

```
 F, T, F, F, F, T] : [bool]
<Nesl> candidates = {c in candidates;
>                     a in teststr -> candidates | a == `s};

candidates = [0, 7, 13, 20, 24] : [int]
<Nesl> candidates = {c in candidates;
>                    a in teststr -> {candidates+1:candidates}
                     | a == `t};

candidates = [0, 13, 24] : [int]
<Nesl> candidates = {c in candidates;
>                    a in teststr -> {candidates+2:candidates}
>                    | a == `r};

candidates = [0, 13, 24] : [int]
<Nesl> candidates = {c in candidates;
>                    a in teststr -> {candidates+3:candidates}
>                    | a == `i};

candidates = [0, 24] : [int]
<Nesl> candidates = {c in candidates;
>                    a in teststr -> {candidates+4:candidates}
>                    | a == `n};

candidates = [0, 24] : [int]
<Nesl> function next_cands(cands, w, s, i) =
>        if (i == #w) then cands
>        else
>          let letter = w[i];
>              next_chars = s -> {cands + i: cands};
>              new_cands = {c in cands; l in next_chars | l == letter}
>          in next_cands(new_cands, w, s, i + 1);


next_cands = fn : ([int], [a], [a], int) -> [int] :: (a in ordinal)
<Nesl> > function string_search(w, s) =
>         next_cands([0:#s - (#w - 1)], w, s, 0);

string_search = fn : ([a], [a]) -> [int] :: (a in ordinal)
<Nesl>  longstr =
> "This course will be a hands on class on programming parallel
algorithms.  It will introduce several parallel data structures and a
variety of parallel algorithms, and then look at how they can be
programmed.  The class will stress the clean and concise expression of
parallel algorithms and will present the opportunity to program
non-trivial parallel algorithms and run them on a few different
parallel machines.  The course should be appropriate for graduate
students in all areas and for advanced undergraduates.  The
prerequisite is an algorithms class.  Undergraduates also require
permission of the instructor.";

longstr =
       "This course will be a hands on class on programming parallel
```

```
algorithms.  It will introduce several parallel data structures and a
variety of parallel algorithms, and then look at how they can be
programmed.  The class will stress the clean and concise expression of
parallel algorithms and will present the opportunity to program
non-trivial parallel algorithms and run them on a few different
parallel machines.  The course should be appropriate for graduate
students in all areas and for advanced undergraduates.  The
prerequisite is an algorithms class.  Undergraduates also require
permission of the instructor."
        : [char]
<Nesl> string_search("will", longstr);

it = [12, 77, 219, 291] : [int]
<Nesl> string_search("student", longstr);

it = [461] : [int]
```

Index