NESL: A Nested Data-Parallel Language
(Version 3.1)

Guy E. Blelloch
September 19, 1995
CMU-CS-95-170

(Updated version of CMU-CS-92-103, January 1992, and CMU-CS-93-129, April 1993)

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.
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.

19951207 024
Keywords: Data-parallel, parallel algorithms, supercomputers, nested parallelism, PRAM model, parallel programming languages, collection-oriented languages.
Abstract

This report describes NESL, a strongly-typed, applicative, data-parallel language. NESL is intended to be used as a portable interface for programming a variety of parallel and vector computers, and as a basis for teaching parallel algorithms. Parallelism is supplied through a simple set of data-parallel constructs based on sequences, including a mechanism for applying any function over the elements of a sequence in parallel and a rich set of parallel functions that manipulate sequences.

NESL fully supports nested sequences and nested parallelism—the ability to take a parallel function and apply it over multiple instances in parallel. Nested parallelism is important for implementing algorithms with irregular nested loops (where the inner loop lengths depend on the outer iteration) and for divide-and-conquer algorithms. NESL also provides a performance model for calculating the asymptotic performance of a program on various parallel machine models. This is useful for estimating running times of algorithms on actual machines and, when teaching algorithms, for supplying a close correspondence between the code and the theoretical complexity.

This report defines NESL and describes several examples of algorithms coded in the language. The examples include algorithms for median finding, sorting, string searching, finding prime numbers, and finding a planar convex hull. NESL currently compiles to an intermediate language called VCODE, which runs on vector multiprocessors (the CRAY C90 and J90), distributed memory machines (the IBM SP2, Intel Paragon, and Connection Machine CM-5), and sequential workstations. For many algorithms, the current implementation gives performance close to optimized machine-specific code for these machines.

Note: This report is an updated version of CMU-CS-92-103, which described version 2.4 of the language, and of CMU-CS-93-129, which described version 2.6 of the language. Some other documents that describe NESL are:

- An overview of the implementation with some timing results [8].
- A formal definition of the NESL cost model [23].
# Contents

1 Introduction
   1.1 Parallel Operations on Sequences ........................................ 4
   1.2 Nested Parallelism ............................................................. 8
   1.3 Pairs ............................................................................. 10
   1.4 Types ............................................................................ 12
   1.5 Deriving Complexity ......................................................... 13

2 Examples
   2.1 String Searching ............................................................... 17
   2.2 Primes ........................................................................... 18
   2.3 Planar Convex-Hull ............................................................ 19

3 Language Definition
   3.1 Data ............................................................................. 22
      3.1.1 Atomic Data Types ...................................................... 22
      3.1.2 Sequences ([ ]) ......................................................... 23
      3.1.3 Record Types (datatype) ............................................. 24
   3.2 Functions and Constructs .................................................... 25
      3.2.1 Conditionals (if) ....................................................... 25
      3.2.2 Binding Local Variables (let) ....................................... 25
      3.2.3 The Apply-to-Each Construct ({}) ............................... 26
      3.2.4 Defining New Functions (function) .............................. 28
      3.2.5 Top-Level Bindings (=) .............................................. 28

Bibliography .......................................................................... 28

A The N E S L Grammar ............................................................ 32

B List of Functions
   B.1 Scalar Functions ................................................................ 35
   B.2 Sequence Functions ........................................................... 40
   B.3 Functions on Any Type ....................................................... 47
   B.4 Functions for Manipulating Strings .................................... 47
   B.5 Functions with Side Effects ............................................... 48

C Implementation Notes .......................................................... 57

Index ................................................................................. 60
1 Introduction

This report describes and defines the data-parallel language NESL. The language was designed with the following goals:

1. To support parallelism by means of a set of data-parallel constructs based on sequences. These constructs supply parallelism through (1) the ability to apply any function concurrently over each element of a sequence, and (2) a set of parallel functions that operate on sequences, such as the permute function, which permutes the order of the elements in a sequence.

2. To support complete nested parallelism. NESL fully supports nested sequences, and the ability to apply any user defined function over the elements of a sequence, even if the function is itself parallel and the elements of the sequence are themselves sequences. Nested parallelism is critical for describing both divide-and-conquer algorithms and algorithms with nested data structures [7].

3. To generate efficient code for a variety of architectures, including both SIMD and MIMD machines, with both shared and distributed memory. NESL currently generates a portable intermediate code called VCOD [9], which runs on vector multiprocessors (the CRAY C90 and J90) as well as distributed memory machines (the IBM SP2, Intel Paragon, and Connection Machine CM-5). Various benchmark algorithms achieve very good running times on these machines [16, 8].

4. To be well suited for describing parallel algorithms, and to supply a mechanism for deriving the theoretical running time directly from the code. Each function in NESL has two complexity measures associated with it, the work and depth complexities [7]. A simple equation maps these complexities to the asymptotic running time on a Parallel Random Access Machine (PRAM) Model.

NESL is a strongly-typed strict first-order functional (applicative) language. It runs within an interactive environment and is loosely based on the ML language [34]. The language uses sequences as a primitive parallel data type, and parallelism is achieved exclusively through operations on these sequences [7]. The set of sequence functions supplied by NESL was chosen based both on their usefulness on a broad variety of algorithms, and on their efficiency when implemented on parallel machines. To promote the use of parallelism, NESL supplies no serial looping constructs (although serial looping can be simulated with recursion). NESL has been used for 3 years now for teaching parallel algorithms [10], and many applications and algorithms have been written in the language [22, 4, 5].

NESL is the first data-parallel language whose implementation supports nested parallelism. Nested parallelism is the ability to take a parallel function and apply it over multiple instances in parallel—for example, having a parallel sorting routine, and then using it to sort several sequences concurrently. The data-parallel languages C* [38], *Lisp [31], and Fortran 90 [1] (with array extensions) support no form of nested parallelism. The parallel collections in these languages can only contain scalars or fixed sized records. There is also no means in these languages to apply a user defined function over each element of a collection.
This prohibits the expression of any form of nested parallelism. The languages Connection Machine Lisp [45], and Paralation Lisp [39] both supply nested parallel constructs, but no implementation ever supported the parallel execution of these constructs. Blelloch and Sabot implemented an experimental compiler that supported nested-parallelism for a small subset of Paralation Lisp [13], but it was deemed near impossible to extend it to the full language.

A common complaint about high-level data-parallel languages and, more generally, in the class of languages based on operations over collections [42], such as SETL [40] and APL [29], is that it can be hard or impossible to determine approximate running times by looking at the code. As an example, the \( \beta \) primitive in CM-Lisp (a general communication primitive) is powerful enough that seemingly similar pieces of code could take very different amounts of time depending on details of the implementation of the operation and of the data structures. A similar complaint is often made about the language SETL—a language with sets as a primitive data structure. The time taken by the set operations in SETL is strongly affected by how the set is represented. This representation is chosen by the compiler.

For this reason, NESL was designed so that the built-in functions are quite simple and so that the asymptotic complexity can be derived from the code. To derive the complexity, each function in NESL has two complexity measures associated with it: the work and depth complexities [7]. The work complexity represents the serial work executed by a program—the running time if executed on a serial RAM. The depth complexity represents the deepest path taken by the function—the running time if executed with an unbounded number of processors. Simple composition rules can be used to combine the two complexities across expressions and, based on Brent’s scheduling principle [14], the two complexities place an upper bound on the asymptotic running times for the parallel random access machine (PRAM) [19].

The current compiler translates NESL to VCODE [9], a portable intermediate language. The compiler uses a technique called flattening nested parallelism [13] to translate NESL into the simpler flat data-parallel model supplied by VCODE. VCODE is a small stack-based language with about 100 functions all of which operate on sequences of atomic values (scalars are implemented as sequences of length 1). A VCODE interpreter has been implemented for running VCODE on the Cray C90 and J90, the Connection Machine CM-5, or any machine serial machine with a C compiler [8]. We also have an MPI [20] version of VCODE [25], which will run on machines that support MPI, such as the IBM SP-2, the Intel Paragon, or clusters of workstations. The sequence functions in this interpreter have been highly optimized [7, 17] and, for large sequences, the interpretive overhead becomes relatively small yielding high efficiencies.

The interactive NESL environment runs within Common Lisp and can be used to run VCODE on remote machines. This allows the user to run the environment, including the compiler, on a local workstation while executing interactive calls to NESL programs on the remote parallel machines. As in the Standard ML of New Jersey compiler [2], all interactive invocations are first compiled (in our case into VCODE), and then executed.

---

1In previous descriptions of the language, the term step was used instead of depth.
Control parallel languages that have some feature that are similar to NESL include ID [35, 3], Sisal [32], and Proteus [33]. ID and Sisal are both side-effect free and supply operations on collections of values.

The remainder of this section discusses the use of sequences and nested parallelism in NESL, and how complexity can be derived from NESL code. Section 2 shows several examples of code, and Section 3 along with Appendix A and Appendix B defines the language. Shortcomings of NESL include the limitation to first-order functions (there is no ability to pass functions as arguments). We are currently working on a follow-up on NESL, which will be based on a more rigorous type system, and will include some support for higher-order functions.

1.1 Parallel Operations on Sequences

NESL supports parallelism through operations on sequences, which are specified using square brackets. For example

\[ [2, 1, 9, -3] \]

is a sequence of four integers. In NESL all elements of a sequence must be of the same type, and all sequences must be of finite length. Parallelism on sequences can be achieved in two ways: the ability to apply any function concurrently over each element of a sequence, and a set of built-in parallel functions that operate on sequences. The application of a function over a sequence is achieved using set-like notation similar to set-formers in SETL [40] and list-comprehensions in Miranda [43] and Haskell [28]. For example, the expression

\[ \{ \text{negate}(a) : a \text{ in } [3, -4, -9, 5] \}; \]

\[ \Rightarrow [-3, 4, 9, -5] : [\text{int}] \]

negates each elements of the sequence \([3, -4, -9, 5]\). This construct can be read as “in parallel for each \(a\) in the sequence \([3, -4, -9, 5]\), negate \(a\)”.

The symbol \(\Rightarrow\) points to the result of the expression, and the expression \([\text{int}]\) specifies the type of the result: a sequence of integers. The semantics of the notation differs from that of SETL, Miranda or Haskell in that the operation is defined to be applied in parallel. Henceforth we will refer to the notation as the apply-to-each construct. As with set comprehensions, the apply-to-each construct also provides the ability to subselect elements of a sequence: the expression

\[ \{ \text{negate}(a) : a \text{ in } [3, -4, -9, 5] \mid a < 4 \}; \]

\[ \Rightarrow [-3, 4, 9] : [\text{int}] \]

negates each element of the sequence \([3, -4, -9, 5]\) such that \(a < 4\). The elements that remain maintain their order relative to each other.

It is also possible to iterate over multiple sequences. The expression

\[ \{ a + b : a \text{ in } [3, -4, -9, 5]; b \text{ in } [1, 2, 3, 4] \}; \]

\[ \Rightarrow [4, -2, -6, 9] : [\text{int}] \]
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>* dist(a,1)</td>
<td>Distribute value a to sequence of length 1.</td>
<td>S(result)</td>
</tr>
<tr>
<td>* #a</td>
<td>Return length of sequence a.</td>
<td>1</td>
</tr>
<tr>
<td>a[i]</td>
<td>Return element at position i of a.</td>
<td>S(result)</td>
</tr>
<tr>
<td>rep(d,v,i)</td>
<td>Replace element at position i of d with v.</td>
<td>S(v), S(v) + S(d)</td>
</tr>
<tr>
<td>[s:a]</td>
<td>Return integer sequence from s to e.</td>
<td>(e - s)</td>
</tr>
<tr>
<td>[s:e:d]</td>
<td>Return integer sequence from s to e by d.</td>
<td>(e - s)/d</td>
</tr>
<tr>
<td>sum(a)</td>
<td>Return sum of sequence a.</td>
<td>S(a)</td>
</tr>
<tr>
<td>* @.scan(a)</td>
<td>Return scan based on operator @.</td>
<td>S(a)</td>
</tr>
<tr>
<td>count(a)</td>
<td>Count number of true flags in a.</td>
<td>S(a)</td>
</tr>
<tr>
<td>permute(a,i)</td>
<td>Permute elements of a to positions i.</td>
<td>S(a)</td>
</tr>
<tr>
<td>Place elements a in d.</td>
<td>l(a)</td>
<td>L(a)</td>
</tr>
<tr>
<td>* d &lt;- a</td>
<td>Write elements a in d.</td>
<td>S(a), S(a) + S(d)</td>
</tr>
<tr>
<td>* a -&gt; i</td>
<td>Read from sequence a based on indices i.</td>
<td>S(result)</td>
</tr>
<tr>
<td>max_index(a)</td>
<td>Return index of the maximum value.</td>
<td>S(a)</td>
</tr>
<tr>
<td>min_index(a)</td>
<td>Return index of the minimum value.</td>
<td>S(a)</td>
</tr>
<tr>
<td>a ++ b</td>
<td>Append sequences a and b.</td>
<td>S(a) + S(b)</td>
</tr>
<tr>
<td>drop(a,n)</td>
<td>Drop first n elements of sequence a.</td>
<td>S(result)</td>
</tr>
<tr>
<td>take(a,n)</td>
<td>Take first n elements of sequence a.</td>
<td>S(result)</td>
</tr>
<tr>
<td>rotate(a,n)</td>
<td>Rotate sequence a by n positions.</td>
<td>S(a)</td>
</tr>
<tr>
<td>* flatten(a)</td>
<td>Flatten nested sequence a.</td>
<td>S(a)</td>
</tr>
<tr>
<td>* partition(a,1)</td>
<td>Partition sequence a into nested sequence.</td>
<td>S(a)</td>
</tr>
<tr>
<td>split(a,f)</td>
<td>Split a into nested sequence based on flags f.</td>
<td>S(a)</td>
</tr>
<tr>
<td>bottop(a)</td>
<td>Split a into nested sequence.</td>
<td>S(a)</td>
</tr>
</tbody>
</table>

Table 1: List of some of the sequence functions supplied by NESL. In the work column, S(v) refers to the size of the object v. The * before certain functions means that those functions are primitives. All the other functions can be built out of the primitives with at most a constant factor overhead in both work and depth. For @.scan the @ can be one of {plus, max, min, or, and}. All the sequence functions are described in detail in Appendix B.2. In rep and <-, the work complexity depends on whether the variable used for d is the final reference to that variable (arguments are evaluated left to right). If it is the final reference, then the complexity before the comma is used, otherwise the complexity after the comma is used.
adds the two sequences elementwise. A full description of the apply-to-each construct is
given in Section 3.2.

In NESL, any function, whether primitive or user defined, can be applied to each element
of a sequence. So, for example, we could define a factorial function

\[
\text{function factorial}(i) = \\
\text{if } (i == 1) \text{ then } 1 \\
\text{else } i \times \text{factorial}(i-1); \\
\]

\[
\text{factorial : int -> int}
\]

and then apply it over the elements of a sequence

\[
\{\text{factorial}(x) : x \text{ in } [3,1,7]\}; \\
\Rightarrow [6,1,5040] : [\text{int}]
\]

In this example, the function name(arguments) = body; construct is used to define
factorial. The function is of type \text{int -> int}, indicating a function that maps inte-
gers to integers. The type is inferred by the compiler.

An apply-to-each construct applies a body to each element of a sequence. We will call
each such application an instance. Since there are no side effects in NESL\(^2\), there is no way
to communicate among the instances of an apply-to-each. An implementation can therefore
execute the instances in any order it chooses without changing the result. In particular,
the instances can be implemented in parallel, therefore giving the apply-to-each its parallel
semantics.

In addition to the apply-to-each construct, a second way to take advantage of parallelism
in NESL is through a set of sequence functions. The sequence functions operate on whole
sequences and all have relatively simple parallel implementations. For example the function
\text{sum} sums the elements of a sequence,

\[
\text{sum}([2, 1, -3, 11, 5]); \\
\Rightarrow 16 : \text{int}
\]

Since addition is associative, this can be implemented on a parallel machine in logarithmic
time using a tree. Another common sequence function is the \text{permute} function, which
permutes a sequence based on a second sequence of indices. For example:

\[
\text{permute("nesl"}, [2,1,3,0]); \\
\Rightarrow "lens" : [\text{char}]
\]

In this case, the 4 characters of the string "nesl" (the term \text{string} is used to refer to a
sequence of characters) are permuted to the indices \([2, 1, 3, 0]\) \((n \rightarrow 2, e \rightarrow 1, s \rightarrow 3,
and 1 \rightarrow 0)\). The implementation of the \text{permute} function on a distributed-memory parallel

\(^2\text{This is not strictly true since some of the utility functions, such as reading or writing from a file, have
side effects. These functions, however, cannot be used within an apply-to-each construct.}\)
function kth_smallest(s, k) =
  let pivot = s[#s/2];
  lesser = {e in s| e < pivot}
  in if (k < #lesser) then kth_smallest(lesser, k)
  else
    let greater = {e in s| e > pivot}
    in if (k >= #s - #greater) then
      kth_smallest(greater, k - (#s - #greater))
    else pivot;

Figure 1: An implementation of order statistics. The function kth_smallest returns the kth smallest element from the input sequence s.

...
1.2 Nested Parallelism

In NESL the elements of a sequence can be any valid data item, including sequences. This rule permits the nesting of sequences to an arbitrary depth. A nested sequence can be written as


This sequence has type: [[[int]]] (a sequence of sequences of integers). Given nested sequences and the rule that any function can be applied in parallel over the elements of a sequence, NESL necessarily supplies the ability to apply a parallel function multiple times in parallel; we call this ability *nested parallelism*. For example, we could apply the parallel sequence function *sum* over a nested sequence:

```
{sum(v) : v in [[2, 1], [7, 3, 0], [4]]};
```

\[=^\text{[3, 10, 4]} : \text{[int]}\]

In this expression there is parallelism both within each *sum*, since the sequence function has a parallel implementation, and across the three instances of *sum*, since the apply-to-each construct is defined such that all instances can run in parallel.

NESL supplies a handful of functions for moving between levels of nesting. These include *flatten*, which takes a nested sequence and flattens it by one level. For example,

```
flatten([[2, 1], [7, 3, 0], [4]]);
```

\[=^\text{[2, 1, 7, 3, 0, 4]} : \text{[int]}\]

Another useful function is *bottop* (for bottom and top), which takes a sequence of values and creates a nested sequence of length 2 with all the elements from the bottom half of the input sequence in the first element and elements from the top half in the second element (if the length of the sequence is odd, the bottom part gets the extra element). For example,

```
bottop("nested parallelism");
```

\[=^\text{["nested pa", "ralellism"]} : \text{[[char]]}\]

Table 2 lists several examples of routines that could take advantage of nested parallelism. Nested parallelism also appears in most divide-and-conquer algorithms. A divide-and-conquer algorithm breaks the original data into smaller parts, applies the same algorithm on the subparts, and then merges the results. If the subproblems can be executed in parallel, as is often the case, the application of the subparts involves nested parallelism. Table 3 lists several examples.

As an example, consider how the function *sum* might be implemented,

```
function my_sum(a) =
    if (#a == 1) then a[0]
    else
        let r = {my_sum(v) : v in bottop(a)};
        in r[0] + r[1];
```

Table 2: Routines with nested parallelism. Both the inner part and the outer part can be executed in parallel.

<table>
<thead>
<tr>
<th>Application</th>
<th>Outer Parallelism</th>
<th>Inner Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of Neighbors in Graph</td>
<td>For each vertex of graph</td>
<td>Sum neighbors of vertex</td>
</tr>
<tr>
<td>Figure Drawing</td>
<td>For each line of image</td>
<td>Draw pixels of line</td>
</tr>
<tr>
<td>Compiling</td>
<td>For each procedure of program</td>
<td>Compile code of procedure</td>
</tr>
<tr>
<td>Text Formatting</td>
<td>For each paragraph of document</td>
<td>Justify lines of paragraph</td>
</tr>
</tbody>
</table>

Table 3: Some divide and conquer algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Outer Parallelism</th>
<th>Inner Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>For lesser and greater elements</td>
<td>Quicksort</td>
</tr>
<tr>
<td>Mergesort</td>
<td>For first and second half</td>
<td>Mergesort</td>
</tr>
<tr>
<td>Closest Pair</td>
<td>For each half of space</td>
<td>Closest Pair</td>
</tr>
<tr>
<td>Strassen’s Matrix Multiply</td>
<td>For each of the 7 sub multiplications</td>
<td>Strassen’s</td>
</tr>
<tr>
<td>Fast</td>
<td>For two sets of interleaved points</td>
<td>Fast</td>
</tr>
<tr>
<td>Fourier Transform</td>
<td></td>
<td>Fourier Transform</td>
</tr>
</tbody>
</table>
function qsort(a) =
if (#a < 2) then a
else
  let pivot = a[#a/2];
  lesser = {e in a| e < pivot};
  equal = {e in a| e == pivot};
  greater = {e in a| e > pivot};
  result = {qsort(v): v in [lesser,greater]}
in result[0] ++ equal ++ result[1];

Figure 2: An implementation of quicksort.

This code tests if the length of the input is one, and returns the single element if it is. If
the length is not one, it uses bottop to split the sequence in two parts, and then applies
itself recursively to each part in parallel. When the parallel calls return, the two results are
extracted and added. The code effectively creates a tree of parallel calls which has depth
\( \lg n \), where \( n \) is the length of \( a \), and executes a total of \( n - 1 \) calls to +.

As another more involved example, consider a parallel variation of quicksort [6] (see
Figure 2). When applied to a sequence \( a \), this version splits the values into three subsets (the
elements lesser, equal and greater than the pivot) and calls itself recursively on the lesser
and greater subsets. To execute the two recursive calls, the lesser and greater sequences
are concatenated into a nested sequence and qsort is applied over the two elements of the
nested sequences in parallel. The final line extracts the two results of the recursive calls
and appends them together with the equal elements in the correct order.

The recursive invocation of qsort generates a tree of calls that looks something like the
tree shown in Figure 3. In this diagram, taking advantage of parallelism within each block
as well as across the blocks is critical to getting a fast parallel algorithm. If we were only
to take advantage of the parallelism within each quicksort to subselect the two sets (the
parallelism within each block), we would do well near the root and badly near the leaves
(there are \( n \) leaves which would be processed serially). Conversely, if we were only to take
advantage of the parallelism available by running the invocations of quicksort in parallel
(the parallelism between blocks but not within a block), we would do well at the leaves
and badly at the root (it would take \( n \) time to process the root). In both cases the parallel
time complexity is \( O(n) \) rather than the ideal \( O(\lg^2 n) \) we can get using both forms (this
is discussed in Section 1.5).

1.3 Pairs

As well as sequences, NESL supports the notion of pairs. A pair is a structure with two
elements, each of which can be of any type. Pairs are often used to build simple structures
or to return multiple values from a function. The binary comma operator is used to create

---

\(^3\) To simulate the built-in sum, it would be necessary to add code to return the identity (0) for empty
sequences.
Figure 3: The quicksort algorithm. Just using parallelism within each block yields a parallel running time at least as great as the number of blocks (O(n)). Just using parallelism from running the blocks in parallel yields a parallel running time at least as great as the largest block (O(n)). By using both forms of parallelism the parallel running time can be reduced to the depth of the tree (expected O(lgn)).

pairs. For example:

9.8,"foo";
⇒ (9.8,"foo") : (float, [char])

2,3;
⇒ (2,3) : (int, int)

The comma operator is right associative (e.g. (2,3,4,5) is equivalent to (2,(3,(4,5))))).

All other binary operators in NESL are left associative. The precedence of the comma operator is lower than any other binary operator, so it is usually necessary to put pairs within parentheses.

Pattern matching inside of a let construct can be used to deconstruct structures of pairs. For example:

let (a,b,c) = (2*4,5-2,4)
in a+b*c;
⇒ 20 : int

In this example, a is bound to 8, b is bound to 3, and c is bound to 4.

Nested pairs differ from sequences in several important ways. Most importantly, there is no way to operate over the elements of a nested pair in parallel. A second important difference is that the elements of a pair need not be of the same type, while elements of a sequence must always be of the same type.
1.4 Types

**NESL** is a strongly typed polymorphic language with a type inference system. Its type system is similar to functional languages such as ML, but since it is first-order (functions cannot be passed as data), function types only appear at the top level. As well as parametric polymorphism, NESL also allows a form of overloading similar to what is supplied by the Haskell Language [28].

The type of a polymorphic function in NESL is specified by using *type-variables*, which are declared in a *type-context*. For example, the type of the `permute` function is:

\[((A), [int]) \rightarrow [A] :: A in any\]

This specifies that for A bound to any type, permute maps a sequence of type [A] and a sequence of type [int] into another sequence of type [A]. The variable A is a type-variable, and the specification *A in any* is the context. A context can have multiple type bindings separated by semicolons. For example, the `zip` function, which zips two sequences of equal length together into one sequence of pairs, has type:

\[(([A], [B]) \rightarrow [(A,B)] :: A in any; B in any\]

User-defined functions can also be polymorphic. For example, we could define

\[
\text{function append3}(s1,s2,s3) = s1 ++ s2 ++ s3;
\]

\[
\Rightarrow \text{append3}(s1,s2,s3) : ([A], [A], [A]) \rightarrow [A] :: A in any
\]

The type inference system will always determine the most general type possible.

In addition to parametric polymorphism, NESL supports a form of overloading by including the notion of *type-classes*. A type-class is a set of types along with an associated set of functions. The functions of a class can only be applied to the types from that class. For example, the base types, int and float are both members of the type class number, and numerical functions such as + and * are defined to work on all numbers. The type of a function overloaded in this way, is specified by limiting the context of a type-variable to a specific type-class. For example, the type of + is:

\[(A, A) \rightarrow A :: A in \text{number}\]

The context "A in number" specifies that A can be bound to any member of the type-class number. The fully polymorphic specification *any* can be thought of as type-class that contains all data types as members. The type-classes are organized into the hierarchy as shown in Figure 4. Functions such as = and < are defined on ordinals, functions such as + and * are defined on numbers, and functions such as or and not are defined on logicals.

User-defined functions can also be overloaded. For example:

\[
\text{function double}(a) = a + a;
\]

\[
\Rightarrow \text{double}(a) : A \rightarrow A :: A in \text{number}
\]

It is also possible to restrict the type of a user-defined function by explicitly typing it. For example,
The diagram in Figure 4 illustrates the type-class hierarchy of NESL. The lower case names are the type classes.

\[
\begin{array}{c}
\text{any} \\
\text{ordinal} \\
\text{number} \\
\text{logical} \\
\text{CHAR} \\
\text{FLOAT} \\
\text{INT} \\
\text{BOOL}
\end{array}
\]

\[
\text{ALL OTHER DATA TYPES}
\]

Figure 4: The type-class hierarchy of NESL. The lower case names are the type classes.

\[
\begin{align*}
\text{function double(a)} &: \text{int -> int = a + a; } \\
\Rightarrow \text{double(a)} &: \text{int -> int}
\end{align*}
\]

limits the type of `double` to \text{int -> int}. The `:` specifies that the next form is a type-specifier (see Appendix A for the full syntax of the function construct and type specifiers).

In certain situations the type inference system cannot determine the type even though there is one. For example the function:

\[
\begin{align*}
\text{function badfunc(a,b)} &= \text{a or (a + b);} \\
\Rightarrow \text{badfunc(a,b)} &: \text{int -> int}
\end{align*}
\]

will not type properly because `or` is defined on the type-class \text{logical} and `+` is defined on the type-class \text{number}. As it so happens, \text{int} is both a logical and an integer, but the NESL inference system does not know how to take intersections of type-classes. In this situation it is necessary to specify the type:

\[
\begin{align*}
\text{function goodfunc(a,b)} &: \text{int, int -> int = a or (a + b); } \\
\Rightarrow \text{goodfunc(a,b)} &: \text{int, int -> int}
\end{align*}
\]

This situation comes up quite rarely.

Specifying the type using ":" serves as good documentation for a function even when the inference system can determine the type. The notion of type-classes in NESL is similar to the type-classes used in the Haskell language [28], but, unlike Haskell, NESL currently does not permit the user to add new type classes.\(^4\)

### 1.5 Deriving Complexity

There are two complexities associated with all computations in NESL.

1. **Work complexity**: this represents the total work done by the computation, that is to say, the amount of time that the computation would take if executed on a serial random access machine. The work complexity for most of the sequence functions is simply the size of one of its arguments. A complete list is given in Table 1. The size of an object is defined recursively: the size of a scalar value is 1, and the size of a sequence is the sum of the sizes of its elements plus 1.

\(^4\)It is likely that future versions of NESL will allow such extensions.
2. **Depth complexity**: this represents the parallel depth of the computation, that is to say, the amount of time the computation would take on a machine with an unbounded number of processors. The depth complexity of all the sequence functions supplied by **Nesl** is constant.

The work and depth complexities are based on the vector random access machine (VRAM) model [7], a strictly data-parallel abstraction of the parallel random access machine (PRAM) model [19]. Since the complexities are meant for determining asymptotic complexity, these complexities do not include constant factors. All the **Nesl** functions, however, can be executed in a small number of machine instructions per element.

The complexities are combined using simple combining rules. Expressions are combined in the standard way—for both the work complexity and the depth complexity, the complexity of an expression is the sum of the complexities of the arguments plus the complexity of the call itself. For example, the complexities of the computation:

\[
\text{sum}(\text{dist}(7,n)) \times \#a
\]

can be calculated as:

<table>
<thead>
<tr>
<th></th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td># (length)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>O(n)</strong></td>
<td><strong>O(1)</strong></td>
</tr>
</tbody>
</table>

The apply-to-each construct is combined in the following way. The work complexity is the sum of the work complexity of the instantiations, and the depth complexity is the maximum over the depth complexities of the instantiations. If we denote the work required by an expression \(\text{exp}\) applied to some data \(a\) as \(W(\text{exp}(a))\), and the depth required as \(D(\text{exp}(a))\), these combining rules can be written as

\[
W(\{\text{el}(a) : a \in e_2(b)\}) = W(e_2(b)) + \sum\{W(\text{el}(a)) : a \in e_2(b)\} \quad (1)
\]

\[
D(\{\text{el}(a) : a \in e_2(b)\}) = D(e_2(b)) + \max\{D(\text{el}(a)) : a \in e_2(b)\} \quad (2)
\]

where \(\text{sum}\) and \(\text{max}\_\text{val}\) just take the sum and maximum of a sequence, respectively.\(^5\)

As an example, the complexities of the computation:

\[
\{[0:i] : i \in [0:n]\}
\]

can be calculated as:

\(^5\)For comments about how these equations relate to the current implementation see Appendix C.
Work Depth

\[
\begin{array}{ll}
[0:n] & n \\
Parallel Calls & 1 \\
[0:1] & \sum_{i=1}^{\min\{i,n\}} i \\
\text{Total} & O(n^2) \quad O(1)
\end{array}
\]

Once the work ($W$) and depth ($D$) complexities have been calculated in this way, the formula

\[ T = O(W/P + D \log P) \]

places an upper bound on the asymptotic running time of an algorithm on the CRCW PRAM model ($P$ is the number of processors). This formula can be derived from Brent’s scheduling principle [14] as shown in [41, 7, 30]. The $\log P$ term shows up because of the cost of allocating tasks to processors, and the cost of implementing the sum and scan operations. On the scan-PRAM [6], where it is assumed that the scan operations are no more expensive than references to the shared-memory (they both require $O(\log P)$ on a machine with bounded degree circuits), then the equation is:

\[ T = O(W/P + \log n) \]  

In the mapping onto a PRAM, the only reason a concurrent-write capability is required is for implementing the $<-$ (write) function, and the only reason a concurrent-read capability is required is for implementing the $\rightarrow$ (read) function. Both of these functions allow repeated indices (“collisions”) and could therefore require concurrent access to a memory location. If an algorithm does not use these functions, or guarantees that there are no collisions when they are used, then the mapping can be implemented with a EREW PRAM. Out of the algorithms in this paper, the primes algorithm (Section 2.2) requires concurrent writes, and the string-searching algorithm (Section 2.1) requires concurrent reads. All the other algorithms can use an EREW PRAM.

As an example of how the work and depth complexities can be used, consider the $k$th smallest algorithm described earlier (Figure 1). In this algorithm the work is the same as the time required by the standard serial version (loops have been replaced by parallel calls), which has an expected time of $O(n)$ [18]. It is also not hard to show that the expected number of recursive calls is $O(\log n)$, since we expect to drop some fraction of the elements on each recursive call [37]. Since each recursive call requires a constant depth, we therefore have:

\[ W(n) = O(n) \quad D(n) = O(\log n) \]

Using Equation 3 this gives us an expected case running time on a PRAM of:

\[ T(n) = O(n/p + \log n \log p) = O(n/p + \log^2 n) \quad \text{EREW PRAM} \]

\[ T(n) = O(n/p + \log n) \quad \text{scan-PRAM} \]

One can similarly show for the quicksort algorithm given in Figure 2 that the work and depth complexities are $W(n) = O(n \log n)$ and $D(n) = O(\log n)$ [37], which give a EREW
PRAM running time of:

\[ T(n) = O(n \log n/p + \log^2 n) \quad \text{EREW PRAM} \]
\[ = O(n \log n/p + \log n) \quad \text{scan-PRAM} \]

In the remainder of this paper we will only derive the work and depth complexities. The reader can plug these into Equation 3 or Equation 4 to get the PRAM running times.

2 Examples

This section describes several examples of NESL programs. Before describing the examples we describe three common operations. The \(-\rightarrow\) binary operator (called read) is used to read multiple elements from a sequence. Its left argument is the sequence to read from, and the right argument is a sequence of integer indices which specify from which locations to read elements. For example, the expression

"an example"\(-\rightarrow\)[7, 0, 8, 4];

\Rightarrow "pale" : [char]

reads the p, a, l and e from locations 7, 0, 8 and 4, respectively. The read function can also be expressed as \text{read}(a,i) instead of \(a \rightarrow i\).

The \(-\leftarrow\) binary operator (called write) is used to write multiple elements into a sequence. Its left argument is the sequence to write into (the destination sequence) and its right argument is a sequence of integer-value pairs. For each element \((i,v)\) in the sequence of pairs, the value \(v\) is written at position \(i\) of the destination sequence. For example, the expression

"an example"\(-\leftarrow\)((4,'s),(2,'d),(3,space)];

\Rightarrow "and sample" : [char]

writes the s, d and space into the string "an example" at locations 4, 2 and 3, respectively (space is a constant that is bound to the space character). The write function can also be expressed as \text{write}(d,iv) instead of \(d \leftarrow iv\).

Ranges of integers can be created using square brackets along with a colon. The notation \([start:end]\) creates a sequence of integers starting at \(start\) and ending one before \(end\). For example:

\[ [10:16]; \]
\Rightarrow [10, 11, 12, 13, 14, 15] : [int]

An additional stride can be specified, as in \([start:end:stride]\), which returns every \(stride^{th}\) integer between \(start\) and \(end\). For example:

\[ [10:25:3]; \]
\Rightarrow [10, 13, 16, 19, 22] : [int]

The integer \(end\) is never included in the sequence.

Using these operations, it is easy to define many of the other NESL functions. Figure 5 shows several examples.
function subseq(a,start,end) = a->[start:end];
function take(a,n) = a->[0:n];
function drop(a,n) = a->[n:#a];
function rotate(a,n) = a->{mod(i-n,#a) : i in [n:n + #a]};
function even_elts(a) = a->[0:#a:2];
function odd_elts(a) = a->[1:#a:2];
function bottop(a) =  [a->[0:#a/2],a->[#a/2:#a]];

Figure 5: Possible implementation for several of the NESL functions on sequences.

function next_character(candidates,w,s,i) =
  if (i == #w) then candidates
  else
    let letter = w[i];
    nextJL = s->{c + i: c in candidates};
    candidates = {c in candidates; n in nextJL | n == letter}
    in next_character(candidates, w, s, i+1);

function string_search(w,s) = next_character([0:#s - #w],w,s,0);

Figure 6: Finding all occurrences of the word w in the string s.

2.1 String Searching

The first example is a function that finds all occurrences of a word in a string (a sequence of characters). The function string_search(w,s) (see Figure 6) takes a search word w and a string s, and returns the starting position of all substrings in s that match w. For example,

    string_search("foo","fobarfoofoofbofoo");

⇒ [5,12] : [int]

The algorithm starts by considering all positions between 0 and #s-#w as candidates for a match (no candidate could be greater than this since it would have to match past the end of the string). The candidates are stored as pointers (indices) into s of the beginning of each match. The algorithm then progresses through the search string, using recursive calls to next_char, narrowing the set of candidate matches on each step.

Based on the current candidates, next_char narrows the set of candidates by only keeping the candidates that match on the next character of w. To do this, each candidate
function primes(n) =
if n == 2 then [2]
else
  let sqr_primes = primes(ceil(sqrt(float(n))));
  sieves = {[2*p:n:p]: p in sqr_primes};
  flat_sieves = flatten(sieves);
  flags = dist(t,n) <- {(i,f): i in flat_sieves}
in drop({i in [0:n]; flag in flags| flag}, 2) ;

Figure 7: Finding all the primes less than n.

checks whether the i*^th^ character in w matches the i*^th^ position past the candidate index. All candidates that do match are packed and passed into the recursive call of next_char. The recursion completes when the algorithm reaches the end of w. The progression of candidates in the "foo" example would be:

<table>
<thead>
<tr>
<th>i</th>
<th>candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, 5, 8, 12]</td>
</tr>
<tr>
<td>1</td>
<td>[0, 5, 12]</td>
</tr>
<tr>
<td>2</td>
<td>[5, 12]</td>
</tr>
</tbody>
</table>

Lets consider the complexity of the algorithm. We assume #w = m and #s = n. The depth complexity of the algorithm is some constant times the number of recursive calls, which is simply $O(m)$. The work complexity of the algorithm is the sum over the recursive calls, of the number of candidates in each recursive call. In practice, this is usually $O(n)$, but in the worst case this can be the product of the two lengths $O(nm)$ (the worst case can only happen if most of the characters in w are repeated). There are parallel string-searching algorithms that give better bounds on the parallel time (depth complexity), and that bound the worst case work complexity to be linear in the length of the search string [15, 44], but these algorithms are somewhat more complicated.

2.2 Primes

Our second example finds all the primes less than n. The algorithm is based on the sieve of Eratosthenes. The basic idea of the sieve is to find all the primes less than $\sqrt{n}$, and then use multiples of these primes to "sieve out" all the composite numbers less than n. Since all composite numbers less than n must have a divisor less than $\sqrt{n}$, the only elements left unsieved will be the primes. There are many parallel versions of the prime sieve, and several naive versions require a total of $O(n^{3/2})$ work and either $O(n^{1/2})$ or $O(n)$ parallel time. A well designed version should require no more work than the serial sieve ($O(n \log \log n)$, and polylogarithmic parallel time.

The version we use (see Figure 7) requires $O(n \log \log n)$ work and $O(\log \log n)$ depth. It works by first recursively finding all the primes up to $\sqrt{n}$, (sqr_primes). Then, for each prime p in sqr_primes, the algorithm generates all the multiples of p up to n (sieves). This
is done with the [s:e:d] construct. The sequence sieves is therefore a nested sequence with each subsequence being the sieve for one of the primes in sqr_primes. The function flatten, is now used to flatten this nested sequence by one level, therefore returning a sequence containing all the sieves. For example,

```
flatten([[4, 6, 8, 10, 12, 14, 16, 18], [6, 9, 12, 15, 18]]);
```

This sequence of sieves is used by the <- function to place a false flag in all positions that are a multiple of one of the sqr_primes. This will return a boolean sequence, flags, which contains a t in all places that were not knocked out by a sieve—these are the primes. However, we want primes to return the indices of the primes instead of flags. To generate these indices the algorithm creates a sequences of all indices between 0 and n ([:n]) and uses subselection to remove the nonprimes. The function drop is then used to remove the first two elements (0 and 1), which are not considered primes but do not get explicitly sieved.

The functions [s:e:d], flatten, dist, <- and drop all require a constant depth. Since primes is called recursively on a problem of size √n the total depth required by the algorithm can be written as the recurrence:

\[
D(n) = \begin{cases} 
  O(1) & n = 1 \\
  D(\sqrt{n}) + O(1) & n > 1 
\end{cases}
= O(\log \log n)
\]

Almost all the work done by primes is done in the first call. In this first call, the work is proportional to the length of the sequence flat_sieves. Using the standard formula

\[
\sum_{p \leq x} \frac{1}{p} = \log \log x + C + O(1/\log x)
\]

where p are the primes [24], the length of this sequence is:

\[
\sum_{p \leq \sqrt{n}} \frac{n}{p} = O(n \log \log \sqrt{n})
\]

\[
= O(n \log \log n)
\]

therefore giving a work complexity of O(n log log n).

### 2.3 Planar Convex-Hull

Our next example solves the planar convex hull problem: given n points in the plane, find which of these points lie on the perimeter of the smallest convex region that contains all points. The planar convex hull problem has many applications ranging from computer graphics [21] to statistics [27]. The algorithm we use to solve the problem is a parallel version [12] of the quickhull algorithm [36]. The quickhull algorithm was given its name because of its similarity to the quicksort algorithm. As with quicksort, the algorithm picks
Figure 8: An example of the quickhull algorithm. Each sequence shows one step of the algorithm. Since A and P are the two x extrema, the line AP is the original split line. J and N are the farthest points in each subspace from AP and are, therefore, used for the next level of splits. The values outside the brackets are hull points that have already been found.

A “pivot” element, splits the data based on the pivot, and is recursively applied to each of the split sets. Also, as with quicksort, the pivot element is not guaranteed to split the data into equally sized sets, and in the worst case the algorithm will require $O(n^2)$ work.

Figure 8 shows an example of the quickhull algorithm, and Figure 9 shows the code. The algorithm is based on the recursive routine hsplit. This function takes a set of points in the plane ($x, y$) coordinates) and two points $p_1$ and $p_2$ that are known to lie on the convex hull, and returns all the points that lie on the hull clockwise from $p_1$ to $p_2$, inclusive of $p_1$, but not of $p_2$. In Figure 8, given all the points [A, B, C, ..., P], $p_1 = A$ and $p_2 = P$, hsplit would return the sequence [A, B, J, O]. In hsplit, the order of $p_1$ and $p_2$ matters, since if we switch A and P, hsplit would return the hull along the other direction [P, N, C].

The hsplit function works by first removing all the elements that cannot be on the hull since they lie below the line between $p_1$ and $p_2$. This is done by removing elements whose cross product with the line between $p_1$ and $p_2$ are negative. In the case $p_1 = A$ and $p_2 = P$, the points [B, D, F, G, H, J, K, M, O] would remain and be placed in the sequence packed. The algorithm now finds the point furthest from the line $p_1$-$p_2$. This point $p_m$ must be on the hull since as a line at infinity parallel to $p_1$-$p_2$ moves toward $p_1$-$p_2$, it must first hit $p_m$. The point $p_m$ (J in the running example) is found by taking the point with the maximum cross-product. Once $p_m$ is found, hsplit calls itself twice recursively using the points ($p_1$, $p_m$) and ($p_m$, $p_2$) ((A, J) and (J, P) in the example). When the recursive calls return, hsplit flattens the result (this effectively appends the two subhulls).
function cross_product(o, line) =
  let (xo, yo) = o;
  ((x1, y1), (x2, y2)) = line
  in (x1-xo)*(y2-yo) - (y1-yo)*(x2-xo);

function hsplit(points, p1, p2) =
  let cross = {cross_product(p, (p1, p2)): p in points};
  packed = {p in points; c in cross | plusp(c)}
  in if (#packed < 2) then [p1] ++ packed
      else
          let pm = points[max_index(cross)]
          in flatten({hsplit(packed, p1, p2): pi in [p1, pm]; p2 in [pm, p2]});

function convex_hull(points) =
  let x = {x : (x, y) in points};
  minx = points[min_index(x)];
  maxx = points[max_index(x)]
  in hsplit(points, minx, maxx) ++ hsplit(points, maxx, minx);

Figure 9: Code for Quickhull. Each point is represented as a pair. Pattern matching is used to
effectively extract the x and y coordinates of each pair.

The overall convex-hull algorithm works by finding the points with minimum and
maximum x coordinates (these points must be on the hull) and then using hsplit to find
the upper and lower hull. Each recursive call has a depth complexity of O(1) and a work
complexity of O(n). However, since many points might be deleted on each step, the work
complexity could be significantly less. For m hull points, the algorithm runs in O(lgm)
depth for well-distributed hull points, and has a worst case depth of O(m).

3 Language Definition

This section defines NESL. It is not meant as a formal semantics but, along with the
full definition of the syntax in Appendix A and description of all the built-in functions in
Appendix B, it should serve as an adequate description of the language. NESL is a strict
first-order strongly-typed language with the following data types:

- four primitive atomic data types: booleans (bool), integers (int), characters (char),
  and floats (float);
- the primitive sequence type;
- the primitive pair type;
- and user definable compound datatypes;

21
and the following operations:

- a set of predefined functions on the primitive types;
- three primitive constructs: a conditional construct if, a binding construct let, and the apply-to-each construct;
- and a function constructor, function, for defining new functions.

This section covers each of these topics.

3.1 Data

3.1.1 Atomic Data Types

There are four primitive atomic data types: *booleans, integers, characters* and *floats*.

The boolean type `bool` can have one of two values `t` or `f`. The standard logical operations (eg. `not`, `and`, `or`, `xor`, `nor`, `nand`) are predefined. The operations `and`, `or`, `xor`, `nor`, `nand` all use infix notation. For example:

```
not(not(t));
⇒ t : bool
```

```
t xor f;
⇒ t : bool
```

The integer type `int` is the set of (positive and negative) integers that can be represented in the fixed precision of a machine-sized word. The exact precision is machine dependent, but will always be at least 32-bits. The standard functions on integers (`+`, `-`, `*`, `/`, `==`, `>`, `<`, `negate`, ...) are predefined, and use infix notation (see Appendix A for the precedence rules). For example:

```
3 * -11;
⇒ -33 : int
```

```
7 == 8;
⇒ f : bool
```

Overflow will return unpredictable results.

The character type `char` is the set of ASCII characters. The characters have a fixed order and all the comparison operations (eg. `==`, `<`, `>`, `>`...) can be used. Characters are written by placing a `'` in front of the character. For example:

```
'8;
⇒ '8 : char
```
The global variables `space`, `newline` and `tab` are bound to the appropriate characters.

The type `float` is used to specify floating-point numbers. The exact representation of these numbers is machine specific, but NESL tries to use 64-bit IEEE when possible. Floats support most of the same functions as integers, and also have several additional functions (e.g. `round`, `truncate`, `sqrt`, `log`, ...). Floats must be written by placing a decimal point in them so that they can be distinguished from integers.

```plaintext
1.2 * 3.0;
⇒ 3.6 : float
    round(2.1);
⇒ 2 : int
```

There is no implicit coercion between scalar types. To add 2 and 3.0, for example, it is necessary to coerce one of them: e.g.

```plaintext
float(2) + 3.0;
⇒ 5.0 : float
```

A complete list of the functions available on scalar types can be found in Appendix B.1.

### 3.1.2 Sequences ([])

A sequence can contain any type, including other sequences, but each element in a sequence must be of the same type (sequences are homogeneous). The type of a sequence whose elements are of type `α`, is specified as `[α]`. For examples:

```plaintext
[6, 2, 4, 5];
⇒ [6, 2, 4, 5] : [int]
```

```plaintext
[[2, 1, 7, 3], [6, 2], [22, 9]];
⇒ [[2, 1, 7, 3], [6, 2], [22, 9]] : [[int]]
```

Sequences of characters can be written between double quotes,

```plaintext
"a string";
⇒ "a string" : [char]
```

but can also be written as a sequence of characters:
Empty sequences must be explicitly typed since the type cannot be determined from the elements. The type of an empty sequences is specified by using empty square braces followed by the type of the elements. For example,

```
[] int;
⇒ [] : [int]

[] (int, bool);
⇒ [] : [(int, bool)]
```

Appendix B.2 describes the functions that operate on sequences.

3.1.3 Record Types (datatype)

Record types with a fixed number of slots can be defined with the `datatype` construct. For example,

```
datatype complex(float, float);
```

defines a record with two slots both which must contain a floating-point number. Defining a record also defines a corresponding function that is used to construct the record. For example,

```
complex(7.1, 11.9);
⇒ complex(7.1, 11.9) : complex
```

creates a `complex` record with 7.1 and 11.9 as its two values.

Elements of a record can be accessed using pattern matching in the `let` construct. For example,

```
let complex(real, imaginary) = a in real;
```

will remove the real part of the variable `a` (assuming it is kept in the first slot). More details on pattern matching are given in the next section.

As with functions, records can be parameterized based on type-variables. For example, `complex` could have been defined as:

```
datatype complex(alpha, alpha) :: alpha in number;
```

```
⇒ complex(a1, a2) : alpha, alpha → complex(alpha) :: alpha in number
```

This specifies that for `alpha` bound to any type in the type-class `number` (either `int` or `float`), both slots must be of type `alpha`. This will allow either,
complex(7.1, 11.9);
⇒ complex(7.1, 11.9) : complex(float)

complex(7, 11);
⇒ complex(7, 11) : complex(int)

but will not allow complex(7, 'a) or complex(2, 2.2). The type of a record is specified by the record name followed by the binding of all its type-variables. In this case, the binding of the type-variable is either int or float.

3.2 Functions and Constructs

3.2.1 Conditionals (if)

The only primitive conditional in NESL is the if construct. The syntax is:

\[
\text{IF } \text{exp} \text{ THEN } \text{exp} \text{ ELSE } \text{exp}
\]

If the first expression is true, then the second expression is evaluated and its result is returned, otherwise the third expression is evaluated and its result is returned. The first expression must be of type bool, and the other two expressions must be of identical types. For example:

\[
\text{if } (t \text{ and } f) \text{ then } 3 + 4 \text{ else } (6 - 2) \times 7
\]
is a valid expression, but

\[
\text{if } (t \text{ and } f) \text{ then } 3 \text{ else } 2.6
\]
is not, since the two branches return different types.

3.2.2 Binding Local Variables (let)

Local variables can be bound with the let construct. The syntax is:

\[
\text{LET expbinds IN exp}
\]

\[
\text{expbinds ::= expbind [; expbinds]} \quad \text{variable bindings}
\]

\[
\text{expbind ::= pattern = exp} \quad \text{variable binding}
\]

\[
\text{pattern ::= ident} \quad \text{variable}
\]

\[
\text{ident(pattern)} \quad \text{datatype pattern}
\]

\[
\text{pattern, pattern} \quad \text{pair pattern}
\]

\[
( \text{pattern } )
\]

The semicolon separates bindings (the square brackets indicate an optional term of the syntax). Each pattern is either a variable name or a pattern based on a record name. Each expbind binds the variables in the pattern on the left of the = to the result of the expression on the right. For example:
let a = 7;
    (b, c) = (1,2)
in a*(b + c);
⇒ 21 : int

Here a is bound to 7, then the pattern (b, c) is matched with the result of the expression on the right so that b is bound to 1 and c is bound to 2. Patterns can be nested, and the patterns are matched recursively.

The variables in each expbind can be used in the expressions (exp) of any later expbind (the bindings are done serially). For example, in the expression

let a = 7;
    b = a + 4
in a * b;
⇒ 77 : int

the variable a is bound to the value 7 and then the variable b is bound to the value of a plus 4, which is 11. When these are multiplied in the body, the result is 77.

3.2.3 The Apply-to-Each Construct ({})

The apply-to-each construct is used to apply any function over the elements of a sequence. It has the following syntax:

```
{[exp :] rbinds []| exp}
```

```
rbinds ::= rbind []| rbinds
```

```
rbind ::= pattern IN exp    full binding
    ident                  shorthand binding
```

An apply-to-each construct consists of three parts: the expression before the colon, which we will call the body, the bindings that follow the body, and the expression that follows the |, which we will call the sieve. Both the body and the sieve are optional: they could both be left out, as in

```
{a in [1, 2, 3]};
⇒ [1, 2, 3] : [int]
```

The rbinds can contain multiple bindings which are separated by semicolons. We first consider the case in which there is a single binding. A binding can either consist of a pattern followed by the keyword IN and an expression (full binding), or consist of a variable name (shorthand binding). In a full binding the expression is evaluated (it must evaluate to a sequence) and the variables in the pattern are bound in turn to each element of the sequence. The body and sieve are applied for each of these bindings. For example:
{a + 2: a in [1, 2, 3]};
⇒ [3, 4, 5] : [int]

{a + b: (a, b) in [(1, 2), (3, 4), (5, 6)]};
⇒ [3, 7, 11] : [int]

In a shorthand binding, the variable must be a sequence, and the body and sieve are applied to each element of the sequence with the variable name bound to the element. For example:

```plaintext
let a = [1, 2, 3]
in {a + 2: a};
⇒ [3, 4, 5] : [int]
```

In the case of multiple rbinds, each of the sequences (either the result of the expression in a full binding or the value of the variable in a shorthand binding) must be of equal length. The bindings are interleaved so that the body is evaluated with bindings made for elements at the same index of each sequence. For example:

```plaintext
{a + b: a in [1, 2, 3]; b in [1, 4, 9]};
⇒ [2, 6, 12] : [int]

{dist(b, a): a in [1, 2, 3]; b in [1, 4, 9]};
⇒ [[1], [4, 4], [9, 9, 9]] : [[int]]
```

An apply-to-each with a body and two bindings,

```plaintext
{body: pattern1 in exp1; pattern2 in exp2 | sieve}
```

is equivalent to the single binding construct

```plaintext
{body: (pattern1, pattern2) in zip(exp1, exp2) | sieve}
```

where zip, as defined in the list of functions, elementwise zips together the two sequences it is given as arguments.

If there is no body in an apply-to-each construct, then the results of the first binding is returned. For example:

```plaintext
{a in [1, 2, 3]; b in [1, 4, 9]};
⇒ [1, 2, 3] : [int]

{a in [1, 2, 3]; b in [2, 4, 9] | b == 2*a};
⇒ [1, 2] : [int]

{b in [2, 4, 9]; a in [1, 2, 3] | b == 2*a};
⇒ [2, 4] : [int]
```
If there is a body and a sieve, the body and sieve are both evaluated for all bindings, and then the subselection is applied. An apply-to-each with a sieve of the form:

\{body : bindings | sieve\}

is equivalent to the construct

pack\((\{(body,sieve) : bindings\})\)

where pack, as defined in the list of functions, takes a sequence of type \([(\alpha,\text{bool})]\) and returns a sequence which contains the first element of each pair if the second element is true. The order of remaining elements is maintained.

3.2.4 Defining New Functions (function)

Functions can be defined at top-level using the function construct. The syntax is:

\begin{verbatim}
FUNCTION ident pattern [: funtype] = exp;
\end{verbatim}

A function has one argument, but the argument can be any pattern. The body of a function (the \texttt{exp} at the end) can only refer to variables bound in the \texttt{pattern}, or variables declared at top-level. Any function referred to in the body can only refer to functions previously defined or to the function itself (at present there is no way to define mutually recursive functions). As with all functional languages, defining a function with the same name as a previous function only hides the previous function from future use: all references to a function before the new definition will refer to the original definition.

3.2.5 Top-Level Bindings (=)

You can bind a variable at top-level using the = operator. The syntax is:

\begin{verbatim}
ident = exp;
\end{verbatim}

For example, \texttt{a = 211;} will bind the variable \texttt{a} to the value \texttt{211}. The variable can now either be referenced at top level, or can be referenced inside of any function. For example, the definition

\begin{verbatim}
function foo(c) = c + a;
\end{verbatim}

would define a function that adds \texttt{211} to its input. Such top-level binding is mostly useful for saving temporary results at top-level, and for defining constants. The variable \texttt{pi} is bound at top level to the value \texttt{\pi}.

Acknowledgments

I would like to thank Marco Zagha, Jay Sipelstein, Margaret Reid-Miller, Bob Harper, Jonathan Hardwick, John Greiner, Tim Freeman, and Siddhartha Chatterjee for many helpful comments on this manual. Siddhartha Chatterjee, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha did all the work getting the intermediate languages VCODE and CVL running so that NESL can actually run on parallel machines. Dafna Talmor and Yury Smirnov helped with the X plotting code.
References


A The NESL Grammar

This appendix defines the grammar of NESL. The grammatical conventions are:

- The brackets [ ] enclose optional phrases, the symbol * means repeat the previous expression any number of times, and the symbol + means repeat the previous expression any number of times, but at least once.

- All symbols in typewriter font are literal tokens, all symbols in boldface are tokens with the lexical definitions given below, and all symbols in italics are variables (nonterminals) of the grammar.

- All uppercase letters can either be upper or lower case. NESL is case insensitive.

Toplevel

\[
toplevel ::= \text{FUNCTION name pattern} \[\text{: typedef}] = \text{exp} ; \quad \text{function definition} \\
\quad \text{DATATYPE name typedef} ; \quad \text{datatype definition} \\
\quad \text{pattern} = \text{exp} ; \quad \text{variable binding} \\
\quad \text{exp} ; \quad \text{expression}
\]

Types

\[
typedef ::= \text{typeexp} \[\text{: ( typebinds )}] \quad \text{type definition} \\
typebinds ::= \text{typebind} \[; typebinds] \quad \text{binding type variables} \\
typebind ::= \text{name IN typeclass} \quad \text{binding a type variable} \\
typeexp ::= \text{basetype} \\
\quad \text{name} \quad \text{base type} \\
\quad \text{typeexp } -> \text{typeexp} \quad \text{type variable or datatype} \\
\quad \text{typeexp , typeexp} \quad \text{function type} \\
\quad \text{name(typeclist)} \quad \text{pair type} \\
\quad \text{‘[‘ typeexp ‘]’} \quad \text{compound datatype} \\
\quad \text{( typeexp )} \quad \text{sequence type} \\
typelist ::= \text{typeexp} \[; typelist] \quad \text{type list} \\
typeclass ::= \text{NUMBER | ORDINAL | LOGICAL | ANY} \quad \text{the type classes} \\
basetype ::= \text{INT | BOOL | FLOAT | CHAR} \quad \text{the base types}
\]
Expressions

\[ \text{exp} \quad ::= \quad \text{const} \]
\[ \qquad \text{name} \]
\[ \qquad \text{IF exp THEN exp ELSE exp} \]
\[ \qquad \text{LET expbinds IN exp} \{[exp : ] rbinds [\text{[exp]}]} \]
\[ \qquad \text{exp exp} \]
\[ \qquad \text{exp binop exp} \]
\[ \qquad \text{unaryop exp} \]
\[ \qquad \text{sequence} \]
\[ \qquad \text{exp [' exp '] } \]
\[ \qquad (\text{exp}) \]

\[ \text{expbinds} \quad ::= \quad \text{pattern = exp [:; expbinds]} \]
\[ \text{pattern} \quad ::= \quad \text{name} \]
\[ \qquad \text{name(pattern)} \]
\[ \qquad \text{pattern, pattern} \]
\[ \qquad (\text{pattern}) \]

\[ \text{rbinds} \quad ::= \quad \text{rbind [:; rbinds]} \]
\[ \text{rbind} \quad ::= \quad \text{pattern IN exp} \]
\[ \qquad \text{name} \]

\[ \text{sequence} \quad ::= \quad [' \text{explist } '] \]
\[ \qquad [' ' ] \text{typeexp} \]
\[ \qquad [' ' \text{exp : exp [: exp] } '] \]

\[ \text{explist} \quad ::= \quad \text{exp [ , explist]} \]

\[ \text{const} \quad ::= \quad \text{intconst} \]
\[ \qquad \text{floatconst} \]
\[ \qquad \text{boolconst} \]
\[ \qquad \text{stringconst} \]

\[ \text{binop} \quad ::= \quad , \]
\[ \qquad \text{OR} \quad | \quad \text{NOR} \quad | \quad \text{XOR} \]
\[ \qquad \text{AND} \quad | \quad \text{NAND} \quad | \quad \text{=} \quad | \quad /= \quad | \quad < \quad | \quad > \quad | \quad <= \quad | \quad >= \quad | \quad | \quad + \quad | \quad - \quad | \quad ++ \quad | \quad <\quad | \quad \rightarrow \quad | \quad \leftarrow \quad | \quad \rightarrow \quad | \quad \rightarrow \quad | \quad \leftarrow \quad | \quad \rightarrow \]

\[ \text{unaryop} \quad ::= \quad \# \quad | \quad @ \quad | \quad - \]

constant
variable
conditional
local bindings
apply-to-each
function application
binary operator
unary operator
sequence
sequence extraction
parenthesized expression
variable bindings
variable
datatype pattern
pair pattern
iteration binding
shorthand form
listed sequence
empty sequence
integer range
fixed precision integer
fixed precision float
boolean (T or F)
character string
precedence 1
precedence 2
precedence 3
precedence 4
precedence 5
precedence 6
precedence 7
precedence 8
Lexical Definitions

The following defines regular expressions for the lexical classes of tokens. The grammatical conventions are:

- All uppercase letters can either be upper or lower case. NESL is case insensitive.

- The brackets ( ) enclose an expression. The brackets [ ] enclose a character set, any one of which must match. The expression 0-9 within square brackets means all digits and the expression A-Z means all letters. The symbol ^ as the first character within square brackets means a compliment character set (all characters excepting the following ones).

- The symbol * means the previous expression can be repeated as many times as needed, the symbol + means the previous expression can be repeated as many times as needed but at least once, and the symbol ? means the previous expression can be matched either once or not at all.

intconst ::= [-+]?[0-9]+  
floatconst ::= [-+]?[0-9]*.[0-9]+([eE][-+]?[0-9]+)?  
name ::= [_A-Z0-9]+  
boolconst ::= [TF]  
stringconst ::= "["[^"]*"
B List of Functions

This section lists the functions and constants available in NESL. Each function is listed in the following way:

\[
\text{function interface} \quad \{\text{type}\}
\]
Definition of function.

The hierarchy of the type classes is shown in Figure 4.

B.1 Scalar Functions

Logical Functions

All the logical functions work on either integers or booleans. In the case of integers, they work bitwise over the bit representation of the integer.

\[\text{not}(a) \quad \{(a ightarrow a : (a \text{ in logical})}\]
Returns the logical inverse of the argument. For integers, this is the ones complement.

\[\text{a or b} \quad \{(a, a) ightarrow a : (a \text{ in logical})}\]
Returns the inclusive or of the two arguments.

\[\text{a and b} \quad \{(a, a) ightarrow a : (a \text{ in logical})}\]
Returns the logical and of the two arguments.

\[\text{a xor b} \quad \{(a, a) ightarrow a : (a \text{ in logical})}\]
Returns the exclusive or of the two arguments.

\[\text{a nor b} \quad \{(a, a) ightarrow a : (a \text{ in logical})}\]
Returns the inverse of the inclusive or of the two arguments.

\[\text{a nand b} \quad \{(a, a) ightarrow a : (a \text{ in logical})}\]
Returns the inverse of the and of the two arguments.

Comparison Functions

All comparison functions work on integers, floats and characters.

\[\text{a == b} \quad \{(a, a) ightarrow \text{bool} : (a \text{ in ordinal})}\]
Returns t if the two arguments are equal.

\[\text{a /= b} \quad \{(a, a) ightarrow \text{bool} : (a \text{ in ordinal})}\]
Returns t if the two arguments are not equal.
a < b \quad \{(a, a) \to bool :: (a in ordinal)\}
Returns t if the first argument is strictly less than the second argument.

a > b \quad \{(a, a) \to bool :: (a in ordinal)\}
Returns t if the first argument is strictly greater than the second argument.

a \leq b \quad \{(a, a) \to bool :: (a in ordinal)\}
Returns t if the first argument is less than or equal to the second argument.

a \geq b \quad \{(a, a) \to bool :: (a in ordinal)\}
Returns t if the first argument is greater or equal to the second argument.

**Predicates**

plusp(v) \quad \{a \to bool :: (a in number)\}
Returns t if v is strictly greater than 0.

minusp(v) \quad \{a \to bool :: (a in number)\}
Returns t if v is strictly less than 0.

zerop(v) \quad \{a \to bool :: (a in number)\}
Returns t if v is equal to 0.

oddp(v) \quad \{int \to bool\}
Returns t if v is odd (not divisible by two).

evenp(v) \quad \{int \to bool\}
Returns t if v is even (divisible by two).

**Arithmetic Functions**

a + b \quad \{(a, a) \to a :: (a in number)\}
Returns the sum of the two arguments.

a - b \quad \{(a, a) \to a :: (a in number)\}
Subtracts the second argument from the first.

-v \quad \{a \to a :: (a in number)\}
Negates a number.

abs(x) \quad \{a \to a :: (a in number)\}
Returns the absolute value of the argument.
\textbf{diff}(x, y) \quad \{(a, a) \rightarrow a :: (a \text{ in } \text{number})\}

Returns the absolute value of the difference of the two arguments.

\textbf{max}(a, b) \quad \{(a, a) \rightarrow a :: (a \text{ in } \text{ordinal})\}

Returns the argument that is greatest (closest to positive infinity).

\textbf{min}(a, b) \quad \{(a, a) \rightarrow a :: (a \text{ in } \text{ordinal})\}

Returns the argument that is least (closest to negative infinity).

\textbf{v * d} \quad \{(a, a) \rightarrow a :: (a \text{ in } \text{number})\}

Returns the product of the two arguments.

\textbf{v / d} \quad \{(a, a) \rightarrow a :: (a \text{ in } \text{number})\}

Returns \(v\) divided by \(d\). If the arguments are integers, the result is truncated towards 0.

\textbf{rem}(v, d) \quad \{(\text{int, int}) \rightarrow \text{int}\}

Returns the remainder after dividing \(v\) by \(d\). The following examples show \textbf{rem} does for negative arguments: \(\text{rem}(5,3) = 2, \text{rem}(5,-3) = 2, \text{rem}(-5,3) = -2,\) and \(\text{rem}(-5,-3) = -2.\)

\textbf{lshift}(a, b) \quad \{(\text{int, int}) \rightarrow \text{int}\}

Returns the first argument logically shifted to the left by the integer contained in the second argument. Shifting will fill with 0-bits.

\textbf{rshift}(a, b) \quad \{(\text{int, int}) \rightarrow \text{int}\}

Returns the first argument logically shifted to the right by the integer contained in the second argument. Shifting will fill with 0-bits or the sign bit, depending on the implementation.

\textbf{sqrt}(v) \quad \{\text{float} \rightarrow \text{float}\}

Returns the square root of the argument. The argument must be nonnegative.

\textbf{isqrt}(v) \quad \{\text{int} \rightarrow \text{int}\}

Returns the greatest integer less than or equal to the exact square root of the integer argument. The argument must be nonnegative.

\textbf{ln}(v) \quad \{\text{float} \rightarrow \text{float}\}

Returns the natural log of the argument.

\textbf{log}(v, b) \quad \{(\text{float, float}) \rightarrow \text{float}\}

Returns the logarithm of \(v\) in the base \(b\).

\textbf{exp}(v) \quad \{\text{float} \rightarrow \text{float}\}
Returns \(e\) raised to the power \(v\).

\[
e^{v}
\]

\textit{expt}(v, p) \quad ((\text{float}, \text{float}) \rightarrow \text{float})

Returns \(v\) raised to the power \(p\).

\[
v^p
\]

\textit{sin}(v) \quad (\text{float} \rightarrow \text{float})

Returns the sine of \(v\), where \(v\) is in radians.

\[
\sin(v)
\]

\textit{cos}(v) \quad (\text{float} \rightarrow \text{float})

Returns the cosine of \(v\), where \(v\) is in radians.

\[
\cos(v)
\]

\textit{tan}(v) \quad (\text{float} \rightarrow \text{float})

Returns the tangent of \(v\), where \(v\) is in radians.

\[
\tan(v)
\]

\textit{asin}(v) \quad (\text{float} \rightarrow \text{float})

Returns the arc sine of \(v\). The result is in radians.

\[
\arcsin(v)
\]

\textit{acos}(v) \quad (\text{float} \rightarrow \text{float})

Returns the arc cosine of \(v\). The result is in radians.

\[
\arccos(v)
\]

\textit{atan}(v) \quad (\text{float} \rightarrow \text{float})

Returns the arc tangent of \(v\). The result is in radians.

\[
\arctan(v)
\]

\textit{sinh}(v) \quad (\text{float} \rightarrow \text{float})

Returns the hyperbolic sine of \(v\) \(\left(\frac{e^v - e^{-v}}{2}\right)\).

\[
\sinh(v)
\]

\textit{cosh}(v) \quad (\text{float} \rightarrow \text{float})

Returns the hyperbolic cosine of \(v\) \(\left(\frac{e^v + e^{-v}}{2}\right)\).

\[
\cosh(v)
\]

\textit{tanh}(v) \quad (\text{float} \rightarrow \text{float})

Returns the hyperbolic tangent of \(v\) \(\left(\frac{e^v - e^{-v}}{e^v + e^{-v}}\right)\).

\[
\tanh(v)
\]

\textbf{Conversion Functions}

\textit{btoi}(a) \quad (\text{bool} \rightarrow \text{int})

Converts the boolean values \textit{t} and \textit{f} into 1 and 0, respectively.

\textit{code_char}(a) \quad (\text{int} \rightarrow \text{char})

Converts an integer to a character. The integer must be the code for a valid character.

\textit{char_code}(a) \quad (\text{char} \rightarrow \text{int})

Converts a character to its integer code.

38
float(v)  \{int \rightarrow float\}
Converting an integer to a floating-point number.

ceil(v)  \{float \rightarrow int\}
Converting a floating-point number to an integer by truncating toward positive infinity.

floor(v)  \{float \rightarrow int\}
Converting a floating-point number to an integer by truncating toward negative infinity.

trunc(v)  \{float \rightarrow int\}
Converting a floating-point number to an integer by truncating toward zero.

round(v)  \{float \rightarrow int\}
Converting a floating-point number to an integer by rounding to the nearest integer; if the
number is exactly halfway between two integers, then it is implementation specific to which
integer it is rounded.

Constants

pi  \{float\}
The value of pi.

max_int  \{int\}

min_int  \{int\}

Other Scalar Functions

rand(v)  \{a \rightarrow a :: (a in number)\}
For a positive value v, rand returns a random value in the range [0..v). Note that the
random number seed is reset each time the user returns to top level. To get different sets
of random numbers, use rand_seed with different seeds.

rand_seed(v)  \{int \rightarrow bool\}
Seed the random number generator. Note that a given seed is only guaranteed to give
the same sequence of random numbers on a fixed machine and with a fixed number of
processors.
B.2 Sequence Functions

Simple Sequence Functions

#v

\{[a] \rightarrow \text{int} :: (a \text{ in any})\}

Returns the length of a sequence.

dist(a, 1)

\{(a, \text{int}) \rightarrow [a] :: (a \text{ in any})\}

Generates a sequence of length 1 with the value \(a\) in each element. For example:

\[
\begin{align*}
a &= a_0 \\
n &= 5
\end{align*}
\]

dist(a, 1) = [a_0, a_0, a_0, a_0, a_0]

elt(a, i)

\{([a], \text{int}) \rightarrow a :: (a \text{ in any})\}

Extracts the element specified by index \(i\) from the sequence \(a\). Indices are zero-based.

rep(d, v, i)

\{([a], a, \text{int}) \rightarrow [a] :: (a \text{ in any})\}

Replaces the \(i\)th value in the sequence \(d\) with the value \(v\). For example:

\[
\begin{align*}
d &= [a_0, a_1, a_2, a_3, a_4] \\
v &= b_0 \\
n &= 3
\end{align*}
\]

rep(d, v, i) = [a_0, a_1, a_2, b_0, a_4]

zip(a, b)

\{([b], [a]) \rightarrow ([b], [a]) :: (a \text{ in any}; b \text{ in any})\}

Zips two sequences of equal length together into a single sequence of pairs.

unzip(a)

\{([b], [a]) \rightarrow ([b], [a]) :: (a \text{ in any}; b \text{ in any})\}

Unzips a sequence of pairs into a pair of sequences.

Scans and Reduces

plus_scan(a)

\{[a] \rightarrow [a] :: (a \text{ in number})\}

Given a sequence of numbers, \text{plus_scan} returns to each position of a new equal-length sequence, the sum of all previous positions in the source. For example:

\[
\begin{align*}
a &= [1, 3, 5, 7, 9, 11, 13, 15] \\
\text{plus_scan}(a) &= [0, 1, 4, 9, 16, 25, 36, 49]
\end{align*}
\]
max_scan(a) \quad \{[a] \rightarrow [a] :: (a \text{ in ordinal})\}

Given a sequence of ordinals, max_scan returns to each position of a new equal-length sequence, the maximum of all previous positions in the source. For example:

\[
\begin{align*}
    a &= [3, 2, 1, 6, 5, 4, 8] \\
    \text{max_scan(a)} &= [-\infty, 3, 3, 3, 6, 6, 6]
\end{align*}
\]

min_scan(a) \quad \{[a] \rightarrow [a] :: (a \text{ in ordinal})\}

Given a sequence of ordinals, min_scan returns to each position of a new equal-length sequence, the minimum of all previous positions in the source.

or_scan(a) \quad \{[a] \rightarrow [a] :: (a \text{ in logical})\}

A scan using logical-or on a sequence of logicals.

and_scan(a) \quad \{[a] \rightarrow [a] :: (a \text{ in logical})\}

A scan using logical-and on a sequence of logicals.

iseq(s, d, e) \quad \{(int, int, int) \rightarrow [int]\}

Returns a set of indices starting at s, increasing by d, and finishing before e. For example:

\[
\begin{align*}
    s &= 4 \\
    d &= 3 \\
    e &= 15 \\
    \text{iseq(s, d, e)} &= [4, 7, 10, 13]
\end{align*}
\]

sum(v) \quad \{[a] \rightarrow a :: (a \text{ in number})\}

Given a sequence of numbers, sum returns their sum. For example:

\[
\begin{align*}
    v &= [7, 2, 9, 11, 3] \\
    \text{sum(v)} &= 32
\end{align*}
\]

max_val(v) \quad \{[a] \rightarrow a :: (a \text{ in ordinal})\}

Given a sequence of ordinals, max_val returns their maximum.

min_val(v) \quad \{[a] \rightarrow a :: (a \text{ in ordinal})\}

See max_val.

any(v) \quad \{[a] \rightarrow a :: (a \text{ in logical})\}

Given a sequence of booleans, any returns t iff any of them are t.

all(v) \quad \{[a] \rightarrow a :: (a \text{ in logical})\}
Given a sequence of booleans, all returns t iff all of them are t.

count(v) \{[bool] -> int\}
Counts the number of t flags in a boolean sequence. For example:

\[ \text{count}(v) = 5 \]

max_index(v) \{[a] -> int :: (a in ordinal)\}
Given a sequence of ordinals, max_index returns the index of the maximum value. If several values are equal, it returns the leftmost index. For example:

\[ v = [2, 11, 4, 7, 14, 6, 9, 14] \]
\[ \text{max_index}(v) = 4 \]

min_index(v) \{[a] -> int :: (a in ordinal)\}
Given a sequence of ordinals, min_index returns the index of the minimum value. If several values are equal, it returns the leftmost index.

Sequence Reordering Functions

values -> indices \{([a], [int]) -> [a] :: (a in any)\}
Given a sequence of values on the left and a sequence of indices on the right, which can be of different lengths, -> returns a sequence which is the same length as the indices sequence and the same type as the values sequence. For each position in the indices sequence, it reads the value at that index of the values sequence. For example:

\[ \text{values} = [\text{tto}, \text{«i»}, \text{«2»}, \text{«3»}, \text{«4»}, \text{«5»}, \text{«6»}, \text{«7»}] \]
\[ \text{indices} = [3, 5, 2, 6] \]
\[ \text{values} \rightarrow \text{indices} = [\text{as}, \text{as}, \text{aj}^*] \]

permute(v, i) \{([a], [int]) -> [a] :: (a in any)\}
Given a sequence v and a sequence of indices i, which must be of the same length, permute permutes the values to the given indices. The permutation must be one-to-one.

\[ d \leftarrow \text{ivpairs} \{([a], [(int, a)]) -> [a] :: (a in any)\} \]
This operator, called write, is used to write multiple elements into a sequence. Its left argument is the sequence to write into (the destination sequence) and its right argument is a sequence of integer-value pairs. For each element \((i, v)\) in the sequence of integer-value pairs, the value \(v\) is written into position \(i\) of the destination sequence.
rotate(a, i) \{([a], int) \rightarrow [a] :: (a \text{ in any})\}

Given a sequence and an integer, rotate rotates the sequence around by i positions to the right. If the integer is negative, then the sequence is rotated to the left. For example:

\[
\begin{align*}
a &= [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \\
i &= 3 \\
\text{rotate}(a, i) &= [a_5, a_6, a_7, a_0, a_1, a_2, a_3, a_4]
\end{align*}
\]

reverse(a) \{[a] \rightarrow [a] :: (a \text{ in any})\}
Reverses the order of the elements in a sequence.

Simple Sequence Manipulation

pack(v) \{([(a, bool)] \rightarrow [a] :: (a \text{ in any})\}

Given a sequence of (value,flag) pairs, pack packs all the values with a \text{t} in their corresponding flag into consecutive elements, deleting elements with an \text{f}.

\[
\begin{align*}
v1 &= [a_0, a_1, a_2] \\
v2 &= [b_0, b_1] \\
v1 ++ v2 &= [a_0, a_1, a_2, b_0, b_1]
\end{align*}
\]

subseq(v, start, end) \{([a], int, int) \rightarrow [a] :: (a \text{ in any})\}

Given a sequence, subseq returns the subsequence starting at position \text{start} and ending one before position \text{end}. For example:

\[
\begin{align*}
v &= [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \\
\text{start} &= 2 \\
\text{end} &= 6 \\
\text{subseq}(v, \text{start}, \text{end}) &= [a_2, a_3, a_4, a_5]
\end{align*}
\]

drop(v, n) \{([a], int) \rightarrow [a] :: (a \text{ in any})\}

Given a sequence, drop drops the first n items from the sequence. For example:

\[
\begin{align*}
v &= [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \\
n &= 3 \\
\text{drop}(v, n) &= [a_3, a_4, a_5, a_6, a_7]
\end{align*}
\]
take(v, n) \quad \{(\mathbb{A}, \text{int}) \rightarrow \mathbb{A} :: (a \text{ in any})\}

Given a sequence, \text{take} takes the first \(n\) items from the sequence. For example:

\[
v = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]\\
n = 3\\
take(v, n) = [a_0, a_1, a_2]
\]

\text{odd_elts}(v) \quad \{\mathbb{A} \rightarrow \mathbb{A} :: (a \text{ in any})\}

Returns the odd indexed elements of a sequence.

\text{even_elts}(v) \quad \{\mathbb{A} \rightarrow \mathbb{A} :: (a \text{ in any})\}

Returns the even indexed elements of a sequence.

\text{interleave}(a, b) \quad \{(\mathbb{A}, \mathbb{A}) \rightarrow \mathbb{A} :: (a \text{ in any})\}

Interleaves the elements of two sequences. The sequences must be of the same length. For example:

\[
a = [a_0, a_1, a_2, a_3]\\
b = [b_0, b_1, b_2, b_3]\\
\text{interleave}(a, b) = [a_0, b_0, a_1, b_1, a_2, b_2, a_3, b_3]
\]

\text{length_from_flags}(flags) \quad \{\mathbb{B} \rightarrow \mathbb{I} \}

Given a sequence of boolean flags, this will return the length from each true flag to the next. A true flag is always assumed in the first position.

**Nesting Sequences**

The two functions \text{partition} and \text{flatten} are the primitives for moving between levels of nesting. All other functions for moving between levels of nesting can be built out of these. The functions \text{split} and \text{bottop} are often useful for divide-and-conquer routines.

\text{partition}(v, counts) \quad \{(\mathbb{A}, \mathbb{I}) \rightarrow [[\mathbb{A}]] :: (a \text{ in any})\}

Given a sequence of values and another sequence of counts, \text{partition} returns a nested sequence with each subsequence being of a length specified by the counts. The sum of the counts must equal the length of the sequence of values. For example:

\[
v = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]\\
counts = [4, 1, 3]\\
\text{partition}(v, counts) = [[a_0, a_1, a_2, a_3], [a_4], [a_5, a_6, a_7]]
\]

\text{flatten}(v) \quad \{[[\mathbb{A}]] \rightarrow \mathbb{A} :: (a \text{ in any})\}

Given a nested sequence of values, \text{flatten} flattens the sequence. For example:
\[ v = [[a_0, a_1, a_2], [a_3, a_4], [a_5, a_6, a_7]] \]
\[ \text{flatten}(v) = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \]

\[ \text{split}(v, \text{flags}) \]
\[ = \{([a], [b]) \rightarrow \langle[a]\rangle :: \langle a \rangle \in \text{any} \} \]

Given a sequence of values \( v \) and a boolean sequence of \( \text{flags} \), \( \text{split} \) creates a nested sequence of length 2 with all the elements with an \( f \) in their flag in the first element and elements with a \( t \) in their flag in the second element. For example:

\[ v = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \]
\[ \text{flags} = [T, F, T, F, F, T, T, T] \]
\[ \text{split}(v, \text{flags}) = [[[a_1, a_3, a_4]], [[a_0, a_2, a_5, a_6, a_7]]] \]

\[ \text{bottop}(v) \]
\[ = \{[a] \rightarrow \langle[a]\rangle :: \langle a \rangle \in \text{any} \} \]

Given a sequence of values \( \text{values} \), \( \text{bottop} \) creates a nested sequence of length 2 with all the elements from the bottom half of the sequence in the first element and elements from the top half of the sequence in the second element. For example:

\[ v = [a_0, a_1, a_2, a_3, a_4, a_5, a_6] \]
\[ \text{bottop}(v) = [[a_0, a_1, a_2, a_3], [a_4, a_5, a_6]] \]

\[ \text{head_rest}(\text{values}) \]
\[ = \{[a] \rightarrow (a, [a]) :: \langle a \rangle \in \text{any} \} \]

Given a sequence of values \( \text{values} \) of length \( > 0 \), \( \text{head_rest} \) returns a pair containing the first element of the sequence, and the remaining elements of the sequence.

\[ \text{rest_tail}(\text{values}) \]
\[ = \{[a] \rightarrow ([a], a) :: \langle a \rangle \in \text{any} \} \]

Given a sequence of values \( \text{values} \) of length \( > 0 \), \( \text{rest_tail} \) returns a pair containing all but the last element of the sequence, and the last element of the sequence.

**Other Sequence Functions**

These are more complex sequence functions. The step complexities of these functions are not necessarily \( O(1) \).

\[ \text{sort}(a) \]
\[ = \{[a] \rightarrow [a] :: \langle a \rangle \in \text{number} \} \]

Sorts the input sequence.

\[ \text{rank}(a) \]
\[ = \{[a] \rightarrow \langle\text{int}\rangle :: \langle a \rangle \in \text{number} \} \]

Returns the rank of each element of the sequence \( a \). The rank of an element is the position it would appear in if the sequence were sorted. A sort of a sequence \( a \) can be implemented as \( \text{permute}(a, \text{rank}(a)) \). The rank is stable.

\[ \text{collect}(\text{key_value_pairs}) \]
\[ = \{([b, a]) \rightarrow ([b, [a]]) :: \langle a \rangle \in \text{any}; \langle b \rangle \in \text{any} \} \]
Takes a sequence of (key, value) pairs, and collects each set of values that have the same key together into a sequence. The function returns a sequence of (key, value-sequence) pairs. Each key will only appear once in the result and the value-sequence corresponding to the key will contain all the values that had that key in the input.

\[
\text{int\_collect}(\text{key\_value\_pairs}) = \{(\text{int, a}) \rightarrow\{(\text{int, \[a\])} :: (a \text{ in any})
\]

Version of collect that works when the keys are integers. As well as collecting, the subsequences are returned with the keys in sorted order.

\[
\text{kth\_smallest}(s, k) = \{(\[a\], \text{int}) \rightarrow a :: (a \text{ in ordinal})
\]

Returns the kth smallest element of a sequence s (k is 0 based). It uses the quick-select algorithm and therefore has expected work complexity of \(O(n)\) and an expected step complexity of \(O(\lg n)\).

\[
\text{find}(\text{element}, \text{seq}) = \{(a, \[a\]) \rightarrow\text{int} :: (a \text{ in any})
\]

Returns the index of the first place that element is found in seq. If element does not appear in the sequence, then -1 is returned.

\[
\text{search\_for\_subseqs}(\text{subseq, sequence}) = \{(\[«'\] N) \rightarrow\text{int} :: (a \text{ in any})
\]

Returns indices of all start positions in sequence where the string specified by subseq appears.

\[
\text{remove\_duplicates}(s) = \{[a] \rightarrow[a] :: (a \text{ in any})
\]

Removes duplicates from a sequence. Elements are considered duplicates if eql on them returns T.

\[
\text{mark\_duplicates}(s) = \{[a] \rightarrow[\text{bool}] :: (a \text{ in any})
\]

Marks the duplicates such that only one instance of each value in a sequence is marked with a true flag. Elements are considered duplicates if eql on them returns T.

\[
\text{union}(a, b) = \{(\[a\], \[a\]) \rightarrow[a] :: (a \text{ in any})
\]

Given two sequences each which has no duplicates, union will return the union of the elements in the sequences.

\[
\text{intersection}(a, b) = \{(\[a\], \[a\]) \rightarrow[a] :: (a \text{ in any})
\]

Given two sequences each which has no duplicates, intersection will return the intersection of the elements in the sequences.

\[
\text{name}(a) = \{[a] \rightarrow[\text{int}] :: (a \text{ in any})
\]

This function assigns an integer label to each unique value of the sequence a. Equal values will always be assigned the same label and different values will always be assigned different labels. All the labels will be in the range \([0..\#a]\) and will correspond to the position in a of one of the elements with the same value. The function remove\_duplicates(a) could be
implemented as \{s \text{ in } a; \ i \text{ in } [0: \#a]; \ r \text{ in } \text{name}(a) \mid r == i\}.

\text{transpose}(a) \quad \{[a] \to [[a]] :: (a \text{ in any})\}

Transposes a nested sequence. For example \text{transpose}([[2,3,4],[5,6,7]]) would return [[2,5],[3,6],[4,7]]. All the subsequence must be the same length.

### B.3 Functions on Any Type

\text{eql}(a, b) \quad \{(a, a) \to \text{bool} :: (a \text{ in any})\}

Given two objects of the same type, \text{eql} will return \text{t} if they are equal and \text{f} otherwise. Two sequences are equal if they are the same length and their elements are elementwise equal. Two records are equal if their fields are equal.

\text{hash}(a, 1) \quad \{(a, \text{int}) \to \text{int} :: (a \text{ in any})\}

Hashes the argument \(a\) and returns an integer in the range \([0..1)\). This will always generate the same result for equal values as long as it is run on the same machine. In particular floating-point hashing can depend on the floating-point representation, which is machine dependent. There is no guarantee about the distribution of the results—returning 0 for all keys would be a valid implementation, although we expect an implementation to do much better than that.

\text{select}(\text{flag}, v_1, v_2) \quad \{(\text{bool}, a, a) \to a :: (a \text{ in any})\}

Returns the second argument if the flag is \text{T} and the third argument if the flag is \text{F}. This differs from an \text{if} form in that both arguments are evaluated.

\text{identity}(a) \quad \{a \to a :: (a \text{ in any})\}

Returns the identity for any type. The identity of a sequence is an empty sequence of the same type. The identity of a number is 0, the identity of a boolean is \text{f} (false), and the identity of a character is the null character. The identity of a pair is a pair of the identities of the two elements.

### B.4 Functions for Manipulating Strings

\@v \quad \{a \to \text{char} :: (a \text{ in any})\}

Given any printable object \(v\), @ converts it into its printable representation as a character string.

\text{exp_string}(\text{val}, \text{digits after point}) \quad \{(\text{float}, \text{int}) \to \text{char}\}

Prints a floating-point number in exponential notation. The second argument specifies how many digits should be printed after the decimal point (currently this cannot be larger than 8).
str || 1 \{([char], int) → [char]\}

Pads a string str into a string of length 1 with the string left justified. If 1 is negative, then the string is right justified.

linify(str) \{[char] → [char]\}

Breaks up a string into lines (a sequence of strings). Only a newline is considered a separator. All separators are removed.

wordify(str) \{[char] → [char]\}

Breaks up a string into words (a sequence of strings). Either a space, tab, or newline is considered a separator. All separators are removed.

lowercase(char) \{char → char\}

Converts a character string into lowercase characters.

uppercase(char) \{char → char\}

Converts a character string into uppercase characters.

string_eq1(str1, str2) \{([char], [char]) → bool\}

Compares two strings for equality without regards to case.

parse_int(str) \{[char] → (int, bool)\}

 Parses a character string into an integer. Returns the integer and a flag specifying whether the string was successfully parsed. The string must be in the format: \([+-]?[0..9]*\).

parse_float(str) \{[char] → (float, bool)\}

 Parses a character string into a float. Returns the float and a flag specifying whether the string was successfully parsed. The string must be in the format: \([+-]?[0..9]*[.]?[0..9]*)((e[+-]?[0..9]*)?\.

B.5 Functions with Side Effects

The functions in this section are not purely functional. Unless otherwise noted, none of them can be called in parallel—they cannot be called within an apply-to-each construct. The routines in this section are not part of the core language, they are meant for debugging, I/O, timing and display. Because these functions are new it is reasonably likely that the interface of some of these functions will change in future versions. The user should check the most recent documentation.

Input and Output Routines

Of the functions listed in this section, only print_char, print_string, write_char, write_string, and write_check can be called in parallel.
print_char(v) \quad \{\text{char} \rightarrow \text{bool}\}

Prints a character to standard output.

print_string(v) \quad \{\text{[char]} \rightarrow \text{bool}\}

Prints a character string to standard output.

write_object_to_file(object, filename) \quad \{(a, [\text{char}]) \rightarrow \text{bool} :: (a \in \text{any})\}

Writes an object to a file. The first argument is the object and the second argument is a filename. For example call write_object_to_file([2,3,1,0],"/tmp/foo") would write a vector of integers to the file /tmp/foo. The data is stored in an internal format and can only be read back using read_object_from_file.

write_string_to_file(a, filename) \quad \{(\text{[char]}, [\text{char}]) \rightarrow \text{bool}\}

Writes a character string to the file named filename.

append_string_to_file(a, filename) \quad \{(\text{[char]}, [\text{char}]) \rightarrow \text{bool}\}

Appends a character string to the file named filename.

read_object_from_file(object_type, filename) \quad \{(a, [\text{char}]) \rightarrow a :: (a \in \text{any})\}

Reads an object from a file. The first argument is an object of the same type as the object to be read, and the second argument is a filename. For example, the call read_object_from_file(0,"/tmp/foo") would read an integer from the file /tmp/foo, and read_object_from_file(\[\text{int}],"/tmp/bar") would read a vector of integers from the file /tmp/foo. The object needs to have been stored using the function write_object_to_file.

read_string_from_file(filename) \quad \{[\text{char}] \rightarrow [\text{char}]\}

Reads a whole file into a character string.

read_int_seq_from_file(filename) \quad \{[\text{char}] \rightarrow [\text{int}]\}

Reads a sequence of integers from the file named filename. The file must start with a left parenthesis, contain the integers separated by either white spaces, newlines or tabs, and end with a right parenthesis. For example:

( 22 33 11
 10 14
12 11 )

represents the sequence [22, 33, 11, 10, 14, 12, 11].

read_float_seq_from_file(filename) \quad \{[\text{char}] \rightarrow [\text{float}]\}

Reads a sequence of floats from the file named filename. The file must start with a left parenthesis, contain the floats separated by either white spaces, newlines or tabs, and end with a right parenthesis. The file may contain integers (no .); these will be coerced to floats.

open_in_file(filename) \quad \{[\text{char}] \rightarrow (\text{stream}, \text{bool}, [\text{char}]\}

49
Opens a file for reading and returns a stream for that file along with an error flag and an error message.

`open_out_file(filename)`

Opens a file for writing and returns a stream for that file along with an error flag and an error message. File pointers cannot be returned to top-level. They must be used within a single top-level call.

`close_file(str)`

Closes a file given a stream. It returns an error flag and an error message.

`write_char(a, stream)`

Prints a character to the stream specified by stream. It returns an error flag and error message.

`write_string(a, stream)`

Prints a character string to the stream specified by stream. It returns an error flag and error message.

`read_char(stream)`

Reads a character from stream. If the end-of-file is reached, the null character is returned along with the success flag set to false.

`read_string(delim, maxlen, stream)`

Reads a string from the stream stream. It will read until one of the following is true (whichever comes first):

1. the end-of-file is reached,
2. one of the characters in the character array `delim` is reached,
3. `maxlen` characters have been read.

If `maxlen` is negative, then it is considered to be infinity. The `delim` character array can be empty.

`read_line(stream)`

Reads all the characters in stream up to a newline or the end-of-file (whichever comes first). The newline is consumed and not returned. As well as returning the line, it returns a boolean flag indicating whether reading was terminated on a newline (f) or EOF (t).

`read_word(stream)`

Reads all the characters in stream up to a newline, space, tab or the end-of-file (whichever
comes first). The newline, space or tab is consumed and not returned. As well as returning the line, it returns a (char,bool) pair that indicates on what character the word was terminated and whether it was terminated on a EOF (the bool is t).

\[ \text{open_check(str, flag, err_message)} \]  
\{ (a, bool, [char]) \rightarrow a :: (a in any) \}  
Checks if an open on a file succeeded and prints an error message if it did not. For example, in the form \text{open_check(open_in_file("/usr/foo/bar"))}, if the open is successful it will return a stream, otherwise it will print an error message and return the null stream.

\[ \text{write_check(flag, err_message)} \]  
\{ (bool, [char]) \rightarrow bool \}  
Checks if a write succeeded and prints an error message if it did not. For example, in the form \text{write_check(write_string("foo", stream))}, if the write is successful it will return t, otherwise it will print an error message and return f.

\[ \text{read_check(val, flag, err_message)} \]  
\{ (a, bool, [char]) \rightarrow a :: (a in any) \}  
Checks if a read succeeded and prints an error message if it did not. It also strips off the error information from the read functions. For example, in the form \text{read_check(read_char(stream))}, if the read is successful it will return the character which is read, otherwise it will print an error message.

\[ \text{close_check(flag, err_message)} \]  
\{ (bool, [char]) \rightarrow bool \}  
Checks if a close on a stream succeeded and prints an error message if it did not. For example, in the form \text{close_check(close_file(stream))}, if the close is successful it will return t, otherwise it will print an error message and return f.

\[ \text{nullstr} \]  
\{ stream \}  
The null stream.

\[ \text{stdin} \]  
\{ stream \}  
The standard input stream.

\[ \text{stdout} \]  
\{ stream \}  
The standard output stream.

\[ \text{stderr} \]  
\{ stream \}  
The standard error stream.

**Plotting Functions**

The functions in this section can be used for plotting data on an Xwindow display. The basic idea of these functions is that you create a window with the \text{w.make_window} command and then can add various features to the window. The most important features are boxes and buttons. A scale box can be used to create a virtual coordinate system on the window on which points, lines, rectangles and polygons can be drawn. A text box can be used to create
a box in which text can be written. A button can be used along with the input functions to
get information back from the window. In all the functions in this section colors are specified
by one of w\_black, w\_white, w\_red, w\_blue, w\_green, w\_cyan, w\_yellow, w\_magenta, w\_pink,
w\_light\_green, w\_light\_blue, w\_purple, w\_gray, w\_dark\_gray, or w\_orange.

```
display
{[char]}
The display variable inherited from your environment.
```

```
\textbf{w\_make\_window((offset, size), title, background\_color, display)}

\{(((int, int), int, int, [char], int, [char], stream) \rightarrow w\_window}\}
```

This makes an X window on the specified display with the given offset, size, title, and background
color.

```
Note that windows get automatically closed when you return to top-level. This means
that you cannot return a window to top-level and then use it—you must create it and use
it within a single top-level call.
```

```
w\_kill\_window(win)
{w\_window \rightarrow bool}
```

Kills a window created by \textbf{w\_make\_window}.

```
w\_add\_box((offset, size), (voffset, vsize), name, color, window)

\{(((int, int), int, int), ((float, float), float, float), [char], int, w\_window) \rightarrow (w\_window, w\_box)\}
```

Creates a scaled box on the specified window, which can be used to draw into using the
various drawing commands. The position and size of the box within the window are specified
by offset and size. The virtual coordinates of the box are specified by the virtual offset
(voffset) and the virtual size (vsize). The color specifies the background color. The
function returns a pair whose first element in the modified window structure and whose
second element is the newly created box. Technically this box is not necessary since it can
be retrieved with \textbf{w\_get\_named\_box}.

```
w\_add\_text\_box((offset, size), name, color, window)

\{(((int, int), int, int), [char], int, w\_window) \rightarrow (w\_window, w\_box)\}
```

Create a text box on the specified window with the given offset, size and color. Such a box
can be used in conjunction \textbf{w\_write\_paragraph}, \textbf{w\_write\_text\_centered} or \textbf{w\_write\_text\_left}.
These will all write text into a text box.

```
w\_add\_button((offset, size), name, color, window)

\{(((int, int), int, int), [char], int, w\_window) \rightarrow w\_window\}
```

Adds a button to a window. The name is printed across the button and is also the name
that is returned by \textbf{w\_get\_input} and related functions if the button is pressed.

```
w\_add\_button\_stack((offset, size, separation), names, color, window)

\{(((int, int), (int, int), int, int), ([char]), int, w\_window) \rightarrow w\_window\}
```

52
Adds a stack of buttons. The separation is specified as an x and y distance (e.g., if the x distance is 0, then the buttons will vertical). names is an array of button names.

\[
\text{w.add_text}(\text{position, text, color, window}) \quad \{((\text{int, int}), [\text{char}], \text{int, w.window}) \rightarrow \text{w.window}\}
\]

Adds a single text string to a window at the specified position. The function \text{w.add.text.box} is often more convenient.

\[
\text{w.get_named.box}(\text{name, window}) \quad \{([\text{char}], \text{w.window}) \rightarrow \text{w.box}\}
\]

Returns the box of the specified name from a window.

\[
\text{w.reset.box.size}(\text{boxname, (voffset, vsize), window}) \quad \{([\text{char}], ((\text{float, float}), \text{float, float}), \text{w.window}) \rightarrow (\text{w.window, w.box})\}
\]

Resets the virtual coordinates of a scaled box inside of a window. You need to pass this command the name of the box rather than the box itself. Like \text{w.make.box}, it returns both the new window structure and a new box. This command does not clear the current contents of the box.

\[
\text{w.clear.box}(\text{box}) \quad \{\text{w.box} \rightarrow \text{bool}\}
\]

Clears the contents of a box.

\[
\text{w.bounds.from.box}(\text{box}) \quad \{\text{w.box} \rightarrow ((\text{float, float}), \text{float, float})\}
\]

Returns the virtual coordinates of a scale box.

\[
\text{w.box.scale}(\text{box}) \quad \{\text{w.box} \rightarrow \text{float}\}
\]

Returns the ratio of the size of a box in pixels to the size in the virtual coordinate system.

\[
\text{w.bounding.box}(\text{points}) \quad \{([a, a]) \rightarrow ((a, a), a, a) :: (a \text{ in number})\}
\]

For a set of points this function returns a bounding box (the smallest box that contains them all). The bounding box is specified as a pair in which the first element is the (x,y) coordinate of the lower left corner and the second element is a pair with the width and height.

\[
\text{w.draw.point}(\text{point, color, box}) \quad \{((\text{float, float}), \text{int, w.box}) \rightarrow \text{bool}\}
\]

Draws a point in a box based on the virtual coordinates.

\[
\text{w.draw.big.point}(\text{point, size, color, box}) \quad \{((\text{float, float}), \text{int, int, w.box}) \rightarrow \text{bool}\}
\]

Draws a big point in a box based on the virtual coordinates. The point size is specified by an integer (pixels).

\[
\text{w.draw.points}(\text{points, color, box}) \quad \{([\text{float, float}]), \text{int, w.box}) \rightarrow \{\text{bool}\}\}
\]

Draws a sequence of points in a box based on the virtual coordinates.
\( \texttt{w\_draw\_segments} \) \( ((\text{endpoints}, \text{width}, \text{color}, \text{box}) \to \text{bool}) \)

Draws a sequence of line segments in a box based on virtual coordinates. Each line-segment is specified as a pair of points. The \texttt{width} argument specifies the width of the lines in pixels. All segments that go outside of the box are clipped.

\( \texttt{w\_draw\_string} \) \( (((\text{point}, \text{string}, \text{color}, \text{box}) \to \text{bool}) \)

Draws a text string in a box at the position specified by \texttt{point} in virtual coordinates.

\( \texttt{w\_draw\_rectangle} \) \( (((\text{offset, size, width, color, box}) \to \text{bool} :: (a \in \text{number}; b \in \text{number})) \)

Draws a rectangle in a box based on the virtual coordinates. This function clips any part of the rectangle that goes outside the box. The \texttt{width} specified the width of the lines.

\( \texttt{w\_shade\_rectangle} \) \( (((\text{offset, size, color, box}) \to \text{bool} :: (a \in \text{number}; b \in \text{number})) \)

Shades a rectangle in a box based on the virtual coordinates. This function clips any part of the rectangle that goes outside the box.

\( \texttt{w\_shade\_polygon} \) \( (((\text{points, color, box}) \to \text{bool}) \)

Shades a polygon in a box based on the virtual coordinates. The polygon is specified as a sequence of points. This function clips any part of the polygon that goes outside of the box.

\( \texttt{w\_write\_text\_centered} \) \( (((\text{text, color, text\_box}) \to \text{bool}) \)

Writes text into a text box such that the text is centered both horizontally and vertically.

\( \texttt{w\_write\_text\_left} \) \( (((\text{text, color, text\_box}) \to \text{bool}) \)

Writes text into a text box such that it is against the left end of the box and centered vertically.

\( \texttt{w\_write\_paragraph} \) \( (((\text{text, color, text\_box}) \to \text{bool}) \)

Writes a paragraph into a text box starting at the top left hand corner of the box and does line wrapping so the text will not go outside of the right margin of the box. Line breaks are ignored although the @ character can be used to force a line break.

\( \texttt{w\_get\_input} \) \( \text{(window)} \)

Gets input from a window. It returns a tuple of the form \texttt{(event\_type, name, position, flags, char)}. The event\_type is one of "button", "key", "box", or "none" depending on the event. The "box" event in evoked if you click the mouse inside of a box. The name is the name of the box or button on which the event took place. The position specifies the the virtual coordinate on which a "box" event took place. The flags specify whether control and shift keys were pressed. The character specifies the key pressed in the case of a "key" event.
This function blocks until an event takes place.

\[
w_{\text{get\_input\_noblock}}(\text{window})
\]
\[
  \{w_{\text{window}} \rightarrow [\text{bool}, [\text{char}], [\text{char}], (\text{float}, \text{float}), (\text{int}, \text{int}), \text{char}]\}
\]

This is the same as \(w_{\text{get\_input}}\) but does not block. It returns an additional flag at the begining, which specifies if an event was found.

\[
w_{\text{get\_button\_input}}(\text{window})
\]
\[
  \{w_{\text{window}} \rightarrow [\text{char}]\}
\]

This function only picks up button events and throws away all other events. It returns the button name.

\[
w_{\text{get\_zoom\_box}}(\text{pt}, \text{width}, \text{box})
\]
\[
  \{(\text{float}, \text{float}), a, w_{\text{box}} \rightarrow ((\text{float}, \text{float}), \text{float}, \text{float}) :: (a \in \text{any})\}
\]

This function allows to to use an elastic band to pick out a region of a box. It is typically used in conjunction with \(w_{\text{get\_input}}\). In particular, if the input returned by \(w_{\text{get\_input}}\) is a mouse click on a particular box (down click) the position of the click and the box can be passed to this function which will then return the other corner (the corner on which the user lets go of the mouse). This function actually returns the lower left and upper right corners of the region. The width specifies the width in pixels of the elastic band.

**Shell Commands**

The functions in this section can be used to execute shell commands from within Nesl.

\[
\text{shell\_command(name, input)}
\]
\[
  \{([\text{char}], [\text{char}]) \rightarrow [\text{char}]\}
\]

Executes the shell command given by \text{name}. If the second argument is not the empty string, then it is passed to the shell command as standard input. The \text{shell\_command} function returns its standard output as a string. For example, the command \text{shell\_command("cat", "dog")} would return "dog".

\[
\text{get\_environment\_variable(name)}
\]
\[
  \{[\text{char}] \rightarrow [\text{char}]\}
\]

Gets the value of an environment variable. Will return the empty string if there is no such variable.

\[
\text{spawn(command, stdin, stdout, stderr)}
\]
\[
  \{([\text{char}], \text{stream}, \text{stream}, \text{stream}) \rightarrow ((\text{stream}, \text{stream}, \text{stream}), \text{bool}, [\text{char}])\}
\]

Creates a subprocess (using unix fork). The \text{spawn} function takes 4 arguments:

- execution string - a string that will be passed to execvp
- input stream - a stream descriptor - stdin of new process
- output stream - a stream descriptor - stdout of new process
- error stream - a stream descriptor - stderr of new process
The function returns three file descriptors a boolean status flag and an error message: `((stdin, stdout, stderr), (flag, message))`. For any non null stream passed to spawn, spawn will return the same stream and use that stream as stdin, stdout or stderr. If the null stream is passed for any of the three stream arguments, then spawn will create a new stream and pass back a pointer to it.

**Other Side Effecting Functions**

```haskell
time(a) \{ a \rightarrow (a, float) :: (a in any) \}
```

The expression `TIME(exp)` returns a pair whose first element is the value of the expression `exp` and whose second element is the time in seconds taken to execute the expression `exp`.
Table 4: Work complexities of some of the sequence functions in the current implementation. In all cases the flat vs. nested refers to the variable a. S(v) refers to the size of the object v and LL(v) is described in the text.

### C Implementation Notes

This section describes how the complexity of the current implementation differs in some ways from the complexity measures defined in Table 1 and Section 1.5. A more detailed description of the cost measures of NESL can be found in a separate document [23].

#### The Sequence Functions

In the current implementation the equations for the work performed by some of the sequence functions depends on whether the sequence is nested or not. Table 4 gives the complexities for these functions. A sequence is considered nested if its elements contain sequences. A sequence of pairs of scalars or pairs is not considered nested. The function LL(a) refers the the length of a if it was flattened until it has just one level of nesting. For example, LL([[2, 3], [1], [8, 9, 10]], [[1, 2, 3, 4]]) would be 4, since when flattened to one level of nesting it has the value [[2, 3], [1], [8, 9, 10], [1, 2, 3, 4]], which has length 4. In rep and <-, the work complexity for flat sequences depends on whether the variable used for d is the final reference to that variable (arguments are evaluated left to right). If it is the final reference, then the complexity before the comma is used, otherwise the complexity after the comma is used.

#### Apply-to-each

Equations 1 and 2 specified how the work and depth complexities could be combined in an apply-to-each. In the current implementation there are a couple caveats. The first concerns work complexity. In the following discussion we will consider a variable constant with regards to an apply-to-each if the variable is free (not bound) in the body of the apply-to-each and is not defined in bindings of the apply-to-each. For example, in

```
{foo(a,b*c): b in s}
```

the variables a and c are free with regards to the apply-to-each, while b is not. We will refer to these variables as free-vars. In the current implementation all free-vars need to
be copied across the instances of an apply-to-each. This copying requires time, and the equation for combining work complexity that includes this cost is:

\[
W(\{s1(a) : a \in s2(b)\}) = W(s2(b)) + \sum\{W(s1(a)) : a \in s2(b)\} + \sum_{c \in \text{free-vars}} (\text{Length}(s2(b)) \times \text{Size}(c))
\]

where the last term has been added to Equation 1 ($\text{Length}(s2(b))$ refers to the length of the sequence returned by $s2(b)$, and $\text{Size}(c)$ refers to the size of each free-var). If a free-var is large, this copy could be the dominant cost of an apply-to-each. Here are some examples of such cases:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Work Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>${#a + i : i \in a}$</td>
<td>$(#a)^2$</td>
</tr>
<tr>
<td>${a[i] : i \in b}$</td>
<td>$#a \times #b$</td>
</tr>
</tbody>
</table>

In both cases the work is a factor of $#a$ greater than we might expect since the sequence $a$ needs to be copied over the instances. As well as requiring extra work these copies require significant extra memory and can be a memory bottleneck in a program. Both the above examples can easily be rewritten to reduce the work and memory:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Work Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $b = #a$ in ${b + i : i \in a}$</td>
<td>$#a$</td>
</tr>
<tr>
<td>$a \rightarrow b$</td>
<td>$#b$</td>
</tr>
</tbody>
</table>

The user should be conscious of these costs and rewrite such expressions.

A second problem with the current implementation is that Equation 2 (the combining rule for the depth) only holds if the body of the apply-to-each is contained. The definition of contained code is code where only one branch of a conditional has a non-constant depth. For example, the following function is not contained because both branches of the inner if have $D(n) > O(1)$:

```plaintext
function power(a, n) =
    if (n == 0) then 1
    else
        if evenp(n)
            then square(power(a, n/2))
        else a * square(power(a, n/2))
```

This can be fixed by calculating $\text{power}(a, n/2)$ outside the conditional:

```plaintext
function power(a, n) =
    if (n == 0) then 1
```
else
    let pow = power(a, n/2)
    in if evenp(n)
        then square(pow)
        else a * square(pow)

In future implementations of NESL it is likely that this restriction will be removed.
Index

*, 37
++ (append), 43
+, 36
, (pair), 10
-> (read), 16, 42
-, 36
/=, 35
/, 37
<- (write), 16, 42
<=, 36
<, 36
==, 35
=, 28, 32
>=, 36
>, 36
@ (string), 47
# (length), 40
{} (apply-to-each), 26
|| (format), 48
abs, 36
acos, 38
all, 42
and_scan, 41
and, 35
any, 41
append_string_to_file, 49
Apply-to-each, 26
asin, 38
Assignment, 28
atan, 38
Booleans (bool), 22
bottop, 17, 45
btoi, 38
ceil, 39
collect, 46
Complexity, 13
Convex Hull, 19
cosh, 38
cos, 38
count, 42
datatype, 24, 32
Depth Complexity, 13
diff, 37
display, 52
dist, 40
drop, 17, 43
e lt, 40
empty sequences, 24
eq, 47
even_elts, 17, 44
evenp, 36
exp_string, 47
expt, 38
exp, 38
find, 46
flatten, 44
Floats (float), 23
float, 39
floor, 39
function, 28, 32
get_environment_variable, 55
Global variables, 28
hash, 47
head_rest, 45
identity, 47
if, 33
int_collect, 46
Integer ranges, 16, 33
Integers (int), 22
interleave, 44
intersection, 46
iseq, 41
isqrt, 37
kth_smallest, 7, 46
length_from_flags, 44
let, 25, 33
linify, 48
ln, 37
Local bindings, 25
logical, 13
log, 37
lowercase, 48
lshift, 37
mark_duplicates, 46
max_index, 42
max_int, 39
max_scan, 41
max_val, 41
max, 37
60
w_draw_points, 53
w_draw_point, 53
w_draw_rectangle, 54
w_draw_segments, 54
w_draw_string, 54
w_get_button_input, 55
w_get_input_noblock, 55
w_get_input, 54
w_get_named_box, 53
w_get_zoom_box, 55
w_kill_window, 52
w_make_window, 52
w_reset_box_size, 53
w_shade_polygon, 54
w_shade_rectangle, 54
w_write_paragraph, 54
w_write_text_centered, 54
w_write_text_left, 54
wordify, 48
Work Complexity, 13
write_char, 50
write_check, 51
write_object_to_file, 49
write_string_to_file, 49
write_string, 50
xor, 35
zerop, 36
zip, 40