NESL: A Nested Data-Parallel Language

Guy E. Blelloch

January 1992

CMU-CS-92-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Keywords: Data-parallel, parallel algorithms, supercomputers, nested parallelism, parallel RAM model

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 supercomputers, and as a basis for teaching parallel algorithms. Parallelism is supplied through a simple set of data-parallel constructs based on vectors, including a mechanism for applying any function over the elements of a vector in parallel, and a broad set of parallel functions that manipulate vectors.

NESL fully supports nested vectors and nested parallelism—the ability to take a parallel function and then apply it over multiple instances in parallel. Nested parallelism is important for implementing algorithms with complex and dynamically changing data structures, such as required in many graph or sparse matrix algorithms. NESL also provides a mechanism for calculating the asymptotic running time for a program on various parallel machine models, including the parallel random access machine (PRAM). This is useful for approximating running times of algorithms on actual machines, and when teaching algorithms to supply 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 the Cray Y-MP, Connection Machine CM-2, and Encore Multimax. For many algorithms, the current implementation gives performance close to optimized machine-specific code for these machines. NESL is the first data-parallel language whose implementation fully supports nested parallelism.

Contents

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 vectors. These constructs supply parallelism through (1) the ability to apply any function concurrently over each element of a vector, and (2) a set of parallel functions that operate on vectors, such as the `permute` function, which permutes the order of the elements in a vector.

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

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 VCODE [5], which runs on the CRAY Y-MP, the Connection Machine CM-2, and the Encore Multimax. Various benchmarks algorithms achieve very good running times on these machines [10, 4].

4. To be well suited for describing *parallel algorithms*, and to supply a mechanism to derive the theoretical running time directly from the code. Each function in NESL has two complexity measures associated with it, the work and step complexities. These can be composed with some simple rules to derive the complexity for any computation. A simple equation maps these complexities to the running time on a PRAM.

NESL is an interactive, strongly typed, applicative language with no side effects and a Lisp-like syntax.[1] The language is based on vectors (sequences) as a primitive parallel data type, and parallelism is achieved exclusively through operations on these vectors [3]. The set of vector 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), and supplies no data-structures that require serial access, such as lists in Lisp or ML.

NESL is the first data-parallel language whose implementation supports nested parallelism. Nested parallelism is the ability to take a parallel function and then apply it over multiple instances in parallel—for example, having a parallel sorting routine, and then using it to sort several sequences concurrently. The languages C* [20], *Lisp [17], and Fortran 90 [1] (the 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

[1] It should not, however, be confused with Lisp or Scheme. In fact, we are currently working on a new version of the language in which the syntax will be much closer to the syntax of the ML language [18].

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 [24], and Paralation Lisp [21] both supply nested parallel constructs, but no implementation ever supported nested parallelism. Blelloch and Sabot implemented an experimental compiler that supported nested-parallelism for a small subset of Paralation Lisp [7], but it was deemed near impossible to extend it to the full language.

Existing data-parallel languages almost all consist of a set of data-parallel constructs added to a serial language (C and C*, Pascal and Parallel Pascal, Fortran 90 and its vector extensions, Lisp and the three languages *Lisp [17], CM-Lisp [24] and Paralation-Lisp [21]). This approach of adding extensions to an existing language has its clear advantages—it allows the creation of a parallel language with only a small effort in language design. The approach, however, can also have some severe disadvantages. In particular data-parallel constructs often do not fit well with the existing serial languages, causing awkward semantic and compiler problems. The interaction of the parallel constructs with the full features of the language can also force severe limitations to be placed on the use of the constructs. For these reasons, NESL is quite intentionally not built on top of any existing language, although it clearly uses many ideas from existing languages. The author believes that there is no possibility that NESL could fully support nested parallelism if it was built on top of any of the standard languages.

A disadvantage of starting from the bottom is that NESL does not subsume all the features of a full language. Because of this NESL currently does not support passing functions as arguments, its typing system is somewhat ad-hoc, and it only supports minimum input and output capabilities. We are currently working on a follow up on NESL, which will include some of the features that are lacking, and will be based on a more rigorous type system.

A common complaint about high-level data-parallel languages, and more generally in the class of Collection-Oriented languages [23], such as SETL [22] and APL [16], is that it can be hard or impossible to determine approximate running times for a computation by looking at the code. As an example, the β primitive in CM-Lisp (a general communication primitive) is sufficiently powerful, 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 effected by how the set is represented. This representation is chosen by the compiler.

For this reason, NESL was designed so that the asymptotic complexity can always be derived from the code as a function of the length of the vectors used in the code. In particular, each function in NESL has two complexities associated with it, and simple composition rules are supplied to combine complexities across expressions. From these rules, both the total work executed by the program (the running time if executed on a serial RAM), and the parallel depth of the computation (the running time if executed with an unbounded number of processors) can be calculated. These complexities can then be combined to determine asymptotic running times on a parallel random access machine [11] (PRAM) for any fixed number of processors. Since the complexities do not include constants, and real machines do not fit the PRAM model, they can only be used to give approximate running times on any real machines, but are very useful for analyzing asymptotic performance of parallel algorithms.

The remainder of this section discusses the use of vectors and nested parallelism in NESL, and how complexity can be derived from NESL code. Section 2 shows several examples of code, Section 3 defines the language, and Section 4 lists all the available functions.

1.1 Parallel Operations on Vectors

NESL supports parallelism through operations on vectors. This parallelism can be achieved in two ways. First, any function can be applied over the elements of a vector. For example, the expression

```
(over ((a #v(7 -2 5 4)))
  (negate a))
   ⇒  #v(-7 2 -5 -4) : v.int
```

negates each elements of the vector `#v(7 -2 5 4)`. This construct can be read as "for each element `a` in the vector `#v(7 -2 5 4)`, negate `a`" (in the example, the symbol ⇒ points to what is returned by the expression above it, and the `v.int` indicates that the result of the expression is of type *vector of integers*). The `over` form can also be expressed shorthand as,

```
(v.negate #v(7 -2 5 4))
   ⇒ #v(-7 2 -5 -4) : v.int
```

This shorthand `v.` form is considered syntactic sugar for the `over` form (see Section 3.2 for a more precise definition).

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

```
(defop (factorial i) (int <- int)
  (if (= i 1)
      1
      (* i (factorial (- i 1)))))
   ⇒ factorial : int <- int
```

and then apply it over the elements of a vector

```
(v.factorial #v(3 1 7))
   ⇒ #v(6 1 5040) : v.int
```

In this example, the `(defop (`*name arguments*`)` *type* *body*`)` form is used to define `factorial`. The *type* argument specifies the type of the function, which in this case is specified as `(int <- int)`, a function that maps integers to integers.

An `over` form applies a body to each element of a vector. We will call each such application an *instance*. Since there are no side effects in NESL, there is no way to communicate

	Operation	Description	Work
*	`dist a l`	*Distribute value* `a` *to vector of length* `l`.	S(`result`)
*	`length a`	*Return length of vector* `a`.	`1`
	`elt a i`	*Return element at position* `i` *of* `a`.	S(`result`)
	`rep v a i`	*Replace element at position* `i` *of* `a` *with* `v`.	`1`, S(`a`)
	`index l`	*Generate an index vector of length* `l`.	`l`
	`iseq s d e`	*Return integer sequence from* `s` *to* `e` *by* `d`.	`(e - s)/d`
	$\oplus$`-reduce a`	*Return sum based on operator* $\oplus$.	S(`a`)
*	$\oplus$`-scan a`	*Return scan based on operator* $\oplus$.	S(`a`)
	`count a`	*Count number of true flags in* `a`.	S(`a`)
	`permute a i`	*Permute elements of* `a` *to positions* `i`.	S(`a`)
	`put a i d`	*Place elements* `a` *in* `d` *based on indices* `i`.	S(`a`), S(`d`)
	`const-put a i d`	*Place element* `a` *in* `d` *based on indices* `i`.	S(`a`), S(`d`)
*	`cond-put a i f d`	*Conditional* `put` *based on masks* `f`.	S(`a`), S(`d`)
*	`get a i`	*Get values from vector* `a` *based on indices* `i`.	S(`i`)
	`pack a f`	*Pack vector* `a` *based on flags* `f`.	S(`a`)
	`pack-index f`	*Packed indices of true positions in vector* `f`.	S(`f`)
	`max-index a`	*Return index of the maximum value.*	S(`a`)
	`min-index a`	*Return index of the minimum value.*	S(`a`)
	`append a b`	*Append vectors* `a` *and* `b`.	S(`a`) + S(`b`)
	`cons a b`	*Append element* `a` *to front of vector* `b`.	S(`a`) + S(`b`)
	`vtup a b`	*Append elements* `a` *and* `b` *into a vector.*	S(`a`) + S(`b`)
	`vsep a`	*Convert two-element vector* `a` *into a tuple.*	S(`a`)
	`drop n a`	*Drop first* `n` *elements of vector* `a`.	S(`a`)
	`take n a`	*Take first* `n` *elements of vector* `a`.	S(`a`)
	`rotate n a`	*Rotate vector* `a` *by* `n` *positions.*	S(`a`)
	`subseq a s e`	*Subsequence of* `a` *from indices* `s` *to* `e`.	S(`a`)
*	`flatten a`	*Flatten nested vector* `a`.	S(`a`)
*	`partition a l`	*Partition vector* `a` *into nested vector.*	S(`a`)
	`split a f`	*Split* `a` *into nested vector based on flags* `f`.	S(`a`)
	`bottop a`	*Split* `a` *into nested vector.*	S(`a`)

Figure 1: List of the vector functions of 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 overhead in both work and number of steps. The $\oplus$ for `reduce` and `scan` can be one of {+, `max`, `min`, `or`, `and`}. All the vector functions are described in detail in Section 4.2.

among the instances of an `over` form. 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 `over` its parallel semantics.

In addition to the `over` form, a second way to take advantage of parallelism in NESL is through a set of vector functions. The vector functions operate on whole vectors and all have relatively simple parallel implementations. For example the function `+-reduce` sums the elements of a vector.

```
(+-reduce #v(2 1 -3 11 5))
   ⇒ 16 : int
```

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

```
(permute "road" #v(2 1 3 0))
   ⇒  "dora" : v.char
```

In this case, the 4 characters of the string `"road"` (the term *string* is used to refer to a vector of characters) are being permuted to the indices `#v(2 1 3 0)` (`r` $\rightarrow 2$, `o` $\rightarrow 1$, `a` $\rightarrow 3$, and `d` $\rightarrow 0$). The implementation of the `permute` function on a distributed-memory parallel machine could use its communication network, and the implementation on a shared-memory machine could use an indirect write into the memory.

Table 1 lists the vector functions available in NESL. A subset of the functions (the stared ones) form a complete set of *primitives*. These primitives, along with the scalar operations and the `over` form, are sufficient for implementing the other functions in the table with at most a constant increase in both the step and work complexities. The table also lists the work complexity of each function. This complexity is typically based on the size of the function's arguments. The size of an object should not be confused with the length of a vector. The size is defined inductively: the size of a scalar value is 1, and the size of a vector is the sum of the sizes of its elements plus 1. The size of a nested vector could therefore be much greater than its length. The step complexity of each function in the table is $O(1)$.

We now consider an example of the use of vectors in NESL. The algorithm we consider solves the problem of finding the $\mathtt{k}^{th}$ smallest element is a set `s`, using a parallel version of the quickorder algorithm [14]. Quickorder is similar to quicksort, but only calls itself recursively on either the elements lesser or greater than the pivot. The NESL code for the algorithm is shown in Figure 2. The `with` form is used to bind local variables and is like a `let*` in Common Lisp. The code first binds `l` to the length of the input vector `s`, and then extracts the element half-way across the vector `s` as a pivot; the `(elt s i)` function extracts the $\mathtt{i}^{th}$ element from a vector `s`. The algorithm then selects all the elements less than the pivot, and places them in a vector that is bound to `lesser`. This is done by comparing each element of `s` to the pivot, and packing the elements with a true flag. The `v.pivot` means that the pivot is extended to the same length as `s`. The `pack` function, which takes a vector of elements and a vector of flags of equal length, packs the elements where the flag is true into a smaller vector. For example,

```
(defop (order s k) (int <- v.int int)
  (with ((l      (length s))
         (pivot  (elt s (/ l 2)))
         (lesser (pack s (v.< s v.pivot))))
    (if (< k (length lesser))
        (order lesser k)
      (with ((greater (pack s (v.> s v.pivot))))
        (if (>= k (- l (length greater)))
            (order greater (- k (- l (length greater))))
         pivot)))))
```

Figure 2: An implementation of order statistics. The function `order` returns the `k`th smallest element from the input vector `s`.

```
s                              =  #v(4 8 2 3 1 7 2)
pivot                          =  3
(v.< s v.pivot)                =  #v(f f t f t f t)
(pack s (v.< s v.pivot))       =  #v(2 1 2)
```

After the pack, if the number of elements in the set `lesser` is greater than `k`, then the $\mathtt{k}^{th}$ smallest element must belong to that set. In this case, the algorithm calls `order` recursively on `lesser` using the same `k`. Otherwise, the algorithm selects the elements that are greater than the pivot, again using `pack`, and can similarly find if the $\mathtt{k}^{th}$ element belongs in the set `greater`. If it does belong in `greater`, the algorithm calls itself recursively, but must now readjust `k` by subtracting off the number of elements lesser and equal to the pivot. If the $\mathtt{k}^{th}$ element belongs in neither `lesser` nor `greater`, then it must be the pivot, and the algorithm returns this value.

1.2 Nested Parallelism

Nested parallelism is the ability to apply a parallel function multiple times in parallel. For example, we could apply the parallel vector function `+-reduce` within the `over` form:

```
(over ((v #v(#v(2 1) #v(7 3 0) #v(4))))
  (+-reduce v))
   ⇒ #v(3 10 4) : v.int
```

In this expression there is parallelism both within each `+-reduce`, since the vector function has a parallel implementation, and across the three instances of `+-reduce`, since `over` is defined such that all instances can run in parallel.

Table 1 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 subparts can be executed in parallel,

Application	Outer Parallelism	Inner Parallelism
Sum of Neighbors in Graph	For each vertex of graph	Sum neighbors of vertex
Figure Drawing	For each line of image	Draw pixels of line
Compiling	For each procedure of program	Compile code of procedure
Text Formatting	For each paragraph of document	Justify lines of paragraph

Table 1: Routines with nested parallelism. Both the inner part and the outer part can be executed in parallel.

Algorithm	Outer Parallelism	Inner Parallelism
Quicksort	For lesser and greater elements	Quicksort
Mergesort	For first and second half	Mergesort
Closest Pair	For each half of space	Closest Pair
Strassen's Matrix Multiply	For each of the 7 sub multiplications	Strassen's Matrix Multiply
Fast Fourier Transform	For two sets of interleaved points	Fast Fourier Transform

Table 2: Some divide and conquer algorithms.

```
(defop (qsort a) (v.int <- v.int)
  (if (< (length a) 2) a
    (with ((pivot   (elt a (/ (length a) 2)))
           (lesser  (pack a (v.< a v.pivot)))
           (equal   (pack a (v.= a v.pivot)))
           (greater (pack a (v.> a v.pivot)))
           ((rl rg) (vsep (v.qsort (vtup lesser greater)))))
      (append rl (append equal rg)))))
```

Figure 3: An implementation of quicksort.

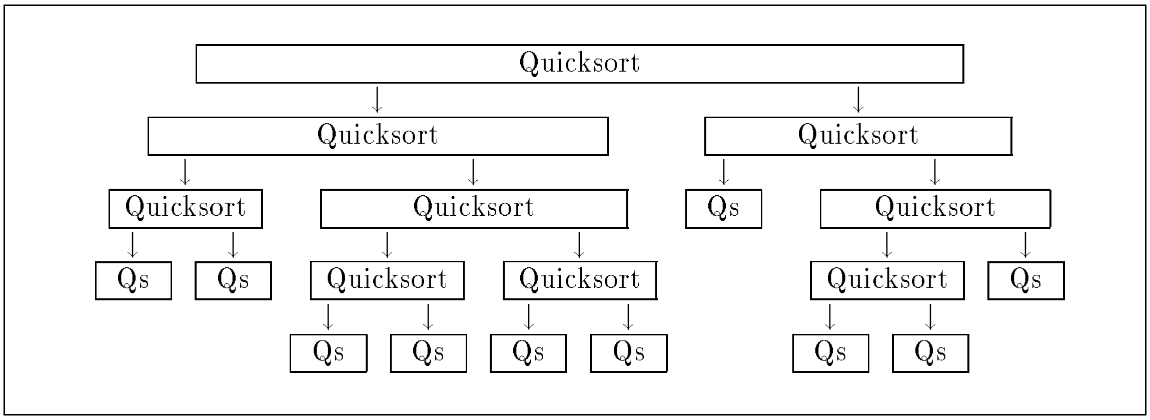


Figure 4: The quicksort algorithm. Just using parallelism within each block yields a parallel running time at least as great as the number of blocks ($O(n)$). Just using parallelism from running the blocks in parallel yields a parallel running time at least as great as the largest block ($O(n)$). By using both forms of parallelism the parallel running time can be reduced to the depth of the tree (expected $O(\lg n)$).

as is usually the case, the application of the subparts involves nested parallelism. Table 2 shows several examples.

As an example, consider a parallel variation of quicksort [2] (see Figure 3). When applied to a vector `s`, 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. The selecting of each set is done using a `pack`, as used in the `order` function discussed in the last section. To execute the two recursive calls, the `vtup` function is used to concatenate two vectors into a nested vector. For example:

```
(vtup #v(5 2 4) #v(8 11 9 16))
   ⇒  #v(#v(5 2 4) #v(8 11 9 16)) :  v.v.int
```

The `v.qsort` then applies `qsort` in parallel over the two elements of the nested vector (the lesser and greater elements). When the `qsort` returns, the `vsep` function separates the resulting nested vector into a tuple (a pair), whose elements are then bound to `rl` (the sorted lesser elements) and `rg` (the sorted greater elements). The final line appends `rl`, `equal`, and `rg` together into sorted order.

The recursive invocation of `qsort` generates a tree of calls which would look something like the tree shown in Figure 4. The step complexity of the algorithm is simply the depth of this tree (expected $O(\lg n)$) and the work is the size of the tree (expected $O(n \lg n)$). If we were to only 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. Inversely, if we were to only 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. In both cases the parallel time complexity is $O(n)$ rather than the ideal $O(\lg n)$ (the expected depth of the tree).

Without nested parallelism, there is no direct way of expressing both kinds of parallelism used in quicksort.

1.3 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 expression would take if executed on a serial random access machine. The work complexity for most of the vector functions is simply the size of the vector being operated on. A complete list is given in Table 1.

2. **Step complexity:** this represents the parallel depth of the computation, that is to say, the amount of time the program would take on a machine with an unbounded number of processors. The step complexity of all the vector functions supplied by NESL is one.

These two complexities are based on the vector random access machine (VRAM) model [3], a strictly data-parallel abstraction of the parallel random access machine (PRAM) model [11]. Since the complexities are meant for determining asymptotic complexity, these complexities do not include constant factors. For all the NESL functions, however, the constants are small.

The complexities are combined using two simple combining rules. Expressions are combined in the standard way—for both the work complexity and the step 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:

```
(* (+-reduce (dist 7 n)) (length a))
```

can be calculated as:

	Work	Step
`dist`	$n+1$	1
`+-reduce`	$n+1$	1
`length`	1	1
`*`	1	1
Total	$O(n)$	$O(1)$

The `over` form is combined in the following way. The work complexity is the sum of the work complexity of the instantiations, and the step complexity is the maximum over the step complexities of the instantiations. For example, the complexities of the computation:

```
(over ((i (index n)))
  (+-reduce (index i))
```

can be calculated as:

	Work	Step
`index`	$n+1$	1
Within Over		
`index`	$\sum_{i=0}^{i<n} i+1$	$\max_{i=0}^{i<n} 1$
`+-reduce`	$\sum_{i=0}^{i<n} i+1$	$\max_{i=0}^{i<n} 1$
Total	$O(n^2)$	$O(1)$

Once the step (S) and work (W) complexities have been calculated, which are the complexities for a VRAM, a simple formula gives the running time on a PRAM:

$$t \quad = \quad O(W/P + S)$$

This formula can be derived from Brent's scheduling principle [8] as shown in [3].

One can clearly argue that the PRAM model is not a particularly practical model of parallel computation since it assumes that references to a shared memory take unit-time. The model, however, gives a balpark prediction of the running times, and will never grossly underestimate the time. The complexity also makes NESL well suited for teaching parallel algorithms since the programming model very closely matches the description of the algorithm used for determening the complexity.

2 Examples

This section illustrates several examples of NESL programs.

2.1 String Searching

The first example is a function that finds the occurrences of a word in a string (a vector of characters). In particular, the function `(string-search w s)` (see Figure 5), takes a search word `w` and a string `s`, and returns the starting position of all words in `s` that match `w`. For example,

```
(string-search "foo" "fobarfoofboofoo")
   ⇒ #v(5 12) : v.int
```

The algorithm works in a number of parallel steps proportional to the length of the search string `w`. It starts at the first character of `w` and generates a set of candidate matches. The algorithm then progresses through the search string, using recursive calls to `next-cands`, narrowing the set of candidate matches on each step. The candidates are stored as pointers (indices) into `s` of the beginning of each match. To determine the initial set of candidates, `string-search` compares every element of the string `s` with the first character of `w`, and uses the `pack-index` function to return the indices where there is a match. The `pack-index` function takes a vector of booleans and returns the indices where a flag is true. For example,

```
(pack-index #v(f t f f t t))
   ⇒ #v(1 4 5) : v.int
```

```
(defop (next-cands cands w s i) (v.int <- v.int v.char v.char int)
  (if (= i (length w))
      cands
    (with ((letter (elt w i))
           (match-flags (v.= v.letter (get s (v.+ cands v.i)))))
      (next-cands (pack cands match-flags) w s (+ i 1)))))

(defop (string-search w s) (v.int <- v.char v.char)
  (next-cands (pack-index (v.= s v.(elt w 0)))
     w s 0))
```

Figure 5: Finding all occurrences of the word `w` in the string `s`.

the first, fourth and fifth flags are true.

Based on the current candidates, `next-cands` narrows the set of candidates by only keeping the candidates that match on the next character of `w`. To do this, each candidate checks whether the $\texttt{i}^{th}$ character in `w` matches the $\texttt{i}^{th}$ position past the candidate index. All candidates that do match are packed and passed into the recursive call of `next-cands`. The recursion completes when the algorithm reaches the end of `w`. The progression of `cands` in the "`foo`" example would be:

i	cands
0	#v(0 5 8 12)
1	#v(0 5 12)
2	#v(5 12)

If the word `w` has length l, the number of steps taken by the algorithm is some constant times the number of recursive calls, which is simply $O(l)$. The work complexity of the algorithm is the sum over the calls of the number of candidates in each step. In practice, this is usually linear in the length of the search string `s` ($l(\texttt{s})$), but in the worst case this can be the product of the two lengths $l(\texttt{s})l(\texttt{w})$ (the worst case can only happen if most of the characters in `w` are repeated). Algorithms for string searching are known [9, 25], that give better bounds on the parallel time (step complexity), and that bound the worst case work complexity to be linear in the length of the search string, but 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 multiple 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

```
(defop (primes n) (v.int <- int)
  (if (= n 1)
      #v.int()
    (with ((sqr-primes (primes (isqrt n)))
           (sieves     (over ((p sqr-primes))
                         (iseq (* 2 p) p n)))
           (flat-sieve (flatten sieves))
           (flags      (const-put f flat-sieve (dist t n))))
      (drop 2 (pack-index flags)))))
```

Figure 6: Finding all the primes less than n.

well designed version should require no more work than the serial sieve ($O(n \lg\lg n)$), and polylogarithmic parallel time.

The version we use (see Figure 6) requires $O(n \lg\lg n)$ work and $O(\lg\lg n)$ steps. 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 primes multiples of `p` up to `n` (`sieves`). This is done with the `(iseq s d e)` function, which returns the integers starting at `s`, increasing by `d`, and up to, but less than, `e`. For example,

```
(iseq 4 2 20)
   ⇒ #v(4 6 8 10 12 14 16 18) : v.int
```

The vector `sieves` is therefore a nested vector with each subvector being the sieve for one of the primes in `sqr-primes`. The function `flatten`, is now used to flatten this nested vector by one level, therefore returning a vector containing all the sieves. For example,

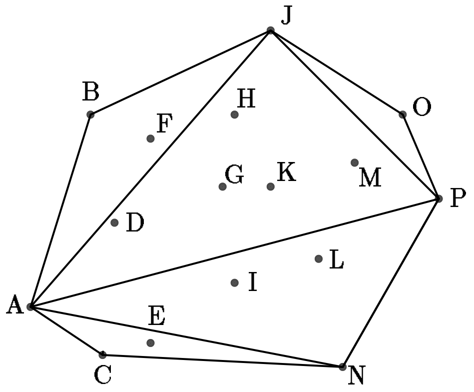
```
(flatten #v(#v(4 6 8 10 12 14 16 18) #v(6 9 12 15 18)))
   ⇒  #v(4 6 8 10 12 14 16 18 6 9 12 15 18) : v.int
```

This vector of sieves is now used by the `const-put` function to place a false flag in all positions that belong to a sieve. The `(const-put c i d)` function places a constant `c` (in this case, the false flag) into a vector `d` (in this case a constant vector of true flags), at positions `i` (in this case, the sieve positions). Finally the `pack-index` function returns a vector of indices where the flags remain true (elements which were not sieved), and `drop`, removes the first two elements (0 and 1), which are not considered primes.

The functions `iseq, flatten, const-put, pack-index` and `drop` all require a constant number of steps. Since `primes` is called recursively on a problem of size $\sqrt{n}$ the total number of steps require by the algorithm can be written as the recurrence:

$$S(n) \quad = \quad \begin{cases} O(1) & n = 1 \\ S(\sqrt{n}) + O(1) & n > 1 \end{cases} \quad = \quad O(\lg\lg n)$$

Almost all the work done by `primes` is done at the top level. At this top level, the work is proportional to the length of the vector `flat-sieve`. Using the standard formula



[A B C D E F G H I J K L M N O P]

A [B D F G H J K M O] P [C E I L N]

A [B F] J [O] P N [C E]

A B J O P N C

Figure 7: An example of the *quickhull* algorithm. Each vector 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.

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

where p are the primes [13], the length of this vector is:

$$\begin{aligned} \sum_{p \leq \sqrt{n}} n/p &= O(n \log\log \sqrt{n}) \\ &= O(n \log\log n) \end{aligned}$$

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

2.3 Planar Convex-Hull using Quickhull

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 [12] to statistics [15]. The algorithm we use to solve the problem is a parallel version [6] of the *quickhull* algorithm [19]. The quickhull algorithm was given its name

```
(defrec point (x int) (y int))

(defop (cross-product o p1 p2) (int <- point point point)
  (- (* (- (x p1) (x o)) (- (y p2) (y o)))
     (* (- (y p1) (y o)) (- (x p2) (x o)))))

(defop (hsplit points p1 p2) (v.point <- v.point point point)
  (if (< (length points) 2)
      points
    (with ((cross   (v.cross-product points v.p1 v.p2))
           (packed  (pack points (v.plusp cross)))
           (pm      (elt points (max-index cross)))
           ((r1 r2) (vsep (v.hsplit (vtup packed packed)
                                    (vtup p1     pm)
                                    (vtup pm     p2)))))
      (append r1 (cons pm r2)))))

(defop (convex-hull points) (v.point <- v.point)
  (with ((min-x (elt points (min-index (v.x points))))
         (max-x (elt points (max-index (v.x points)))))
    (append (cons min-x (hsplit points min-x max-x))
            (cons max-x (hsplit points max-x min-x)))))
```

Figure 8: Code for Quickhull. The `defrec` form defines a record with two slots, an `x` slot and a `y` slot, which are both integers. The commands `(x p)` and `(y p)` are used to access the two slots from the record `p`.

because of its similarity to the quicksort algorithm. As with quicksort, the algorithm picks 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 equal sized sets, and in the worst case the algorithm will require $O(n^2)$ work.

Figure 7 shows an example of the quickhull algorithm, and Figure 8 shows the code. The algorithm is based on the recursive routine `hsplit`. This function takes a set of points in the plane ($\langle x, y \rangle$ coordinates) and two points `p1` and `p2` that are known to lie on the convex hull, and returns all the points lie on the hull clockwise from `p1` to `p2`. In the figure, given all the points `#v(A B C ... P)`, `p1 = A` and `p2 = P`, `hsplit` would return the vector `#v(B J O)`. In `hsplit`, the order of `p1` and `p2` matters, since if we switch `A` and `P`, `hsplit` would return the hull along the other direction `#v(N C)`

The `hsplit` function works by first removing all the elements that cannot be on the hull since they lie below or at the line between `p1` and `p2`. This is done by removing elements whose cross product are negative. In the case `p1 = A` and `p2 = P`, the points `#v(B D F G H J K M O)` would remain and be placed in the vector `packed`. The algorithm now finds the point furthest from the line `p1-p2`. This point `pm` must be on the hull since as a line parallel to `p1-p2` moves toward `p1-p2`, it must first hit `pm`. The point `pm` (`J` in the running example) is found by taking the point with the maximum cross-product. Once `pm` is found, `hsplit` calls itself twice recursively using the points `(p1, pm)` and `(pm, p2)` (`(A, J)` and `(J, P)` in the example). When the recursive calls return, `hsplit` appends `r1` (the hull between `p1` and `pm`), the point `pm`, and `r2` (the hull between `pm` and `p2`). This gives the desired result.

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 step 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 $O(\lg m)$ steps for well-distributed hull points, and has a worst case running time of $O(m)$ steps.

3 Language Definition

NESL is a strongly typed language with the following data types:

- four primitive atomic data types, booleans (`bool`), integers (`int`), characters (`char`), and reals (`real`);
- two primitive compound types, vectors and tuples;
- and a user type constructor, `defrec`, for defining new types.

And the following operations:

- a set of predefined functions on the primitive types;
- three primitive forms, a conditional `if`, a binding form `with`, and an iterator `over`;
- and a function constructor, `defop`, for defining new functions.

With the property that data is not mutable and functions cannot have side effects. This section covers each of these topics. It is not meant as a formal semantics but along with the list of functions in the next section it should serve as an adequate definition of the language.

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 (ex. `not, and, or, xor, nor, nand`) are predefined and all require a fixed number of arguments (this is unlike lisp where the `and` function can take any number of arguments). Some examples:

```
(not (not t))
   ⇒ t : bool

(xor t 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 worth. The standard functions on integers (`+, -, *, div, =, <, negate, ...`) are predefined. Some examples:

```
(* 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 (ex. `=, <, >=, ..`) can be used on. Characters are written by placing a `#\` in front of the character. For example:

```
#\8
   ⇒ #\8 : char

(= #\a #\d)
   ⇒ F : bool

(< #\a #\d)
   ⇒ T : bool
```

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 there is a choice. Floats support most of the same functions as integers, and also have several additional functions (ex. `round, truncate, sqrt, log, ..`). Floats must be written by placing a decimal point in them so that they can be distinguished from integers.

```
(* 1.2 3.0)
   ⇒ 3.6 : float

(round 2.1)
   ⇒ 2 : int
```

A complete list of the functions available on scalar types can be found in Section 4.1.

3.1.2 Vectors

A vector is a sequence of values. A vector can contain any type, including other vectors, but each element in a vector must be of the same type (vectors are homogeneous). The type of a vector is specified by placing a `v.` in front of the element type. Some examples:

```
#v(6 2 4 5)
   ⇒ #v(6 2 4 5) : v.int
#v(#v(2 1 7 3) #v(6 2) #v(22 9))
   ⇒  #v(#v(2 1 7 3) #v(6 2) #v(22 9)) : v.v.int
```

Vectors of characters can be written between double quotes,

```
"a string"
   ⇒ "a string" : v.char
```

but can also be written as a vector of characters:

```
#v(#\a #\Space #\s #\t #\r #\i #\n #\g)
   ⇒ "a string" : v.char
```

NESL supplies many functions that operate on vectors. For example:

```
(append #v(1 2 3) #v(4 5))
   ⇒ #v(1 2 3 4 5) : v.int
(extract #v(3 7 1 9 2) 3)
   ⇒ 9 : int
(index 4)
   ⇒ #v(0 1 2 3) : int
(length "a string")
   ⇒ 8 : int
```

See Section 4.2 for a description of all the available vector functions.

3.1.3 Tuples

A tuple is a pair of elements. Tuples can be created with the `tup` function, and their elements can be extracted with the `first` and `second` functions, or with patter matching in a `with` form. Unlike vectors, the two elements can be of different types, and a tuple with element types α and β has type (α β). For example:

```
(tup 3 t)
   ⇒ (tup 3 t) : (int bool)
#v((tup 3 t) (tup 7 f) (tup 11 t))
   ⇒  #v((tup 3 t) (tup 7 f) (tup 11 t)) : v.(int bool)
(first (tup 3 t))
   ⇒ 3 : int
(second (first (tup (tup 3 "horse") t)))
   ⇒ "horse" : v.char
```

Tuples are useful for returning a pair of values from a function. Because of the typing scheme of NESL, it is not possible to use tuples to build lists, such as CONS is used in Lisp.

3.1.4 Defining New Types

Record types with a fixed number of slots can be defined with the `defrec` operation. For example:

```
(defrec complex (float int) (imag int))
   ⇒ complex : (int int)
```

defines a record with two slots, `float` and `imag`, both which must contain an integer. Once a record type is define, a record can be created using its name.

```
(complex 7 11)
   ⇒ (complex 7 11) : complex
```

An element of a record can be extracted by using the slot name.

```
(float (complex 7 11))
   ⇒ 7 : int
```

3.2 Functions

3.2.1 Forms

NESL has three primitive *forms*: `if`, `with` and `over`. These forms are different from functions in that they either have a different syntax from functions or they do not always evaluate all their arguments, which functions do. In NESL users can define new functions, but not new forms.

If: The Conditional Form

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

(`if` *cond-exp then-exp else-exp*)

If the *cond-expression* is true, then the *then-expression* is evaluated and its result is returned, otherwise the *else-expression* is evaluated and its result is returned. The *cond-expression* must be of type `bool`, and the other two expressions must be of matching types. Here is an example:

```
(if (and t f) (+ 3 4) (* (- 6 2) 7))
   ⇒ 28 : int
```

On the other hand `(if x 3 2.6)` is not a valid expression since the two branches return different types.

`With`: The Binding Form

Local variables can be allocated with the `with` form. the syntax is:

(`with` ((*variable-name value-exp*)`*`)
  body-exp)

The `*` signifies that the name value pairs can be repeated any number of times. A *variable-name* appearing in an earlier pair can be used in the *value-exp* of a later pair (this is similar to `let*` in Common Lisp). The *body-exp* can refer to any of the *variable-names* and its result is the result of the *body-exp*. For example, in

```
(with ((a 7)
       (b (+ a 4)))
  (* a b))
   ⇒ 77 : int
```

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

`Over`: The Apply to Each Form

As well as functions on vectors you can apply any function over the elements of a vector. The `over` form is used to do this.

It has the following syntax:

(`over` ((*variable-name value-exp*)`+`)
  body-exp)

The `+` signifies that the name value pairs can be repeated any number of times, but at least once. This syntax is similar to the syntax of the `with` form, but in an `over` each of the `value-exps` must evaluate to equal length vectors. The *variable-name* is then bound to each element of the vectors, and the *body-exp* is applied for each of these bindings. For example:

```
(over ((a #v(1 2 3))
       (b #v(1 4 9)))
  (+ a b))
   ⇒ #v(2 6 12) : v.int

(over ((a #v(1 2 3))
       (b #v(1 4 9)))
  (dist b a))
   ⇒ #v(#v(1) #v(4 4) #v(9 9 9)) : v.v.int
```

There is also a shorthand way of applying an `over` by placing a `v.` in front of the function name. For example:

```
(v.+ #v(1 2 3) #v(1 4 9))
   ⇒ #v(2 6 12) : v.int
```

```
(v.+_reduce #v(#v(7 3) #v(6 2 8 -5)))
   ⇒ #v(10 11) : v.int
```

The first example is equivalent to the first example of the `over` form.

Placing a `v.` in front of a function is called *vector-extending* the function. It is also possible to vector-extend an argument. For example:

```
(v.+ #v(1 2 3) v.3)
   ⇒ #v(4 5 6) : v.int
```

Which is semantically equivalent to:

```
(over ((a #v(1 2 3)))
  (+ a 3))
   ⇒ #v(4 5 6) : v.int
```

3.2.2 Defining New Functions

Functions can be defined using the `defop` form. The syntax is:

(defop (*function-name variable**) (*result-type* <- *source-type**)
  body-exp)

A function can have any number of variables. Its type list specifier contains a *result-type* and a list of *source-types*. The number of source-types must be equal to the number of variables and specifies the type of each of these input variables. The *body-exp* can only refer to variables in the `variable` list. Consider the definitions:

```
(defop (square A) (int <- int)
  (* A A))
   ⇒ square : (int <- int)
```

This defines a function that squares an integer by multiplying it by itself.

3.2.3 Typing

NESL is strongly typed, monomorphic language with overloading on the built-in functions. All user defined functions must be defined on a single type, but many of the functions supplied by NESL are overloaded. For example, `(+ a b)` can be used on either integers or floating point numbers, although both numbers must be of the same type. As a more interesting example, the function `(append a b)`, can be applied to two vectors of any type, as long as they are the same. The applications:

```
(append "dog" "mouse")

(append #v(2 1 5) #v(3 8))

(append #v(#v(7 8)) #v(#v(1 2 6) #v(9 7)))
```

are all valid. The types permitted by each function are specified in the list of functions given in the next section. The typing system in NESL is ad-hoc and will be improved in the next version of the language.

4 List of Functions

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

`(function-name arguments)` $\{$*result-type* $\leftarrow$ *source-types* : *type-bindings*$\}$

Definition of function.

In the type specifications, the following assignments are valid:

```
ordinal = bool, int, char
number  = int, float
logical = bool, int
any     = any valid type
```

4.1 Scalar Functions

4.1.1 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.

`(not a)` $\{alpha \leftarrow alpha : \ alpha \text{ in } logical\}$

Returns the logical inverse of the argument. For integers, this is the ones complement.

`(or a b)` $\{alpha \leftarrow alpha\ alpha : \ alpha \text{ in } logical\}$

Returns the inclusive or of the two arguments.

`(and a b)` $\{alpha \leftarrow alpha\ alpha : \ alpha \text{ in } logical\}$

Returns the logical and of the two arguments.

`(xor a b)` $\{alpha \leftarrow alpha\ alpha : \ alpha \text{ in } logical\}$

Returns the exclusive or of the two arguments.

`(nor a b)` $\{alpha \leftarrow alpha\ alpha : \ alpha \text{ in } logical\}$

Returns the inverse of the inclusive or of the two arguments.

`(nand a b)` $\{alpha \leftarrow alpha\ alpha : \ alpha \text{ in } logical\}$

Returns the inverse of the and of the two arguments.

4.1.2 Comparison Functions

All comparison functions work on integers, floats and characters.

`(= a b)` $\{alpha \leftarrow alpha\ alpha : \ alpha \text{ in } ordinal\}$

Returns T if the two arguments are equal.

`(/= a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

Returns T if the two arguments are not equal.

`(< a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

Returns T if the first argument is strictly less than the second argument.

`(> a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

Returns T if the first argument is strictly greater than the second argument.

`(<= a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

Returns T if the first argument is less than or equal to the second argument.

`(>= a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

Returns T if the first argument is greater or equal to the second argument.

`(select flag a b)` $\{alpha \leftarrow bool\ alpha\ alpha :\ alpha \text{ in } any\}$

Returns the second argument if the flag is T and the third argument if the flag is F.

4.1.3 Predicates

`(plusp v)` $\{bool \leftarrow alpha :\ alpha \text{ in } number\}$

Returns T if `V` is strictly greater than 0.

`(minusp v)` $\{bool \leftarrow alpha :\ alpha \text{ in } number\}$

Returns T if `V` is strictly less than 0.

`(zerop v)` $\{bool \leftarrow alpha :\ alpha \text{ in } number\}$

Returns T if `V` is equal to 0.

`(oddp v)` $\{bool \leftarrow int\}$

Returns T if `V` is odd (not divisible by two).

`(evenp v)` $\{bool \leftarrow int\}$

Returns T if `V` is even (divisible by two).

4.1.4 Arithmetic Functions

`(+ a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } number\}$

Returns the sum of the two arguments.

`(- a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } number\}$

Subtracts the second argument from the first.

`(negate v)` $\{alpha \leftarrow alpha :\ alpha \text{ in } number\}$

Negates a number.

`(abs x)` $\{alpha \leftarrow alpha :\ alpha \text{ in } number\}$

Returns the absolute value of the argument.

`(diff x y)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } number\}$

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

`(max a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

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

`(min a b)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } ordinal\}$

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

`(* v d)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } number\}$

Returns the product of the two arguments.

`(/ v d)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } number\}$

Returns `V` divided by `D`. If the arguments are integers, the result is truncated towards 0.

`(rem v d)` $\{alpha \leftarrow alpha\ alpha :\ alpha \text{ in } number\}$

Returns the remainder after dividing `V` by `D`.

`(lshift a b)` $\{int \leftarrow int\ int\}$

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

`(rshift a b)` $\{int \leftarrow int\ int\}$

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

`(sqrt v)` $\{float \leftarrow float\}$

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

`(isqrt v)` $\{int \leftarrow int\ int\}$

Returns the greatest integer less than or equal to the exact square root of the integer argument.

`(ln v)` $\{float \leftarrow float\}$

Returns the natural log of the argument.

`(log v b)` $\{\mathit{float} \leftarrow \mathit{float}\ \mathit{float}\}$

Returns the logarithm of `V` in the base `B`.

`(exp v)` $\{\mathit{float} \leftarrow \mathit{float}\}$

Returns e raised to the power `V`.

`(expt v p)` $\{\mathit{float} \leftarrow \mathit{float}\ \mathit{float}\}$

Returns `V` raised to the power `P`.

4.1.5 Conversion Functions

`(btoi a)` $\{\mathit{int} \leftarrow \mathit{bool}\}$

Converts the boolean values T and F into 1 and 0, respectively.

`(code-char a)` $\{\mathit{char} \leftarrow \mathit{int}\}$

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

`(char-code a)` $\{\mathit{int} \leftarrow \mathit{char}\}$

Converts a character to its integer code.

`(float v)` $\{\mathit{float} \leftarrow \mathit{int}\}$

Converts an integer to a floating-point number.

`(ceil v)` $\{\mathit{int} \leftarrow \mathit{float}\}$

Converts a floating-point number to an integer by truncating toward positive infinity.

`(floor v)` $\{\mathit{int} \leftarrow \mathit{float}\}$

Converts a floating-point number to an integer by truncating toward negative infinity.

`(trunc v)` $\{\mathit{int} \leftarrow \mathit{float}\}$

Converts a floating-point number to an integer by truncating toward zero.

`(round v)` $\{\mathit{int} \leftarrow \mathit{float}\}$

Converts 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.

4.1.6 Other Scalar Functions

`(rand v)` $\{\mathit{int} \leftarrow \mathit{int}\}$

Returns a random integer value between 0 and `V`.

4.2 Vector Functions

4.2.1 Simple Vector Functions

`(dist a l)` $\{v.alpha \leftarrow v.alpha\ int :\ alpha \text{ in } any\}$

Generates a vector of length `L` with the value `A` in each element. For example:

$$\begin{array}{lcl} \texttt{a} & = & a_0 \\ \texttt{l} & = & 5 \\ \texttt{(dist a l)} & = & [a_0 \quad a_0 \quad a_0 \quad a_0 \quad a_0] \end{array}$$

`(elt v i)` $\{alpha \leftarrow v.alpha\ int :\ alpha \text{ in } any\}$

Extracts the element specified by index `i` from the vector `v`.

`(rep v a i)` $\{v.alpha \leftarrow v.alpha\ alpha\ int :\ alpha \text{ in } any\}$

Replaces the `ith` value in the vector `v` with the value `a`. For example:

$$\begin{array}{lcl} \texttt{v} & = & [a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4] \\ \texttt{a} & = & b_0 \\ \texttt{i} & = & 3 \\ \texttt{(rep v a i)} & = & [a_0 \quad a_1 \quad a_2 \quad b_0 \quad a_4] \end{array}$$

`(length v)` $\{int \leftarrow v.alpha :\ alpha \text{ in } any\}$

Returns the length of a vector.

`(index l)` $\{int \leftarrow int\}$

Given an integer, `index` returns a vector of that length with consecutive integers starting at 0 in the elements. For example:

$$\begin{array}{lcl} \texttt{l} & = & 8 \\ \texttt{(index l)} & = & [0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7] \end{array}$$

4.2.2 Scans and Reduces

`(+-scan a)` $\{v.alpha \leftarrow v.alpha :\ alpha \text{ in } number\}$

Given a vector of integers, `+-scan` returns to each position of a new equal-length vector, the sum of all previous positions in the source. For example:

$$\begin{array}{lcl} \texttt{a} & = & [1 \quad 3 \quad 5 \quad 7 \quad 9 \quad 11 \quad 13 \quad 15] \\ \texttt{(+-scan a)} & = & [0 \quad 1 \quad 4 \quad 9 \quad 16 \quad 25 \quad 36 \quad 49] \end{array}$$

`(max-scan a)` $\{v.alpha \leftarrow v.alpha : \; alpha \text{ in } ordinal\}$

Given a vector of integers, `max-scan` returns to each position of a new equal-length vector, the maximum of all previous positions in the source. For example:

`a`	=	[3	2	1	6	5	4	8]
`(max-scan a)`	=	[$-\infty$	3	3	3	6	6	6]

`(min-scan a)` $\{v.alpha \leftarrow v.alpha : \; alpha \text{ in } ordinal\}$

Given a vector of integers, `min-scan` returns to each position of a new equal-length vector, the minimum of all previous positions in the source.

`(iseq start stride end)` $\{v.int \leftarrow int \; int \; int\}$

Returns a set of indices starting at `start`, increasing by `stride`, and finishing before `end`. For example:

`start`	=	4			
`stride`	=	3			
`end`	=	15			
`(iseq start stride end)`	=	[4	7	10	13]

`(+-reduce v)` $\{alpha \leftarrow v.alpha : \; alpha \text{ in } number\}$

Given a vector of integers, `+-reduce` returns the sum of the integers. For example:

`v`	=	[7	2	9	11	3]
`(+-reduce v)`	=	32				

`(max-reduce v)` $\{alpha \leftarrow v.alpha : \; alpha \text{ in } ordinal\}$

Given a vector of integers, `max-reduce` returns the maximum of the integers.

`(min-reduce v)` $\{alpha \leftarrow v.alpha : \; alpha \text{ in } ordinal\}$

See max-reduce.

`(or-reduce v)` $\{alpha \leftarrow v.alpha : \; alpha \text{ in } logical\}$

A reduce with logical or.

`(and-reduce v)` $\{alpha \leftarrow v.alpha : \; alpha \text{ in } logical\}$

A reduce with logical and.

`(count flags)` $\{int \leftarrow v.bool\}$

Counts the number of T flags in a boolean vector. For example:

`flags`	=	[T F T T F T F T]
`(count flags)`	=	5

`(max-index v)` $\{int \leftarrow v.alpha :\ alpha \text{ in } ordinal\}$

Given a vector of integers, `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]
`(max-index v)`	=	4

`(min-index v)` $\{int \leftarrow v.alpha :\ alpha \text{ in } ordinal\}$

Given a vector of integers, `min-index` returns the index of the minimum value. If several values are equal, it returns the leftmost index.

4.2.3 Vector Reordering Functions

`(get values indices)` $\{v.alpha \leftarrow v.alpha\ v.int :\ alpha \text{ in } any\}$

Given a vector of `values` and a vector of `indices`, which can be of different length, `get` returns a vector which is the same length as the indices vector and the same type as the values vector. For each position in the `indices` vector, it extracts the value at that index of the `values` vector.

`(cond-get values indices flags)` $\{v.alpha \leftarrow v.alpha\ v.int\ v.bool :\ alpha \text{ in } any\}$

Similar to the `get` function, but the `flags` vector, which must be the same length as the `indices` vector, masks out positions where the flag is F such that nothing is fetched in those positions and the identity is returned. For example:

`values`	=	$[a_0\ \ a_1\ \ a_2\ \ a_3\ \ a_4\ \ a_5\ \ a_6\ \ a_7]$
`indices`	=	[3 1 5 2 5 6]
`flags`	=	[T F T T F T]
`(cond-get values indices flags)`	=	$[a_3\ \ 0\ \ a_5\ \ a_2\ \ 0\ \ a_6]$

`(permute values indices)` $\{v.alpha \leftarrow v.alpha\ v.int :\ alpha \text{ in } any\}$

Given a vector of `values` and a vector of `indices`, which must be of the same length, `permute` permutes the values to the given indices. The permutation must be 1-to-1.

`(put values indices defaults)` $\{v.alpha \leftarrow v.alpha\ v.int\ v.alpha :\ alpha \text{ in } any\}$

Given a vector of `values` and a vector of `indices`, which must be of the same length, `put` places the values to the given indices in the `defaults` vector. For example:

`values`	=	$[a_0 \quad a_1 \quad a_2 \quad a_3]$
`indices`	=	$[1 \quad 5 \quad 2 \quad 3]$
`defaults`	=	$[b_0 \quad b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5]$
`(put values indices defaults)`	=	$[b_0 \quad a_0 \quad a_2 \quad a_3 \quad b_4 \quad a_1]$

`(const-put value indices defaults)` $\{v.alpha \leftarrow alpha\ v.int\ v.alpha :\ alpha \text{ in } any\}$

Given a vector of `indices`, `put` places the constant `value` at each index in the `defaults` vector.

`(cond-put v i flags d)` $\{v.alpha \leftarrow v.alpha\ v.int\ v.bool\ v.alpha :\ alpha \text{ in } any\}$

Similar to the `put` function, but the extra `flags` vector, which must be the same length as the indices (`i`) vector, masks out positions where the flag is F such that nothing is placed by those positions. For example:

`v`	=	$[a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7]$
`i`	=	$[1 \quad 5 \quad 4 \quad 6 \quad 2 \quad 3 \quad 7 \quad 0]$
`flags`	=	[T F F F F T F F]
`d`	=	$[b_0 \quad b_1 \quad b_2 \quad b_3]$
`(cond-put v i flags d)`	=	$[b_0 \quad a_0 \quad b_2 \quad a_5]$

`(rotate a i)` $\{v.alpha \leftarrow v.alpha\ int :\ alpha \text{ in } any\}$

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

`a`	=	$[a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7]$
`i`	=	3
`(rotate a i)`	=	$[a_5 \quad a_6 \quad a_7 \quad a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4]$

4.2.4 Vector Manipulation

`(pack v flags)` $\{v.alpha \leftarrow v.alpha\ v.bool :\ alpha \text{ in } any\}$

Given a sequence of values, and an equal length boolean sequence of flags, `pack` packs all the elements with a T in their Flag into consecutive elements, deleting elements with an F in their flag. For example:

`v`	=	$[a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7]$
`flags`	=	[T F T F F T T T]
`(pack v flags)`	=	$[a_0 \quad a_2 \quad a_5 \quad a_6 \quad a_7]$

`(pack-index flags)` $\{v.int \leftarrow v.bool\}$

Given a boolean sequence of flags, `pack-index` returns a vector containing the indices of each of the true flags. For example:

`flags`	=	[T F T F F T T T]
`(pack-index flags)`	=	[0 2 5 6 7]

`(append v1 v2)` $\{v.alpha \leftarrow v.alpha\ v.alpha : \ alpha \text{ in } any\}$

Given two sequences, `append` appends them. For example:

`v1`	=	$[a_0 \quad a_1 \quad a_2]$
`v2`	=	$[b_0 \quad b_1]$
`(append v1 v2)`	=	$[a_0 \quad a_1 \quad a_2 \quad b_0 \quad b_1]$

`(cons a v)` $\{v.alpha \leftarrow alpha\ v.alpha : \ alpha \text{ in } any\}$

Given a value `a` and a sequence of values `v`, `cons` concatenates the value onto the front of the sequence. For example:

`a`	=	a_0
`v`	=	$[b_0 \quad b_1 \quad b_2 \quad b_3]$
`(cons a v)`	=	$[a_0 \quad b_0 \quad b_1 \quad b_2 \quad b_3]$

`(vtup a b)` $\{v.alpha \leftarrow alpha\ alpha : \ alpha \text{ in } any\}$

Given two values of the same type, `vtup` puts them together into a sequence of length 2. For example:

`a`	=	a_0
`b`	=	a_1
`(vtup a b)`	=	$[a_0 \quad a_1]$

`(vsep a)` $\{(alpha\ alpha) \leftarrow v.alpha : \ alpha \text{ in } any\}$

Given a vector of length 2, `vsep` returns a tuple with the first vector element in the first position and the second vector element in the second position.

`(subseq v start end)` $\{v.alpha \leftarrow v.alpha\ int\ int : \ alpha \text{ in } any\}$

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

`v`	=	$[a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7]$
`start`	=	2
`end`	=	6
`(subseq v start end)`	=	$[a_2 \quad a_3 \quad a_4 \quad a_5]$

`(drop n v)` $\{v.alpha \leftarrow int\ v.alpha : \ alpha \text{ in } any\}$

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

`n`	=	3							
`v`	=	$[a_0$	a_1	a_2	a_3	a_4	a_5	a_6	$a_7]$
`(drop n v)`	=	$[a_3$	a_4	a_5	a_6	$a_7]$			

`(take n v)` $\{v.alpha \leftarrow int\ v.alpha : \ alpha \text{ in } any\}$

Given a sequence, `take` takes the first `n` items from the sequence. For example:

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

4.2.5 Nesting Vectors

The two functions `partition` and `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 `split` and `bottop` are often useful for divide-and-conquer routines.

`(partition values counts)` $\{v.v.alpha \leftarrow v.alpha\ v.int : \ alpha \text{ in } any\}$

Given a sequence of values and another sequence of counts, `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:

`values`	=	$[a_0$	a_1	a_2	a_3	a_4	a_5	a_6	$a_7]$
`counts`	=	[4	1	3]					
`(partition values counts)`	=	$[[a_0$	a_1	a_2	$a_3]$	$[a_4]$	$[a_5$	a_6	$a_7]]$

`(flatten values)` $\{v.alpha \leftarrow v.v.alpha : \ alpha \text{ in } any\}$

Given a nested sequence of values, `flatten` flattens the sequence. For example:

`values`	=	$[[a_0$	a_1	$a_2]$	$[a_3$	$a_4]$	$[a_5$	a_6	$a_7]]$
`(flatten values)`	=	$[a_0$	a_1	a_2	a_3	a_4	a_5	a_6	$a_7]$

`(split v flags)` $\{v.v.alpha \leftarrow v.alpha\ v.bool\}$

Given a sequence of values `A` and a boolean sequence of `flags`, `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	$a_7]$
`flags`	=	[T	F	T	F	F	T	T	T]
`(split v flags)`	=	$[[a_1$	a_3	$a_4]$	$[a_0$	a_2	a_5	a_6	$a_7]]$

`(bottop values)` $\{v.v.alpha \leftarrow v.alpha\}$

Given a sequence of values `values`, `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:

`values`	=	$[a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6]$
`(bottop values)`	=	$[[a_0 \quad a_1 \quad a_2 \quad a_3] \quad [a_4 \quad a_5 \quad a_6]]$

Acknowledgments

I would like to thank Marco Zagha, Tim Freeman, Jay Sipelstein, Margaret Reid-Miller, and John Greiner, Jonathan Hardwick, and Siddhartha Chatterjee for many helpful comments on this manual. Tim Freeman implemented much of the NESL/VCODE interface.

References

[1] ANSI. *ANSI Fortran Draft S8, Version 111.* ANSI.

[2] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[4] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Portable data-parallel algorithms. In Preparation, 1992.

[5] Guy E. Blelloch, Siddhartha Chatterjee, Fritz Knabe, Jay Sipelstein, and Marco Zagha. VCODE reference manual (version 1.1). Technical Report CMU-CS-90-146, School of Computer Science, Carnegie Mellon University, July 1990.

[6] Guy E. Blelloch and James J. Little. Parallel solutions to geometric problems on the scan model of computation. In *Proceedings International Conference on Parallel Processing*, pages Vol 3: 218–222, August 1988.

[7] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2), February 1990.

[8] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–206, 1974.

[9] D. Breslauer and Z. Galil. An optimal o(log log n) time parallel string matching algorithm. *SIAM Journal on Computing*, 19(6):1051–1058, December 1990.

[10] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1991.

[11] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[12] H. Freeman. Computer processing of line-drawing images. *Computer Surveys*, 6:57–97, 1974.

[13] Godfrey Harold Hardey and Edward Maitland Wright. *An Introduction to the Theory of Numbers, 5th ed.* Oxford at the Carendon Press, Oxford, New York, 1983.

[14] C. A. R. Hoare. Algorith 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[15] J. G. Hocking and G. S. Young. *Topology.* Addison-Wesley, Reading, MA, 1961.

[16] Kenneth E. Iverson. *A Programming Language.* Wiley, New York, 1962.

[17] Clifford Lasser. The essential *Lisp manual. Technical report, Thinking Machines Corporation, Cambridge, MA, July 1986.

[18] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.

[19] Franco P. Preparata and Michael I. Shamos. *Computational Geometry—An Introduction.* Springer-Verlag, New York, 1985.

[20] John Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL87-5, Thinking Machines Corporation, April 1987.

[21] Gary Sabot. *Paralation Lisp Reference Manual*, May 1988.

[22] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[23] Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[24] Guy L. Steele Jr. CM-Lisp. Technical report, Thinking Machines Corporation, 1986.

[25] U. Vishkin. Deterministic sampling-a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, February 1991.