Matrix Inversion using Quadtrees Implemented in Gofer [1] Technical Report 433

Jeremy D. Frens & David S. Wise [2]
Computer Science Department
Indiana University
Bloomington, Indiana 47405–4101
`jfrens@cs.indiana.edu`

May 1995

[2] Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 90-027092.

Abstract

Using literate programming, complete Gofer code to invert a floating-point quadtree matrix is presented along with exposition. The code is a full implementation some of the algorithms presented in "Undulant-Block Pivoting and Integer-Preserving Matrix Inversion" [4] by the second author.

CR categories and Subject Descriptors:
G.1.3 **[Numerical Linear Algebra]**: Matrix inversion, Sparse and very large systems; E.1 **[Data Structures]**: Trees; D.1.1 **[Applicative (Functional) Programming Techniques]**; I.1.2 **[Algebraic Manipulation]**: Algebraic algorithms; D.2.7 **[Distribution and Maintenance]**: Documentation; C.1.2 **[Multiple Data Stream Architectures (Multiprocessors)]**: Parallel processors.
General Term: Algorithms.
Additional Key Words and Phrases: Gaussian elimination, block algorithm, undulant pivoting, *LU* factorization, quaternary trees, Haskell, literate programming.

Contents

0 Introduction

This paper implements and explains the code described by Wise in [4] for undulant-block pivoting and rational-arithmetic matrix inversion. This code for rational arithmetic was originally written by the second author in Scheme and is published here in readable Haskell. Similar code for exact arithmetic [4, Section 4] has also been written in Scheme.

0.1 Overview of this Report

The code presented here is written for Gofer [2], a dialect of Haskell [1]. The code is intermixed with comments to make this report by using the literate programming tool `noweb` [3]. Similar to Knuth's `WEB`, `noweb` is a system where documentation and code are written in the same files. These `noweb` files can be processed to generate code or documentation.

In this report, each of the following sections represents one code file which implements in Gofer one conceptual portion of the matrix inversion algorithms. Each section begins with a brief description of the purpose of the code in that section. The first subsection is a file layout for the code. The code chunk for the file layout is named with the file name for the code in the section. This code chunk refers to other code chunks which are defined in subsequent subsections.

The source code presented in this report is publicly available at `ftp.cs.indiana.edu` through anonymous FTP in the directory `/pub/dswise/TR433code/`.

0.2 Overview of the Code

The first six sections of this report define the basic data types and the support functions for them. These data types include pair and quadtree data types, a vector definition, and matrix definitions.

The last two sections implement the inversion algorithm for quadtree matrices as described in [4]. To invert a matrix A, it must be decorated, decomposed, triangulated, and then finally inverted. Decoration is handled by the decorated matrix and its support functions. Decomposition returns five results: the determinant of A, the decomposition S, series encodings Π and Ψ for permutation matrices P and Q, respectively, and a list of pivot orders Ω.

Since this code implements undulant-block pivoting, the pivot blocks can come from anywhere and be of various sizes. So S, the decomposition of A, has pivot blocks scattered all over. However, conventional Gaussian elimination requires that pivoting be done along the main diagonal. The permutation matrices P and Q make row and column exchanges, respectively, to move the pivot blocks onto the main diagonal. This results in a triangulation $L + U'$ where $L + U' = PSQ$. The list of pivot orders Ω makes it clear where the break is between L and U'. The triangulation is then passed on to three functions to complete the inversion.

The appendices contain several program files. The first defines a project file for Gofer which will load the program files in the correct order. One program file contains definitions of elementary functions that Gofer lacks. The other program files in the appendices implement book keeping algorithms which are not covered in [4] and could be replaced with other algorithms.

0.3 Notation

0.3.1 Vectors, Lists, Tuples, and Matrices

Matrices have a two-dimensional shape and so appear as blocks. Vectors are designated by the angle brackets, $\langle$ and $\rangle$. Lists are designated with square brackets, [and]. Tuples are represented with parentheses (and). Pair vectors and quadtree matrices are represented with tuple notation.

0.3.2 Coding

The notation used in Haskell in this paper is based on the idea that capital letters signify data types. Data types use capital letters not only to start the name of the data type, but also to highlight subnames. For example, `Quad` ("quad" for "quadtree") and `AhnenOrdList` ("Ahnentafel/order list") are data types.

Identifiers for functions and variables do not use capital letters except when referring to a data type. Identifiers like `addresses_padding` and `full_ord` are used for function and variable names, utilizing the underscore to break up the identifier for easy reading. Occasionally a capital letter is used in an function identifier to designate the data type that the function uses. For example, `mtxRMtoQ` converts a row-major matrix into a quadtree matrix. Similarly, the function `convALtoPM` converts a reverse-Ahnentafel list into a permutation matrix.

0.4 Acknowledgments

Thanks must go out to Mark Jones for Gofer [2], an implementation of Haskell [1]. Thanks must also go to Norman Ramsey for his `noweb` literate programming tool [3].

1 Pairs and Quadtrees

A pair is a homogeneous binary tuple; a quadtree is a homogeneous four-tuple. The main motivation for the pair and quadtree is so that we can use mapping functions over them, breaking a problem into subproblems fit for parallelization. Since Gofer and Haskell do not provide mapping functions over tuples, mapping functions are defined in this section.

For the purposes of this report, the pair is used to store vectors (Section 2) while the quadtree is used to store matrices (Section 3).

1.1 File Layout

3a ⟨`pandq.gs` 3a⟩≡
  ⟨*Data definitions* 3b⟩
  ⟨*Pair mapping functions* 4a⟩
  ⟨*Pair rearrangement* 4b⟩
  ⟨*Quadtree mapping functions* 5a⟩
  ⟨*Quadtree rearrangement* 6⟩

Root chunk (not used in this document).

1.2 Pair and Quadtree Data Definitions

A pair is a homogeneous binary tuple. Note that `a` is any abstract data type.

3b ⟨*Data definitions* 3b⟩≡

```
type Pair a  =  (a, a)
```

Defines:
  `Pair`, used in chunks 4 and 8a.
This definition is continued in chunk 3c.
This code is used in chunk 3a.

The quadtree is a homogeneous four-tuple:

3c ⟨*Data definitions* 3b⟩+≡

```
type Quad a  =  (a, a, a, a)
```

Defines:
  `Quad`, used in chunks 5, 6, 13a, 20b, and 24b.

1.3 Pair Mapping Functions

Mapping over a pair is not different from mapping over a list, except for two things: the construction of the data structure and the arity of the structure. Deconstructing and reconstructing the pair as opposed to the list is just a minor change. The arity of the structure is most important because the compiler could schedule the mapping over a pair very precisely wheras a list could be of any length.

The functions `pmap` and `pmap2` map a function over a pair and two pairs, respectively.[1] The function `andpmap2` maps a Boolean function over two pairs and returns the conjunction of the results.

Only the functions required in the matrix inversion algorithms have defined.

4a ⟨*Pair mapping functions* 4a⟩≡
```
pmap :: (a -> b) -> Pair a -> Pair b
pmap f (p, q)  =  (f p, f q)

pmap2 :: (a -> b -> c) -> Pair a -> Pair b -> Pair c
pmap2 f (p0, q0) (p1, q1)  =  (f p0 p1, f q0 q1)

andpmap2 :: (a -> b -> Bool) -> Pair a -> Pair b -> Bool
andpmap2 f (p0, q0) (p1, q1)  =  (f p0 p1) && (f q0 q1)
```
Defines:
  `andpmap2`, used in chunk 9a.
  `pmap`, used in chunks 9c, 54, and 55.
  `pmap2`, used in chunk 9c.
Uses `Pair` 3b.
This code is used in chunk 3a.

1.4 Pair Rearrangement

There are only two basic ways to rearrange a pair.

4b ⟨*Pair rearrangement* 4b⟩≡
```
identP :: Pair a -> Pair a
identP p  =  p

exchangeP :: Pair a -> Pair a
exchangeP (n, s)  =  (s, n)
```
Defines:
  `exchangeP`, used in chunk 39a.
  `identP`, used in chunk 39a.
Uses `Pair` 3b.
This code is used in chunk 3a.

[1] While Haskell uses `map` for one list and `zipWith` for multiple lists, the `map` nomenclature has been used in this report.

1.5 Quadtree Mapping Functions

Similarly to the pair mapping functions, there is no difference between the mapping functions for quadtrees and the mapping functions for lists, except in deconstruction/reconstruction and arity.

The conventional mapping functions are defined first:

5a ⟨*Quadtree mapping functions* 5a⟩≡

```
qmap :: (a -> b) -> Quad a -> Quad b
qmap f (p,q,r,s) = (f p, f q, f r, f s)

qmap2 :: (a -> b -> c) -> Quad a -> Quad b -> Quad c
qmap2 f (p,q,r,s) (p',q',r',s') = (f p p', f q q', f r r', f s s')

qmap3 :: (a -> b -> c -> d) -> Quad a -> Quad b -> Quad c -> Quad d
qmap3 f (p0,q0,r0,s0) (p1,q1,r1,s1) (p2,q2,r2,s2)  =
  (f p0 p1 p2, f q0 q1 q2, f r0 r1 r2, f s0 s1 s2)

qmap4 :: (a -> b -> c -> d -> e) -> Quad a -> Quad b -> Quad c
  -> Quad d -> Quad e
qmap4 f (p0,q0,r0,s0) (p1,q1,r1,s1) (p2,q2,r2,s2) (p3,q3,r3,s3)  =
  (f p0 p1 p2 p3, f q0 q1 q2 q3, f r0 r1 r2 r3, f s0 s1 s2 s3)
```

Defines:
  `qmap`, used in chunks 15, 18c, 25–28, and 54.
  `qmap2`, used in chunks 15, 25c, 33, and 34.
  `qmap3`, used in chunk 40b.
  `qmap4`, used in chunk 40a.
Uses `Quad` 3c.
This definition is continued in chunk 5b.
This code is used in chunk 3a.

Logical mapping functions are also useful:

5b ⟨*Quadtree mapping functions* 5a⟩+≡

```
andqmap :: (a -> Bool) -> Quad a -> Bool
andqmap f (p,q,r,s)  =  (f p) && (f q) && (f r) && (f s)

andqmap2 :: (a -> b -> Bool) -> Quad a -> Quad b -> Bool
andqmap2 f (p0,q0,r0,s0) (p1,q1,r1,s1)  =
  (f p0 p1) && (f q0 q1) && (f r0 r1) && (f s0 s1)
```

Defines:
  `andqmap`, used in chunk 27b.
  `andqmap2`, used in chunks 14b and 25a.
Uses `Quad` 3c.

1.6 Quadtree Rearrangement

Occasionally, the quadrants of a quadtree must be shuffled around. The names here are motivated directly from quadtree matrices (see Section 3). The first function `identQ` is the identity function. The `exchange` functions exchange the columns, rows or diagonals of the quadtree matrix. The `squash` functions are used to squash out one of the diagonals of the quadtree matrix.

6 ⟨*Quadtree rearrangement* 6⟩≡

```
identQ :: Quad a -> Quad a
identQ q  =  q

col_exchangeQ :: Quad a -> Quad a
col_exchangeQ (w, x, y, z)  =  (x, w, z, y)

row_exchangeQ :: Quad a -> Quad a
row_exchangeQ  (w, x, y, z)  =  (y, z, w, x)

diag_exchangeQ :: Quad a -> Quad a
diag_exchangeQ (w, x, y, z)  =  (z, y, x, w)

off_squashQ :: Quad a -> Quad a
off_squashQ (_, x, y, _)  =  (y, x, y, x)

prim_squashQ :: Quad a -> Quad a
prim_squashQ (w, _, _, z)  =  (w, z, w, z)
```

Defines:
- `col_exchangeQ`, used in chunks 15, 25c, 33, 34, and 39a.
- `diag_exchangeQ`, used in chunk 39a.
- `identQ`, used in chunk 39a.
- `off_squashQ`, used in chunks 15, 25c, 33, and 34.
- `prim_squashQ`, used in chunks 15, 25c, 33, and 34.
- `row_exchangeQ`, used in chunk 39a.

Uses `Quad` 3c.
This code is used in chunk 3a.

2 Binary Vectors

Vectors are stored in pairs which were defined in the previous section. Thus a binary vector is either a scalar value or a pair containing "north" and "south" binary vectors. For example, the vector

$$\langle 8, 12, 10, 14, 9, 13, 11, 15\rangle$$

would be stored in the form

$$(((8, 12), (10, 14)), ((9, 13), (11, 15)))$$

where $((8, 12), (10, 14))$ is the north half of the vector and $((9, 13), (11, 15))$ is the south half. (See the formal definition in [4, Section 1.1].)

To divide a vector recursively in half like this, the vector must have a length which is a power of two. Not every vector will conform to this requirement, so padding is added to the original vector to pad it out to a power of two. Specifically, zeros are added to the end of the given vector. Simply adding this padding might increase the length of the binary vector considerably, possibly doubling it. Instead of using scalar zeros for padding, a special zero vector Z is used which represents a zero vector of *any* length. The length of a given Z is determined by the length of a vector. Thus, a vector like

$$\langle 1, 2, 3, 4, 5\rangle$$

is implicitly padded out to be

$$\langle 1, 2, 3, 4, 5, 0, 0, 0\rangle,$$

but explicitly stored as

$$(((1, 2), (3, 4)), ((5, Z), Z)).$$

Note that the last two zeros of the padded linear vector are normalized into one single Z. The algebra of Z permits this normalization without fear of adverse side effects. In fact, *all* zeros in a vector are normalized in this way which makes the binary vector a good choice for representing sparse vectors. Thus, $\langle 1, 0, 0, 0, 0, 0, 7\rangle$ is stored as $(((1, Z)Z), (Z, (7, Z)))$.

In order to recognize the actual vector from its padding, the length of the vector must be known. This is most important when converting from the pair representation back to a list on output. (See Footnote 2 in [4].)

2.1 File Layout

7 ⟨`bin-vector.gs` 7⟩≡
  ⟨*Vector definition* 8a⟩
  ⟨*Vector inheritance* 8c⟩
  ⟨*Vector normalization functions* 8b⟩
  ⟨*Top level vector definition* 10a⟩
  ⟨*Top level vector inheritance* 10b⟩
  ⟨*Vector conversion functions* 10c⟩
Root chunk (not used in this document).

2.2 Binary Vector Data Definition

The `Vectr` class is the finite binary vector data type. It has three cases (see also [4, Section 1.1]):

- The `ZeroV` constructor is the sentinel value Z which represents a zero vector of arbitrary length.
- The `ScalarV` constructor stores non-zero scalar data of any type.
- The `Vec` constructor stores a pair of north and south homogeneous, equal-sized vectors.

8a ⟨*Vector definition* 8a⟩≡

```
type PairV a  =  Pair (Vectr a)
data Vectr a  =  ZeroV | ScalarV a| Vec (PairV a)
```

Defines:
`PairV`, used in chunk 8b.
`ScalarV`, used in chunks 8, 9, 36, 37a, 40, 42, 45, 46, 54, and 55.
`Vec`, used in chunks 8, 9, 37a, 40, 41a, 45, 46, 54, and 55.
`Vectr`, used in chunks 8–11, 36, 37a, 39–42, 45, 46, 54, and 55.
`ZeroV`, used in chunks 8–11, 39–42, 54, and 55.
Uses `Pair` 3b.
This code is used in chunk 7.

2.3 Vector Normalization Functions

These functions insure that `ZeroV` is used whenever possible. Normalization consists of turning a scalar zero or `Vec(ZeroV,ZeroV)` into `ZeroV`.

The function names used are straightforward: `normalizeSV` normalizes a scalar values into a binary vector and `normalizePV` normalizes a pair of vectors into a vector.

8b ⟨*Vector normalization functions* 8b⟩≡

```
normalizeSV :: (Eq a, Num a) => a -> Vectr a
normalizeSV n | n == fromInteger 0  =  ZeroV
              | otherwise           =  ScalarV n

normalizePV :: PairV a -> Vectr a
normalizePV (ZeroV, ZeroV)  =  ZeroV
normalizePV p               =  Vec p
```

Defines:
`normalizePV`, used in chunks 9c, 11b, 39a, 42, and 55.
`normalizeSV`, used in chunks 9c and 10c.
Uses `PairV` 8a, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, and `ZeroV` 8a.
This code is used in chunk 7.

2.4 Vector Inheritance

Vectors inherit operations from three classes: the `Eq`, `Text`, and `Num` classes.

8c ⟨*Vector inheritance* 8c⟩≡
⟨*Vector inheritance of* `Eq` 9a⟩
⟨*Vector inheritance of* `Text` 9b⟩
⟨*Vector inheritance of* `Num` 9c⟩
This code is used in chunk 7.

The first class to be inherited is the `Eq` class. The definition for `Eq` inheritance is straightforward: vectors are equal to each other when the components match up.

Note the use of `andpmap2` in the recursive case.

9a ⟨*Vector inheritance of* `Eq` 9a⟩≡
```
  instance Eq a => Eq (Vectr a) where
     ZeroV      == ZeroV      =  True
     ScalarV x == ScalarV y  =  x == y
     Vec p0     == Vec p1     =  andpmap2 (==) p0 p1
     _          == _          =  False
```
Uses `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroV` 8a, and `andpmap2` 4a.
This code is used in chunk 8c.

The inheritance of `Text` contains no definitions presently, but Gofer insists on this inheritance for type checking. It would be possible to include the conversion functions defined in Section 2.6 that go from a list representation to our binary vector representation.

9b ⟨*Vector inheritance of* `Text` 9b⟩≡
```
  instance Text a => Text (Vectr a)
```
Uses `Vectr` 8a.
This code is used in chunk 8c.

The class `Num` is the most important inheritance. The `Num` class defines the ring operations `+`, `-` and `negate`. Note that the results of `+` and `-` must be normalized so that the `ZeroV` data type is used to its full potential. Also note the use of the mapping functions `pmap2` and `pmap`.

9c ⟨*Vector inheritance of* `Num` 9c⟩≡
```
  instance Num a => Num (Vectr a) where
     fromInteger 0           =  ZeroV
     fromInteger n           =  ScalarV (fromInteger n)
     u         + ZeroV       =  u
     ZeroV     + v           =  v
     ScalarV x + ScalarV y   =  normalizeSV (x + y)
     Vec p0    + Vec p1      =  normalizePV (pmap2 (+) p0 p1)
     u         - ZeroV       =  u
     ZeroV     - v           =  negate v
     ScalarV x - ScalarV y   =  normalizeSV (x - y)
     Vec p0    - Vec p1      =  normalizePV (pmap2 (-) p0 p1)
     negate ZeroV            =  ZeroV
     negate (ScalarV x)      =  ScalarV (negate x)
     negate (Vec p)          =  Vec (pmap negate p)
```
Uses `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroV` 8a, `normalizePV` 8b, `normalizeSV` 8b, `pmap` 4a, and `pmap2` 4a.
This code is used in chunk 8c.

2.5 Top Level Binary Vector

The top level binary vector is simply the binary vector defined above plus the order of the actual vector. This is important for the conversion from the binary pair representation to a linear list representation.

2.5.1 Top Level Binary Vector Definition

The `OVec` constructor makes a top level vector containing the order (or length) of the vector and the binary vector representation of the vector.

10a ⟨*Top level vector definition* 10a⟩≡

```
data Vector a = OVec Int (Vectr a)
```

Defines:
  `OVec`, used in chunk 10.
  `Vector`, used in chunk 10.
Uses `Vectr` 8a.
This code is used in chunk 7.

2.5.2 Top Level Binary Vector Inheritance

Inheritance for the top level vector is very simple since it is based on the inheritance of `Int` and `Vectr`. Note that binary operations first carry out a check of the orders of the two vectors to make sure that the vectors are compatible.

10b ⟨*Top level vector inheritance* 10b⟩≡

```
instance (Eq a, Eq (Vectr a)) => Eq (Vector a) where
  (OVec n u) == (OVec m v)  =  (n == m) && (u == v)

instance (Text a, Text (Vectr a)) => Text (Vector a)

instance (Num a, Num (Vectr a)) => Num (Vector a) where
  (OVec n u) + (OVec m v)  |  n == m     =  OVec n (u + v)
                           |  otherwise  =  error "Vectors are incompatible."
  (OVec n u) - (OVec m v)  |  n == m     =  OVec n (u - v)
                           |  otherwise  =  error "Vectors are incompatible."
  negate (OVec n v)  =  OVec n (negate v)
```

Uses `OVec` 10a, `Vector` 10a, and `Vectr` 8a.
This code is used in chunk 7.

2.6 Vector Conversion Functions

It is easier for people to think of vectors as a linear sequential list of fixed size, rather than as a binary tree. The functions in this section take a linear vector represented as a list and return the equivalent binary vector.

The top level function `vecLtoP` (i.e., "vector list to pair") first catches the empty list and returns a binary vector of no length. If the list is non-empty, the order of the vector is computed by taking the length of the list. Then the binary vector is constructed by passing the normalized scalars to `pairup`.

10c ⟨*Vector conversion functions* 10c⟩≡

```
vecLtoP :: (Eq a, Num a) => [a] -> Vector a
vecLtoP []  =  OVec 0 ZeroV
vecLtoP xs  =  OVec (length xs) (pairup (map normalizeSV xs))
                  where ⟨Vector conversion support functions 11a⟩
```

Defines:
  `vecLtoP`, never used.
Uses `OVec` 10a, `Vector` 10a, `ZeroV` 8a, `normalizeSV` 8b, and `pairup` 11a.
This code is used in chunk 7.

The division of a list into a binary vector is done bottom up. The function `pairup` monitors the process of pairing north and south halves together. When it finds a single vector in the list passed to it, processing is finished. Otherwise, the vectors in the list are paired off by `pair_off_list`, stepping up one level in the binary vector.

11a ⟨*Vector conversion support functions* 11a⟩≡

```
pairup :: [Vectr a] -> Vectr a
pairup [x]  =  x
pairup xs   =  pairup (pair_off_list xs)
```

Defines:
`pairup`, used in chunk 10c.
Uses `Vectr` 8a and `pair_off_list` 11b.
This definition is continued in chunk 11b.
This code is used in chunk 10c.

The function `pair_off_list` pairs off two vectors at a time. Single vectors at the end of the list are paired off with a `ZeroV` which adds zero padding to the end of the vector.

11b ⟨*Vector conversion support functions* 11a⟩+≡

```
pair_off_list :: [Vectr a] -> [Vectr a]
pair_off_list []        =  []
pair_off_list (x:[])    =  [normalizePV (x, ZeroV)]
pair_off_list (x:y:xs)  =  normalizePV (x, y) : pair_off_list xs
```

Defines:
`pair_off_list`, used in chunk 11a.
Uses `Vectr` 8a, `ZeroV` 8a, and `normalizePV` 8b.

3 Quadtree Matrix Data Definitions

The quadtree representation for matrices is examined in [4, Section 1.1]. The definition of a quadtree matrix is an extension of the definition of a binary vector. The matrix is divided into four quadrants where each quadrant has the same order. The four quadrants are stored in one quadtree.

For example, the matrix

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

would be stored as $(Z, 1, 2, 3)$. The matrix

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

is represented by

$$((Z, 1, 4, 5), (2, 3, 6, 7), (8, 9, 12, 13), (10, 11, 14, 15)).$$

In general, a matrix is broken into northwest, northeast, southwest, and southeast quadrants like so:

$$\begin{bmatrix} A_{nw} & A_{ne} \\ A_{sw} & A_{se} \end{bmatrix}$$

where each of the quadrants are of the same size. To store this in a quadtree, each quadrant is recursively defined as quadtrees A'_{nw}, A'_{ne}, A'_{sw}, and A'_{se} and then assembled into the quadtree $(A'_{nw}, A'_{ne}, A'_{sw}, A'_{se})$.

As with vectors, this recursive structure requires that the order of each matrix be a power of two which does not hold true for most matrices. The solution used for binary vectors also works for quadtree matrices: pad the original matrix to the next power of two. Again, the zero matrix used for the padding is a sentinel value which has no implicit order which allows for compression of the zero values.

3.1 File Layout

12 ⟨`reg-qmtx.gs` 12⟩≡
  ⟨*Quadtree matrix definition* 13a⟩
  ⟨*Quadtree matrix inheritance* 14a⟩
  ⟨*Quadtree matrix normalization* 13b⟩
  ⟨*Top level quadtree matrix definition* 16a⟩
  ⟨*Top level quadtree matrix inheritance* 16b⟩
  ⟨*Row major to quadtree matrix conversion* 17a⟩
  ⟨*Quadtree to row major matrix conversion* 18b⟩
Root chunk (not used in this document).

3.2 Quadtree Matrix Data Definition

The zero matrix is represented by `ZeroM`. The `IdentM` constructor is used for a single quadrant that is the identity matrix (i.e., the zero matrix with units down the diagonal). The identity matrix is not really intended for arbitrary use, just in special cases like internal padding in matrix inversion. The `ScalarM` constructor builds a scalar matrix while `Mtx` employs the quadtree to store the four quadrants of general matrix.

13a ⟨*Quadtree matrix definition* 13a⟩≡

```
type QuadM a  =  Quad (Matrx a)
data Matrx a  =  ZeroM | IdentM | ScalarM a | Mtx (QuadM a)
```

Defines:
`IdentM`, used in chunks 13–15, 34, 37c, and 38a.
`Matrx`, used in chunks 13–18, 26c, 27a, 33, 34, and 36–46.
`Mtx`, used in chunks 13–15, 18c, 26c, 27a, 33, 34, 37c, and 44b.
`QuadM`, used in chunk 13.
`ScalarM`, used in chunks 13–15, 18c, 26c, 27a, 33, 36, 43a, and 44a.
`ZeroM`, used in chunks 13–15, 17, 18, 26c, 27a, 33, 34, 37–39, 43a, 44b, and 46.
Uses `Quad` 3c.
This code is used in chunk 12.

3.3 Quadtree Matrix Normalizing and Unnormalizing Functions

The normalizing functions here are straightforward, the basic idea being to compress zeros as much as possible inside the structure.

13b ⟨*Quadtree matrix normalization* 13b⟩≡

```
normalizeSM :: (Eq a, Num a) => a -> Matrx a
normalizeSM x | x == fromInteger 0   =  ZeroM
              | otherwise            =  ScalarM x

normalizeQM :: QuadM a -> Matrx a
normalizeQM (ZeroM, ZeroM, ZeroM, ZeroM)    =  ZeroM
normalizeQM (IdentM, ZeroM, ZeroM, IdentM)  =  IdentM
normalizeQM q                               =  Mtx q
```

Defines:
`normalizeQM`, used in chunks 15, 18a, 33a, 38a, 39a, 41, 42, 45, and 46.
`normalizeSM`, used in chunks 15, 17a, and 33a.
Uses `IdentM` 13a, `Matrx` 13a, `Mtx` 13a, `QuadM` 13a, `ScalarM` 13a, and `ZeroM` 13a.
This definition is continued in chunk 13c.
This code is used in chunk 12.

The `unnormalizeM` function does three special operations. For a zero matrix, `unnormalizeM` returns a quadtree filled with zero matrices. An identity matrix is unnormalized into its equivalent four quadrants. The function also expands a scalar matrix $[n]$ to

$$\begin{bmatrix} n & 0 \\ 0 & n \end{bmatrix}.$$

13c ⟨*Quadtree matrix normalization* 13b⟩+≡

```
unnormalizeM :: Matrx a -> QuadM a
unnormalizeM ZeroM          =  (ZeroM, ZeroM, ZeroM, ZeroM)
unnormalizeM IdentM         =  (IdentM, ZeroM, ZeroM, IdentM)
unnormalizeM s@(ScalarM x)  =  (s, ZeroM, ZeroM, s)
unnormalizeM (Mtx q)        =  q
```

Defines:
`unnormalizeM`, used in chunks 34, 38a, 39a, 41, 42, 45, and 46.
Uses `IdentM` 13a, `Matrx` 13a, `Mtx` 13a, `QuadM` 13a, `ScalarM` 13a, and `ZeroM` 13a.

3.4 Quadtree Matrix Inheritance

Quadtree matrices inherit from three classes: `Eq`, `Text` and `Num`. The definitions are straightforward.

14a ⟨*Quadtree matrix inheritance* 14a⟩≡
  ⟨*Quadtree matrix inheritance of* `Eq` 14b⟩
  ⟨*Quadtree matrix inheritance of* `Text` 14c⟩
  ⟨*Quadtree matrix inheritance of* `Num` 15⟩

This code is used in chunk 12.

Equality breaks down in the obvious way. Note the use of `andmap2`.

14b ⟨*Quadtree matrix inheritance of* `Eq` 14b⟩≡

```
instance Eq a => Eq (Matrx a) where
   ZeroM      == ZeroM      =  True
   IdentM     == IdentM     =  True
   ScalarM x  == ScalarM y  =  x == y
   Mtx q0     == Mtx q1     =  andqmap2 (==) q0 q1
   _          == _          =  False
```

Uses `IdentM` 13a, `Matrx` 13a, `Mtx` 13a, `ScalarM` 13a, `ZeroM` 13a, and `andqmap2` 5b.
This code is used in chunk 14a.

`Text` inheritance is again just a formality so that the Gofer type checker does not complain.

14c ⟨*Quadtree matrix inheritance of* `Text` 14c⟩≡

```
instance Text a => Text (Matrx a)
```

Uses `Matrx` 13a.
This code is used in chunk 14a.

The heart of matrix inheritance comes in the `Num` class. For the most part, the definitions here are straightforward and follow the definitions in [4, Section 1.4]. Note that addition and subtraction are not defined for `IdentM`. The identity matrix is intended only for multiplication.

The definitions for addition and subtraction are straightforward and lend themselves immediately to a mapping recursion:

$$\begin{bmatrix} A_{nw} & A_{ne} \\ A_{sw} & A_{se} \end{bmatrix} \pm \begin{bmatrix} B_{nw} & B_{ne} \\ B_{sw} & B_{se} \end{bmatrix} = \begin{bmatrix} A_{nw} \pm B_{nw} & A_{ne} \pm B_{ne} \\ A_{sw} \pm B_{sw} & A_{se} \pm B_{se} \end{bmatrix}.$$

Multiplication also lends itself to a mapping recursion, after the matrices have been rearranged:

$$\begin{bmatrix} A_{nw} & A_{ne} \\ A_{sw} & A_{se} \end{bmatrix} \cdot \begin{bmatrix} B_{nw} & B_{ne} \\ B_{sw} & B_{se} \end{bmatrix} = \begin{bmatrix} A_{ne} \cdot B_{sw} + A_{nw} \cdot B_{nw} & A_{nw} \cdot B_{ne} + A_{ne} \cdot B_{se} \\ A_{se} \cdot B_{sw} + A_{sw} \cdot B_{nw} & A_{sw} \cdot B_{ne} + A_{se} \cdot B_{se} \end{bmatrix}.$$

Normalization is done at every opportunity, including multiplication to catch infinitesimal floating-point numbers. Note that the addition in the definition of multiplication takes care of normalization.

15 ⟨*Quadtree matrix inheritance of* `Num` 15⟩≡
```
instance Num a => Num (Matrx a) where
  fromInteger 0          =  ZeroM
  fromInteger n          =  ScalarM (fromInteger n)
  x         + ZeroM      =  x
  ZeroM     + y          =  y
  IdentM    + _          =  error "Adding IdentM is not allowed."
  _         + IdentM     =  error "Adding IdentM is not allowed."
  ScalarM x + ScalarM y  =  normalizeSM (x + y)
  Mtx q0    + Mtx q1     =  normalizeQM (qmap2 (+) q0 q1)
  x         - ZeroM      =  x
  ZeroM     - y          =  negate y
  IdentM    - _          =  error "Subtracting IdentM is not allowed."
  _         - IdentM     =  error "Subtracting IdentM is not allowed."
  ScalarM x - ScalarM y  =  normalizeSM (x - y)
  Mtx q0    - Mtx q1     =  normalizeQM (qmap2 (-) q0 q1)
  _         * ZeroM      =  ZeroM
  ZeroM     * _          =  ZeroM
  x         * IdentM     =  x
  IdentM    * y          =  y
  ScalarM x * ScalarM y  =  normalizeSM (x * y)
  Mtx x     * Mtx y      =  Mtx (qmap2 (*) (col_exchangeQ x) (off_squashQ y))
                            +
                            Mtx (qmap2 (*) x                (prim_squashQ y))
  negate ZeroM           =  ZeroM
  negate (ScalarM x)     =  ScalarM (negate x)
  negate (Mtx q)         =  Mtx (qmap negate q)
```
Uses `IdentM` 13a, `Matrx` 13a, `Mtx` 13a, `ScalarM` 13a, `ZeroM` 13a, `col_exchangeQ` 6, `normalizeQM` 13b, `normalizeSM` 13b, `off_squashQ` 6, `prim_squashQ` 6, `qmap` 5a, and `qmap2` 5a.
This code is used in chunk 14a.

3.5 Top Level Quadtree Matrix

It is necessary to keep track of the actual order of a matrix to differentiate between the actual matrix from its padding. Thus, a top level matrix is used to carry around an explicit order with a quadtree matrix. In most matrix algorithms, the top level matrix is ignored. If necessary, the order is passed around as a parameter.

3.5.1 Top Level Quadtree Matrix Data Definition

Similarly to the top level binary vector, a top level quadtree matrix carries around a quadtree matrix and its order.

16a ⟨*Top level quadtree matrix definition* 16a⟩≡

```
data Matrix a = OMtx Int (Matrx a)
```

Defines:
`Matrix`, used in chunks 16–18 and 43.
`OMtx`, used in chunks 16–18 and 43.
Uses `Matrx` 13a.
This code is used in chunk 12.

3.5.2 Top Level Quadtree Matrix Inheritance

Inheritance of `Eq`, `Text` and `Num` are simple, based mainly on the inheritance of the same classes for `Matrx`.

16b ⟨*Top level quadtree matrix inheritance* 16b⟩≡
⟨*Top level quadtree matrix inheritance of* `Eq` 16c⟩
⟨*Top level quadtree matrix inheritance of* `Text` 16d⟩
⟨*Top level quadtree matrix inheritance of* `Num` 16e⟩
This code is used in chunk 12.

Equality is true for two top level matrices if their orders are the same and their matrices are the same.

16c ⟨*Top level quadtree matrix inheritance of* `Eq` 16c⟩≡

```
instance (Eq a, Eq (Matrx a)) => Eq (Matrix a) where
  (OMtx o m) == (OMtx o' m')  =  (o == o') && (m == m')
```

Uses `Matrix` 16a, `Matrx` 13a, and `OMtx` 16a.
This code is used in chunk 16b.

The `Text` inheritance is provided so that Gofer can carry out its type checking. The conversion functions defined later in this section could be included with this definition.

16d ⟨*Top level quadtree matrix inheritance of* `Text` 16d⟩≡

```
instance (Text a, Text (Matrx a)) => Text (Matrix a)
```

Uses `Matrix` 16a and `Matrx` 13a.
This code is used in chunk 16b.

For a `Num` operation on top level matrices, the corresponding operation for a quadtree matrix are used. When operating on two matrices, they must be compatible (i.e., their orders are the same). If they are not, an error message halts the computation.

16e ⟨*Top level quadtree matrix inheritance of* `Num` 16e⟩≡

```
instance (Num a, Num (Matrx a)) => Num (Matrix a) where
  OMtx o m + OMtx o' m'
    | o == o'     =  OMtx o (m+m')
    | otherwise   =  error "Matrices are not compatible (+)."
  OMtx o m - OMtx o' m'
    | o == o'     =  OMtx o (m-m')
    | otherwise   =  error "Matrices are not compatible (-)."
  OMtx o m * OMtx o' m'
    | o == o'     =  OMtx o (m*m')
    | otherwise   =  error "Matrices are not compatible (*)."
  negate (OMtx o m)  =  OMtx o (negate m)
```

Uses `Matrix` 16a, `Matrx` 13a, and `OMtx` 16a.
This code is used in chunk 16b.

3.6 Matrix Conversion

For input and output purposes, we define functions that translate from a row major matrix to quadtree matrix and vice versa. These functions could be extended to provide conversion to and from Haskell arrays or any (e.g. stream-oriented) I/O protocol.

3.6.1 Row Major Matrix to Quadtree Matrix

The function for row major to quadtree conversion is defined first. This conversion is very similar to the conversion from a list to binary vector. (See Section 2.6.)

In the case of an empty list, a top level quadtree matrix is returned with order 0. Otherwise, the rows in the row major matrix are counted, and this number is taken to be the order of the matrix. After normalizing scalars in the row major matrix, `quadup` is called upon to make the quadtree matrix.

17a ⟨*Row major to quadtree matrix conversion* 17a⟩≡

```
mtxRMtoQ :: (Eq a, Num a) => [[a]] -> Matrix a
mtxRMtoQ []  =  OMtx 0 ZeroM
mtxRMtoQ xs  =  OMtx (length xs) (quadup (map (map normalizeSM) xs))
                    where ⟨Row major to quadtree matrix support functions 17b⟩
```

Defines:
`mtxRMtoQ`, never used.
Uses `Matrix` 16a, `OMtx` 16a, `ZeroM` 13a, `normalizeSM` 13b, and `quadup` 17b.
This code is used in chunk 12.

The function `quadup` makes the test to discover when enough passes have been made over the matrix to give us one single quadtree matrix. If the list of lists passed to `quadup` contains a single element, that is the final quadtree matrix. Otherwise, the matrices in the list of lists is processed by `quad_off_list`.

17b ⟨*Row major to quadtree matrix support functions* 17b⟩≡

```
quadup :: [[Matrx a]] -> Matrx a
quadup [[x]]  =  x
quadup xs     =  quadup (quad_off_list xs)
```

Defines:
`quadup`, used in chunk 17a.
Uses `Matrx` 13a and `quad_off_list` 17c.
This definition is continued in chunks 17c and 18a.
This code is used in chunk 17a.

The function `quad_off_list` takes two rows and passes them to `quad_two_rows`.

17c ⟨*Row major to quadtree matrix support functions* 17b⟩+≡

```
quad_off_list :: [[Matrx a]] -> [[Matrx a]]
quad_off_list []         =  []
quad_off_list [x]        =  [quad_two_rows x []]
quad_off_list [x,y]      =  [quad_two_rows x y]
quad_off_list (x:y:xs)   =  quad_two_rows x y : quad_off_list xs
```

Defines:
`quad_off_list`, used in chunk 17b.
Uses `Matrx` 13a and `quad_two_rows` 18a.

The function quad_two_rows takes two elements at a time from both rows and normalizes them together into a quadtree matrix. Padding is added when quad_two_rows comes up with only one element in both lists (padding to the east) or when the second list is consistently empty (padding to the south).

18a ⟨*Row major to quadtree matrix support functions* 17b⟩+≡

```
quad_two_rows :: [Matrx a] -> [Matrx a] -> [Matrx a]
quad_two_rows []  []               =  []
quad_two_rows [x] []               =  [normalizeQM (x, ZeroM, ZeroM, ZeroM)]
quad_two_rows (x:x':xs) []         =
  normalizeQM (x, x', ZeroM, ZeroM) : quad_two_rows xs []
quad_two_rows [x] [y]              =  [normalizeQM (x, ZeroM, y, ZeroM)]
quad_two_rows (x:x':xs) (y:y':ys)  =
  normalizeQM (x, x', y, y') : quad_two_rows xs ys
```

Defines:
quad_two_rows, used in chunk 17c.
Uses Matrx 13a, ZeroM 13a, and normalizeQM 13b.

3.6.2 Quadtree Matrix to Row Major Matrix

Converting from quadtree matrices to row-major matrices is a much simpler process.

The top level function mtxQtoRM calls the workhorses mgrab and q2rm. The function q2rm (i.e., "quad to row major") makes the conversion from quadtree to row major form while mgrab worries about the order of the actual matrix.

18b ⟨*Quadtree to row major matrix conversion* 18b⟩≡

```
mtxQtoRM :: (Eq a, Num a) => Matrix a -> [[a]]
mtxQtoRM (OMtx o q)  =  mgrab o (q2rm (power2 (ceil_lg o)) q)
                                where ⟨Quadtree to row major support functions 18c⟩
```

Defines:
mtxQtoRM, never used.
Uses Matrix 16a, OMtx 16a, ceil_lg 48b, mgrab 19b, power2 48b, and q2rm 18c.
This code is used in chunk 12.

The function q2rm returns the actual row major matrix, but with all of the padding still in place. This function behaves recursively:

- In the ZeroM base case, generate_zero_rm generates a row major zero matrix of the needed size.
- A scalar matrix is returned for a ScalarM.
- A quadtree matrix requires mapping q2rm over the quadrants. The results of the mapping are appended to each other appropriately.

18c ⟨*Quadtree to row major support functions* 18c⟩≡

```
q2rm :: (Num a) => Int -> Matrx a -> [[a]]
q2rm o ZeroM         =  generate_zero_rm o
q2rm o (ScalarM x)   =  [[x]]
q2rm o (Mtx q)       =
  let  (nw', ne', sw', se')  =  qmap (q2rm (halve o)) q  in
     (zipWith (++) nw' ne') ++ (zipWith (++) sw' se')
```

Defines:
q2rm, used in chunk 18b.
Uses Matrx 13a, Mtx 13a, ScalarM 13a, ZeroM 13a, generate_zero_rm 19a, halve 49a, and qmap 5a.
This definition is continued in chunk 19.
This code is used in chunk 18b.

The function `generate_zero_rm` generates a row major zero matrix of any given size.

19a ⟨*Quadtree to row major support functions* 18c⟩+≡
```
generate_zero_rm :: (Num a) => Int -> [[a]]
generate_zero_rm o  =  iterate (iterate (fromInteger 0) o) o
  where  iterate _ 0  =  []
         iterate x n  =  x : (iterate x (n-1))
```
Defines:
`generate_zero_rm`, used in chunk 18c.

The padded row major matrix and the order of the actual matrix are passed to `mgrab` to dispose of the padding. Suppose *o* is the order of the original matrix. Then the function `mgrab` takes *o* rows from the padded row major matrix and *o* elements from each row. This way the actual matrix is returned without padding.

Note that `mgrab` and lazy evaluation in Gofer prevent this conversion from being too wasteful of memory. Whenever a `ZeroM` is expanded, it is not *really* expanded until `mgrab` forces the evaluation. Thus the padding is not expanded since `mgrab`, by design, ignores it.

19b ⟨*Quadtree to row major support functions* 18c⟩+≡
```
mgrab :: Int -> [[a]] -> [[a]]
mgrab 0 _  =  [[]]
mgrab o x  =  ((map (take o)) . (take o)) x
```
Defines:
`mgrab`, used in chunk 18b.

4 Permutation Quadtrees

Permutation matrices are necessary in the inversion algorithm to create the triangulation from the decomposition. Gaussian pivoting is supposed to follow along the diagonal of the matrix, but this path is not always possible or advisable. For example, elimination cannot occur on a zero matrix, and, for numeric stability, pivoting should eliminate a block with a large determinant. By permuting the matrix, we can migrate any desired elimination blocks onto the main diagonal.

Actually moving the blocks onto the main diagonal is the task of the series encodings explained in Appendix C. This section deals with the permutation quadtree matrix itself. The basic permutation matrix consists of zero and identity matrices such that exactly one identity matrix is in each column and row.

Permutation matrices are discussed by Wise [4, Section 1.3].

4.1 File Layout

20a ⟨`perm-qmtx.gs` 20a⟩≡
⟨*Quadtree permutation matrix definition* 20b⟩
⟨*Quadtree permutation matrix normalization* 20c⟩
⟨*Quadtree permutation matrix inheritance* 21a⟩

Root chunk (not used in this document).

4.2 Quadtree Permutation Matrix Data Definition

The definition of the permutation quadtree matrix is similar to the other quadtree matrices, except that there are two base cases, `ZeroPM` and `IdentPM`, which take no external information. The `ZeroPM` constructor serves the same function as `ZeroM` and `ZeroDM`: it represents a zero matrix of arbitrary order. `IdentPM` serves a similar purpose: it represents an identity matrix of arbitrary order.

20b ⟨*Quadtree permutation matrix definition* 20b⟩≡
```
type QuadPM  =  Quad PMatrx
data PMatrx = ZeroPM | IdentPM | PMtx QuadPM
```
Defines:
`IdentPM`, used in chunks 20c, 21b, 34, 36, and 54.
`PMatrx`, used in chunks 20c, 21b, 34, 36, and 54.
`PMtx`, used in chunks 20c and 34.
`QaudPM`, never used.
`ZeroPM`, used in chunks 20c, 21b, 34, and 54.

Uses `Quad` 3c.
This code is used in chunk 20a.

Normalization has a interesting twist this time: in addition to the (Z, Z, Z, Z) normalization to Z, the matrix (I, Z, Z, I) normalizes to I. Unnormalizing a matrix is simply the inverse of this process.

20c ⟨*Quadtree permutation matrix normalization* 20c⟩≡
```
normalizeQPM :: QuadPM -> PMatrx
normalizeQPM (ZeroPM,  ZeroPM, ZeroPM, ZeroPM)   =  ZeroPM
normalizeQPM (IdentPM, ZeroPM, ZeroPM, IdentPM)  =  IdentPM
normalizeQPM q                                   =  PMtx q

unnormalizePMQ :: PMatrx -> QuadPM
unnormalizePMQ ZeroPM    =  (ZeroPM, ZeroPM, ZeroPM, ZeroPM)
unnormalizePMQ IdentPM   =  (IdentPM, ZeroPM, ZeroPM, IdentPM)
unnormalizePMQ (PMtx q)  =  q
```
Defines:
`normalizeQPM`, used in chunk 54.
`unnormalizePMQ`, never used.

Uses `IdentPM` 20b, `PMatrx` 20b, `PMtx` 20b, and `ZeroPM` 20b.
This code is used in chunk 20a.

4.3 Quadtree Permutation Matrix Inheritance

The only class that permutation matrices need for matrix inversion is the `Eq` class. Permutation arithmetic is never homogeneous in the matrix inversion algorithm, so inheritance from the `Num` class is not necessary. Permutation and regular matrix operations are defined in Section 6.4.

21a ⟨*Quadtree permutation matrix inheritance* 21a⟩≡
  ⟨*Quadtree permutation matrix inheritance of* `Eq` 21b⟩

This code is used in chunk 20a.

Equality is defined as one expects.

21b ⟨*Quadtree permutation matrix inheritance of* `Eq` 21b⟩≡

```
instance Eq PMatrx where
  ZeroPM  == ZeroPM   =  True
  IdentPM == IdentPM  =  True
  _       == _        =  False
```

Uses `IdentPM` 20b, `PMatrx` 20b, and `ZeroPM` 20b.
This code is used in chunk 21a.

5 Decorated Quadtree Matrix

Decorated quadtree matrices are regular quadtree matrices with extra information indicating which block is the best to use for pivoting during matrix inversion. This extra information is called decoration.

Decoration contains a *signpost* which provides two pieces of information: a depth and a pointer toward the nested pivot block. The depth expresses how many levels down the pivot block is located. The location denotes which quadrant the pivot block is in.

The non-signpost portion of the decoration is used to compute future decorations. The code here uses decorations to preserve stability as described in [4, Section 3.3.1]. The matrix is searched for *nown-singular* blocks. A quadtree matrix is nown-singular if it is a non-zero scalar, a 2×2 nonsingular matrix, or, in general, if one of the four quadrants is 0 and the two adjacent quadrants are nown-singular.

The non-signpost portion of the decoration consists of two *known-determinants*, one for the pivot block inside the matrix and a second for the matrix itself. A known-determinant is zero for all matrices except for nown-singular matrices. For a nown-singular matrix, the known-determinant is the magnitude (i.e., logarithm) of the predicted determinant of the matrix.

5.1 File Layout

22a ⟨`dec-qmtx.gs` 22a⟩≡
  ⟨*Decoration data definition* 22b⟩
  ⟨*Decoration inheritance* 23c⟩
  ⟨*Decorated quadtree matrix definition* 24b⟩
  ⟨*Decorated quadtree matrix inheritance* 24c⟩
  ⟨*Decorated quadtree matrix normalization* 26a⟩
  ⟨*Special decoration functions* 24a⟩
  ⟨*Undecorating a decorated matrix* 26c⟩
  ⟨*Decoration makers* 27a⟩

Root chunk (not used in this document).

5.2 Matrix Decoration

5.2.1 Decoration Definition

The signpost of the decoration is defined by the data type `Signpost` which consists of a position and depth. The `Pos` data type defines the four compass points (i.e., northwest, northeast, southwest, and southeast) for matrix quadrants to give the position of the nested pivot block. (In [4, Section 3.3], `Pos` is represented by two boolean values.) The depth is represented by an integer.

22b ⟨*Decoration data definition* 22b⟩≡

```
data Pos = NW | NE | SW | SE
type SignPost = (Pos, Int)
```

Defines:
  `NE`, used in chunks 23e, 27b, and 39a.
  `NW`, used in chunks 23e, 27b, and 39a.
  `Pos`, used in chunks 23e, 28b, and 29b.
  `SE`, used in chunks 23e, 27b, and 39a.
  `SW`, used in chunks 23e, 27b, and 39a.
  `SignPost`, used in chunks 23b and 24a.

This definition is continued in chunk 23.
This code is used in chunk 22a.

The `Det` data type is introduced for the known-determinants so that the worst possible determinant can be easily tagged, using `WorstDet`. If there is a usable known-determinant, the logarithm of the known-determinant is saved using the `Detm` constructor.

23a ⟨*Decoration data definition* 22b⟩+≡
```
data Det = WorstDet | Detm Int
```
Defines:
`Det`, used in chunks 23, 24a, 28, and 29.
`Detm`, used in chunks 23d and 28–30.
`WorstDec`, used in chunks 26b, 29b, and 30.

A decoration is either a signpost with two known-determinants (the known-determinant of the nested pivot block and the known-determinant of the full quadtree) or the worst decoration possible.

23b ⟨*Decoration data definition* 22b⟩+≡
```
data Dec = WorstDec | Deco (SignPost, Det, Det)
```
Defines:
`Dec`, used in chunks 24b and 28–30.
`Deco`, used in chunks 24a and 28–30.
`WorstDec`, used in chunks 26b, 29b, and 30.
Uses `Det` 23a and `SignPost` 22b.

5.2.2 Decoration Inheritance

For pattern matching purposes, the inheritance of `Eq` is defined for `Det` and `Dec`.

23c ⟨*Decoration inheritance* 23c⟩≡
```
⟨Determinant inheritance of Eq 23d⟩
⟨Decoration inheritance of Eq 23e⟩
```
This code is used in chunk 22a.

The inheritance of `Eq` for `Det` is not surprising.

23d ⟨*Determinant inheritance of* `Eq` 23d⟩≡
```
instance Eq Det where
  WorstDet == WorstDet  =  True
  (Detm x) == (Detm y)  =  x == y
  _        == _         =  False
```
Uses `Det` 23a and `Detm` 23a.
This code is used in chunk 23c.

The inheritance of `Eq` for `Pos` is also not surprising.

23e ⟨*Decoration inheritance of* `Eq` 23e⟩≡
```
instance Eq Pos where
  NW == NW  =  True
  NE == NE  =  True
  SW == SW  =  True
  SE == SE  =  True
  _  == _   =  False
```
Uses `NE` 22b, `NW` 22b, `Pos` 22b, `SE` 22b, and `SW` 22b.
This code is used in chunk 23c.

5.2.3 Special Decoration Functions

Since the decorations are likely to change with pivoting strategy, a special deconstructor `signpost` returns the signpost of the decoration of a decorated matrix. The non-signpost decoration can be ignored in algorithms where only the signpost matters.

The function `ln_known_det` retrieves the logarithm of the known-determinant of the matrix out of the decoration (the second determinant in the non-signpost decoration).

24a ⟨*Special decoration functions* 24a⟩≡
```
signpost :: DMatrx a -> SignPost
signpost (DMtx (Deco (sp, _, _)) _)  =  sp

ln_known_det :: DMatrx a -> Det
ln_known_det (DMtx (Deco (_, _, det)) _)  =  det
```
Defines:
`ln_known_det`, used in chunks 29a and 31.
`signpost`, used in chunks 38c and 39a.
Uses `DMatrx` 24b, `DMtx` 24b, `Deco` 23b, `Det` 23a, and `SignPost` 22b.
This code is used in chunk 22a.

5.3 Decorated Quadtree Matrix

5.3.1 Decorated Quadtree Matrix Data Definition

The decorated matrix is a regular quadtree matrix with some extra information. Thus, the definition of a decorated quadtree matrix follows very closely to the definition of a regular quadtree matrix. (See Section 3.2.)

24b ⟨*Decorated quadtree matrix definition* 24b⟩≡
```
type A = Float
type QuadDM a  =  Quad (DMatrx a)
data DMatrx a  =  ZeroDM | ScalarDM a | DMtx Dec (QuadDM a)
```
Defines:
`DMatrx`, used in chunks 24–29, 31, 33, 36, and 38–44.
`DMtx`, used in chunks 24–27, 29b, 33, and 39–42.
`QuadDM`, used in chunks 26–28.
`ScalarDM`, used in chunks 25–28, 31, 33, 36, and 44a.
`ZeroDM`, used in chunks 25–29, 31, 33, 38–42, and 44a.
Uses `Dec` 23b and `Quad` 3c.
This code is used in chunk 22a.

Note the `type A = Float` type definition. This is necessary because of the logarithm in the function `det_lg` defined later in this section which is used in computing determinants.

In a full Haskell system, this logarithm function could be defined for the `Fractional` class, which is actually the class for which the matrix inversion algorithms in this paper works. However, Gofer does not implement the full range of numeric data types available in Haskell. Instead of implementing a `Fractional` class or switch to a full Haskell system, this type definition has been made.

5.3.2 Decorated Quadtree Matrix Inheritance

Decorated quadtree matrix inheritance is the same as regular quadtree matrix inheritance. (See Section 3.4.)

24c ⟨*Decorated quadtree matrix inheritance* 24c⟩≡
  ⟨*Decorated quadtree matrix inheritance of* `Eq` 25a⟩
  ⟨*Decorated quadtree matrix inheritance of* `Text` 25b⟩
  ⟨*Decorated quadtree matrix inheritance of* `Num` 25c⟩
This code is used in chunk 22a.

Equality is straight forward.

25a ⟨*Decorated quadtree matrix inheritance of* `Eq` 25a⟩≡
```
instance Eq a => Eq (DMatrx a) where
   ZeroDM     == ZeroDM      =  True
   ScalarDM x == ScalarDM y  =  x == y
   DMtx _ q   == DMtx _ q'   =  andqmap2 (==) q q'
   _          == _           =  False
```
Uses `DMatrx` 24b, `DMtx` 24b, `ScalarDM` 24b, `ZeroDM` 24b, and `andqmap2` 5b.
This code is used in chunk 24c.

Again, `Text` is required for type checking in Gofer.

25b ⟨*Decorated quadtree matrix inheritance of* `Text` 25b⟩≡
```
instance Text a => Text (DMatrx a)
```
Uses `DMatrx` 24b.
This code is used in chunk 24c.

The `Num` class inheritance is straightforward, following the same pattern as for regular quadtree matrices except that quadtrees are redecorated, not just simply normalized.

25c ⟨*Decorated quadtree matrix inheritance of* `Num` 25c⟩≡
```
instance Num Float => Num (DMatrx Float) where
  fromInteger 0            =  ZeroDM
  fromInteger n            =  ScalarDM (fromInteger n)
  x          + ZeroDM      =  x
  ZeroDM     + y           =  y
  ScalarDM x + ScalarDM y  =  normalizeSDM (x + y)
  DMtx _ q   + DMtx _ q'   =  normalizeQDM (qmap2 (+) q q')
  x          - ZeroDM      =  x
  ZeroDM     - y           =  negate y
  ScalarDM x - ScalarDM y  =  normalizeSDM (x - y)
  DMtx _ q   - DMtx _ q'   =  normalizeQDM (qmap2 (-) q q')
  x          * ZeroDM      =  ZeroDM
  ZeroDM     * y           =  ZeroDM
  ScalarDM x * ScalarDM y  =  normalizeSDM (x * y)
  DMtx _ q   * DMtx _ q'   =
    normalizeQDM (qmap2 (+) (qmap2 (*) (col_exchangeQ q) (off_squashQ q'))
                            (qmap2 (*) q                 (prim_squashQ q')))
  negate ZeroDM            =  ZeroDM
  negate (ScalarDM x)      =  ScalarDM (negate x)
  negate (DMtx dec q)      =  DMtx dec (qmap negate q)
```
Uses `DMatrx` 24b, `DMtx` 24b, `ScalarDM` 24b, `ZeroDM` 24b, `col_exchangeQ` 6, `normalizeQDM` 26a, `normalizeSDM` 26a, `off_squashQ` 6, `prim_squashQ` 6, `qmap` 5a, and `qmap2` 5a.
This code is used in chunk 24c.

5.3.3 Decorated Quadtree Matrix Normalization

In addition to compressing zero matrices (see Section 3.3), normalization of decorated matrices must also decorate the new quadtree matrix. Decoration is taken care of by `decorateDQ`.

26a ⟨*Decorated quadtree matrix normalization* 26a⟩≡
```
normalizeSDM :: (Eq a, Num a) => a -> DMatrx a
normalizeSDM x | x == fromInteger 0  =  ZeroDM
               | otherwise           =  ScalarDM x

normalizeQDM :: QuadDM A -> DMatrx A
normalizeQDM (ZeroDM, ZeroDM, ZeroDM, ZeroDM)  =  ZeroDM
normalizeQDM q                                 =  decorateDQ q
```
Defines:
`normalizeQDM`, used in chunks 25c and 33b.
`normalizeSDM`, used in chunks 25c and 33b.
Uses `DMatrx` 24b, `QuadDM` 24b, `ScalarDM` 24b, `ZeroDM` 24b, and `decorateDQ` 27b.
This definition is continued in chunk 26b.
This code is used in chunk 22a.

Unnormalizing a decorated matrix is the same as unnormalizing a regular matrix. For a quadtree matrix where every quadrant is either a zero matrix or a scalar, `unnormalizeDMS` is used to extract their scalar values.

26b ⟨*Decorated quadtree matrix normalization* 26a⟩+≡
```
unnormalizeDMS :: (Num a) => DMatrx a -> a
unnormalizeDMS ZeroDM        =  fromInteger 0
unnormalizeDMS (ScalarDM x)  =  x

unnormalizeDM :: DMatrx a -> DMatrx a
unnormalizeDM ZeroDM          =  DMtx WorstDec (ZeroDM, ZeroDM, ZeroDM, ZeroDM)
unnormalizeDM s@(ScalarDM x)  =  DMtx WorstDec (s, ZeroDM, ZeroDM, s)
unnormalizeDM m               =  m
```
Defines:
`unnormalizeDM`, never used.
`unnormalizeDMS`, used in chunk 28a.
Uses `DMatrx` 24b, `DMtx` 24b, `ScalarDM` 24b, `WorstDec` 23a 23b, and `ZeroDM` 24b.

5.4 Decorating a Quadtree Matrix

In this section, we develop the tools to make the decoration for a quadtree matrix.

5.4.1 Undecorating a Decorated Quadtree Matrix

Converting a decorated quadtree matrix into a regular quadtree matrix is a trivial task.

26c ⟨*Undecorating a decorated matrix* 26c⟩≡
```
undecorateDM :: DMatrx a -> Matrx a
undecorateDM ZeroDM        =  ZeroM
undecorateDM (ScalarDM x)  =  ScalarM x
undecorateDM (DMtx _ q)    =  Mtx (qmap undecorateDM q)
```
Defines:
`undecorateDM`, used in chunk 33a.
Uses `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `Mtx` 13a, `ScalarDM` 24b, `ScalarM` 13a, `ZeroDM` 24b, `ZeroM` 13a, and `qmap` 5a.
This code is used in chunk 22a.

5.4.2 Decorating an Undecorated Quadtree Matrix

A regular quadtree matrix is turned into a decorated matrix by the function `decorateM`. It maps itself over the quadtree, using `decorateDQ` to put together the new decorated quadtree matrix.

27a ⟨*Decoration makers* 27a⟩≡
```
decorateM :: Matrx A -> DMatrx A
decorateM ZeroM        =  ZeroDM
decorateM (ScalarM s)  =  ScalarDM s
decorateM (Mtx quad)   =  decorateDQ (qmap decorateM quad)
```
Defines:
  `decorateM`, used in chunk 43.
Uses `DMatrx` 24b, `Matrx` 13a, `Mtx` 13a, `ScalarDM` 24b, `ScalarM` 13a, `ZeroDM` 24b, `ZeroM` 13a, `decorateDQ` 27b, and `qmap` 5a.
This definition is continued in chunk 27b.
This code is used in chunk 22a.

The function `decorateDQ` operates over a quadtree of decorated quadtree matrices. It breaks down in the following recursive manner:

- If each quadrant of the quadtree is `ZeroDM`, the quadtree is compressed to a single `ZeroDM`.
- If the four quadrants are all scalars (where zero matrices are considered to be scalars), the known-determinant of the 2×2 matrix is computed by `det_2x2` and placed into the decoration. The function `best` is used to pick the best pivot based on the decoration constructed by `dec_2x2` from each scalar.
- Otherwise, the quadtree is a general $n \times n$ matrix. The known-determinant of the quadtree is calculated by `det_nxn`, using the known-determinants of the quadrants. The best decoration is picked by `best` from candidates created by `dec_nxn`.

27b ⟨*Decoration makers* 27a⟩+≡
```
decorateDQ :: QuadDM A -> DMatrx A
decorateDQ (ZeroDM, ZeroDM, ZeroDM, ZeroDM)  =  ZeroDM
decorateDQ q@(nw, ne, sw, se)
  | andqmap is_scalarDM q  =
      let  known_ln_det  =  det_2x2 q
           new_dec  =  best [dec_2x2 nw NW known_ln_det,
                             dec_2x2 se SE known_ln_det,
                             dec_2x2 sw SW known_ln_det,
                             dec_2x2 ne NE known_ln_det]  in
        DMtx new_dec q
  | otherwise   =
      let  known_ln_det  =  det_nxn q
           new_dec  =  best [dec_nxn nw NW known_ln_det,
                             dec_nxn sw SW known_ln_det,
                             dec_nxn se SE known_ln_det,
                             dec_nxn ne NE known_ln_det]  in
        DMtx new_dec q
    where ⟨Two by two matrix decoration functions 28a⟩
          ⟨General matrix decoration functions 28c⟩
          ⟨Picking the best decoration 29d⟩
          ⟨Decoration helpers 29c⟩
```
Defines:
  `decorateDQ`, used in chunks 26a, 27a, and 39–42.
Uses `DMatrx` 24b, `DMtx` 24b, `NE` 22b, `NW` 22b, `QuadDM` 24b, `SE` 22b, `SW` 22b, `ZeroDM` 24b, `andqmap` 5b, `best` 29d, `dec_nxn` 29b, `dec_2x2` 28b, `det_nxn` 28c, `det_2x2` 28a, and `is_scalarDM` 31.

The known-determinant computed by `det_2x2` is the logarithm of the standard definition of a determinant:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc.$$

If $ad = bc$, then the determinant is zero, the worst possible. So `WorstDet` is returned.

Note that `unnormalizeDMS` returns the scalar value from a `ScalarDM` and a `0` for a `ZeroDM` — scalar values, appropriate for multiplication.

28a ⟨*Two by two matrix decoration functions* 28a⟩≡

```
det_2x2 :: QuadDM A -> Det
det_2x2 q =
  let  (nwS, neS, swS, seS)  =  qmap unnormalizeDMS q
       maindiag  =  nwS * seS
       offdiag   =  swS * neS  in
    if (maindiag == offdiag)
    then WorstDet
    else let  d_lg  =  det_lg (maindiag - offdiag)  in
           Detm ((1 + d_lg) / 2)
```

Defines:
`det_2x2`, used in chunk 27b.
Uses `Det` 23a, `Detm` 23a, `QuadDM` 24b, `det_lg` 29c, `qmap` 5a, and `unnormalizeDMS` 26b.
This definition is continued in chunk 28b.
This code is used in chunk 27b.

The function `dec_2x2` computes the decoration for a 2×2 matrix. Each quadrant (i.e., scalar or zero matrix) is passed to `dec_2x2` which determines what the decoration would be like if that scalar were the pivot for the 2×2 matrix.

28b ⟨*Two by two matrix decoration functions* 28a⟩+≡

```
dec_2x2 :: DMatrx A -> Pos -> Det -> Dec
dec_2x2 ZeroDM       p det_ln  =  Deco ((p, 1), WorstDet, det_ln)
dec_2x2 (ScalarDM s) p det_ln  =  Deco ((p, 1), Detm (det_lg s), det_ln)
```

Defines:
`dec_2x2`, used in chunk 27b.
Uses `DMatrx` 24b, `Dec` 23b, `Deco` 23b, `Det` 23a, `Detm` 23a, `Pos` 22b, `ScalarDM` 24b, `ZeroDM` 24b, and `det_lg` 29c.

The determinant of a general $n \times n$ block is a bit more tricky than the 2×2 case. The function `det_nxn` looks for a nown-singular block (see [4, Section 3.3.1]). If the quadtree is not nown-singular, its determinant is `WorstDet`. Otherwise, the non-zero diagonal of the matrix is used to compute the known-determinant of the matrix by taking the geometric mean of the known-determinants of the non-zero diagonal.

28c ⟨*General matrix decoration functions* 28c⟩≡

```
det_nxn :: QuadDM A -> Det
det_nxn q@(nw, ne, sw, se)
  | (is_zeroDM nw || is_zeroDM se)
    && not (is_zeroDM ne)
    && not (is_zeroDM sw)
    && not (has_bad_det ne || has_bad_det sw)  =  geometric_meanDM ne sw
  | (is_zeroDM ne || is_zeroDM sw)
    && not (is_zeroDM nw)
    && not (is_zeroDM se)
    && not (has_bad_det nw || has_bad_det se)  =  geometric_meanDM nw se
  | otherwise                                  =  WorstDet
  where ⟨Geometric mean of two matrices 29a⟩
```

Defines:
`det_nxn`, used in chunk 27b.
Uses `Det` 23a, `QuadDM` 24b, `geometric_meanDM` 29a, `has_bad_det` 31, and `is_zeroDM` 31.
This definition is continued in chunk 29b.
This code is used in chunk 27b.

To calculate the known-determinant of an $n \times n$ matrix, the function `geometric_meanDM` is used on the known-determinants on the non-zero diagonal. Since the geometric mean of a and b is $\sqrt{ab}$ and since $\lg a$ and $\lg b$ are given, the computation for the rounded logarithm of the geometric mean is $(\lg a + \lg b + 1)/2$.

29a ⟨*Geometric mean of two matrices* 29a⟩≡

```
geometric_meanDM :: DMatrx A -> DMatrx A -> Det
geometric_meanDM m1 m2  =
  let  Detm det1  =  ln_known_det m1
       Detm det2  =  ln_known_det m2  in
    Detm ((det1 + det2 + 1) / 2)
```

Defines:
  `geometric_meanDM`, used in chunk 28c.
Uses `DMatrx` 24b, `Det` 23a, `Detm` 23a, and `ln_known_det` 24a.
This code is used in chunk 28c.

Analogous to `dec_2x2`, the function `dec_nxn` examines each quadrant of a decorated quadtree matrix to see what decoration would result from picking that quadrant as a pivot block. Zero matrices receive the worst decoration. For non-zero matrices, if the determinant of a quadrant is `WorstDet`, it is should not be a pivot block. Instead the pivot block inside the quadrant is offered up as a possible pivot block. On the other hand, if the known-determinant of the quadrant is acceptable, it is compared as a pivot block with the pivot block inside.

29b ⟨*General matrix decoration functions* 28c⟩+≡

```
dec_nxn :: DMatrx A -> Pos -> Det -> Dec
dec_nxn ZeroDM _ _   =  WorstDec
dec_nxn (DMtx (Deco ((_, depth), piv_ln_det, WorstDet)) _ )   p ln_det  =
  best [WorstDec, Deco ((p, 1 + depth), piv_ln_det, ln_det)]
dec_nxn (DMtx (Deco ((_, depth), piv_ln_det, cur_ln_det)) _ ) p ln_det  =
  best [Deco ((p, 1),         cur_ln_det, ln_det),
        Deco ((p, 1 + depth), piv_ln_det, ln_det)]
```

Defines:
  `dec_nxn`, used in chunk 27b.
Uses `DMatrx` 24b, `DMtx` 24b, `Dec` 23b, `Deco` 23b, `Det` 23a, `Pos` 22b, `WorstDec` 23a 23b, `ZeroDM` 24b, and `best` 29d.

The function `det_lg` forces us to use the `Float` data type. In Gofer, `log` works only only `Float`. There are other ways this logarithm could be computed — it is only supposed to be approximate.

Operationally, the definition is not surprising.

29c ⟨*Decoration helpers* 29c⟩≡

```
det_lg :: A -> Int
det_lg a  =  truncate ((log (abs a) / log 2.0) + 0.0)
```

Defines:
  `det_lg`, used in chunk 28.
This definition is continued in chunk 31.
This code is used in chunk 27b.

The function `best` picks out the best decoration, given a list of candidates. The actual picking is done by `better`.

29d ⟨*Picking the best decoration* 29d⟩≡

```
best :: [Dec] -> Dec
best [x]      =  x
best (x:xs)   =  better xs x
                   where ⟨Best-decoration helpers 30⟩
```

Defines:
  `best`, used in chunks 27b and 29b.
Uses `Dec` 23b and `better` 30.
This code is used in chunk 27b.

The function `better` does the dirty work of `best`. It takes the current best decoration, and compares it with the first decoration in the candidate list. The better of the two is compared with the remainder of the list. The function makes its decision, accumulator style, based on the following factors (in this order):

1. When the list of candidates is exhausted, the current best is returned.
2. If either the current best or the candidate is `WorstDec`, the other one is selected for further comparisons. (If they are both `WorstDec`, it does not matter which one is picked.)
3. The determinants of the pivot blocks from the current best and the candidate are checked. If either one is the worst possible (i.e., `WorstDet`), the other one is selected for further comparisons.
4. The decoration with the smallest depth (i.e., largest pivot block) is selected.
5. If the depths are equal, the decoration with the larger determinant is selected for numeric stability.
6. Otherwise, the candidate is selected for further comparisons.

30 ⟨*Best-decoration helpers* 30⟩≡
```
better :: [Dec] -> Dec -> Dec
better []            curr                     =  curr
better (x:xs)        WorstDec                 =  better xs x
better (WorstDec:xs) curr                     =  better xs curr
better (x:xs)        (Deco (_, WorstDet, _))  =  better xs x
better ((Deco (_, WorstDet, _)):xs) curr      =  better xs curr
better (x:xs) curr
   | xDepthPivot < cbDepthPivot  =  better xs x
   | xDepthPivot > cbDepthPivot  =  better xs curr
   | xLnDet < cbLnDet            =  better xs curr
   | xLnDet > cbLnDet            =  better xs x
   | otherwise                   =  better xs x
     where  Deco (( _, xDepthPivot),  (Detm xLnDet),  _)  =  x
            Deco (( _, cbDepthPivot), (Detm cbLnDet), _)  =  curr
```
Defines:
  `better`, used in chunk 29d.
Uses `Dec` 23b, `Deco` 23b, `Detm` 23a, and `WorstDec` 23a 23b.
This code is used in chunk 29d.

5.5 Special Decoration Recognizers

Certain decorated matrix types need to be easily recognized, like scalars and zero matrices. The functions `is_scalarDM` and `is_zeroDM` serve these functions. Note that `ZeroDM` is considered to be a scalar matrix by `is_scalarDM`.

The last function, `has_bad_det`, picks out the logarithm of the known-determinant from the decoration of a decorated matrix and checks to see if it is the worst determinant possible, signified by `WorstDet`.

31 ⟨*Decoration helpers* 29c⟩+≡

```
is_scalarDM :: DMatrx a -> Bool
is_scalarDM ZeroDM         =  True
is_scalarDM (ScalarDM _)   =  True
is_scalarDM _              =  False

is_zeroDM :: DMatrx a -> Bool
is_zeroDM ZeroDM   =  True
is_zeroDM _        =  False

has_bad_det :: DMatrx a -> Bool
has_bad_det m  =  ln_known_det m == WorstDet
```

Defines:
  `has_bad_det`, used in chunk 28c.
  `is_scalarDM`, used in chunk 27b.
  `is_zeroDM`, used in chunk 28c.
Uses `DMatrx` 24b, `ScalarDM` 24b, `ZeroDM` 24b, and `ln_known_det` 24a.

6 Quadtree Matrix Arithmetic

The `Num` class works only for homogeneous data types. That is, multiplication represented by the symbol `*` can only be defined from `Matrx` and `Matrx` to `Matrx`, not `DMatrx` and `Matrx` to `DMatrx`. In order to multiply a decorated quadtree matrix by a regular quadtree matrix, a new operator must be defined.

This section defines the extra arithmetic operators we need for quadtree matrices based on the needs of the matrix inversion algorithm. The functions defined here are by no means exhaustive.

6.1 File Layout

32a ⟨`math-qmtx.gs` 32a⟩≡
  ⟨*Infix specifications* 32b⟩
  ⟨*Decorated and undecorated arithmetic* 33a⟩
  ⟨*Permutation and undecorated arithmetic* 34⟩

Root chunk (not used in this document).

6.2 Infix Specifications

In order to use these arithmetic operators as infix operators, the Haskell `infix` command specifies their precedence and associativity.

32b ⟨*Infix specifications* 32b⟩≡
```
infixl 7 %*#, @*#, #*@, #*%%
infixl 6 #+%
```
Defines:
- `#*%%`, used in chunk 40.
- `#*@`, used in chunks 36 and 44a.
- `#+%`, used in chunk 41.
- `%*#`, used in chunk 42.
- `@*#`, used in chunks 36 and 44a.

This code is used in chunk 32a.

The symbols used in the operators require some explanation. The arithmetic operators `*`, `/` and `+` represent multiplication, division and addition, respectively, as one would expect. The other symbols represent the different data types:

Symbol	Meaning
`#`	regular quadtree matrix, `Matrx` (see Section 3)
`%`	decorated quadtree matrix, `DMatrx` (see Section 5)
`@`	permutation quadtree matrix, `PMatrx` (see Section 4)

These symbols indicate what type each operand is supposed to be. For example, `@*#` represents multiplication of a permutation matrix by a regular matrix. By default, each operator returns a regular quadtree matrix. Exceptions are noted by adding the symbol of the returned type to the operator. Thus, `.*%%` is the product of a scalar and a decorated matrix, returned as a decorated scalar matrix.

6.3 Arithmetic for Decorated and Undecorated Matrices

Addition and multiplication of decorated and regular quadtree matrices are defined first. The definitions are the same as multiplication and addition for any homogeneous definition (see Section 3.4), only with a different operator.

33a ⟨*Decorated and undecorated arithmetic* 33a⟩≡
```
(#+%) :: (Num a) => Matrx a -> DMatrx a -> Matrx a
ZeroM      #+% m1           =  undecorateDM m1
m0         #+% ZeroDM       =  m0
ScalarM x #+% ScalarDM y   =  normalizeSM (x + y)
Mtx q0     #+% DMtx _ q1    =  normalizeQM (qmap2 (#+%) q0 q1)

(%*#) :: (Num a, Num (DMatrx a), Num (Matrx a)) =>
  DMatrx a -> Matrx a -> Matrx a
ZeroDM      %*# _           =  ZeroM
_           %*# ZeroM       =  ZeroM
ScalarDM x %*# ScalarM y   =  normalizeSM (x * y)
DMtx _ q0   %*# Mtx q1      =
  Mtx (qmap2 (+) (qmap2 (%*#) (col_exchangeQ q0) (off_squashQ q1))
                 (qmap2 (%*#) q0                (prim_squashQ q1)))
```
Defines:
`#+%`, used in chunk 41.
`%*#`, used in chunk 42.
Uses `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `Mtx` 13a, `ScalarDM` 24b, `ScalarM` 13a, `ZeroDM` 24b, `ZeroM` 13a, `col_exchangeQ` 6, `normalizeQM` 13b, `normalizeSM` 13b, `off_squashQ` 6, `prim_squashQ` 6, `qmap2` 5a, and `undecorateDM` 26c.
This definition is continued in chunk 33b.
This code is used in chunk 32a.

This undecorated/decorated matrix product returns a decorated result.

33b ⟨*Decorated and undecorated arithmetic* 33a⟩+≡
```
(#*%%) :: (Num A, Ord A, Num (DMatrx A)) => Matrx A -> DMatrx A -> DMatrx A
ZeroM      #*%% _           =  ZeroDM
_          #*%% ZeroDM      =  ZeroDM
ScalarM x #*%% ScalarDM y  =  normalizeSDM (x * y)
Mtx q1     #*%% DMtx _ q2   =
  normalizeQDM (qmap2 (+) (qmap2 (#*%%) (col_exchangeQ q1) (off_squashQ q2))
                          (qmap2 (#*%%) q1                (prim_squashQ q2)))
```
Defines:
`#*%%`, used in chunk 40.
Uses `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `Mtx` 13a, `ScalarDM` 24b, `ScalarM` 13a, `ZeroDM` 24b, `ZeroM` 13a, `col_exchangeQ` 6, `normalizeQDM` 26a, `normalizeSDM` 26a, `off_squashQ` 6, `prim_squashQ` 6, and `qmap2` 5a.

6.4 Permutation Matrix and Undecorated Matrix Arithmetic

Multiplication of a permutation quadtree matrix and a regular quadtree matrix is straightforward. Note that scalar matrices are never referred to here since it is the identity and zero matrices which govern the behavior of the multiplication.

34 ⟨*Permutation and undecorated arithmetic* 34⟩≡

```
(@*#) :: Num (Matrx a) => PMatrx -> Matrx a -> Matrx a
ZeroPM  @*# _       =  ZeroM
_       @*# ZeroM   =  ZeroM
IdentPM @*# m       =  m
pm      @*# IdentM  =  pm @*# Mtx (unnormalizeM IdentM)
PMtx pq @*# Mtx mq  =
  Mtx (qmap2 (@*#) (col_exchangeQ pq) (off_squashQ mq))
  +
  Mtx (qmap2 (@*#) pq                (prim_squashQ mq))

(#*@) :: Num (Matrx a) => Matrx a -> PMatrx -> Matrx a
_      #*@ ZeroPM   =  ZeroM
ZeroM  #*@ _        =  ZeroM
m      #*@ IdentPM  =  m
IdentM #*@ pm       =  Mtx (unnormalizeM IdentM) #*@ pm
Mtx mq #*@ PMtx pq  =
  Mtx (qmap2 (#*@) (col_exchangeQ mq) (off_squashQ pq))
  +
  Mtx (qmap2 (#*@) mq                (prim_squashQ pq))
```

Defines:
  `#*@`, used in chunks 36 and 44a.
  `@*#`, used in chunks 36 and 44a.
Uses `IdentM` 13a, `IdentPM` 20b, `Matrx` 13a, `Mtx` 13a, `PMatrx` 20b, `PMtx` 20b, `ZeroM` 13a, `ZeroPM` 20b, `col_exchangeQ` 6, `off_squashQ` 6, `prim_squashQ` 6, `qmap2` 5a, and `unnormalizeM` 13c.
This code is used in chunk 32a.

7 $L + U'$ Decomposition

The remaining two sections implement the matrix decomposition and backsubstitution to do the actual inversion of a matrix as described in [4, Section 2]. The code in this section implements the decomposition algorithm described in [4, Sections 2.3, 3.1].

Decomposition is carried out through an undulant-pivot block scheme which means that pivots from A come from anywhere in the matrix and are of any size which is a power of two. (The selection of pivots is done by the decorated matrix, covered in Section 5.) However, conventional Gaussian elimination uses pivots along the main diagonal. Moving the decomposed pivot blocks in S to the main diagonal is done through permutation matrices P and Q for row and column exchanges, respectively. The permutation matrices P and Q are built from series encodings of reverse-Ahnentafel indices Π and Ψ which are assembled during decomposition. (See [4, Section 2.2] and Appendix C.) The triangulation of A is therefore $L + U' = PSQ$.

Since the pivot blocks are not necessarily scalars, decomposition also returns a list of pivot orders Ω. This allows us to differentiate between L and U' in the triangulation $L + U'$. This is most important for the three functions defined in Section 8 which take the triangulation and complete the inversion.

7.1 File Layout

35 ⟨`lu.gs` 35⟩≡
  ⟨*Decomposition* 36⟩
  ⟨*Pivoting* 38b⟩
  ⟨*Inverting and determinant* 43a⟩
Root chunk (not used in this document).

7.2 Decomposition of a Matrix

The function `decompose` implements Algorithm 2 of [4]. It takes the matrix A and its order, yielding the determinant of PAQ, the triangulation $L+U'$, permutation matrices P and Q, the list of pivot block orders Ω, the amount of padding added by the permutations, and a positive/negative sign.

The determinant returned by `pivots` is the determinant of PAQ, which is the determinant of A multiplied by the parities of P and Q. To find the determinant of A, the determinant of PAQ is multiplied by the parities of P^T and Q^T. Since the parities of P and P^T are the same,[2] we compute the parities of P and Q, instead of P^T and Q^T, with the function `parityAL`. Based on these parities, if the sign of the determinant must be changed, the sign is set to be negative.

$L+U'$ is the triangulation of A, as explained at the beginning of this section. P and Q are the permutation matrices also described at the beginning of this section. Ω is a list of the pivot block orders.

The amount of added padding is an integer denoting the amount of padding that was added to the matrix in order to get the pivot blocks onto the diagonal. (See Appendix D and [4, Section 3.2].)

36 ⟨*Decomposition* 36⟩≡
```
decompose :: (Num A, Num (Matrx A), Num (DMatrx A)) =>
   Int -> DMatrx A -> (A, Matrx A, PMatrx, PMatrx, Vectr Int, Int, Bool)
decompose ord (ScalarDM n)  =
  (n, ScalarM (fromInteger 1), IdentPM, IdentPM, ScalarV ord, 0, True)
decompose ord a  =
  let  (det, s, ahnenordL)  =  pivots (ceil_lg ord) a
       (_, _, piv_ordL)     =  unzip3 ahnenordL
       elim_ord             =  sum piv_ordL
       (new_ord, (pi, psi, omega))  =  insert_padding ord ahnenordL
       piV      =  list_to_tree new_ord omega pi
       psiV     =  list_to_tree new_ord omega psi
       (p, q)   =  convALtoPM piV psiV
       det' | ord == elim_ord   =  det
            | otherwise         =  fromInteger 0
       sign     =  (parityAL pi ord) * (parityAL psi ord)
       omegaV   =  list_to_tree new_ord omega omega
       pad_ord  =  new_ord - ord
       lu'      =  p @*# (pad (prefix_ident s ord) ord new_ord) #*@ q  in
    (det', lu', p, q, omegaV, pad_ord, sign > 0)
      where ⟨Decomposition support functions 37a⟩
```
Defines:
  `decompose`, used in chunk 44a.
Uses `#*@` 32b 34, `@*#` 32b 34, `DMatrx` 24b, `IdentPM` 20b, `Matrx` 13a, `PMatrx` 20b, `ScalarDM` 24b, `ScalarM` 13a, `ScalarV` 8a, `Vectr` 8a, `ceil_lg` 48b, `convALtoPM` 54a, `insert_padding` 57a, `list_to_tree` 37a, `pad` 37c, `parityAL` 63c, `pivots` 38b, `prefix_ident` 38a, and `unzip3` 50.
This code is used in chunk 35.

[2]If P is a permutation, then $I = PP^T$. So $\det I = \det(PP^T) = \det P \det P^T$. Since $\det I = +1$, the conclusion is that $\det P = \det P^T = \pm 1$.

The function `list_to_tree` takes the order of the matrix, the list of pivot block orders Ω, and a list. It returns a binary vector version of the list. The binary vector is the same one defined in Section 2 but used in a different way. In the standard binary vector, the split in a binary vector is based on the length of the vector. Alternatively, `list_to_tree` splits the vector based on the pivot sizes in Ω. The split is made so that the *sum* of the pivot sizes in each half are equal to the same power of 2, creating what is termed a *balanced* vector.

For example, suppose that $\Omega = [2, 1, 1]$. Let $[10, 20, 30]$ be the list being turned into a tree. Based on Ω, the list would be converted into $(10, (20, 30))$.

This function is used to balance Ω itself as well as Π and Ψ. Using this variant on the binary vector allows for quick recursing down a quadtree matrix.

37a ⟨*Decomposition support functions* 37a⟩≡

```
list_to_tree :: Int -> [Int] -> [a] -> Vectr a
list_to_tree ord [o] [x]
  | ord == o   =  ScalarV x
  | otherwise  =  error "Unexpected orders in list_to_tree."
list_to_tree ord omega xs  =
    let  new_ord  =  halve ord
         (omega_n, lis_n, omega_s, lis_s)
                  =  partition new_ord 0 omega xs
         n        =  list_to_tree new_ord omega_n lis_n
         s        =  list_to_tree new_ord omega_s lis_s  in
      Vec (n, s)
        where ⟨List to tree support functions 37b⟩
```

Defines:
`list_to_tree`, used in chunk 36.
Uses `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `halve` 49a, and `partition` 37b.
This definition is continued in chunks 37c and 38a.
This code is used in chunk 36.

The function `partition` divides the incoming lists into disjoint partitions. Specifically, `partition` does the dirty work for `list_to_tree`: it partitions the orders in Ω into two halves so that that orders sum up to a power of two. The corresponding elements in the second list are split based on the split of Ω.

37b ⟨*List to tree support functions* 37b⟩≡

```
partition :: Int -> Int -> [Int] -> [a] -> ([Int], [a], [Int], [a])
partition ord i omega@(o:omega') xL@(x:xL')
  | ord == i   =  ([], [], omega, xL)
  | otherwise  =  let  (omega_n, xL_n, omega_s, xL_s)
                             =  partition ord (o + i) omega' xL'  in
                       (o : omega_n, x : xL_n, omega_s, xL_s)
```

Defines:
`partition`, used in chunk 37a.
This code is used in chunk 37a.

Occasionally, when converting from the decomposition S to the triangulation $L + U'$, padding must be added to S. (See [4, Section 3.2] and Appendix D.) The function `pad` adds the extra padding to S if it needs any.

37c ⟨*Decomposition support functions* 37a⟩+≡

```
pad :: Matrx a -> Int -> Int -> Matrx a
pad s ord new_ord
  | (2 * ord) > new_ord  =  s
  | otherwise            =  Mtx (pad s (twice ord) new_ord, ZeroM,
                                 ZeroM,                        IdentM)
```

Defines:
`pad`, used in chunk 36.
Uses `IdentM` 13a, `Matrx` 13a, `Mtx` 13a, `ZeroM` 13a, and `twice` 49a.

The function `prefix_ident` is used to change the padding in the decomposition. Normally, matrices are padded with zeros. However, the matrix inversion algorithm here requires that identity matrices be placed along the main diagonal in the pre-existing padding. Note that `pad` (defined above) adds *extra* padding while `prefix_ident` changes existing zero padding on the main diagonal to identity padding.

38a ⟨*Decomposition support functions* 37a⟩+≡
```
prefix_ident :: Matrx a -> Int -> Matrx a
prefix_ident s ord  =  helper s (power2 (ceil_lg ord)) ord
  where helper s full_ord ord
          | full_ord == ord  =  s
          | ord  == 0        =  IdentM
          | otherwise        =
              let  (nw, ne, sw, se)  =  unnormalizeM s
                   full_ord'         =  halve full_ord
                   nw'  =  helper nw full_ord' (min full_ord' ord)
                   se'  =  helper se full_ord' (max (ord - full_ord') 0)  in
                normalizeQM (nw', ne, sw, se')
```
Defines:
`prefix_ident`, used in chunk 36.
Uses `IdentM` 13a, `Matrx` 13a, `ceil_lg` 48b, `halve` 49a, `normalizeQM` 13b, `power2` 48b, and `unnormalizeM` 13c.

7.3 Pivoting a Matrix

The function `pivots` starts the actual decomposition as described in Algorithm 1 of [4, Section 2.3]. It takes the height of A and A itself. It yields the determinant of PAQ, the decomposition S, the list of pivot orders Ω, and series encodings Π and Ψ. (The `AhnenOrdList` is a list of tuples which effectively contain Ω, Π and Ψ. See Appendix C.)

38b ⟨*Pivoting* 38b⟩≡
```
pivots :: (Num A, Num (Matrx A), Num (DMatrx A)) =>
  Int -> DMatrx A -> (A, Matrx A, AhnenOrdList)
pivots h a  =  pvots h a (fromInteger 1) ZeroM []
```
⟨*Pivoting support functions* 38c⟩

Defines:
`pivots`, used in chunks 36 and 43a.
Uses `AhnenOrdList` 56b, `DMatrx` 24b, `Matrx` 13a, `ZeroM` 13a, and `pvots` 38c.
This code is used in chunk 35.

The function `pvots` repeatedly calls the function `piv` to carry out the iteration of Algorithm 1 of [4] until A is fully eliminated and the decomposition is in S.

38c ⟨*Pivoting support functions* 38c⟩≡
```
pvots :: (Num (Matrx A), Num (DMatrx A)) =>
  Int -> DMatrx A -> A -> Matrx A -> AhnenOrdList
    -> (A, Matrx A, AhnenOrdList)
pvots _ ZeroDM det s ahnen  =  (det, s, ahnen)
pvots h a det s ahnen       =
  let  (_, dep)                    =  signpost a
       pivSize                     =  power2 (h - dep)
       (a', _, s', _, i, j, det')  =  piv dep pivSize a s  in
    pvots h a' (det * det') s' ((i, j, pivSize) : ahnen)
```
Defines:
`pvots`, used in chunk 38b.
Uses `AhnenOrdList` 56b, `DMatrx` 24b, `Matrx` 13a, `ZeroDM` 24b, `piv` 39a, `power2` 48b, and `signpost` 24a.
This definition is continued in chunks 39–42.
This code is used in chunk 38b.

The function `piv` (see [4, Section 3.1.3]) works on the quadrant which actually contains the pivot block. On iteration l, `piv` takes the depth and size of the pivot block, A_l and S_l. In the base case, A_l is *the* pivot block. Processing is passed on to `invertDM` to do the inverting of the pivot block. Otherwise, the quadrants of A_l are rearranged so that the pivot block is in the northwest quadrant, the row quadrant is in the northeast, the column quadrant is in the southwest, and the off quadrant is in the southeast. The appropriate functions are called on these quadrants.

The function yields A_{l+1}, the inverse V of the pivot block, S_{l+1}, the pivot column from A_l, indices i_{l+1} and j_{l+1} to identify the location of the pivot block, and d_{l+1}, the determinant of the pivot block.

39a ⟨*Pivoting support functions* 38c⟩+≡
```
piv :: (Num A, Num (DMatrx A), Num (Matrx A)) =>
  Int -> Int -> DMatrx A -> Matrx A
    -> (DMatrx A, Matrx A, Matrx A, Vectr (Matrx A), Int, Int, A)
piv 0 pivSize a ZeroM  =
  let  (v, det)  =  invertDM pivSize a  in
    (ZeroDM, v, v, ZeroV, 1, 1, det)
piv (dep+1) pivSize a@(DMtx _ aQ) s  =
  let  (dir, _)  =  signpost a
       (vecperm, rowindex, quadperm, colindex)  =
         case dir of
           NW -> (identP,    leftson,  identQ,          leftson)
           NE -> (identP,    leftson,  col_exchangeQ,   rightson)
           SW -> (exchangeP, rightson, row_exchangeQ,   leftson)
           SE -> (exchangeP, rightson, diag_exchangeQ, rightson)
       (a_nw, a_ne, a_sw, a_se)  =  quadperm aQ
       sQ                        =  unnormalizeM s
       (s_nw, s_ne, s_sw, s_se)  =  quadperm sQ
       (a'_nw, v, s'_nw, gauss_n, i, j, det)  =  piv dep pivSize a_nw s_nw
       (a'_ne, s'_ne)  =  row a_ne s_ne gauss_n i
       (a'_sw, s'_sw, gauss_s)  =  col a_sw s_sw a_nw i j v
       a'_se  =  off a_se gauss_s a_ne i  in
    (decorateDQ (quadperm (a'_nw, a'_ne, a'_sw, a'_se)),
     v,
     normalizeQM (quadperm (s'_nw, s'_ne, s'_sw, s_se)),
     normalizePV (vecperm (gauss_n, gauss_s)),
     rowindex i, colindex j, det)
      where  ⟨Pivot permutation functions 39b⟩
```
Defines:
  `piv`, used in chunk 38c.
Uses `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `NE` 22b, `NW` 22b, `SE` 22b, `SW` 22b, `Vectr` 8a, `ZeroDM` 24b, `ZeroM` 13a, `ZeroV` 8a, `col` 42, `col_exchangeQ` 6, `decorateDQ` 27b, `diag_exchangeQ` 6, `exchangeP` 4b, `identP` 4b, `identQ` 6, `invertDM` 44a, `leftson` 39b, `normalizePV` 8b, `normalizeQM` 13b, `off` 40a, `rightson` 39b, `row` 41a, `row_exchangeQ` 6, `signpost` 24a, and `unnormalizeM` 13c.

The functions here are for computing new reverse-Ahnentafel indices.

39b ⟨*Pivot permutation functions* 39b⟩≡
```
leftson  i  =  2*i
rightson i  =  2*i+1
```
Defines:
  `leftson`, used in chunk 39a.
  `rightson`, used in chunk 39a.
This code is used in chunk 39a.

On iteration l, the function `off` (see [4, Section 3.1.1]) takes A_l, the Gaussian pivot column, the pivot row from A_l, and an index i, indexing the pivot row from the given matrix. It yields A_{l+1}.

This function handles the quadrants of the matrix which lie outside the row and column of the pivot block. It is multiplied by the pivot column and the pivot row. No actual elimination is done here, just an updating of A_l.

40a ⟨*Pivoting support functions* 38c⟩+≡
```
off :: (Num (DMatrx A)) =>
  DMatrx A -> Vectr (Matrx A) -> DMatrx A -> Int -> DMatrx A
off a _      ZeroDM _  =  a
off a ZeroV  _      _  =  a
off ZeroDM gauss piv_r i  =  neg_outer_product i gauss piv_r
                               where ⟨Negative outer product 40b⟩
off a (ScalarV gauss) pivr 1  =  a - (gauss #*%% pivr)
off (DMtx _ aQ) (Vec (gauss_n, gauss_s)) (DMtx _ (r_nw, r_ne, r_sw, r_se)) i  =
  let  ires                  =  halve i
       iresT                 =  tuple4 ires
       pcol                  =  (gauss_n, gauss_n, gauss_s, gauss_s)
       prow | even i         =  (r_nw, r_ne, r_nw, r_ne)
            | otherwise      =  (r_sw, r_se, r_sw, r_se)  in
    decorateDQ (qmap4 off aQ pcol prow iresT)
```
Defines:
`off`, used in chunks 39a, 41a, and 42.
Uses `#*%%` 32b 33b, `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroDM` 24b, `ZeroV` 8a, `decorateDQ` 27b, `halve` 49a, `neg_outer_product` 40b, `qmap4` 5a, and `tuple4` 50.

The negative outer product is needed to multiply the pivot column by the pivot row in `off`. The outer product is defined to be $\vec{x}^T\vec{y}$ which results in a matrix. The function `off` requires the negation of the outer product, so we compute the outer product and the negation simultaneously.

40b ⟨*Negative outer product* 40b⟩≡
```
neg_outer_product :: (Num (DMatrx A)) =>
  Int -> Vectr (Matrx A) -> DMatrx A -> DMatrx A
neg_outer_product _ ZeroV  _         =  ZeroDM
neg_outer_product _ _      ZeroDM    =  ZeroDM
neg_outer_product 1 (ScalarV ca') b  =  negate (ca' #*%% b)
neg_outer_product i (Vec (gauss_n, gauss_s)) (DMtx _ (nw, ne, sw, se))  =
  let  ires              =  halve i
       iresT             =  tuple4 ires
       pcol              =  (gauss_n, gauss_n, gauss_s, gauss_s)
       prow | even i     =  (nw, ne, nw, ne)
            | otherwise  =  (sw, se, sw, se)  in
    decorateDQ (qmap3 neg_outer_product iresT pcol prow)
```
Defines:
`neg_outer_product`, used in chunk 40a.
Uses `#*%%` 32b 33b, `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroDM` 24b, `ZeroV` 8a, `decorateDQ` 27b, `halve` 49a, `qmap3` 5a, and `tuple4` 50.
This code is used in chunk 40a.

The function `row` (see [4, Section 3.1.2]) takes A_l, S_l, a pivot column, and an index i. The pivot column is needed for the off quadrants. The value i is used to identify the pivot row in A_l.

The returned result is A_{l+1} and S_{l+1}.

41a ⟨*Pivoting support functions* 38c⟩+≡

```
row :: (Num A, Ord A, Num (DMatrx A)) =>
  DMatrx A -> Matrx A -> Vectr (Matrx A) -> Int -> (DMatrx A, Matrx A)
row ZeroDM s _ _  =  (ZeroDM, s)
row a      s _ 1  =  (ZeroDM, s #+% a)
row a s ZeroV i   =  row_extract a s i
                       where ⟨Row extraction 41b⟩
row (DMtx _ (a_nw, a_ne, a_sw, a_se)) s (Vec (gauss_n, gauss_s)) i  =
  let  (s_nw, s_ne, s_sw, s_se)  =  unnormalizeM s
       ires                      =  halve i  in
    if even i
    then let  a'_sw  =  off a_sw gauss_s a_nw ires
              a'_se  =  off a_se gauss_s a_ne ires
              (a'_nw, s'_nw)  =  row a_nw s_nw gauss_n ires
              (a'_ne, s'_ne)  =  row a_ne s_ne gauss_n ires  in
           (decorateDQ (a'_nw, a'_ne, a'_sw, a'_se),
            normalizeQM (s'_nw, s'_ne, s_sw, s_se))
    else let  a'_nw  =  off a_nw gauss_n a_sw ires
              a'_ne  =  off a_ne gauss_n a_se ires
              (a'_sw, s'_sw)  =  row a_sw s_sw gauss_s ires
              (a'_se, s'_se)  =  row a_se s_se gauss_s ires  in
           (decorateDQ (a'_nw, a'_ne, a'_sw, a'_se),
            normalizeQM (s_nw, s_ne, s'_sw, s'_se))
```

Defines:
`row`, used in chunk 39a.

Uses `#+%` 32b 33a, `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `Vec` 8a, `Vectr` 8a, `ZeroDM` 24b, `ZeroV` 8a, `decorateDQ` 27b, `halve` 49a, `normalizeQM` 13b, `off` 40a, `row_extract` 41b, and `unnormalizeM` 13c.

The function `row_extract` takes A_l, S_l and i. The value i is the reverse-Ahnentafel index for the row from A_l that contains the pivot block. The result is A_{l+1} and S_{l+1}.

41b ⟨*Row extraction* 41b⟩≡

```
row_extract :: (Num A, Ord A) =>
  DMatrx A -> Matrx A -> Int -> (DMatrx A, Matrx A)
row_extract ZeroDM s _  =  (ZeroDM, s)
row_extract a      s 1  =  (ZeroDM, s #+% a)
row_extract (DMtx _ (a_nw, a_ne, a_sw, a_se)) s i  =
  let  (s_nw, s_ne, s_sw, s_se)  =  unnormalizeM s
       ires                      =  halve i  in
    if even i
    then let  (a'_nw, s'_nw)  =  row_extract a_nw s_nw ires
              (a'_ne, s'_ne)  =  row_extract a_ne s_ne ires  in
           (decorateDQ (a'_nw, a'_ne, a_sw, a_se),
            normalizeQM (s'_nw, s'_ne, s_sw, s_se))
    else let  (a'_sw, s'_sw)  =  row_extract a_sw s_sw ires
              (a'_se, s'_se)  =  row_extract a_se s_se ires  in
           (decorateDQ (a_nw, a_ne, a'_sw, a'_se),
            normalizeQM (s_nw, s_ne, s'_sw, s'_se))
```

Defines:
`row_extract`, used in chunk 41a.

Uses `#+%` 32b 33a, `DMatrx` 24b, `DMtx` 24b, `Matrx` 13a, `ZeroDM` 24b, `decorateDQ` 27b, `halve` 49a, `normalizeQM` 13b, and `unnormalizeM` 13c.

This code is used in chunk 41a.

The function col (see [4, Section 3.1.2]) works on the quadrant which contains the column of the pivot block (but not the pivot block itself). It takes A_l, S_l, a pivot row, indices i and j, and V. V is the inverted pivot block. The indices i and j pinpoint the pivot block. The returned results are A_{l+1}, S_{l+1} and a pivot column for the off quadrants.

42 ⟨*Pivoting support functions* 38c⟩+≡
```
col :: (Num (Matrx A), Num (DMatrx A)) =>
   DMatrx A -> Matrx A -> DMatrx A -> Int -> Int -> Matrx A
      -> (DMatrx A, Matrx A, Vectr (Matrx A))
col ZeroDM s _ _ _ _  =  (ZeroDM, s, ZeroV)
col a      s r 1 1 v  =  let  ca'  =  a %*# v  in
                                  (ZeroDM, ca' + s, ScalarV ca')
col (DMtx _ (a_nw, a_ne, a_sw, a_se)) s
      (DMtx _ (r_nw, r_ne, r_sw, r_se)) i j v  =
   let  (row_w, row_e) | even i     =  (r_nw, r_ne)
                       | otherwise  =  (r_sw, r_se)
         (s_nw, s_ne, s_sw, s_se)  =  unnormalizeM s
         ires  =  halve i
         jres  =  halve j  in
      if even j
      then let  (a'_nw, s'_nw, gauss_n)  =  col a_nw s_nw row_w ires jres v
                (a'_sw, s'_sw, gauss_s)  =  col a_sw s_sw row_w ires jres v
                a'_ne  =  off a_ne gauss_n row_e ires
                a'_se  =  off a_se gauss_s row_e ires  in
           (decorateDQ (a'_nw, a'_ne, a'_sw, a'_se),
            normalizeQM (s'_nw, s_ne, s'_sw, s_se),
            normalizePV (gauss_n, gauss_s))
      else let  (a'_ne, s'_ne, gauss_n)  =  col a_ne s_ne row_e ires jres v
                (a'_se, s'_se, gauss_s)  =  col a_se s_se row_e ires jres v
                a'_nw  =  off a_nw gauss_n row_w ires
                a'_sw  =  off a_sw gauss_s row_w ires  in
           (decorateDQ (a'_nw, a'_ne, a'_sw, a'_se),
            normalizeQM (s_nw, s'_ne, s_sw, s'_se),
            normalizePV (gauss_n, gauss_s))
```
Defines:
col, used in chunk 39a.

Uses %*# 32b 33a, DMatrx 24b, DMtx 24b, Matrx 13a, ScalarV 8a, Vectr 8a, ZeroDM 24b, ZeroV 8a, decorateDQ 27b, halve 49a, normalizePV 8b, normalizeQM 13b, off 40a, and unnormalizeM 13c.

7.4 Calculating the Determinant and Inverse of a Matrix

In addition to the decomposed matrix, decomposition also returns the determinant of A as described in Algorithm 3 of [4]. The function `determinant` implements this algorithm, returning just the determinant of A.

Note the function `parity` which calculates the parity of a permutation matrix based on its series encoding. This function is explained in Appendix E, and the need for this function is described in the beginning of Section 7.2.

43a ⟨*Inverting and determinant* 43a⟩≡

```
determinant :: (Num A, Ord A, Num (Matrx A), Num (DMatrx A)) =>
  Matrix A -> A
determinant (OMtx _ ZeroM)        =  fromInteger 0
determinant (OMtx _ (ScalarM x))  =  x
determinant (OMtx ord a)          =
  let  (det, _, ahnenordL)  =  pivots (ceil_lg ord) (decorateM a)
       (pi, psi, ordL)      =  unzip3 ahnenordL  in
    if (ord > sum ordL)
    then fromInteger 0
    else (det * fromInteger (parityAL pi ord) * fromInteger (parityAL psi ord))
```

Defines:
`determinant`, never used.
Uses `DMatrx` 24b, `Matrix` 16a, `Matrx` 13a, `OMtx` 16a, `ScalarM` 13a, `ZeroM` 13a, `ceil_lg` 48b, `decorateM` 27a, `parityAL` 63c, `pivots` 38b, and `unzip3` 50.
This definition is continued in chunks 43 and 44.
This code is used in chunk 35.

The function `invert` is the top level inversion function. It takes a top level quadtree matrix, decorates the quadtree matrix inside and passes the processing on to `invertDM`.

43b ⟨*Inverting and determinant* 43a⟩+≡

```
invert :: (Num A, Ord A, Num (Matrx A), Num (DMatrx A), Num (Matrix A)) =>
  Matrix A -> (Matrix A, A)
invert (OMtx ord m)  =
  let  (a_inv, det)  =  invertDM ord (decorateM m)  in
    (OMtx ord a_inv, det)
```

Defines:
`invert`, never used.
Uses `DMatrx` 24b, `Matrix` 16a, `Matrx` 13a, `OMtx` 16a, `decorateM` 27a, and `invertDM` 44a.

The function `invertDM` takes an order and a decorated matrix, and returns the inverse and determinant of the matrix. This function is used by the top level inversion function to invert entire matrices as well as by `piv` to invert pivot blocks. It is based on Algorithm 6 in [4, Section 2.5].

The function `inv_f`, defined in Section 8, implements the function f described in [4, Section 2.5]. This function does the backsubstitution to compute A^{-1} from the triangulation.

44a ⟨*Inverting and determinant* 43a⟩+≡
```
invertDM :: (Num A, Num (Matrx A), Num (DMatrx A)) =>
  Int -> DMatrx A -> (Matrx A, A)
invertDM _ ZeroDM        =  error "Inverting a zero matrix."
invertDM _ (ScalarDM x)  =  (ScalarM (fromInteger 1 / x), x)
invertDM ord a  =
  let  (det, lu, p, q, omegaV, pad_ord, sign)  =  decompose ord a  in
    if (det == fromInteger 0)
    then error "Inverting singular matrix."
    else let  v        =  inv_f lu omegaV
              inverse  =  unpad (q @*# v #*@ p) ord pad_ord  in
           if sign
           then (inverse, det)
           else (inverse, negate det)
```
Defines:
`invertDM`, used in chunks 39a and 43b.
Uses `#*@` 32b 34, `@*#` 32b 34, `DMatrx` 24b, `Matrx` 13a, `ScalarDM` 24b, `ScalarM` 13a, `ZeroDM` 24b, `decompose` 36, `inv_f` 45b, and `unpad` 44b.

The function `unpad` removes the padding that was added by `pad`.

44b ⟨*Inverting and determinant* 43a⟩+≡
```
unpad :: Matrx a -> Int -> Int -> Matrx a
unpad ZeroM ord new_ord
  | ord' > new_ord  =  ZeroM
  | otherwise       =  error "Order does not match in unpad."
      where  ord'  =  twice ord
unpad v@(Mtx (_,_,_,se)) ord new_ord
  | ord' > new_ord  =  v
  | otherwise       =  unpad se ord' new_ord
     where  ord'  =  twice ord
```
Defines:
`unpad`, used in chunk 44a.
Uses `Matrx` 13a, `Mtx` 13a, `ZeroM` 13a, and `twice` 49a.

8 Inverting the Triangulation

This section represents the heart of matrix inversion, completing the inversion from a decomposition. The decomposition algorithms (especially the bookkeeping algorithms in the appendices) are long and complex; however, two-thirds of the execution for matrix inversion is spent executing the code in this section. These functions are very tight and highly parallel. (See [4, Section 2.5].)

Decomposition of A returns $L + U'$. The inversion of A is $[(I + L)U]^{-1} = U^{-1}(I + L)^{-1}$. Traditionally, U^{-1} and $(I + L)^{-1}$ are computed separately and then multiplied together for the final answer. However, the three functions in this section compute $[(I + L)U]^{-1}$ directly.

The functions are explained in detail in [4, Section 2.5], including correctness proofs.

8.1 File Layout

45a ⟨`circles.gs` 45a⟩≡
  ⟨*Function f definition* 45b⟩
  ⟨*Function g definition* 46a⟩
  ⟨*Function h definition* 46b⟩

Root chunk (not used in this document).

8.2 The Function f

The function f defined in Algorithm 6 of [4] is called `inv_f` here to prevent identifier confusion in `noweb`. The invariant of this function is $f(L + U', \Omega) = U^{-1}(I + L)^{-1}$ where $L + U'$ is the triangulation of A and Ω is the vector of pivot orders used in the decomposition. Thus, f is the function which is used to initiate the inversion of the triangulation. The functions g and h assist in the process.

Note that the vector used to hold Ω in all of these functions is the balanced binary vector created by `list_to_tree` defined in Section 7.2.

45b ⟨*Function f definition* 45b⟩≡

```
inv_f :: (Num (Matrx a)) => Matrx a -> Vectr Int -> Matrx a
inv_f lu (ScalarV _)  =  lu
inv_f lu (Vec (omega_n, omega_s))  =
  let  (lu_nw, ee, ww, lu_se)  =  unnormalizeM lu
       kk  =  inv_f lu_se omega_s
       bb  =  inv_f lu_nw omega_n
       jj  =  inv_h lu_nw (kk * ww) omega_n
       hh  =  inv_g lu_nw (ee * kk) omega_n
       cc  =  inv_g lu_nw (ee * jj) omega_n
       gg  =  bb + cc  in
     normalizeQM (gg, hh, jj, kk)
```

Defines:
  `inv_f`, used in chunk 44a.
Uses `Matrx` 13a, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `inv_g` 46a, `inv_h` 46b, `normalizeQM` 13b, and `unnormalizeM` 13c.
This code is used in chunk 45a.

8.3 The Function g

This is the function g defined in Algorithm 6 of [4], called `inv_g` here. The invariant is $g(L + U', X, \Omega) = -U^{-1}X$ where $L+U'$ is the triangulation of A, Ω is the balanced vector of pivot orders and X is any matrix.

46a ⟨*Function g definition* 46a⟩≡
```
inv_g :: (Num (Matrx a)) => Matrx a -> Matrx a -> Vectr Int -> Matrx a
inv_g _ ZeroM _           =  ZeroM
inv_g lu xx (ScalarV _)   =  lu * (negate xx)
inv_g lu xx (Vec (omega_n, omega_s))  =
  let  (lu_nw, ee, _, lu_se)          =  unnormalizeM lu
       (xx_nw, xx_ne, xx_sw, xx_se)   =  unnormalizeM xx
       jj  =  inv_g lu_se xx_sw omega_s
       kk  =  inv_g lu_se xx_se omega_s
       gg  =  inv_g lu_nw (xx_nw + ee * jj) omega_n
       hh  =  inv_g lu_nw (xx_ne + ee * kk) omega_n  in
    normalizeQM (gg, hh, jj, kk)
```
Defines:
  `inv_g`, used in chunk 45b.
Uses `Matrx` 13a, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroM` 13a, `normalizeQM` 13b, and `unnormalizeM` 13c.
This code is used in chunk 45a.

8.4 The Function h

This is the function h defined in Algorithm 6 of [4]. The mathematic definition of h is $h(L + U', X, \Omega) = -X(I + L)^{-1}$ where $L + U'$ is the triangulation of A, Ω is the balanced vector of pivot orders and X is any matrix.

46b ⟨*Function h definition* 46b⟩≡
```
inv_h :: (Num (Matrx a)) => Matrx a -> Matrx a -> Vectr Int -> Matrx a
inv_h _ ZeroM _           =  ZeroM
inv_h _  xx (ScalarV _)   =  negate xx
inv_h lu xx (Vec (omega_n, omega_s))  =
  let  (lu_nw, _, ww, lu_se)          =  unnormalizeM lu
       (xx_nw, xx_ne, xx_sw, xx_se)   =  unnormalizeM xx
       hh  =  inv_h lu_se xx_ne omega_s
       kk  =  inv_h lu_se xx_se omega_s
       gg  =  inv_h lu_nw (xx_nw + hh * ww) omega_n
       jj  =  inv_h lu_nw (xx_sw + kk * ww) omega_n  in
    normalizeQM (gg, hh, jj, kk)
```
Defines:
  `inv_h`, used in chunk 45b.
Uses `Matrx` 13a, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroM` 13a, `normalizeQM` 13b, and `unnormalizeM` 13c.
This code is used in chunk 45a.

A The Project File

Matrix inversion requires the following files which are defined in this report. The project file defined here can be used in Gofer to load the files in the proper order.

47 ⟨invert.gp 47⟩≡

```
math.gs
pandq.gs
bin-vector.gs
reg-qmtx.gs
dec-qmtx.gs
perm-qmtx.gs
math-qmtx.gs
circles.gs
ahnen.gs
padding.gs
parity.gs
lu.gs
```

Root chunk (not used in this document).

B Mathematical Utility Functions

Gofer does not define all of the mathematical functions needed for matrix inversion, so they are defined here.

B.1 File Layout

48a ⟨math.gs 48a⟩≡
  ⟨*Logarithmic and exponential functions* 48b⟩
  ⟨*Division functions* 49a⟩
  ⟨*Logic functions* 49b⟩
  ⟨*Miscellaneous math definitions* 50⟩

Root chunk (not used in this document).

B.2 Logarithmic and Exponential Functions

Gofer is missing logarithmic functions for integers. The definitions here are obvious and straightforward: `floor_lg` and `ceil_lg` are the floor and the ceiling of the logarithm base 2, respectively. The function `lg` is simply a renaming of `floor_lg`. To calculate powers of 2, we use `power2`. Finally, `floor_power2` returns the greatest power of 2 which is less than or equal to the input.

All of these functions should be bit operations on the integer which could be done in two or three processor cycles.

48b ⟨*Logarithmic and exponential functions* 48b⟩≡
```
floor_lg :: Int -> Int
floor_lg 1  =  0
floor_lg n  =  1 + floor_lg (halve n)

lg  =  floor_lg

ceil_lg :: Int -> Int
ceil_lg 1   =  0
ceil_lg n   =  ceil_lg (halve (n+1)) + 1

power2 :: Int -> Int
power2 0      =  1
power2 (n+1)  =  twice (power2 n)

floor_power2 :: Int -> Int
floor_power2 n | n <= 1     =  1
               | otherwise  =  twice (floor_power2 (halve n))
```
Defines:
  `ceil_lg`, used in chunks 18b, 36, 38a, 43a, 57a, and 63c.
  `floor_lg`, used in chunk 52c.
  `floor_power2`, never used.
  `lg`, used in chunk 53c.
  `power2`, used in chunks 18b, 38, 52c, 53c, 57a, and 63c.
Uses `halve` 49a and `twice` 49a.
This code is used in chunk 48a.

B.3 Division and Multiplication Functions

These functions are used to multiply and divide by 2, which should really be bit shifts of the integer.

49a ⟨*Division functions* 49a⟩≡
```
halve :: Int -> Int
halve n  =  div n 2

twice  =  (2 *)
```
Defines:
  `halve`, used in chunks 18c, 37a, 38a, 40–42, 48b, 49b, 53c, 55, and 58–60.
  `twice`, used in chunks 37c, 44b, 48b, 49b, and 57–59.
This code is used in chunk 48a.

B.4 Logic Functions

Some logical functions are missing in Gofer like `xor`. The function `fixnumlogand` computes "fixed number logical and".

49b ⟨*Logic functions* 49b⟩≡
```
xor :: Bool -> Bool -> Bool
xor True  q  =  not q
xor False q  =  q

fixnumlogand :: Int -> Int -> Int
fixnumlogand m 0 = 0
fixnumlogand m 1 | odd m      =  1
                 | otherwise  =  0
fixnumlogand m n  =
  let  bit | odd m && odd n  =  1
           | otherwise       =  0  in
    bit + twice (fixnumlogand (halve m) (halve n))
```
Defines:
  `fixnumlogand`, used in chunk 58b.
  `xor`, used in chunks 64 and 65.
Uses `halve` 49a and `twice` 49a.
This code is used in chunk 48a.

B.5 Miscellaneous Definitions

The function `tuple4` is useful for mapping. Sometimes one value is needed for all four branches of a quadtree mapping. This function takes the single value and returns it in all four slots of a quadtree.

When padding a matrix during inversion, lists of three-tuples and three-tuples of lists are passed back and forth. In order to use these data structures, the `Eq` class is asserted over three-tuples, and `unzip3` is defined to turn a list of three-tuples into a three-tuple of lists. The definition of `unzip3` comes from the standard Haskell prelude.

50 ⟨*Miscellaneous math definitions* 50⟩≡

```
tuple4 :: a -> (a, a, a, a)
tuple4 x  =  (x,x,x,x)

instance Eq a => Eq (a, a, a)  where
  (x0, x1, x2) == (y0, y1, y2)  =  (x0 == y0) && (x1 == y1) && (x2 == y2)

unzip3 :: [(a,b,c)] -> ([a],[b],[c])
unzip3  =  foldr (\(a, b, c) ~(as, bs, cs) -> (a:as,b:bs,c:cs)) ([],[],[])
```

Defines:
`tuple4`, used in chunk 40.
`unzip3`, used in chunks 36 and 43a.
This code is used in chunk 48a.

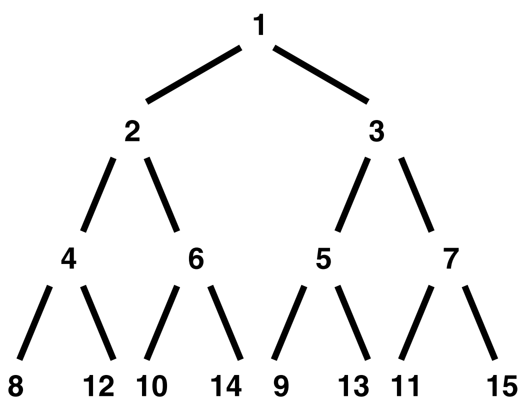

Figure 1: Reverse-Ahnentafel tree to Level 3

C Reverse-Ahnentafel Indexing and Permutation Matrices

Permutation matrices are used to move the pivot blocks onto the main diagonal. Decomposition does not build the permutation matrices directly, but uses a series encoding for the matrices consisting of reverse-Ahnentafel indices. (See Section 7.)

Standard Ahnentafel indexing consists of numbering the nodes of a binary tree sequentially across a level. The beginning of each level continues where the previous level left off.

For computational ease of moving from level to level, reverse-Ahnentafel indexing (see [4, Section 1.2]) is used for the inversion of quadtree matrices:

- The root (level 0) is numbered 1.
- The left child of a node i at level j is numbered by $i + 2^j$.
- The right child of a node i at level j is numbered by $i + 2^{j+1}$.

Note that $j = \lfloor \lg i \rfloor$ (see Theorem 1 of [4]).

See Figure 1 for a reverse-Ahnentafel binary tree out to the third level.

Reverse-Ahnentafel indexing allows indices to be built from the bottom up using addition and doubling. Descending the tree is also easy, only using quotient and remainder on 2 at each interior node. Computationally these are very quick when implemented using shifting. Moving across a level can also be done, although with more computational work, because the reverse-Ahnentafel index is equivalent to the Ahnentafel index for the same node through a bit reversal.

C.1 File Layout

51 ⟨`ahnen.gs` 51⟩≡
  ⟨*Ahnentafel indices* 52a⟩
  ⟨*Ahnentafel to permutation quadtree matrix* 54a⟩
Root chunk (not used in this document).

C.2 Reverse-Ahnentafel Indexing of Padding Blocks

The function `addresses_padding` computes reverse-Ahnentafel indices for the internal padding inserted into the triangulation. (See Appendix D.)

For example, let S be a 4×4 matrix which is padded with blocks of sizes 1, 2 and 1 (in that sequence) to S', the padded form of S of size 8. (See the example in Appendix D.) The padding count for S' would be $[0, 0, 1, 2]$ representing the number of padding blocks of size 8, 4, 2, and 1, respectively.

The function `addresses_padding` takes such a list of counts, and computes the reverse Ahnentafel indices for each block. Indices are pulled off from the right of the tree (the southeast of the decomposition, its padding). The largest blocks are processed first, working down to the smallest. Padding blocks of the same size are processed at the same time.

Consider the padding list $[0, 0, 1, 2]$ and the reverse-Ahnentafel tree in Figure 1. Level 0 of the reverse-Ahnentafel tree represents the entire matrix S' which is of size 8. Since there is no padding of size 8, processing moves on to the next padding size. Level 1 represents blocks of size 4. Since there is no padding of size 4, processing recurs to blocks of size 2. There is one block of padding of size 2. The rightmost index from level 2 is pulled off the tree: 7. This value is the row and column indices of the identity matrix in the padding of S'. Since 7 is used for this block of padding, the subtree underneath it is now condemned. The two blocks of padding of size 1 get their indices from level 3 of the reverse-Ahnentafel tree. Since the subtree under 7 has been condemned, the indices come from the next rightmost tree: 13 and 9.

The indices are returned in a list of lists of reverse Ahnentafel indices. This list of lists is analogous to the list of padding counts. The first list is the reverse-Ahnentafel indices for padding the size of the entire matrix (which should always be empty), the last list is the reverse-Ahnentafel indices for padding of size 1, and the lists in between are for the padding sizes between 1 and the order of the matrix.

Finishing our example, given the list $[0, 0, 1, 2]$, `addresses_padding` returns the list $[[], [], [7], [13, 9]]$.

52a ⟨*Ahnentafel indices* 52a⟩≡
```
addresses_padding :: Int -> [Int] -> [[Int]]
addresses_padding _ []       =  []
addresses_padding i (x:xs)   =  sisterL i x : addresses_padding (niece i x) xs
```
Defines:
`addresses_padding`, used in chunk 57a.
Uses `niece` 53a and `sisterL` 53b.
This definition is continued in chunks 52 and 53.
This code is used in chunk 51.

The following functions compute reverse Ahnentafel indices given an index and a relationship. Male relationships are indices to the right in the binary tree, female relationships to the left. Siblings exist on the same level of the tree; children/parents are descendants/ancestors in the tree.

A sister of an index i is an index to the left of i in the reverse-Ahnentafel tree. The `skip` parameter allows any sister of a node to be selected.

52b ⟨*Ahnentafel indices* 52a⟩+≡
```
sister :: Int -> Int -> Int
sister i skip  =  reverse_bits ((reverse_bits i) - skip)
```
Defines:
`sister`, used in chunk 53.
Uses `reverse_bits` 53c.

The son of i is the right child of i.

52c ⟨*Ahnentafel indices* 52a⟩+≡
```
son :: Int -> Int
son i  =  i + (power2 (1 + (floor_lg i)))
```
Defines:
`son`, used in chunk 53a.
Uses `floor_lg` 48b and `power2` 48b.

The niece of i is the son of the sister of i. (The sexual nomenclature is still consistent, because the niece of i is an index that lies to the left of i in the tree.)

53a ⟨*Ahnentafel indices* 52a⟩+≡
```
niece :: Int -> Int -> Int
niece i skip  =  son (sister i skip)
```
Defines:
  `niece`, used in chunk 52a.
Uses `sister` 52b and `son` 52c.

The function `sisterL` computes a list of reverse-Ahnentafel sisters starting from a given index.

53b ⟨*Ahnentafel indices* 52a⟩+≡
```
sisterL :: Int -> Int -> [Int]
sisterL _ 0      =  []
sisterL i (n+1)  =  i : sisterL (sister i 1) n
```
Defines:
  `sisterL`, used in chunk 52a.
Uses `sister` 52b.

The function `reverse_bits` is used to reverse the bits to the right of to the most significant 1 in the number. For example, 001011 reverses to 001110.

Reversing the bits like this allows us to switch between a reverse-Ahnentafel index and its equivalent Ahnentafel index. (See Theorem 4 and Corollary 1 in [4].) Computing the sister of a reverse-Ahnentafel index i becomes conceptually (but not computationally) easy: convert i into its equivalent Ahnentafel index by applying `reverse_bits`, subtract 1 for the sister, and finally convert the new index back into a reverse-Ahnentafel index by reversing the bits.

The code here is based on the definition in [4, Section 1.2]. For efficiency, this function should really be machine code.

53c ⟨*Ahnentafel indices* 52a⟩+≡
```
reverse_bits :: Int -> Int
reverse_bits i  =  b
  where  (_, b)    =  helper i
         helper 1  =  (0, 1)
         helper i  =  let (lg, ahnen)  =  helper (halve i)
                          power        =  power2 lg
                          bit | (mod i 2) == 1  =  power
                              | otherwise       =  0  in
                        (lg+1, ahnen + power + bit)
```
Defines:
  `reverse_bits`, used in chunk 52b.
Uses `halve` 49a, `lg` 48b, and `power2` 48b.

C.3 Conversion from Series Encodings to Permutation Quadtree Matrices

The permutation matrices P and Q turn the decomposition S into the triangulation $L+U'$. The permutation matrices are constructed using series encodings Π and Ψ. Series encodings are lists of reverse-Ahnentafel indices, representing the location of the pivot blocks. (See [4, Sections 1.2 and 2.2].)

The function `convALtoPM` is the master routine to convert a reverse-Ahnentafel list to a permutation matrix. The actual permutation matrices are built by `build_p` and `build_q` which make P (for row permutations) and Q (for column permutations), respectively. Note that Π and Ψ are passed in as vectors. These vectors are the balanced vectors created by `list_to_tree` (see Section 7.2).

54a ⟨*Ahnentafel to permutation quadtree matrix* 54a⟩≡

```
convALtoPM :: Vectr Int -> Vectr Int -> (PMatrx, PMatrx)
convALtoPM pi psi  =  (build_p pi, build_q psi)
                           where ⟨Permutation conversion helpers 54b⟩
```

Defines:
`convALtoPM`, used in chunk 36.
Uses `PMatrx` 20b, `Vectr` 8a, `build_p` 54b, and `build_q` 54c.
This code is used in chunk 51.

The function `build_p` takes Π and builds P to make the row exchanges. The north half of the vector for Π is used for the north quadrants of P, the south half for the south quadrants. The east/west split is based on even/odd indices, respectively, in the vector. The functions `left_half` and `right_half` pull out the indices from a vector that belong to the left half or right half of the reverse-Ahnentafel tree, respectively. After pulling out an index, these functions also divide it by two so that the index is appropriate for the subquadrant.

54b ⟨*Permutation conversion helpers* 54b⟩≡

```
build_p :: Vectr Int -> PMatrx
build_p ZeroV        =  ZeroPM
build_p (ScalarV 1)  =  IdentPM
build_p (Vec piP)    =
  let  (nw_pi, sw_pi)  =  pmap left_half piP
       (ne_pi, se_pi)  =  pmap right_half piP  in
    normalizeQPM (qmap build_p (nw_pi, ne_pi, sw_pi, se_pi))
```

Defines:
`build_p`, used in chunk 54a.
Uses `IdentPM` 20b, `PMatrx` 20b, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroPM` 20b, `ZeroV` 8a, `left_half` 55a, `normalizeQPM` 20c, `pmap` 4a, `qmap` 5a, and `right_half` 55b.
This definition is continued in chunks 54 and 55.
This code is used in chunk 54a.

The function `build_q` is a dual to `build_p`; it takes a balanced vector holding Ψ to make Q for column exchanges. The methodology is the same for `build_q` as for `build_p` except that the vector halves distinguish between east and west quadrants while `left_half` and `right_half` distinguish between north and south quadrants.

54c ⟨*Permutation conversion helpers* 54b⟩+≡

```
build_q :: Vectr Int -> PMatrx
build_q ZeroV        =  ZeroPM
build_q (ScalarV 1)  =  IdentPM
build_q (Vec psiP)   =
  let  (nw_psi, ne_psi)  =  pmap left_half psiP
       (sw_psi, se_psi)  =  pmap right_half psiP  in
    normalizeQPM (qmap build_q (nw_psi, ne_psi, sw_psi, se_psi))
```

Defines:
`build_q`, used in chunk 54a.
Uses `IdentPM` 20b, `PMatrx` 20b, `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroPM` 20b, `ZeroV` 8a, `left_half` 55a, `normalizeQPM` 20c, `pmap` 4a, `qmap` 5a, and `right_half` 55b.

As mentioned above, `left_half` pulls out the even indices from a vector which represent the indices on the left side of a reverse-Ahnentafel tree. The function also halves the index, effectively moving it up one level (since the recursion has moved down one level).

55a ⟨*Permutation conversion helpers* 54b⟩+≡
```
left_half :: Vectr Int -> Vectr Int
left_half ZeroV        =  ZeroV
left_half (ScalarV n)
    | even n     =  ScalarV (halve n)
    | otherwise  =  ZeroV
left_half (Vec p)  =  normalizePV (pmap left_half p)
```
Defines:
`left_half`, used in chunk 54.
Uses `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroV` 8a, `halve` 49a, `normalizePV` 8b, and `pmap` 4a.

The function `right_half` is a dual to `left_half`. The difference is that it looks for odd numbered indices which are reverse-Ahnentafel indices in the right main subtree.

55b ⟨*Permutation conversion helpers* 54b⟩+≡
```
right_half :: Vectr Int -> Vectr Int
right_half ZeroV        =  ZeroV
right_half (ScalarV n)
    | odd n      =  ScalarV (halve n)
    | otherwise  =  ZeroV
right_half (Vec p)  =  normalizePV (pmap right_half p)
```
Defines:
`right_half`, used in chunk 54.
Uses `ScalarV` 8a, `Vec` 8a, `Vectr` 8a, `ZeroV` 8a, `halve` 49a, `normalizePV` 8b, and `pmap` 4a.

D Padding the Triangulation

To move the pivot blocks onto the main diagonal, it is essential that the pivot blocks not straddle quadtree boundaries. To prevent this, padding is added between elimination blocks in the triangulation. (See [4, Section 3.2].)

For example, let S be a 4×4 matrix. Let $\Omega = [1, 2, 1]$. The first pivot block (the first 1 in Ω) is supposed to appear in the southeast corner of the triangulation. Since the triangulation is empty, this is no problem.

The next pivot block is supposed to be placed to the northwest of the first. However, the second pivot block is of size 2. Placing it directly to the northwest of the first block will force it to straddle a quadtree boundary. Thus, we add a 1×1 block of padding between the two pivot blocks. Thus, the first pivot block and the padding make up a 2×2 block which is paired off conveniently with the second pivot block.

The last pivot block starts a new quadrant, padded with blocks of size 1 and 2 to make its total size be 4. Thus, the final matrix is of size 8.

In this example, the size of the matrix doubled. Due to zero and identity compression, it is not much of a problem. As mentioned in [4, Section 3.2], such doubling occurs very rarely in practice.

This task is accomplished by paddding Ω out into a *balanced vector*, described in [4, Section 3.2]. A balanced vector stores integers such that the sum of them is a power of 2. For a given vector which sums to 2^p, its north and south halves sum up to 2^{p-1}. The code in this section adds integers, representing padding blocks, to Ω so that the balanced vector can be made. Converting the list into a vector is done by `list_to_tree` in Section 7.2.

While there is a lot of complicated code to complete this task, it takes very little time to run, especially in proportion to the running time of the whole matrix inversion.

D.1 File Layout

56a ⟨`padding.gs` 56a⟩≡
  ⟨*Padding data types definitions* 56b⟩
  ⟨*Master padding routine* 57a⟩
Root chunk (not used in this document).

D.2 Padding Data Types

The `AhnenOrdList` data type represents of list of `AhnenOrd`s. An `AhnenOrd` is a triplet storing reverse-Ahnentafel indices i and j and a pivot order o. This means that an `AhnenOrdList` effectively stores away Π, Ψ and Ω.

The `PaddingLists` data type is similar to the `AhnenOrdList` data type. However, `PaddingLists` is a tuple of lists whereas `AhnenOrdList` is a list of tuples. In addition to what amount to Π, Ψ and Ω, `PaddingLists` also stores a list of Boolean flags. The Boolean flags keep track of which blocks are padding and which are data from the original matrix.

56b ⟨*Padding data types definitions* 56b⟩≡
```
type AhnenOrd = (Int, Int, Int)
type AhnenOrdList = [AhnenOrd]
type PaddingLists = ([Bool], [Int], [Int], [Int])
```
Defines:
  `AhnenOrd`, never used.
  `AhnenOrdList`, used in chunks 38, 57, and 58a.
  `PaddingLists`, used in chunks 57–59.
This code is used in chunk 56a.

D.3 Padding Functions

The function `insert_padding` is the top level function for adding the padding. Its job is to find out where padding is necessary and add the necessary reverse-Ahnentafel indices to the series encodings.

57a ⟨*Master padding routine* 57a⟩≡
```
insert_padding :: Int -> AhnenOrdList -> (Int, ([Int], [Int], [Int]))
insert_padding ord aoL  =
  let  ord'          =  power2 (ceil_lg ord)
       (pad_flagL, pi, psi, omega)
         =  expand_as_necessary ord' 0 aoL ([], [], [], [])
       full_ord      =  sum omega
       pad_count     =  reverse (counts_padding pad_flagL omega full_ord)
       pad_addrL     =  addresses_padding 1 pad_count
       pi'           =  replace_size_with_index pad_flagL pi pad_addrL
       psi'          =  replace_size_with_index pad_flagL psi pad_addrL  in
    (full_ord, (pi', psi', omega))
      where ⟨Indexing code 60b⟩
            ⟨Expanding code 57b⟩
```
Defines:
`insert_padding`, used in chunk 36.
Uses `AhnenOrdList` 56b, `addresses_padding` 52a, `ceil_lg` 48b, `counts_padding` 59c, `expand_as_necessary` 57b, `power2` 48b, and `replace_size_with_index` 60b.
This code is used in chunk 56a.

The function `expand_as_necessary` expands the original matrix by adding padding blocks to Ω, Π and Ψ. The indices put into Π and Ψ are actually filler values. The sentinel values are replaced with usable indices by `replace_size_with_index` once the final matrix size is known.

57b ⟨*Expanding code* 57b⟩≡
```
expand_as_necessary :: Int -> Int -> AhnenOrdList -> PaddingLists
  -> PaddingLists
expand_as_necessary full_ord cum_ord aoL@((_, _, o) : aoL') padL
  | res == []  =  let  (_, _, _, omega')  =  padL'
                       full_ord'  =  full_ord - (sum omega')  in
                    extend_it full_ord' padL' full_ord 1
  | otherwise  =
      expand_as_necessary (twice full_ord) cum_ord res (expand padL')
    where  (res, padL')  =  trav_omega full_ord o cum_ord aoL padL
```
Defines:
`expand_as_necessary`, used in chunk 57a.
Uses `AhnenOrdList` 56b, `PaddingLists` 56b, `expand` 59b, `extend_it` 58b, `trav_omega` 58a, and `twice` 49a.
This definition is continued in chunks 58 and 59.
This code is used in chunk 57a.

The function trav_omega works down Ω, adding padding where necessary. The base case is triggered when the cumulative order equals the full order, indicating that processing is finished. The first recursive case is invoked when there is no padding needed to place the next pivot block. The last case adds the necessary padding using extend_it.

58a ⟨*Expanding code* 57b⟩+≡
```
  trav_omega :: Int-> Int-> Int -> AhnenOrdList -> PaddingLists
    -> (AhnenOrdList, PaddingLists)
  trav_omega _ _ _ [] padL  =  ([], padL)
  trav_omega full_ord next_ord cum_ord aoL@((i, j, piv_ord) : aoL')
             padL@(pad_flagL, pi, psi, omega)
      | full_ord == cum_ord  =  (aoL, padL)
      | pad_size == 0        =
          let  next_ord' | aoL' == []  =  0
                         | otherwise   =  piv_ord'
                              where  (_, _, piv_ord') : _  =  aoL'
               padL'     =  (False : pad_flagL, i : pi, j : psi, piv_ord : omega)
               cum_ord'  =  next_ord + cum_ord  in
            trav_omega full_ord next_ord' cum_ord' aoL' padL'
      | otherwise          =
          let  cum_ord'  =  pad_size + cum_ord
               padL'     =  extend_it pad_size padL full_ord 1  in
            trav_omega full_ord next_ord cum_ord' aoL padL'
        where  pad_size  =  mod cum_ord next_ord
```
Defines:
  trav_omega, used in chunk 57b.
Uses AhnenOrdList 56b, PaddingLists 56b, and extend_it 58b.

The function extend_it adds padding to Ω. Its first parameter pad_size is an integer which is the amount of padding to be added to Ω. The PaddingLists holds the padding flags, Ω, Π and Ψ.

The index parameter is supposed to be the index of the southeast block where the padding will come from. Currently, this index is just a sentinel value (the current full size of the matrix). The actual index cannot be determined without knowing if the matrix is going to double in size. The actual index is added later by replace_size_with_index.

The last parameter is a power of 2 used to compare with pad_size. If the power of 2 is contained in pad_size, then a padding block of that power of 2 is required.

At the end of its execution, extend_it returns a new PaddingLists with each of its list extended appropriately.

58b ⟨*Expanding code* 57b⟩+≡
```
  extend_it :: Int -> PaddingLists -> Int -> Int -> PaddingLists
  extend_it pad_size paddingL index twopower
      | twopower > pad_size                    =  paddingL
      | fixnumlogand twopower pad_size == 0   =
          extend_it pad_size paddingL (halve index) (twice twopower)
      | otherwise      =
          let  paddingL'  =  extend index twopower paddingL
               index'     =  halve index
               twopower'  =  twice twopower  in
            extend_it pad_size paddingL' index' twopower'
```
Defines:
  extend_it, used in chunks 57b and 58a.
Uses PaddingLists 56b, extend 59a, fixnumlogand 49b, halve 49a, and twice 49a.

The function extend does the extending of the PaddingLists for extend_it. It takes an index for the padding block (a sentinel value), a power of two, and a PaddingLists. A new PaddingLists is made: a True flag is added to the padding flags, the index is added to the series encodings Π and Ψ, and the block size is added to Ω.

59a ⟨*Expanding code* 57b⟩+≡
```
extend :: Int -> Int -> PaddingLists -> PaddingLists
extend index twopower (pad_flagL, pi, psi, omega)  =
  (True : pad_flagL, index : pi, index : psi, twopower : omega)
```
Defines:
extend, used in chunk 58b.
Uses PaddingLists 56b.

Sometimes it becomes necessary to double the size of the matrix due to the extra padding. (See the example at the beginning of this appendix.) When this happens, the reverse-Ahnentafel indices are recomputed, moving them down one level and into the left main subtree. (See Figure 1.) The function expand does exactly this by multiplying the index by two. (See Theorem 5 of [4].)

59b ⟨*Expanding code* 57b⟩+≡
```
expand :: PaddingLists -> PaddingLists
expand (pad_flagL, pi, psi, omega) =
  let  pi'   =  zipWith helper pad_flagL pi
       psi'  =  zipWith helper pad_flagL psi  in
    (pad_flagL, pi', psi', omega)
       where helper True  index  =  index
             helper False index  =  twice index + 1
```
Defines:
expand, used in chunk 57b.
Uses PaddingLists 56b and twice 49a.

The function counts_padding counts the padding in a matrix by block size. That is, in the example in Section C.2, the list $[0, 0, 1, 2]$ is such a list, stating that the resulting 8×8 matrix has no pivots of size 8 or size 4, one of size 2, and two of size 1.

59c ⟨*Expanding code* 57b⟩+≡
```
counts_padding :: [Bool] -> [Int] -> Int -> [Int]
counts_padding pad_flagL omega full_ord  =
  helper pad_flagL omega (fillzeros full_ord)
    where helper [] _ ans  =  ans
          helper (pad_flag:pad_flagL) (ord:ordL) ans
            | pad_flag    =  helper pad_flagL ordL (tally_padding ord ans)
            | otherwise   =  helper pad_flagL ordL ans
          ⟨Helper functions for counts_padding 59d⟩
```
Defines:
counts_padding, used in chunk 57a.
Uses fillzeros 59d and tally_padding 60a.

The function fillzeros returns a list of zeros whose length is the logarithm of the order of the matrix. This forms the basis for the padding count list which counts_padding returns.

59d ⟨*Helper functions for* counts_padding 59d⟩≡
```
fillzeros :: Int -> [Int]
fillzeros 1    =  [0]
fillzeros ord  =  0 : fillzeros (halve ord)
```
Defines:
fillzeros, used in chunk 59c.
Uses halve 49a.
This definition is continued in chunk 60a.
This code is used in chunk 59c.

The function tally_padding works itself down a list of padding counts to find the appropriate tally for a given order. Once this tally is found, it is incremented.

60a ⟨*Helper functions for* counts_padding 59d⟩+≡
```
tally_padding :: Int -> [Int] -> [Int]
tally_padding _   []     =  error "tally padding"
tally_padding 1   (x:xs) =  x+1 : xs
tally_padding ord (x:xs) =  x   : tally_padding (halve ord) xs
```
Defines:
  tally_padding, used in chunk 59c.
Uses halve 49a.

The function replace_size_with_index replaces the sentinel value with the reverse-Ahnentafel index for padding blocks for the series encodings Π and Ψ. While adding the padding, it is not known whether or not the matrix size will double. However, the final size of the matrix must be known before the indices for the padding are added to the series encodings. Thus, replace_size_with_index is used to replace the sentinel values with the correct indices.

The function replace_size_with_index uses the list returned by addresses_padding defined in Section C.2. When a padding block is found (signified by a flag in the padding list), the appropriate index for the padding size is taken from the reverse-Ahnentafel list for padding and inserted into the series encodings in place of the sentinel value.

60b ⟨*Indexing code* 60b⟩≡
```
replace_size_with_index :: [Bool] -> [Int] -> [[Int]] -> [Int]
replace_size_with_index _      []     _         = []
replace_size_with_index (pad_flag:pad_flagL) (i:pi) pad_addrL
  | pad_flag    =  let  (i', pad_addrsL')  =  extract i pad_addrL  in
                    i' : (replace_size_with_index pad_flagL pi pad_addrsL')
  | otherwise  =  i : (replace_size_with_index pad_flagL pi pad_addrL)
    where ⟨Pad size extraction 60c⟩
```
Defines:
  replace_size_with_index, used in chunk 57a.
Uses extract 60c.
This code is used in chunk 57a.

The function extract takes a padding size and a list of lists of reverse-Ahnentafel indices as returned by addresses_padding. The list of lists is deconstructed until the appropriate row for the given size is found. The first index there is removed from the list of lists and returned as the index for the padding.

60c ⟨*Pad size extraction* 60c⟩≡
```
extract :: Int -> [[Int]] -> (Int, [[Int]])
extract 0 _              =  error "Exhausted padding list."
extract _ []             =  error "Exhausted padding list."
extract 1 ((i:pi):piL)  =  (i, pi : piL)
extract size (pi:piL)      =
  let  (i, piL')  =  extract (halve size) piL  in
    (i, pi : piL')
```
Defines:
  extract, used in chunk 60b.
Uses halve 49a.
This code is used in chunk 60b.

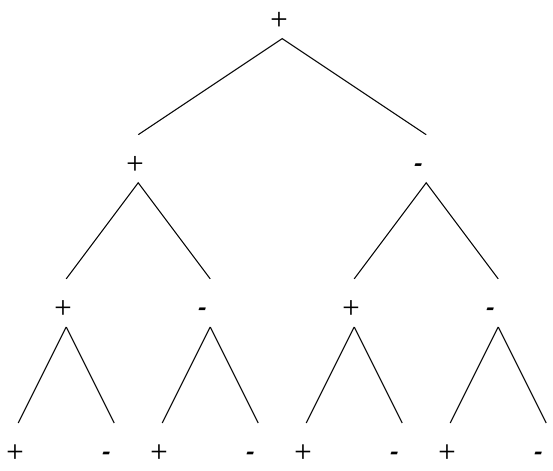

Figure 2: Parity tree

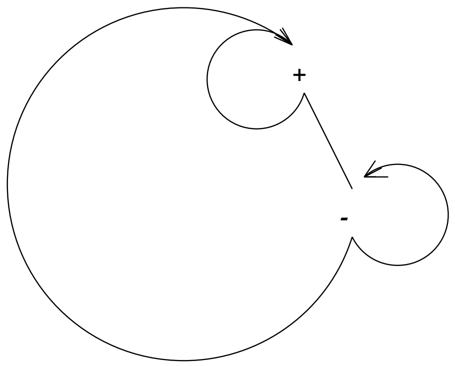

Figure 3: Simulated parity tree

E Parity of a Permutation Matrix

Parity is determined by looking at the series encodings for permutation matrices. (See [4, Section 2.2].) Each of the reverse-Ahnentafel indices in the series encodings represents an identity block in the permutation matrix. The reverse-Ahnentafel indices can be used with the parity tree in Figure 2 to compute the parity of a permutation matrix represented in a series encoding.

However, the full parity tree can actually be stored in a much simpler tree, as seen in Figure 3. When the parity tree changes (as explained below), the parity tree is reconstructed into a fuller form.

To compute the parity of a permutation matrix based on its series encoding, the reverse-Ahnentafel indices are examined one at a time, starting with a positive cumulative parity. For a reverse-Ahnentafel index i, the corresponding node for i in the parity tree is examined. The node in the parity tree reflects the parity associated with that position when the determinant of the matrix is computed by minors. The positive and negative signs for these indices are mupltiplied together for the parity of the permutation matrix.

However, there are two twists on this. Once the parity for i is retrieved, the subtree under the node for index i is dead — identity matrices have one entry per row and one per column. Thus the retrieved node is nulled out of the parity tree. In addition, all of the nodes leading down to i should never be used again, so their parities are zeroed.

Secondly, the parities to the right of i must all flip signs. The reason for this is best seen through an ex-

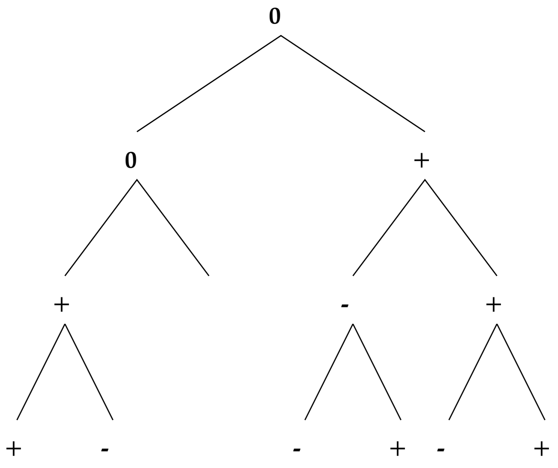

Figure 4: Parity tree after accessing node 6

ample. Consider the permutation matrix

$$\begin{bmatrix} 0 & I_6 & 0 & 0 \\ 0 & 0 & 0 & I_7 \\ I_4 & 0 & 0 & 0 \\ 0 & 0 & I_5 & 0 \end{bmatrix}$$

where I_i is an identity matrix with reverse-Ahnentafel index i. When computing a determinant by minors, each column is assigned a parity. The parity tree in Figure 2 assigns the proper parities to the different positions: positive for I_4 and I_5, negative for I_6 and I_7.

To compute the parity of a permutation matrix, the matrix is analyzed from the first row down to the last. In this example, the first index is 6 which, as already seen, has a negative parity. This is taken as the cumulative parity, and processing recurs to the minor of this matrix after eliminating the first row and second column. This results in the minor

$$\begin{bmatrix} 0 & 0 & I_7 \\ I_4 & 0 & 0 \\ 0 & I_5 & 0 \end{bmatrix}$$

where the indices of I_5 and I_7 do not change, even though they are technically wrong for this matrix. In this new matrix, I_4 and I_7 have positive parities, and I_5 has a negative parity. Comparing this with the parity tree in Figure 2 shows that the parities of I_5 and I_7 have flipped. In fact, this holds true for any index to the right of 6 in the reverse-Ahnentafel tree.

So all signs to the right of 6, the probed index, are flipped. However, this flip is done is a lazy fashion. When subsequent probes require that previously flipped subtrees be flipped again, the two flips are annihilated.

See Figure 4 for the result of probing index 6 in the parity tree.

E.1 File Layout

62 ⟨`parity.gs` 62⟩≡
  ⟨*Parity tree definition* 63a⟩
  ⟨*Parity tree inheritance* 63b⟩
  ⟨*Parity functions* 63c⟩

Root chunk (not used in this document).

E.2 Parity Data Types

The parity data type `ParSign` keeps track of the parity at each node: positive, negative and zero parity. In the last case, this is for the parents of probed nodes which can no longer be used for their parity.

The parity tree can either be null or a node with two children. The boolean value stored at each active node indicates whether or not the subtree needs to be negated due to previous probings. The parity sign is the sign for that node.

63a ⟨*Parity tree definition* 63a⟩≡

```
data ParSign  =  PositivePar | NegativePar | ZeroPar
data ParTree  =  NullPar | Branch Bool ParSign ParTree ParTree
```

Defines:
`Branch`, used in chunks 63–65.
`NegativePar`, used in chunks 63 and 65b.
`NullPar`, used in chunks 63–65.
`ParSign`, used in chunks 63b and 65b.
`ParTree`, used in chunks 63–65.
`PositivePar`, used in chunks 63 and 65b.
`ZeroPar`, used in chunks 63–65.
This code is used in chunk 62.

`Eq` inheritance is needed for pattern matching.

63b ⟨*Parity tree inheritance* 63b⟩≡

```
instance Eq ParSign where
  PositivePar  == PositivePar  =  True
  NegativePar  == NegativePar  =  True
  ZeroPar      == ZeroPar      =  True
  _            == _            =  False

instance Eq ParTree where
  NullPar == NullPar  =  True
  _       == _        =  False
```

Uses `NegativePar` 63a, `NullPar` 63a, `ParSign` 63a, `ParTree` 63a, `PositivePar` 63a, and `ZeroPar` 63a.
This code is used in chunk 62.

E.3 Parity Functions

The function `parity` is the top level function which computes the parity of a permutation matrix based on its series encoding. The `p` and `q` defined here represent the parity tree pictured in Figure 3.

63c ⟨*Parity functions* 63c⟩≡

```
parityAL :: Eq ParTree => [Int] -> Int -> Int
parityAL ahnenList order  =
  all_probes ahnenList (power2 (ceil_lg order)) p 1
    where  p  =  Branch False PositivePar p q
           q  =  Branch False NegativePar p q

⟨Parity support functions 64a⟩
```

Defines:
`parityAL`, used in chunks 36 and 43a.
Uses `Branch` 63a, `NegativePar` 63a, `ParTree` 63a, `PositivePar` 63a, `all_probes` 64a, `ceil_lg` 48b, and `power2` 48b.
This code is used in chunk 62.

The function `all_probes` works itself down the series encoding of a permutation matrix. If the series encoding is exhausted, the accumulated parity is returned. Otherwise, the first index in the series encoding is examined. If the index refers to padding, it can be ignored since it is not truly part of the matrix. If the node is null, an error has occured. Otherwise, the first index in the list is examined by `probe` which returns an accumulated parity and an updated parity tree.

64a ⟨*Parity support functions* 64a⟩≡

```
all_probes :: Eq ParTree => [Int] -> Int -> ParTree -> Int -> Int
all_probes []       _         _     ans   =  ans
all_probes (i:iL) min_index tree ans
  | i < min_index     =  all_probes iL min_index tree ans
  | tree == NullPar   =  error "Exhausted parity tree."
  | otherwise         =  let  (ans', tree')  =  probe i tree False  in
                           all_probes iL min_index tree' (ans' * ans)
```

Defines:
  `all_probes`, used in chunk 63c.
Uses `NullPar` 63a, `ParTree` 63a, and `probe` 64b.
This definition is continued in chunk 64b.
This code is used in chunk 63c.

The function `probe` takes one reverse-Ahnentafel index and probes that node in the parity tree. The recursive case works itself down the parity tree, using i to guide it. Once the node has been found, the actual parity for the node is computed by `node_parity`. Then processing backs out of the recursion, rebuilding the parity tree using `revise_tree`.

64b ⟨*Parity support functions* 64a⟩+≡

```
probe :: Eq ParTree => Int -> ParTree -> Bool -> (Int, ParTree)
probe _ NullPar _       =  (1, NullPar)
probe i tree@(Branch lazy _ leftT rightT) inverted
  | i == 1      =  node_parity inverted tree
  | even i      =
      let  (sgnm, leftT')  =  probe (div i 2) leftT (xor inverted lazy)  in
        if (rightT == NullPar) && (leftT' == NullPar)
        then (sgnm, NullPar)
        else let  rightT'  =  revise_tree rightT (inverted == lazy)  in
               (sgnm, Branch False ZeroPar leftT' rightT')
  | otherwise  =
      let  (sgnm, rightT')  =  probe (div i 2) rightT (xor inverted lazy)  in
        if (leftT == NullPar) && (rightT' == NullPar)
        then (sgnm, NullPar)
        else let  leftT'  =  revise_tree leftT (xor inverted lazy)  in
               (sgnm, Branch False ZeroPar leftT' rightT')
    where ⟨Probe support functions 65a⟩
```

Defines:
  `probe`, used in chunk 64a.
Uses `Branch` 63a, `NullPar` 63a, `ParTree` 63a, `ZeroPar` 63a, `node_parity` 65a, `revise_tree` 65c, and `xor` 49b.

The function node_parity returns a -1 or $+1$ based on the parity of the given node and the negation flag. In addition to the actual parity, a new node is passed back to replace the given node.

65a ⟨*Probe support functions* 65a⟩≡
```
node_parity :: Bool -> ParTree -> (Int, ParTree)
node_parity inverted NullPar  =  error "Series encoding is not a permutation."
node_parity inverted tree@(Branch flag sgn _ _)
  | sgn == ZeroPar      =  error "Series encoding is not a permutation."
  | xor flag inverted   =  (par_negate sgn, NullPar)
  | otherwise           =  (par_ident sgn, NullPar)
```
Defines:
node_parity, used in chunk 64b.
Uses Branch 63a, NullPar 63a, ParTree 63a, ZeroPar 63a, par_ident 65b, par_negate 65b, and xor 49b.
This definition is continued in chunk 65.
This code is used in chunk 64b.

The functions par_negate and par_ident are used to convert from a ParSign to an integer (appropriate for arithmetic).

65b ⟨*Probe support functions* 65a⟩+≡
```
par_negate :: ParSign -> Int
par_negate PositivePar  =  -1
par_negate NegativePar  =  1
par_negate ZeroPar      =  0

par_ident :: ParSign -> Int
par_ident PositivePar  =  1
par_ident NegativePar  =  -1
par_ident ZeroPar      =  0
```
Defines:
par_ident, used in chunk 65a.
par_negate, used in chunk 65a.
Uses NegativePar 63a, ParSign 63a, PositivePar 63a, and ZeroPar 63a.

The function revise_tree revises the parity tree, setting the negation flag appropriately.

65c ⟨*Probe support functions* 65a⟩+≡
```
revise_tree :: ParTree -> Bool -> ParTree
revise_tree NullPar _ = NullPar
revise_tree tree@(Branch flag sgn leftT rightT) lazy  =
  Branch (xor flag lazy) sgn leftT rightT
```
Defines:
revise_tree, used in chunk 64b.
Uses Branch 63a, NullPar 63a, ParTree 63a, and xor 49b.

F Index

F.1 Macros

F.2 Identifiers

References

[1] J. Fasel, P. Hudak, S. Peyton Jones & P. Wadler (eds.). HASKELL special issue. *ACM SIGPLAN Notices* **27**, 5 (May 1992).

[2] Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Computer Science Department, Yale University (May 1994).

[3] Norman Ramsey. Literate programming simplified. *IEEE Software* **11**, 5 (Sept 1994). pp. 97–105.

[4] David S. Wise. Undulant-block pivoting and integer-preserving matrix inversion. Technical Report 418, Computer Science Department, Indiana University (January 1995).