Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

\[
\text{accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]}
\]
\[
= \text{listArray (0,2) [101,103,104]}
\]
Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

\[
\text{accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]} \\
= \text{listArray (0,2) [101,103,104]}
\]

\[
\text{accumArray (+) 100 (0,2) [(0,1),(1,3),(0,4)]} \\
= \text{listArray (0,2) [105,103,100]}
\]
Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

\[
\text{accumArray (+) 100 (0,2) } \[(0,1),(1,3),(2,4)\] \\
= \text{listArray (0,2) } [101,103,104]
\]

\[
\text{accumArray (+) 100 (0,2) } \[(0,1),(1,3),(0,4)\] \\
= \text{listArray (0,2) } [105,103,100]
\]

- The type of `accumArray` is

\[
\text{accumArray :: Ix i => (a -> b -> a) -> a ->} \\
\rightarrow (i,i) \rightarrow \{i,b\} \rightarrow \text{Array } i \text{ a}
\]
The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

\[ \text{accumArray} (+) 100 (0,2) [(0,1),(1,3),(2,4)] \]
\[ = \text{listArray} (0,2) [101,103,104] \]

\[ \text{accumArray} (+) 100 (0,2) [(0,1),(1,3),(0,4)] \]
\[ = \text{listArray} (0,2) [105,103,100] \]

The type of `accumArray` is

\[ \text{accumArray} :: \text{Ix } i \Rightarrow (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow \]
\[ \rightarrow (i,i) \rightarrow [(i,b)] \rightarrow \text{Array } i \text{ a} \]
Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]
= listArray (0,2) [101,103,104]
```

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(0,4)]
= listArray (0,2) [105,103,100]
```

- The type of `accumArray` is

```
accumArray :: Ix i => (a -> b -> a) -> a ->
             -> (i,i) -> [(i,b)] -> Array i a
```

- Also works in linear time on the length of the associative list plus the range.
An old example: \texttt{minout}

\begin{itemize}
  \item \texttt{minout :: [Int] -> Int}
  \item \texttt{minout \ \texttt{l} \ is \ the \ minimum \ nonnegative \ number \ not \ in \ \texttt{l}}
  \item assuming that all elements in \texttt{l} are nonnegative and distinct.
    \begin{itemize}
      \item \texttt{minout [3,1,2] = 0}
      \item \texttt{minout [1,5,3,0,2] = 4}
      \item \texttt{minout [11,5,3,0] = 1}
    \end{itemize}
  \item How do we compute \texttt{minout}?
\end{itemize}
An old example: \texttt{minout}

\begin{itemize}
  \item \texttt{minout} :: [Int] \rightarrow Int
  \texttt{minout} \; l \text{ is the minimum nonnegative number not in } l
  \text{ assuming that all elements in } l \text{ are nonnegative and distinct.}
  \begin{itemize}
    \item \texttt{minout} \; [3,1,2] = 0
    \item \texttt{minout} \; [1,5,3,0,2] = 4
    \item \texttt{minout} \; [11,5,3,0] = 1
  \end{itemize}
  \item How do we compute \texttt{minout}?
  \item The linear time solution via lists involved a rather clever divide and conquer algorithm.
An old example: **minout**

- **minout :: [Int] -> Int**
- **minout l** is the minimum nonnegative number not in **l** assuming that all elements in **l** are nonnegative and distinct.
  - **minout [3,1,2] = 0**
  - **minout [1,5,3,0,2] = 4**
  - **minout [11,5,3,0] = 1**

- How do we compute **minout**?

- The linear time solution via lists involved a rather clever divide and conquer algorithm.

- With arrays the solution is simpler
Our strategy is the following. Let $ln$ be the length of the given list $ls$. Initialize an array with indices $0, \ldots, ln-1$ with $0$. Create an associative list $\{(i,1) | i \leftarrow ls, 0 \leq i, i \leq ln-1\}$. Accumulate values from this associative list using the function $f x y = y$. The index of the first entry in the array with $0$ is the answer.
Our strategy is the following. Let \( l_n \) be the length of the given list \( ls \)

- Initialize an array with indices \( 0, \ldots, l_n-1 \) with 0.
minout via arrays

- Our strategy is the following. Let $\ln$ be the length of the given list $\mathbf{l}$.
  - Initialize an array with indices $0, \ldots, \ln-1$ with 0.
  - Create an associative list $[(i,1) \mid i \leftarrow \mathbf{l}, 0 \leq i, i \leq \ln-1]$.
Our strategy is the following. Let $l_n$ be the length of the given list $\text{ls}$

- Initialize an array with indices 0, \ldots, $l_n$-1 with 0.
- Create an associative list
  $$[(i,1) \mid i \leftarrow \text{ls}, 0 \leq i, i \leq l_n-1]$$
- Accumulate values from this associative list using the function
  $$f \ x \ y = y$$
Our strategy is the following. Let \( l_n \) be the length of the given list \( l_s \)

- Initialize an array with indices \( 0, \ldots, l_n-1 \) with 0.
- Create an associative list
  \[ [(i,1) \mid i \leftarrow l_s, 0 \leq i, i \leq l_n-1] \]
- Accumulate values from this associative list using the function
  \[ f \ x \ y = y \]
- The index of the first entry in the array with 0 is the answer.
minout via arrays ...

import Data.Array

myArray ls = accumArray f 0 (0,ln)
   [(i,1) | i <- ls, 0 <= i, i <= ln-1]
where
  ln = length ls
  f x y = y

firstZero :: Array Int Int -> Int -> Int
firstZero ar i
  | (ar!i == 0) = i
  | otherwise = firstZero ar (i+1)

minout ls = firstZero (myArray ls) 0
Two dimensional arrays

- The definition of an array makes no reference to a dimension.

```haskell
idMat n = accumArray f 0 ((0,0),(n-1,n-1))
    [((i,i),1) | i <- [0..(n-1)]]
where f x y = y
```
Two dimensional arrays

- The definition of an array makes no reference to a dimension.

- So, two or $k$-dimensional arrays are essentially same, with just a different sent of indices.

Here is way to generate an $n \times n$ identity matrix.

```haskell
idMat n = accumArray f 0 ((0,0),(n-1,n-1))
    [ ((i,i),1) | i <- [0..(n-1)] ]
where f x y = y
```
Two dimensional arrays

- The definition of an array makes no reference to a dimension.

- So, two or $k$-dimensional arrays are essentially same, with just a different set of indices.

- Here is way to generate an $n \times n$ identity matrix.

```haskell
idMat n = accumArray f 0 ((0,0),(n-1,n-1))
    [((i,i),1) | i <- [0..(n-1)]]
where
    f x y = y
```