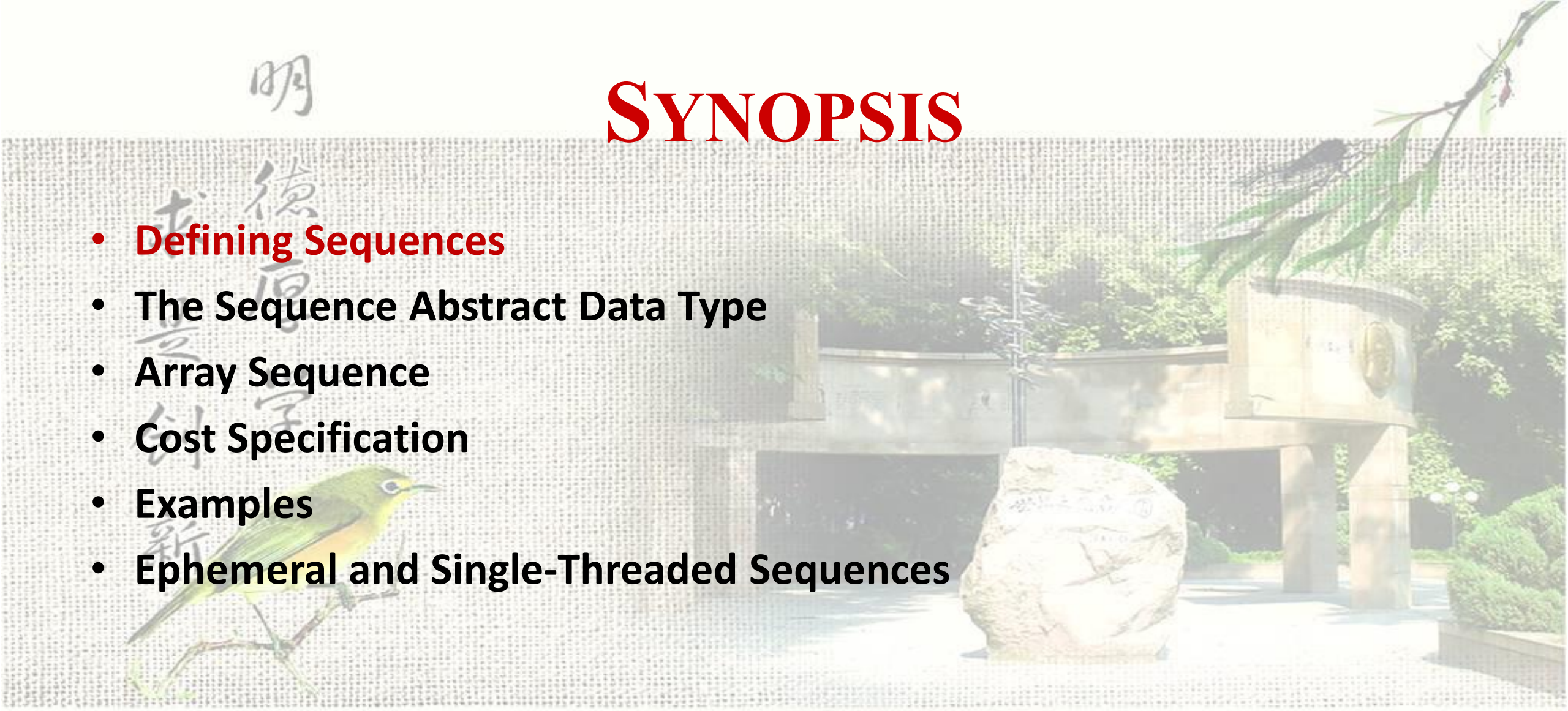
Parallel and Sequential Algorithms and Data Structures

LECTURE 4 SEQUENCE

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

SYNOPSIS

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



Definition

- An α **sequence** is a mapping (function) from $\mathbb{N}$ to α with domain {0, …, n-1 } for some $n \in \mathbb{N}$
  - A sequence is an **ordered set**, i.e., is a **collection** of elements that are totally ordered
    - ✓Use N to indicate the natural numbers, including zero
    - ✓Traditionally sequences are indexed from 1 not 0, but being computer scientists, we violate the tradition here
    - ✓e.g., < a, b, c >

Sequence

- Let A = {0, 1, 2, 3} and B = {a, b, c}
- The relation *R = {(0, a), (1, b), (3, a)}*
  - Is function? Is sequence?
- The relation *Z = {(1, b), (3, a), (2, a), (0, a)}* from A to B
  - Is function? Is sequence?

Relations and Functions

- A (binary) **relation** *from a set A to set B* is a subset of the Cartesian product of A and B
  - For a relation $R \subseteq A \times B$, the set $\{a : (a, b) \in R\}$ is referred to as the **domain** of R, and the set $\{b: (a, b) \in R\}$ is referred to as the **range** of R
- A **mapping** from A to B is a relation $R \subset A \times B$ such that $|R|=|\text{domain}(R)|$, i.e., *for every a in the domain of R there is only one b such that* $(a,b) \in R$
  - A mapping is also called a **function**

How to understand "every"?

Sequences

- The notation $< a_0, a_1, \ldots, a_{n-1} >$ is shorthand for the sequence $\{(0, a_0), (1, a_1), \ldots, ((n-1), a_{n-1})\}$
  - $a[i]$ refers to the element of a at position i
  - $a[l \cdots h]$ refers to the subsequence of a restricted to the position between l and h
- Since they occur frequently, we use special notation and terminology for sequences with two elements and sequences of characters
  - An *ordered pair (a, b)* is a pair of elements in which the element on the left, *a*, is identified as the first entry, and the one on the right, *b*, as the second entry
  - We refer to a sequence of charters as a *string*, and use the standard double-quote syntax for them, e.g., "$c_0c_1c_2\ldots c_{n-1}$" is a string consisting of the n characters $c_0, c_1, c_2, \ldots, c_{n-1}$

SEQUENCE COMPREHENSION

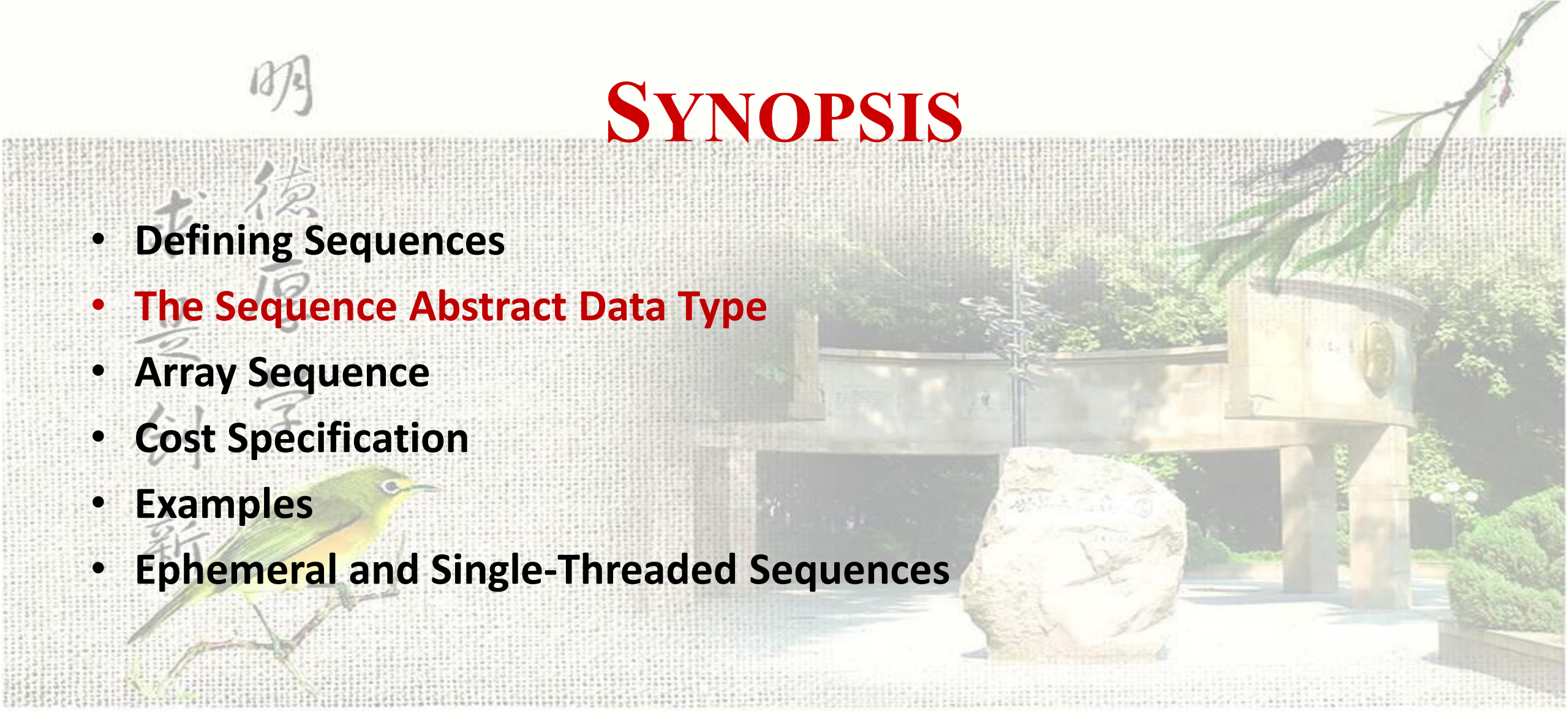
- *For the integer sequence (or $\mathbb{Z}$ sequence),* $A = \langle 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 \rangle$, *we have* $A[0] = 2$, $A[2] = 5$, *and* $A[1, 4] = \langle 3, 5, 7, 11 \rangle$.
- *A character sequence, or a string:* $\langle \texttt{'s'}, \texttt{'e'}, \texttt{'q'} \rangle \equiv \texttt{"seq"}$.
- *An (integer* $\times$ *string) sequence:* $\langle (10, \texttt{"ten"}), (1, \texttt{"one"}), (2, \texttt{"two"}) \rangle$.
- *A (string sequence) sequence:* $\langle \langle \texttt{"a"} \rangle, \langle \texttt{"nested"}, \texttt{"sequence"} \rangle \rangle$.
- *A function sequence, or more specifically a* $(\mathbb{Z} \to \mathbb{Z})$ *sequence:*

$$\langle (\textbf{\textit{fn}}\ x \Rightarrow x^2), (\textbf{\textit{fn}}\ y \Rightarrow y + 2), (\textbf{\textit{fn}}\ x \Rightarrow x - 4) \rangle \ .$$

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

SYNOPSIS

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



THE SEQUENCE ADT

- shows the specification of an abstract data type for sequences
  - For a value type α, the sequence data type is the type S_α consisting the set of all α sequences
  - B = {true, false} and O = {less, greater, equal}
  - i is a variable ranging over natural numbers
  - x is a variable ranging over the elements of a sequence
  - e is a SPARC expression, e_n and e'_n are SPARC expressions whose values are natural numbers, e_s is a SPARC expression whose value is a sequence
  - p is a SPARC pattern that binds one or more variables

$$
\begin{array}{lcl}
\mathit{length} & : & \mathbb{S}_\alpha \to \mathbb{N} \\
\mathit{nth} & : & \mathbb{S}_\alpha \to \mathbb{N} \to \alpha \\
\mathit{empty} & : & \mathbb{S}_\alpha \\
\mathit{singleton} & : & \alpha \to \mathbb{S}_\alpha \\
\mathit{tabulate} & : & (\mathbb{N} \to \alpha) \to \mathbb{N} \to \mathbb{S}_\alpha \\
\mathit{map} & : & (\alpha \to \beta) \to \mathbb{S}_\alpha \to \mathbb{S}_\beta \\
\mathit{subseq} & : & \mathbb{S}_\alpha \to \mathbb{N} \to \mathbb{N} \to \mathbb{S}_\alpha \\
\mathit{append} & : & \mathbb{S}_\alpha \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha \\
\mathit{filter} & : & (\alpha \to \mathbb{B}) \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha \\
\mathit{flatten} & : & \mathbb{S}_{\mathbb{S}_\alpha} \to \mathbb{S}_\alpha \\
\mathit{update} & : & \mathbb{S}_\alpha \to (\mathbb{N} \times \alpha) \to \mathbb{S}_\alpha \\
\mathit{inject} & : & \mathbb{S}_\alpha \to \mathbb{S}_{\mathbb{N}\times\alpha} \to \mathbb{S}_\alpha \\
\mathit{isEmpty} & : & \mathbb{S}_\alpha \to \mathbb{B} \\
\mathit{isSingleton} & : & \mathbb{S}_\alpha \to \mathbb{B} \\
\mathit{collect} & : & (\alpha \times \alpha \to \mathcal{O}) \to \mathbb{S}_{\alpha\times\beta} \to \mathbb{S}_{\alpha\times\mathbb{S}_\beta} \\
\mathit{iterate} & : & (\alpha \times \beta \to \alpha) \to \alpha \to \mathbb{S}_\beta \to \alpha \\
\mathit{reduce} & : & (\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to \alpha \\
\mathit{scan} & : & (\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to (\mathbb{S}_\alpha \times \alpha)
\end{array}
$$

$$
\begin{array}{ll}
|e_s| & \equiv \mathit{length}\ e_s \\
e_s[i] & \equiv \mathit{nth}\ e_s \\
\langle\ \rangle & \equiv \mathit{empty} \\
\langle\ e\ \rangle & \equiv \mathit{singleton}\ e \\
\langle\ e : 0 \le i < e_n\ \rangle & \equiv \mathit{tabulate}\ (\texttt{lambda}\ i\ .\ e)\ e_n \\
\langle\ e : p \in e_s\ \rangle & \equiv \mathit{map}\ (\texttt{lambda}\ p\ .\ e)\ e_s \\
\langle\ p \in e_s \mid e\ \rangle & \equiv \mathit{filter}\ (\texttt{lambda}\ p\ .\ e)\ e_s \\
e_s[e_n, \cdots, e_{n'}] & \equiv \mathit{subseq}\ (e_s, e_n, e'_n - e_n + 1) \\
e_s\ +\!\!+\ e'_s & \equiv \mathit{append}\ e_s\ e'_s
\end{array}
$$

Basic Functions

- **Length and indexing**
  - The function ***length*** returns the length of a given sequence and the function ***nth*** returns the element of a sequence at a specified index

$$
\begin{aligned}
length(A) &: \mathbb{S}_\alpha \to \mathbb{N} \\
&= |A|
\end{aligned}
$$

$$
\begin{aligned}
nth\, A\, i &: \mathbb{S}_\alpha \to \mathbb{N} \to (\alpha \cup \{\bot\}) \\
&= \begin{cases} v & (i, v) \in A \\ \bot & otherwise \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
|e_s| &\equiv length\,(e_s) \\
e_s[i] &\equiv nth\,(e_s)
\end{aligned}
$$

Basic Functions

- **Empty and singleton**
  - To construct a sequence, we can use the function **empty** which returns and empty sequence,
  - the function **singleton** which takes an element and returns a sequence containing that element

$$
\begin{array}{lcl}
empty & : & \mathbb{S}_\alpha \\
 & = & \{\} \\
 & & \\
singleton(v) & : & \alpha \rightarrow \mathbb{S}_\alpha \\
 & = & \{(0, v)\}
\end{array}
$$

$$
\begin{array}{lcl}
\langle\,\rangle & \equiv & empty \\
 & & \\
\langle e \rangle & \equiv & singleton(e)
\end{array}
$$

Tabulate

- Given a function *f* and a natural number *n*, *tabulate f n* generates a sequence of length *n* consisting of *< f(0), f(1), …… , f(n-1)>*

$$tabulate\ (f : \mathbb{N} \to \alpha)\ (n : \mathbb{N}) : \mathbb{S}_\alpha$$
$$= \langle\ f(0), f(1), \ldots, f(n-1)\ \rangle\ .$$

  - ➢ syntax for *tabulate* function

$$\langle\ e : 0 \le i < e_n\ \rangle \equiv tabulate\ (\texttt{lambda}\ i\ .\ e)\ e_n,$$

    - ✓ where e and e_n are expressions, the second evaluating to an integer, and i is a variable

- can also start at any other index, as in:

$$\langle\ e : e_j \le i < e_n\ \rangle$$

Tabulate Example: Fibonacci Number

- Given the function fib i, which returns the ith Fibonacci number, the expression

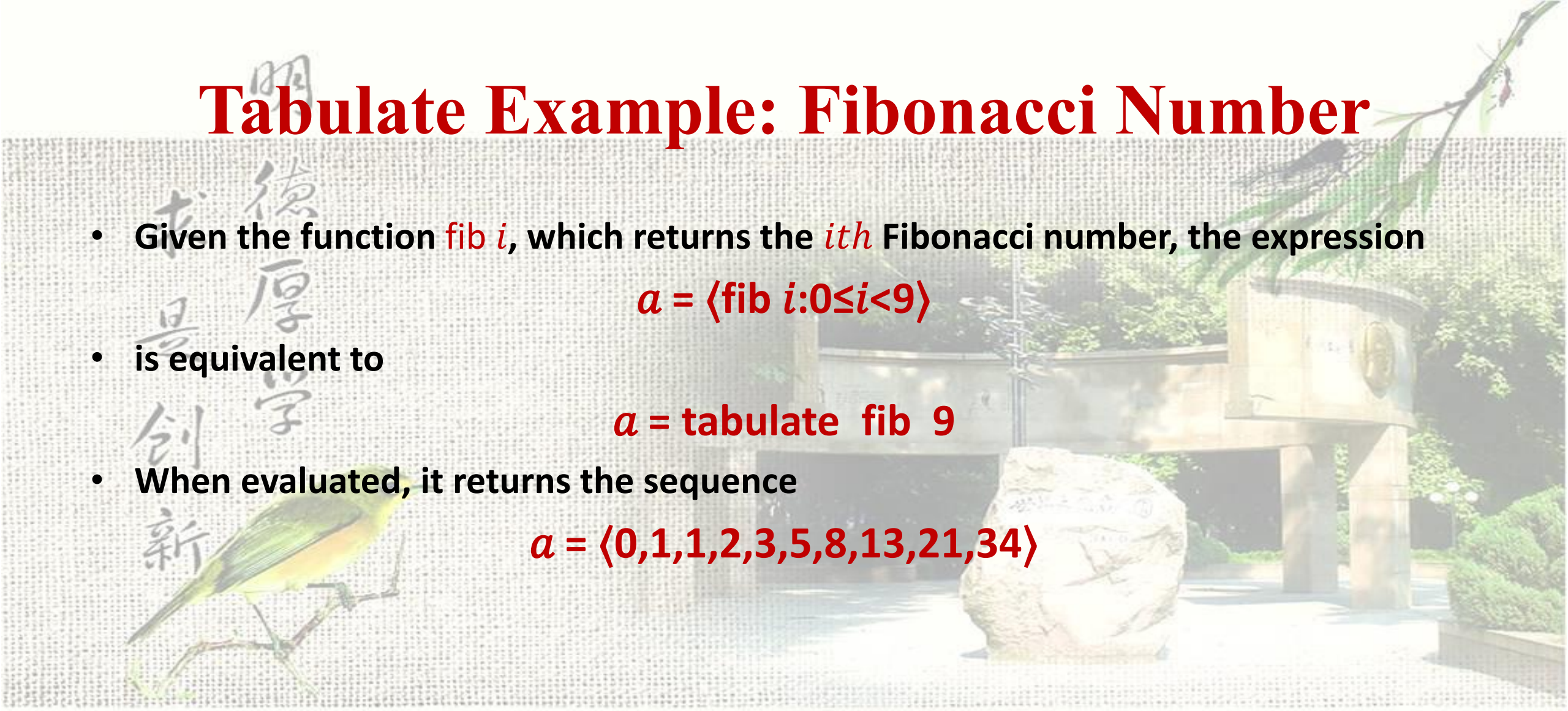
$$a = \langle \text{fib } i : 0 \le i < 9 \rangle$$

- is equivalent to

$$a = \text{tabulate fib 9}$$

- When evaluated, it returns the sequence

$$a = \langle 0,1,1,2,3,5,8,13,21,34 \rangle$$



Map and Filter

- **Map**
  - A ***map f A*** function that applies the function ***f*** to each element of ***A*** returning a sequence of equal length with the results

$$map\,(f : \alpha \to \beta)\,(a : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \{(i, f(x)) : (i, x) \in a\}$$

Map or tabulate?

  - ✓ or

$$map\,(f : \alpha \to \beta)\ \langle a_1, \ldots, a_{n-1} \rangle : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \langle f(a_1), \ldots, f(a_{n-1}) \rangle .$$

  - Syntax for the ***map*** function

$$\langle e : p \in e_s \rangle \equiv map\,(\texttt{lambda}\ p\ .\ e)\ e_s,$$

    - ✓ where e and e_s are expressions, the second evaluating to a sequence, and p is a a pattern of variables (e.g., x or (x,y))

Map and Filter

- **Map**
  - **What is the result sequence:**

$$A = \langle 9, -1, 4, 11, 13, 2 \rangle$$

$$\langle a^2 : a \in A \rangle$$

$$map\ (\textit{fn}\ a \Rightarrow a^2)\ A$$

  - **Can we implement map by using tabulate?**

$$\texttt{map}\ \ f\ \ A\ \ =\ \ \langle\, f(A[i]) : 0 \leq i < |A| \,\rangle$$

$$\texttt{map}\ \ f\ \ A\ \ =\ \ \texttt{tabulate}\ \ (\textbf{fn}\ \ i\ \ \Rightarrow\ \ f(\texttt{nth}(A, i)))\ \ (\texttt{length}\ \ A)$$

Map and Filter

- **Filter: *filter f A* function**
  - **that applies a Boolean function *f* to each element of *A*, and returns the sequence consisting exactly of those elements of $s \in A$ for which *f(s)* returns true**

$$A = \langle 9, 7, 13, 4, 11, 21 \rangle$$

$$\mathit{filter}\ \mathit{isPrime}\ A$$

$$\langle x \in A \mid \mathit{isPrime}(x) \rangle$$

$$\langle 7, 13, 11 \rangle$$

$$\mathit{filter}\,(f : \alpha \to \mathbb{B})\,(a : \mathbb{S}_\alpha) : \mathbb{S}_\alpha =$$
$$\{(|\{(j, y) \in a \mid j < i \wedge f(y)\}|, x) : (i, x) \in a \mid f(x)\}\,.$$

Map and Filter

- **Filter: *filter f A* function**
  - **syntax for the** filter **function**

$$\langle x \in e_s \mid e \rangle \equiv \textbf{filter (}\texttt{lambda}\ x.e\textbf{)}\ e_s$$

  - ✓ **where e and e_s are expressions**
  - **We can** map **and** filter **at the same time:**

$$\langle e : x \in e_s \mid e_f \rangle \equiv \textbf{map (}\texttt{lambda}\ x.e\textbf{) (filter (}\texttt{lambda}\ x.e_f\textbf{)}\ e_s\textbf{)}$$

Map and Filter

- $\langle x^2 : x \in a \rangle$ ≡ map (lambda $x.x^2$) a
  - a=⟨0,1,1,2,5,8,13,21,34⟩
  - **yields** ⟨0,1,1,4,25,64,169,441,1156⟩
- $\langle x : x \in a$ | isPrime $x \rangle$ ≡ filter isPrime a
  - **yields** ⟨2,5,13⟩

Subsequences

- The subseq(a,i,j) function extracts a contiguous subsequence of a starting at location i and with length j
  - If the subsequence is out of bounds of a, only the part within a is returned
- We can specify subseq as follows

$$\text{subseq}\ (a{:}\mathbb{S}_\alpha)\ (i{:}\mathbb{N})\ (j{:}\mathbb{N}){:}\mathbb{S}_\alpha = \{(k-i,x):(k,x)\in a \mid i\le k<i+j\}$$

- syntax for denoting subsequences

$$a[e_i\cdots e_j] \equiv \text{subseq}\ (a,e_i,e_j-e_i+1)$$

Append and Flatten

- **Append**
  - The function append (a,b) appends the sequence b after the sequence a.

$$\textbf{append } (a{:}\mathbb{S}_\alpha)\ (b{:}\mathbb{S}_\alpha){:}\mathbb{S}_\alpha = a \cup \{(i+|a|,x){:}(i,x)\in b\}$$

  - short form: a ++ b

$$\langle 1,2,3\rangle +\!+ \langle 4,5\rangle = \langle 1,2,3,4,5\rangle$$

- **Definition(Flatten)**
  - For the input is a sequence $a=\langle a_1,a_2,\ldots,a_n\rangle$, flatten returns a sequence whole elements consists of those of all the a_i in order

$$flatten\,(a:\mathbb{S}_{\mathbb{S}_\alpha}):\mathbb{S}_\alpha = \left\{\left(i+\sum_{(k,c)\in a,k<j}|c|,x\right) : (i,x)\in b,(j,b)\in a\right\}.$$

$$flatten(\langle\langle 1,2,3\rangle,\langle 4\rangle,\langle 5,6\rangle\rangle) = \langle 1,2,3,4,5,6\rangle$$

Update and Inject

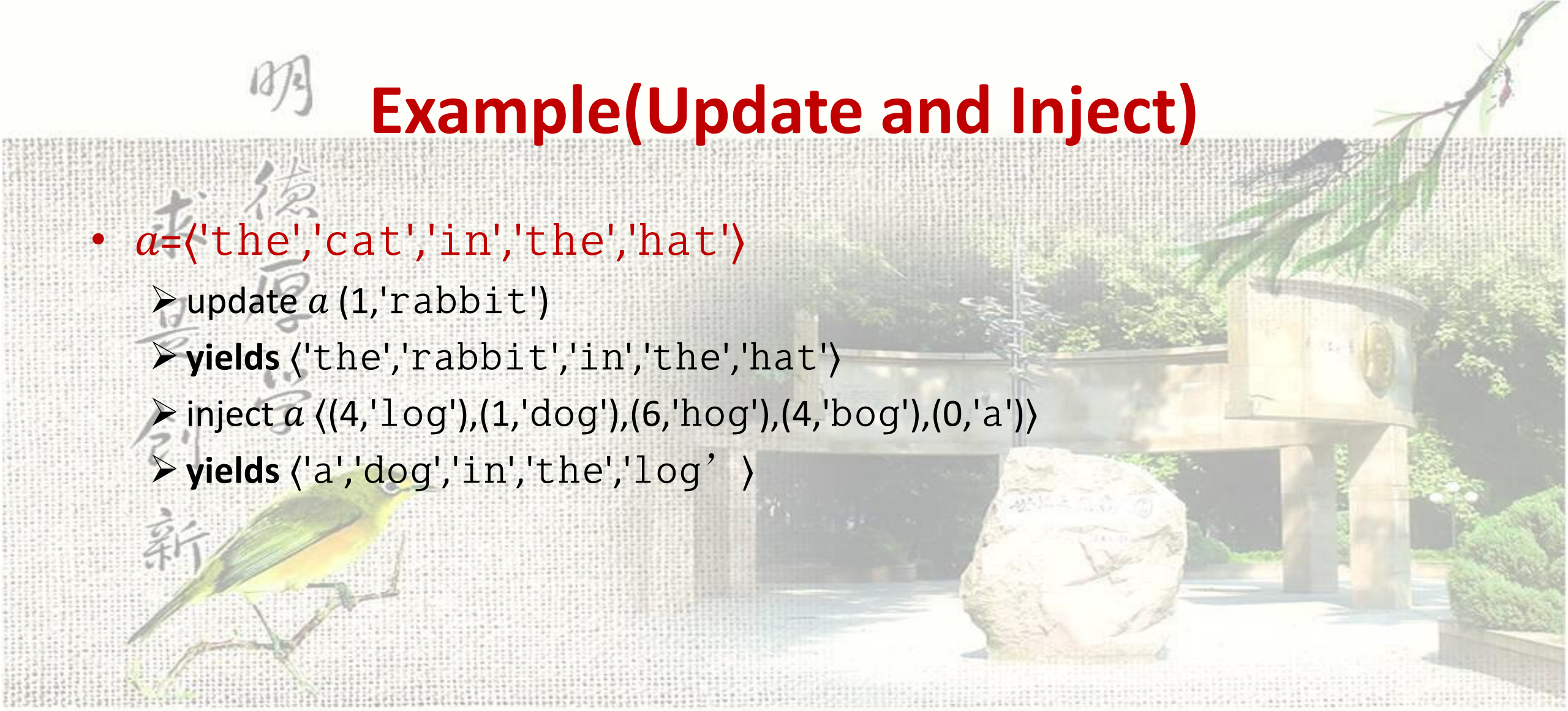
- **Definition (Update)**
  - **The function update $(a,(i,x))$, updates location i of sequence a to contain the value x**
    - ✓ **If the location is out of range for the sequence, the function returns the input sequence unchanged**

$$update\ (a : \mathbb{S}_\alpha)\ (i : \mathbb{N}, x : \alpha) : \mathbb{S}_\alpha$$
$$= \begin{cases} \{(j, y) : (j, y) \in a \mid j \neq i\} \cup \{(i, x)\} & \text{if } 0 \leq i < |a| \\ a & \text{otherwise.} \end{cases}$$

- **Definition(Inject)**
  - To update multiple positions at once, we can use inject
  - The function **inject** $a\ b$ takes a sequence b of position-value pairs and updates each position with its associated value
    - ✓ If a position is out of range, then the corresponding update is ignored.
    - ✓ If multiple positions are the same, the first update in the ordering of b take effect

Example(Update and Inject)

- a=⟨'the','cat','in','the','hat'⟩
  - update a (1,'rabbit')
  - **yields** ⟨'the','rabbit','in','the','hat'⟩
  - inject a ⟨(4,'log'),(1,'dog'),(6,'hog'),(4,'bog'),(0,'a')⟩
  - **yields** ⟨'a','dog','in','the','log' ⟩



Update and Inject

- **Definition (Nondeterministic Inject)**
  - To update multiple positions at once, we can also use nondetermistic inject ninject.
  - The function ninject $a\ b$ takes a sequence b of position-value pairs and updates each position with its associated value
    - ✓ If a position is out of range, then the corresponding update is ignored
    - ✓ If multiple positions are the same, any one of the updates may take effect

Collect

- **Collect together all the values for a given key**
- **Key-value pairs**
- **Group items that share a common key**

```
val Data = <("jack sprat", "15-210"),
            ("jack sprat", "15-213"),
            ("mary contrary", "15-210"),
            ("mary contrary", "15-251"),
            ("mary contrary", "15-213"),
            ("peter piper", "15-150"),
            ("peter piper", "15-251"), . . .>
                          ↓
val rosters = <("jack sprat", "15-210", "15-213", . . . )
               ( "mary contrary", "15-210", "15-251","15-213", . . . )
               ( "peter piper", "15-150", "15-251", . . . )
               ……>
```

Collect

- **Given a sequence of *key-value* pairs, the operation** collect **"collects" together all the values for a given key**

$$collect : (cmp : \alpha \times \alpha \to \mathcal{O}) \to (a : \mathbb{S}_{\alpha \times \beta}) \to \mathbb{S}_{\alpha \times \mathbb{S}_\beta}.$$

Here the "order set" $\mathcal{O} = \{less, equal, greater\}$.

- cmp **is a function for comparing keys of type** α
- a: $\mathbf{S_{\alpha \times \beta}}$ **is a sequence of key-value pairs**
- $\mathbf{S_{\alpha \times s\beta}}$ **seq is the resulting sequence**
- collect **function collects all values in** a **that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence**

Aggregation by Iteration

- **Definition (The *iterate* and *iteratePrefixes*)**
  - The function *iterate* iterates over a sequence while accumulating a "running sum", i.e., a result that changes at each step

$$\text{iterate } (f{:}\alpha \times \beta \to \alpha)\ (x{:}\alpha)\ (a{:}\mathbb{S}_\beta){:}\alpha$$

    - ✓ where f is a function mapping a state and an element of a to a new state, x is the initial state, a is a sequence
    - ✓ *Iterate - 0 <2,5,1,6>*
  - the semantics of *iterate* is

$$iterate\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ iterate\ f\ (f(x, a[0]))\ (a[1 \cdots |a| - 1]) & \text{otherwise.} \end{cases}$$

Aggregation by Iteration

- the function *iteratePrefixes* takes the same arguments as iterate but returns a pair, where the first component is a sequence consisting of all the intermediate result computed by iteration, up to and excluding the last element, and the second component is the final results

```
iteratePrefixes f x a =
    let g (b, x) y = (b ++ x, f(x, y))
    in iterate g (⟨ ⟩, x) a end
```

- *iteratePrefixes* returns all the intermediate values computed as a sequence, i.e., $\langle v_0, v_1, \ldots, v_{n-1} \rangle$

Aggregation by Iteration

- **Parentheses matching**
  - finding whether a string (sequence) of left and right parentheses is properly matched or nested
  - It can be described by

$$p = \langle\rangle \mid pp \mid '(p')$$

''(()) ()''

''()) (()''

  - The **parentheses matching problem** asks determining whether a given a string of parentheses is matched

Aggregation by Iteration

- **Can you think of a way of using iterate to solve this problem?**

```
1  matchParens A =
2  let
3     count (s,c) =
4       case (s,c)
5       | (None,_) ⇒ None
6       | (Some(n),)) ⇒ if (n = 0) then None else Some(n − 1)
7       | (Some(n),() ⇒ Some(n + 1)
8  in
9     (iterate count Some(0) A) = Some(0)
10 end
```

Aggregation by Reduction

- The term ***reduction*** refers to a computation that repeatedly applies an associative binary operation to a collection of elements until the result is reduced to a single value

- **Definition (Assocative Function)**
  - A function $f:\alpha\times\alpha\rightarrow\alpha$ is associative if $f(f(x,y),z)=f(x,f(y,z))$ for all x,y and z of type α

Aggregation by Reduction

- **The operation function has the type signature**

$$reduce\ (f{:}\alpha\times\alpha\rightarrow\alpha)\ (id{:}\alpha)\ (a{:}\mathbb{S}_\alpha){:}\alpha$$

  - f **is an associative function,** a **is the sequence, and** id **is the *left identity* of** f**, i.e.,** $f(id,x)=x$ **for all** $x\in\alpha$
  - **When applied to an input sequence with a function** f**,** reduce **returns the "sum" with respect to** f **of the input sequence**

$$reduce\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\left(reduce\ f\ id\ (a[0 \cdots \lfloor \frac{|a|}{2} \rfloor - 1]),\right. & \\ \quad \left. reduce\ f\ id\ (a[\lfloor \frac{|a|}{2} \rfloor \cdots |a| - 1]\right) & \text{otherwise.} \end{cases}$$

Aggregation by Reduction

- ***iterate*** **cannot always be defined in terms of** ***reduce***
  - ***iterate*** **can use the results of intermediate states computed on the prefixes of the sequence, whereas** ***reduce*** **cannot because such intermediate states are not available**
  - **How could we get parallel computing?**

Aggregation with Scan

- **Definition (The functions scan and iScan)**

  $$scan\ (f{:}\alpha{*}\alpha \rightarrow \alpha)\ (id{:}\alpha)\ (a{:}\mathbb{S}_d) : (\mathbb{S}_\alpha{*}\alpha)$$

  - where f is an associative function, a is the sequence, and id is the left identity element of f
  - The expression scan f a evaluates to the cumulative "sum" with respect to f of all prefixes of the sequence a
  - the scan function is referred to as ***prefix sums***
  - The semantics of ***scan***

$$scan\ f\ id\ a = (\langle\ reduce\ f\ id\ a[0 \cdots (i-1)] : 0 \le i < |a|\ \rangle,\ reduce\ f\ id\ a)$$

$$scanI\ f\ id\ a = \langle\ reduce\ f\ id\ a[0 \cdots i] : 0 \le i < |a|\ \rangle$$

Aggregation with Scan

- **Example: copy scan**
  - **Scan can also be used to pass information along a sequence**

< NONE, SOME(7), NONE, NONE, SOME(3), NONE >

↓

< NONE, NONE, SOME(7), SOME(7), SOME(7), SOME(3) >

  - **How would you do this with scan?**

Aggregation with Scan

- **Example: copy scan**

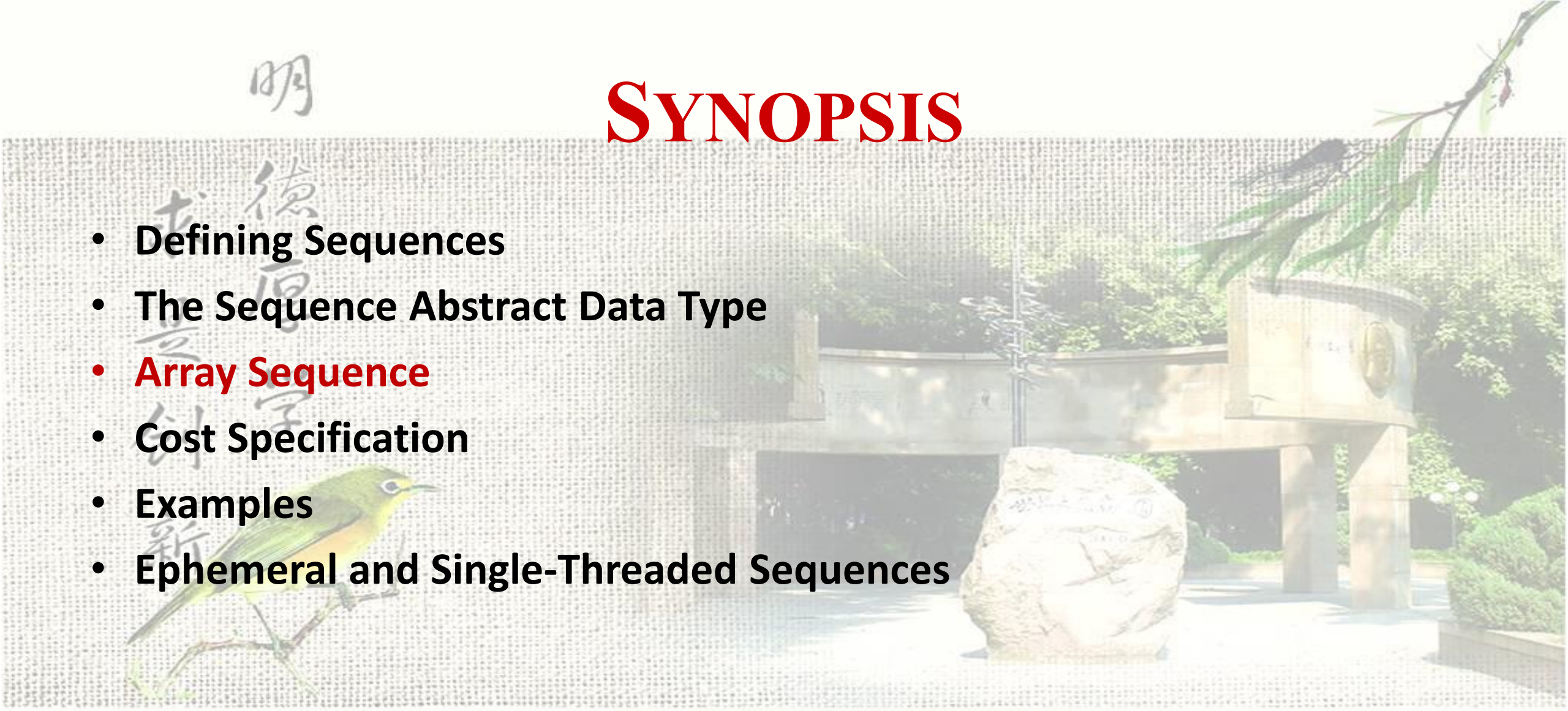
$$f : \alpha\ \text{option} \times \alpha\ \text{option} \rightarrow \alpha\ \text{option}$$

```
1    fun copy(a, b) =
2      case b of
3          SOME( _) ⇒ b
4        | NONE ⇒ a
```

- **Passes its right argument if it is SOME, else passes its left argument.**
- **How do you show copy is associative?**
- **Write down the scan expression!**

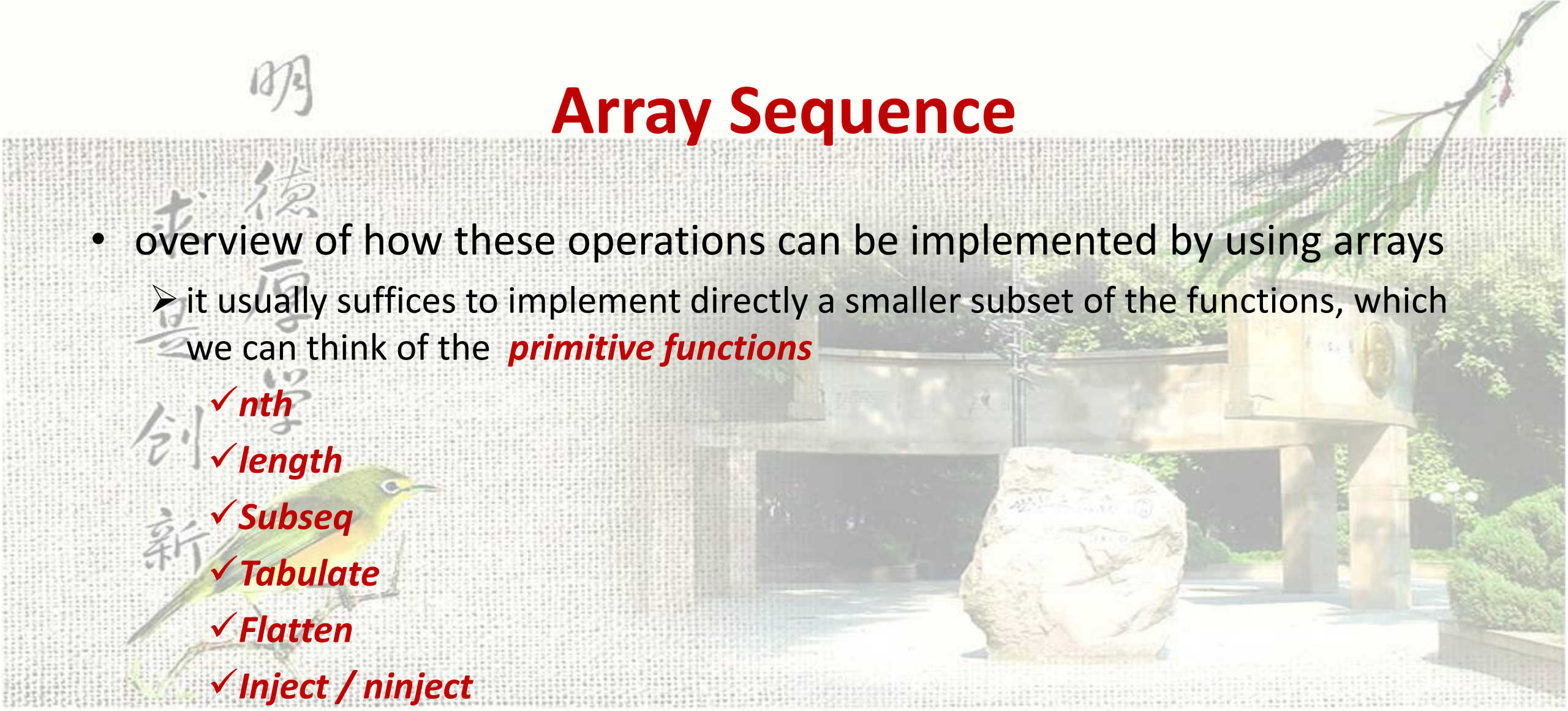
SYNOPSIS

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



Array Sequence

- overview of how these operations can be implemented by using arrays
  - it usually suffices to implement directly a smaller subset of the functions, which we can think of the ***primitive functions***
    - ✓ ***nth***
    - ✓ ***length***
    - ✓ ***Subseq***
    - ✓ ***Tabulate***
    - ✓ ***Flatten***
    - ✓ ***Inject / ninject***



A Parametric Implementation

- **Algorithm (Function** empty**)**

  empty = tabulate (`lambda` *i.i)* **0**

- **Algorithm (Function** singleton**)**

  singleton x = tabulate (`lambda` *i.x)* **1**

- **Algorithm (Function** map**)**

  map f a = tabulate (`lambda` *i.f (a [i])) |a|*

A Parametric Implementation

- **Algorithm (Function** append**)**
- ***append a b = flatten ⟨a,b⟩***
- ***append a b = tabulate (select (a,b)) (|a|+|b|)***

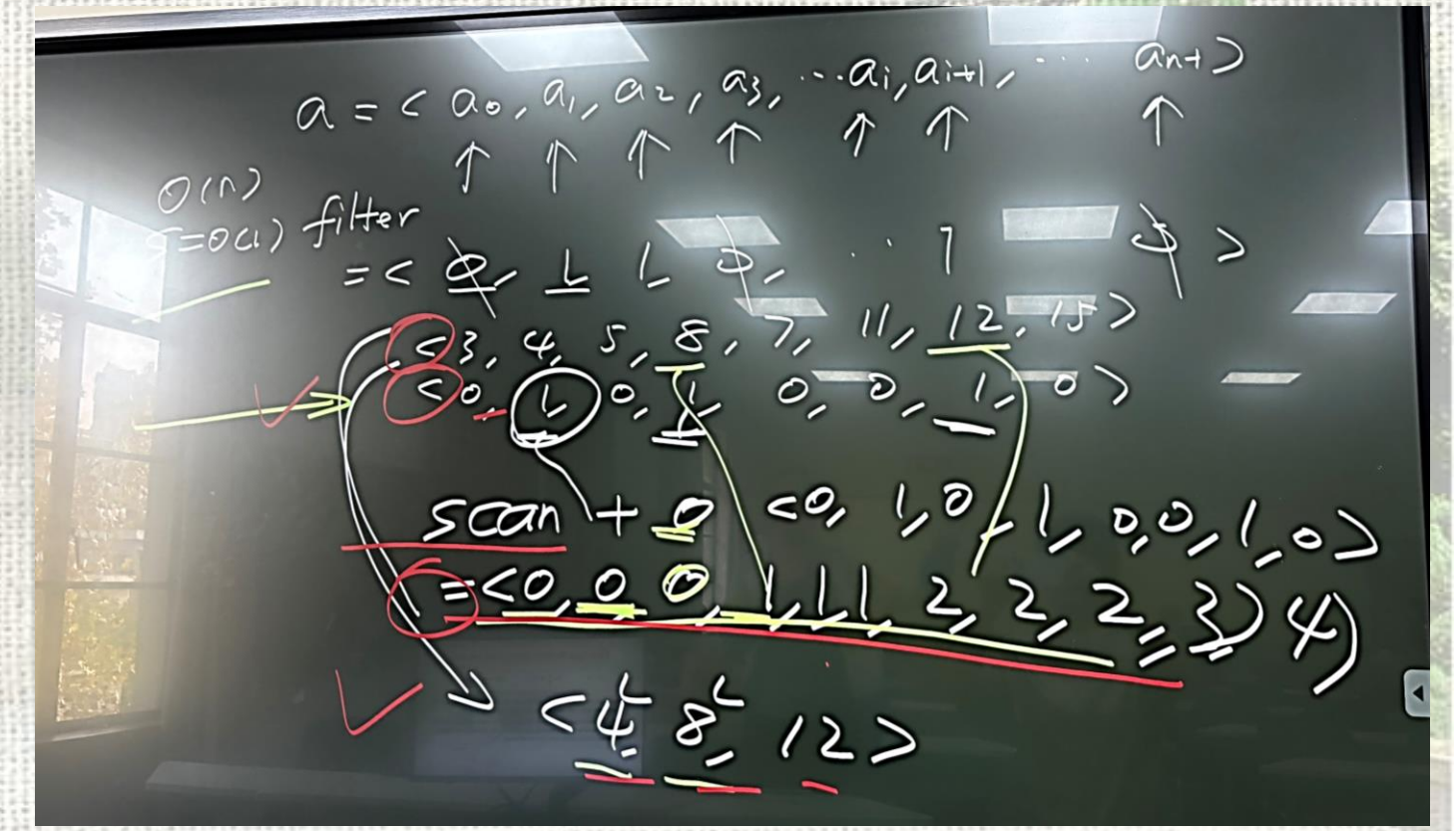
$$
\begin{aligned}
&\mathit{select}\ (a, b)\ i = \texttt{lambda}\ i.\\
&\qquad\qquad \texttt{if}\ i < |a|\ \texttt{then}\ a[i]\\
&\qquad\qquad \texttt{else}\ b[i - |a|].
\end{aligned}
$$

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

A Parametric Implementation

- **Algorithm (Function** filter**)**
- **filter** $f\ a$ **=**
  let b **= map (deflate** f**)** a
  in flatten b **end**

```
deflate f x =
    if (f x) then ⟨ x ⟩
    else ⟨ ⟩.
```



Can use tabulate?

A Parametric Implementation

- **Algorithm (Function** update**)**

  update a (i,x) **=**

  tabulate (`lambda` j. `if` i=j `then` x `else` $a[i]$) $|a|$

- **Algorithm (Functions** isEmpty **and** isSingleton**)**

  `isEmpty` a = $|a|$=0

  `isSingleton` a = $|a|$=1

A Parametric Implementation

- **Algorithm (Functions** iterate**)**

iterate $f\ x\ a$=

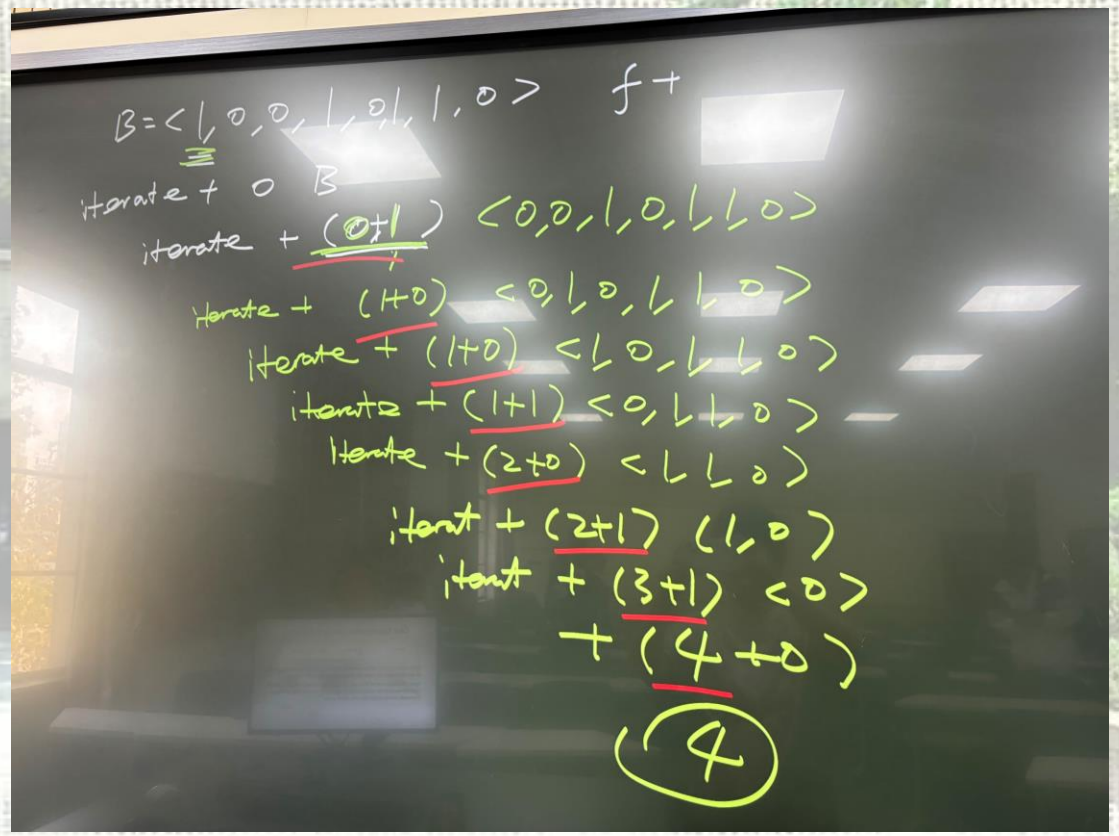
`if` $|a|$=**0** `then`

x

`else if` $|a|$=**1** `then`

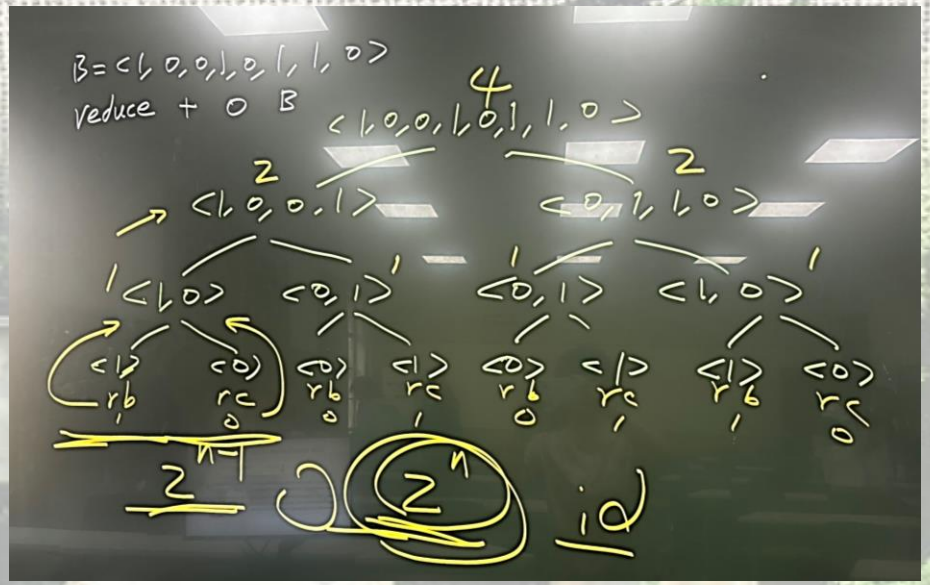
$f(x,a[0])$ `else`

iterate $f\ (f(x,a[0]))\ a[1...|a|-1]$



A Parametric Implementation

- **Algorithm (Functions** reduce**)**



reduce f **id** a=

if $|a|$=0 **then** **id**

else if $|a|$=1 **then** a[0]

else let

mid = $floor(|a|/2)$

(b, c) = (a[0...mid−1],a[mid...$|a|$−1])

(rb, rc) = (**reduce** f **id** b) || (**reduce** f **id** c)

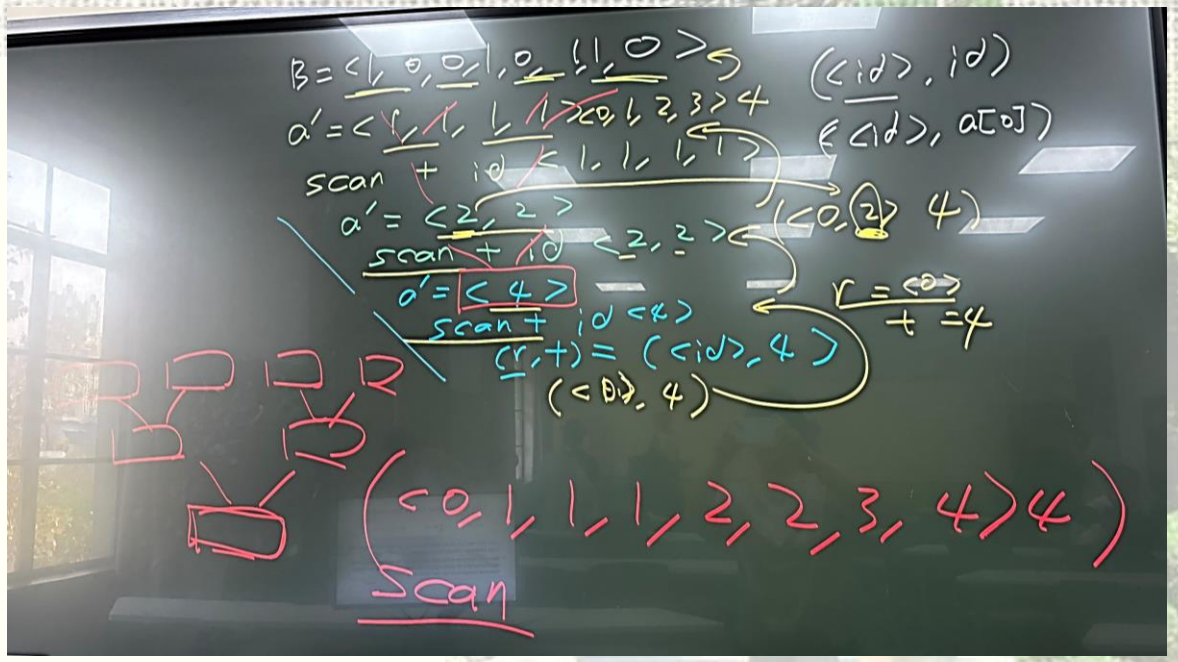
in $f(rb,rc)$

end

A Parametric Implementation

- **Algorithm (Scan Using Contraction)**

```
(* Assumption: |a| is a power of two. *)
scan f id a =
  case |a|
  | 0 ⇒ (⟨ ⟩, id)
  | 1 ⇒ (⟨ id ⟩, a[0])
  | n ⇒
    let
      a' = ⟨ f(a[2i], a[2i + 1]) : 0 ≤ i < n/2 ⟩
      (r, t) = scan f id a'
    in
      (⟨ p_i : 0 ≤ i < n ⟩, t), where p_i = { r[i/2]              even(i)
                                            { f(r[i/2], a[i − 1])  otherwise
    end
```



Implementing the Primitive Functions

- **a sequence is represented as**

$$< a_{i+0}, \ldots, a_{i+n-1} > = < a_{i+0}, a_{i+1}, \ldots, a_{i+n-2}, a_{i+n-1} >$$

$$< a[i], \ldots, a[j] > = < a[i], a[i+1], \ldots, a[j-i], a[j-i+1] >$$

- We made careful use of **side effects** (memory updates) in implementing the primitive functions
  - The side effects are not visible to the programmer, the ADT remains to be purely functional and thus is safe for parallelism
  - Effects such as these that have no impact on purity are sometimes called ***benign effects***

Implementing the Primitive Functions

- **Algorithm (Function** nth**)**
  - ➢ **constant work and span**
- **Algorithm (Function** length**)**
  - ➢ **calculate length by using simple arithmetic in constant work and span**
- **Algorithm (Function** subseq**)**
  - ➢ **requires basic arithmetic and thus can be done in constant work and span**
    - ✓ **determining the boundaries for the new slice**
    - ✓ **do not need to copy the elements of the array within the boundary**

How?

Implementing the Primitive Functions

- **Algorithm (Function** tabulate**)**

tabulate f n

$$1 + \sum_{i=0}^{n} W(f(i)) \qquad\qquad 1 + \max_{i=0}^{n} S(f(i))$$

1. allocate a fresh array of n elements
2. evaluate f at each position i and write the result into position i of the array
3. the function f can be evaluated at each element independently in parallel

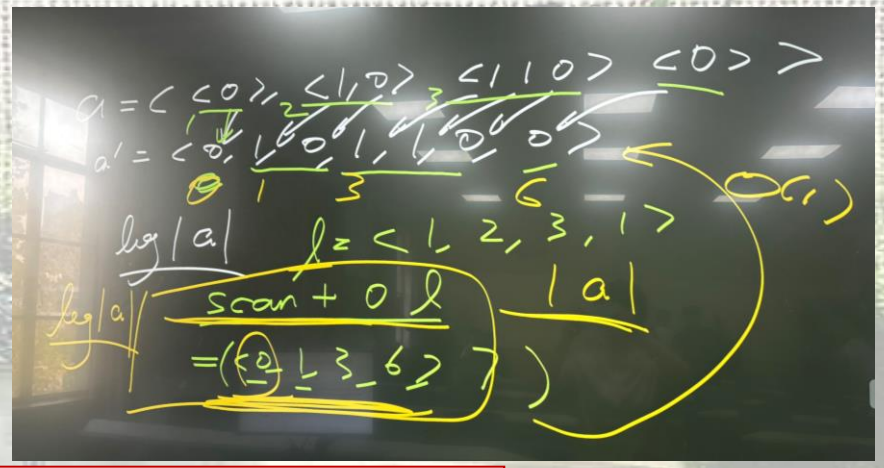
Implementing the Primitive Functions

- **Algorithm (Function** flatten**)**

  flatten a

  ➢ **where a is a sequences of sequences**

  total work is $O(|a|+||a||)$, span is $O(\lg|a|)$



$$||a|| = \sum_{i=0}^{|a|-1} |a[i]|$$

1. **map each element of a to its length; let ℓ be the resulting sequence**
2. **perform the scan** scan + 0 ℓ**, returns for each element of a its position in the result sequence of** flatten
3. **allocate an array that can hold all of the elements of the sequences in a and write each element of a into its corresponding segment in parallel**

Implementing the Primitive Functions

- **Algorithm (Function** inject**)**

inject a b

➢ **where a is a sequence of length n and b is a sequence of m updates**

total work is $O(|a|+|b|)$, span is lg(degree(b))

1. **create a new array aa from a, where for all $0 \le i < |a|$, $aa[i]=(a[i],|a|)$**
2. **then "inject" all updates in b into aa in parallel**
3. **To handle each update of the form (j,v) at position k, we perform an atomic update operation**
4. **After all updates complete, we take aa and create another array consisting of only the value component of each element**

Implementing the Primitive Functions

- **an atomic update operation: update of the form (j,v) at position k**

```
atomicWrite aa b k =
  atomically do:
    (j, v) ← b[k]
    (w, i) ← aa[j]
    if k < i then
      aa[j] ← (v, k)
```

a=⟨'the','cat','in','the','hat'⟩
inject a ⟨(4,'log'),(1,'dog'),(6,'hog'),(4,'bog'),(0,'a')⟩
yields ⟨'a','dog','in','the','log' ⟩

- **This update operation guarantees that of the possibly many conflicting operations the first (leftmost) one in the update sequence wins and is transferred to the result; the other updates either don't take place or are overwritten.**

Implementing the Primitive Functions

- **Algorithm(Function** ninject**)**

ninject a b

➢ **where a is a sequence to be injected into and b is the updates.**

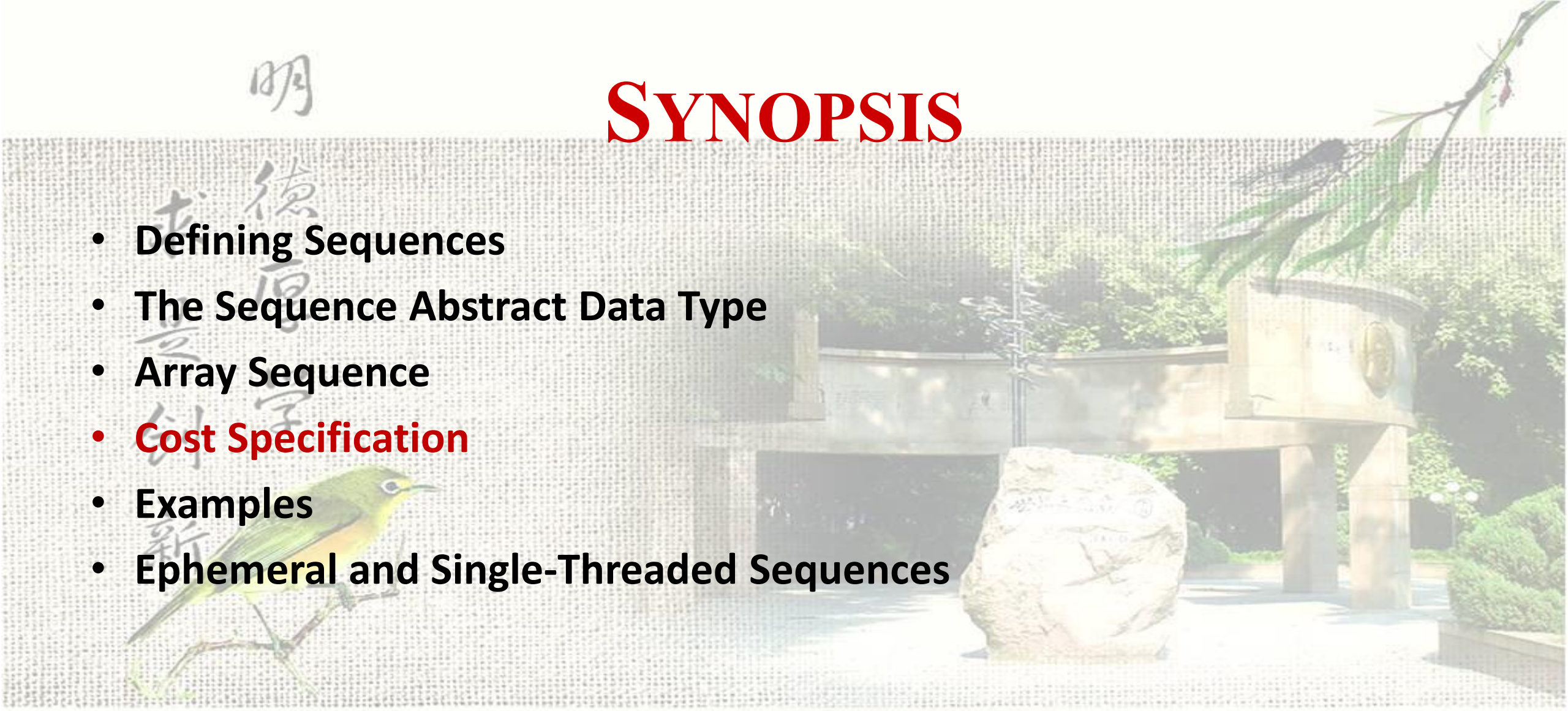
total work is $O(|a|+|b|)$, span is $O(1)$

1. **make a copy of the array a and then use an atomic write operation to write or "inject" each update independently in parallel**
2. **Each instance of the atomic write operation updates the relevant element of the copied array atomically**

➢ **Assuming that each atomic write operation requires constant-work, this implementation requires linear work in the number of updates and constant span.**

SYNOPSIS

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



COST SPECS

- consider sequences based on arrays, trees, and lists
  - **Definition (Domination of cost specifications)**
    - ✓one cost specification ***dominates*** another if for each and every function, its asymptotic costs are no higher than those of the latter
  - **Choosing cost specifications**
    - ✓When deciding which of the possibly many cost specification to use for a particular ADT, we usually notice that there are certain trade-offs: some functions will be cheaper in one and while others are cheaper in another

Cost Specification (Array Sequences)

Operation	Work	Span
length a	1	1
nth a i	1	1
singleton x	1	1
empty	1	1
isSingleton x	1	1
isEmpty x	1	1

- How?

Cost Specification (Array Sequences)

tabulate f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \max_{i=0}^{n} S(f(i))$
map f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \max_{x \in a} S(f(x))$

- **total work: the sum of the work of applying f at each position, as well as an additional unit cost to account for** tabulate **or** map **itself**
- **Span: the maximum of the span of applying f at each position, plus 1 for the function call itself**
  - **it is possible to apply the function f in parallel—there are no dependencies among the different positions**

Cost Specification (Array Sequences)

filter f a	$1 + \sum_{x \in a} W(f(x))$	$\lg\|a\| + \max_{x \in a} S(f(x))$

- **Total work**: the sum of the work of applying f at each position, as well as an additional unit cost, for the function call itself
- **Span**: the maximum of the span of applying f at each position, plus a logarithmic term for performing ***compaction***, i.e., packing the chosen elements contiguously into the result array

How?

Cost Specification (Array Sequences)

subseq a (i, j)	1	1
append a b	$1 + \|a\| + \|b\|$	1
flatten a	$1 + \|a\| + \sum_{x \in a} \|x\|$	$1 + \lg\|a\|$

- The operation ***append*** requires work proportional to the length of the sequences given as input, can be implemented in constant span
- The operation **flatten** generalizes append, requiring work proportional to the total length of the sequences flattened, and can be implemented in parallel in logarithmic span in the number of sequences flattened

How?

Cost Specification (Array Sequences)

update a (i, x)	$1 + \|a\|$	1
inject a b	$1 + \|a\| + \|b\|$	$\lg(\text{degree}(b))$
ninject a b	$1 + \|a\| + \|b\|$	1

- **Work: proportional to the length of the sequences they are given as input**
  - **the interface is purely functional so that the input sequence needs to be copied–we are not allowed to update the old copy**
- **Span**
  - update **and** ninject **can be implemented in constant span**
  - inject **requires** $O(\lg(\text{degree}(b)))$ **span where the degree of the update sequence** b **is the maximum number of updates targeting the same position in the sequence being updated**

how?

Cost Specification (Array Sequences)

collect f a	$1 + W(f) \cdot \lvert a \rvert \lg \lvert a \rvert$	$1 + S(f) \cdot \lg^2 \lvert a \rvert$

how?

- **The primary cost in implementing** collect **is a sorting step that sorts the sequence based on the keys**
- **The work and span of collect is therefore determined by the work and span of (comparison) sorting with the specified comparison function** f

Cost Specification (Array Sequences)

- **Specification of** iterate
  - **we consider the intermediate values generated by an evaluation of** iterate

$$iterate\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ iterate\ f\ (f(x, a[0]))\ (a[1 \cdots |a| - 1]) & \text{otherwise.} \end{cases}$$

  - Let $\mathcal{T}(iterate\ f\ v\ a)$

denote the set of calls to $f(\cdot,\cdot)$ performed along with the arguments, as defined by the specification above

Cost Specification (Array Sequences)

$$iterate\ f\ x\ a \qquad 1 + \sum_{f(y,z)\in T(-)} W(f(y,z)) \qquad 1 + \sum_{f(y,z)\in T(-)} S(f(y,z))$$

- **Cost Specification (Cost for** iterate**)**

$$W(iterate\ f\ x\ a) = O\left(1 + \sum_{f(y,z)\in T(iterate\ f\ x\ a)} W(f(y,z))\right)$$

$$S(iterate\ f\ x\ a) = O\left(1 + \sum_{f(y,z)\in T(iterate\ f\ x\ a)} S(f(y,z))\right)$$

Cost Specification (Array Sequences)

- **Specification of** reduce

$$reduce\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\left(reduce\ f\ id\ (a[0 \cdots \lfloor \frac{|a|}{2} \rfloor - 1]),\right. & \\ \quad \left. reduce\ f\ id\ (a[\lfloor \frac{|a|}{2} \rfloor \cdots |a| - 1]\right) & \text{otherwise.} \end{cases}$$

Cost Specification (Array Sequences)

- **Cost Specification (Cost of** reduce**)**

$$W(\textit{reduce } f \; x \; a) = O\left(1 + \sum_{f(y,z) \in T(\textit{reduce } f \; x \; a)} W(f(y,z))\right), \text{ and}$$

$$S(\textit{reduce } f \; x \; a) = O\left(\lg |a| \cdot \max_{f(y,z) \in T(\textit{reduce } f \; x \; a)} S(f(y,z))\right).$$

how?

- **The** $\lg|a|$ **term expresses the fact that the recursion tree in the specification of** reduce **is at most** $O(\lg|a|)$ **deep**
- **Since each node in the recursion tree has span at most** $\max_{f(y,z)} S(f(y,z))$**, any root-to-leaf path, has at most** $O(\lg|a| \cdot \max_{f(a,b)} S(f(a,b)))$ **span**

Cost Specification (Array Sequences)

$scan\ f\ x\ a$	$\|a\|$	$\lg\|a\|$

- **Cost Specification (Cost for** scan**)**
  - **Consider the expression** scan $f\ x\ a$**, where** $f(\cdot,\cdot)$ **always requires** $\mathcal{O}(1)$ **work and span**

$$W(scan\ f\ x\ a) = O(|a|), \text{ and}$$

$$S(scan\ f\ x\ a) = O(\lg|a|).$$

Cost Specification (Tree Sequences)

- The specification represents the cost for a class of implementations that use **a balanced tree to represent the sequence**

How to realize?

Operation	Work	Span
length a	1	1
singleton x	1	1
isSingleton x	1	1
isEmpty x	1	1
nth a i	$\lg\|a\|$	$\lg\|a\|$

why?

why?

Cost Specification (Tree Sequences)

+log|A|?

How to realize?

tabulate f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \lg n + \max_{i=0}^{n} S(f(i))$
map f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \lg\|a\| + \max_{x \in a} S(f(x))$
filter f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \lg\|a\| + \max_{x \in a} S(f(x))$

Cost Specification (Tree Sequences)

How to realize?

$subseq(a, i, j)$	$1 + \lg(\|a\|)$	$1 + \lg(\|a\|)$
append a b	$1 + \|\lg(\|a\|/\|b\|)\|$	$1 + \|\lg(\|a\|/\|b\|)\|$
flatten a	$1 + \|a\| \lg\left(\sum_{x \in a} \|x\|\right)$	$1 + \lg(\|a\| + \sum_{x \in a} \|x\|)$
inject a b	$1 + (\|a\| + \|b\|) \lg\|a\|$	$1 + \lg(\|a\| + \|b\|)$
ninject a b	$1 + (\|a\| + \|b\|) \lg\|a\|$	$1 + \lg(\|a\| + \|b\|)$
collect f a	$1 + W(f) \cdot \|a\| \lg\|a\|$	$1 + S(f) \cdot \lg^2\|a\|$

Cost Specification (Tree Sequences)

How to realize?

iterate f x a	$1 + \sum_{f(y,z)\in\mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z)\in\mathcal{T}(-)} S(f(y,z))$
reduce f x a	$1 + \sum_{f(y,z)\in\mathcal{T}(-)} W(f(y,z))$	$\lg\|a\| \cdot \max_{f(y,z)\in\mathcal{T}(-)} S(f(y,z))$
scan f x a	$\|a\|$	$\lg\|a\|$

COST OF ITERATE

- **Examples:**
  - **How to sort a sequence by using mergeOne?**

$$\texttt{iterSort}\ a = \texttt{iterate}\ \texttt{mergeOne}\ \langle\rangle\ a$$

  - **How to get the cost specification?**
    - ✓ **Since we iterate adding in each element after the previous, there is no parallelism between merges, but there is parallelism within a mergeOne**
    - ✓ **W=?**
    - ✓ **S=?**

how?
Array sequence or tree?

$$W(\texttt{iterSort}\ a) \leq \sum_{i=1}^{n-1} c \cdot (1+i) = O(n^2).$$

$$S(\texttt{iterSort}\ a) \leq \sum_{i=1}^{n-1} c \cdot \lg(1+i) = O(n \lg n).$$

COST OF REDUCE

- Consider evaluation of *reduce f I A* and let $\mathcal{T}$ *(reduce f v A)* denote the set of calls to *f(.,.)* performed along with the arguments

$$W(\texttt{reduce}\ f\ x\ a) = O\left(1 + \sum_{f(y,z) \in \mathcal{T}(\texttt{reduce}\ f\ x\ a)} W(f(y,z))\right)$$

$$S(\texttt{reduce}\ f\ x\ a) = O\left(\lg |a| \cdot \max_{f(y,z) \in \mathcal{T}(\texttt{reduce}\ f\ x\ a)} S(f(y,z))\right)$$

why?

COST OF SCAN

- For scan, we will stop at giving a cost specification by assuming that the *function* that we are scanning with performs *O(1)* work and span
- Consider evaluation of scan f I A, for both the array-sequence and tree-sequence specification

$$W(\texttt{scan}\ f\ x\ a) = O(|a|)$$

$$S(\texttt{scan}\ f\ x\ a)) = O(\lg |a|).$$

why?

Cost Specification (List Sequences)

- use (linked) lists to represent the sequence
  - accessing the element at position i requires traversing the list from the head to i, which leads to $\mathcal{O}(i)$ work and span
- List-based implementations therefore expose hardly any parallelism
- Their main advantage is that they require quick access to the *head* and the *tail* of the sequence, which are defined as the first element and the suffix of the sequence that starts at the second element respectively

Cost Specification (List Sequences)

Operation	Work	Span
`length` a `singleton` x `isSingleton` x `isEmpty` x	1	1
`nth` a i	i	i

How to realize?

Cost Specification (List Sequences)

Operation	Work	Span
`tabulate` $f\ n$	$1+\sum_{i=0}^{n} W(f(i))$	$1+\sum_{i=0}^{n} S(f(i))$
`map` $f\ a$	$1+\sum_{x\in a} W(f(x))$	$1+\sum_{x\in a} S(f(x))$
`filter` $f\ a$	$1+\sum_{x\in a} W(p(x))$	$1+\sum_{x\in a} S(p(x))$

How to realize?

Cost Specification (List Sequences)

- Can get better answer?

How to realize?

Operation	Work	Span
`subseq` $a\ (i, j)$	$1 + i$	$1 + i$
`append` $a\ b$	$1 + \|a\|$	$1 + \|a\|$
`flatten` a	$1 + \|a\| + \sum_{x \in a} \|x\|$	$1 + \|a\| + \sum_{x \in a} \|x\|$
`update` $a\ (i, x)$	$1 + \|a\|$	$1 + \|a\|$
`inject` $a\ b$	$1 + \|a\| + \|b\|$	$1 + \|a\| + \|b\|$
`collect` $f\ a$	$1 + W(f) \cdot \|a\| \lg \|a\|$	$1 + S(f) \cdot \|a\| \lg \|a\|$

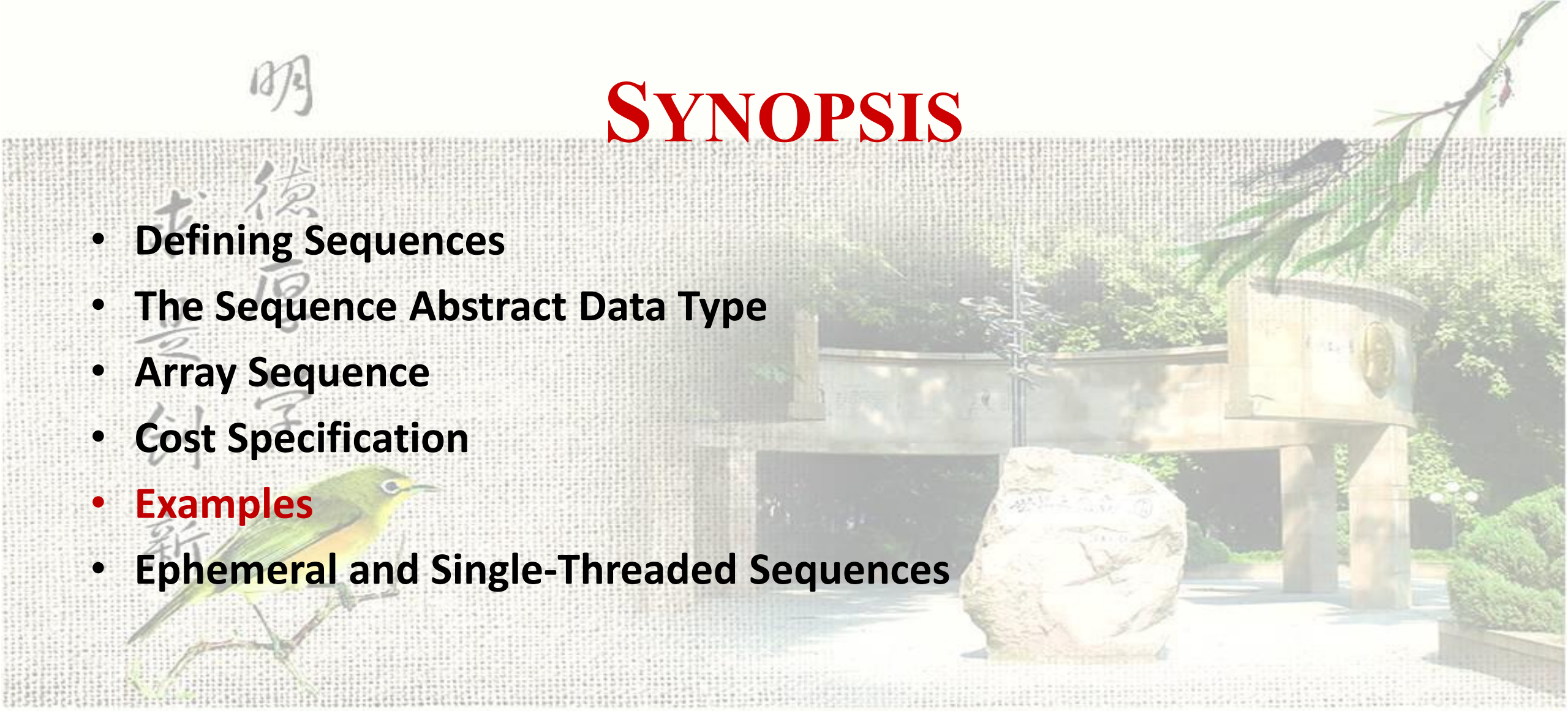
Cost Specification (List Sequences)

Operation	Work	Span
`iterate` $f\ x\ a$	$1 + \sum_{f(y,z) \in T(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in T(-)} S(f(y,z))$
`reduce` $f\ x\ a$	$1 + \sum_{f(y,z) \in T(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in T(-)} S(f(y,z))$
`scan` $f\ a$	$\lvert a \rvert$	$\lvert a \rvert$

How to realize?

SYNOPSIS

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



Miscellaneous Examples

- **Problem (Points in 2D)**
  - **We wish to create a sequence consisting of points in two dimensional space (x,y) whose coordinates are natural numbers that satisfy the conditions:**

$$0 \le x < n \text{ and } 1 \le y < n$$

  - ✓ **For example, for n=3, we would like to construct the sequence** ⟨(0,1),(0,2),(1,1),(1,2),(2,1),(2,2)⟩

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Miscellaneous Examples

- **Algorithm (2D Points)**

$$points2D\ n = \langle\, (x, y) : 1 \leq y < n, 0 \leq x < n \,\rangle\,.$$

$$points2D\ n = flatten\ \langle\, \langle\, (x, y) : 1 \leq y < n \,\rangle : 0 \leq x < n \,\rangle\,.$$

$$\begin{aligned} &points2D\ n = \\ &\quad flatten\ (tabulate\ (lambda\ x\ .\ tabulate\ (lambda\ y\ .\ (x, y + 1)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (n - 1)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad n). \end{aligned}$$

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Miscellaneous Examples

- **Problem (Points in 3D)**
  - **Building on Problem Points in 2D, suppose that we wish to generate points in 3D and restrict the points to those whose coordinates (x,y,z) are natural numbers that satisfy the conditions that**

$$0 \le x \le n-1, 1 \le y \le n, \text{ and } 2 \le z \le n+1$$

Miscellaneous Examples

- **Algorithm (Points in 3D)**

$$points3D\ n = \langle (x, y, z) : 0 \leq x \leq n-1, 1 \leq y \leq n, 2 \leq z \leq n+1 \rangle.$$

$$points3D\ n = flatten \langle\ flatten\ \langle \langle (x, y, z) : 2 \leq z \leq n+1 \rangle : 1 \leq y \leq n \rangle : 0 \leq x \leq n-1 \rangle.$$

How to use lambda?

Miscellaneous Examples

- **Problem (Cartesian Product)**
  - **Present an algorithm that returns the Cartesian product of two sequences**
  - **For example, for the sequences a=⟨1,2⟩, and b=⟨3.0,4.0,5.0⟩, the algorithm should return the Cartesian product of a and b, which is**

$$a \times b = \langle (1,3.0),(1,4.0),(1,5.0),(2,3.0),(2,4.0),(2,5.0) \rangle$$

Miscellaneous Examples

- **Algorithm(Cartesian Product)**

$$CartesianProduct\ (a, b) =$$
$$\langle\ (x, y) : x \in a, y \in b\ \rangle\ .$$

$$CartesianProduct\ (a, b) =$$
$$flatten\ (map\ (\texttt{lambda}\ x\ .\ map\ (\texttt{lambda}\ y\ .\ (x, y))\ b)\ a).$$

AN EXAMPLE: PRIMES

- The primes problem
  - given an integer *n* to find all prime numbers up to, and including n
- Using of *isPrime(n)*
  - We note that if an integer n is not prime, then it must have a divisor that is at most $\lceil \sqrt{n} \rceil$ — why?
  - So, for any n, we can get

Algorithm (Brute Force Primality Test)

$isPrime\ n =$

```
let
  all = ⟨ n mod i : 1 ≤ i ≤ ⌊√n⌋ ⟩
  divisors = ⟨ x : x ∈ all | x = 0 ⟩
in
  |divisors| = 1
end
```

AN EXAMPLE: PRIMES

- The W & S of *isPrime(n)*
  - We note that the work for evaluating *n mod j = 0* is constant for each *j*
  - *W*

$$W_{\texttt{isPrime}}(n) = O\left(1 + \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} O(1)\right) = O\left(\sqrt{n}\right).$$

  - *S*

$$S_{\texttt{isPrime}}(n) = O\left(\lg \sqrt{n} + \max_{i=1}^{\lfloor \sqrt{n} \rfloor} O(1)\right) = O(\lg n).$$

how?

AN EXAMPLE: PRIMES

- The W & S of *primes(n)*
  - Algorithm (Brute Force Solution to the Primes Problem)

$$
\begin{array}{l}
primesBF\ n = \\
\quad \texttt{let} \\
\quad\quad all = \langle\, i : 1 < i < n \,\rangle \\
\quad\quad primes = \langle\, x : x \in all \mid isPrime(x) \,\rangle \\
\quad \texttt{in} \\
\quad\quad primes \\
\quad \texttt{end}
\end{array}
$$

AN EXAMPLE: PRIMES

- The W & S of *primes(n)*
  - *W*
  - *S*

$$W_{primesBF}(n) = O\left(\sum_{i=2}^{n-1} 1 + W_{isPrime}(i)\right)$$
$$= O\left(\sum_{i=2}^{n-1} 1 + \sqrt{i}\right)$$
$$= O\left(n^{3/2}\right).$$

how?

$$S_{primesBF}(n) = = O\left(\lg n + \max_{i=2}^{n} S_{isPrime}(i)\right)$$
$$= O\left(\lg n + \max_{i=2}^{n} \lg i\right)$$
$$= O(\lg n)$$

how?any question?

AN EXAMPLE: PRIMES

- How about the parallelism of *primes(n)* ?

$$P(n) = W(n)/S(n) = n^{3/2}/\log n$$

- Is there any wasteful in the *primes(n)* ?

```
primesBF n =
  let
    all = ⟨ i : 1 < i < n ⟩
    primes = ⟨ x : x ∈ all | isPrime(x) ⟩
  in
    primes
  end
```

AN EXAMPLE: PRIMES

- primeSieve
  - How about n=12?

```
primeSieve n =
  let
    (* Composite numbers. *)
    cs = ⟨ i * j : 2 ≤ i ≤ ⌊√n⌋, 2 ≤ j ≤ n/i ⟩
    sieve = ⟨ (x, false) : x ∈ cs ⟩
    all = ⟨ true : 0 ≤ i < n ⟩
    isPrime = ninject all sieve
    primes = ⟨ i : 2 ≤ i < n | isPrime[i] = true ⟩
  in
    primes
  end
```

Why?

AN EXAMPLE: PRIMES

- The W & S of *primes⁺(n) primeSieve*

```
primeSieve n =
  let
    (*  Composite numbers. *)
    cs = ⟨ i * j : 2 ≤ i ≤ ⌊√n⌋ , 2 ≤ j ≤ n/i ⟩
    sieve = ⟨ (x, false) : x ∈ cs ⟩
    all = ⟨ true : 0 ≤ i < n ⟩
    isPrime = ninject all sieve
    primes = ⟨ i : 2 ≤ i < n | isPrime[i] = true ⟩
  in
    primes
  end
```

- Cs, sieve : Work : proportional to the number of composites m, Span : *O(log(m))* — how?
- inject: Work=proportional to the length of sieve, m, span: constant
- primes: *W=O(n)*, span: *O(log n)* — how?

AN EXAMPLE: PRIMES

- The W & S of *primeSieve*
  - calculate the number of composties

$$
\begin{aligned}
m &= \sum_{i=2}^{\lfloor \sqrt{n} \rfloor} \left\lceil \frac{n}{i} \right\rceil \\
&\le (n+1) \sum_{i=2}^{\lfloor \sqrt{n} \rfloor} \frac{1}{i} \\
&\le (n+1) H(\lfloor \sqrt{n} \rfloor) \\
&\le (n+2) \ln n^{1/2} \\
&= \frac{n+2}{2} \ln n.
\end{aligned}
$$

why?

- Totally, W=? S=?

AN EXAMPLE: PRIMES

- Is there any wasteful in the *primeSieve* ?
  - *j* is not a *prime* we do not have to use its multiples
  - e.g, n=12

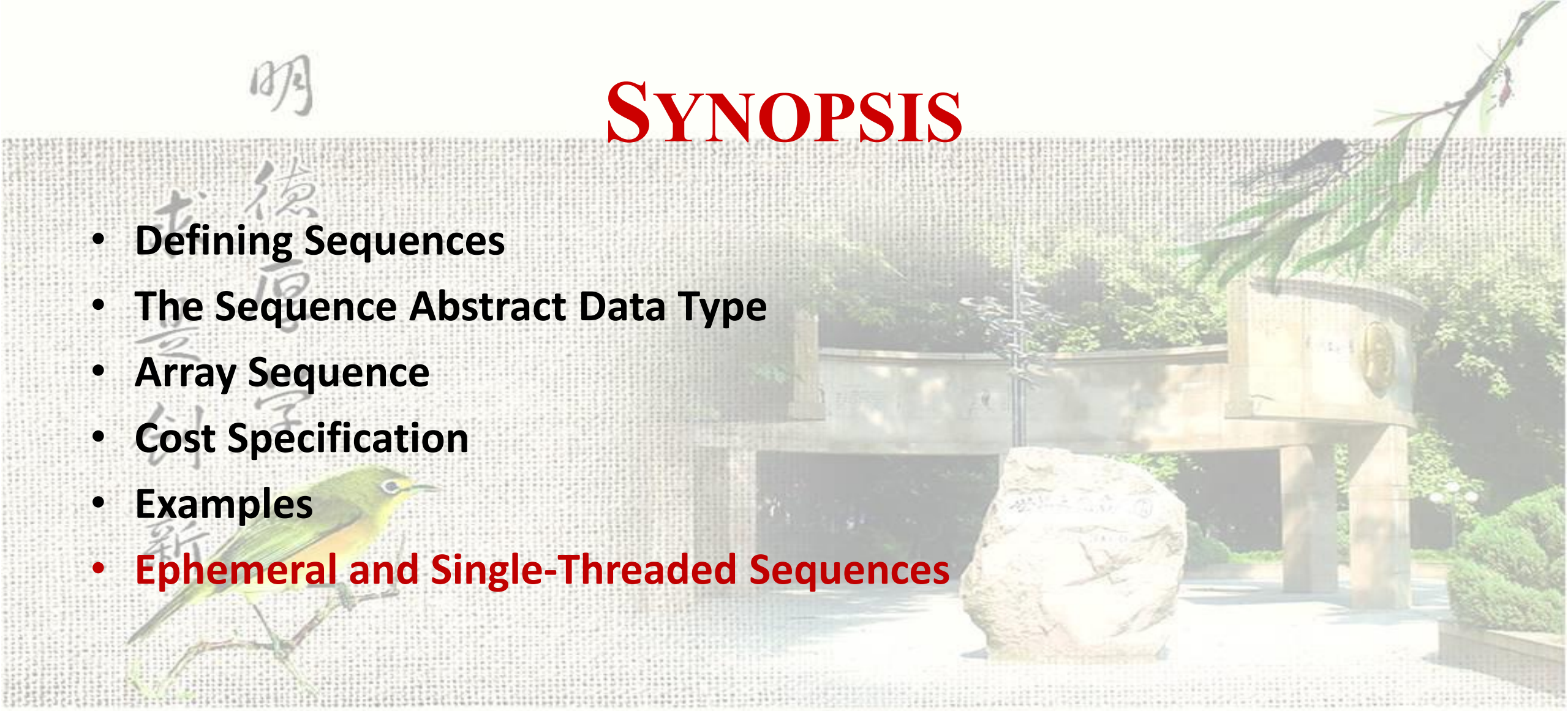
why?

```
function primes(n) =
if (n < 2) then ⟨ ⟩
else let
    val P = primes(⌈√n⌉)
    val sieves = ⟨(p × i, false) : p ∈ P, 1 ≤ i ≤ ⌈n/p⌉⟩
    val R = inject({true : 0 ≤ i ≤ n}, sieves)
in
    ⟨i : 2 ≤ i ≤ n | R[i]⟩
end
```

  - the analysis of this algorithm as an exercise?

SYNOPSIS

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



Ephemeral Data Structures

- Ephemeral data structure: an implementation of a data structure that destroys existing instances as ***ephemeral***
  - Not safe for parallelism
    - ✓ a=⟨0,1,…,n−1⟩
    - ✓ b=⟨(0,0),(1,2),(2,4),…,(n−1,2n−2)⟩
    - ✓ c=⟨(0,1),(1,3),(2,5),…,(n−1,2n−1)⟩
    - ✓ **inject** $a\ b$ **||** **inject** $a\ c$

How about the result?

Ephemeral Sequences

- **Constant Work Updates**
  - **We can create an ephemeral version of array sequences by changing the update, inject, and ninject primitives to update the input array destructively**
    - ✓ **For an update sequence of length** m
    - ✓ $\mathcal{O}(1)$ **work and span for update**
    - ✓ $\mathcal{O}(m)$ **work and** $\mathcal{O}(\lg d)$ **span for inject, where** d **is the degree of the update sequence**
    - ✓ $\mathcal{O}(m)$ **work and** $\mathcal{O}(1)$ **span for ninject**

how?

Persistent and Emphemeral Implementations

- **Persistent Data Structures**
  - ***pure*** or ***persistent*** implementations: "**non-destructive**"
    - ✓safe for parallelism
    - ✓does come with a cost: not allowed to update data destructively
      - in array sequences, the update $a\ (i,v)$ and inject $a\ b$ operations require $\Omega(|a|)$ work
      - In tree sequences, update $a\ b$ and inject $a\ b$ require $\Theta(\lg|a|)$ and $\Theta(\lg|a|+\lg|b|)$ work
      - In tree sequences, update $a\ b$ and inject $a\ b$ require $\Theta(|a|\ \lg|a|)$ and $\Theta((|a|+\lg|b|)\lg|a|)$ work

any question?

How to realize?

Single-Threaded Sequences

- **Data Type (Single Threaded Sequences)**
  - For any element type α, the α- ***single threaded sequence*** (stseq) data type is the type $\mathbb{T}_\alpha$ consisting of the set of all α stseq's, and the following functions
    - ✓fromSeq : $\mathbb{S}_\alpha \to \mathbb{T}_\alpha$
    - ✓toSeq : $\mathbb{T}_\alpha \to \mathbb{S}_\alpha$
    - ✓Nth : $\mathbb{T}_\alpha \to \mathbb{N} \to \alpha$
    - ✓Update : $\mathbb{T}_\alpha \to (\mathbb{N} \times \alpha) \to \mathbb{T}_\alpha$
    - ✓Inject : $\mathbb{T}_\alpha \to \mathbb{S}_{\mathbb{N}\times\alpha} \to \mathbb{T}_\alpha$
      - – where $\mathbb{S}_\alpha$ are standard sequences, and nth, update, and inject behave as they do for standard sequences

Single-Threaded Sequences

- **Cost Specification(Single Threaded Array Sequence)**

	Work	Span
fromSeq a	$O(\|a\|)$	$O(1)$
toSeq a	$O(\|a\|)$	$O(1)$
nth a i	$O(1)$	$O(1)$
update a (i, v)	$O(1)$	$O(1)$
inject a b	$O(\|b\|)$	$O(\lg(\text{degree}(d)))$

how?

Single-Threaded Sequences

- Implementation
  - **each sequence keep a version number**
    - ✓ **The version number is incremented each time the sequence is updated either with update or inject**
  - **each position in the sequence keep a version list**
    - ✓ **A version list is a linked list of all the updates to the corresponding position, each with the version number of when the update was made**
  - **The most recent version is kept at the head of the list, and the rest are kept in decreasing order**
    - ✓ **A mutable (impure) array is used to keep a pointer to the head of each list, and the version number is also mutable.**

How to realize?

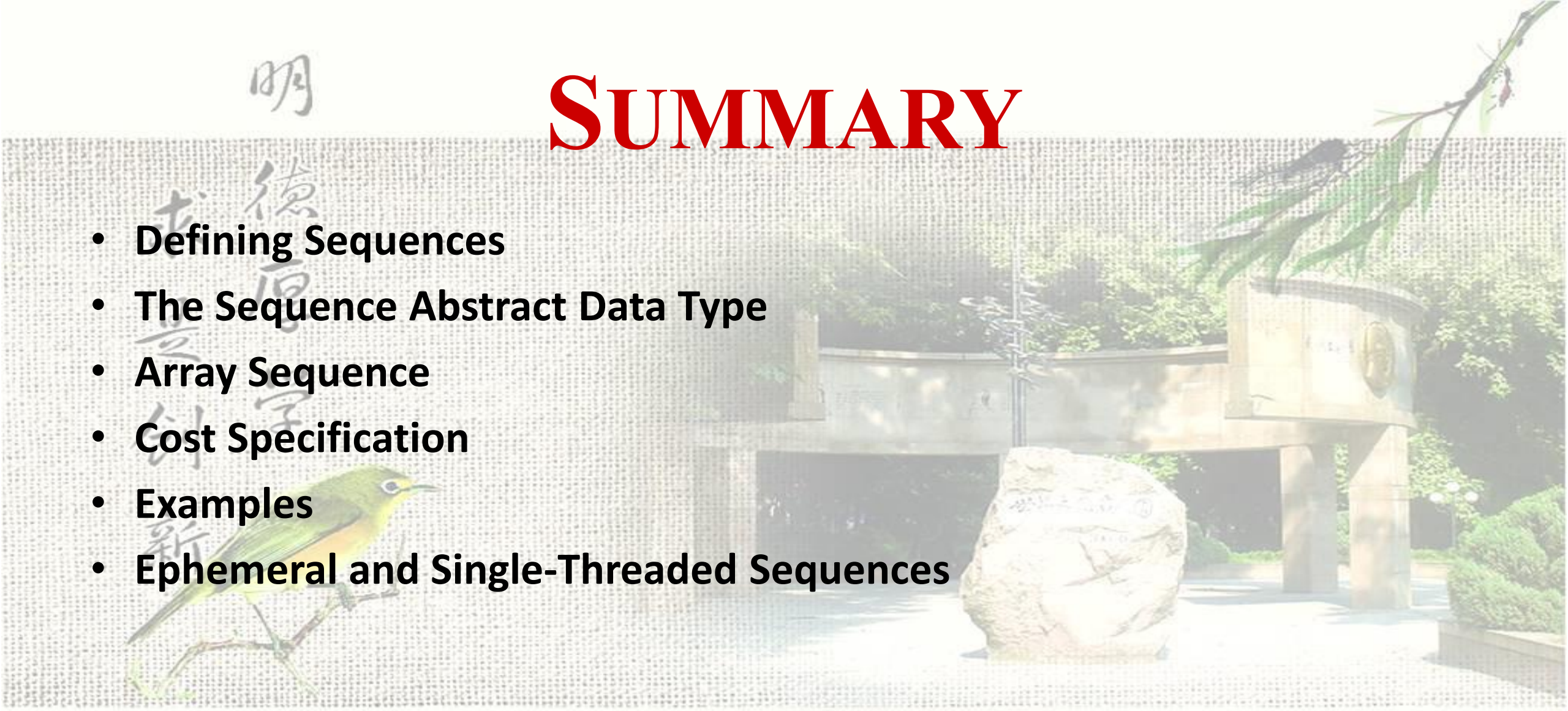
Single-Threaded Sequences

- how to do lookups and updates on the most recent version?
  - Lookup
    - ✓ look into the given location and take the first value from the head of the linked list, takes constant work.
  - Update
    1. updating the version number
    2. grabbing the appropriate list
    3. adding the value and version number to the front of the list
    4. writing back the new head of the list using mutation (if done right, this is a benign effect)
    - ✓ takes constant work
  - Looking up or updating an old version is more expensive

适合用在什么场景?

SUMMARY

- Defining Sequences
- The Sequence Abstract Data Type
- Array Sequence
- Cost Specification
- Examples
- Ephemeral and Single-Threaded Sequences



Excercise

1. **Design an algorithm that, for each element in a sequence of integers, finds the rightmost positive number to its left. If there is no positive element to the left of an element, the algorithm returns $-\infty$ for that element. For example, given the sequence $\langle 1,0,-1,2,3,0,-5,7\rangle$ the algorithm would return $\langle -\infty,1,1,1,2,3,3,3\rangle$.**
2. **Give an example function f, a left identity x, and an input sequence a such that** iterate $f\ x\ a$ **and** reduce $f\ x\ a$ **return different results.**
3. **Given that** reduce **and** iterate **are equivalent for assocative functions, why would we use** reduce**?**
4. **Analyze the cost of the algorithm for generating 2D points given above.**
5. **Present an algorithm that generates all contiguous subsequences of a given sequence.**
6. **Analyze the cost of your algorithm for Exercise All contiguous subsequences.**
7. **Given sequences a of natural numbers and b of letters of the alphabet, we wish to compute the sequence that pairs each even element of a with all elements of b that are vowels.**
8. **To generate all composite numbers between 2 and n, prove that it suffices to consider all $i \in \mathbb{N}$ i $1 \le i \le \sqrt{n}$ and all of i's multiples up to n/i.**