Introduction

• In this chapter you will learn about
  – Advanced Programming in F#
    • Composing functions
    • Tail recursion and continuations
    • Currying and closures
    • Sequences and lazy evaluation
    • Active patterns and computational expressions
    • Concurrency
  – An overview of other functional programming languages
• The chapter ends with a case study where you will build an interpreter for Wren Intermediate Code
Function Composition

• You want to write a function to return the number of digits in the square of a number

```latex
> lengthOfSquare 5;;
val it : int = 2
```

• Here are some helper functions

```latex
let square x = x * x
let toString (x : int) = x.ToString()
let strLen (x : string) = x.Length
```

• We now use the forward composition operator (>>) to build the desired function

```latex
> let lengthOfSquare = square >> toString >> strLen;;
val lengthOfSquare : (int -> int)
```

• Notice that formal parameters did not have to be specified
Composition Operators

• Forward and backward composition
  - let (>>>) f g x = g (f x)
  - let (<<) f g x = f (g x)

• Reworking the previous example using <<

> let lengthOfSquare = strLen <<
    toString << square;;
val lengthOfSquare : (int -> int)

> lengthOfSquare 5;;
val it : int = 2
The pipe forward operator

• The pipe forward operator ( |> ) allows you to pass the result from one operation to the next operation

```plaintext
> 5 |> square;;
val it : int = 25
> 5 |> square |> toString;;
val it : string = "25"
> 5 |> square |> toString |> strLen;;
val it : int = 2
```

• The pipe forward operator is defined as

```plaintext
let (|>) x f = f x
```
Supplying partial arguments using |> 

- Consider the following example
  "World" |> (fun x y -> x + y) "Hello";;
  - The anonymous function is expecting two arguments
  - The first argument appears to the right of the function
  - The second argument is supplied by the pipe
  - So the result is “HelloWorld”

- A more complex example
  > "World" |> ("Hello " |> (fun x y z -> x + y + z)) "Cruel ";;
  val it : string = "Hello Cruel World"

- Notice the use of parentheses
The pipe backward operator

• The pipe backward operator is defined as
  \[
  \text{let } (<|) \ f \ x = f \ x
  \]

• A familiar example
  \[
  \text{let } (<|) \ f \ x = f \ x
  \]

> strLen <| (toString <| (square <| 5));;
val it : int = 2

• Since the definition of <| is the same as normal function application, it is rarely used. Consider
  \[
  \text{let } (<|) \ f \ x = f \ x
  \]

> strLen (toString (square 5));;
val it : int = 2
Lab Activity Ch05.01

• Define two simple functions, cube and increment, that perform the operations indicated by the name.

• Define a function doCubeIncr that acts as follows:
  doCubeIncr 3 returns 28
  Use the >> operator.

• Define a function doCubeIncr2 that acts as follows:
  doCubeIncr2 3 returns 28
  Use the << operator.
Implementing Recursion

• The stack of activation records
  – Every time a function is called it causes an activation record to be added to the runtime stack
  – This is true for a recursive function calling itself
  – If a function recurses over a long list one item at a time there will be an activation record for each item
  – This can be very slow and wasteful

• Characteristics of tail recursion
  – No work is done when returning from recursion
  – The final answer is known when the base case is reached
  – F# recognizes this situation and avoids building a stack of activation records to improve efficiency
A First Example - 1

• Here is a factorial function written “as usual”

```fsharp
let rec factorial n =
    if n <= 1
    then 1
    else n * factorial(n-1);

factorial 10
```

A breakpoint has been set on the line when the base case is reached

• Here is the call stack at the breakpoint

![Call Stack](image)
A First Example - 2

• Since F\# did not recognize this as tail recursion there is a full stack of activation records

• You might ask why this isn’t tail recursion since the last thing that happens is the recursive call
  – There is work being performed when returning from recursion, namely $n \times \text{<recursive result>}$
  – The final result is not known when the base case is encountered

• Designing for tail recursion
  – Do the work ($\times$ in this case) going into recursion
  – Introduce new parameters to hold accumulated results
  – At the base case the accumulated result is the answer
A First Example - 3

• A tail recursive factorial function

```fsharp
let factorialTest n =
    let rec TRfactorial n acc =
        if n <= 1 then
            acc
        else
            TRfactorial (n-1) (acc * n)
    TRfactorial n 1

factorialTest 10
```

Multiplication is done going into recursion and the result is accumulated.

Notice the use of an inner function that is TR.

• The call stack at the breakpoint; F# recognized tail recursion and did not stack the activation records.
Importance of Tail Recursion - 1

• Why you need to design your functions this way
  – At the end of this chapter you will write a WIC interpreter in F#; if each instruction is interpreted by a function call then a non-tail recursive version would stack an activation record for each instruction executed.
  – If the program has a loop that executes tens of thousands or hundreds of thousands of times (e.g., the gcd of 100000 and 1) then the call stack would quickly exceed the allowed limits and the interpreter would crash.
  – The interpreter must be written tail recursively; the “accumulator” will be a record structure representing the current state of the machine.
Importance of Tail Recursion - 2

- Have the functions written in chapter 4 been tail recursive?
  - Most were not tail recursive; if you are processing a list and the base case is: \[
  \[
  \vec{} \vec{[]}
  \rightarrow \vec{[]}
  \]
  then it is clearly not tail recursive
  - Once you have a working function that is not tail recursive, try to make it tail recursive
  - Can the work being performed after recursion (e.g., cons an item onto a list) be done going into recursion?
  - Can extra parameters be used to accumulate the final result?

- We practice this process on the next slides
Lab Activity Ch05.02

• Here is a non-TR version of the fibonacci function

```ml
let rec fibonacci n =
    if n <= 1 then 1
    else fibonacci(n-1) + fibonacci(n-2);
```

• Write a TR version
  – Hint: you need two extra parameters to hold the previous and current values
  – You will need an inner TR function that uses these values and knows the final result when the base case is encountered
  – The outer function will only have the parameter n and supply values for the other parameters
Converting the map function to TR - 1

• Here is the original function, it is not TR

> let rec map func list =
   match list with
   | [] -> []
   | hd::tl -> (func hd):: map func tl;;
val map : ('a -> 'b) -> 'a list -> 'b list

• Some ideas on converting to tail recursion
  – The cons operation (::) is done coming out of recursion, we must somehow do this while going into recursion
  – This will require an extra parameter to hold the accumulated result
Converting the map function to TR - 2

• Here is a first attempt an the inner function mapTR

```haskell
> let rec mapTR func list acc =
  
  match list with
  | [] -> acc
  | hd :: tl -> mapTR func tl (func hd :: acc);

val mapTR : ('a -> 'b) -> 'a list -> 'b list
```

• Testing this function

```haskell
> mapTR (fun n -> n * n) [1; 2; 3; 4] [] ;;
val it : int list = [16; 9; 4; 1]
```

• The result is correct but in reverse order

  – This is not surprising, if we push items onto a stack then pop them off they are reversed
  
  – This is what happens when we do the cons first and not afterward
Converting the map function to TR - 3

• A first possible fix – use @ instead of ::
  
  ```
  | hd :: tl -> mapTR func tl (acc @ [func hd])
  ```

  – This will work correctly but the implementation of the append operation @ where a new list has to be build based on two list parameters is significantly less efficient than the cons operation :: where a single item has to be inserted at the head of a given list

• A second possible fix – reverse the given list

```
let map func list =
  let rec mapTR func list acc =
    match list with
    | [] -> acc
    | hd :: tl -> mapTR func tl (func hd :: acc)
  mapTR func (List.rev list) []
```
Lab Activity Ch05.03

• Here is a non-TR version of a filter function

```plaintext
let rec filter fcn lst =
    match lst with
    | [] -> []
    | hd::tl -> let r = filter fcn tl
                if fcn hd then hd :: r
                else r
```

• Write a TR version

  – Add an accumulator and use the cons operation in an inner TR version of the function
  – Your outer function will need to reverse the given list
Function Continuations

• What is a continuation?
  – Rather than passing the current state of the accumulator as a parameter to the next function call, F# allows you to pass a function value representing the rest of the code to be executed as a parameter
  – Continuations are placed on the heap and not the runtime call stack

• When can we use continuations?
  – Many doubly recursive functions, such as traversing a tree, cannot be made tail recursive directly
  – Continuation can be used to remember operations that have not been completed yet
A First Simple Example

• A factorial function with continuations

```plaintext
let factorial n =
    let rec factorialCont n cont =
        match n with
        | 0 -> cont()
        | _ -> factorialCont (n-1) (fun() -> n * cont())
    factorialCont n (fun () -> 1);;

val factorial : int -> int

> factorial 5;;
val it : int = 120
```

• This example shows the syntax of using continuations; since factorial is singly recursive you don’t need continuations to write a TR version
Traversing a Binary Tree

- A recursive algorithm for a preorder traversal

```ocaml
type BinTree<'a> =
  | Node of 'a * BinTree<'a> * BinTree<'a>
  | Empty

let rec iter func binTree =
  match binTree with
  | Empty -> ()
  | Node(data, left, right) ->
    func data
    iter func left
    iter func right
```

- Testing the traversal

```ocaml```
```ocaml
let T1 = Node(6, Node(9, Empty, Node(3, Empty, Empty)), Node(2,
                      Node(7, Empty, Empty), Empty))

> iter (fun d -> printf "%d " d) T1;;
6 9 3 2 7
```
```ocaml```
```ocaml```
• First we present the code

type ContinuationStep<'a> =
    | Finished
    | Step of 'a * (unit -> ContinuationStep<'a>)

let iter func binTree =

    let rec linearize binTree cont =
        match binTree with
        | Empty -> cont()
        | Node(data, left, right) ->
            Step(data, (fun () -> linearize left
                         (fun() -> linearize right cont)))

    let steps = linearize binTree (fun () -> Finished)
Using Continuations and TR - 2

```
let rec processSteps step =
  match step with
  | Finished -> ()
  | Step(data, getNext) -> func data
      processSteps (getNext())

  processSteps steps

> iter (fun d -> printf "%d " d) T1;;
6 9 3 2 7 val it : unit = ()
```

- This algorithm was borrowed from Programming F# by Chris Smith
- The next slide explains some of the details
Using Continuations and TR - 3

- ContinuationStep is a custom iterator with two data tags Finished and Step.
- Step is a tuple of a value and a function to produce the next ContinuationStep.
- Linearize function
  - uses a continuation to convert the binary tree into a ContinuationStep object.
  - Each node is converted into a Step in addition to breaking the recursive calls to the left and right subtrees into lambda function.
- processSteps breaks apart the ContinuationStep object to actually perform the iter operation on the binary tree.
Currying

• Consider the following example involving cars

let automobiles = ["ford fusion", 22000.00<usd>, 28<mpg>);
    ("bmw 3 series", 35000.00<usd>, 24<mpg>);
    ("toyota prius", 26000.00<usd>, 45<mpg>)]

let price (name:string, cost:float<usd>, mileage:int<mpg>) = cost
let milesPerGal (name:string, cost:float<usd>, mileage:int<mpg>) = mileage

let pickLeastExpensive automobiles =
    List.reduce
        (fun acc item -> if price item < price acc
            then item
            else acc)
        automobiles

let pickMostEconomical automobiles =
    List.reduce
        (fun acc item -> if milesPerGal item > milesPerGal acc
            then item
            else acc)
        automobiles

Notice the use of <measure> types

Only the comparison differs in these two function definitions
Reducing Duplicate Code

- Testing the original version of these functions

```ocaml
> pickLeastExpensive automobiles;;
val it : string * float<usd> * int<mpg> =
    ("ford fusion", 22000.0, 28)
> pickMostEconomical automobiles;;
val it : string * float<usd> * int<mpg> =
    ("toyota prius", 26000.0, 45)
```

- Extracting out the common element of pickMostEconomical and pickLeastExpensive

```ocaml
let pickAuto cmp automobiles =
    let reduceFunc acc item =
        match cmp acc item with
        | true  -> item
        | false -> acc
    in
    List.reduce reduceFunc automobiles
```
Writing new versions of our functions

• Using currying to write the new versions of our functions

```plaintext
let pickLeastExpensive2   =
    pickAuto (fun acc item -> price item < price acc)
let pickMostEconomical2 =
    pickAuto (fun acc item-> milesPerGal item > milesPerGal acc)
```

• Testing our new functions

```plaintext
> pickLeastExpensive2 automobiles;;
val it : string * float<usd> * int<mpg> =
    ("ford fusion", 22000.0, 28)
> pickMostEconomical2 automobiles;;
val it : string * float<usd> * int<mpg> =
    ("toyota prius", 26000.0, 45)
```

• Currying is taking a function with multiple arguments and decomposing it into a sequence of function calls using single arguments
Closure

• Closure allows a function to access variables outside its typical scope
  – The referencing environment binds the nonlocal names to the corresponding variables in scope at the time the closure is created
  – The key idea is the names are bound not the values

• An earlier example that used closure
  
  let taxCollected purchases taxRate = List.fold (+) 0.0 (List.map (fun item -> item*taxRate) purchases)
  – In the anonymous function the tax rate is being captured from the enclosing environment
Using a partial function and closure

• Define a function to apply a tax

```ocaml
let applyTax func purchases = List.map func purchases;;
val applyTax : ('a -> 'b) -> 'a list -> 'b list
```

• Now use this function with a particular tax rate

```ocaml
let purchases = [7.98; 15.67; 24.32; 8.75];;
val purchases : float list = [7.98; 15.67; 24.32; 8.75]
let taxRate = 0.075;;
val taxRate : float = 0.075
applyTax (fun item -> item * taxRate) purchases;;
val it : float list = [0.5985; 1.17525; 1.824; 0.65625]
```

• Now assign a higher tax rate call applyTax again.

```ocaml
let taxRate = 0.085;;
val taxRate : float = 0.085
applyTax (fun item -> item * taxRate) purchases;;
val it : float list = [0.6783; 1.33195; 2.0672; 0.74375]
```
Lab Activity Ch05.04

• You are to write a function to print a message base on the current value stored as a balance in a bank account:
  Over $500: print “You are OK; your balance is over $500”
  $100 - $500: print “Your balance is OK but less than $500”
  $1 - $99: print “Your balance is below $100; spend carefully”
  $0 or lower: print “You have no money left in your account”

• Use closure to capture the current balance amount.

• Demonstrate your program running where you get a different message as you change the value of the balance
Sequences and Lazy Evaluation

- Sequences are similar to lists
  ```ocaml
  let oddNumbers = seq { 1..2..20 };;
  val oddNumbers : seq<int>
  ```
  - Notice the values were not printed out
  - Values are generated on an “as needed” basis
  - This allows you to define a very long sequence without using a lot of memory

- Using our oddNumbers sequence
  ```ocaml
  Seq.iter (fun num -> if num % 5 = 0 then
                      printf "%d " num else printf "" ) oddNumbers;;
  val it : unit = ()
  ```
Using Sequences

• Define a sequence of random numbers

```csharp
open System
let randomIntegers =
    seq { let randomNumGen = new Random()
        while true do
            yield randomNumGen.Next()
    }
```

• Use the take function to retrieve five values

```csharp
> Seq.take 5 randomIntegers;;
val it : seq<int> = seq [244533994; 1522820776;
    1700673971; 2110949447; ...]
```

• The Seq.exists function will test if a particular value is contained in the sequence
Other functions on sequences

• Using `seq.filter` on our sequence of odd numbers

```ocaml
> Seq.filter (fun num -> num % 5 = 0) oddNumbers;;
val it : seq<int> = seq [5; 15]
```

• Using a map over a sequence

```ocaml
> let multiplesOfThree = Seq.map (fun num -> if num % 3 = 0
    then num else 0) <| seq{1..20};;
val multiplesOfThree : seq<int>
> multiplesOfThree;;
val it : seq<int> = seq [0; 0; 3; 0; ...
```

• Now apply a fold operation

```ocaml
> Seq.fold (+) 0 <| multiplesOfThree;;
val it : int = 63
```
A More Complex Example

- Terms in a sequence based on $\sum_{j=1}^{n} j$
  - The first few values are 1, 3 (=1+2), 6 (=1+2+3), 10 (=1+2+3+4), and so forth
  - You will use a tuple of the form: (value at current position, tuple for the next generator)
  - For example after the third value = 6 the tuple to generate the next value is (6, (4,6))

- Here is the code

```haskell
> let Summation (a, b) =
  if a > 12 then None
  else let nextValue = a + b
       Some(nextValue, (a + 1, nextValue))
val Summation : int * int -> (int * (int * int)) option
```
Using the summation sequence

- Unfold the sequence
  
  ```
  let result = Seq.unfold Summation (1,0);
  val result : seq<int>
  ```

- Now use take to get the first 12 values
  ```
  Seq.take 12 result;;
  val it : seq<int> = seq [1; 3; 6; 10; ...
  ```

- If you want to see all 12 values use the toList function
  ```
  Seq.toList result;;
  val it : int list = [1; 3; 6; 10; 15; 21; 28; 36; 45; 55; 66; 78]
  ```
Lab Activity Ch05.05

• Write a seq to generate the terms in the Fibonacci sequence.

• You will use the tuple approach shown previously in summation but your tuple will now contain the previous value and the current value.
The Eliza Program

• History
  – Joseph Weizenbaum, a professor at MIT, introduced a program named Eliza in 1966 where a computer mimics a person talking in a conversation
  – The simple version shown here is based on some patterns suggested by Ravi Sethi using ML

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>RESPONSE TEMPLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>my single</td>
<td>tell me about your single</td>
</tr>
<tr>
<td>i am</td>
<td>i am sorry to hear you are</td>
</tr>
<tr>
<td>am i</td>
<td>do you believe you are</td>
</tr>
<tr>
<td>you</td>
<td>why do you think i</td>
</tr>
<tr>
<td>me</td>
<td>in what way</td>
</tr>
</tbody>
</table>
well my friend made me come here
tell me about your friend
he says i am depressed
   i am sorry to hear you are depressed
i think i need help
   in what way
oh am i making sense so far
   do you believe you are making sense so far
you are making fun of me
   why do you think i are making fun of you
Using regular expressions

• Consider the first pattern
  \[ _____1 \text{my } \text{single} \] \[ ____2 \text{me} \] \[ ______3 \text{which produces the computer response “tell me about your single”} \]

• The regular expression is

  Match "^.*my\s+(\w+)\s+.\*.me.*$" result ->
  "Tell me about your " + (List.nth result 0).ToString()

  • ^ anchors the start of the expression
  • .*my means any sequence of characters before “my”, this corresponds to phrase 1 in the pattern; this isn’t used in the response so it is discarded
  • \s+ indicates one of more spaces
  • (\w+) captures the sequence of characters making up the word single; the (..) indicates the capture, it is put into the result variable, and \w+ indicates characters in a word (they cannot be white space)
  • \s+.\*me indicates one or more spaces followed by any sequence of characters before “me” is encountered; the sequence of characters is phrase 2, which is discarded
  • .*$ denotes any sequence of characters after me which is phrase 3 that is discarded; $ is the closing anchor
An Active Pattern

- The Match active pattern
  - It has two parameters, the input the user typed in and the pattern in a regular expression that is being matched.

It uses an optional type in case there is no match

```csharp
let (|Match|_|) (pat:string) (inp:string) =
  let m = Regex.Match(inp, pat)
  if m.Success
    then Some (List.tail [for g in m.Groups->g.Value ])
  else None
```
let matchInput (sentence:string) =
    let theMatch = match sentence with
        | Match "^.*my\s+(\w+)\s+.*me.*$" result ->
            "Tell me about your " + (List.nth result 0).ToString()
        | Match "^.*i\s+am\s+(.*)$" result ->
            "I am sorry to hear you are " +
             (List.nth result 0).ToString()
        | Match "^.*am\s+i\s+(.*)$" result ->
            "Do you believe you are " +
             (List.nth result 0).ToString()
        | Match "^.*you\s+(.*)\s+me.*$" result ->
            "Why do you think I " +
             (List.nth result 0).ToString() + " you";
        | _ -> "In what way"
    printfn "%A\n" theMatch
The main function and Testing

let rec eliza() =
    let sentence = Console.ReadLine()
    if sentence <> "" then
        matchInput sentence
        eliza()
    else
        System.Environment.Exit;

• Running the program

let test = eliza();;
my friend hit me
"Why do you think your friend hit you?"
because i am dumb
"I am sorry to hear you are dumb"
am i crazy or something
"Do you believe you are crazy"
you hate me
"Why do you think I hate you"
Lab Activity Ch05.06

• Get the Eliza program running and then add two more patterns to be matched to the list of patterns. Make the patterns significantly different than the given patterns. You can insert them anywhere in the sequence of patterns.

• Hold a conversation with Eliza that demonstrates all of the possible patterns
A Computational Expression

> let daysOfYear =
  seq {
    let months =
      [ "Jan"; "Feb"; "Mar"; "Apr"; "May"; "Jun";
        "Jul"; "Aug"; "Sep"; "Oct"; "Nov"; "Dec"
      ]
    let daysInMonth month =
      match month with
      | "Feb" -> 28
      | "Apr" | "Jun" | "Sep" | "Nov"
        -> 30
      | _     -> 31
    for month in months do
      for day = 1 to daysInMonth month do
        yield (month, day)
  };;
val daysOfYear : seq<string * int>
Testing the Days in the Year

• Printing out the first few days
  > daysOfTheYear;;
val it : seq<string * int> =
  seq [("Jan", 1); ("Jan", 2); ("Jan", 3); ("Jan", 4); ...

• Finding a particular day in the year (zero based)
  > Seq.nth 35 daysOfTheYear;;
val it : string * int = ("Feb", 5)

• Finding the number of days in the year
  > Seq.length daysOfTheYear;;
val it : int = 365

• Finding the last day of the year
  > Seq.nth 364 daysOfTheYear;;
val it : string * int = ("Dec", 31)
Lab Activity Ch05.07

- Add a binding “let year = ####” to the computational expression so that leap day (“Feb”, 29) is added for each leap year. Here is the pseudocode algorithm given in Wikipedia to determine if a year is a leap year or not:

```python
if year modulo 400 is 0
    then is_leap_year
else if year modulo 100 is 0
    then not_leap_year
else if year modulo 4 is 0
    then is_leap_year
else not_leap_year
```
Concurrency

• Defining and running a single thread

```fsharp
let thread1() =
    for i in 0 .. 7 do
        Thread.Sleep(70)
        printfn "[Thread %d] %d..." Thread.CurrentThread.ManagedThreadId i;;
val thread1 : unit -> unit
> let thread = new Thread(thread1)
thread.Start();;
val thread : Thread
> [Thread 11] 0...
[Thread 11] 1...
[Thread 11] 2...
[Thread 11] 3...
[Thread 11] 4...
[Thread 11] 5...
[Thread 11] 6...
[Thread 11] 7...
```

Yes, there are for loops in F#
• A function to start threads

```c
let startThread (t : unit -> unit) =
    let thread = new Thread(t)
    thread.Start();

val startThread : (unit -> unit) -> unit
```

• And a second thread

```c
> let thread2() =
    for i in 6..-1..1 do
        Thread.Sleep(80)
        printfn "[Thread %d] %d..."
        Thread.CurrentThread.ManagedThreadId i;;

val thread2 : unit -> unit
```

• Starting the two threads

```c
> startThread thread1
startThread thread2;;
```

• The output will have one thread counting from 1 to 7 and the other thread counting from 6 down to 1
Occasionally the Output Messes Up

The system has swapped between the two threads in the middle of the print of values.

We need to make the print operation atomic; we will learn how to use a lock to do this.
Using lock to fix the lost updates

- Two or more threads update a global variable

```fsharp
let raceDemo =
    let count = ref 0;
    let increment1() =
        for i in 1 .. 500000 do
            count := !count + 1
    let increment2() =
        for i in 1 .. 500000 do
            count := !count + 1
    startThread increment1
    startThread increment2
    Thread.Sleep(2000) // pause to let updates finish
    printfn "count = %d" !count
```

Yes, you can have global variables in F#

And, yes, there is an assignment statement in F#
The Lost Update Problem

- When two threads try to update the same global variable at the same time, one of the updates may be lost
- Sample outputs from the raceDemo function
  count = 657580
  count = 599945
  count = 616581
- The expected value is 1000000 so many updates were lost
- What would be the smallest possible value if the two thread updates always collided?
Race Conditions

• Lock the count variable when incrementing

```ml
> let raceDemoWithLock =
  let count = ref 0;
  let increment1() =
    for i in 1 .. 500000 do
      lock(count) (fun () -> count := !count + 1)
  let increment2() =
    for i in 1 .. 500000 do
      lock(count) (fun () -> count := !count + 1)
  startThread increment1
  startThread increment2
  Thread.Sleep(2000)
  printfn "count = %d" !count;;

val raceDemoWithLock : unit = ()
```

Every run of this program will print 1000000
Potential Problems When Using Locks

- The textbook introduces a bank account program where two users try to transfer funds from one account to a second account at the same time.

- The transfers use a lock to protect the transfers from interference; but here is what was printed:

```
Locking Jane Doe's account to deposit funds...
Locking John Smith's account to deposit funds...
Locking John Smith's account to withdraw funds...
Locking Jane Doe's account to withdraw funds...
```

  - **Interrupt**

- Both accounts are locked by one user so the other user cannot complete the transaction.

- This is known as deadlock.
A More Complex Example

• Counting the spaces in three separate files

```fsharp
> open System.IO
open System
let countSpace filename size =
    let space = Convert.ToByte ' '
    use stream = File.OpenRead filename
    let bytes = Array.create size space
    let nbytes = stream.ReadUInt8Array (bytes,0,size)
    let count =
        bytes |
        Array.fold (fun acc x -> if (x=space) then
            acc + 1
        else acc) 0
    count

let files = (DirectoryInfo @"C:\Users\kurtzbl\text documents").GetFiles "*.txt"
let counted =
    files |> Array.map (fun f -> countSpace (f.FullName) (int f.Length));;
val countSpace : string -> int -> int
val files : IO.FileInfo [] = [|document1.txt; document2.txt; document3.txt|]
val counted : int [] = [|114; 231; 98|]```

Since these are different files there is a potential for concurrency here.

The `[ | ... | ]` is an array structure

The number of spaces in each file
Creating an Asynchronous Block

open System.IO
open System

let aCountSpace filename size =
    async {
        let space = Convert.ToByte ' ' 
        use stream = File.OpenRead (filename)
        let bytes = Array.create size space
        let! nbytes = stream.AsyncRead (bytes,0,size)
        let count =
            bytes |
            Array.fold (fun acc x -> if (x=space) then acc + 1 else acc) 0
        return count
    }

val aCountSpace : string -> int -> Async<int>
Using a Workflow

• Here is how the asynchronous block is invoked

```plaintext
> let aCounted =
  files
  |> Array.map (fun f -> aCountSpace (f.FullName) (int f.Length))
  |> Async.Parallel
  |> Async.RunSynchronously;;
val aCounted : int [] = [|114; 231; 98|]

• The answers are the same as the non-concurrent version
Applying a Function Across a List

• Consider our TR factorial function

```ocaml
let factorial x =
  let rec factorialTR x acc =
    if x = 0 then acc
    else factorialTR (x-1) (x*acc)
  factorialTR x 1;;
val factorial : int -> int
```

• We want to apply this function in parallel across an array of integer values

```ocaml
> let concurrentFactorial arr =
  Array.Parallel.map factorial arr;;
val concurrentFactorial : int [] -> int []
> concurrentFactorial [|1;3;9;2;7|];;
val it : int [] = [|1; 6; 362880; 2; 5040|]
```
Lab Activity Ch05.08

• Write a tail recursive version of the Fibonacci function and use it to write a concurrentFibonacci function. Apply your function to a small list of integer values.
A Final Example

- This example illustrates the object-oriented aspects of F# by using generic concurrent queue

```fsharp
> open System.Collections.Concurrent

let cq = new ConcurrentQueue<int>()

[1..10] |> List.iter cq.Enqueue

let success, firstItem = cq.TryDequeue()

if success then
    printfn "%d was successfully dequeued" firstItem
else
    printfn "Data race, try again later"

1 was successfully dequeued

val cq : Collections.Concurrent.ConcurrentQueue<int>
val success : bool = true
val firstItem : int = 1
```
Set up Queue and Define a Producer

open System.Collections.Concurrent
let buffer = new ConcurrentQueue<int>();
val buffer : ConcurrentQueue<int>

let producer1() =
    for i = 1 to 7 do
        Thread.Sleep(50)
        buffer.Enqueue i;;
val producer1 : unit -> unit
Define another Producer and a Consumer

let producer2() =
    for i = -1 downto -4 do
        Thread.Sleep(80)
        buffer.Enqueue i;;
val producer2 : unit -> unit

let consumer() =
    Thread.Sleep(500); // let the buffer fill up
    while buffer.Count > 0 do
        let success,item = buffer.TryDequeue()
        if success then printfn "%d removed" item
        else printfn "buffer is empty";;
val consumer : unit -> unit
Let her rip!

startThread(producer1)
startThread(producer2)
startThread(consumer);

> 1 removed
-1 removed
2 removed
3 removed
-2 removed
4 removed
-3 removed
5 removed
6 removed
-4 removed
-7 removed