

Programming Paradigms

Lecture 4

Slides are from Prof. Chin Wei-Ngan and Prof. Seif Haridi from NUS

Higher-Order Programming

Reminder of last lecture

- Kernel language
 - linguistic abstraction
 - data types
 - variables and partial values
 - statements and expressions
- Kernel language semantics
 - Use operational semantics
 - Aid programmer in reasoning and understanding
 - Abstract machine model without details about registers and explicit memory address
 - Aid implementer to do an efficient execution on a real machine

Overview

- Computing with procedures
 - lexical scoping
 - closures
 - procedures as values
 - procedure call
- Higher-Order Programming
 - proc. abstraction
 - lazy arguments
 - genericity
 - loop abstraction
 - folding

Procedures

- Defining procedures
 - how to handle external references?
 - which variables matter?

- Calling procedures
 - what do the variables refer to?
 - how to pass parameters?
 - how about external references?
 - where to continue execution?

Identifiers in Procedures

```
P = proc { $ X Y }
      if X>Y then Z=1 else Z=0 end
end
```

- P captures the declared procedure
- x and Y are called *(formal) parameters*
- z is called an *external reference*

Free and Bound Identifiers

```
local z in
    if x>y then z=1 else z=0 end
end
```

- *x* and *y* are *free (variable) identifiers* in this statement
- *z* is a *bound (variable) identifier* in this statement

Free and Bound Identifiers

```
local z in
  if x>y then z=1 else z=0 end
end
```

Declaration Occurrence

- `x` and `y` are *free variable identifiers* in this statement (declared outside)
- `z` is a *bound variable identifier* in this statement (declared inside)

Free and Bound Occurrences

- An occurrence of `x` is *bound*, if it is inside the body of either local, proc or case.

```
local x in ...x... end
```

```
proc {$ ...x...} in ...x... end
```

```
case Y of f(x) then ...x... end
```

- An occurrence of `x` is *free* in a statement, if it is not a bound occurrence.

Free Identifiers and Free Occurrences

- *Free occurrences* can only exist in incomplete program fragments, i.e., statements that cannot run.
- In a running program, it is always true that every *identifier occurrence* is *bound*. That is it is in *closed-form*.

Free Identifiers and Free Occurrences

A1=15

A2=22

B=A1+A2

- The identifiers occurrences A1, A2, and B, are free.
- This statement cannot be run.

Free Identifiers and Free Occurrences

```
local A1 A2 in  
  A1=15  
  A2=22  
  B=A1+A2  
end
```

- The identifier occurrences A1 and A2 are bound and the occurrence B is free.
- This statement still cannot be run.

Free Identifiers and Free Occurrences

```
local B in  
  local A1 A2 in  
    A1=15  
    A2=22  
    B=A1+A2  
  end  
  {Browse B}  
end
```

- This is in closed-form since it has no free identifier occurrences.
- It can be executed!

Procedures

```
proc {Max X Y ?Z}      % "?" is just a comment
  if X>=Y then Z=X else Z=Y end
end
{Max 15 22 C}
```

- When Max is called, the identifiers x, y, and z are bound to 15, 22, and the unbound variable referenced by c.
- Can this code be executed?

Procedures.

- No, because Max and c are free identifiers!

```
local Max C in
    proc {Max X Y ?Z}
        if X>=Y then Z=X else Z=Y end
    end
    {Max 15 22 C}
    {Browse C}
end
```

Procedures with external references

```
proc {LB X ?Z}  
  if X>=Y then Z=X else Z=Y end  
end
```

- The identifier Y is not one of the procedure arguments.
- Where does Y come from? The value of Y *when the procedure is defined*.
- This is a consequence of static scoping.

Procedures with external references

```
local Y LB in
  Y=10
  proc {LB X ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  local Y=3 Z1 in
    {LB 5 Z1}
  end
end
```

- Call {LB 5 Z} bind z to 10.
- Binding of Y=3 when LB is called is ignored.
- Binding of Y=10 when the procedure is defined is important.

Lexical Scoping or Static Scoping

- The meaning of an identifier like `x` is determined by the innermost **local** statement that declares `x`.
- The area of the program where `x` keeps this meaning is called the **scope** of `x`.
- We can find out the scope of an identifier by inspecting the text of the program.
- This scoping rule is called **lexical scoping** or **static scoping**.

Lexical Scoping or Static Scoping

```
local x in
    x=15
    local x in
        x=20
        {Browse x}
    end
{Browse X}
end
```

$E_2 = \{x \rightarrow x_2\}$

$E_1 = \{x \rightarrow x_1\}$

- There is just one identifier, x, but at different points during the execution, it refers to different variables (x_1 and x_2).

Lexical Scoping

```
local z in
    z=1
    proc {P X Y} Y=X+z end
end
```

- A *procedure value* is often called a *closure* because it contains an *environment* as well as a *procedure definition*.

Dynamic versus Static Scoping

- *Static scope.*
 - The variable corresponding to an identifier occurrence is the one defined in the *textually innermost declaration* surrounding the occurrence in the source program.
- *Dynamic scope.*
 - The variable corresponding to an identifier occurrence is the one in the *most-recent declaration seen* during the execution leading up to the current statement.

Dynamic scoping versus static scoping

```
local P Q in
  proc {Q X} {Browse stat(X)} end
  proc {P X} {Q X} end
  local Q in
    proc {Q X} {Browse dyn(X)} end
    {P hello}
  end
end
```

- What should this display, `stat(hello)` or `dyn(hello)`?
- Static scoping says that it will display `stat(hello)`, because `P` uses the version of `Q` that exists at `P`'s definition.

Contextual Environment

- When defining procedure, construct *contextual environment*
 - maps all external references...
 - ...to values at the time of definition
- Procedure definition creates a closure
 - pair of procedure and contextual environment
 - this closure is written to store

Example of Contextual Environment

```
local Inc in
    local Z = 1 in
        proc {Inc X Y} Y = X + Z end
        local Y in
            {Inc 2 Y}
            {Browse Y}
    end
end
local Z = 2 in
    local Y in
        {Inc 2 Y}
        {Browse Y}
    end
end
end
```

Closure for
{Inc X Y}
has the mapping
{Z → 1}
based on where it was
defined.

Procedure Declaration

- Semantic statement is

$$(\text{proc } \{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\} \langle s \rangle \text{ end}, E)$$

- Formal parameters $\langle y \rangle_1, \dots, \langle y \rangle_n$
- External references $\langle z \rangle_1, \dots, \langle z \rangle_m$
- Contextual environment

$$CE = E \mid \{\langle z \rangle_1, \dots, \langle z \rangle_m\}$$

Procedure Declaration

- Semantic statement is

$(\text{proc } \{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\} \langle s \rangle \text{ end}, E)$

with $E(\langle x \rangle) = x$

- Create procedure value in the store and bind it to x

$(\text{proc } \{\$ \langle y \rangle_1 \dots \langle y \rangle_n\} \langle s \rangle \text{ end},$
 $E \upharpoonright \{\langle z \rangle_1, \dots, \langle z \rangle_m\})$

Execution of Procedure Call

- Semantic statement is

$$(\{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\}, E)$$

- If $\langle x \rangle$ is not bound, then
 - suspend the execution
- If $E(\langle x \rangle)$ is not a procedure value, then
 - raise an error
- If $E(\langle x \rangle)$ is a procedure value, but with different number of arguments ($\neq n$), then
 - raise an error

Procedure Call

- If semantic statement is

$$(\{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\}, E)$$

with

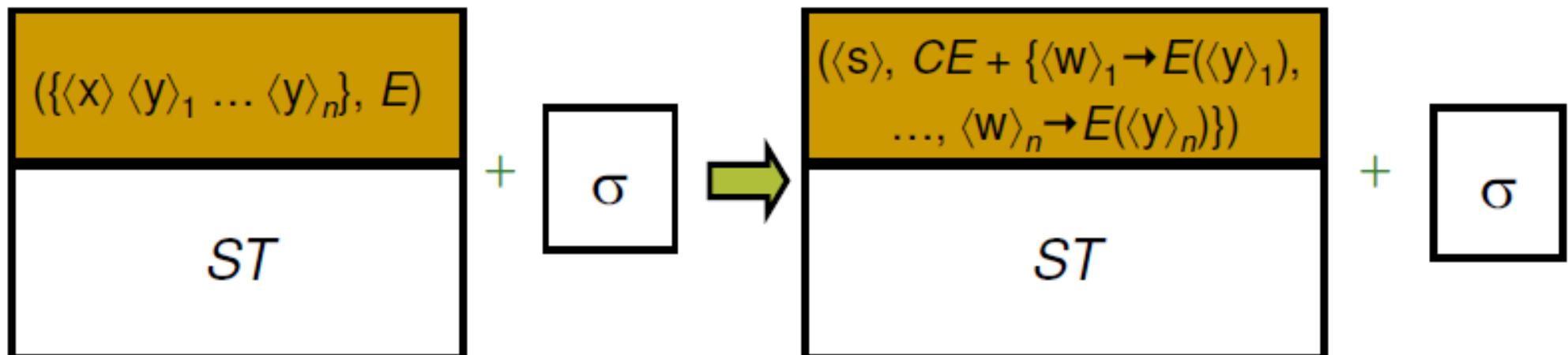
$$E(\langle x \rangle) = (\text{proc } \{ \$ \langle w \rangle_1 \dots \langle w \rangle_n \} \langle s \rangle \text{ end}, CE)$$

- then push

$$(\langle s \rangle, CE + \{\langle w \rangle_1 \rightarrow E(\langle y \rangle_1), \dots, \langle w \rangle_n \rightarrow E(\langle y \rangle_n)\})$$

Executing a Procedure Call

- If the activation condition “ $E(\langle x \rangle)$ is determined” is true
 - if $E(\langle x \rangle)$ equals to (proc {\$ $\langle w \rangle_1 \dots \langle w \rangle_n$ } $\langle s \rangle$ end, CE)



Summary so far

- Procedure values
 - go to store
 - combine procedure body and contextual environment
 - contextual environment defines external references
 - contextual environment is defined by lexical scoping
- Procedure call
 - checks for the right type
 - passes arguments by environments
 - contextual environment for external references

Discussion

- Procedures take the values upon definition.
- Application invokes these values.
- Not possible in Java, C, C++
 - procedure/function/method just code
 - environment is lacking
 - Java needs an object to do this
 - one of the most powerful concepts in computer science
 - pioneered in Lisp/Algol 68

Summary so far

- Procedures are values as anything else!
- Allow breathtaking programming techniques
- With environments, it is easy to understand what is the value for each identifier

Higher-Order Programming

Higher-Order Programming

- Higher-order programming = the set of programming techniques that are possible with procedure values (lexically-scoped closures)
- higher-order programming is the foundation of secure data abstraction component-based programming and object-oriented programming

Higher-order Programming

- Use of procedures as *first-class* values
 - can be passed as arguments
 - can be constructed at runtime
 - can be stored in data structures
- procedures are simply values!
- Will present a number of programming techniques using this idea

Remember (I)

- Functions are procedures
 - Special syntax, nested syntax, expression syntax
 - They have one argument to capture its result.

■ Example:

```
fun {F X}  
    fun {$ Y} X+Y end  
end
```

- A function that returns a function that is specialized on x
- Add result parameters to both `{F X}` and `{$ Y}` to convert to procedures.

Remember (II)

```
declare
fun {F X}
  fun {$ Y} X+Y end
end
{Browse F}
G={F 1}
{Browse G}
{Browse {G 2}}
```

- F is a function of one argument, which corresponds to a procedure having two arguments
→ <P / 2 F>
- G is an unnamed function
→ <P / 2>
- {G Y} returns 1+Y
→ 3

Remember (III)

- ```
fun {F X}
 fun {$ Y} X+Y end
end
```

**Type :** <Num> -> (<Num> -> <Num>)

- ```
fun {F X Y}
      X+Y
end
```

Type : (<Num>, <Num>) -> <Num>

Higher-Order Programming

- Basic operations:

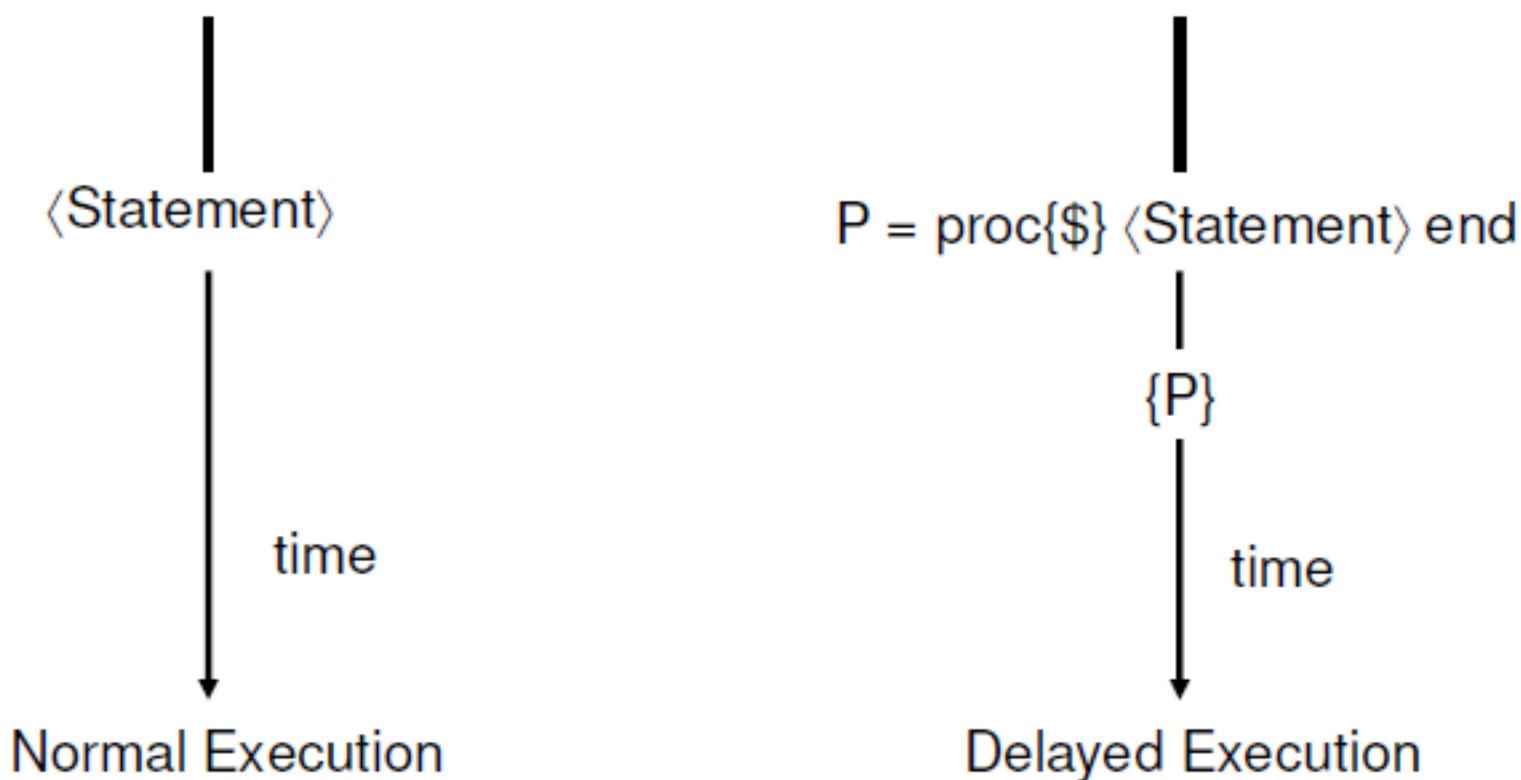
- Procedural abstraction: the ability to convert any statement into a procedure value
- Genericity: the ability to pass procedure values as arguments to a procedure call
- Instantiation: the ability to return procedure values as results from a procedure call
- Embedding: the ability to put procedure values in data structures

Higher-Order Programming

- Control abstractions
 - The ability to define control constructs
 - Integer and list loops, accumulator loops,
folding a list (left and right)

Procedural Abstraction

- Procedural abstraction is the ability to convert any statement into a procedure value



Procedural Abstraction

- A procedure value is usually called a **closure**, or more precisely, a **lexically-scoped closure**
 - A procedure value is a pair: it combines the procedure code with the contextual environment
- Basic scheme:
 - Consider any statement $\langle s \rangle$
 - Convert it into a procedure value:
 $P = \text{proc } \{ \$ \} \langle s \rangle \text{ end}$
 - Executing $\{ P \}$ has **exactly the same effect** as executing $\langle s \rangle$

Same Holds for Expressions

■ Basic scheme:

- Consider any expression $\langle E \rangle$
- Convert it into a function value:
 $F = \text{fun } \{ \$ \} \; \langle E \rangle \; \text{end}$
- Executing $x=\{F\}$ has exactly the same effect as executing $x=E$

The Arguments are Evaluated

```
declare Z=3
```

```
fun {F X}
```

```
  {Browse X} 2
```

```
end
```

```
Y={F Z+1}
```

```
{Browse Y}
```

- x is evaluated as $3+1$

$\rightarrow 4$

$\rightarrow 2$

```
declare Z=3
```

```
fun {F X}
```

```
  {Browse X}
```

```
  {Browse {X}} 2
```

- x is evaluated as function
value `fun {$} Z+1 end`

$\rightarrow <P/1>$

$\rightarrow 4$ ($3+1$ is evaluated)

```
end
```

```
Y={F fun {$} Z+1 end}
```

```
{Browse Y}
```

$\rightarrow 2$

Example

- Suppose we want to define the operator `andthen` (`&&` in Java) as a function, namely `<expr1> andthen <expr2>` is false if `<expr1>` is false, avoiding the evaluation of `<expr2>` (Exercise 2.8.6, page 109)
- Attempt:

```
fun {AndThen B1 B2}
    if B1 then B2 else false end
end

if {AndThen X>0 Y>0} then ... else ...
```

Example

```
if {AndThen X>0 Y>0} then ... else ...
```

- Does not work because both $x > 0$ and $y > 0$ are evaluated
- So, even if $x > 0$ is false, y should be bound in order to evaluate the expression $y > 0$!

Example

```
declare
fun {AndThen B1 B2}
    if B1 then B2 else false end
end
X=~3
Y
if {AndThen X>0 Y>0} then
    {Browse 1}
else
    {Browse 2}
end
```

- Display nothing since Y is unbound!
- When called, all function's arguments are evaluated, *unless* it is procedure value.

Solution: Use Procedural Abstractions

```
fun {AndThen B1 B2}
    if {B1} then {B2} else false end
end

if {AndThen
    (fun{$} X>0 end)
    (fun{$} Y>0 end) }
then ... else ... end
```

Example. Solution

```
declare
fun {AndThen BP1 BP2}
    if {BP1} then {BP2} else false end
end
X=~3
Y
if {AndThen
    fun{$} X>0 end
    fun{$} Y>0 end }
then {Browse 1} else {Browse 2} end
```

- Display 2 (even if Y is unbound)

Genericity/ Parameterization

- To make a function generic is to let any specific entity (i.e. operation or value) in the function body become an argument.
- The entity is abstracted out of the function body.

Genericity

- Replace specific entities (zero `0` and addition `+`) by function arguments

```
fun {SumList Ls}
  case Ls
  of nil then 0
    [] X|Lr then X+{SumList Lr}
  end
end
```

Genericity

```
fun {SumList L}
  case L of
    nil then 0
    [] X|L2 then X+{SumList L2}
  end
end
```



```
fun {FoldR L F U}
  case L of
    nil then U
    [] X|L2 then {F X {FoldR L2 F U}}
  end
end
```

Types of Functions

```
fun {SumList L}  
...  
SumList :: (List Int) -> Int
```



```
fun {FoldR L F U}  
...  
FoldR :: { (List A) ( {A B} -> B) B } -> B
```

Genericity SumList

```
fun {SumList Ls}
    {FoldR Ls (fun {$ X Y} X+Y end) 0}
end

{Browse {SumList [1 2 3 4] } }
```

Genericity ProductList

```
fun {ProductList Ls}
    {FoldR Ls (fun {$ X Y} X*Y end) 1 }
end

{Browse {ProductList [1 2 3 4]}}}
```

Genericity Some

```
fun {Some Ls}
  {FoldR Ls
    (fun {$ X Y} X orelse Y end) false }
end

{Browse {Some [false true false] } }

Some :: (List Bool) -> Bool
```

List Mapping

- **Mapping**
 - each element recursively
 - *calling function for each element*
 - Construct a new list from the input list
- Separate function calling by passing function as argument

Other Generic Functions: Map

```
fun {Map Xs F}
  case Xs of
    nil then nil
    [] X|Xr then {F X} | {Map Xr F}
  end
end

{Browse {Map [1 2 3]
  fun {$ X} X*X end} } % [1 4 9]
```

Other Generic Functions: Filter

```
fun {Filter Xs P}
  case Xs of
    nil then nil
    [] X|Xr then
      if {P X} then X|{Filter Xr P}
      else {Filter Xr P} end
    end
End
```

```
{Browse {Filter [1 2 3] IsOdd}} % [1 3]
```

Types of Functions

```
fun {Map Xs F}  
...  
Map :: { (List A) (A->B) } -> List B
```

```
fun {Filter Xs P}  
...  
Filter :: { (List A) (A->Bool) } -> List A
```

Instantiation

- Instantiation: ability to return procedure values as results from a procedure call
- A factory of specialized functions

```
declare
fun {Add X}
    fun {$ Y} X+Y end
end

Inc = {Add 1}
{Browse {Inc 5}} % shows 6
```

Embedding

- Embedding is when procedure values are put in data structures
- Embedding has many uses:
 - **Modules**: that groups together a set of related operations (procedures)
 - **Software components** : takes a set of modules as its arguments and returns a new module. Can be viewed as **specifying** a new module in terms of the modules it needs.

Embedding. Example

```
declare Algebra
local
    proc {Add X Y ?Z} Z=X+Y end
    proc {Mul X Y ?Z} Z=X*Y end
in
    Algebra=op(add:Add mul:Mul)
end
A=2
B=3
{Browse {Algebra.add A B} }
{Browse {Algebra.mul A B} }
```

- Add and Mul are procedures embedded in a data structure

Control Construct - For Loop

- Integer loop: repeats an operation with a sequence of integers

```
proc {For I J P}
    if I > J then skip
    else {P I} {For I+1 J P} end
end
{For 1 10 Browse}
```

For :: {Int Int (Int->())} -> ()

- Linguistic abstraction for integer loops

```
for I in 1..10 do {Browse I} end
```

Control Construct – ForAll Loop

- List loop: repeats an operation for all elements of a list

```
proc {ForAll Xs P}
  case Xs of
    nil then skip
    [] X|Xr then {P X} {ForAll Xr P}
  end
end
```

ForAll :: { (List A) A->() } -> ()

```
{ForAll [a b c d] proc{$ I} {Browse I} end}
```

- Linguistic abstraction for list loops

```
for I in [a b c d] do
  {Browse I}
end
```

Control Construct – Pipe/ Compose

- Can compose two functions together

```
fun {Compose P1 P2}  
  fun {$ X} {P1 {P2 X}} end  
end
```

Compose :: { (B->C) (A->B) } -> (A->C)

- Similar to pipe command used in Unix

P2 | P1

Folding Lists

- Consider computing the sum of list elements
 - ...or the product
 - ...or all elements appended to a list
 - ...or the maximum
 - ...or number of elements, etc
- What do they have in common?
- Example: `SumList`

SumList/Length

```
fun {SumList Xs}
  case Xs of
    nil then 0
    [] X|Xr then X + {SumList Xr} end
end

fun {Length Xs}
  case Xs of
    nil then 0
    [] X|Xr then 1 + {Length Xr} end
end
```

Right-Folding

■ Right-folding {FoldR [x₁ ... x_n] F S}

$$\{ F \ x_1 \ \{ F \ x_2 \ \dots \ \{ F \ x_n \ S \} \ \dots \} \}$$

or

$$x_1 \otimes_F (x_2 \otimes_F (\dots (x_n \underset{\text{...}}{\otimes}_F S) \dots))$$


FoldR

```
fun {FoldR Xs F S}
  case Xs
  of nil  then S
  [] X|Xr then {F X {FoldR Xr F S}} end
end
```

- Not tail-recursive
- Elements folded in order

Instances of FoldR

```
fun {SumList Xs}
  {FoldR Xs (fun {$ X R} X+R end) 0}
end

fun {Length Xs}
  {FoldR Xs (fun {$ X R} 1+R end) 0}
end
```

SumListT: Tail-Recursive

```
fun {SumListT Xs N}
  case Xs of
    nil  then N
    [] X|Xr then {SumListT Xr N+X}
  end
end
{SumListT Xs 0}
```

- Question:
 - How is this computation different from SumList?

Computation of Original sumList

$$\{ \text{SumList} [2 \ 5 \ 7] \} =$$

$$2 + \{ \text{SumList} [5 \ 7] \} =$$

$$2 + (5 + \{ \text{SumList} [7] \}) =$$

$$2 + (5 + (7 + \{ \text{SumList} \text{ nil} \})) =$$

$$2 + (5 + (7 + 0)) =$$

$$2 + (5 + 7) =$$

$$2 + 12 =$$

$$14$$

How Tail-Recursive SumListT Compute?

```
{ SumListT [2 5 7] 0 }           =
{ SumListT [5 7] 0+2 }           =
{ SumListT [5 7] 2 }             =
{ SumListT [7] 2+5 }             =
{ SumListT [7] 7 }               =
{ SumListT [] 7+7 }              =
{ SumListT [] 14 }               =
14
```

SumListT Slightly Rewritten...

{ SumListT [2 5 7] 0 } =

{ SumListT [5 7] {F 0 2} } =

{ SumListT [7] {F {F 0 2} 5} } =

{ SumListT nil {F {F {F 0 2} 5} 7} } =

...

where F is

fun {F X Y} X+Y **end**

Left-Folding

Left-folding {`foldL [x1 ... xn] F S`}

{F ... {F {F S X₁} X₂} ... X_n}

or

(... ((S \otimes_F X₁) \otimes_F X₂) ... \otimes_F X_n)



left is here!

FoldL and SumListT

```
fun {FoldL Xs F S}
  case Xs
    of nil  then S
    [] X|Xr then {FoldL Xr F {F S X}}
  end
end
```

FoldL :: { (List A) ((B A)->B) B } -> B

```
fun {SumListT Xs}
  {FoldL Xs (fun {Plus X Y} X+Y end) 0}
end
```

Properties of FoldL

- Tail recursive
- First element of list folded first...
 - that is evaluated first.

FoldL or FoldR?

- `FoldL` and `FoldR` can be transformed to each other, if function `F` is associative:

$$\{F \ X \ \{F \ Y \ Z\}\} == \{F \ \{F \ X \ Y\} \ Z\}$$

Other conditions possible.

- Otherwise: choose `FoldL` or `FoldR`
 - depending on required order of result

Example: Appending Lists

- Given: list of lists

$[[a\ b]\ [1\ 2]\ [e]\ [g]] \Rightarrow [a\ b\ 1\ 2\ e\ g]$

- Task: compute all elements in one list in order

- Solution:

```
fun {AppAll Xs}  
  {FoldR Xs Append nil}  
end
```

- Question: What would happen with FoldL?

What would happen with FoldL?

```
fun {AppAllLeft Xs}  
    {FoldL Xs Append nil}  
end
```

{AppAllLeft [[a b] [1 2] [e] [g]]} =

{FoldL [[a b] [1 2] [e] [g]] Append nil} =

{FoldL [[1 2] [e] [g]] Append {Append nil [a b]}} =

...

How Does AppAllLeft Compute?

```
{FoldL [[1 2] [e] [g]] Append [a b]} =  
  
{FoldL [[e] [g]] Append {Append [a b] [1 2]}} =  
  
{FoldL [[e] [g]] Append [a b 1 2]} =  
  
{FoldL [[g]] Append {Append [a b 1 2] [e]}} =  
  
{FoldL [[g]] Append [a b 1 2 e]} =  
  
{FoldL nil Append {Append [a b 1 2 e] [g]}} =  
  
{FoldL nil Append [a b 1 2 e g]} =  
= [a b 1 2 e g]
```

Summary so far

- Many operations can be partitioned into
 - pattern implementing
 - recursion
 - application of operations
 - operations to be applied
- Typical patterns
 - Map mapping elements
 - FoldL/FoldR folding elements
 - Filter filtering elements
 - Sort sorting elements
 - ...

Goal

- Programming as an engineering/scientific discipline
- An engineer can
 - understand abstract machine/properties
 - apply programming techniques
 - develop programs with suitable techniques

Summary

- Computing with procedures
 - lexical scoping
 - closures
 - procedures as values
 - procedure call
- Higher-Order Programming
 - proc. abstraction
 - lazy arguments
 - genericity
 - loop abstraction
 - folding