Programming Languages (CSCI 4430/6430)

Part 1: Functional Programming: Summary

Carlos Varela
Rennselaer Polytechnic Institute

September 27, 2016

Other programming languages

Imperative

Algol (Naur 1958)
Cobol (Hopper 1959)
BASIC (Kennedy and Kurtz 1964)
Pascal (Wirth 1970)
C (Kernighan and Ritchie 1971)
Ada (Whitaker 1979)

Functional

ML (Milner 1973)
Scheme (Sussman and Steele 1975)
Haskell (Hughes et al 1987)

Object-Oriented

Smalltalk (Kay 1980)
C++ (Stroustrop 1980)
Eiffel (Meyer 1985)
Java (Gosling 1994)
C# (Hejlsberg 2000)

Actor-Oriented

Act (Lieberman 1981)
ABCL (Yonezawa 1988)
Actalk (Briot 1989)
Erlang (Armstrong 1990)
E (Miller et al 1998)
SALSA (Varela and Agha 1999)

Scripting

Python (van Rossum 1985)
Perl (Wall 1987)
Tcl (Ousterhout 1988)
Lua (Ierusalimschy et al 1994)
JavaScript (Eich 1995)
PHP (Lerdorf 1995)
Ruby (Matsumoto 1995)

Language syntax

- Defines what are the legal programs, i.e. programs that can be executed by a machine (interpreter)
- Syntax is defined by grammar rules
- A grammar defines how to make 'sentences' out of 'words'
- For programming languages: sentences are called statements (commands, expressions)
- For programming languages: words are called tokens
- Grammar rules are used to describe both tokens and statements

Language Semantics

- Semantics defines what a program does when it executes
- Semantics should be simple and yet allows reasoning about programs (correctness, execution time, and memory use)

Lambda Calculus Syntax and Semantics

The syntax of a λ-calculus expression is as follows:

e	::=	**v**	variable
	|	***λ*v.e**	functional abstraction
	|	***(*e e*)***	function application

The semantics of a λ-calculus expression is called beta-reduction:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

where we alpha-rename the lambda abstraction **E** if necessary to avoid capturing free variables in **M**.

α-renaming

Alpha renaming is used to prevent capturing free occurrences of variables when beta-reducing a lambda calculus expression.

In the following, we rename ***x*** to ***z***, (or any other *fresh* variable):

$$(\lambda x.(y\ x)\ x)$$

$$\xrightarrow{\alpha} \quad (\lambda z.(y\ z)\ x)$$

Only *bound* variables can be renamed. No *free* variables can be captured (become bound) in the process. For example, we *cannot* alpha-rename ***x*** to ***y***.

β-reduction

$$(\lambda x.E\ M) \xrightarrow{\beta} E\{M/x\}$$

Beta-reduction may require alpha renaming to prevent capturing free variable occurrences. For example:

$$(\lambda x.\lambda y.(x\ y)\ (y\ w))$$

$$\xrightarrow{\alpha} (\lambda x.\lambda z.(x\ z)\ (y\ w))$$

$$\xrightarrow{\beta} \lambda z.((y\ w)\ z)$$

Where the *free* ***y*** remains free.

η-conversion

$$\lambda x.(E\ x) \xrightarrow{\eta} E$$

if ***x*** is *not* free in ***E***.

For example:

$$(\lambda x.\lambda y.(x\ y)\ (y\ w))$$

$$\xrightarrow{\alpha} (\lambda x.\lambda z.(x\ z)\ (y\ w))$$

$$\xrightarrow{\beta} \lambda z.((y\ w)\ z)$$

$$\xrightarrow{\eta} (y\ w)$$

Currying

The lambda calculus can only represent functions of *one* variable. It turns out that one-variable functions are sufficient to represent multiple-variable functions, using a strategy called *currying*.

E.g., given the mathematical function: $h(x,y) = x+y$
of type $h: Z \times Z \rightarrow Z$

We can represent h as h' of type: $h': Z \rightarrow Z \rightarrow Z$
Such that

$$h(x,y) = h'(x)(y) = x+y$$

For example,

$$h'(2) = g, \text{ where } g(y) = 2+y$$

We say that h' is the *curried* version of h.

Function Composition in Lambda Calculus

S:	$\lambda x.(s\ x)$	(Square)
I:	$\lambda x.(i\ x)$	(Increment)
C:	$\lambda f.\lambda g.\lambda x.(f\ (g\ x))$	(Function Composition)

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

((C S) I)

$$((\lambda f.\lambda g.\lambda x.(f\ (g\ x))\ \lambda x.(s\ x))\ \lambda x.(i\ x))$$

$$\Rightarrow (\lambda g.\lambda x.(\lambda x.(s\ x)\ (g\ x))\ \lambda x.(i\ x))$$

$$\Rightarrow \lambda x.(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))$$

$$\Rightarrow \lambda x.(\lambda x.(s\ x)\ (i\ x))$$

$$\Rightarrow \lambda x.(s\ (i\ x))$$

Order of Evaluation in the Lambda Calculus

Does the order of evaluation change the final result?
Consider:

$$\lambda x.(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))$$

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

There are two possible evaluation orders:

$$\lambda x.(\lambda x.(s\ x)\ \underline{(\lambda x.(i\ x)\ x)})$$
$$\Rightarrow \lambda x.\underline{(\lambda x.(s\ x)\ (i\ x))}$$
$$\Rightarrow \lambda x.(s\ (i\ x))$$

Applicative Order

and:

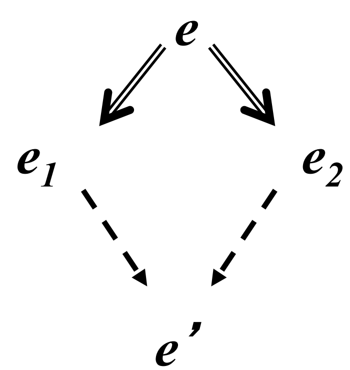
$$\lambda x.\underline{(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))}$$
$$\Rightarrow \lambda x.(s\ \underline{(\lambda x.(i\ x)\ x)})$$
$$\Rightarrow \lambda x.(s\ (i\ x))$$

Normal Order

Is the final result always the same?

Church-Rosser Theorem

If a lambda calculus expression can be evaluated in two different ways and both ways terminate, both ways will yield the same result.



Also called the *diamond* or *confluence* property.

Furthermore, if there is a way for an expression evaluation to terminate, using normal order will cause termination.

Order of Evaluation and Termination

Consider:

$$(\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

There are two possible evaluation orders:

$$(\lambda x.y\ \underline{(\lambda x.(x\ x)\ \lambda x.(x\ x))})$$
$$\Rightarrow (\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

Applicative Order

and:

$$\underline{(\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))}$$
$$\Rightarrow y$$

Normal Order

In this example, normal order terminates whereas applicative order does not.

Free and Bound Variables

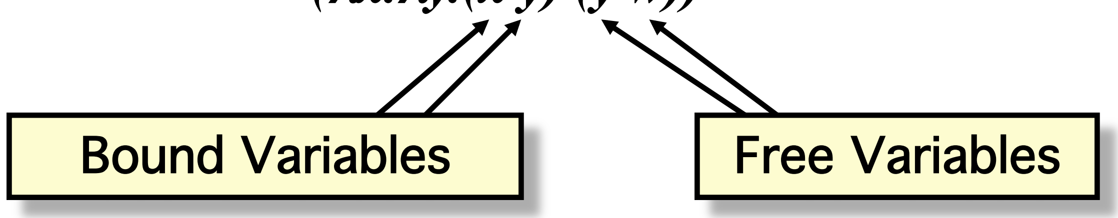
The lambda functional abstraction is the only syntactic construct that *binds* variables. That is, in an expression of the form:

$$\lambda \mathbf{v}.\mathbf{e}$$

we say that free occurrences of variable **v** in expression **e** are *bound*. All other variable occurrences are said to be *free*.

E.g.,

$$(\lambda x.\lambda y.(x\ y)\ (y\ w))$$



Combinators

A lambda calculus expression with *no free variables* is called a *combinator*. For example:

I:	$\lambda x.x$	(Identity)
App:	$\lambda f.\lambda x.(f\ x)$	(Application)
C:	$\lambda f.\lambda g.\lambda x.(f\ (g\ x))$	(Composition)
L:	$(\lambda x.(x\ x)\ \lambda x.(x\ x))$	(Loop)
Cur:	$\lambda f.\lambda x.\lambda y.((f\ x)\ y)$	(Currying)
Seq:	$\lambda x.\lambda y.(\lambda z.y\ x)$	(Sequencing--normal order)
ASeq:	$\lambda x.\lambda y.(y\ x)$	(Sequencing--applicative order)

where ***y*** denotes a *thunk, i.e.*, a lambda abstraction wrapping the second expression to evaluate.

The meaning of a combinator is always the same independently of its context.

Currying Combinator in Oz

The currying combinator can be written in Oz as follows:

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

It takes a function of two arguments, F, and returns its curried version, e.g.,

{{{Curry Plus} 2} 3} ⇒ 5

Recursion Combinator (Y or *rec*)

X can be defined as ***(Y f)***, where ***Y*** is the *recursion combinator*.

Y: $\lambda f.(\lambda x.(f\ \lambda y.((x\ x)\ y))\ \lambda x.(f\ \lambda y.((x\ x)\ y)))$ — Applicative Order

Y: $\lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$ — Normal Order

You get from the normal order to the applicative order recursion combinator by η-expansion (η-conversion from right to left).

Natural Numbers in Lambda Calculus

$\|0\|$:	$\lambda x.x$	(Zero)
$\|1\|$:	$\lambda x.\lambda x.x$	(One)
…		
$\|n+1\|$:	$\lambda x.\|n\|$	(N+1)
s:	$\lambda n.\lambda x.n$	(Successor)

$$(s\ 0)$$

$$\underline{(\lambda n.\lambda x.n\ \lambda x.x)}$$

$$\Rightarrow \lambda x.\lambda x.x$$

> ***Recall semantics rule:***
>
> $(\lambda x.E\ M) \Rightarrow E\{M/x\}$

Booleans and Branching (*if*) in λ Calculus

|*true*|:	$\lambda x.\lambda y.x$	(True)
|*false*|:	$\lambda x.\lambda y.y$	(False)
|*if*|:	$\lambda b.\lambda t.\lambda e.((b\ t)\ e)$	(If)

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

$$(((if\ true)\ a)\ b)$$

$$(((\underline{\lambda b.\lambda t.\lambda e.((b\ t)\ e)\ \lambda x.\lambda y.x})\ a)\ b)$$
$$\Rightarrow ((\underline{\lambda t.\lambda e.((\lambda x.\lambda y.x\ t)\ e)\ a})\ b)$$
$$\Rightarrow \underline{(\lambda e.((\lambda x.\lambda y.x\ a)\ e)\ b)}$$
$$\Rightarrow ((\underline{\lambda x.\lambda y.x\ a})\ b)$$
$$\Rightarrow \underline{(\lambda y.a\ b)}$$
$$\Rightarrow a$$

Church Numerals

$\|0\|$:	$\lambda f.\lambda x.x$	(Zero)
$\|1\|$:	$\lambda f.\lambda x.(f\ x)$	(One)
…		
$\|n\|$:	$\lambda f.\lambda x.(f \ldots (f\ x)\ldots)$	(N applications of f to x)
s:	$\lambda n.\lambda f.\lambda x.(f\ ((n\ f)\ x))$	(Successor)

Recall semantics rule:

$(\lambda x.E\ M) \Rightarrow E\{M/x\}$

$(s\ 0)$

$$
\begin{aligned}
&(\lambda n.\lambda f.\lambda x.(f\ ((n\ f)\ x))\ \lambda f.\lambda x.x) \\
&\Rightarrow \lambda f.\lambda x.(f\ (\underline{(\lambda f.\lambda x.x\ f)}\ x)) \\
&\Rightarrow \lambda f.\lambda x.(f\ \underline{(\lambda x.x\ x)}) \\
&\Rightarrow \lambda f.\lambda x.(f\ x)
\end{aligned}
$$

Church Numerals: isZero?

Recall semantics rule:

$(\lambda x.E\ M) \Rightarrow E\{M/x\}$

isZero?: $\lambda n.((n\ \lambda x.false)\ true)$ (Is n=0?)

$$(isZero?\ 0)$$
$$(\lambda n.((n\ \lambda x.false)\ true)\ \lambda f.\lambda x.x)$$
$$\Rightarrow ((\lambda f.\lambda x.x\ \lambda x.false)\ true)$$
$$\Rightarrow (\lambda x.x\ true)$$
$$\Rightarrow true$$

$$(isZero?\ 1)$$
$$(\lambda n.((n\ \lambda x.false)\ true)\ \lambda f.\lambda x.(f\ x))$$
$$\Rightarrow ((\lambda f.\lambda x.(f\ x)\ \lambda x.false)\ true)$$
$$\Rightarrow (\lambda x.(\lambda x.false\ x)\ true)$$
$$\Rightarrow (\lambda x.false\ true)$$
$$\Rightarrow false$$

Functions

- Compute the factorial function:
- Start with the mathematical definition

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

- Fact is declared in the environment
- Try large factorial {Browse {Fact 100}}

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

$$0! = 1$$

$$n! = n \times (n-1)! \text{ if } n > 0$$

Factorial in Haskell

```
factorial :: Integer -> Integer
factorial  0              = 1
factorial  n | n > 0      = n * factorial (n-1)
```

Structured data (lists)

- Calculate Pascal triangle
- Write a function that calculates the nth row as one structured value
- A list is a sequence of elements:

  [1 4 6 4 1]
- The empty list is written nil
- Lists are created by means of "|" (cons)

```
declare
H=1
T = [2 3 4 5]
{Browse H|T}  % This will show [1 2 3 4 5]
```

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Pattern matching

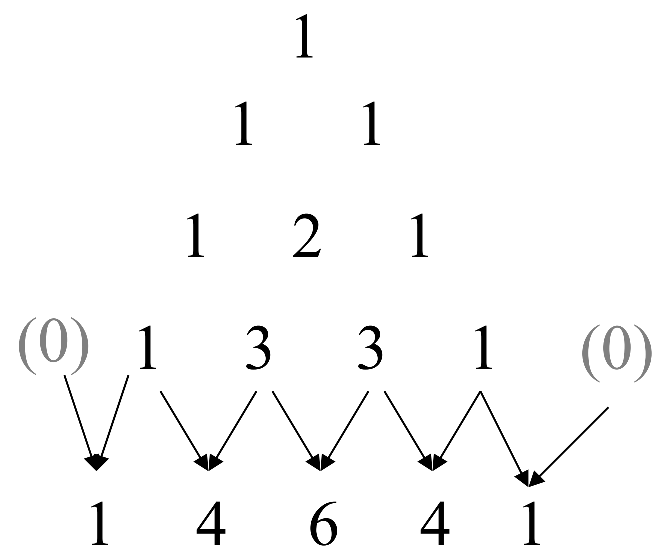
- Another way to take a list apart is by use of pattern matching with a case instruction

```
case L of H|T then {Browse H} {Browse T}
          else {Browse ‘empty list’}
end
```

Functions over lists

- Compute the function {Pascal N}
- Takes an integer N, and returns the Nth row of a Pascal triangle as a list

1. For row 1, the result is [1]
2. For row N, shift to left row N-1 and shift to the right row N-1
3. Align and add the shifted rows element-wise to get row N

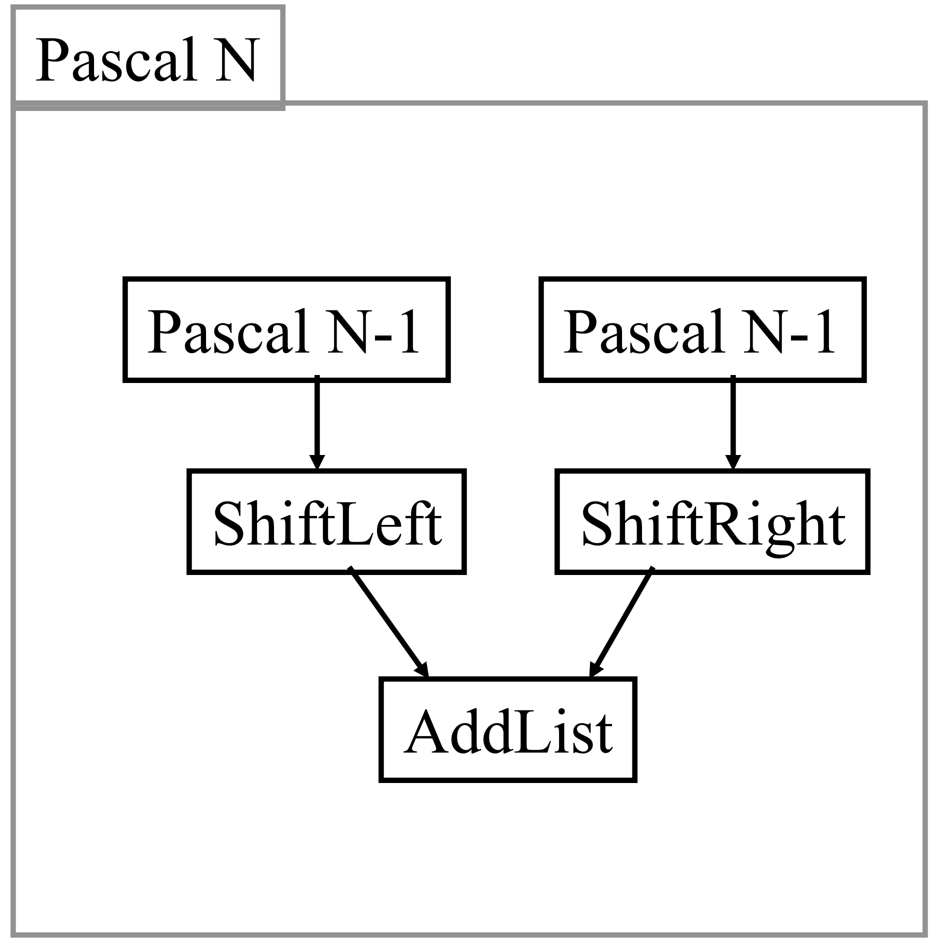


Shift right [0 1 3 3 1]

Shift left [1 3 3 1 0]

Functions over lists

```
declare
fun {Pascal N}
  if N==1 then [1]
  else
    {AddList
     {ShiftLeft {Pascal N-1}}
     {ShiftRight {Pascal N-1}}}
  end
end
```



Functions over lists (2)

```
fun {ShiftLeft L}
  case L of H|T then
    H|{ShiftLeft T}
  else [0]
  end
end

fun {ShiftRight L}  0|L end
```

```
fun {AddList L1 L2}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      H1+H2|{AddList T1 T2}
    end
  else nil end
end
```

Pattern matching in Haskell

- Another way to take a list apart is by use of pattern matching with a case instruction:

```
case l of (h:t) -> h:t
          []    -> []
end
```

- Or more typically as part of a function definition:

```
id (h:t) -> h:t
id []    -> []
```

Functions over lists in Haskell

```
--- Pascal triangle row
pascal :: Integer -> [Integer]
pascal 1 = [1]
pascal n = addList (shiftLeft (pascal (n-1)))
                   (shiftRight (pascal (n-1)))
  where
    shiftLeft []     = [0]
    shiftLeft (h:t) = h:shiftLeft t
    shiftRight l     = 0:l
    addList [] []   = []
    addList (h1:t1) (h2:t2) = (h1+h2):addList t1 t2
```

Mathematical induction

- Select one or more inputs to the function
- Show the program is correct for the *simple cases* (base cases)
- Show that if the program is correct for a *given case*, it is then correct for the *next case*.
- For natural numbers, the base case is either 0 or 1, and for any number n the next case is n+1
- For lists, the base case is nil, or a list with one or a few elements, and for any list T the next case is H|T

Correctness of factorial

```
fun {Fact N}
  if N==0 then 1 else N*{Fact N-1} end
end
```

$$\underbrace{1 \times 2 \times \cdots \times (n-1)}_{Fact(n-1)} \times n$$

- Base Case N=0: {Fact 0} returns 1
- Inductive Case N>0: {Fact N} returns N*{Fact N-1} assume {Fact N-1} is correct, from the spec we see that {Fact N} is N*{Fact N-1}

Iterative computation

- An iterative computation is one whose execution stack is bounded by a constant, independent of the length of the computation
- Iterative computation starts with an initial state S_0, and transforms the state in a number of steps until a final state S_{final} is reached:

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{final}$$

The general scheme

fun {Iterate S_i}
    if {*IsDone* S_i} then S_i
    else S_{i+1} in
        S_{i+1} = {*Transform* S_i}
        {Iterate S_{i+1}}
    end
end

- *IsDone* and *Transform* are problem dependent

From a general scheme to a control abstraction (2)

```
fun {Iterate S IsDone Transform}
    if {IsDone S} then S
    else S1 in
        S1 = {Transform S}
        {Iterate S1 IsDone Transform}
    end
end
```

fun {Iterate S_i}
    if {*IsDone* S_i} then S_i
    else S_{i+1} in
        S_{i+1} = {*Transform* S_i}
        {Iterate S_{i+1}}
    end
end

Sqrt using the control abstraction

```
fun {Sqrt X}
   {Iterate
     1.0
     fun {$ G} {Abs X - G*G}/X < 0.000001 end
     fun {$ G} (G + X/G)/2.0 end
   }
end
```

Iterate could become a linguistic abstraction

Sqrt in Haskell

```
let sqrt x = head (dropWhile (not . goodEnough) sqrtGuesses)
    where
      goodEnough guess = (abs (x – guess*guess))/x < 0.00001
      improve guess = (guess + x/guess)/2.0
      sqrtGuesses = 1:(map improve sqrtGuesses)
```

This sqrt example uses infinite lists enabled by lazy evaluation, and the map control abstraction.

Higher-order programming

- Higher-order programming = the set of programming techniques that are possible with procedure values (lexically-scoped closures)
- Basic operations
  - Procedural abstraction: creating procedure values with lexical scoping
  - Genericity: procedure values as arguments
  - Instantiation: procedure values as return values
  - Embedding: procedure values in data structures
- Higher-order programming is the foundation of component-based programming and object-oriented programming

Procedural abstraction

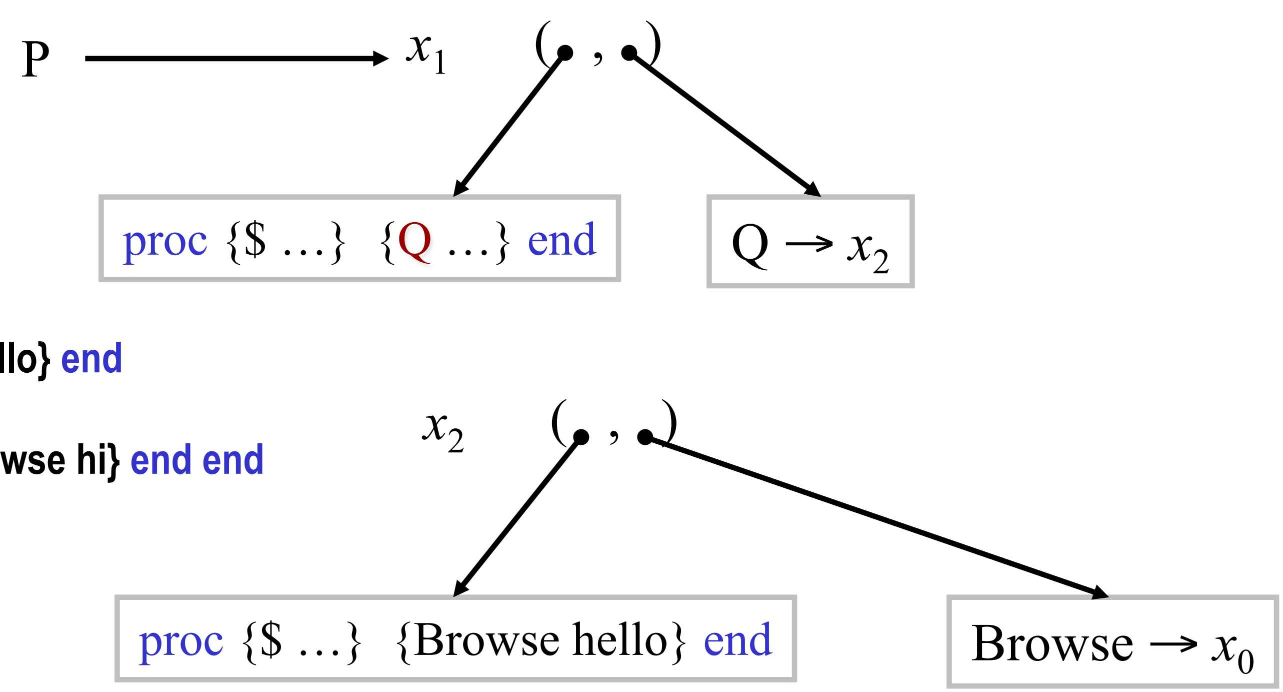
- Procedural abstraction is the ability to convert any statement into a procedure value
  - A procedure value is usually called a closure, or more precisely, a lexically-scoped closure
  - A procedure value is a pair: it combines the procedure code with the environment where the procedure was created (the contextual environment)
- Basic scheme:
  - Consider any statement <s>
  - Convert it into a procedure value: P = proc {$} <s> end
  - Executing {P} has exactly the same effect as executing <s>

Procedure values

- Constructing a procedure value in the store is not simple because a procedure may have external references

```
local P Q in
    P = proc {$ …}  {Q …} end
    Q = proc {$ …} {Browse hello} end
    local Q in
        Q = proc {$ …} {Browse hi} end
        {P …}
    end
end
```

Procedure values (2)




```
local P Q in
    P = proc {$ …} {Q …} end
    Q = proc {$ …} {Browse hello} end
    local Q in
        Q = proc {$ …} {Browse hi} end end
        {P …}
    end
end
```

Genericity

- Replace specific entities (zero 0 and addition +) by function arguments
- The same routine can do the sum, the product, the logical or, etc.

```
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
```

Genericity in Haskell

- Replace specific entities (zero 0 and addition +) by function arguments
- The same routine can do the sum, the product, the logical or, etc.

```
sumlist :: (Num a) => [a] -> a
sumlist []    = 0
sumlist (h:t) = h+sumlist t
```

⇩

```
foldr' :: (a->b->b) -> b -> [a] -> b
foldr' _ u []    = u
foldr' f u (h:t) = f h (foldr' f u t)
```

Instantiation

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

- Instantiation is when a procedure returns a procedure value as its result
- Calling {FoldFactory fun {$ A B} A+B end 0} returns a function that behaves identically to SumList, which is an « instance » of a folding function

Embedding

- Embedding is when procedure values are put in data structures
- Embedding has many uses:
  - Modules: a module is a record that groups together a set of related operations
  - Software components: a software component is a generic function that takes a set of modules as its arguments and returns a new module. It can be seen as specifying a module in terms of the modules it needs.
  - Delayed evaluation (also called explicit lazy evaluation): build just a small part of a data structure, with functions at the extremities that can be called to build more. The consumer can control explicitly how much of the data structure is built.

Control Abstractions

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

What does this program do ?

```
{Browse {FoldL [1 2 3]
     fun {$ X Y} X|Y end nil}}
```

FoldL in Haskell

```
foldl' :: (b->a->b) -> b -> [a] -> b
foldl' _ u []    = u
foldl' f u (h:t) = foldl' f (f u h) t
```

Notice the unit u is of type b, and the function f is of type b->a->b.

List-based techniques

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

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

Map in Haskell

```
map' :: (a -> b) -> [a] -> [b]
map' _ []    = []
map' f (h:t) = f h:map' f t
```

_ means that the argument is not used (read "don't care").
map' is to distinguish it from the Prelude map function.

Filter in Haskell

```
filter' :: (a-> Bool) -> [a] -> [a]
filter' _ []    = []
filter' p (h:t) = if p h then h:filter' p t
                         else filter' p t
```

Filter as FoldR application

```
fun {Filter L P}
   {FoldR L fun {$ H T}
                 if {P H} then
                       H|T
                 else T end
              end nil}
end
```

```
filter'' :: (a-> Bool) -> [a] -> [a]
filter'' p l = foldr
          (\h t ->  if p h
                    then h:t
                    else t) [] l
```

Lazy evaluation

- The functions written so far are evaluated eagerly (as soon as they are called)
- Another way is lazy evaluation where a computation is done only when the results is needed

- Calculates the infinite list:
  0 | 1 | 2 | 3 | ...

```
declare
fun lazy {Ints N}
  N|{Ints N+1}
end
```

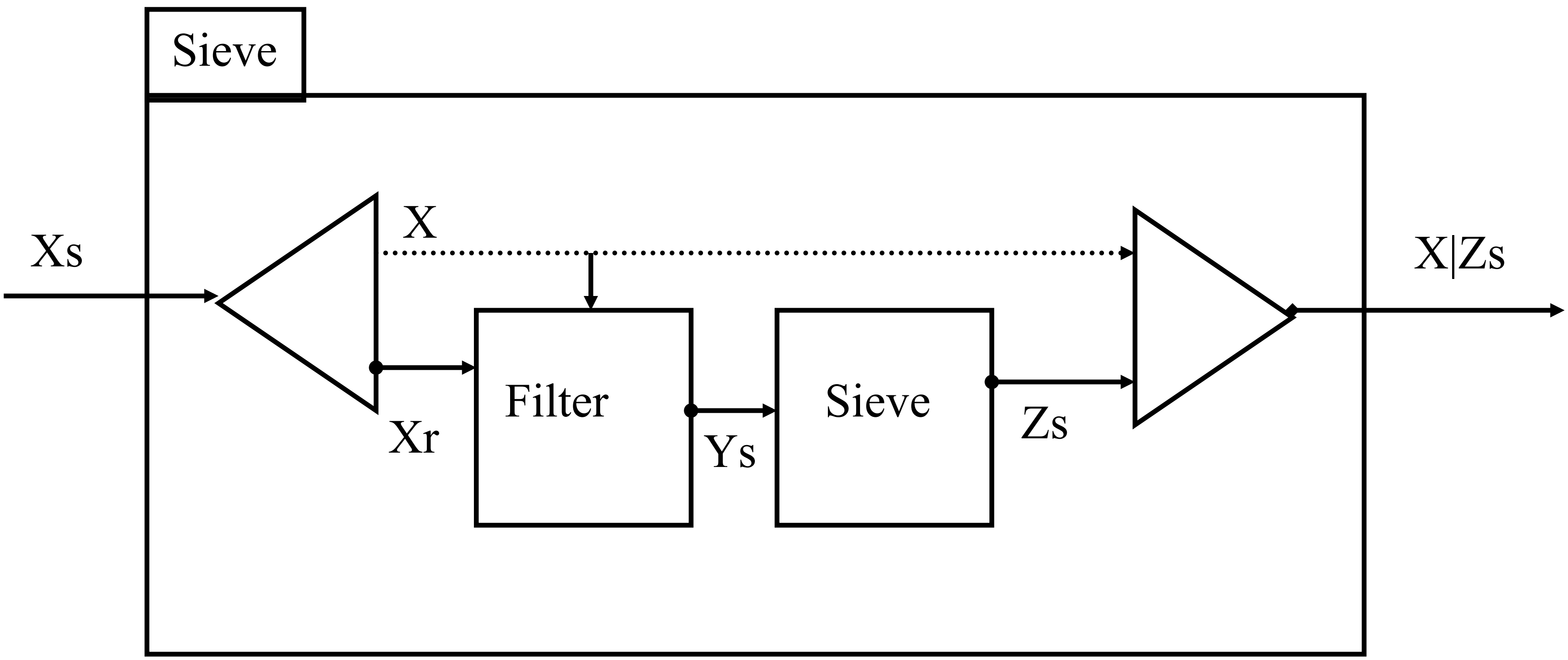
Lazy evaluation (2)

- Write a function that computes as many rows of Pascal's triangle as needed
- We do not know how many beforehand
- A function is *lazy* if it is evaluated only when its result is needed
- The function PascalList is evaluated when needed

```
fun lazy {PascalList Row}
    Row | {PascalList
              {AddList
          {ShiftLeft Row}
          {ShiftRight Row}}}
        end
```

Larger Example: The Sieve of Eratosthenes

- Produces prime numbers
- It takes a stream 2...N, peals off 2 from the rest of the stream
- Delivers the rest to the next sieve



Lazy Sieve

```
fun lazy {Sieve Xs}
   X|Xr = Xs in
   X | {Sieve {LFilter
         Xr
         fun {$ Y} Y mod X \= 0 end
        }}
end

fun {Primes} {Sieve {Ints 2}} end
```

Lazy Filter

For the Sieve program we need a lazy filter

```
fun lazy {LFilter Xs F}
   case Xs
   of nil then nil
   [] X|Xr then
      if {F X} then X|{LFilter Xr F} else {LFilter Xr F} end
   end
end
```

Primes in Haskell

```
ints :: (Num a) => a -> [a]
ints n = n : ints (n+1)

sieve :: (Integral a) => [a] -> [a]
sieve (x:xr) = x:sieve (filter (\y -> (y `mod` x /= 0)) xr)

primes :: (Integral a) => [a]
primes = sieve (ints 2)
```

Functions in Haskell are lazy by default. You can use `take 20 primes` to get the first 20 elements of the list.

List Comprehensions

- Abstraction provided in lazy functional languages that allows writing higher level set-like expressions
- In our context we produce lazy lists instead of sets
- The mathematical set expression
  - $\{x*y \mid 1 \leq x \leq 10,\ 1 \leq y \leq x\}$
- Equivalent List comprehension expression is
  - [X*Y | X = 1..10 ; Y = 1..X]
- Example:
  - [1*1 2*1 2*2 3*1 3*2 3*3 ... 10*10]

List Comprehensions

- The general form is
- [f(x,y, ...,z) | x ← gen(a1,...,an) ; guard(x,...)

  y ← gen(x, a1,...,an) ; guard(y,x,...)

  ....

  ]
- No linguistic support in Mozart/Oz, but can be easily expressed

Example 1

- z = [x#x | x ← from(1,10)]
- Z = {LMap {LFrom 1 10} fun{$ X} X#X end}

- z = [x#y | x ← from(1,10), y ← from(1,x)]
- Z = {LFlatten

```
Z = {LFlatten
       {LMap {LFrom 1 10}
        fun{$ X} {LMap {LFrom 1 X}
                    fun {$ Y} X#Y end
                   }
        end
       }
     }
```

Example 2

- z = [x#y | **x** ← from(1,10), **y** ← from(1,**x**), **x**+**y**≤10]
- Z **={LFilter**

```
        {LFlatten
           {LMap {LFrom 1 10}
            fun{$ X} {LMap {LFrom 1 X}
                          fun {$ Y} X#Y end
                         }
            end
           }
        }
        fun {$ X#Y} X+Y=<10 end} }
```

List Comprehensions in Haskell

```
lc1 = [(x,y) | x <- [1..10], y <- [1..x]]

lc2 = filter (\(x,y)->(x+y<=10)) lc1

lc3 = [(x,y) | x <- [1..10], y <- [1..x], x+y<= 10]
```

Haskell provides syntactic support for list comprehensions. List comprehensions are implemented using a built-in list monad.

Quicksort using list comprehensions

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []    = []
quicksort (h:t) = quicksort [x | x <- t, x < h] ++
                  [h] ++
                  quicksort [x | x <-t, x >= h]
```

Types of typing

- Languages can be ***weakly typed***
  - Internal representation of types can be manipulated by a program
    - e.g., a string in C is an array of characters ending in '\0'.

- ***Strongly typed*** programming languages can be further subdivided into:
  - ***Dynamically typed*** languages
    - Variables can be bound to entities of any type, so in general the type is only known at **run-time**, e.g., Oz, SALSA.
  - ***Statically typed*** languages
    - Variable types are known at **compile-time**, e.g., C++, Java.

Type Checking and Inference

- *Type checking* is the process of ensuring a program is well-typed.
  - One strategy often used is *abstract interpretation:*
    - The principle of getting partial information about the answers from partial information about the inputs
    - Programmer supplies types of variables and type-checker deduces types of other expressions for consistency

- *Type inference* frees programmers from annotating variable types: types are inferred from variable usage, e.g. ML, Haskell.

Abstract data types

- A datatype is a set of values and an associated set of operations
- A datatype is abstract only if it is completely described by its set of operations regardless of its implementation
- This means that it is possible to change the implementation of the datatype without changing its use
- The datatype is thus described by a set of procedures
- These operations are the only thing that a user of the abstraction can assume

Example: A Stack

- Assume we want to define a new datatype ⟨stack T⟩ whose elements are of any type T

  fun {NewStack}: ⟨Stack T⟩

  fun {Push ⟨Stack T⟩ ⟨T⟩ }: ⟨Stack T⟩

  fun {Pop ⟨Stack T⟩ ⟨T⟩ }: ⟨Stack T⟩

  fun {IsEmpty ⟨Stack T⟩ }: ⟨Bool⟩

- These operations normally satisfy certain laws:

  {IsEmpty {NewStack}} = true

  for any *E* and *S0*, *S1*={Push *S0 E*} and *S0* ={Pop *S1 E*} hold

  {Pop {NewStack} E} raises error

Stack (another implementation)

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of  X|S1 then E = X  S1 end end
fun {IsEmpty S} S==nil end
```

```
fun {NewStack} emptyStack end
fun {Push S E} stack(E S) end
fun {Pop S E} case S of stack(X S1) then E = X S1 end end
fun {IsEmpty S} S==emptyStack end
```

Stack data type in Haskell

```
data Stack a  = Empty | Stack a (Stack a)

newStack :: Stack a
newStack = Empty
push :: Stack a -> a -> Stack a
push s e = Stack e s
pop :: Stack a -> (Stack a,a)
pop (Stack e s) = (s,e)
isempty :: Stack a -> Bool
isempty Empty = True
isempty (Stack _ _) = False
```

Secure abstract data types: A secure stack

With the wrapper & unwrapper we can build a secure stack

```
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push S E} {Wrap E|{Unwrap S}} end
   fun {Pop S E}
        case {Unwrap S} of X|S1 then
             E=X  {Wrap S1} end
   end
   fun {IsEmpty S} {Unwrap S}==nil end
end
```

```
proc {NewWrapper
        ?Wrap ?Unwrap}
   Key={NewName}
in
   fun {Wrap X}
      fun {$ K}
        if K==Key then X end
      end
   end
   fun {Unwrap C}
      {C Key}
   end
end
```

Stack abstract data type as a module in Haskell

```
module StackADT (Stack,newStack,push,pop,isEmpty) where

data Stack a  = Empty | Stack a (Stack a)
newStack = Empty
…
```

- Modules can then be imported by other modules, e.g.:

```
module Main (main) where
import StackADT ( Stack, newStack,push,pop,isEmpty )

main = do print (push (push newStack 1) 2)
```

Declarative operations (1)

- An operation is *declarative* if whenever it is called with the same arguments, it returns the same results independent of any other computation state
- A declarative operation is:
  - *Independent* (depends only on its arguments, nothing else)
  - *Stateless* (no internal state is remembered between calls)
  - *Deterministic* (call with same operations always give same results)
- Declarative operations can be composed together to yield other declarative components
  - All basic operations of the declarative model are declarative and combining them always gives declarative components

Why declarative components (1)

- There are two reasons why they are important:
- *(Programming in the large)* A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
  - The complexity (reasoning complexity) of a program composed of declarative components is the *sum* of the complexity of the components
  - In general the reasoning complexity of programs that are composed of nondeclarative components explodes because of the intimate interaction between components
- *(Programming in the small)* Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).
  - Simple algebraic and logical reasoning techniques can be used

Monads

- Purely functional programming is declarative in nature: whenever a function is called with the same arguments, it returns the same results independent of any other computation state.
- How to model the real world (that may have context dependences, state, nondeterminism) in a purely functional programming language?
  - Context dependences: e.g., does file exist in expected directory?
  - State: e.g., is there money in the bank account?
  - Nondeterminism: e.g., does bank account deposit happen before or after interest accrual?
- Monads to the rescue!

Monad class

- The Monad class defines two basic operations:

```
class Monad m where
    (>>=)      ::  m a -> (a -> m b) -> m b   -- bind
    return     ::  a -> m a
    fail       ::  String -> m a
    m >> k     =   m >>= \_ -> k
```

- The `>>=` infix operation binds two monadic values, while the `return` operation injects a value into the monad (container).
- Example monadic classes are `IO`, lists (`[]`) and `Maybe`.

do syntactic sugar

- In the IO class, x >>= y, performs two actions sequentially (like the Seq combinator in the lambda-calculus) passing the result of the first into the second.
- Chains of monadic operations can use do:

```
do e1 ; e2          =       e1 >> e2
do p <- e1; e2      =       e1 >>= \p -> e2
```

- Pattern match can fail, so the full translation is:

```
do p <- e1; e2      =       e1 >>= (\v -> case of p -> e2
                                                  _ -> fail “s”)
```

- Failure in IO monad produces an error, whereas failure in the List monad produces the empty list.

Monad class laws

- All instances of the Monad class should respect the following laws:

```
return a >>= k                = k a
m >>= return                  = m
xs >>= return . f             = fmap f xs
m >>= (\x -> k x >>= h)       = (m >>= k) >>= h
```

- These laws ensure that we can bind together monadic values with >>= and inject values into the monad (container) using return in consistent ways.
- The MonadPlus class includes an mzero element and an mplus operation. For lists, mzero is the empty list ([]), and the mplus operation is list concatenation (++).

List comprehensions with monads

```
lc1 = [(x,y) | x <- [1..10], y <- [1..x]]

lc1' = do x <- [1..10]
          y <- [1..x]
          return (x,y)

lc1" = [1..10] >>=  (\x ->
                     [1..x] >>= (\y ->
                                 return (x,y)))
```

List comprehensions are implemented using a built-in list monad. Binding (l >>= f) applies the function f to all the elements of the list l and concatenates the results. The return function creates a singleton list.

List comprehensions with monads (2)

```
lc3 = [(x,y) | x <- [1..10], y <- [1..x], x+y<= 10]
lc3' = do x <- [1..10]
          y <- [1..x]
          True <- return (x+y<=10)
          return (x,y)

lc3'' = [1..10] >>=  (\x ->
            [1..x] >>= (\y ->
               return (x+y<=10) >>=
                  (\b -> case b of True -> return (x,y); _ -> fail "")))
```

Guards in list comprehensions assume that fail in the List monad returns an empty list.

Monads summary

- Monads enable keeping track of imperative features (state) in a way that is modular with purely functional components.
  - For example, fib remains functional, yet the R monad enables us to keep a count of instructions separately.
- Input/output, list comprehensions, and optional values (Maybe class) are built-in monads in Haskell.
- Monads are useful to modularly define semantics of domain-specific languages.