We will cover theoretical and practical aspects of three different programming paradigms:

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Theory</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Programming</td>
<td>Lambda Calculus</td>
<td>Oz, Haskell</td>
</tr>
<tr>
<td>Concurrent Programming</td>
<td>Actor Model</td>
<td>SALSA, Erlang</td>
</tr>
<tr>
<td>Logic Programming</td>
<td>First-Order Logic Horn Clauses</td>
<td>Prolog, Oz</td>
</tr>
</tbody>
</table>

Each paradigm will be evaluated with a Programming Assignment (PA) and an Exam.

- Two highest PA grades count for 40% of total grade. Lowest PA grade counts for 10% of the total grade.
- Two highest Exam grades count for 40% of total grade. Lowest Exam grade counts for 10% of the total grade.
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)
- Go (Griesemer, Pike, Thompson 2009)

**Object-Oriented**
- Smalltalk (Kay 1980)
- C++ (Stroustrup 1980)
- Eiffel (Meyer 1985)
- Java (Gosling 1994)
- C# (Hejlsberg 2000)
- Scala (Odersky et al 2004)
- Swift (Lattner 2014)

**Actor-Oriented**
- Act (Lieberman 1981)
- ABCL (Yonezawa 1988)
- Actalk (Briot 1989)
- Erlang (Armstrong 1990)
- E (Miller et al 1998)
- SALSA (Varela and Agha 1999)
- Elixir (Valim 2011)

**Functional**
- ML (Milner 1973)
- Scheme (Sussman and Steele 1975)
- Haskell (Hughes et al 1987)
- Clojure (Hickey 2007)

**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 allow reasoning about programs (correctness, execution time, and memory use)
Lambda Calculus Syntax and Semantics

The syntax of a \( \lambda \)-calculus expression is as follows:

\[
\begin{align*}
e \ ::= & \ v \quad \text{variable} \\
& \lambda v.e \quad \text{functional abstraction} \\
& (e \ e) \quad \text{function application}
\end{align*}
\]

The semantics of a \( \lambda \)-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):

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

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

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

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

\[ (\lambda x. \lambda y. (x \ y) \ (y \ w)) \]

\[ \alpha \rightarrow (\lambda x. \lambda z. (x \ z) \ (y \ w)) \]

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

Where the free \( y \) remains free.
η-conversion

\[ \lambda x. (E \ x) \xrightarrow{\eta} E \]

if \( x \) is \textit{not} free in \( E \).

For example:

\[
(\lambda x. \lambda y. (x \ y) \ (y \ w))
\]

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

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

\[
\xrightarrow{\eta} (y \ w)
\]
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 \( h: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \)

We can represent \( h \) as \( h' \) of type: \( h': \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \)

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:

\[
\lambda x. E \, M \Rightarrow E\{M/x\}
\]
Order of Evaluation in the Lambda Calculus

Does the order of evaluation change the final result?

Consider:

\[ \lambda x. (\lambda x. (s x) (\lambda x. (i x) x)) \]

There are two possible evaluation orders:

1. \[ \lambda x. (\lambda x. (s x) (\lambda x. (i x) x)) \]
   \[ \Rightarrow \lambda x. (\lambda x. (s x) (i x)) \]
   \[ \Rightarrow \lambda x. (s (i x)) \]

2. \[ \lambda x. (\lambda x. (s x) (\lambda x. (i x) x)) \]
   \[ \Rightarrow \lambda x. (s (\lambda x. (i x) x)) \]
   \[ \Rightarrow \lambda x. (s (i x)) \]

Is the final result always the same?

Recall semantics rule:

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

Applicative Order

Normal Order

C. Varela
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 \textit{diamond} or \textit{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:

\[(\lambda x. y (\lambda x. (x x) \lambda x. (x x)))\]

There are two possible evaluation orders:

\[(\lambda x. y (\lambda x. (x x) \lambda x. (x x)))\]

\[\Rightarrow (\lambda x. y (\lambda x. (x x) \lambda x. (x x)))\]

and:

\[(\lambda x. y (\lambda x. (x x) \lambda x. (x x))}\]

\[\Rightarrow y\]

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

Recall semantics rule:

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

Applicative Order

Normal Order

C. Varela
Free and Bound Variables

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

\[ \lambda v. e \]

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

E.g.,

\[ (\lambda x. \lambda y. (x y) (y w)) \]
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 \((Yf)\), where \(Y\) is the recursion combinator.

\[
Y: \quad \lambda f. (\lambda x. (f \lambda y. ((x x) y)))
\]

Applicative Order

\[
Y: \quad \lambda f. (\lambda x. (f (x x)))
\]

Normal Order

You get from the normal order to the applicative order recursion combinator by \(\eta\)-expansion (\(\eta\)-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)

Recall semantics rule:

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

\[
(s \ 0)
\]

\[
(\lambda n.\lambda x.n \ \lambda x.x)
\]

\[\Rightarrow \lambda x.\lambda x.x\]
Booleans and Branching (if) in \( \lambda \) 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:

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

\[
(((\lambda b. \lambda t. \lambda e. ((b t) e) \lambda x. \lambda y. x) a) b) \\
\Rightarrow ((\lambda t. \lambda e. ((\lambda x. \lambda y. x t) e) a) b) \\
\Rightarrow (\lambda e. ((\lambda x. \lambda y. x a) e) b) \\
\Rightarrow ((\lambda x. \lambda y. x a) b) \\
\Rightarrow (\lambda y. a b) \\
\Rightarrow a
\]
# 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:  

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

\[
\begin{align*}
(s \ 0) &= \lambda n. \lambda f. \lambda x. (f \ ((n \ f) \ x)) \\
&= \lambda f. \lambda x. (f \ ((\lambda f. \lambda x. f) \ x)) \\
&= \lambda f. \lambda x. (f \ (\lambda x. x \ x)) \\
&= \lambda f. \lambda x. (f \ x)
\end{align*}
\]
Church Numerals: isZero?

Recall semantics rule:

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

\[
isZero? : \quad \lambda n.((n \ \lambda x.\text{false}) \ \text{true}) \quad (\text{Is n=0?})
\]

\[
\begin{align*}
\text{isZero? 0) } & \\
(\lambda n.((n \ \lambda x.\text{false}) \ \text{true}) \ \lambda f.\lambda x.x) & \Rightarrow ((\lambda f.\lambda x.x \ \lambda x.\text{false}) \ \text{true}) \\
& \Rightarrow (\lambda x.x \ \text{true}) \\
& \Rightarrow \text{true}
\end{align*}
\]

\[
\begin{align*}
\text{isZero? 1) } & \\
(\lambda n.((n \ \lambda x.\text{false}) \ \text{true}) \ \lambda f.\lambda x.(f \ x)) & \Rightarrow ((\lambda f.\lambda x.(f \ x) \ \lambda x.\text{false}) \ \text{true}) \\
& \Rightarrow (\lambda x.(\lambda x.\text{false} \ x) \ \text{true}) \\
& \Rightarrow (\lambda x.\text{false} \ \text{true}) \\
& \Rightarrow \text{false}
\end{align*}
\]
Functions

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

\[
\text{declare}
\]
\[
\text{fun} \{\text{Fact N}\}
\]
\[
\quad \text{if } N==0 \text{ then 1 else } N*\{\text{Fact N-1}\} \text{ end}
\]
\[
\text{end}
\]

• Fact is declared in the environment
• Try large factorial \( \{\text{Browse } \{\text{Fact 100}\}\} \)

\[
n! = 1 \times 2 \times \cdots \times (n-1) \times n
\]

\[
0! = 1
\]

\[
n! = n \times (n-1)! \text{ if } n > 0
\]
Factorial in Haskell

\[
\text{factorial} :: \text{Integer} \to \text{Integer}
\]

\[
\text{factorial} \ 0 \quad = \quad 1
\]

\[
\text{factorial} \ n \mid n > 0 \quad = \quad n \times \text{factorial} \ (n-1)
\]
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 \texttt{nil}
- Lists are created by means of \\\	exttt{”|” (cons)}

```declare
H=1
T = [2 3 4 5]
{Browse H|T} \% This will show [1 2 3 4 5]
```
A typical way to take a list apart is by use of pattern matching with a case instruction:

```plaintext
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 (2)

Declare
Fun {Pascal N}
  If N == 1 then [1]
  Else
    {AddList
      {ShiftLeft {Pascal N - 1}}
      {ShiftRight {Pascal N - 1}}}!
  End
End
Functions over lists (3)

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

- A typical 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 as part of a function definition:

```
id (h:t) -> h:t
id []    -> []
```
Functions over lists in Haskell

--- Pascal triangle row

```haskell
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

• 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 $S_{\text{final}}$ is reached:

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{\text{final}}$$
The general scheme

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

• \textit{IsDone} and \textit{Transform} are problem dependent
From a general scheme to a control abstraction

fun {Iterate S \ IsDone S \ Transform} 
  if \ IsDone S \ then \ S 
  else \ S1 \ in 
  S1 = \ Transform S 
  \ {Iterate S1 \ IsDone S1 \ 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
iterate' s isDone transform =
  if isDone s then s
  else let s1 = transform s in
       iterate' s1 isDone transform

sqrt' x = iterate' 1.0 goodEnough improve
  where goodEnough = \g -> (abs (x - g*g))/x < 0.00001
              improve = \g -> (g + x/g)/2.0
Sqrt in Haskell

\[
\text{sqrt } x = \text{head} \left( \text{dropWhile} \left( \text{not} \ . \ \text{goodEnough} \right) \ \text{sqrtGuesses} \right)
\]

where

\[
\text{goodEnough guess} = \frac{\text{abs} \left( x - \text{guess} \times \text{guess} \right)}{x} < 0.00001
\]

\[
\text{improve guess} = \frac{\text{guess} + \frac{x}{\text{guess}}}{2.0}
\]

\[
\text{sqrtGuesses} = 1 : (\text{map} \ \text{improve} \ \text{sqrtGuesses})
\]

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: \(P = \text{proc} \{\$\} <s> \text{ end}\)
  - Executing \(\{P\}\) has exactly the same effect as executing \(<s>\)
• Constructing a procedure value in the store is not simple because a procedure may have external references

\begin{verbatim}
local P Q in
  P = proc {$ \ldots \}$ {Q \ldots} end
  Q = proc {$ \ldots \}$ {Browse hello} end
local Q in
  Q = proc {$ \ldots \}$ {Browse hi} end
  {P \ldots}
end
end
\end{verbatim}
local P Q in
P = proc { $ ... } { Q ... } end
Q = proc { $ ... } { Browse hello } end
local Q in
Q = proc { $ ... } { Browse hi } end end
end
end

P \rightarrow x_1

\begin{array}{rcl}
\text{proc} & \{ \$ \ldots \} & \{ Q \ldots \} \text{ end} \\
\text{Q} & \rightarrow & x_2
\end{array}

\begin{array}{rcl}
\text{proc} & \{ \$ \ldots \} & \{ \text{Browse hello} \} \text{ end} \\
\text{Browse} & \rightarrow & x_0
\end{array}

(\ldots, \ldots)

x_2

(\ldots, \ldots)
### Genericity

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

```plaintext
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.

\[
\text{sumlist} :: (\text{Num a}) \Rightarrow [\text{a}] \rightarrow \text{a} \\
\text{sumlist} [] = 0 \\
\text{sumlist} (h:t) = h + \text{sumlist} t
\]

\[
\text{foldr'} :: (\text{a} \rightarrow \text{b} \rightarrow \text{b}) \rightarrow \text{b} \rightarrow [\text{a}] \rightarrow \text{b} \\
\text{foldr'} \_ \_ [] = \_ \\
\text{foldr'} f \_ (h:t) = f h (\text{foldr'} f \_ t)
\]
Instantiation

- 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
end

What does this program do?
{Browse {FoldL [1 2 3]
  fun {($) X Y} X|Y end nil}}
FoldL in Haskell

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

Notice the unit u is of type b, list elements are of type a, and the function f is of type a->b->b.
Two more folding functions

Given a list \([e_1 \ e_2 \ ... \ e_n]\) and a binary function \(\otimes\), with unit \(U\), the previous folding functions do the following:

\[
(e_1 \otimes ... (e_{n-1} \otimes (e_n \otimes U))...) \quad \text{fold right}
\]
\[
(e_n \otimes ... (e_2 \otimes (e_1 \otimes U))...) \quad \text{fold left}
\]

But there are two other possibilities:

\[
(...((U \otimes e_n) \otimes e_{n-1})... \otimes e_1) \quad \text{fold right unit left}
\]
\[
(...((U \otimes e_1) \otimes e_2)... \otimes e_n) \quad \text{fold left unit left}
\]
FoldL unit left in Haskell

\[ \text{foldlul} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \]

\[ \text{foldlul } u \ [] = u \]

\[ \text{foldlul } f u (h:t) = \text{foldlul } f (f u h) t \]

Notice the unit \( u \) is of type \( b \), list elements are of type \( a \), and the function \( f \) is of type \( b \rightarrow a \rightarrow b \).
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

\[
\text{map'} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map'} _ [] = []
\]

\[
\text{map'} f (h:t) = f h : \text{map'} f t
\]

_ means that the argument is not used (read “don’t care”).
map' is to distinguish it from the Prelude map function.
Filter in Haskell

\texttt{filter'} :: (a\rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]

\texttt{filter'} _ [] = []

\texttt{filter'} p (h:t) = \texttt{if } p h \texttt{ then } h:\texttt{filter'} p t
\texttt{else } \texttt{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

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 \texttt{PascalList} is evaluated when needed

\begin{verbatim}
fun lazy \{\texttt{PascalList Row}\}  
  Row | \{\texttt{PascalList} 
      \{AddList 
        \{ShiftLeft Row\} 
        \{ShiftRight Row\}\}\}\}
end
\end{verbatim}
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

C. Varela; Adapted from S. Haridi and P. Van Roy
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

```haskell
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

\textbf{ints} :: (\text{Num } a) \Rightarrow a \rightarrow [a]
\text{ints } n = n : \text{ints} (n+1)

\textbf{sieve} :: (\text{Integral } a) \Rightarrow [a] \rightarrow [a]
\text{sieve} (x:xr) = x : \text{sieve} (\text{filter} (\lambda y \rightarrow (y \ `\mod` \ x /= 0)) \ xr)

\textbf{primes} :: (\text{Integral } a) \Rightarrow [a]
\text{primes} = \text{sieve} (\text{ints } 2)

Functions in Haskell are lazy by default. You can use \text{take} 20 \text{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
  – \([X*Y \mid X = 1..10 ; Y = 1..X]\)
• Example:
  – \([1*1 2*1 2*2 3*1 3*2 3*3 \ldots 10*10]\)
List Comprehensions

• The general form is
• \[
\{ f(x,y, ...,z) \mid x \leftarrow \text{gen}(a_1,...,an) ; \text{guard}(x,...) \\
y \leftarrow \text{gen}(x, a_1,...,an) ; \text{guard}(y,x,...) \\
....
\}
\]

• No linguistic support in Mozart/Oz, but can be easily expressed
Example 1

- $z = [x \# x \mid x \leftarrow \text{from}(1,10)]$
- $Z = \{\text{LMap } \{\text{LFrom 1 10}\} \ \text{fun } \{\$ X\} \ X\#X \ \text{end}\}$

- $z = [x \# y \mid x \leftarrow \text{from}(1,10), \ y \leftarrow \text{from}(1,x)]$
- $Z = \{\text{LFlatten}\$
  \{\text{LMap } \{\text{LFrom 1 10}\} \$
    \ \text{fun } \{\$ X\} \ \{\text{LMap } \{\text{LFrom 1 X}\} \$
      \ \text{fun } \{\$ Y\} \ X\#Y \ \text{end}\$
    \end{fun}$
  \} \$
end\$
\}$

C. Varela; Adapted from S. Haridi and P. Van Roy
Example 2

- $z = \{x \# y \mid x \leftarrow \text{from}(1,10), y \leftarrow \text{from}(1,x), x+y \leq 10\}$
- $Z = \{\text{LFilter}
\begin{array}{ll}
\text{LFlatten} \\
\text{LMap} \{\text{LFrom 1 10}\} \\
\text{fun} \{\$ X\} \{\text{LMap} \{\text{LFrom 1 X}\} \\
\text{fun} \{\$ Y\} X \# Y \text{ end} \}
\end{array}\text{ end} \}
\}
fun \{\$ X\#Y\} X+Y \leq 10 \text{ end}\} $
List Comprehensions in Haskell

\[ lc1 = [(x,y) | x \leftarrow [1..10], y \leftarrow [1..x]] \]

\[ lc2 = \text{filter (}\lambda (x,y) \rightarrow (x+y \leq 10)) \text{ } lc1 \]

\[ lc3 = [(x,y) | x \leftarrow [1..10], y \leftarrow [1..x], x+y \leq 10] \]

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

```haskell
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 \( \langle \text{stack } T \rangle \) whose elements are of any type \( T \)
  
  \[
  \begin{align*}
  \text{fun } & \{\text{NewStack}\}: \langle \text{Stack } T \rangle \\
  \text{fun } & \{\text{Push } \langle \text{Stack } T \rangle \langle T \rangle \}: \langle \text{Stack } T \rangle \\
  \text{fun } & \{\text{Pop } \langle \text{Stack } T \rangle \langle T \rangle \}: \langle \text{Stack } T \rangle \\
  \text{fun } & \{\text{IsEmpty } \langle \text{Stack } T \rangle \}: \langle \text{Bool} \rangle \\
  \end{align*}
  \]

• These operations normally satisfy certain laws:
  
  \[
  \{\text{IsEmpty } \{\text{NewStack}\}\} = \text{true}
  \]
  
  for any \( E \) and \( S0, S1=\{\text{Push } S0 E\} \) and \( S0 = \{\text{Pop } S1 E\} \) hold
  
  \[
  \{\text{Pop } \{\text{NewStack}\} E\} \text{ raises error}
  \]
Stack (two implementations)

```
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
```
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

```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:

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

• 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 \( \text{do} \):
  
  \[
  \text{do } e_1 \ ; \ e_2 = e_1 >> e_2 \\
  \text{do } p \leftarrow e_1; e_2 = e_1 >>> \text{\textbackslash }p \rightarrow e_2
  \]

- Pattern match can fail, so the full translation is:
  
  \[
  \text{do } p \leftarrow e_1; e_2 = e_1 >>> (\text{\textbackslash }v \rightarrow \text{case of } p \rightarrow e_2 \\
  \_ \rightarrow \text{fail “s”})
  \]

- 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:
  
  \[
  \begin{align*}
  \text{return } a \mathrel{>>=} k &= k \ a \\
  m \mathrel{>>=} \text{return} &= m \\
  xs \mathrel{>>=} \text{return} \cdot f &= \text{fmap } f \ xs \\
  m \mathrel{>>=} (\lambda x \to k \ x \mathrel{>>=} h) &= (m \mathrel{>>=} k) \mathrel{>>=} h
  \end{align*}
  \]

- These laws ensure that we can bind together monadic values with \( \mathrel{>>=} \) and inject values into the monad (container) using \text{return} in consistent ways.

- The MonadPlus class includes an \text{mzero} element and an \text{mplus} operation. For lists, \text{mzero} is the empty list \([\ ]\), and the \text{mplus} operation is list concatenation (++).
List comprehensions with monads

\[ \text{l}c1 = [(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..x]] \]

\[ \text{l}c1' = \text{do } x \leftarrow [1..10] \]
\[ \quad y \leftarrow [1..x] \]
\[ \quad \text{return } (x,y) \]

\[ \text{l}c1'' = [1..10] >>= (\lambda x \rightarrow [1..x] >>= (\lambda y \rightarrow \text{return } (x,y))) \]

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)

\[
\text{lc3} = [(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..x], x+y \leq 10] \\
\text{lc3'} = \text{do} x \leftarrow [1..10] \\
\text{\quad y \leftarrow [1..x] \\
\text{\quad True \leftarrow return (x+y \leq 10) \\
\text{\quad return (x,y)} \\
\]

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

\[
\text{lc3''} = [1..10] >>= (\lambda x \rightarrow \\
\text{\quad [1..x] >>= (\lambda y \rightarrow \\
\text{\quad \quad return (x+y \leq 10) >>= \\
\text{\quad \quad \quad (\lambda b \rightarrow \text{case b of True \rightarrow return (x,y); _ \rightarrow fail ""})}))}
\]
An instruction counter monad

- We will create an instruction counter using a monad $R$:

```haskell
data R a = R (Resource -> (a, Resource)) -- the monadic type

instance Monad R where
  -- (>>=) :: R a -> (a -> R b) -> R b
  R c1 >>= fc2 = R (\r -> let (s, r') = c1 r
                    R c2 = fc2 s in
                    c2 r')

  -- return :: a -> R a
  return v = R (\r -> (v, r))
```

A computation is modeled as a function that takes a resource $r$ and returns a value of type $a$, and a new resource $r'$. The resource is implicitly carried state.
An instruction counter monad (2)

- Counting steps:
  ```haskell
type Resource = Integer  -- type synonym
step :: a -> R a
step v = R (\r -> (v, r+1))
count :: R Integer -> (Integer, Resource)
count (R c) = c 0
```

- Lifting a computation to the monadic space:
  ```haskell
incR :: R Integer -> R Integer
incR n = do nValue <- n
           step (nValue+1)
count (incR (return 5))  -- displays (6,1)
```

An inc computation (Integer -> Integer) is lifted to the monadic space: (R Integer -> R Integer).
An instruction counter monad (3)

- Generic lifting of operations to the R monad:
  
  ```haskell
  lift1 :: (a->b) -> R a -> R b
  lift1 f n = do nValue <- n
                   step (f nValue)
  lift2 :: (a->b->c) -> R a -> R b -> R c
  lift2 f n1 n2 = do n1Value <- n1
                        n2Value <- n2
                        step (f n1Value n2Value)
  
  instance Num a => Num (R a) where
    (+)       = lift2 (+)
    (-)       = lift2 (-)
    fromInteger = return . fromInteger
  ```

  With generic lifting operations, we can define
  
  ```haskell
  incR = lift1 (+1)
  ```
Lifting conditionals to the R monad:

```haskell
ifR :: R Bool -> R a -> R a -> R a
ifR b t e = do bVal <- b
    if bVal then t
    else e
```

```haskell
(<=*) :: (Ord a) => R a -> R a -> R Bool
(<=*) = lift2 (<=)
```

```haskell
fib :: R Integer -> R Integer
fib n = ifR (n <=* 1) n (fib (n-1) + fib (n-2))
```

We can now count the computation steps with:
```
count (fib 10) => (55,1889)
```
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.