

Functional Programming:

Lists, Pattern Matching, Recursive Programming
(CTM Sections 1.1-1.7, 3.2, 3.4.1-3.4.2, 4.7.2)

Carlos Varela

RPI

September 8, 2026

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Introduction to Functional Programming

- Declarative variables
- Structured data (example: lists)
- Functions over lists
- Correctness and complexity

Variables

- Variables are short-cuts for values, they cannot be assigned more than once

`v = 9999*9999`

`v * v`

- Variable identifiers: is what you type
- You **cannot** change a variable's value (e.g., `v++`).

Functions

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

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

$$0! = 1$$

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

`factorial` :: Integer -> Integer

`factorial` 0 = 1

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

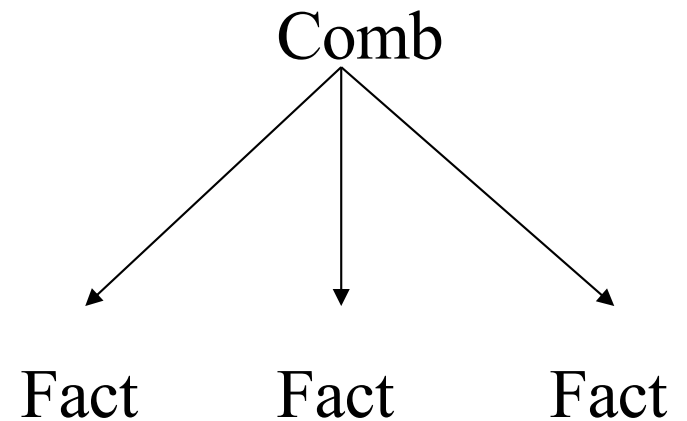
- `factorial` is declared in the environment
- Try large factorial: `factorial 100`
 - Integers are arbitrary precision

Composing functions

- Combinations of r items taken from n .
- The number of subsets of size r taken from a set of size n

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

`comb` n r = (factorial n) ``div`` (factorial r * factorial ($n-r$))



- Example of functional abstraction

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 []
- Lists are created by means of ":" (cons)

```
      1
    1  1
  1  2  1
1  3  3  1
1  4  6  4  1
```

h = 1

t = [2,3,4,5]

h:t --- This will show [1,2,3,4,5]

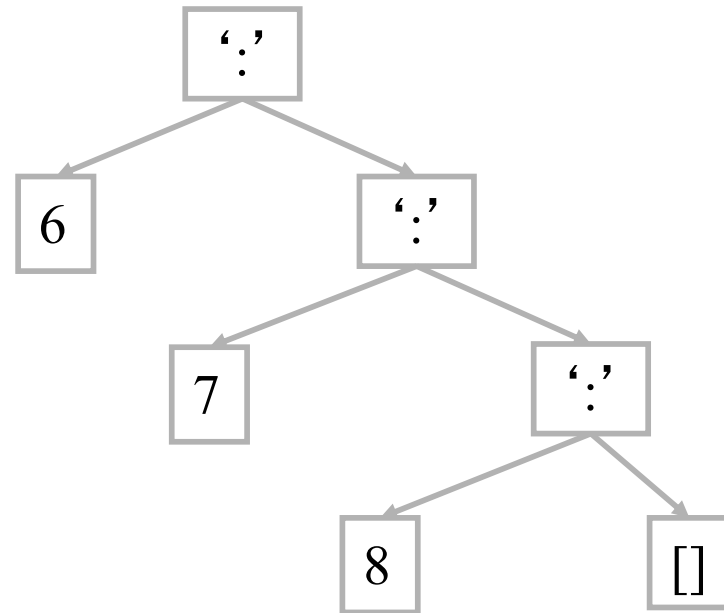
Lists (2)

- Taking lists apart (selecting components)
- A list has two components: a head, and a tail

`l = [5,6,7,8]`

`head l` gives 5

`tail l` gives [6,7,8]



Pattern matching

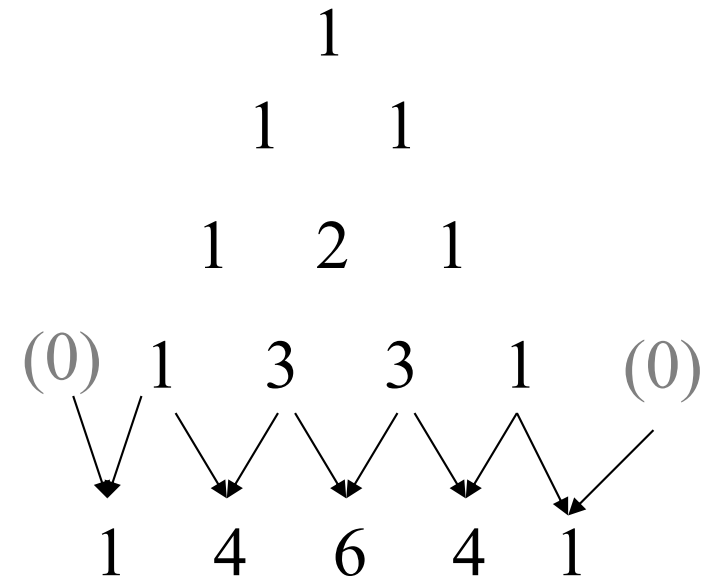
- Another way to take a list apart is by use of pattern matching with or without a case instruction:

```
length' c l = case l of []    -> 0  
                  (h:t) -> 1 + length' c t
```

```
length' []      = 0  
length' (h:t)   = 1 + length' t
```

Functions over lists

- Compute the function: `pascal n`
 - Takes an integer n , and returns the n th 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)

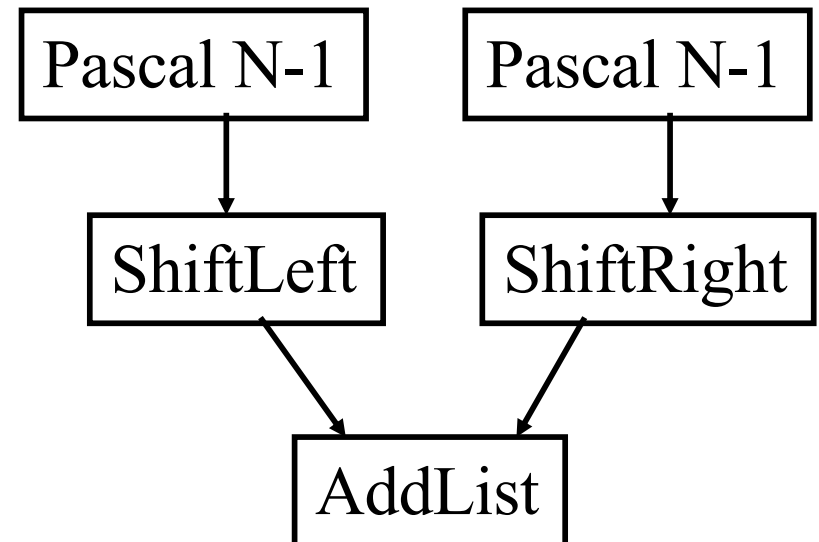
-- Pascal triangle row

`pascal :: Integer -> [Integer]`

`pascal 1 = [1]`

`pascal n = addList (shiftLeft (pascal (n-1)))
 (shiftRight (pascal (n-1)))`

Pascal N



Functions over lists (3)

`shiftLeft [] = [0]`

`shiftLeft (h:t) = h:shiftLeft t`

`shiftRight l = 0:l`

`addList [] [] = []`

`addList (h1:t1) (h2:t2) = (h1+h2):addList t1 t2`

Top-down program development

- Understand how to solve the problem by hand
- Try to solve the task by decomposing it to simpler tasks
- Devise the main function (main task) in terms of suitable auxiliary functions (subtasks) that simplify the solution (shiftLeft, shiftRight and addList)
- Complete the solution by writing the auxiliary functions
- Test your program bottom-up: auxiliary functions first.

Is your program correct?

- “A program is correct when it does what we would like it to do”
- In general we need to reason about the program:
- **Semantics for the language**: a precise model of the operations of the programming language
- **Program specification**: a definition of the output in terms of the input (usually a mathematical function or relation)
- Use mathematical techniques to reason about the program, using programming language semantics

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

`factorial` :: Integer -> Integer

`factorial` 0 = 1

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

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

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

- Base Case $n=0$: factorial 0 returns 1
- Inductive Case $n>0$: factorial n returns n * factorial (n-1)
 - Assume factorial n-1 is correct, from the spec we see that factorial n is equal to n * factorial (n-1)

Complexity

- Pascal runs very slow, try:
pascal 20
- pascal 20 calls: pascal 19 twice, pascal 18 four times, pascal 17 eight times, ..., pascal 1 2^{19} times
- Execution time of a program up to a constant factor is called the program's *time complexity*.
- Time complexity of pascal n is proportional to 2^n (exponential)
- Programs with exponential time complexity are impractical

```
pascal 1 = [1]
```

```
pascal n = addList (shiftLeft (pascal (n-1)))  
              (shiftRight (pascal (n-1)))
```

Faster Pascal

- Introduce a local variable l
- Compute `fastPascal (n-1)` only once
- Try with 30 rows.
- `fastPascal` is called n times, each time a list on the average of size $n/2$ is processed
- The time complexity is proportional to n^2 (polynomial)
- Low order polynomial programs are practical.

```
fastPascal 1 = [1]
fastPascal n = addList (shiftLeft l)
                  (shiftRight l)

where
  l = fastPascal (n-1)
  shiftLeft [] = [0]
  shiftLeft (h:t) = h:shiftLeft t
  shiftRight l = 0:l
  addList [] [] = []
  addList (h1:t1) (h2:t2) = (h1+h2):addList t1 t2
```

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:

$$S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_{\text{final}}$$

The general scheme

iterate' $s_i =$

if *isDone* s_i then s_i

else let $s_{i+1} = \text{transform } s_i$ in

iterate' s_{i+1}

- *isDone* and *transform* are problem dependent

The computation model

- STACK : [$\mathbf{r} = \{\text{iterate}' s_0\}$]
- STACK : [$\mathbf{s}_1 = \{\text{transform } s_0\}$,
 $\mathbf{r} = \{\text{iterate}' s_1\}$]
- STACK : [$\mathbf{r} = \{\text{iterate}' s_i\}$]
- STACK : [$\mathbf{s}_{i+1} = \{\text{transform } s_i\}$,
 $\mathbf{r} = \{\text{iterate}' s_{i+1}\}$]
- STACK : [$\mathbf{r} = \{\text{iterate}' s_{i+1}\}$]

Newton's method for the square root of a positive real number

- Given a real number x , start with a guess g , and improve this guess iteratively until it is accurate enough
- The improved guess g' is the average of g and x/g :

$$g' = (g + x / g) / 2$$

$$\varepsilon = g - \sqrt{x}$$

$$\varepsilon' = g' - \sqrt{x}$$

For g' to be a better guess than g : $\varepsilon' < \varepsilon$

$$\varepsilon' = g' - \sqrt{x} = (g + x / g) / 2 - \sqrt{x} = \varepsilon^2 / 2g$$

$$\text{i.e. } \varepsilon^2 / 2g < \varepsilon, \quad \varepsilon / 2g < 1$$

$$\text{i.e. } \varepsilon < 2g, \quad g - \sqrt{x} < 2g, \quad 0 < g + \sqrt{x}$$

Newton's method for the square root of a positive real number

- Given a real number x , start with a guess g , and improve this guess iteratively until it is accurate enough
- The improved guess g' is the average of g and x/g :
- Accurate enough is defined as:

$$|x - g^2| / x < 0.00001$$

SqrtIter

```
sqrtIter guess x =  
  if goodEnough guess x then guess  
  else let guess1 = improve guess x  
    in sqrtIter guess1 x
```

- Compare to the general scheme:
 - The state is the pair guess and x
 - *isDone* is implemented by the procedure goodEnough
 - *transform* is implemented by the procedure improve

The program version 1

```
sqrtlter guess x =  
    if goodEnough guess x  
    then guess  
    else sqrtlter (improve guess x) x
```

```
sqrt1 x = let guess = 1  
    in sqrtlter guess x
```

```
improve guess x =  
    (guess + x / guess) / 2  
  
goodEnough guess x =  
    abs (x-guess*guess)/x < 0.00001
```

Using local functions

- The main procedure `sqrt1` uses the helper procedures `sqrtlter`, `goodEnough`, `improve`, and `abs`
- `sqrtlter` is only needed inside `sqrt`
- `goodEnough` and `improve` are only needed inside `sqrtlter`
- `abs` (absolute value) is a general utility
- The general idea is that helper procedures should not be visible globally, but only locally

Sqrt version 2

```
sqrt2 x = let guess = 1
          in sqrtIter guess x
          where
            sqrtIter guess x = if goodEnough guess x
                                then guess
                                else sqrtIter (improve guess x) x
            improve guess x = (guess + x / guess) / 2
            goodEnough guess x = abs (x - guess * guess) / x < 0.00001
```

Sqrt version 3

- Define goodEnough and improve inside sqrtIter

```
sqrt3 x = let guess = 1
          sqrtIter guess x = if goodEnough
                              then guess
                              else sqrtIter improve x
          where
            improve = (guess + x / guess) / 2
            goodEnough = abs (x-guess*guess)/x < 0.00001
          in sqrtIter guess x
```

Sqrt version 3

- Define goodEnough and improve inside sqrtIter

```
sqrt3 x = let guess = 1
          sqrtIter guess x = if goodEnough
                             then guess
                             else sqrtIter improve x
          where
            improve = (guess + x / guess) / 2
            goodEnough = abs (x-guess*guess)/x < 0.00001
          in sqrtIter guess x
```

The program has a single drawback: on each iteration two function values are created, one for improve and one for goodEnough

Sqrt final version

The final version is
a compromise between
abstraction and efficiency

```
sqrt4 x = let guess = 1
          improve guess = (guess + x / guess) / 2
          goodEnough guess = abs (x - guess * guess) / x < 0.00001
          sqrtIter guess = if goodEnough guess
                           then guess
                           else sqrtIter (improve guess)
          in sqrtIter guess
```

From a general scheme to a control abstraction (1)

iterate' s =

if *isDone* s_i then s_i

else let $s_{i+1} = \text{transform } s_i$ in

iterate' s_{i+1}

- *isDone* and *transform* are problem dependent

From a general scheme to a control abstraction (2)

```
iterate' s isDone transform =  
  if isDone s then s  
  else let s1 = transform s in  
    iterate' s1 isDone transform
```

```
iterate' s =  
  if isDone si then si  
  else let si+1 = transform si in  
    iterate' si+1
```

Sqrt using the Iterate abstraction

```
sqrt5 x = iterate' 1.0 goodEnough improve  
  where goodEnough = \g -> (abs (x - g*g))/x < 0.00001  
        improve = \g -> (g + x/g)/2.0
```

Sqrt using the control abstraction

`sqrt6` $x = \text{iterate}'\ 1$

$(\backslash g \rightarrow (\text{abs } (x - g * g)) / x < 0.00001)$

$(\backslash g \rightarrow (g + x / g) / 2.0)$

`iterate'` can be a linguistic abstraction

Sqrt using infinite lists

```
sqrt7 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.

Sqrt using iterate abstraction

`sqrt8` $x = \text{head } (\text{dropWhile } (\text{not } . \text{goodEnough}) \text{ sqrtGuesses})$

where

$\text{goodEnough guess} = (\text{abs } (x - \text{guess} * \text{guess})) / x < 0.00001$

$\text{improve guess} = (\text{guess} + x / \text{guess}) / 2.0$

$\text{sqrtGuesses} = \text{iterate improve } 1$

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

Exercises

12. Prove the correctness of `addList` and `shiftLeft`.
13. Create an alternative version of Pascal triangle, not using `shiftLeft`. Instead, extend `addList` to allow for lists of different length.
14. Prove your new `pascal` function is correct. Make `addList` and a generic `opList` version commutative.
15. CTM Exercise 3.10.2 (page 230)
16. CTM Exercise 3.10.3 (page 230)
17. Develop a control abstraction for iterating over a list of elements.