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 12, 2017

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Introduction to Oz

- An introduction to programming concepts
- 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

```
declare
V = 9999*9999
{Browse V*V}
```

- Variable identifiers: is what you type
- Store variable: is part of the memory system
- The **declare** statement creates a store variable and assigns its memory address to the identifier ’V’ in the environment

Functions

- Compute the factorial function: $n! = 1 \times 2 \times \cdots \times (n-1) \times n$
- Start with the mathematical definition

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

$$0! = 1$$

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

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

Factorial in Haskell

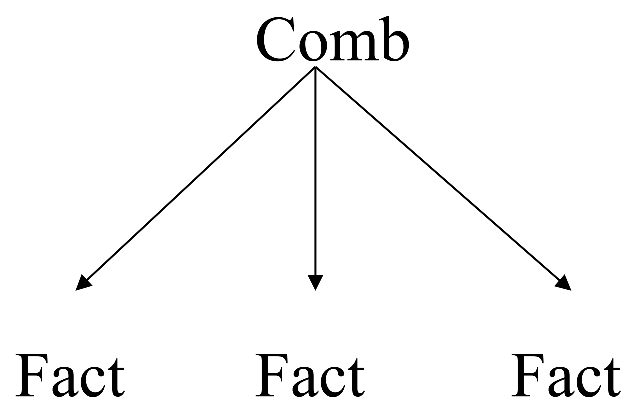
```
factorial :: Integer -> Integer
factorial  0              = 1
factorial  n | n > 0      = n * factorial (n-1)
```

Composing functions

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

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

```
declare
fun {Comb N R}
  {Fact N} div ({Fact R}*{Fact N-R})
end
```



- 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 nil
- Lists are created by means of "|" (cons)

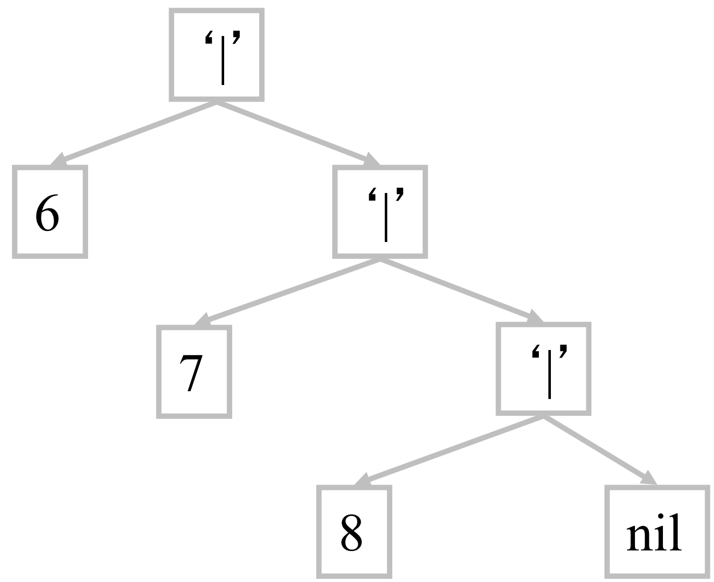
```
declare
H=1
T = [2 3 4 5]
{Browse H|T}  % This will show [1 2 3 4 5]
```

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

Lists (2)

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

```
declare L = [5 6 7 8]
L.1 gives 5
L.2 give [6 7 8]
```



Pattern matching

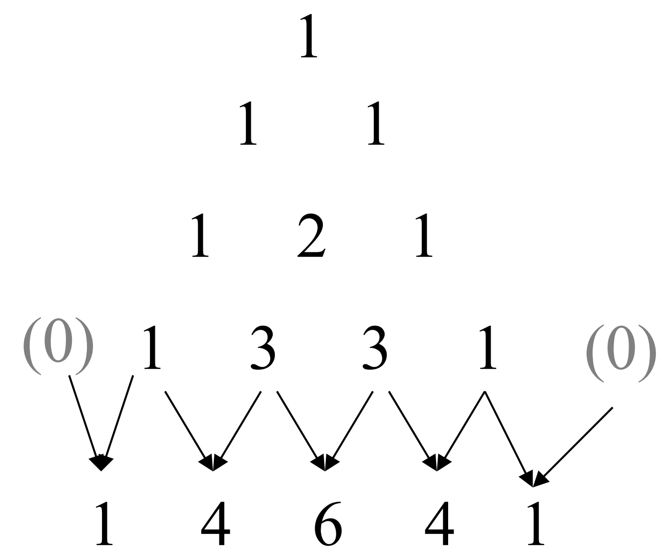
- Another way to take a list apart is by use of pattern matching with a case instruction

```
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

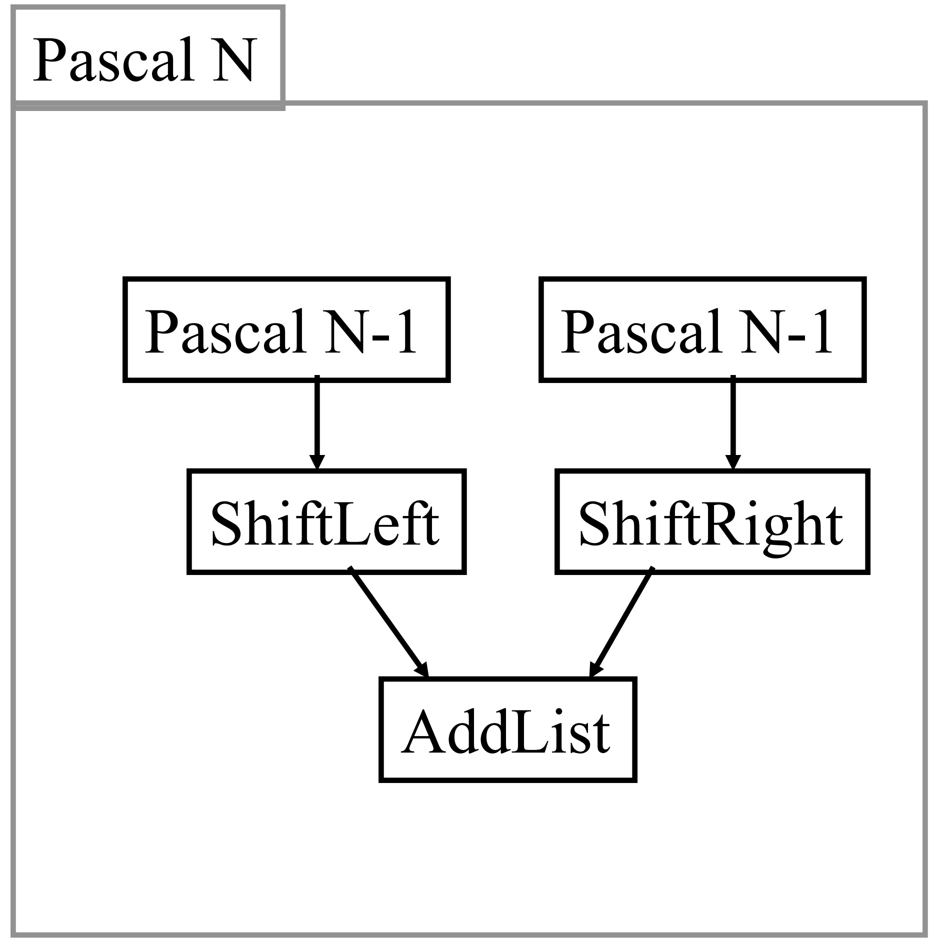


Shift right [0 1 3 3 1]

Shift left [1 3 3 1 0]

Functions over lists (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
```

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

```
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}

Complexity

- Pascal runs very slow, try {Pascal 24}
- {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

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

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.

```
fun {FastPascal N}
  if N==1 then [1]
  else
      local L in
         L={FastPascal N-1}
         {AddList {ShiftLeft L} {ShiftRight L}}
      end
  end
end
```

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

The computation model

- STACK : [R={Iterate S_0}]
- STACK : [S_1 = {*Transform* S_0},
  R={Iterate S_1}]

- STACK : [R={Iterate S_i}]
- STACK : [S_{i+1} = {*Transform* S_i},
  R={Iterate S_{i+1}}]

- STACK : [R={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: $\varepsilon' < \varepsilon$

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

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

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

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

```
fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then Guess
  else
    Guess1 = {Improve Guess X} in
    {SqrtIter Guess1 X}
  end
end
```

- 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

```
fun {Sqrt X}
  Guess = 1.0
in {SqrtIter Guess X}
end
fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then
     Guess
  else
     {SqrtIter {Improve Guess X} X}
  end
end
```

```
fun {Improve Guess X}
  (Guess + X/Guess)/2.0
end
fun {GoodEnough Guess X}
  {Abs X - Guess*Guess}/X < 0.00001
end
```

Using local procedures

- The main procedure Sqrt uses the helper procedures SqrtIter, GoodEnough, Improve, and Abs
- SqrtIter is only needed inside Sqrt
- GoodEnough and Improve are only needed inside SqrtIter
- 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

```
local
  fun {SqrtIter Guess X}
    if {GoodEnough Guess X} then Guess
    else {SqrtIter {Improve Guess X} X} end
  end
  fun {Improve Guess X}
    (Guess + X/Guess)/2.0
  end
  fun {GoodEnough Guess X}
    {Abs X - Guess*Guess}/X < 0.000001
  end
in
  fun {Sqrt X}
    Guess = 1.0
  in {SqrtIter Guess X} end
end
```

Sqrt version 3

- Define GoodEnough and Improve inside SqrtIter

```
local
  fun {SqrtIter Guess X}
    fun {Improve}
      (Guess + X/Guess)/2.0
    end
    fun {GoodEnough}
      {Abs X - Guess*Guess}/X < 0.000001
    end
  in
     if {GoodEnough} then Guess
     else {SqrtIter {Improve} X} end
  end
in fun {Sqrt X}
     Guess = 1.0 in
     {SqrtIter Guess X}
  end
end
```

Sqrt version 3

- Define GoodEnough and Improve inside SqrtIter

```
local
  fun {SqrtIter Guess X}
    fun {Improve}
      (Guess + X/Guess)/2.0
    end
    fun {GoodEnough}
      {Abs X - Guess*Guess}/X < 0.000001
    end
  in
     if {GoodEnough} then Guess
     else {SqrtIter {Improve} X} end
  end
in fun {Sqrt X}
     Guess = 1.0 in
     {SqrtIter Guess X}
  end
end
```

The program has a single drawback: on each iteration two procedure values are created, one for Improve and one for GoodEnough

Sqrt final version

```
fun {Sqrt X}
  fun {Improve Guess}
    (Guess + X/Guess)/2.0
  end
  fun {GoodEnough Guess}
    {Abs X - Guess*Guess}/X < 0.000001
  end
  fun {SqrtIter Guess}
     if {GoodEnough Guess} then Guess
     else {SqrtIter {Improve Guess} } end
  end
  Guess = 1.0
in {SqrtIter Guess}
end
```

The final version is a compromise between abstraction and efficiency

From a general scheme to a control abstraction (1)

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 Iterate abstraction

```
fun {Sqrt X}
  fun {Improve Guess}
    (Guess + X/Guess)/2.0
  end
  fun {GoodEnough Guess}
    {Abs X - Guess*Guess}/X < 0.000001
  end
  Guess = 1.0
in
  {Iterate Guess GoodEnough Improve}
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)
```

Exercises

12. Prove the correctness of AddList and ShiftLeft.
13. Prove that the alternative version of Pascal triangle (not using ShiftLeft) is correct. Make AddList and OpList commutative.
14. Modify the Pascal function to use local functions for AddList, ShiftLeft, ShiftRight. Think about the abstraction and efficiency tradeoffs.
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.