Introduction to Programming Concepts (CTM 1.1-1.11)

Carlos Varela

RPI

February 26, 2015

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
- Lazy functions
- Higher-order programming
- Concurrency and dataflow

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:
- 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}}

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

$$0! = 1$$

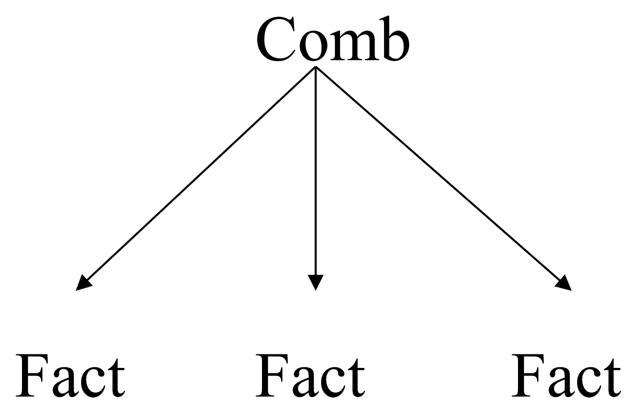
$$n! = n \times (n-1)! \text{ if } n > 0$$

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)

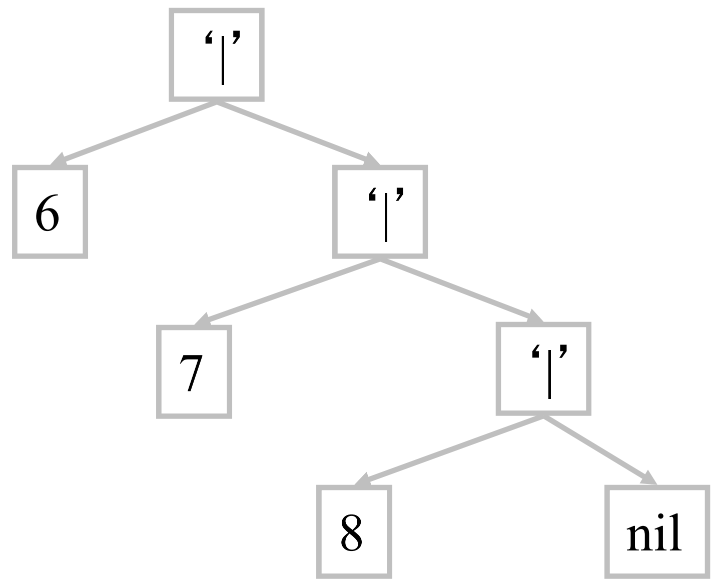
```
declare
H=1
T = [2 3 4 5]
{Browse H|T}  % This will show [1 2 3 4 5]
```

```
        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

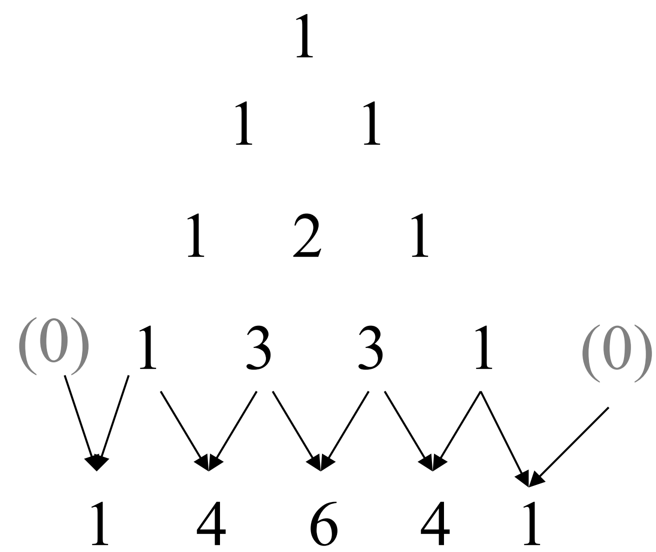
- Another way to take a list apart is by use of pattern matching with a case instruction

```
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

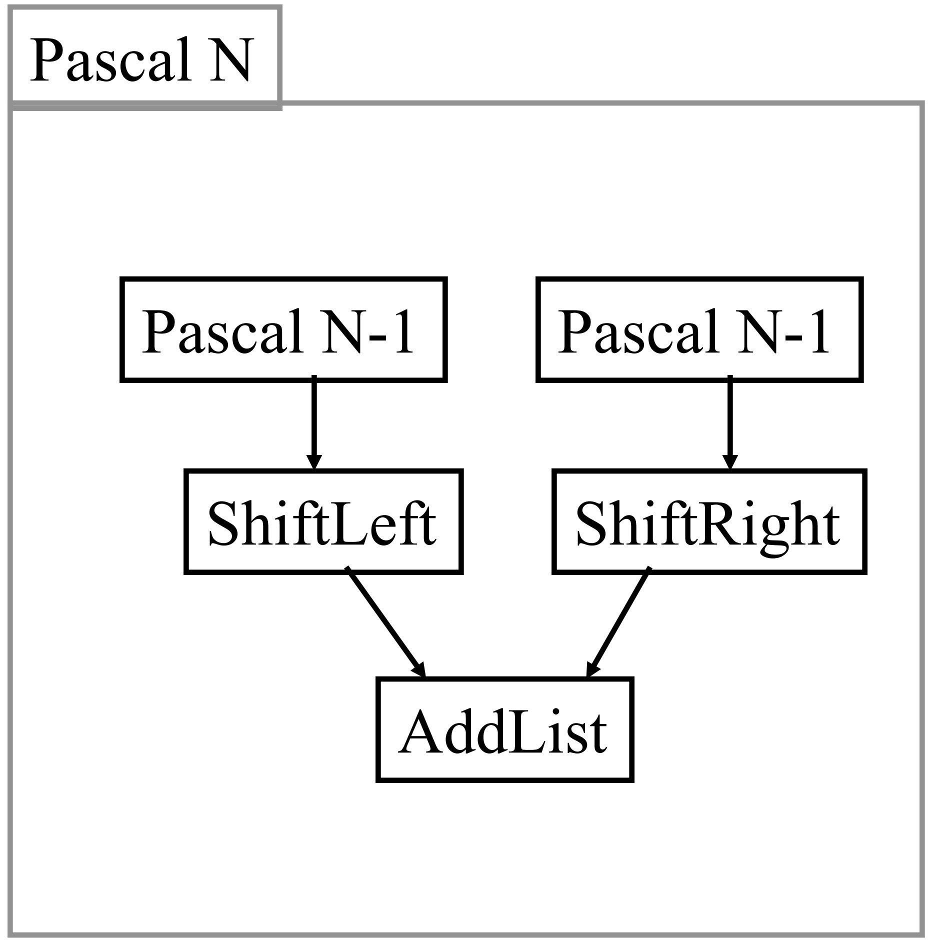


Shift right [0 1 3 3 1]

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
```

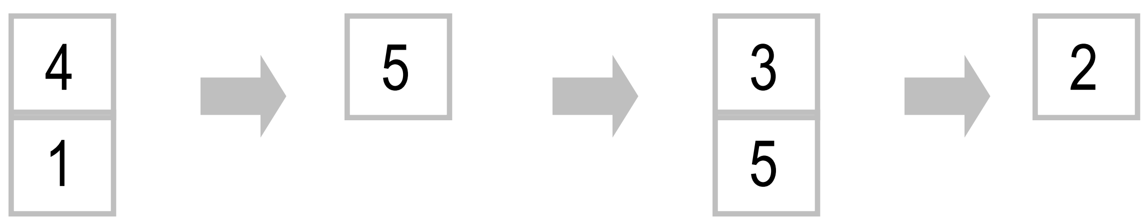
$$\underbrace{1\times 2\times \cdots \times (n-1)}_{Fact(n-1)}\times n$$

- 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}

Multiple accumulators

- Consider a stack machine for evaluating arithmetic expressions
- Example: (1+4)-3
- The machine executes the following instructions

push(1)
push(4)
plus
push(3)
minus



Multiple accumulators (2)

- Example: (1+4)-3
- The arithmetic expressions are represented as trees:

  minus(plus(1 4) 3)

- Write a procedure that takes arithmetic expressions represented as trees and output a list of stack machine instructions and counts the number of instructions

```
proc {ExprCode Expr Cin Cout Nin Nout}
```

- Cin: initial list of instructions
- Cout: final list of instructions
- Nin: initial count
- Nout: final count

Multiple accumulators (3)

```
proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then C1 N1 in
    C1 = plus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] minus(Expr1 Expr2) then C1 N1 in
    C1 = minus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] I andthen {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

Multiple accumulators (4)

```
proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then C1 N1 in
    C1 = plus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] minus(Expr1 Expr2) then C1 N1 in
    C1 = minus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] I andthen {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

```
proc {SeqCode Es C0 C N0 N}
  case Es
  of nil then C = C0 N = N0
  [] E|Er then N1 C1 in
    {ExprCode E C0 C1 N0 N1}
    {SeqCode Er C1 C N1 N}
  end
end
```

Shorter version (4)

```
proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] plus|C0 C N0 + 1 N}
  [] minus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] minus|C0 C N0 + 1 N}
  [] I andthen {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

```
proc {SeqCode Es C0 C N0 N}
  case Es
  of nil then C = C0 N = N0
  [] E|Er then N1 C1 in
    {ExprCode E C0 C1 N0 N1}
    {SeqCode Er C1 C N1 N}
  end
end
```

Functional style (4)

```
fun {ExprCode Expr t(C0 N0) }
   case Expr
   of plus(Expr1 Expr2) then
      {SeqCode [Expr2 Expr1] t(plus|C0 N0 + 1)}
   [] minus(Expr1 Expr2) then
      {SeqCode [Expr2 Expr1] t(minus|C0 N0 + 1)}
   [] I andthen {IsInt I} then
      t(push(I)|C0 N0 + 1)
   end
end
```

```
fun {SeqCode Es T}
   case Es
   of nil then T
   [] E|Er then
      T1 = {ExprCode E T} in
      {SeqCode Er T1}
   end
end
```

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
```

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 result 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
```

Lazy evaluation (3)

- Lazy evaluation will avoid redoing work if you decide first you need the 10th row and later the 11th row
- The function continues where it left off

```
declare
L = {PascalList [1]}
{Browse L}
{Browse L.1}
{Browse L.2.1}
```

```
L<Future>
[1]
[1 1]
```

Higher-order programming

- Assume we want to write another Pascal function, which instead of adding numbers, performs exclusive-or on them
- It calculates for each number whether it is odd or even (parity)
- Either write a new function each time we need a new operation, or write one generic function that takes an operation (another function) as argument
- The ability to pass functions as arguments, or return a function as a result is called *higher-order programming*
- Higher-order programming is an aid to build generic abstractions

Variations of Pascal

- Compute the parity Pascal triangle

```
fun {Xor X Y} if X==Y then 0 else 1 end end
```

1	1
1 1	1 1
1 2 1	1 0 1
1 3 3 1	1 1 1 1
1 4 6 4 1	1 0 0 0 1

Higher-order programming

```
fun {GenericPascal Op N}
  if N==1 then [1]
  else L in L = {GenericPascal Op N-1}
    {OpList  Op {ShiftLeft L} {ShiftRight L}}
  end
end
fun {OpList Op L1 L2}
     case L1 of H1|T1 then
          case L2 of H2|T2 then
               {Op H1 H2}|{OpList Op T1 T2}
          end
     else nil end
end
```

```
fun {Add N1 N2} N1+N2 end
fun {Xor N1 N2}
     if N1==N2 then 0 else 1 end
end

fun {Pascal N} {GenericPascal Add N} end
fun {ParityPascal N}
     {GenericPascal Xor N}
end
```

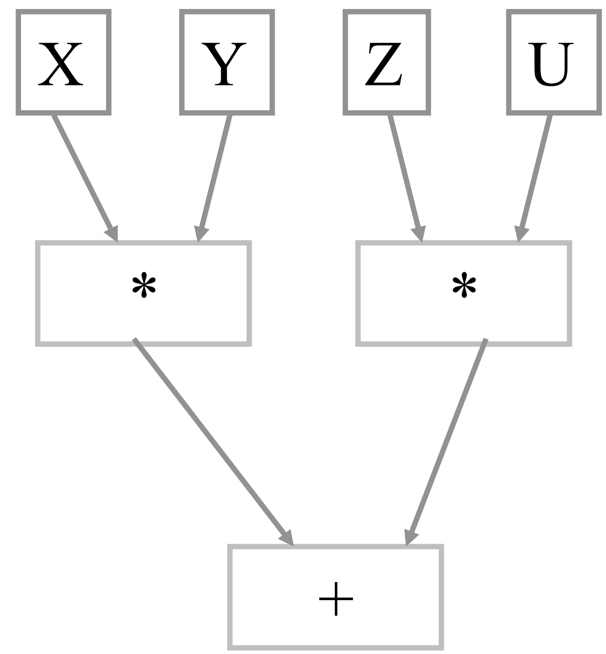
Concurrency

- How to do several things at once
- Concurrency: running several activities each running at its own pace
- A *thread* is an executing sequential program
- A program can have multiple threads by using the `thread` instruction
- `{Browse 99*99}` can immediately respond while Pascal is computing

```
thread
   P in
   P = {Pascal 21}
   {Browse P}
end
{Browse 99*99}
```

Dataflow

- What happens when multiple threads try to communicate?
- A simple way is to make communicating threads synchronize on the availability of data (data-driven execution)
- If an operation tries to use a variable that is not yet bound it will wait
- The variable is called a *dataflow variable*



Dataflow (II)

- Two important properties of dataflow
  - Calculations work correctly independent of how they are partitioned between threads (concurrent activities)
  - Calculations are patient, they do not signal error; they wait for data availability
- The dataflow property of variables makes sense when programs are composed of multiple threads

```
declare X
thread
  {Delay 5000} X=99
End
{Browse 'Start'} {Browse X*X}
```

```
declare X
thread
  {Browse 'Start'} {Browse X*X}
end
{Delay 5000} X=99
```

Exercises

30. Prove the correctness of AddList and ShiftLeft.
31. CTM Exercise 1.18.5. (page 24)
32. CTM Exercise 1.18.6. (page 24)
    - c) Change GenericPascal so that it also receives a number to use as an identity for the operation Op: {GenericPascal Op I N}. For example, you could then use it as:

      {GenericPascal Add 0 N}, or

      {GenericPascal fun {$ X Y} X*Y end 1 N}
33. Prove that the alternative version of Pascal triangle (not using ShiftLeft) is correct. Make AddList and OpList commutative.
34. When combining concurrency and dataflow behavior, do you ever get non-determinism? Explain why or why not.