Chapter 9 – Functional Programming

Introduction to Programming Languages

First Edition, 2013

Author: Arvind Bansal

© Chapman Hall / CRC Press

ISBN: 978-146-6565142

Topics Covered

- λ-expressions and evaluation techniques
- Functional Programming without variables
  - Kernel functions and function forming operators
- Abstractions and programming in functional programming languages
- Implementation Models for functional languages
  - SECD machine and eager evaluation
  - Graph reduction strategies
  - Implementing lazy evaluation
- Integration with Other Programming Paradigms
  - Concurrency in functional programming
- Summary

Introduction

- Functional programming is important
  - Declarative style enhances comprehension by removing control
  - More concise
  - Key paradigm in scripting languages *Closure, Ruby, Scala, and Python*
- History: from early 1970s. Initial major language is LISP
- Characteristics of pure functional programming
  - Assign-once variable, no support for global variables
  - Based upon λ–calculus and function forming operators
  - λ-function has three components: variable, body and expression
- Example of λ-expression

λx.	***(x + 4)***	***3***
Var	**expression**	**value**

- Implementation of functional languages
  - SECD machine: a four stack abstract model
  - G-machine based upon graph reduction techniques
  - Evaluation strategy can be eager evaluation or lazy evaluation

Higher Order Function

- Higher order function takes function as one of the arguments
  - The same higher order function can be used to invoke multiple functions
  - Functions can be manipulated as data
- <u>Example</u>

```
(defun   foo (powerFunction   Arg)
        (+ (apply    powerFunction    Arg)  4 )
)

(foo square 4) returns 16

(foo sqrt  4) returns 2
```

λ-expressions

- Components: variable, expression, parameter-value
  - The scope of the variable is in the following expression
  - Variable in expression is binding occurrence of declared variable
  - Parameter value is bound to the declared variable and substituted in the expression
  - λ-expression can be nested. The variable declared at a nesting level is visible only in that nesting level. In nested expression outer level expression can be used as parameter to the next inner level
- Example
  - ***(λx. λy. (λz. z + 2) x + y) 3 4***
  - Has two levels: first level has two variables x and y; X is bound to value 3, and y is bound to value 4. Their scope is the expression x + y
  - The inner level has variable z, and its scope is the inner expression z + 2
  - Expression x + y is parameter for the inner level

Evaluation of λ-expression

- Technique
  - Parameters are bound to variables in left to right order
  - Binding of parameter value to declared variables is called β-reduction
  - The simplification of arithmetic expression is called δ-reduction
  - α-substitution renames the variables in inner level to remove naming conflicts with the same name variables in outer level
- Reduction techniques: AOR vs. NOR (Result is the same)
  - AOR (Application Order Reduction) technique solves from inner level first
  - NOR (Normal Order Reduction) technique solves from outer level first
- ***λx. (+ x x) (λy. y + 4) 3***

NOR Technique	AOR Technique
(+ ((λy. y + 4) 3) ((λy. y + 4) 3))	λx. (+ x x) (λy. y + 4) 3
→β (+ (3 + 4) ((λy. y + 4) 3))	→β λx. (+ x x) (3 + 4)
→δ (+ 7 ((λy. y + 4) 3))	→δ λx. (+ x x) 7
→β (+ 7 (3 + 4))	→β (+ 7 7)
→δ (+ 7 7)	→δ 14
→β 14	

FPP

- Two parts: kernel functions and functional forms
  - Kernel functions are basic functions that can not be split further
  - Functional-forms that combine functions to make bigger function
- FP programming
  - FP separates functions from parameters, and does not have variables
  - It pulls parameter values using identity function
  - FP has kernel functions for arithmetic operations, comparison operations, metalogical predicates, constructing sequences, selector functions(accessing elements from sequences), insertion functions (inserting an element in a sequence), transpose functions, and miscellaneous functions such as length, reverse, identity, rotate etc.
  - Functional forms are composition, apply-all, insertion, construction, conditionals, iteration and recursion
  - Parameters are stored in a sequence in the form $\langle d_1, \ldots d_N \rangle$
  - Functions are written in the form <function-name> : <parameters>
- Example: + : < 2, 3> evaluates to five; >: < 3, 2> returns true

Selector Functions

- Selects an element indexed from left or right
  - **left-selector** - **1l**:<1, 2, 3> gives 1; **3 l:**<a, b, c> gives c
  - **right-selector** - **1r:**<1, 2, 3> gives 3; **2 r**:<a, b, c> gives b
  - **left-tail – tl**:<1, 2, 3> gives <2, 3>; **tl:** <a, b, c> gives < b, c>
  - **right-tail – tlr:** <1, 2, 3> gives <1, 2>; **tlr:** <a, b, c> gives <a, b>
  - **1l:** < > will give the bottom symbol ⊥
  - **tlr:** <> will give the bottom symbol ⊥
- Construction function
  - insert an element in the sequence or joins two sequences
  - **apndl:**<1, <a, b, c>> will derive <1, a, b, c>
  - **apndr:** <1, <a, b, c>> will derive <a, b, c, 1>
  - **apndl:** (1, 2) will derive bottom symbol ⊥ since second element is atom
  - **insert:**<3, <a, b, c, d>, x> will derive <a, b, x, c, d>
  - **append:** <<1, 2, 3>, <a, b, c>> will return <1, 2, 3, a, b, c>

Kernel Functions II

- Transpose functions
  - Make ith row as ith column; jth column as jth row
  - ***transpose:*** *<<1, 2, 3>, <4, 5, 6>>* gives *<<1, 4>, <2, 5>, <3, 6>>*
- Metalogical Predicates
  - Check the type of the objects
  - **Example**: is_float, is_ null, is_nonnull, is_atom, is_sequence
  - **is_float:** 4.5 returns true; is_float: a returns false
  - **is_atom :** <1, 2, 3> will return false
- Miscellaneous functions
  - **length:**<a, b, c> will return 3
  - **distl:** <1, <a, b, c>> will return <<1, a>, <1, b>, <1, c>>
  - **distl:**<1, <>> will return < >
  - **distr:** <1, <a, b, c>> will return <<a, 1>, <b, 1>, <c, 1>>
  - **rotl :** <2, <a, b, c, d, e>> would return a sequence <c, d, e, a, b>
  - **rotr :** <2, <a, b, c, d, e>> would return a sequence <d, e, a, b, c>
  - **reverse:** <1, 2, 3> would return <3, 2, 1>

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Special Functions

- Constant function
  - maps every input value to a constant function
  - 4(x) → 4
- Identity function
  - returns the input value as itself: id(x) → x
  - Used to pull in the parameter value inside the function

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Functional Forms I

- Combines functions to form complex functions
- Composition: cascades the results: f•g(x) ≡ f(g(x))
  - *square* ***• 1l*** *: <4, 5, 6>* will first apply *1l : <4, 5, 6>* to derive the value *4*, and then *square : 4* will derive *16*.
- Construction: applies each function in a sequence of functions to the input parameter to derive a sequence
  - *[square* ***• 1l, length, 1r****]* : *<4, 5, 6>* → <16, 3, 6>
- Insert: inserts a dyadic operator *f* between elements
  - /+:<1, 2, 3> is equivalent to 1 + 2 + 3 = 6
  - Formal definition of insert function is recursive
    - **/f:<x> → x and /f : <x1, …, xN> → f: <x1, /f: <x2, … xN>>**
  - **average** ≡ divide • [/+, length]
- Apply-all: applies a function to all elements of a sequence and returns a sequence
  - αsquare: <1, 2, 3> → <1, 4, 9>

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Functional Forms II

- Condition functional form: similar to if-then-else
  - Format is (**if** *<predicate>* *<then-function>* *<else-function>*)
  - (**if** >•[**Id**, 0] Id * • [-1, **Id**]) is equivalent to
    if (n > 0) **return** x **else return** (- n)
- Iterative functional-form : similar to while loop
  - Format is *(**while** <predicate> <function>)*
  - The function is applied every time the predicate returns true, and the intermediate input in the next iteration is altered to <function>: <input>. If the predicate returns false then output becomes the current input
  - factorial ≅ **2l** • (**while** > • [**1l**, 0] [− • [**1l**, 1], * • [**1l**, **2l**]]) • [**Id**, 1]
- Recursive form: use recursion to form functions
  - factorial ≅ (**if** **>= •**[**Id**, 1] * • [**Id**, factorial • − • [**Id,** 1]] 1)

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Programming in FPP

- Simple sort by finding maximum value

```
minimum ≅ /<
delete ≅  (if  =•[2l, 1l•1r]     tl•1r          1r)
 • (while  and •  [>• [1l, 0],  not • =•[2l, 1l•1r]]    [ − • [1l, 1], 2l, rotl•1r ])
 • [length•2l, 1l, 2l]
sort ≅  (if  null   < >    apndl • [minimum,  sort • delete • [minimum, Id ]])
```

- Adding two matrices

```
group  ≅  (if  null • 1l  <>  apndl • [[1l • 1l, 1l • 1r], group •  [tl • 1l, tl • 1r]])
add-row  ≅  α+ • group
add-matrix ≅  (if  and • [= • [length • 1l,  length • 1r],
                     = • [length • 1l • 1l, length • 1l • 1r]  α add-row • group)
```

λ-expression	FPP
1. λ-expressions use variables. Variables are convenient.	1. FPP does not use variables
	2. FPP uses functional forms.
2. λ-expression uses nesting.	3. FP allows naming for callable functions

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142 Slide 13
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Functional Programming Languages

- Classification of functional programming languages
  - Pure functional programming languages
  - That support mutative objects and destructive updates
  - That integrate functional programming with object-oriented programming
  - That support concurrent programming
  - Multiparadigm languages
- <u>Examples</u>
  - Haskell is a pure functional programming language
  - LISP, Scheme and ML mix up imperative programming paradigm with functional programming paradigm
  - Ruby, Scala, and Emerald mix up object-oriented programming with functional programming paradigm
  - Scala and Ruby are multiparadigm languages that support functional programming, concurrent programming, and object-oriented programming.

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142 Slide 14
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Functional Programming Abstractions

- Pure functional programming
  - Does not support destructive updates and global variables
  - Data structures are immutable and assign once
  - Control abstractions based upon destructive updates of index variables such as conventional for-loop are missing
  - Programming uses more recursive style programming
  - Tail recursion is used to simulate iteration
  - Control abstractions are mainly based upon lists
  - Supports functions as first class objects
- Facilities in later functional programming languages
  - Iterators are used instead of index based iteration
  - Limited destructive update in global variables and array based operations
  - Multiparadigm languages like Ruby allow both mutable and immutable variables and mutable dynamic arrays

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142 Slide 15
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Data Abstractions

- Functional programming mainly uses sequences
  - Lisp uses linked-lists in early days to implement sequences
  - All the languages use immutable assign-once variables
- Lisp family of languages (Lisp, Scheme) use linked-lists, arrays, association-lists, global variables and frames
- CLOS, Scala and Ruby support OOP
- Haskell also supports tuples
- Functional languages inherently support polymorphism
  - ML, Hope and Scala are strongly typed polymorphic languages
- Haskell also supports modules

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142 Slide 16
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Control Abstractions

- Support for kernel functions
  - Almost all languages support the kernel function features of FPP
  - Some languages do not support explicit rotation
- Support for functional forms
  - All support *composition, apply-all, conditionals, iteration*, and *recursion.*
  - Lisp family supports Mapcar that is same as apply-all.
  - All functional programming languages support iterators.
  - Functional forms *insertion* and *construction* are simulated.
  - Lisp family and Ruby support *indefinite and definite iteration*, and *iterators*.
  - Ruby and Scala support while-loop.
- Evaluation strategy: eager evaluation or lazy evaluation
  - Lisp uses applicative eager evaluation
  - Haskell uses call by need and lazy evaluation
  - Functional languages uses three techniques to perform I/O operations: 1) *stream based IO*; 2) *continuation based IO*; and 3) *monads*.
  - *Monads* are abstractions for side-effect based I/O programming.
  - *Continuation based IO* refers to read and write operation as a transaction

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Programming in LISP Family

- **Lisp / Scheme use s-expression**
  - *(function Arg$_1$, …., Arg$_N$)*
  - Data uses a quote
  - Use variables instead of identity function or constant functions
- **LISP vs. Scheme**
  - Minor syntactic differences such as 'defun' and 'define'
  - Scheme is statically scoped
- **LISP program explanation**
  - *construction* is recursive
  - calls itself with rest of FuncList
  - *cons* concatenates
- **Scheme program explanation**
  - apply_all is recursive
  - Apply applies function one element at a time

LISP Program

```
(defun add5( Value)  (+  5   Value))
(defun square( Value)  (*  Value  Value))
(defun constr( FuncList    Argument)
   (if  ( null   FuncList)   nil
      (cons  (apply   (first   FuncList)
                       (list   Argument))
       (const  (rest    FuncList)  ArgsList)))
     )
)
```

Scheme Program

```
(define apply_all(MyFunc     ArgsList)
   (if   (null  ArgsList)       nil
       (cons  (apply MyFunc  (list ( first
                              ArgsList)))
                  (apply_all    MyFunc ( rest
                               ArgsList)))
        )
)
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

More LISP Examples I

```
(defun   greet ( )  (print  "Hello  World"))  ; defining a function greet
(defun   add5 (x) (+ x 5)) ; defining a function
(defun   square(x)  (*   x  x))
(setq   m   5)  ; setting the value of a global variable m to 5
(setq   p  '(4    "Hello World"))  ; binding a global variable p to a list
(setq   q   (*  (first    p)   m))
;  Implementing apply-all using mapcar
(defun   add5_to_all (ArgList) (mapcar  'add_5  ArgList))
(defun   square_and_add( x ) (add5  (square    x )))   ; composition
(defun  hypotenuse (x   y)  (sqrt   (+   (square  x)  (square  y))))
; Recursive  definition of  factorial using if-then-else
(defun factorial(n) (if  (=   n    0)     1     (*    n    ( factorial    (-   n    1)))))
; Recursive definition using conditional statement
(defun  my_sum(DataList)
          (cond    ( (null    DataList)   nil)
                     (t    (+  (first     DataList)  (my_sum  (rest      DataList))))
          )
)
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

More LISP Examples II

```
; apply-all  using mapcar to add two sequences
(defun   add_row (Seq1, Seq2)
  (mapcar   '+   Seq1   Seq2)  ; add corresponding elements of Seq1 and Seq2
; recursion with multiple arguments to add two matrices
(defun    add_matrix [(Matrix1, Matrix2)
     (if   (null Matrix1)   nil    (cons (add_row  (first Matrix1)   (first  Matrix2))
        (add_matrix   (rest  Seq1)  (rest Seq2)))
           )
)
; printing using dolist
(defun    print-matrix (Matrix)
          (dolist    (V Matrix) (print   V)))  ; iterate until the rows are consumed
;printing using dotimes
(defun print-matrix (Matrix)
          (let ((size (length Matrix)))  ; variable size = length of the matrix
              (dotimes (Index size) (print (nth  Index Matrix)))
          )
)
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Programming in Hope

- Hope is
  - Polymorphic language
  - Integrates functional programming and pattern matching
  - Uses type variables
- Pattern matching is between RHS call and LHS of the rule
- Control abstractions
  - If-then-else
  - While loop and recursion
  - Higher order functions
- **First program appends two sequences**
- **Second program implements apply_all**

```
typevar alpha
dec append: list(alpha)  #
          list(alpha)  → list(alpha)
;'#' is Cartesian product
append(nil, Ys) <=  Ys.
append(x :: Xs, Ys) <=
          x :: append(Xs, Ys).
; :: means concatenation

dec apply_all : list ( num ) #
( num → num ) → list ( num )
apply_all ( nil, function ) <= nil ; base case
apply_all ( first :: rest, function ) <=
function ( first ) :: apply_all( rest, function)
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Programming in Haskell

- Statically typed type-safe
  - Supports parametric polymorphism
- Programs divided into modules
  - Module name same as file name
  - Functions private to modules unless exported
  - Functions in the form LHS = RHS. LHS is function name followed by parameters
- Supported control abstractions
  - Composition, iteration, recursion, insert, conditionals, and apply_all
  - (square.add5) x ≡ 'square • add5
  - map (add5) [1, 2, 3] → [6, 7, 8]
- Supported data abstractions
  - Lists within square brackets
  - Expressions in infix form
  - Tuples are in parenthesis
  - Fst is first; snd is second;
- Comments are written as {- -}.

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Haskell Programming I

```
module main  where
main   =   putStrLn "Hello World" {-
add5   x = x + 5  {- Add 5 to parameter -}
square    x = x * x  {-  return square of number -}
m = 5  {- assigning a value to a variable   -}
p = (4, "Hello World")   {- assign    tuple -}
q = fst p  *  m
hypotenuse  :: Float → Float → Float
hypotenuse  x  y   = sqrt   (square x  + square y)
square_and_add   x = (add5.square)    x
add5_to_all   x  =  map   (add5)    x
{- finding factorial using case statement   -}
factorial n = case n of
      { 1 -> 1;  {- base case   -}
        _ ->  n * factorial (n - 1)
      }
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Haskell Programming II

```
factorial1 1 = 1  {-    base case  -}
factorial1 n = n * factorial1 (n-1)  {- recursive definition of factorial   -}
{-  finding factorial using if-then-else   -}
factorial2 n = if n == 0 then 1 else n * factorial ( n - 1)
{-  Guards   -}
my_minimum    x    y   | x   <=  y  = x
                       | y    <=   x  = y
{-   recursive programming and concatenation at the end   -}
my_reverse [ ] = [ ]   {- base case   -}
my_reverse  (x : xs) = my_reverse   xs ++ [x]  {-  '++' adds at the end   -}
{-     Recursive programming with multiple arguments    -}
add_row  [ ] [ ] = [ ]   {- base case -}
add_row (x : xs) (y : ys) = (x + y : add_row   xs   ys)
add_matrix [ ] [ ] =  [ ]  {- base case   -}
add_matrix (r : rs) (w : ws)  = (add_row  r  w : add_matrix   rs  ws)
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Scala Language

- Scala integrates functional and object oriented programming
  - Treats every value as an object
  - Statically typed type-safe language; supports polymorphism
  - infers type if not declared.
  - Block structured language
- Scala is built on top of Java
  - Programs are compiled to Java bytecode.
  - Supports both mutable and immutable data structures.
- Data abstractions
  - Supports arrays, associative maps, lists, tuples and sets
  - Sets and associative maps can be used in both mutable and immutable way using *traits* - abstract interfaces extending class of the data objects
  - Creating array: *val studentNames = new array[String](20).*
  - Creating lists: *List(1, 2, 3); List*(*1, 2*) **:::** *List*(*3, 4, 5*) derives *List*(*1, 2, 3, 4, 5*).
- Control abstractions
  - supports *if-then-else, case statement, while-loop, do-while-loop, iterator foreach-loop, definite iteration for-loop*, and recursive function calls.
  - Supports destructive update in index variables.

Examples of Scala I

```
val x = 2 + 3  // declare a global variable
println("Hello World")
def add5(n: Int): Int = {n + 5}  // Add 5
def square(x: Double): Double = {x * x}  // square using double_float
def int_square(x: Int): Int = {x * x}
def square_add(x: Int): Int = int_square(add5(x))  // Composition
def power_rec(x: Double, n:Int): Double =
    { if  (n == 0)  1
      else x * power_rec(x, n-1)}
def power_iter(x : Int, n : Int) : Int =   //iterative version of power
    { var a = n;   var b = 1;
      while (a > 0 ) {b = x * b; a = a - 1}
            b // return final value of b
    }
```


Examples of Scala II

```
def sum_list(xs:List[Int]):  Int =
{   if (xs.isEmpty)   0
    else xs.head  +  sum_list(xs.tail)
}
def add_rows(xs : List[Int],   ys : List[Int]) : List[Int] =
    {   if (xs.isEmpty) Nil
        else xs.head  +   ys.head ::
               add_rows(xs.tail,   ys.tail)
}
def apply_all(my_func:Int => Int,  xs:List[Int]) : List[Int] =  {xs  map my_func}
def construction(my_funcs : List[Int => Int], n:Int) : List[Int] =
    { if (my_funcs.isEmpty)  Nil
     else my_funcs.head(n) :: construction(my_funcs.tail,   n)
    }
```


Ruby

- Integrates imperative, object oriented and functional paradigm
- Used as a scripting language (to be discussed later)
- Supports mutable objects and immutable objects
- Is a dynamically typed polymorphic language
  - Supports integers, floating point, strings, indexible sequences, sets, hash tables, and class. Indexible sequences are dynamic arrays or vectors
  - *multiarr A = [[a, b, c], [1, 2, 3], ["Hello", "There", "Friends "]*
  - Rich library to manipulate matrices; '+' can add two matrices
  - Strings are treated as indexible sequence
- Control abstractions
  - Nested blocks, parallel assignment, if-then-else, unless (opposite semantics compared to if-then-else), case statement, for-loop, while-loop, until-loop (equivalent to repeat until), a loop-construct that needs a conditional exit, multiple syntax for iterators, recursion, explicit λ-expressions and function calls. Uses 'def' to define a function
  - Supports multithreading and exception handling

Ruby Programming I

```
def greet  # illustrating function
puts("Name:"); gets(Name ); puts("Hello "  +  Name)
end
m = [[1, 2, 3], ['a', 'b', 'c']]  # Array has different types of objects
m = "cat";  n = 4;  m1, m2  = m2, m1  # parallel assignment

def factorial(n)  # illustrating recursion and if-then-else
        if (n == 0) then 1
        else n * factorial(n – 1)
        end
end
def fibonacci(n)  #   Illustrates the syntax of case statement
        case (n)
                when 0 then 1
                when 1 then 1
                else fibonacci(n – 1) + fibonacci(n – 2)
        end
end
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Ruby Programming II

```
def sum_seq(xs)  # illustrating iterators and destructive update
        accumulator = 0
        for n in xs do   accumulator = accumulator  * n    end
        return acc
end
def append(xs, ys)  # appends two sequences
        zs = xs + ys
end
def add_seq(xs, ys)
        zs = Array.new  # creating a dynamic array
        length1 = xs.length - 1
        for n in 0..length1  # another form of iterator
                zs[n] = xs[n] + ys[n]  # expanding dynamic array
        end
        return zs
end
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Ruby Programming III

```
def add_matrix(m1, m2) # use of while-loop
  m_final = Array.new ;  size = m1.length ; index = 0
  while (index < size)                # while loop
        m_final[index] = add_seq(m1[index], m2[index])  # function call
        index += 1
  end
  return m_final
end
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Implementation Models

- Applicative order reduction
  - Uses eager evaluation - evaluates parameters eagerly before passing the parameters
  - LISP is a language that uses AOR
  - A popular implementation model for eager evaluation is SECD machine
  - SECD machine has four stacks: 1) evaluation stack; 2) environment; 3) command string; 4) dump. Dump stores the environment of the calling functions
- Normal order reduction
  - Uses lazy evaluation – delays evaluation of parameter expression
  - Haskell is a language that uses NOR
  - Uses call by need to improve efficiency
  - A popular implementation model for lazy evaluation is graph reduction.
  - ABC machine is used to implement graph reduction

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

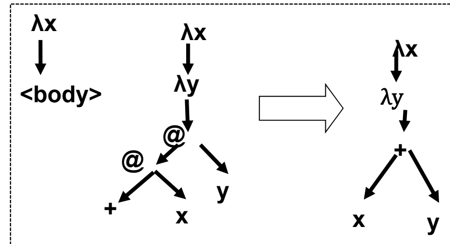
SECD Machine

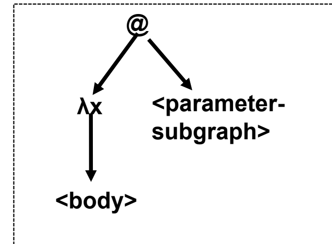
- SECD machine is a state transition machine with 4 stacks
  - Set of states is the Cartesian product of all possible sets of four stacks
  - β-reduction: (id ↦ Value) goes into the environment stack E
  - δ-reduction: done on the expression stack S by looking at operator on C
  - Upon a function call, triple (S, E, C) is dumped on stack D
- State transitions based upon input symbol
  - *<literal>: new state becomes (<literal>* **::** *S, E, rest(C), D).*
  - *Identifier X: new state is (value-of(X)* **::** *S, E, rest(C), D).*
  - λ-expression [<bound-variables>, <body>]: new state is *([<bounded-variables>, <body>, E]* **::** *S, E, C, D) closure on top of S*
  - *Closure on top of S:* state transition is given by *([bounded-variables, body, E_1]* **::** *<args>* **::** *rest(S), E, C, D) → (nil, {bounded_variables → <args>} ⊕ E_1, [<body>], (rest(S), E, C)* **::** *D)*
  - *<kernel> <args>* : new state is *(eval(<kernel>(<args>))* **::** *rest(S), E, C, D).*
  - *Top(c) == apply (<func>, <args>)* new state is *(S, E, <args>* **::** *<func>* **::** *@* **::** *rest(C), D).*
  - *nil: (<result>* **::** *rest(S), E, C, (S^{pre}, E^{pre}, C^{pre})* **::** *D^{pre}) → (result*::*S^{pre}, E^{pre}, C^{pre}, D^{pre})* where *(Closure* **::** *<args>* **::** *S^{pre}, E^{pre}, C^{pre}, D^{prev})*
  - *Conditional form <predicate> cond <func1> <func2> @ skip <func2> if true*

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Graph Representation

- Uses directed graphs to model expressions
  - Shared variables are modeled as single nodes
  - The nodes can be bounded variable, operator, or apply-node '@'.
  - Common subexpressions are subgraphs excessed using pointers
  - λ-expression *λx. <body>* represented as a tree with edges from declared variable x to the body
- λ-expression with parameter
  - Left subtree of apply node is function; right subtree is argument (Fig. 2)
  - Traverses the expression-graph until apply node





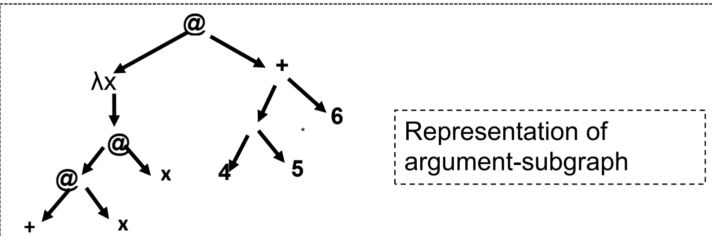
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Graph Reduction

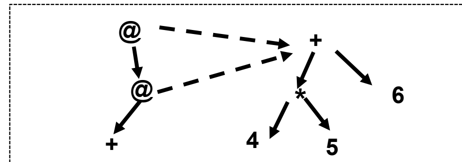
- There are four valid possibilities for the graph G
  - Single node atomic data object;
  - Another λ-expression;
  - Composite tuple with n≥ 1
  - Primitive function of arity k.
- β-reduction and δ–reductions are needed in two cases
  - G is a primitive function
  - G is a user defined λ-expression
- β-reduction using graphs
  - Short-circuit the edges to the body of λ-expressions
  - Connect nodes of the substituted variables to the argument-subgraph
  - Multiple occurrence of variables need multiple edges
- δ–reduction
  - Argument subgraph is reduced, and the value is passed to the nodes connecting to the argument subgraph
  - δ-reduction of the parameter expression behaves like call-by-need
  - Only one evaluation for multiple subexpressions, and the value is passed to all the nodes connected to the reduced value node

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

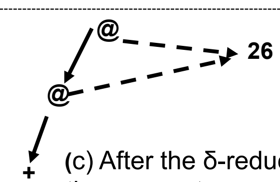
Example of Subgraph Reduction



(a) Original graph representation for λx. (x + x) 4 * 5 + 6



(b) After β-reduction . Broken pointers point to the same argument-subgraph



(c) After the δ-reduction of the parameter-expression

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

ABC Machine

- ABC Machine implements graph reduction for NOR reduction
  - ABC machine is also a state transition machine
  - Program is translated to a set of microinstructions that alters state
- Composition of ABC Machine
  - A graph store to store the graph to be rewritten
  - A program store to store instructions
  - A-stack to store reference to the graph nodes
  - B-stack to handle reduction of basic values
  - C-stack like traditional control stack
  - A descriptor store to translate the coded value to actual symbol
  - An I/O channel to display the results
- Microinstructions classification
  - Get an instruction into a program store;
  - Increment and update the program counter;
  - Get a node, create a new node, delete a node and update a node value;
  - Extract information stored in a node;
  - Redirect an edge to another node
  - Get the description of a symbol from the descriptor-store

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Strictness Analysis

- Deferring evaluation in NOR reduction is called nonstrictness
  - Nonstrictness is lazy evaluation
  - Lazy evaluation is suitable for handling infinite data structures
- Issues in lazy evaluation
  - Substitution to multiple occurrence of subexpression is demand based
  - Lack of eager evaluation causes computational and memory overhead
  - Part of overhead is reduced by call by need
  - Part of overhead is reduced by strictness analysis
- Strictness analysis
  - A compile-time program analysis technique in abstract domain
  - It identifies complex subexpressions that can be evaluated first before parameter passing to reduce the overhead

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Concurrency in Functional Languages

- Concurrency can be exploited in functional languages by
  - Making parallel binding of the values to the bounded variables
  - By concurrently reducing the arguments
  - By spawning separate processes for conditionals
  - By spawning separate processes / threads for closures
- Concurrency in Lisp
  - Uses future to eagerly evaluate expressions in advance
  - Common lisp uses thread based library interface
- Concurrency in Haskel
  - Supports concurrency using forkio and Mvar
  - *MVar* is a shared box that is either full or empty. It can be used as lock, shared channel between two threads, or 3) asynchronous I/O
- Concurrency in Scala
  - Uses Java concurrency model and asynchronous message passing
- Concurrency in Ruby
  - Interpreter supports multithreading, multiprocessing, mutex locks for synchronization, conditional variables for waiting for resources while in a critical section, and pipelining

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

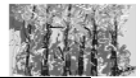
Summary I

- Functional programming is based on mathematical functions
- Pure functional programming does not support destructive update and global variables
- λ–expressions have three components: variable, expression and arguments
- Reduction involves two steps: α-reduction and β-reduction
- There are two reduction techniques: AOR and NOR.
- AOR reduces from innermost level outwards, and NOR reduces from the outermost level inwards
- AOR is suitable for eager evaluation and NOR is suitable for lazy evaluation
- Eager evaluation evaluates the arguments first before binding to the variables, and uses call by value
- Lazy evaluation defers the evaluation until needed, and uses call by need
- FPP is a variableless way of writing functions
- FPP has kernel predicates and functional forms to form complex functions
- **Major limitation of functional programming is the lack of archiving partial computations, and excessive recursive programming**

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Summary II



- Lisp, Scheme, Haskell, Hope, Ruby and Scala support functional programming paradigm
  - Haskell and Scala are statically typed type safe programming languages
  - Ruby is a dynamically typed language
- Functional programming languages have been implemented using SECD machine, G-machine and ABC machine
  - SECD machine uses eager evaluation and is a 4-stack machine
  - ABC machine is a graph reduction machine, and uses microinstructions
- Graph reduction uses lazy evaluation and is used in Haskell
- Functional programming is integrated with various paradigms
  - Imperative, concurrent, object oriented, and logic programming
- Concurrency has been exploited in
  - Concurrent evaluation of arguments, future evaluation of expressions, concurrent evaluation of conditionals, concurrent execution of closure

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved