Case Study
Functional Programming

Outline

Overview of Functional Languages

- They emerged in the 1960's with Lisp
- Functional programming mirrors *mathematical functions*: domain = input, range = output
- *Variables* are mathematical *symbols*: not associated with memory locations.
- Pure functional programming is *state-free*: no assignment
- *Referential transparency*: a function's result depends only upon the values of its parameters.

SCHEME

- A derivative of Lisp
- Our subset:
  - omits assignments
  - simulates looping via recursion
  - simulates blocks via functional composition
- Scheme is Turing complete, but
- Scheme programs have a different flavor

Expressions

- Cambridge prefix notation for *all* Scheme expressions:

```
(f x1 x2 … xn)
```

- E.g.,

```
(+ 2 2)                           ; evaluates to 4
(+ (* 5 4) (- 6 2))               ; means 5*4 + (6-2)
(define (Square x) (* x x))       ; defines a function
(define f 120)                    ; defines a global
```

- *Note*: Scheme comments begin with ";"

Expression Evaluation

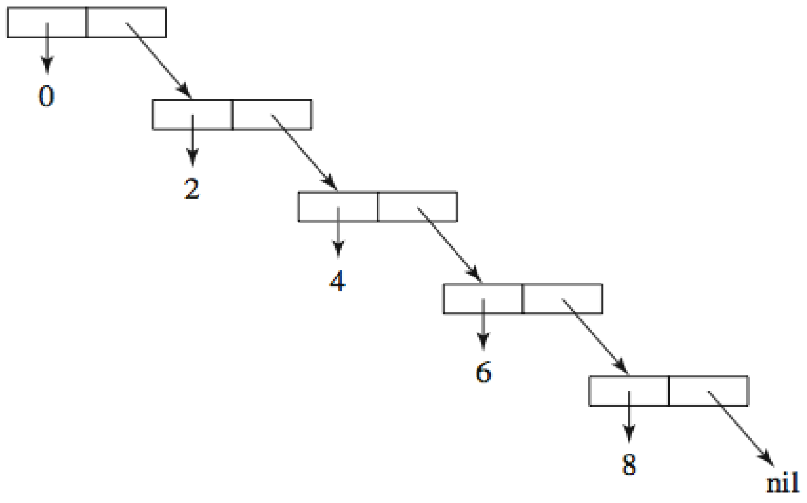
- Three steps:
  1. Replace names of symbols by their current bindings.
  2. Evaluate lists as function calls in Cambridge prefix.
  3. Constants evaluate to themselves.

E.g.,

```
x                       ; evaluates to 5
(+ (* x 4) (- 6 2))     ; evaluates to 24
5                       ; evaluates to 5
‘red                    ; evaluates to ‘red
                        ; ‘ prevents lists or symbol being eval’ed
```

Lists

- A *list* is a series of expressions enclosed in parentheses.
  - Lists represent both functions and data.
  - The empty list is written ().
  - E.g., (0 2 4 6 8) is a list of even numbers.
  - List is actually a pair: (head, tail), where tail is another list.
  - Here's how it's stored:



List Transforming Functions

- Suppose we define the list *evens* to be (0 2 4 6 8). i.e., we write (define evens ‘(0 2 4 6 8)). Then:

```
(car evens)                  ; gives 0
(cdr evens)                  ; gives (2 4 6 8)
(cons 1 (cdr evens))         ; gives (1 2 4 6 8)
(null? ‘())                  ; gives #t, or true
(equal? 5 ‘(5))              ; gives #f, or false
(append ‘(1 3 5) evens)      ; gives (1 3 5 0 2 4 6 8)
(list ‘(1 3 5) evens)        ; gives ((1 3 5) (0 2 4 6 8))
```

- *Note*: the last two lists are different!

Elementary Values

- Numbers
  - integers
  - floats
  - rationals
- Symbols
- Characters
- Functions
- Strings

```
(list? evens)
(symbol?  'evens)
```

Control Flow

- Conditional

```
(if   (< x 0)  (- 0 x))                    ; if-then
(if  (< x y)  x  y)                        ; if-then-else
```

This is similar to “test” function in lambda calculus.

- Case selection

```
(case month
        ((sep apr jun nov)  30)
        ((feb)    28)
        (else     31)
)
```

Defining Functions

- (define (*name arguments*) *function-body*)

```
(define (min x y) (if (< x y) x y))
(define (abs x) (if (< x 0) (- 0 x) x))

(define (factorial n)
        (if (< n 1) 1 (* n (factorial (- n 1)))
))
```

- *Note*: be careful to match all parentheses.

The subst Function

```
(define (subst y x alist)
    (if (null? alist) ‘()
          (if (equal? x (car alist))
                (cons y (subst y x (cdr alist)))
                (cons (car alist) (subst y x (cdr alist)))
          )
    )
)
```

- E.g., (subst ‘x 2 ‘(1 (2 3) 2))

  returns (1 (2 3) x)
- Only compare at the top level of the list!

Let Expressions

- Allows simplification of function definitions by defining intermediate expressions. E.g.,

```
(define (subst y x alist)
        (if (null? alist) ‘()
                (let ((head (car alist)) (tail (cdr alist)))
                        (if (equal? x head)
                            (cons y (subst y x tail))
                            (cons head (subst y x tail))
)))
```

Functions as arguments

```
(define (mapcar fun alist)
        (if (null? alist) ‘()
                (cons (fun (car alist))
                        (mapcar fun (cdr alist)))
))
```

- E.g., if (define (square x) (* x x)) then

  ```
  (mapcar square ‘(2 3 5 7 9)) returns
          (4 9 25 49 81)
  ```

Haskell

- A more modern functional language
- Many similarities with Lisp and Scheme
- Key distinctions:
  - Lazy Evaluation
  - An Extensive Type System
  - Cleaner syntax
  - Notation closer to mathematics
  - Infinite lists

Introduction

- Minimal syntax for writing functions. E.g.,

```
-- two equivalent definitions of factorial
fact1 n = if n==0 then 1 else n * fact1(n-1)
fact2 n
        | n==0          = 1
        | otherwise     = n * fact2(n-1)
```

- *Note*: Haskell comments begin with --

- Infinite precision integers:

```
> fact2 30
> 265252859812191058636308480000000
```

Expressions

- Infix notation. E.g.,

```
5 * (4+6) - 2          -- evaluates to 48
5 * 4^2 - 2            -- evaluates to 78
```

- Or prefix notation. E.g.,

```
(-) ((*) 5 ((+) 4 6)) 2
```

- Many operators:

```
! !! // ^ **
* / `div` `mod` `rem` `quot`
+ -  :
/= < <= == > >= `elem`
&& ||
```

Lists and List Comprehensions

- A *list* is a series of expressions separated by commas and enclosed in brackets.

  The empty list is written [].

  evens = [0, 2, 4, 6, 8] declares a list of even numbers.

  evens = [0, 2 .. 8] is equivalent.

- A *list comprehension* can be defined using a *generator*:

  ```
  moreevens = [2*x | x <- [0..10]]
  ```

  - The condition that follows the vertical bar says, "all integers x from 0 to 10." The symbol `<-` suggests set membership (∈).
  - What's `moreevens`?
  - Answer: [0, 2, 4, 6 .. 20]
  - Looks exactly like the mathematically definition of *sets*.

Infinite Lists

- Generators may include additional conditions, as in:

```
factors n = [f | f <- [1..n], n `mod` f == 0]
```

- This means "all integers from 1 to n that divide f evenly."
- List comprehensions can also be infinite. E.g.:

```
mostevens = [2*x | x <- [0,1..]]
mostevens = [0,2..]
```

List Transforming Functions

- The operator : concatenates a new element onto the head of a list. E.g., 4:[6, 8] gives the list [4, 6, 8].
- Suppose we define evens = [0, 2, 4, 6, 8]. Then:

```
head evens                      -- gives 0
tail evens                      -- gives [2,4,6,8]
head (tail evens)               -- gives 2
tail (tail evens)               -- gives [4,6,8]
tail [6,8]                      -- gives [8]
tail [8]                        -- gives []
```

List Transforming Functions

- The operator ++ concatenates two lists.
  E.g., [2, 4]++[6, 8] gives the list [2, 4, 6, 8].

- Here are some more functions on lists:

```
null []                -- gives True
null evens             -- gives False
[1,2]==[1,2]           -- gives True
[1,2]==[2,1]           -- gives False
5==[5]                 -- gives an error (mismatched args)
type evens             -- gives [Int] (a list of integers)
```

Elementary Types and Values

Numbers	
integers	types Int (finite; like int in C, Java) and Integer (infinitely many digits)
floats	type Float
Numerical Functions	abs, acos, atan, ceiling, floor, cos, sin, log, logBase, pi, sqrt
Booleans	type Bool; values True and False
Characters	type Char; e.g., \`a\`, \`?\`
Strings	type String = [Char]; e.g., “hello”

Control Flow

- Conditional

```
if x>=y && x>=z then x
else if y>=x && y>=z then y
        else z
```

- Guarded command (used widely in defining functions)

```
| x>=y && x>=z        = x
| y>=x && y>=z        = y
| otherwise           = z
```

-- Borrowed from Dijkstra’s guards.

Defining Functions

- A Haskell Function is defined by writing:
  - its *prototype* (name, domain, and range) on the first line
  - its *parameters and body* (meaning) on the remaining lines.

```
max3 :: Int -> Int -> Int -> Int    -- "Curry" form
max3 x y z
        | x>=y && x>=z              = x
        | y>=x && y>=z              = y
        | otherwise                 = z
```

- *Note*: if the prototype is omitted, Haskell interpreter will *infer* it.

FUNCTIONS ARE POLYMORPHIC

- Omitting the prototype gives the function its most *general* meaning. E.g.,

```
max3 x y z
      | x>=y && x>=z            = x
      | y>=x && y>=z            = y
      | otherwise               = z
```

is now well-defined for any argument types:

```
> max3 6 4 1
6
> max3 “alpha” “beta” “gamma”
“gamma”
```

Because >= is an ad-hoc polymorphic operator

Tuples

- A *tuple* is a collection of values of different types. Its values are surrounded by parens and separated by commas. E.g., (“Bob”, 2771234) is a tuple.
- Tuple types can be defined by the types of their values. E.g.,

```
type Entry = (Person, Number)
type Person = String
type Number = String
```

- And lists of tuples be defined as well:

```
type Phonebook = [(Person, Number)]
```

Functions on Tuples

- Standard functions on tuples (first and second members):

  fst (“Bob”, 2771234) returns “Bob”

  snd (“Bob”, 2771234) returns 2771234

- We can also define new functions like find to search a list of tuples:

```
find :: Phonebook -> Person -> [Number]
find pb p = [n | (person, n) <- pb, person == p]
```

- For instance, if:

```
pb = [(“Bob”, 2771234), (“Allen”, 2772345),
                                    (“Bob”, 2770123)]
```

then the call find pb “Bob” returns all of Bob’s phone numbers:

[2771234, 2770123]

Functions as arguments

- Here is a function that applies another function to every member of a list, returning another list.

```
maphead :: (a -> b) -> [a] -> [b]
maphead f alist = [ f x | x <- alist ]
```

- E.g., if square x = x*x then

```
maphead square [2,3,5,7,9]
```

returns

```
[4,9,25,49,81]
```

Example: Semantics of Simple C

- Program state can be modeled as a list of pairs.

```
type State = [(Variable, Value)]
type Variable = String
data Value = Intval Integer | Boolval Bool
                        deriving (Eq, Ord, Show)
```

E.g.,

```
[(“x”, (Intval 1)), (“y”, (Intval 5))]
```

- Note: difference between “type” and “data”:
  - type: defines a “type synonym” – not really a new type
  - Data: defines a new “algebraic type” – often variant types.
- Function to retrieve the value of a variable from the state:

```
get var (s:ss)
            | var == (fst s)       = snd s
            | otherwise            = get var ss
```

State transformation

- Function to store a new value for a variable in the state:

```
        onion :: Variable -> Value -> State -> State
        onion var val ([]) = [(var, val)]
        onion var val (s:ss)
                | var == (fst s)            = (var, val) : ss
                | otherwise                 = s : (onion var val ss)
```

- E.g.,

```
onion ‘y’ (Intval 4) [(‘x’, (Intval 1)), (‘y’, (Intval 5))]
  = (‘x’, (Intval 1)) : onion ‘y’ (Intval 4) [(‘y’, (Intval 5))]
  = [(‘x’, (Intval 1)), (‘y’, (Intval 4))]
```

Modeling Simple C Abstract Syntax

```
data Statement = Skip | Assignment Target Source |
                 Block   [ Statement ] | Loop Test Body |
                 Conditional Test Thenbranch Elsebranch
                 deriving (Show)
type Target = Variable
type Source = Expression
type Test = Expression
type Body = Statement
type Thenbranch = Statement
type Elsebranch = Statement
```

Semantics of Statements

- A statement transforms a state to another state:

```
m :: Statement -> State -> State
```

- Skip is easy!

```
m (Skip) state = state
```

- Assignments aren't too bad either:

```
m (Assignment target source) state
= onion target (eval source state) state
```

Loops and Conditionals

Loops are a bit trickier:

```
m (Loop t b) state
| (eval t state) == (Boolval True)
                        = m (Loop t b) (m b state)
| otherwise             = state
```

-- *eval* evaluates an expression to a value

Conditionals are straightforward:

```
m (Conditional test thenbranch elsebranch)
| (eval test state) == (Boolval True)
                                    = m thenbranch state
| otherwise                         = m elsebranch state
```

Expressions

```
data Expression = Var Variable | Lit Value |
                  Binary Op Expression Expression
                  deriving (Eq, Ord, Show)
type Op = String
```

The meaning of an expression is a value, delivered by the function:

```
    eval :: Expression -> State -> Value
    eval (Var v) state = get v state
    eval (Lit v) state = v
```

Example: Symbolic Differentiation

- Symbolic Differentiation Rules

$$\frac{d}{dx}(c) = 0$$ *c* is a constant

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx}$$ *u* and *v* are functions of *x*

$$\frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}(u/v) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right)/v^2$$

Haskell Encoding

- Uses Cambridge Prefix notation

  E.g., 2x + 1 is written:

  ```
  (Add (Mul (Num 2) (Var “x”)) (Num 1))
  ```

- Function diff incorporates these rules. E.g.,

  ```
  diff “x” (Add (Mul (Num 2) (Var “x”)) (Num 1))
  ```

  should give an answer.

- However, no simplification is performed.
- E.g. the answer for the above is:

  ```
  Add (Add (Mul (Num 2) (Num 1))
                 (Mul (Var “x”) (Num 0))) (Num 0))
  ```

Haskell Program

```
data Expr = Num Int | Var String | Add  Expr Expr |
                  Sub Expr Expr | Mul Expr Expr |
                  Div Expr Expr deriving (Eq, Ord, Show)
diff :: String -> Expr -> Expr

diff x (Num c) = Num 0
diff x (Var y) = if x == y then Num 1 else Num 0
diff x (Add u v) = Add (diff x u) (diff x v)
diff x (Sub u v) = Sub (diff x u) (diff x v)
diff x (Mul u v) = Add (Mul u (diff x v))
                  (Mul v (diff x u))
diff x (Div u v) = Div (Sub (Mul v (diff x u))
                  (Mul u (diff x v))) (Mul v v)
```

Trace of the Example

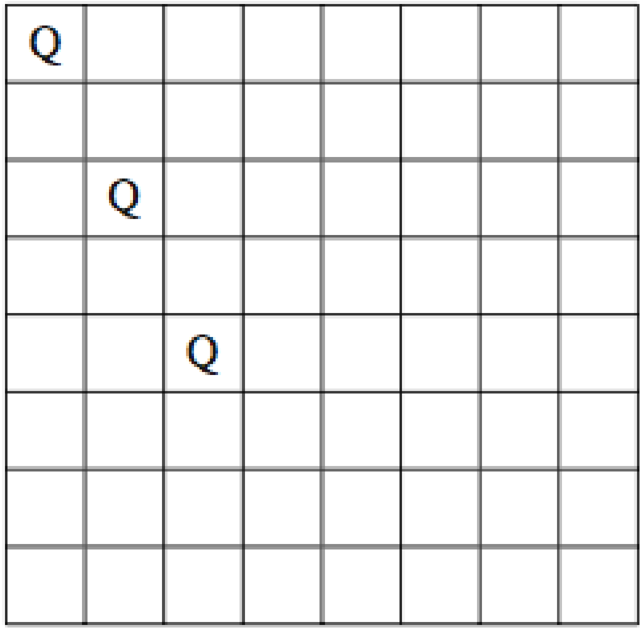
```
diff “x” (Add (Mul (Num 2) (Var “x”)) (Num 1))
        = Add (diff “x” (Mul (Num 2) (Var “x”)))
                  (diff “x” (Num 1))
        = Add (Add (Mul (Num 2) (diff “x” (Var “x”)))
                            (Mul (Var “x”) (diff “x” (Num 2))))
                   (diff “x” (Num 1))
        = Add (Add (Mul (Num 2) (Num 1))
                             (Mul (Var “x”) (Num 0)))
                   (diff “x” (Num 1))
         = Add (Add (Mul (Num 2) (Num 1))
                              (Mul (Var “x”) (Num 0)))
                   (Num 0)
```

Example: Eight Queens

- A backtracking algorithm for which each trial move's:

1. Row must not be occupied,
2. Row and column's SW diagonal must not be occupied, and
3. Row and column's SE diagonal must not be occupied.

If a trial move fails any of these tests, the program backtracks and tries another. The process continues until each row has a queen (or until all moves have been tried).



Representing the Developing Solution

- Positions of the queens are in a list whose nth entry gives the row position of the queen in column n, in reverse order. Row and column numbers are zero-based.
- E.g., the list [0,2,4] represents queens in (*row, col*) positions (0,0), (2,1), and (4,2); i.e., see earlier slide.
- A safe move can be made in (*row, col*) if
  1. The trial *row* is not in the existing solution list, and
  2. The southwest and southeast diagonals are unoccupied.
- For trial row q and existing solution list b, these conditions are (b!!i means b[i]):
  1. q /= b!!i for each i from 0 to length b -1.
  2. q - b!!i /= -i-1 and q-b!!i /= i+1 for each i.

The Program

```
-- Finds all solutions for an n x n board
queens n = solve n
        where
        solve 0 = [ [] ]
        solve (k+1) = [q : b | b <- solve k, q <- [0..(n-1)],
                                    safe q b ]
        safe q b = and [not (checks q b i) |
                  i <- [0..(length b - 1) ] ]
        checks q b i = q == b!!i || abs (q-b!!i) == i+1
```

Sample output

```
> Queens 0
[ [] ]                     --  no queens for a 0x0 board
> Queens 1
[ [0] ]                    -- one queen in position (0,0)
> Queens 2
[ [] ]                     -- no solutions for a 2x2 board
> Queens 3
[ [] ]                     -- no solutions for a 3x3 board
> Queens 4
[ [2,0,3,1],[1,3,0,2] ]    -- two solutions for a 4x4 board
```