Functional Programming in
Scheme

CSCE A331

Functional Programming

• Online textbook: http://www.htdp.org/
• Original functional language is LISP
  – LISt Processing
  – The list is the fundamental data structure
  – Developed by John McCarthy in the 60’s
    • Used for symbolic data processing
    • Example apps: symbolic calculations in integral and differential calculus, circuit design, logic, game playing, AI
    • As we will see the syntax for the language is extremely simple
  – Scheme
    • Descendant of LISP
Functional Languages

• “Pure” functional language
  – Computation viewed as a mathematical function mapping inputs to outputs
  – No notion of state, so no need for assignment statements (side effects)
  – Iteration accomplished through recursion

• In practicality
  – LISP, Scheme, other functional languages also support iteration, assignment, etc.
  – We will cover some of these “impure” elements but emphasize the functional portion

• Equivalence
  – Functional languages equivalent to imperative
    • Core subset of C can be implemented fairly straightforwardly in Scheme
    • Scheme itself implemented in C
    • Church-Turing Thesis

Lambda Calculus

• Foundation of functional programming
• Developed by Alonzo Church, 1941
• A lambda expression defines
  – Function parameters
  – Body
• Does NOT define a name; lambda is the nameless function. Below x defines a parameter for the unnamed function:

  \[(\lambda x \cdot x \ast x)\]
Lambda Calculus

• Given a lambda expression
  \((\lambda x \cdot x \cdot x)\)

• Application of lambda expression
  \(((\lambda x \cdot x \cdot x)2) \rightarrow 4\)

• Identity \((\lambda x \cdot x)\)

• Constant 2: \((\lambda x \cdot 2)\)

Lambda Calculus

• Any identifier is a lambda expression
• If \(M\) and \(N\) are lambda expressions, then the application of \(M\) to \(N\), \((MN)\) is a lambda expression
• An abstraction, written \((\lambda x \cdot M)\) where \(x\) is an identifier and \(M\) is a lambda expression, is also a lambda expression
Lambda Calculus

\[
\text{LambdaExpression} \rightarrow \text{ident} | (MN) | (\lambda \text{ ident} \cdot M)
\]

\[ M \rightarrow \text{LambdaExpression} \]
\[ N \rightarrow \text{LambdaExpression} \]

Examples

\[ x \]
\[ (\lambda x \cdot x) \]
\[ (((\lambda x \cdot x)(\lambda y \cdot y)) \]

Lambda Calculus

First Class Citizens

• Functions are \textit{first class citizens}
  – Can be returned as a value
  – Can be passed as an argument
  – Can be put into a data structure as a value
  – Can be the value of an expression

\[
((\lambda x \cdot x \cdot x)(\lambda y \cdot x)) = (\lambda x \cdot 2 \cdot 2) = 4
\]
\[
((\lambda x \cdot (\lambda y \cdot x+y)) \cdot 2 \cdot 1) = ((\lambda y \cdot x+y) \cdot 1) = 3
\]
Lambda Calculus

Functional programming is essentially an applied lambda calculus with built in
- constant values
- functions

E.g. in Scheme, we have (* x x) for x*x instead of \( \lambda x \cdot x^2 \)

Functional Languages

• Two ways to evaluate expressions
• Eager Evaluation or Call by Value
  – Evaluate all expressions ahead of time
  – Irrespective of if it is needed or not
  – May cause some runtime errors

• Example

(foo 1 (/ 1 x)) Problem; divide by 0
Lambda Calculus

• Lazy Evaluation
  – Evaluate all expressions only if needed
    (foo 1 (/ 1 x)) ; (/ 1 x) not needed, so never eval’d
  – Some evaluations may be duplicated
  – Equivalent to call-by-name
  – Allows some types of computations not possible in eager evaluation

• Example
  – Infinite lists
    • E.g., Infinite stream of 1’s, integers, even numbers, etc.
  – Replaces tail recursion with lazy evaluation call
  – Possible in Scheme using (force/delay)

Running Scheme for Class

• A version of Scheme called Racket
  (formerly PLT/Dr Scheme) is available on the Windows machines in the NSB/ENGR Labs

• Download: http://racket-lang.org/
• Unix, Mac versions also available if desired
Racket

- You can type code directly into the interpreter and Scheme will return with the results:
Make sure right Language is selected

Use the “Pretty Big” language choice – it is closer to Scheme than others

Welcome to DrRacket, version 5.2.1 [DrRacket]
Language: Beginner Student; memory limit: 128 MB
> (define factorial (lambda (n)
  (cond
    ((= n 1) 1)
    (else (* n (factorial (- n 1))))
  )
  )
)

Racket – Loading Code

• You can open code saved in a file. Racket uses the extension “.rkt” so consider the following file “factorial.rkt” created with a text editor or saved from Racket:

1: Open

2: Run

3: Invoke functions
Functional Programming Overview

• Pure functional programming
  – No implicit notion of state
  – No need for assignment statement
    • No side effect
  – Looping
    • No state variable
    • Use Recursion

• Most functional programming languages have side effects, including Scheme
  – Assignments
  – Input/Output

Scheme Programming Overview

- Refreshingly simple
  - Syntax is learned in about 10 seconds
- Surprisingly powerful
  - Recursion
  - Functions as first class objects (can be value of an expression, passed as an argument, put in a data structure)
- Implicit storage management (garbage collection)
- Lexical scoping
  - Earlier LISP's did not do that (dynamic)
- Interpreter
  - Compiled versions available too
Expressions

• Syntax - Cambridge Prefix
  – Parenthesized
  – (* 3 4)
  – (* (+ 2 3) 5)
  – (f 3 4)

• In general:
  – (functionName arg1 arg2 …)

• Everything is an expression
  – Sometimes called s-expr (symbolic expr)

Expression Evaluation

• Replace symbols with their bindings
• Constants evaluate to themselves
  – 2, 44, #f
  – No nil in Racket; use ‘()
    • Nil = empty list, but Racket does have empty
• Lists are evaluated as function calls written
  in Cambridge Prefix notation
    (+ 2 3)
    (* (+ 2 3) 5)
Scheme Basics

• **Atom**
  – Anything that can’t be decomposed further
  • a string of characters beginning with a letter, number or special character other than ( or )
  • e.g. 2, #t, #f, “hello”, foo, bar
  • #t = true
  • #f = false

• **List**
  – A list of atoms or expressions enclosed in ()
  – (), empty,(1 2 3), (x (2 3)), (()(()))
Quote

• If we want to represent the literal list (a b c)
  – Scheme will interpret this as apply the arguments b and c to function a

• To represent the literal list use “quote”
  – (quote x) → x
  – (quote (a b c)) → (a b c)

• Shorthand: single quotation mark
  ‘a == (quote a)
  ‘(a b c) == (quote (a b c))

Global Definitions

• Use define function

(define f 20)
(define evens ‘(0 2 4 6 8))
(define odds ‘(1 3 5 7 9))
(define color ‘red)
(define color blue) ; Error, blue undefined
(define num f) ; num = 20
(define num ‘f) ; symbol f
(define s “hello world”) ; String
Lambda functions

- Anonymous functions
  - (lambda (<formals>) <expression>)
  - (lambda (x) (* x x))
  - ((lambda (x) (* x x)) 5) → 25

- Motivation
  - Can create functions as needed
  - Temporary functions: don’t have to have names

- Can not use recursion

Named Functions

- Use define to bind a name to a lambda expression

  (define square (lambda (x) (* x x)))
  (square 5)

- Using lambda all the time gets tedious; alternate syntax:

  (define (<function name> <formals>) <expression1> <expression2> …)

  Last expression evaluated is the one returned

  (define (square x) (* x x))
  (square 5) → 25
Conditionals

(if <predicate> <expression1> <expression2>)
- Return value is either expr1 or expr2

(cond (P1 E1)
   (P2 E2)
   (P_n E_n)
   (else E_{n+1}))
- Returns whichever expression is evaluated

Common Predicates

- Names of predicates end with ?
  - Number? : checks if the argument is a number
  - Symbol? : checks if the argument is a symbol
  - Equal? : checks if the arguments are structurally equal
  - Null? : checks if the argument is empty
  - Atom? : checks if the argument is an atom
    - Appears undefined in Racket but can define ourselves
  - List? : checks if the argument is a list
Conditional Examples

- (if (equal? 1 2) ‘x ‘y) ; y
- (if (equal? 2 2) ‘x ‘y) ; x
- (if (null? ‘()) 1 2)) ; 1
- (cond
  ((equal? 1 2) 1)
  ((equal? 2 3) 2)
  (else 3)) ; 3
- (cond
  ((number? ‘x) 1)
  ((null? ‘x) 2)
  ((list? ‘(a b c)) (+ 2 3)) ; 5
  )

Dissecting a List

- **Car** : returns the first argument
  – (car ‘(2 3 4))
  – (car ‘((2) 4 4))
  – Defined only for non-null lists
- **Cdr** : (pronounced “could-er”) returns the rest of the list
  – Racket: list must have at least one element
  – Always returns a list
    • (cdr ‘(2 3 4))
    • (cdr ‘(3))
    • (cdr ‘((3))))

- **Compose**
  • (car (cdr ‘(4 5 5)))
  • (cdr (car ‘((3 4))))
Shorthand

- \((\text{cadr } x) = (\text{car } (\text{cdr } x))\)
- \((\text{cdar } x) = (\text{cdr } (\text{car } x))\)
- \((\text{caar } x) = (\text{car } (\text{car } x))\)
- \((\text{cddr } x) = (\text{cdr } (\text{cdr } x))\)
- \((\text{cadar } x) = (\text{car } (\text{cdr } (\text{car } x)))\)
- … etc… up to 4 levels deep in Racket
- \((\text{cddadr } x) = ?\)

Why Car and Cdr?

- Leftover notation from original implementation of Lisp on an IBM 704
- \(\text{CAR} = \text{Contents of Address part of Register}\)
  - Pointed to the first thing in the current list
- \(\text{CDR} = \text{Contents of Decrement part of Register}\)
  - Pointed to the rest of the list
Building a list

• Cons
  – Cons(construct) a new list from first and rest
  – Takes two arguments
  – Second should be a list
    • If it is not, the result is a “dotted pair” which is typically considered a malformed list
  – First may or may not be a list
  – Result is always a list

Building a list

\[
\begin{align*}
X &= 2 \text{ and } Y = (3 \ 4 \ 5) : (\text{cons } x \ y) \rightarrow \\
&= (2 \ 3 \ 4 \ 5) \\
X &= () \text{ and } Y = (a \ b \ c) : (\text{cons } x \ y) \rightarrow \\
&= () \ a \ b \ c \\
X &= a \text{ and } Y = () : (\text{cons } x \ y) \rightarrow \\
&= (a)
\end{align*}
\]

• What is
  – (cons 'a (cons 'b (cons 'c '())))
  – (cons (cons 'a (cons 'b '())) (cons 'c '()))
Numbers

• Regular arithmetic operators are available
  +, -, *, /
  – May take variable arguments
    (+ 2 3 4), (* 4 5 9 11)
• (/ 9 2) \(\Rightarrow\) 4.5 ;  (quotient 9 2) \(\Rightarrow\) 4
• Regular comparison operators are available
  < > <= >= =
  • E.g.  (= 5 (+ 3 2)) \(\Rightarrow\) #t

= only works on numbers, otherwise use equal?

Example

• Sum all numbers in a list

  (define (sumall list)
    (cond
      ((null? list) 0)
      (else (+ (car list) (sumall (cdr list))))))

  Sample invocation:  (sumall '(3 4 5 1))
Example

• Make a list of n identical values

(define (makelist n value)
  (cond
   ((= n 0) '())
   (else
    (cons value (makelist (- n 1) value))
   )
  )
)

In longer programs, careful matching parenthesis.

Example

• Determining if an item is a member of a list

(define (member? item list)
  (cond ((null? list) #f)
        ((equal? (car list) item) #t)
        (else (member? item (cdr list)))
  )
)

Scheme already has a built-in (member item list) function that returns the list after a match is found
Example

• Remove duplicates from a list

(define (remove-duplicates list)
  (cond ((null? list) '())
        ((member? (car list) (cdr list))
         (remove-duplicates (cdr list)))
        (else
         (cons (car list) (remove-duplicates (cdr list))))
  )
  )