Function design Basics in Programming Languages

(using Scheme)

Class poll on Function design

* How many know C++ (default), Java, Python…
* Are the functions designed the same way?
* Personally, my design for an application changes depending on the language!

Designing a function

* A functions give the ability to accept value, evaluate and return results
* Domain
  + Argument values entered
* Range
  + Return values
  + **can it return more than 1 value? more than 1 type?**
* (as a programmer)
  + How am I passing values
    - By reference
    - By copy
    - Do I have an option
    - Can I cheat and “const” a pass by reference?

(example next page)

|  |  |
| --- | --- |
| Design of functions can vary! | |
| Java | C++ |
| **class** driver  {  **public** **static** **int** main(String args[])  {  **int** Alabama = 100;  **int** Clemson = 77;  **int** Texas = 7;  Alabama = *reset*(Clemson, Texas);  **return** 0;  }  **public** **static** **int** reset(**int** x, **int** y)  {  x = 0;  y = 0;  **return** 0;  }  } | **#include** <iostream>  **using** **namespace** std;  **int** **main**()  {  **int** Alabama = 100;  **int** Clemson = 77;  **int** Texas = 7;  Alabama = reset(Clemson, Texas);  **return** 0;  }  **int** **reset**(**int** &x, **int** &y)  {  x = 0;  y = 0;  **return** 0;  } |

Scheme - not the best fit for all P.L. options

* For functions, Scheme lacks in
  + Pattern matching (2 forms)
    - Overloading
    - Regular Expressions
  + Currying
* but it depends on the Flavor!!!

|  |
| --- |
| No overloading for you! (In Scheme) |
| (define (add x y) (+ x y) )  (define (add x y z)  (+ x y z)  )  (define (add x y z a) ; this one will be the live version  (+ x y z a) ; of “add” since declared last  )  (display (add 23 42))  (display (add 23 42 63))  (display (add 23 42 63 70)) |
| Result |
| \*\*\* ERROR IN "overloadEx.scm"@15.10 -- Wrong number of arguments passed to procedure  (add 23 42) |

Flavor?? What … BBQ??

* there are many different Scheme versions available
  + Gambit
  + MzScheme
  + DrRacket
* all have their different built-in functions
* we have to be careful when researching for answers, use the manual!!
  + we will later

What does pure a function mean?

* Predictable
  + the function is not affecting its external context at all
  + everything is happening ***inside*** the function and produces a new result without altering (mutating) its arguments
* Safe since immutable
  + Creates a new state when values are changed
  + Does not rely on an outside state (global variable) for it to do it’s job
* does not require anything from the outside (global variables/states) for it to do it’s job

*When calculating its result, the function doesn’t cause any observable side effect, including output to I/O devices, mutation of objects, change to program state outside of the function, and so on.*

* Federico Kereki,  [Mastering Javascript Functional Programming](https://www.amazon.com/Mastering-JavaScript-Functional-Programming-depth/dp/1787287440/ref=sr_1_2?crid=38BOR9WB4Y0VK&keywords=mastering+functional+programming&qid=1575845783&sprefix=mastering+funct%2Caps%2C285&sr=8-2)
* must simpler to test since it does not require outside influences

|  |  |
| --- | --- |
| Examples of Pure **Functions** | |
| C++ | Scheme |
| size\_t add (int x, int y)  {  return x + y;  } | (define (add x y)  (+ x y)  )  (display (add 6 7)) |
| What does the function entail   * depends on it’s arguments * return values depend on the arguments   Notice what the pure functions doesn’t have   * does not print (printing is a side effect) * does not depend on global variables * does not mutate global variables | |

Every “reserved word” is a function

* Think about it… after the reserved word, doesn’t require parameters?

|  |  |
| --- | --- |
| Reserved words == Functions | |
| For Loop | If statement |
| for( int a = 10; a < 20; a = a + 1 )  {  cout << "value of a: " << a << endl;  } | if (x == 100)  { cout << "x is 100"; } |
|  | if (< n 2) n ;true  (+ (fib (- n 1)) (fib (- n 2)));false  ) |

Reading Function parameters and returns

* remember the important and consistent parts of a function
  + return
  + name
  + arguments
  + function body

|  |
| --- |
| Haskell Function Make-up |
|  |
| boolean not (boolean x)  {  if (x == True) return false;  return true; // don’t need else, since it made it here!  } |

1. As a team, think of one simple mathematical equation to create in Scheme. Do it!
   1. Must use power or sqrt or…. (use below to help)
   2. Use the example below to help
   3. Notice where the “displays” are placed in the code below
      1. (not within in the function)
2. What math functions does Gambit Scheme support?

Gambit Scheme Manual ([here](http://www.iro.umontreal.ca/~gambit/doc/gambit.html))

|  |
| --- |
| Example Mathematical Equations in Scheme |
| ; John Marrs, Dan McBride, Herman Vizcarrondo  (define (pythagorean\_theorem a b)    (sqrt (+ (\* a a) (\* b b)))  )  (define (quadratic a b c)    (/     (+       (\* -1 b) (sqrt (- (\* b b) (\* 4 a c)))     )     (\* 2 a)    )  )  (display (pythagorean\_theorem 5 4))  (display "\n")  (display (quadratic 1 2 -5)) |

Pattern Matching and Overloading

* Not really regular expression matching stuff
* many functions are located using pattern matching on their arguments
* but conditions inside the functions can also use pattern matching
* A call to a function can be compared to its definition just a few lines earlier
  + function call is evaluated by first evaluating the procedure name and all arg-exprs in order (left to right)
* The pattern can be
  + a constant value
  + include one or more variables
* Scheme (Gambit) is not the best for pattern matching (Reg Ex) unless
  + ***Scheme (and Python) does not support overloading***
  + Added library
  + But Haskell has some neat stuff

|  |
| --- |
| Wildcards in Haskell |
| In Haskell’s && “and” function, the long way it on top, using wildcards is on the bottom |

Dealing with Parameters/Arguments

* In remember that
  + Explicit parameter that is passed by specifying the parameter in the parenthesis of a method call
  + Implicit parameter that is passed by specifying an object variable (object reference) before the name of a method
  + In Scheme, you do have both options
* A function’s arity is the number of arguments that it accepts
* Some functions
  + accept a fixed number of arguments
    - such as Scheme/Haskell’s “cons”
  + accept any number of arguments
    - such as + or list
  + accept a range of argument counts
    - substring accepts either two or three arguments.

|  |  |  |  |
| --- | --- | --- | --- |
| Passing various type of parameters in SCheme | | | |
| Integers | String | Multiple values | Another Function |
| (function 5) | (function “cool”) | (function 4 5 “crap”) | (function area) |
| Spread | Default | Optional |  |
| (f x . rest) | (f (pi 3.14)) |  |  |

* one nice feature Scheme does offer is a “spread” parameter
  + not exactly overloading, but offers to place any “extra” parameters into a list
    - can break up that list into usable data later
  + in many examples called “rest”, but can be any name

|  |  |
| --- | --- |
| The spread operator | |
| Proving that the spread becomes a list | |
| (define (add x . rest)  (+ x rest)  )  (display (add 23 42)) (display "\n")  (display (add 23 42 63)) (display "\n")  (display (add 23 42 63 70)) (display "\n") | (+ 23 '(42))  (+ 23 '(42 63))  (+ 23 '(42 63 70))  (these were errors using the  function 🡨) |
| Working spread example | |
| (define (add x . rest)  (+ x (list-sum rest)) ; since rest is a list, sum it!  )  (define (list-sum lst)  (cond  ((null? lst) 0) ; #1  (else (+ (car lst) (list-sum (cdr lst)))) ; #2  )  )  (display (add 23 42)) (display "\n")  (display (add 23 42 63)) (display "\n")  (display (add 23 42 63 70)) (display "\n") | |

The last parameter option is #!optional. Find it in the HTML documentation given and

1. Go to Gambit Scheme Manual ([here](http://www.iro.umontreal.ca/~gambit/doc/gambit.html))
2. Understand optional’s use
3. code an easy function that explanation it’s use
   1. what is the optional’s value if none given? (answerb)
4. (nothing to do about optional) Much like Java’s Math.pi (apparently C++ does not have a shortcut), how would you get a Pi value in Scheme?

|  |
| --- |
| Example optional function |
| ; Asaf Drizlikh  ; 3 way addition function  (define (add3 a b #!optional (c 0)) ; default value of c is 0  (+ a b c) ; could also be (+ (+ a b) c)  )  (display (add3 2 1)) (display "\n")  (display (add3 2 1 4)) |

Totality of Functions

* (Total) function maps every element in the function’s domain to an element in its codomain.
* Partial function maps zero or more elements in the function’s domain to an element in its codomain, and can leave some elements undefined.

Let (local state/variable)

* Not following a functional mandate of no side effects
* let is the location to create local variables ***if really needed***
* within the function
* usually immediately after the function’s parameter list
* all local variables are declared within the let
  + then normal code after
  + but notice double ()s around the local variables
* also notice how I am indenting everything to be able to read it better
* if there happens to be an overlap, the recently declared version is used

|  |
| --- |
| Example of Let |
| (define (positive value)  (let ((true "greater that 0") (false "less than 0"))  (if (> value 0) true ; true section of if  false ; false section  )  ) ; end of let section  ) ; end of function  (display (positive 5)) (display "\n")  (display (positive -5)) (display "\n")  In the example above, “true” and “false” die after the code is completed. (looses scope) |

set! is the same as “=”

* mutate is to change the contents of a variable
  + what set! does
* try to avoid even though it might be beneficial in some cases
  + since each time it generate new structures

Closures

* “poor man's” (or woman’s) object orientation
* trap variables into a coded function
  + remember, a function is an object
* more on this later

|  |
| --- |
| Closure Example |
| (define (container value)  (lambda ()  (string-append "This container contains " value ".")  )  )    (define apple (container "an apple"))  (define pie (container "a pie"))  (apple) "This container contains an apple."  (apple) "This container contains an apple."  (pie) "This container contains a pie."  (pie) "This container contains a pie."  (apple) "This container contains an apple." |

Function Naming Schemes in well… Scheme

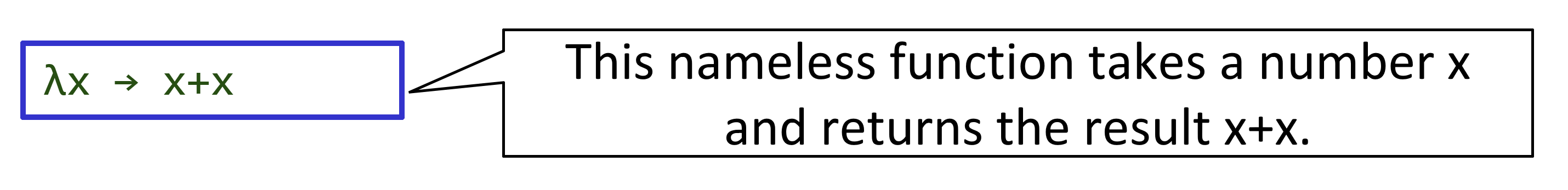
* there is there some help on determining what a pre-built (library) function does
* on some pre-built functions there is a special character at the end of the name

Using Gambit’s Manual:

1. find the list of pre-build functions (under General Index)
2. identify the special characters at the end of ***some*** pre-built functions
3. determine why those functions have those special characters

Lambda Expressions

* The symbol λ is the Greek letter lambda
* In mathematics, nameless functions are usually denoted using the ↦ symbol, as in x ↦ x+x.



* uses the λ symbol for nameless functions comes from the lambda calculus
* Lambda expressions can be used to avoid naming functions that are only referenced once.
* Lambda expressions can be bound to a name (function argument)

**Theory of Lambda Functions**

* remember that defining has a “side effect” (uses memory)
* anytime we “define”, we create a space in the global symbolic map
  + creates a “built-in” object
* much like a function pointer
  + pass in the function itself
  + code as a parameter
* BUT lambda functions
  + are created “on the fly”
  + think as a local variable
    - lives and dies quickly
    - can be passed around like data
    - applied arbitrarily
  + you give it the header, formal parameters, body and actual parameters ALL IN ONE!!
  + But YOU get to pass in the function (mapping) that you want

|  |  |
| --- | --- |
| **Defining Side effects** | |
| > (define (add x y) (+ x y) )  #<procedure:add>  since add is in the 0 position, it evaluates to it’s code  “add” tells us what symbol to look up in symbol of definitions, | symbolicMap  Symbol table of functions |

**Simple Unnamed Lambda function**

* there are many examples where functions are really one liners, or used only once
* so in order TO REDUCE code we use a lambda
  + instead of creating a function
  + then calling that function
  + etc…
* simple lambda’s only pass in simple arguments (values), ***or a***  function
  + notice the parameter does not have ( )
  + the function option will try to reduce the function to a single value, then pass into the REAL lambda part of the function

|  |
| --- |
| **Formal of a simple lambda function** |
| **(** **(** lambda (formal parameters) (body)**)** actual parameters**)**  **(** **(** lambda (formal parameters) (body)**)** (function)**)** |

**Behind the Scenes with Lambda**

* creates a new variables x
* binds variable x to the value 5
* evaluates the BODY of the function (\* x x)

|  |  |
| --- | --- |
| Examples of Lambda Functions | |
| Named Function | Unnamed Function |
| ; create function  (define square (x) (\* x x)))  ; this has a ***side effect*** “square”  ; call the function  (square 5) | ( (lambda (x) (\* x x)) 5)  ; no side effect since created AND  ; used at the same time |

|  |
| --- |
| **Simple lambda function examples** |
| (lambda (x) (+ x 3)) 0 #<procedure>  #From here down, notice a second set of )’s since we are passing a value  ((lambda (x) (+ x 3)) 7) 0 10 ((lambda (x y) (\* x (+ x y))) 7 13) 0 140 ((lambda (f x) (f x x)) + 11) 0 22 ((lambda () (+ 3 4))) 0 7  ((lambda (x . y) (list x y)) 28 37) 0 (28 (37)) ((lambda (x . y) (list x y)) 28 37 47 28) 0 (28 (37 47 28)) ((lambda (x y . z) (list x y z)) 1 2 3 4) 0 (1 2 (3 4)) ((lambda x x) 7 13) 0 (7 13)  Think of the highlighted GREEN area as a function body  ((lambda (x) (+ x x)) **(\* 3 4)**) ; passed in a function "\*", needed ()  ((lambda (x) (+ x x)) **12**) ; passed in a value, no ()s  both produce 24 |

**More complicated Lambda functions**

* Remember, we can pass in more than just parameters
  + ‘symbols
  + entire functions
    - +, cons, null, etc…
  + we need the more complicated Lambda functions if the function being passed in cannot simplify
    - like lists
    - tuples without 3 values
      * (+ 13) == 2
* we have to bind the function to a ***name*** using define
  + which looks like the NAME of the function!!

|  |
| --- |
| **Formal of a complicated lambda function** |
| **(** **define** name **(** lambda (formal parameters) (body)**)** **)** |

|  |
| --- |
| **Binding to a name** |
| (define double-any (lambda (f x) (f x x)))  (double-any + 10) => 20  (double-any cons 'a) => (a . a) |

Using Parameters with Lambda

* the parameters do not need to be a list
* they can be any of these
  + proper list (typical)
  + single variable (typical)
  + improper list
    - (var1 var2 … . varRest)
    - it needs to match # of variables BEFORE ., after are considered rest and placed in a list

|  |
| --- |
| **The “rest” parameter** |
| (define g (lambda(x . y) (list x y)))  (g 1 2 3 4) => (1 (2 3 4))  (define h (lambda(x y . z) (list x y z)))  (h 1 2 3 4) => (1 2 (3 4)) |

* Lambda function can lists (data structure)
  + really common

|  |
| --- |
| Lists as a Parameter for a Lambda Function |
| #The following is a recursive procedure that removes an item from a list  (define remove  (lambda (lst item)  (cond  ((null? lst) '())  ((equal? item (car lst)) (cdr lst))  (else (cons (car lst) (remove (cdr lst) item)))  )  )  )  The call (remove '(a b c) 'b) will yield the list (a c).  The call (remove '(1 2 3) '3) will yield the list (1 2). |

**But why use Lambda functions??**

1. Since you don’t have to name the function, one less name you have to keep track
   1. Useful if you only use it instead of defining an entirely new function you use once!
2. the version with lambda is also a bit shorter than the one with define
3. create a function to use it as an argument to another function
4. Remember some USEFUL functions require another function in order to do it’s job.
   1. so if we don’t have to name a function, less side effects for us.

|  |
| --- |
| Function that requires another to work!!! |
| (remove odd? '(1 2 3 4 5)) => (2 4)  odd? is a function itself. It is telling WHAT remove to remove (weird English)  The “remove” function accepts TWO parameters, a function to use (think of a comparator) and the actual data |

|  |
| --- |
| Higher-level function needing another |
| (define square (lambda (x) (\* x x)))  (apply + '(1 2 3 4 5)) ; returns 15  (apply square '(5)) ; square was already defined returns 25  (apply (lambda (x) (\* x x)) '(5)) ; passed in actual function, returns 25 |

|  |  |
| --- | --- |
| Conversion from Regular Function to Lambda #1 | |
| Regular Function | Lambda Function |
| (define (fName parameter) (body)) (map fNametoMap list)  (define (add42 x) (+ x 42)) (map add42 ‘(1 2 3 4))  // notice two steps  // 1. declare function  // 2. use function, pass values | (map (lambda (parameter) (body)) list)  (map (lambda (x) (+ x 5)) ‘(1 2 3 4)) => (6 7 8 9)  (map (lambda (x) (\* x 2)) ‘(1 2 3 4)) => (2 4 6 8)  // notice two steps COMBINED  // 1. “declare” function  // 2. use function, pass values |

|  |  |
| --- | --- |
| **Conversion from Regular Function to Lambda #2** | |
| Regular Function | Lambda Function |
| (define (countNegative aList)   (cond     ((null? aList) 0)     ((not (list? aList)) (if (< aList 0) 1 0))     (else (apply + (cons (countNegative (car aList))                          (map countNegative (cdr aList))))))) | (define (countNegative2 aList)   (length (filter (lambda (x) (< x 0)) aList)))  Both filter and length are HIGHER order functions (pre-built)  Filter function requires 2 parameters, the algorithm and the list |

Answers

|  |
| --- |
| #!Optional usage (or not) |
|  |

Sources

Dr. Dylan Shell 314 notes

Gambit Scheme Manual

<http://gambitscheme.org/wiki/images/a/a7/A_Tour_of_Scheme_in_Gambit.pdf>

Basics of Function Programming

<https://www.youtube.com/watch?v=FYXpOjwYzcs>

Total Functions

<https://en.wikipedia.org/wiki/Total_functional_programming>

Federico Kereki,  [Mastering Javascript Functional Programming](https://www.amazon.com/Mastering-JavaScript-Functional-Programming-depth/dp/1787287440/ref=sr_1_2?crid=38BOR9WB4Y0VK&keywords=mastering+functional+programming&qid=1575845783&sprefix=mastering+funct%2Caps%2C285&sr=8-2)