Functions I

**Functions in Theory**

* Couple of definitions
  + a procedure to complete a SPECIALIZED task, no matter how small or large
  + a PACKAGE of code
* Functions can be run any numbers of time you want
* Will do the SAME THING each time it is run, no exception
* Functions are SEQUENTIAL – starts from top, ends a bottom
  + so declare variables at beginning of method/program so you can use later
  + compiler will READ top to bottom
* Functions in Scheme return a value PERIOD
  + might need to display that value in order to see the work

|  |  |
| --- | --- |
| **Lupoli’s Routine** | **Lupoli’s Routine using subroutines** |
| wake up  shower, dress  drive to work  go to frig  find food  grab a coke  find a seat  eat food  **teach**  go to frig  find food  grab a coke  find a seat  eat food  drive home  wake up | wake up  shower, dress  drive to work  EATING   |  | | --- | | EATING | | go to frig  find food  grab a coke  find a seat  eat food |   **teach**  EATING  drive home  wake up |

**The Scheme .scm file Setup**

* in many examples, there really isn’t a certain way of setting up a file
* there does not have to be a main()
  + but notice the position it HAS to be
  + ***eerily similar to Python!!***
* each function is it’s on object, it’s own being
* no prototype is needed (like C, C++, etc..)
* ***unlike some programming languages, the ORDER of the functions do not matter!!***
  + ***BUT FOR ME TO READ, please follow the example***

|  |
| --- |
| **functionExample.scm** |
| ; full header here   |  | | --- | | (define (helloWorld)          ; prints and returns the message.          (printf  "Hello World\n")  ) | | (define (square n)          ; returns square of a numeric argument          (\* n  n)  ) | | (define (main)          (helloWorld) ; didn’t need a display since printf inside function          (display (square 20) )          (newline)  ) | | (main) ; call to the function main | |

**Display vs. Printf Scheme Built-in Functions**

* We will use display/printf for debugging and showing the result
* I’ve had different tools use one or the other, sometimes both
  + GL uses both
  + PVTS uses display
  + Dr. Scheme ??

**Interacting w/ functions using the interpreter**

* many functions can be in the same file
* must load the file into the interpreter before we can use
* the up, down, and tab keyboard shortcuts don’t work on mzscheme
  + booo, very inconvenient
* no “main” function

|  |  |
| --- | --- |
| **Functions and the Interpreter** | |
| Command line procedures | Explanation |
| [slupoli@linux2 functions]$ mzscheme  Welcome to Racket v5.1.3.  > (load "functionsEx1.scm")  > (helloWorld)  Hello World  > (square 20)  400  > (exit) | starts Scheme interpreter  load file with code into interpreter  call function “helloWorld”  response from interpreter  call function “square” w/ parameter  response from interpreter  exit interpreter |

* ***there are issues in “loading’!!!***
  + load checks for errors in the file!! (nice)
  + if you change (edit) the file you loaded, you need to reload!!!

**Creating a Simple function**

* reuses the define reserved word to create an instance of a function
* the overall function “form” is
  + (define (funName arguments) function-body)
* in the simple one below
* notice no parameters
  + called a “thunk”!!!
* get into more why later

|  |
| --- |
| **Simple function setup** |
| ❶     ❷         ❸  (define (printmenu) first ***line*** of ANY function is called a “function header”  ❹   (printf “Menu Choices\n”)  (printf “Z – zip code\n”)  (printf “M – middle initial\n”)  (printf “T – Temporary Swap\n”)  (printf “P – Permanent Swap\n”)  ) ❺  ❶ define  ❷ function name – case sensitive, make sure EXACT name  ❸ ’s – parameter list, (get into later), ***not*** ALWAYS needed  ❹ **indent (for style)**, start of function body  ❺ end of function |

**Defining has a “side effect”**

* anytime we “define”, we create a space in the global symbolic map
  + creates a “built-in” object

|  |  |
| --- | --- |
| **Defining Side effects** | |
| > (define (add x y) (+ x y) )  #<procedure:add>  since add is in the 0 position, it evaluates to its code | symbolicMap  Symbol table of functions |

**See how a method works (Mechanics)**

|  |  |  |
| --- | --- | --- |
| (define (main)  …                            ❶  …  (printmenu) ; calling  ; NEXT LINE                                          ❸  ) |  | (define (printmenu)        ❷  ) |

|  |
| --- |
| (define (main)  …  …  …  (printmenu)  …  …  …  ) |
| (define (printmenu)  …  …  ) |
| (main) |

**Simple Function Calling**

* To “Call” a function, it must be called from the MAIN(), or ***another function***

|  |
| --- |
| **Calling other functions** |
| (define (main)  (greeting)  )  (define (greeting)  (printf  “Hi, my name is Mr. Lupoli\n”)  )  (main) |

* The greeting would be displayed when running the program
* Notice how we get to the printf statement – called tracing
* Notice all we needed was the function name AND ’s
* ANY function could call another function (greeting could call “address” )

**What can you put into a method??**

* Anything you have already learned!!
  + Recursive Loops
  + Lists
  + If-else
  + Variables
  + Etc…

**Parameter passing in functions**

* same as what you are used to in other languages
* argument expressions are always evaluated, even if the function never uses them
  + called eager evaluation

**Creating a method that returns a value**

* means the method will return a value to YOU AS THE PROGRAMMER
* The method does some work!!
* Scheme functions can return MULTIPLE VALUES!!!
  + but harder
  + need you to think like a Java programmer to make it easier
    - just **one value** to return
* usually the same variable that goes in is changed to return a NEW value

|  |  |
| --- | --- |
| **Methods returning values** | |
| (define getZip -1) ; global variable  (define (main)  …                            ❶  …  (set! getZip (zipcode)) ; calling  ❸  ; NEXT LINE    )  (main) | (define (zipcode)  ❷  21118 ; put in YOUR zip code  )  ; notice no “return” keyword!! |

|  |
| --- |
| File Setup |
| (define (zipcode)  …  …   21118  ) |
| (define (main)  …  …  …  (set! getZip (zipcode))  …  …  …  ) |

you see: 21118 NOT zipcode

zipcode acts like a VALUE

THIS HAPPENS FOR ***ANY*** METHOD THAT IS A VALUE

float

double

char

…

|  |
| --- |
| **Example of a function #1** |
| (define (has-value x) ; x is the input value      ( if(null? x)            #f ; then something there (double negative) (then expression)            #t ; then really no value entered (else expression)            ; “false” and “true” also work in mzscheme/Racket      )  )  > (load "hasValue.scm")  > (has-value? 2)  #t  > (has-value? 0)  #t  > (has-value? null)  #f |

|  |  |
| --- | --- |
| **Identify Method Exercise #1 and #2** | |
| (define (attendance price)    (+ (\* (/ 15 .10) (- 5.00 price))       120)) | (define (interest-rate amount)    (cond      [(<= amount 1000) 0.040]      [(<= amount 5000) 0.045]      [(> amount 5000) 0.050])) |
| 1. Identify the parameters for each method? 2. What possible values could be returned from these methods? | |

**Scheme and returning values**

* not only does Scheme function return values, the values can be different
  + nothing defines what type have to be returned!!
  + unlike many of the programming languages you know

|  |  |
| --- | --- |
| **Returning values from a Function Design** | |
| C++ | Scheme |
| void insert( const Comparable & x, BinaryNode \*  string getFname() { return Fname; }  friend bool compare(STUDENT &e1, | (define (return-example a)    (cond     ((boolean? a) "was a boolean")     ((rational? a)(/ a 2)) ; a got divided in half     (else #f)     )   )  (display (return-example 11))  ; 11/2  (newline)  (display (return-example 12))  ; 6  (newline)  (display (return-example #t))  ; was a boolean  (newline)  (display (return-example "Lupoli")) ; #f |

**Naming Convention of Functions**

* if you create a predicate function, place a ? at the end of the name to be consistent with other predicate functions
  + string?, char?,
* many example of scheme functions use – to separate words in the functions instead of camel case.

|  |
| --- |
| **Scheme function naming conventions** |
| (define (is-null? value)  …  (define (has-a-zero? list)  … |

**Calling other Methods, determining order**

* when calling another function name sure to be CASE SENTATIVE with the name of the function
* calls are sequential!!!
  + ORDER in when they are called matters!!
* Functions can be called from:
  + the main()
  + or OTHER methods!!
  + ***BUT ALWAYS START AT THE MAIN!!!!***

|  |
| --- |
| **Method Calling Exercise #1** |
| (define (main)    (getFunction1)  (getFunction2)  (getFunction4)  )  ; one line to save space  (define (getFunction1) (printf "A") (getFunction3) )  (define (getFunction2) (printf "B") )  (define (getFunction3) (printf "C") )  (define (getFunction4) (printf "D") (getFunction3) )  (define (getFunction5) (printf "E") )  (define (getFunction6) (printf "F") )  (main) |
| 1. Which method in this ENTIRE program is called 1st? **(except main() )** 2. Which method in this ENTIRE program is called 2nd? 3. What will be displayed from the code above |

|  |  |
| --- | --- |
| **Method Calling Exercise #2** | |
| (define (main)  [  (getFunction3)  (getFunction1)  (getFunction5)  (getFunction6)  (getFunction2)    )  (define (getFunction1) (printf "A") (getFunction3) )  (define (getFunction2) (printf "B") )  (define (getFunction3) (printf "C") )  (define (getFunction4) (printf "D") (getFunction3) )  (define (getFunction5) (printf "E") )  (define (getFunction6) (printf "F") )  (main) |  |
| ; what will be displayed from the code above |  |

**Variables within a Function (LET)**

* anything declared with let is a local variable
* the name of the value is temporarily bound to the function
* complicated part
  + let is declared just after the header, all code must be INSIDE the let ( ) in order to have the variable available!!

**Scope (“Jurisdiction”) of variables**

Global variables

* “live” and can be accessed thru-out the ENTIRE program
* ***is NOT declared WITHIN a method*** 🡪 dead giveaway
* ***only globals you should use are CONSTANTS!!!***

Local

* live and die in method it was created
* can only be accessed by that method
* ***is declared WITHIN a method*** 🡪 dead giveaway

Global vs Local (variable with the same name, one local, one global)

* in Scheme, this is possible, (hopefully won’t happen)

|  |  |  |
| --- | --- | --- |
| **Scope of variable exercises** | | |
| Example 1 | Example 2 | Example 3 |
| (define (main)       (printmenu)  )  (define (printmenu)       (printf "~s" x)       ; what will x display??  )  (main) | (define x 0)  (define (main)       (printmenu)  )  (define (printmenu)   (printf "~s" x)   ; what will x display??  )  (main) | (define (main)    (let ((x 0)          (y -1)          )       (printmenu)    )  )  (define (printmenu)     (let ((x 30)           (y -10)           )           (printf "~s" x)           ; what will x display??      )  )  (main) |

What will X display in EACH example? (options are 0, 30, error) for each

**Returning a Scheme value from a function**

* it can only  ONE value (easily)
  + but that one value could be ANY datatype
    - list, atom, number, etc…
* not really meant to return more

|  |
| --- |
| **Function returning a value with conditions** |
| (define (has-value? x) ; x is the input value      (         cond         ( (eq? x "") #f )         ( else #t )      )  ) |

How many POSSIBLE values can this return?

How many values WILL BE returned?

**Built-In datatype check functions**

* there is a datatype check for each type of number
  + predicate (true/false) if it has a ? with the function name
* the response will always be #t/#f

|  |  |
| --- | --- |
| **Checking Datatypes** | |
| **real-life use** | **examples** |
| (define x 10)  …  …  if(number? x) ; will return #t | (number? 42)       **=>**  #t  (number? #t)       **=>**  #f  (complex? 2+3i)    **=>**  #t  (real? 2+3i)       **=>**  #f  (real? 3.1416)     **=>**  #t  (real? 22/7)       **=>**  #t  (real? 42)         **=>**  #t  (rational? 2+3i)   **=>**  #f  (rational? 3.1416) **=>**  #t  (rational? 22/7)   **=>**  #t  (integer? 22/7)    **=>**  #f  (integer? 42)      **=>**  #t  (char? #\c) **=>**  #t  (char? 1)   **=>**  #f  (char? #\;) **=>**  #t  (boolean? #t)              **=>** #t (boolean? "Hello, World!") **=>** #f  (symbol? 'xyz) **=>** #t (symbol? 42)   **=>** #f  (zero? 42)   **=>** #fc |

**Built-In Predicate Value checker**

* there is a value check for each type of datatype
  + a single value
* the response will always be #t/#f

|  |  |
| --- | --- |
| **Checking Values** | |
| **real-life use** | **examples** |
| (define x 10)  …  …  if(zero? x) ; will return #t | (zero? 42)   => #f  (even? arg )  (odd? arg )  (null? x) |

**Functions in Scheme – First Class Objects**

* theory item
* This means you can use functions as if they were variables or constants

Resources:

<http://docs.racket-lang.org/guide/lambda.html>

<http://www.phyast.pitt.edu/~micheles/scheme/scheme16.html>

Function Examples:

<http://htdp.org/2003-09-26/Solutions/movie-examples.html>