CS 115 – SOS Review Package – Midterm 1

Srinidhi Sridharan (Tutor) – srinidhi22@hotmail.com – 226 808 8764

DEFINING FUNCTIONS

Similar to mathematics, every definition of a function has three main components

1. Name of the Function
2. Parameters that the function consumes
3. An expression that uses these parameters

General Format

```
(define (function_name parameter1 parameter2 .....)(function_expression))
(define (f x)(* x x))
(define (g y z)(* (+ y z)(* y y)))
```

BUILT IN FUNCTIONS

1. The basic 4 math functions are built into Scheme

```
(+ 6 (* 5 5)(/ 45 5))
(* 21 (/ 1 21))
```

General Format

```
(operator item1 item2)
(operator (operator item1 item2) item3)
```

incorporate basic rules of BEDMAS

2. Other basic useful functions

```
(quotient 70 8) ; just gives you the integer value when dividing
(remainder 70 8) ; will give the remainder when dividing
(max 20 25) ; gives the maximum of a group of numbers
(min 20 25) ; gives the minimum of a group of numbers
(floor 20.8) -> 20
(ceiling 20.8) -> 21
(sqr 2) -> 4
(sqrt 4) -> 2
```

DEFINING CONSTANTS

```
(define x 4) ; simply write define, name the constant, and say what it equals
(define v (* 4 2 x)) ; now if we type in v, we would get a result of 32.
```

Basic rules when using constants

a) you cannot define the same variable name twice:

```
(define x 4)
(define x 5)
```

b) Scheme keeps track of variables. Once you define "x" you can then use

the definition of x in other functions as well. Once it's defined, you can't redefine it. You will learn how to change a definition in cs116.
(define x 5)
(define (y z) (* z z x))

Key Example:

A ->(define y 5)
B ->(define (x y z)(* y z))
C -> (define (y z)(* 4 z))
D ->(define (z x)(* 3 x))

a) How do the x, y and z terms relate to each other?

x-Terms: The first x is the name of a function, and even though we used x before, we can still use the x as a parameter. The 2nd x is simply a parameter for the function z. The 2nd x is "binded" to the function z, and is not referring to the first x. However, if we tried defining a new function with the NAME x, it would yield an error.
If you define x to be a function, the letter x can still be used as a parameter for other functions

y-Terms: since y is defined to be 5, scheme remembers the value of y for the rest of the code. In the next line, since y is equal to 5, we know that the function x is simply multiplying the parameter z, with the number 5. However, function C is going to yield an error. Since y is already defined as 5, we cannot redefine it to be something else.

z-Terms: similar to the x-terms, z is defined once as a function, and as a parameter. This is fine.

b) if you eliminate the (define y 5), then all three functions will work.

TRACING
A step by step simplification of a scheme function by applying substitution rules
One step at a time is the key to getting these questions correct

1) (/ (+ 40 (* (min 5 6 9 3) 2)) 2)
➔ (/ (+ 40 (* 3 2)) 2)
➔ (/ (+ 40 6) 2)
➔ (/ 46 2)
➔ 23

2) (define (square x y) (+ (* x x) (* y y)))
Using the above definition, trace:
(square (- 5 2) (* 3 4))
➔ (square 3 (* 3 4))
➔ (square 3 12)
➔ (+ (* 3 3) (* 12 12))
➔ (+ 9 (* 12 12))
➔ (+ 9 144)
➔ 153

When you are tracing an expression which has a function within it, you have to use the following process

1. Simplify the expression one step at a time, until you are left with the function name and the parameters it's consuming
2. Substitute ALL PARAMETERS into the function expression at once. Then continue one step at a time

THE DESIGN RECIPE
The design recipe is a key component of cs115. Particularly on midterms, the importance of the design recipe can't be undervalued, and can result in a lot of marks for you.

THE CONTRACT
A contract is a simple statement of the function name, the types of parameters that it consumes and the type of output. You're telling the user what are the types of things they can input, and what types of things they will get out of it
Ie. If the function squarepower takes in a number and squares it, here is what we get

```
;;squarepower: num -> num
```

1. num : any numeric value
2. int : for an integer
3. nat : a natural number (the set of all non-negative numbers)
4. any : any scheme value
5. (int[>5]) : such a format is used for integers with given restrictions

We will see that there are other types of values that can be put in the contract (such as bool, str and more). We'll get to them soon.
THE PURPOSE
A statement that describes the function, its parameters (by name) and what the function should produce
EXAMPLES
Provide representations of all the possible cases and outcomes that could arise with the function
BODY
The actual code
TESTING

```
(check-expect (function_name   sample_values ) expected_value)
```

If your function output and the expected value are equal, then the test will pass. Scheme will tell you if one or more of your tests have passed or not.

DESIGN RECIPE EXAMPLE
1. *Produce a function called sum_square that produces the sum of the squares of any two real numbers, but rounds the number down to the nearest integer*

```
;;sum_square: num num -> int
;;Purpose: the function consumes two numbers, a and b, produces the sum of the squares of these two numbers, rounded down to the nearest integer.
;;Example: (sum_square 2 4) => 20
;; (sum_square 3.4 6.3) => 51
;; (sum_square -5 6) => 61
(define  (sum_square  a  b)  (floor  (+  (* a a)  (* b b))))
;;Tests
(check_expect  (sum_square   2   4)  20)
(check_expect  (sum_square 3.4    6.3)  51)
(check_expect  (sum_square  -5   6)  61)
```

STRINGS
A value made up of words, letters, numbers, spaces, and punctuation marks that are enclosed in quotation marks (Ex. “srinidhi” “srinidhi22” “srinidhi sridharan” “dfjak dsfjkdsfkl 8 dfkdf 9”)

“Srinidhi”
0 1 2 3 4 5 6 7 8

(string-append “joe” “ clark”) => “joe clark”
(string-length “joe clark”) => 9
(substring “srinidhi” 0 4) => “srin”
(substring “srinidhi” 0 1) => “s”
(substring “srinidhi” 1) => “rinidhi”
(string>? “nowhere” “here”) => true

HELPER FUNCTION
A helper function is defined before the main function, and is used to general similar expressions, express easy/repetitive computations in a clear form, and to avoid really long complex definitions
- You still need the full design recipe for helper functions
- Choose meaningful names (not just helper1 etc.)

SWAP PARTS EXAMPLE
Take a string and swap the front of the string with the back of it. If the string length is odd, make the front part shorter.
;;front_part: string -> string
;;Purpose: to take a string, str, and produce the front part of it
;;Examples: (front_part “srinidhi”) => “srin”
;; (front_part “srinidh”) => “sri”
(define (front_part str) (substring str 0 (floor (/ (string-length str) 2))))
;;tests
(check-expect (front_part "srinidhi")"srin")
(check-expect (front_part "srinidh")"sri")

;;back_part: string -> string
;;Purpose: to take a string, str, and produce the back part of it
;;Examples: (front_part “srinidhi”) => “idhi”
;; (front_part “srinidh”) => “nidh”
(define (front_part str) (substring str (floor (/ (string-length str) 2))))
;;tests
(check-expect (front_part "srinidhi")"idhi")
(check-expect (front_part "srinidh")"nidh")

;;swap_parts: string -> string
;;Purpose: to take a string, str, and swap the front and back parts
;;Examples: (swap_parts “srinidhi”) => “idhisrin”
;; (swap_parts “srinidh”) => “nidhsri”
(define (swap_parts str) (string-append (back_part str) (front_part str)))
;;tests
(check-expect (swap_parts "srinidhi")"idhisrin")
(check-expect (swap_parts "srinidh")"nidhsri")

BOOLEAN VALUES
A boolean value is either true or false
Boolean Functions are functions that produce either true and false. Examples include

(= x y)
(< x y)
(> =x y)

Predicates: are asking questions such as zero? number? Integer? equal? Etc.
These ones are built in, but we can define our own predicates too:

(define (long_name? name)(>= (string-length name) 8))

COMPLEX RELATIONSHIPS
AND, OR, NOT

- AND statements mean that both conditions must be true, for the entire statement to be true
    - (and (= 5 (+ 3 2))(= 1 1)) => true
- OR statements indicate that only one of the conditions must be true
    - (or (= 5 5)(= 1 0)) => true
- NOT statements require the inner value to be false, for the NOT statement to be true
    - (not (= 5 6)) => true

Complex Relationship Trace

(define str “srinidhi”)
(and (<= 6 (string-length str)) (equal? “inid” (substring str 2 6))

- ⇨ (and (<= 6 (string-length “srinidhi”))(equal? “inid” (substring str 2 6)))
- ⇨ (and (<= 6 8)(equal? “inid” (substring str 2 6)))
- ⇨ (and (<= 6 8)(equal? “inid” (substring str 2 6)))
- ⇨ (and true (equal? “ind” (substring str 2 6)))
- ⇨ (and (equal? “ind” (substring str 2 6)))
- ⇨ (and (equal? “ind” (substring “srinidhi” 2 6)))
- ⇨ (and (equal? “ind” “inid”))
- ⇨ (and true)
- ⇨ (and)
- ⇨ True

CONDITIONAL EXPRESSIONS
Conditional expressions are basically a series of questions being asked, and answers being produced if the question is true or false. They allow you to analyze functions by 'cases'.

The general form is

```
(define (function parameter ...)
        (cond
                [ (question 1) answer1]
                [(question 2) answer 2]
                ...
                [else answer]))
```

1. Evaluates from top to bottom
2. If one of the expressions is found to be true: function stops
3. If you get to the else part, that answer will be a given result

Example: make a function perfect_relationship, that consumes two strings, representing the names of two people, a positive integer, and a number representing how many hours the two people have been together. If they have known each other for more than 24 hours, they love each other. If they have known each other for less than 24 hours, but the positive integer is even, they still love each other. Otherwise, they hate each other. Make your function produce "name1 loves name2" or "name1 hates name2".
Ie. (perfect_relationship "Miss Universe" "Srinidhi" 5 92) => "Miss Universe loves Srinidhi"
(perfect_relationship "Srinidhi" "you" 5 23) => "Srinidhi Hates you"

SYMBOL
A symbol starts with a single quote ` followed by something (ie. `blue, `red)
Best used for comparisons: it's more efficient than comparing two strings: a symbol is a value, like the number 6. Type symbol in the contract.

CHARACTER
#\space #\1 #\a
Built in predicates include: char-upper-case?, char-lower-case?, char-numeric?, char-whitespace?
A character data type allows you to distinguish uppercase, lowercase letters, numbers, whitespace etc.
Type char in the contract

INTRO TO STRUCTURES: POSN STRUCTURES
A structure is a way of bundling data together to form a single package
1. You can create functions that consume and or produce structures
2. Define our own structures and/or
3. Each has a
    - Constructor function
    - Selector Functions
    - Type Predicate Function

For Example:
A Posn is a structure that has two fields containing numbers that represent x & y coordinates

1. Constructor function : (make-posn num1 num2)

2. Selector functions : (posn-x (make-posn num1 num2)) => num1
                        (posn-y (make-posn num1 num2)) => num2

3. Type Predicate: (posn? (make-posn num1 num2)) => true

If we construct a posn, it would look like this:
(define point (make-posn 20 45))

This gives us selector functions
(posn-x point) => 20
(posn-y point) => 45

And a type predicate:
(posn? point) => true

If I want to create a function that scaled the posn, point, by a factor
(define (posn_scale factor)(make-posn (* (posn-x point) factor) (* (posn-y point) factor)))

DATA DEFINITIONS & STRUCTURE DEFINITIONS

1) A Data Definition: is a comment specifying the types of data being used

;;a posn is a structure (make-posn   xcoord   ycoord), where
;;xcoord is a number
;;ycoord is a number

2) A Structure Definition: is a code defining the structure (like we did above) and allowing us to then use constructor, selector and predicate function from the definition

(define-struct sname (field1 field2.....))

Doing this gives you three functions

make-sname	;constructor
sname-field1, sname-field2....	;selectors
sname?	;type predicate

Write a function called `reply` that consumes an `email` and a string and produces an `email` that is a reply to the `email` consumed using the message given. The `contact` that sends the reply email will be the same as the `contact` that received the original `email`. The `contact` that receives the reply `email` will be the same as the `contact` that sent the original `email`. The subject of the reply `email` will be the same as the subject of the original `email` except that it will start with `"RE: "`. The `message` of the reply `email` will be the string consumed by the `reply` function, followed by a space, followed by the `name` of the sender, followed by a space, followed by the word `"said"` followed by a space followed by the original message.

For example `(reply (make-email (make-contact "Sandy Graham" "slgraham@uwaterloo.ca") (make-contact "J.P. Pretti" "jpretti@uwaterloo.ca") "Hello" "I love CS115!") "Me too.")` produces `(makeemail (make-contact "J.P. Pretti" "jpretti@uwaterloo.ca") (make-contact "Sandy Graham" "slgraham@uwaterloo.ca") "RE: Hello" "Me too. Sandy Graham said I love CS115!")`

```
(define-struct contact (name email_address))
;; A contact is a structure (make-contact n e) where
;; n is a non-empty string representing a person's name and
;; e is a non-empty string representing a person's email address


(define-struct email (from to subject message))
;; An email is a structure (make-email f t s m) where
;; f is a contact representing who sent the email,
;; t is a contact representing who received the email,
;; s is a string representing the subject of the email, and
;; m is a non-empty string representing the text of the message

(define (reply an_email new_message)
  (make-email (email-to an_email)
              (email-from an_email)
              (string-append "RE: " (email-subject an_email))
              (string-append new_message " " (contact-name(email-from an_email)) "
said " (email-message an_email))))

;test

(check-expect (reply (make-email (make-contact "Sandy Graham"
"slgraham@uwaterloo.ca") (make-contact "J.P. Pretti" "jpretti@uwaterloo.ca") "Hello"
"I love CS115!") "Me too.")(make-email(make-contact "J.P. Pretti"
"jpretti@uwaterloo.ca")(make-contact "Sandy Graham" "slgraham@uwaterloo.ca") "RE:
Hello" "Me too. Sandy Graham said I love CS115!"))

The test passed!
```

LISTS

Formal Data Definition
A list is either:
- empty or
- (cons f r), where
- *f is a value and
- *r is a list

My Explanation
In real life, a list is just a list....there isn't much to it.
In computer science, we deal with lists differently than we did in real life. A program's speed is dependent on how large the actual list is.

Henceforth, when we talk about a list, think of a list with two parts
1. The FIRST element of the list
2. The REST of the list (which is a list as well)

A list in scheme looks like this:
(cons 'a (cons 'b (cons 'c (cons 'd empty))))
While the empty list is just:
empty

To make our lives easier, we can do this
(define alist ((cons 'a (cons 'b (cons 'c (cons 'd empty)))))   ; and now refer to alist in our code

Using our definition of our list above:

```
(first alist)                           => 'a
(rest alist)                            => (cons 'b (cons 'c (cons 'd empty))))
(first (rest alist))                    => 'b
(rest (rest alist))                     => (cons 'c  (cons 'd empty))
(rest (rest (rest (rest alist))))       => empty
```

A list Template:

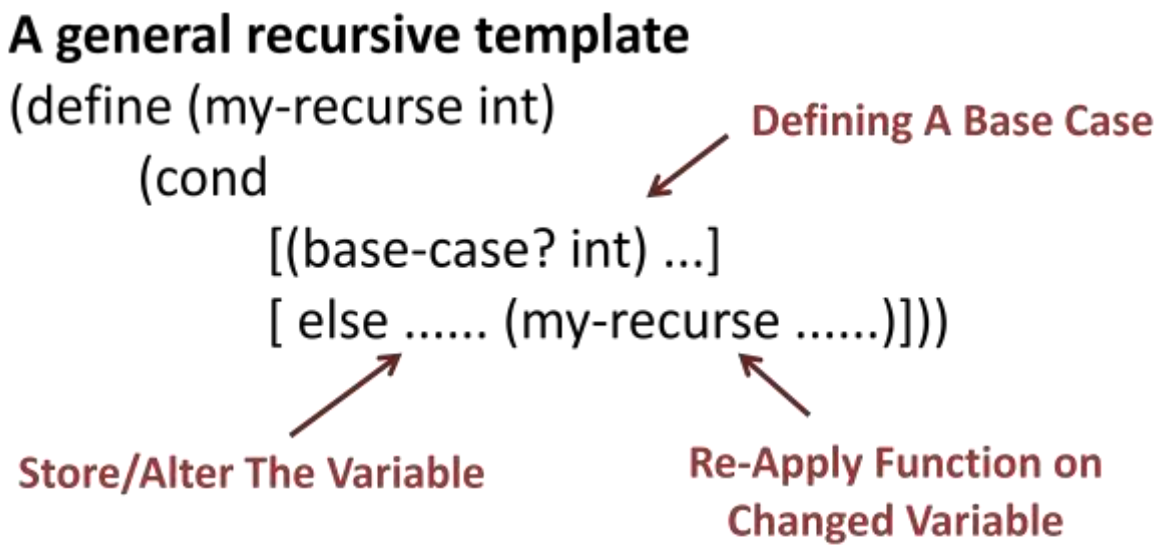
```
(define (mylist alist)
        (cond
          [(empty? alist)...]
          [(cons? alist) ...]))
```

When you are completely lost on a question, using the list template will help get you started and help you to avoid losing too many marks.

RECURSION
The continuous application of a variable that is defined in terms of a past.

Recursive Process:

1. Create a base case that ends the program when true
2. Test your base case with given variables
3. Change the variable and recurse the formula


A general recursive template
Defining A Base Case
Store/Alter The Variable
Re-Apply Function on Changed Variable

```
(define (my-recurse int)
      (cond
            [(base-case? int) ...]
            [ else ...... (my-recurse ......)]))
```

Create a function, countdown, that takes in a integer, int, and creates a descending list of numbers, starting with the int, and ending with zero.
(countdown 0) -> (cons 0 empty)
(countdown 2) -> (cons 2 (cons 1 (cons 0 empty)))

```
(define (countdown int)
        (cond
                [(zero? int)(cons 0 empty)]
                [else (cons int (countdown (- int 1)))]))
```

Using structural recursion, write a Scheme function called reciprocate that consumes a list and produces a list of numbers and symbols by reciprocating each entry in the list (the reciprocal of x is 1=x). If the input list entry is 0, then its reciprocal is the symbol ’Infinity. Likewise, if the list entry is ’Infinity, then its reciprocal is 0. Finally, if the list entry is not a number or the symbol ’Infinity, then it should be skipped.

For example,
(reciprocate (cons 2 (cons 0 (cons ’Infinity (cons "boo" empty)))))
produces (cons 0.5 (cons ’Infinity (cons 0 empty))).

```
;; reciprocate: list -> list

(define (reciprocate alist)
  (cond
    [(empty? alist)empty]
    [(equal? 'Infinity (first alist)))(cons 0 (reciprocate (rest alist)))]
    [(not (number? (first alist)))(reciprocate (rest alist))]
    [(zero? (first alist))(cons 'infinity (reciprocate (rest alist)))]
    [else (cons (/ 1 (first alist))(reciprocate (rest alist)))]]))

(reciprocate empty)
(reciprocate (cons 2 empty))
(reciprocate (cons 0 (cons 5 (cons 'infinity (cons 2 empty)))))
(reciprocate (cons 'infinity (cons 0 (cons 5 (cons "srinidhi" empty)))))
```