Franklin Smith CIS 425 Homework 7 Assignment

8.1A. twice(Pred, 1);

Pred 1 -> 0

Pred 0 raise Except(x) -> 0

F(f(x)) -> evaluates inside first in order to solve for the outside.

It evaluates to 0

8.1B. twice(Dumb, 1);

Dumb(Dumb 1)) -> 1

Dumb(1) -> 1

Exception is raised on line 4. The function evaluates to 1.

8.1C. twice(Smart, 1);

1 + Pred (0)

1 + raise exception(x)

1 + 1 = 2

Exception raised on line 3 in Pred where x is 0, exception

Is handled on line 5, evaluates it to 1.

8.2 Exceptions

Because there is an exception handler for hd. Since nil is passed into hd first, the exception will be raised.

What does hd(l) :: tl(l) mean?

What makes g return properly is that hd is called first, they both set the same formal and actual input parameter.

8.4 fun f(0) = 1

| f(1) = raise Odd

| f(3) = f(3-2)

| f(5) = (f(5-2) handle Odd => ~5)

| f(7) = (f(7-2) handle Odd => ~7)

| f(9) = (f(9-2) handle Odd => ~9)

| f(11) = (f(11-2) handle Odd => ~11)

With every call new function is pattern matched, the input parameter is subtracted by 2. F(3) goes to f(1) which raises the odd exception, this is handled on line 4 since that’s the last recursively called f that handles the odd exception.

8.5 exception OddNum;

Let fun f(0, count) = count

| f(x,count) = if x =1 then-1 else f(x-2, count + 1)

I used tail recursion to optimize the program by checking if x was 1 then returning -1. Otherwise using the tail recursive call. I think this optimized the program because went from 3 lines to 2 lines and was able to remove the exception in program. Thus having less activation records.

Problem 2: Continuation passing style

(\* pow1 is a simple recursive definition of a power function \*)

fun pow1 n 0 = 1

| pow1 n m = n \* (pow1 n (m-1))

val test = pow1 5 2

fun return n = n;

(\* pow is the definition of a power function that uses continuation passing style with the type

powk : int -> int -> (int -> a) -> \*)

fun pow n 0 k = k 1

| pow n m k = (pow n ( m - 1 )) (fn v => k ( n \* v ));

val test1 = pow 6 2 return

(\* prod1 is a simple recursive definition that computes the product of a list \*)

fun prod1 [] = 1

| prod1 (x::xs) = x \* (prod1 xs)

val test2 = prod1([1,2,3,4])

(\* prod is the definition that computes the product of a list using continuation passing style with the type

prodk : int list -> (int -> a) -> a \*)

fun prod [] k = k 1

| prod (x::xs) k = (prod xs) (fn v => k ( x \* v ));

val test3 = prod([1,2,3]) return