**5**

**Why should we be more interest in functional programming than we are?**

* Computation is the evaluation of math and logic functions.
* State is not changed – no side effects (no assignment statement)
* Developed from lambda calculus
* Referential transparency- execution results are the same, regardless of time that is, results only depend on argument values loops are discarded in favor of recursive calls
* Anonymous subroutines
* Lazy evaluation (sometimes manually via above)
* Procedures are first class objects (as is data)

**What is a type system and what benefits does it provide?**

* A grouping of values into types syntax and grammar of available types & portions of parser handling type declarations & portions of compiler checking types of parameters enables coders to describe data effectively and to prevent operations that make no sense during run-time Types of operands must be checked before op is applied
* Statically typed language (Java, Haskell): all variables and expressions have fixed types. Types are checked at compile-time
* Dynamically typed language (Scheme, Python): variables and expressions do not have fixed type types are checked at run-time
* Static typing is great because it keeps you out of trouble. Dynamic typing is great because it gets out of your way and lets you get your work done faster.

A grouping of values into types enables coders to describe data effectively and to prevent operations that make no sense during run-time

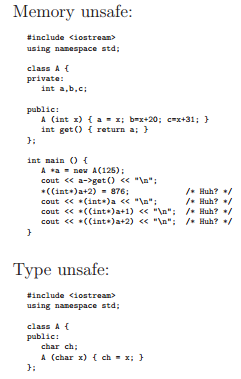
Statically typed languages are more efficient since fewer run-time type checks are needed and is more secure because what would be run-time typing errors are caught at compile time.

**How are call-by-value, call-by-reference, call-by-value-result, call-byname implemented?**

* Call-by-value: A copy of the value of the variable is made and passed to the corresponding formal parameter - variable’s value is unchanged after the procedure returns
* Call-by-reference: The location of the variable is passed to the corresponding formal parameter - the variable may be updated several times in the body of the procedure
* Call-by-value-result: Same as call-by-value except that the variable takes the value of the corresponding formal parameter at termination
* Call-by-name: The address of a function that computes the value of the actual parameter - effectively substitutes the parameter

**2. What is type safety, thread safety, memory safety and why are these  
 things important?**

* Type safety: The absence of erroneous or undesirable program behavior caused by a discrepancy between differing data types. Java provides Generics.
* Thread safety: Manipulation of data structures shared by multiple threads results in completely predictable and deterministic changes. Java provides synchronized, notify, wait
* Memory safety: aims to avoid software bugs that cause security vulnerabilities dealing with RAM access, such as buffer overflows and dangling pointers.
  + Buffer Overflow
  + Dynamic memory errors → incorrect management of dynamic memory and pointers
  + Out of memory

  
 **3. Rewrite again in continuation passing style**

Continuation passing style:  
---------------------------  
 Sample solution for findmax:  
 ----------------------------  
 (define f$   
 (lambda (l acc)  
 (if (null? l)  
 '()  
 (cons (max (car l) acc)   
 (lambda () (f$ (cdr l) (max (car l) acc)))))))  
  
 (define x (f '(3 4 5 6 2 3 4 5) 3))  
  
 (define take$  
 (lambda (n s$)  
 (if (or (zero? n) (null? s$))  
 '()  
 (cons (car s$) (take$ (- n 1) ((cdr s$)))))))  
  
 (take$ 10 x)

**4. Rewrite a more difficult function in CPS  
  
 5. How are call-by-value, call-by-value-result, call-by-reference,  
 call-by-name implemented?  
  
 6. What makes a data type a Monad?  
  
 7. Use call/cc and assert to solve a logic problem**Consider the following program after it is loaded in Scheme:

(define call/cc call-with-current-continuation)

(define succ

(lambda ()

(let ((a (call/cc (lambda (k) (cons 0 k)))))

(cons (+ (car a) 1) (cdr a)))))

(define first (succ))

Solution: (2. #[Continuation])

**8. Use foldr and foldl (Haskell) in the proper way for map-reduce**

**foldl (-) 100 [1] = 99 = ((100)-1)**

**foldl (-) 100 [1,2] = 97 = (( 99)-2) = (((100)-1)-2)**

**foldl (-) 100 [1,2,3] = 94 = (( 97)-3)**

**foldl (-) 100 [1,2,3,4] = 90 = (( 94)-4)**

**foldl (-) 100 [1,2,3,4,5] = 85 = (( 90)-5)**

**foldr (-) 100 [1] = -99 = (1-(100))**

**foldr (-) 100 [2,1] = 101 = (2-(-99)) = (2-(1-(100)))**

**foldr (-) 100 [3,2,1] = -98 = (3-(101))**

**foldr (-) 100 [4,3,2,1] = 102 = (4-(-98))**

**foldr (-) 100 [5,4,3,2,1] = -97 = (5-(102)**

**9. Why should we be more interested in functional programming than we are?**

* Scheme and Haskell are functional programming languages
* Treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
* Programming is done with expressions
* Functional programming avoids side effects unlike imperative programming

**10. Create a new data type, supporting polymorphism, and type constructor**

* **11. Use Maybe Monad to make code more generally applicable  
    
  12. Use Maybe Monad to handle exceptional cases  
    
  13. Solve a problem in Haskell using a list comprehension  
    
  14. Modify a given recursive Scheme function to be tail recursive**

**Solutions to 6-9 of Final from 2013**

(define follow

(lambda (lst x)

(if (null? lst)

'()

(if (eq? (car lst) x)

(follow (cdr lst) x)

(cons (car lst) (follow (cdr lst) x))))))

(define followF

(lambda (lst x accm)

(if (null? lst)

accm

(if (eq? (car lst) x)

(followF (cdr lst) x accm)

(followF (cdr lst) x (append accm (list (car lst))))))))

(define follow$

(lambda (lst x)

(if (null? lst)

'()

(if (eq? (car lst) x)

(follow$ (cdr lst) x);put extra parenthesis around (cdr lst) to evaluate it if using streams

(cons (car lst) (lambda() (follow$ (cdr lst) x))))))) ;; same down here

(define int-builder$

(lambda (x)

(cons x (lambda ()

(int-builder$ (+ x 1))))))

(define take$

(lambda (m s)

(if (or (= m 0) (null? s))

'()

(cons (car s) (take$ (- m 1) ((cdr s)))))))

1. What is a type system and what benefits does it provide?

**A grouping of values into types enables coders to describe data effectively and to prevent operations that make no sense during run-time**

**Statically typed languages are more efficient since fewer run-time type checks are needed and is more secure because what would be run-time typing errors are caught at compile time.**  
  
 2. What is type safety, thread safety, memory safety and why are these  
 things important?

**type safety:**

**The absence of erroneous or undesirable behavior caused by a discrepancy between differing data types**

**int main() {**

**int ival = 5;**

**void \*pval = &ival;**

**double dval = \*((double\*)pval);**

**cout << dval << endl; // 5 is not output return 0;**

**}**

**memory safety:**

**Avoid jumps to invalid data addresses and manipulation of code addresses**

**Thread safety:**

**Manipulation of data structures shared by multiple threads results in completely predictable and deterministic changes. Java provides synchronized, notify, wait**

3. Rewrite again in continuation passing style

**(define call/cc call-with-current-continuation)**

**(define succ**

**(lambda ()**

**(let ((a (call/cc (lambda (k) (cons 0 k)))))**

**(cons (+ (car a) 1) (cdr a)))))**

**(define first (succ))**

**What is the value of first after executing ((cdr first) first)?**

**(2 . #[continuation])**  
 4. Rewrite a more difficult function in CPS  
  
 5. How are call-by-value, call-by-value-result, call-by-reference,  
 call-by-name implemented?

**Call-by-value**

* A copy of the value of the variable is made and passed to the corresponding formal parameter - variable’s value is unchanged after the procedure returns

**Call-by-reference:**

* The location of the variable is passed to the corresponding formal parameter - the variable may be updated several times in the body of the procedure

**Call-by-value-result:**

* Same as call-by-value except that the variable takes the value of the corresponding formal parameter at termination

**Call-by-name:**

* The address of a function that computes the value of the actual parameter - effectively substitutes the parameter
* /\* call-by-name could look like this: \*/
* int f (int j) {
* i = j;
* int k = j;
* return k;
* }

Exam Q:

**Describe how call-by-name works?**

**The calling function passes to a called function the address of a third function that is used to compute the value of the actual parameter inside the called function.**

**Ex of call-by-name:**

**function foo(x, y) { return y+1; }**

**Now let's call it with some arguments:**

**foo(p(123),p(456))**

**x and y are going to be substituted for the parameters, so the call to foo above is going to result in:**

**return p(456)+1;**

**So we're going to print 456 to the screen and return 457. In another evaluation strategy, we would evaluate the function parameters first (printing 123 and 456 to the screen in the process) and then substitute 456 for y in the function body, eventually returning 457.**

**not-acceptable: It works great!**  
  
 6. What makes a data type a Monad?

**Francos Definition:**

**A Monad is a structure that represents computations defined as sequences**

**of steps: a type with a Monad structure (a Monadic type) defines what it**

**means to chain operations, or nest functions of that type together**

**(for example, instance Monad Maybe where ... (>>=) ... - from Real World**

**Haskell, O'Reilly, 2009).**

**Ex:**

**(>>=) :: m a -> (a -> m b) -> m b**

**(>>) :: m a -> m b -> m b**

**return :: a -> m a**

**fail :: String -> m a**

**Has 4 things:**

1. **Bind >>=**

**An infix operator with polymorphic type**

**(>>=) :: m a -> (a -> m b) -> m b**

**The first argument has monadic type. The second argument**

**is a function that maps from the underlying type of the**

**first argument to another monadic type. The type of the**

**result is in that other monadic type.**

**2.Bind >>**

**Same as above except there is no function on the right.**

**3. return**

**An identity function that maps a value in an underlying type**

**to a value in the corresponding monadic type. The polymorphic**

**type of return is**

**return :: a -> m a**

**4. fail**

**Invoked when an error is to be reported. Has polymorphic type**

**fail :: [Char] -> m a**

**My definition:**

**Monad is a computation builder that is pipelined. It is a set of instructions for a certain data type.**   
  
 7. Use call/cc and assert to solve a logic problem

**(define-syntax amb**

**(syntax-rules ()**

**((amb alt ...)**

**(let ((prev-amb-fail amb-fail))**

**(call/cc**

**(lambda (sk)**

**(call/cc**

**(lambda (fk)**

**(set! amb-fail (lambda ()**

**(set! amb-fail prev-amb-fail)**

**(fk 'fail)))**

**(sk alt)))**

**...**

**(prev-amb-fail)))))))**

**;; forces pred to be true**

**(define assert (lambda (pred) (if (not pred) (amb))))**

**Go to lab 8.**

8. Use foldr and foldl (Haskell) in the proper way for map-reduce

**listAdd x Empty = (Cell x Empty)**

**listAdd x lst = (Cell x lst)**

**appendList l lst = foldl (\acc x -> listAdd x acc) l lst**  
 9. Why should we be more interested in functional programming than we are?

**Functional Property: procedures are first-class objects**

**Feature: easier to prove correctness, prototyping is faster**

**Daniel’s Definition: Functional programming is used to help your code/software have safety. For example in Haskell you can create the type Monad. Monads are great because you can “pipeline” or create a set of instructions. These instructions can be functions or “steps”. It allows for the security of not accessing the user environment but passing a function or series of calls .**   
  
10. Create a new data type, supporting polymorphism, and type constructor

* **Code is applicable to numerous data types including those that have not yet been defined.**
* **Haskell example:**
* **reverse :: [a] -> [a]**
* **reverses a list of objects of any type.**

**This would be fine:**

**data KnapObj a = KnapObj a a deriving (Show)**

**This is extra stuff:**

**val (KnapObj x y) = x**

**wgt (KnapObj x y) = y**

**toKnapObjs lst = [ (KnapObj a b) | (a,b) <- lst ]**

11. Use Maybe Monad to make code more generally applicable

**Implement bind for Maybe (recall, m in this case is one of the type constructors of Maybe - recall also that Maybe is defined like this:**

**data Maybe a = Nothing | Just a )**

**instance Monad Maybe where Nothing >>= f = Nothing Just x >>= f = f x**

12. Use Maybe Monad to handle exceptional cases

**An instance of a Monad, for example Maybe, needs to implement the bind (>>=) operator that has type signature**

**(>>=) :: m a -> (a -> m b) -> m b where m is a Monad (recall m a means object of type a is “encased” in Monad m and can be operated on only by a function that takes type a objects as input). So, what is bind trying to accomplish?**

**Set up a pipe of functions that operate while staying in the context of the Monad m.**

13. Solve a problem in Haskell using a list comprehension

**Write a procedure prob1 of the following type:**

**prob1 :: Ord a => [a] -> Maybe a**

**which returns Nothing if the input is an empty list and otherwise returns Just N where N is the 'greatest' object in the input list. The function will be tested on lists of Ints, Chars, Chars and Strings, in that order.**

**fm [] = Nothing  
fm lst = Just (last y)  
 where y = head lst : [ max a b | a <- (drop 1 lst) | b <- y ]**

14. Modify a given recursive Scheme function to be tail recursive

**(define follow**

**(lambda (lst x)**

**(if (null? lst)**

**’()**

**(if (eq? (car lst) x)**

**(follow (cdr lst) x)**

**(cons (car lst) (follow (cdr lst) x))))))**

**Goes to:**

**(define follow**

**(lambda (lst x acc)**

**(if (null? lst)**

**acc**

**(if (eq? (car lst) x)**

**(follow (cdr lst) x acc)**

**(follow (cdr lst) x (append acc (list (car lst))))))))**

(define r  
 (lambda (l)

(if (null? l)  
 l

(append (r (cdr l)) (list (car l))))))

(define r  
 (lambda (l)

(letrec ((q (lambda (l acc)  
 (if (null? l)

acc  
 (q (cdr l) (cons (car l) acc))))))

(q l ’()))))

Evaluates to Stream

(define r$  
 (lambda (l)

(letrec  
 ((q (lambda (l acc)

(if (null? l)  
 (letrec

((p$ (lambda (l)  
 (if (null? l)

’()  
 (if (null? (cdr l))

(cons (car l) (lambda () ’()))  
 (cons (car l) (lambda () (p$ (cdr l)))))))))

(p$ acc))  
 (q (cdr l) (cons (car l) acc))))))

(q l ’()))))