# 3007 Final Exam Review

*William Findlay*

*April 22, 2018*

# Contents

# 1 Definitions

Define the following terms and provide examples or sample code as appropriate.

## 1.1 Imperative vs Declarative

### 1.1.1 Imperative

- Series of instructions
- Iterative functions
- Command driven, statement oriented
- Procedural
  - C
  - Pascal
  - Assembly
- Object oriented
  - C++
  - Java

### 1.1.2 Declarative

- No side effects
- Focus on relations
- “What to get” instead of “How to get”
- Order of statements *shouldn’t* matter
- Examples:
  - SQL
  - Prolog
  - Regex

## 1.2 Scope vs Visibility

### 1.2.1 Scope

- The set of expressions for which the variable *exists*
- In lexical scoping
  - variables in the scope we were *defined* in
  - and local variables
  - who uses this?
    - C-family languages
    - Scheme
    - Algol
- In dynamic scoping
  - variables in the scope we were *called* in
  - and local variables
  - who uses this?
    - early LISP
    - APL
    - BASH

### 1.2.2 Visibility

- The set of expressions for which the variable *can be reached*
- If we **declare a local variable** with the *same name* as a variable in enclosing scope
  - that enclosing scope variable is now hidden
  - all references to *name* are to our locally scoped variable instead

## 1.3 Lexical Scope vs Dynamic Scope

### 1.3.1 Lexical

- Function scope is enclosed in the scope which *defined us*
  - if you can't find a binding, recursively search in the function that defined you

### 1.3.2 Dynamic

- Function scope is enclosed in the scope which *called us*
  - if you can't find a binding, recursively search in the function that called you

## 1.4 Free Variables

- **Used locally** but **bound in an enclosing scope**
- In the following example:

```
(define (f x y)
  (define z 2)
  (define (g)
    (* x y z)
  )
)
```

  - `x,y,z` are free variables in `(g)`
  - `(g)` *looks them up* in its *enclosing scope*, `(f)`

## 1.5 Applicative Order Evaluation vs Normal Order Evaluation

### 1.5.1 Applicative Order Evaluation

- **Strict evaluation**
- Evaluate an expression *before* it is passed in as an argument
  - go as deep as you can until you hit primitives, then evaluate and go back
  - as deep into the nest as possible and work backwards
  - e.g.,

```
(double (* (+ 1 3) 4))
(double (* 4 4))
(double 16)
(* 16 2)
32
```

### 1.5.2 Normal Order Evaluation

- **Lazy evaluation**
- Evaluate an expression *only* when its value is needed
  - first **expand**, then **reduce**
  - e.g.,

    ```
    (double (* (+ 1 3) 4))
    (* (* (+ 1 3) 4) 2)
    (* (* 4 4) 2)
    (* 16 2)
    32
    ```

## 1.6 Special Forms

- **Exceptions** to the usual evaluation order
  - they have their own evaluation rules
  - e.g., take the first argument without evaluating right away, evaluate the second symbol right away
- Use constructs like `(delay foo)`, `(force foo)` behind the scenes

## 1.7 Tail Recursion

- **Linear iterative processes** in Scheme
- No *deferred operations*
  - **recusrive call** is the **last operation** of the procedure
- In Scheme, recursion is *tail optimized*
  - this means that it will run in *constant space*
  - number of steps will **grow linearly**, but memory will **remain constant**
- Even though the *program* is still recursive, the *process* is linear iterative because of tail-recursion optimization
- E.g., to compute a factorial using tail recursion, we do the following:

```
(define (factorial x)
  (define (iter prod i)
    (if (> i x)
      prod
      (iter (* i prod) (+ i 1))
    )
  )
  (iter 1 1)
)
```

- To compute a factorial using normal recursion, we would do the following instead:

```
(define (factorial x)
  (if (= x 1)
    x
    (* x (factorial - x 1))
  )
)
```

## 1.8 First Class and Higher Order Procedures

### 1.8.1 First Class Procedures

- When procedures (functions) behave like variables
  - procedures can be *passed as arguments* into other procedures
  - or they can be *returned* from another procedure
- E.g.,

```
(define (f g)
  (g 2)
)
(define (h x)
  (+ x 3)
)
(f h) ; this would yield (+ 2 3), which evaluates to 5
```

- This is how *closures* work
  - more on this in a following subsection

### 1.8.2 Higher Order Procedures

- A procedure which *accepts one or more procedure(s)* as argument(s)
- In other words, a procedure which *uses* the **first class procedures** property of a language
- In the above codeblock, `(f g)` is an example of a **higher order procedure**

## 1.9 Closures

- When a nested function is *returned* by its **enclosing scope**
- In practice, the returned function is typically a `lambda` (anonymous procedure)
- E.g.,

```
(define (multBy x)
  (lambda (y)(* x y)) ; lambda captures the free variable x
)

((multBy 12) 3) ; 36

(define (double) (multBy 2))
(define (triple) (multBy 3))

(double 2) ; 4
(triple 2) ; 6
```

## 1.10 Abstraction Barriers

- **Hide implementation** within complex procedures
  - user does not need to know how they work
  - they need only be guaranteed that they *will* work
- Prevents pollution of the global namespace

- Prevents excess free variables

### 1.11 Referential Transparency

- The idea that *references* can be substituted for their values without changing result of an expression
- Purely functional languages are referentially transparent
- Imperative languages are *by definition* **not** referentially transparent

### 1.12 Clause (Prolog)

- **Facts** and **rules** about the domain
- They specify truths and relations between symbols/entries in the domain
  - facts:
    - `cold(ottawa).`
    - `rainy(ottawa).`
  - rules:
    - `icy(X):- cold(X),rainy(X).`
- **Read from a file** or **asserted** in the REPL with `assert()`
- **Removed** with `retract()`

### 1.13 Unification

- Prolog attempts to unify variables, atoms, and predicates
  - predicates unify with predicates with the same number of functors and if the functors can be unified
  - variables unify with variables and atoms
  - an atom will always unify with itself
- The query succeeds if all *can be unified*
  - fails otherwise

### 1.14 Resolution

- *Algorithm* to resolve queries
- The algorithm:

```
Resolve:
  Input:  A query Q and program P
  Output: True if Q can be inferred by P, false otherwise

  Algorithm:
    Start with a goal G, initially set to Q
      Attempt to unify the first subgoal G1 from G

      If no unification possible, then backtrack
        If no backtrack possible, FAIL
      Else, extend the goal G to G' with the following:
        If unified with a rule, substitute G1 with the body of that rule
        If unified with a fact, remove G1 from F

      If G' is empty, SUCCESS
      Else, Resolve G'
```

- **If a clause unifies** with a goal, it satisfies the goal
  - a **fact** satisfies the goal immediately
  - a **rule** substitutes subgoals for the original goal
- **Backtracking** here means the following:
  - attempt another clause to satisfy the subgoal
  - if we are out of clauses to try, undo a previously satisfied subgoal and attempt to satisfy it another way
    - if we are out of subgoals, we can fail