Programming Paradigms (COMP3007) Exam Review

IMPORTANT *Structure of exam as per professor:*

- A total of 8 pages with 4 questions including sub questions
    - **Expect the following definitions and comprehension questions**
        - Iterative vs. recursive
        - Comparisons
        - Small code snippets
        - Let, lambda, define, cons
        - Coding in scheme and prolog
            - Lists, objects, streams, data abstraction, sum of squares, complex objects, local state, closure, recursion, iteration
    - **Contour diagrams**
    - In terms of prolog, queries similar to assignment #4 as well as recursion and list problems
        - Possible applications of cut

IMPORTANT Definitions and Comprehension:

Prolog is declarative programming.
Scheme is Functional programming.

Special Forms is an expression that follows special rules.
eg)Define, Let, Lambda, If, Cond, And, Or, Begin,Sequence

Applicative Order: Evaluates expressions before they are bound to a variable. This is more efficient because it can avoid multiple evaluations.

Normal Order: Evaluates expressions only when their value is needed. Unneeded code may never be evaluated.

Procedural abstraction is a way of hiding the implementation of the function using a black box method. You hide the implementation.

Lexical Scoping: variables in the containing scope are visible within the nested scopes. Scoping can be nested to any depth.

Linear Recursive Process: Expansion followed by contraction. There is a chain of deferred operations(*)The interpreter keeps track of operations to perform later. The amount of information grows linearly.

Linear Iterative Process: State can be summarized by a fixed number of state variables together with a rule on how to update them. The end test is optional. The number of steps grows linearly.

Scheme executes an iterative process in constant space which is called tail-recursive.

Abstraction barriers: isolate difference levels of the system.
Referentially Transparent: A language that supports the notion that references can be substituted for their values, without hanging the result of an expression.

Imperative Programming: Programming that makes extensive use of assignment

Binding: is an association of a name with a value.

Bound variable is a variable for which binding exists.
Free variable is used locally but is bound in an enclosing scope.
The duration of time that a variable is bound is called its extent.
A **frame** is a table of bindings.
An **environment** is a sequence of frames. The **root frame** is the primitive environment.

Contour Model a graphical model of runtime environments for block structured programming languages. Model consists of a set of contours each corresponding to a given environment both lexical and dynamic.

Lexical Scoping: The environments are searched in the order matching the nesting of their definitions.
Dynamic scoping: The environments are searched in the reverse order of invocation.

The **interpreter** which determines the meaning of expressions in a programming language is just another program.
An **evaluator** that is written in the same language that it evaluates is said to be metacircular.
-Show how to define the operational semantics of the language
-Provides an executable specifications for the language
-Demonstrates how bindings and environments work
-Provides a non-trivial test-case for the language developed.
Two main **Functions for the Scheme Metacircular Interpreter**:

1. Eval and then 2. Apply

Prolog has only one data type: Terms: Number, Atom (any symbol that starts with a lowercase, constant value), Variable(any symbol that starts with an uppercase, un specified value), and Compound Term(a tuple of terms tagged with a relation name)

Two **Forms of Clauses**.

1. A **fact** is a statement that is universally true.
2. A **rule** is a statement that is conditionally true.

Cuts: how to stop prolog from continuing to succeed.

1. Faster Execution
2. Fewer backtrack points

Green cut: only prune computational paths that do not lead to new solutions. Cuts that do not affect the programs meaning.

Red cut: Cuts whose presence in a program changes the meaning of the program.

IMPORTANT Studying Components:

1. Lecture material on *COMP3007* website
2. Assignment material + Contour diagram examples
3. Midterm material located in appendix and test.pdf in current folder

Lecture Material Notes:

Introduction:

Program - a thing that might be edited with a text editor
Process - an abstract computation to which the program gives rise, upon being executed.
Domain - subject matter that the process is about.

Attributes of a good programming language:

1. Readability and clarity
2. Writability
3. Naturalness for application
4. Reliability
5. Ease of program verification
6. Programming environment
7. Portability
8. Cost

Types of languages:

Imperative programming: command driven, sequence of statements (C, Pascal).
Procedural programming: provides modularity, subroutines (C, Pascal).
OO Programming: data encapsulated, complex objects built from simple ones, inheritance, polymorphism (C++, Java).
Declarative programming: focus on statements, does not describe the control flow (SQL, Prolog).
Functional programming: primitive functions to build complex ones, evaluate mathematical functions (Scheme).
Logic programming: facts and logical rules, filters applied to data, queries (Prolog).
Meta programming: genetic programming, compilers.
Parallel programming: division of labour, multiple processes, multiple partial solutions.
Event driven programming: proceed in response to events, handlers registered to events (User interfaces).
Visual programming: programs display visually (XCode).

Programming languages are comprised of:

1. Primitive expressions
2. Means of combination
3. Means of abstraction

Scheme:

The part of the program where a binding applies is the **scope** of the binding.
A set a bindings in memory is called the **environment**.

Evaluation rule:

1. Evaluate subexpressions of combination
2. Apply procedure to resulting arguments

Terminals: operators or numbers
Nodes: combinations

Applicative-Order evaluation (Scheme approach): Evaluates expressions before they're bound to a variable

1. Evaluate operator and operand subexpressions
2. Apply the resulting procedure to the resulting arguments

Normal-Order evaluation: Evaluates expressions only when their value is needed

1. Evaluate the operator and apply to (unevaluated) operands
2. Repeat until reaching *primitive* operators
3. Then evaluate operands and apply the primitive operators

Which is better?
Applicative-order is more efficient therefore **wins this debate**-> avoids multiple evaluations
Normal-order avoids unneeded code being evaluated

Procedural Abstraction:

Conditional expressions and predicates are either true or false, they are used by using **special forms** such as **cond** and **if** statements.

A problem broken up into smaller problems is called a **decomposition** strategy, hidden functionality (black box) is creating a **procedural abstraction**.

Formal parameter of procedure has a special role, it doesn't matter what name of the formal parameter has

- Such a name is called a **bound variable**, the procedure definition **binds** its formal parameters

The set of expressions for which binding defines a name is called the **scope** of the name

- A definition of a function creates a new *local* scope

Variables being used in a function while being unbound in the local scope are considered **free** variables
Environments consist of many nested scopes

Lexical scoping consists of:

- Variables in the containing scope are visible within nested scopes
- Free variables can be used or hidden in the nested scopes
- Scoping can be nested to any depth

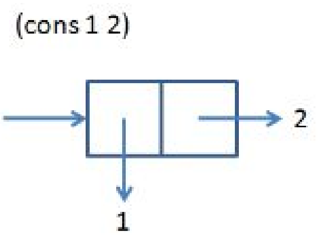
Recursive process is expansion by contraction, a chain of deferred operations, interpreter keeps track of operations to perform later, information grows linearly
Iterative process is a fixed number of state variables with a rule to update them, end test is optional, steps grow linearly
Scheme executes an iterative process in constant space (tail-recursive)

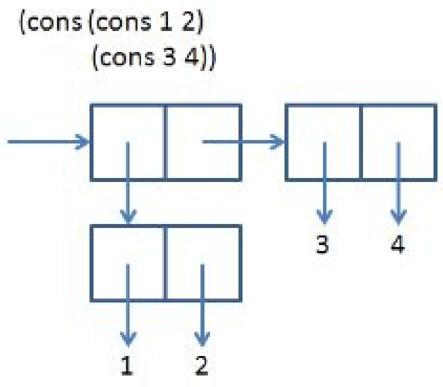
Special form let: syntactic sugar for using lambda to create local variables

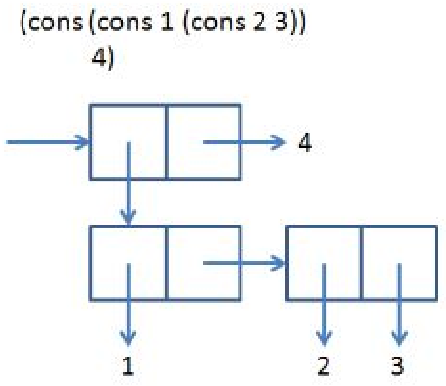
<u>Building Abstractions with Data:</u>
Methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects, enabling one to replace/swap/switch implementations

- Inferface between two parts of system will be a set of procedures called **selectors** and **constructors**

Pairs: created by procedure **cons** with **car** and **cdr** fields







Abstraction barriers isolate different levels of the system

Find the nth element? **Cdring down**
Append two lists? **Consing up**
Scaling numbers? **Mapping over lists**
Selecting numbers? **Filtering over lists**
Summation of numbers? **Accumulating over lists**

Equality:
What does it mean for two symbols to be equal?
= <- value comparison of **numbers only**
Eq? <- identity comparison. Compares references, *may* not work for primitives

Eqv? <- like eg? For objects and = for primitives
Equal? <- value comparison for objects

Streams:
Stream is a sequence with *delayed evaluation on demand*
- Use **delay** and **force**

Mutable State, Environments, and Objects:
A language that supports the notion that references can be substituted for their values, without changing the result of an expression, is said to be **referentially transparent**.

Special forms include **set!** and **begin**

A **binding** is an association of a name with a value
- A **bound variable** is a variable for which binding exists
- A **free variable** is used locally but is bound in an enclosing scope
- The duration of a variable that is bound is called its **extent**
- A **frame** is a table of bindings
- An **environment** is a sequence of frames

Contour Model:
- Sequence of snapshots depicted of the program

Scoping includes **lexical scoping** where environments are searched in the order matching the nested of their definitions, if a variable is not found in the current environment, search is continued in the procedure in which current procedure is defined and so on. Determined at **compile time. Dynamic scoping** environments are nested in reverse order of function invocation, if a variable is not found in current environment, then search in environment of function which called the current function, scope determined at **runtime**.

Design of complex systems should be modular to facilitate maintenance, extension and reusability.
Computational **objects** Require:
1. Their own persistent local state variables
2. Behaviours which change that state over time
3. A public interface to invoke those behaviours

Meta-Circular Interpreter:
Interpreter for a programming language is a process that when applied to an expression of the language, performs the actions required to evaluate that expression.
- Just another program

Useful because:
- Shows how to define the operational semantics of the language
- Executable specification for the language
- Demonstrates how bindings work

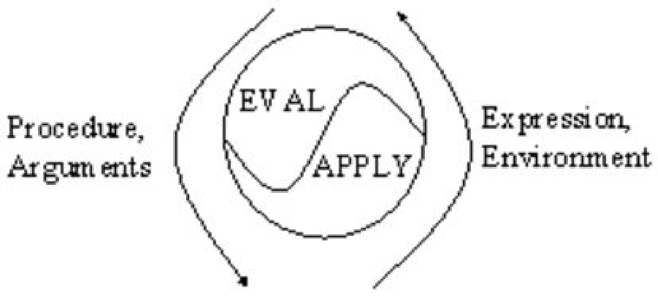
Contains two main functions:

Eval to evaluate a combination of subexpressions and then apply the value of the operator subexpression to the values of the operand subexpressions.

- Takes an **expression** and an **environment**
- Classifies expression as a **primitive, special form, combinations**

Apply a compound procedure to a set of arguments, evaluate the body of the procedure **in a new environment**.

- Takes a procedure and a list of arguments to which the procedure should apply, classify into either primitive or compound



<u>Prolog:</u>

Approach is to express programs in a form of symbolic logic and use a logical inferencing process to produce results, a form of **logic programming** or declarative. Uses **declarative semantics**

- Simple way of determining the meaning of each statement
- Meaning can be determined from statement itself

Execution is *non-procedural*

- Not concerned with the process of evaluation, just the result
- Set of facts defining relationships between objects

Used when application involves pattern matching with backtacking search on incomplete information

Algorithm = Logic + Control

Logic = Facts and rules

Control = How to apply the rules

Specified **relations** give rules and facts about a domain by declaring relations of symbols

Eg. owns(John,Book)

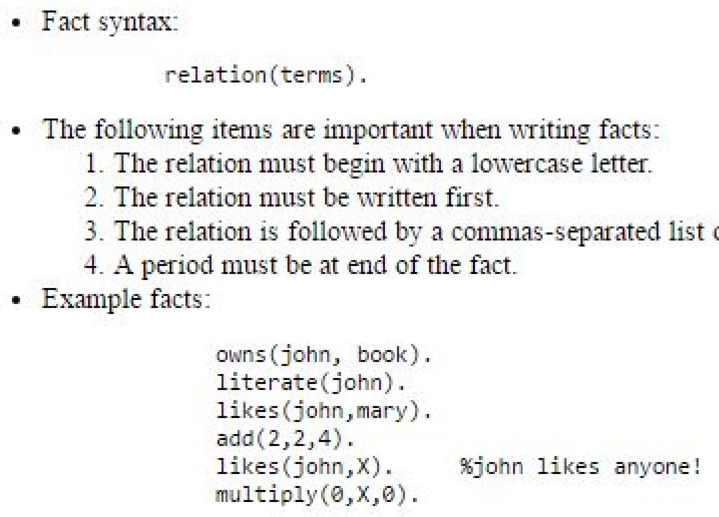
Also called **predicates** because the relation name can be thought of as a test of the form “is a given tuple in the relation”

Queries are questions based on clauses and **relations**

- Prompted by ?-
-

Terms include numbers, atom (any symbol with a lowercase letter is a constant value), variable (any symbol with an uppercase letter is a unspecified value), compound term (a tuple of terms tagged with relation name).

The following defines **facts:**



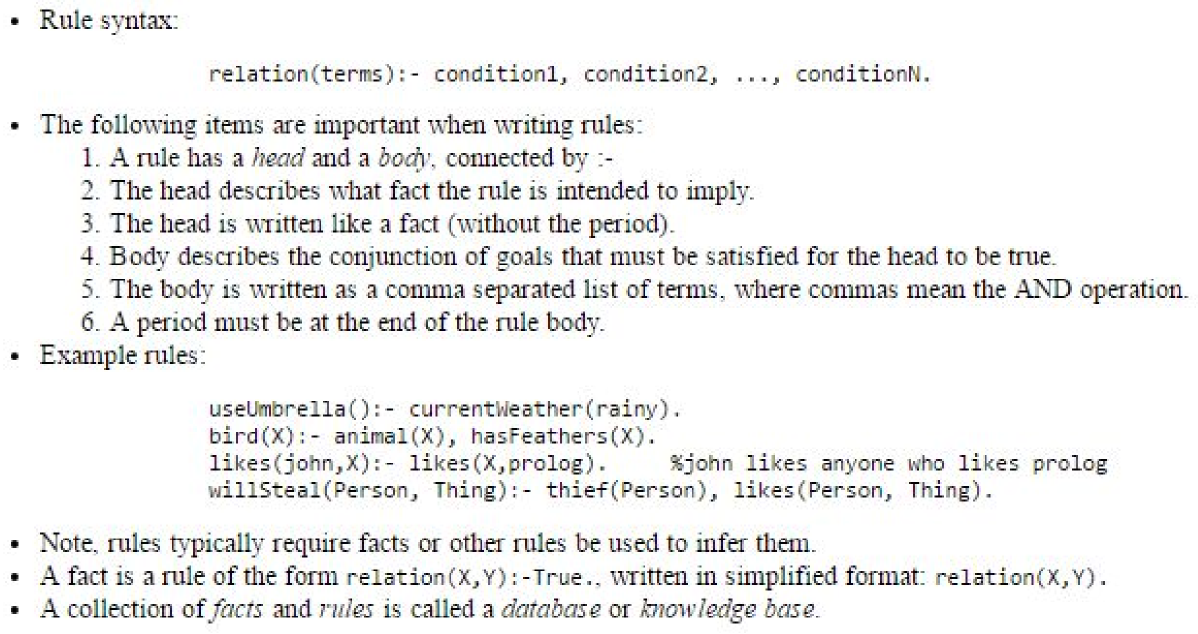
- Fact syntax:

```
relation(terms).
```

- The following items are important when writing facts:
  1. The relation must begin with a lowercase letter.
  2. The relation must be written first.
  3. The relation is followed by a commas-separated list of terms in round brackets.
  4. A period must be at end of the fact.
- Example facts:

```
owns(john, book).
literate(john).
likes(john,mary).
add(2,2,4).
likes(john,X).     %john likes anyone!
multiply(0,X,0).
```

The following defines **rules:**



- Rule syntax:

```
relation(terms):- condition1, condition2, ..., conditionN.
```

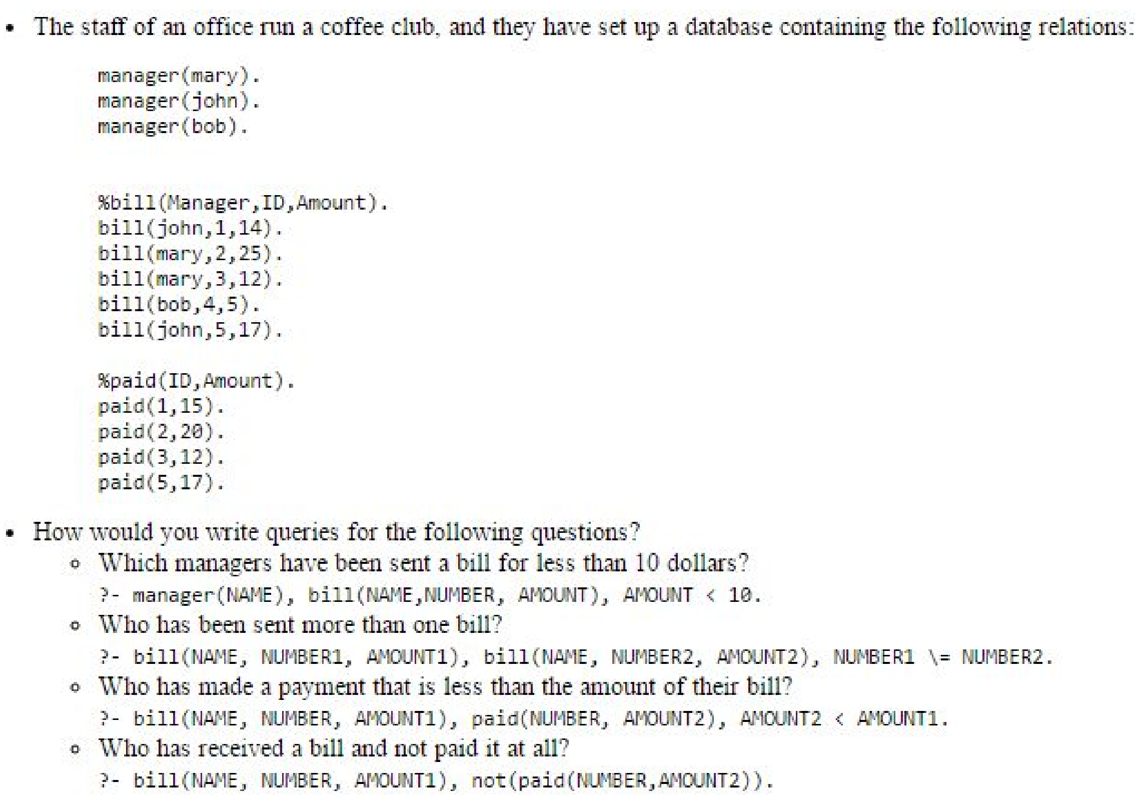
- The following items are important when writing rules:
  1. A rule has a *head* and a *body*, connected by :-
  2. The head describes what fact the rule is intended to imply.
  3. The head is written like a fact (without the period).
  4. Body describes the conjunction of goals that must be satisfied for the head to be true.
  5. The body is written as a comma separated list of terms, where commas mean the AND operation.
  6. A period must be at the end of the rule body.
- Example rules:

```
useUmbrella():- currentWeather(rainy).
bird(X):- animal(X), hasFeathers(X).
likes(john,X):- likes(X,prolog).     %john likes anyone who likes prolog
willSteal(Person, Thing):- thief(Person), likes(Person, Thing).
```

- Note, rules typically require facts or other rules be used to infer them.
- A fact is a rule of the form `relation(X,Y):-True.`, written in simplified format: `relation(X,Y).`
- A collection of *facts* and *rules* is called a *database* or *knowledge base*.

To add a fact or rule to the prolog environment use the **assert** predicate
assert (likes(john,mary)).
assert(likes(john,X): likes(X,prolog))
To remove a fact or rule from the prolog environment use the **retract** predicate
retract (likes(john,mary)).
retract(likes(john,X):-likes(X,prolog)).

Example of prolog in a nutshell:



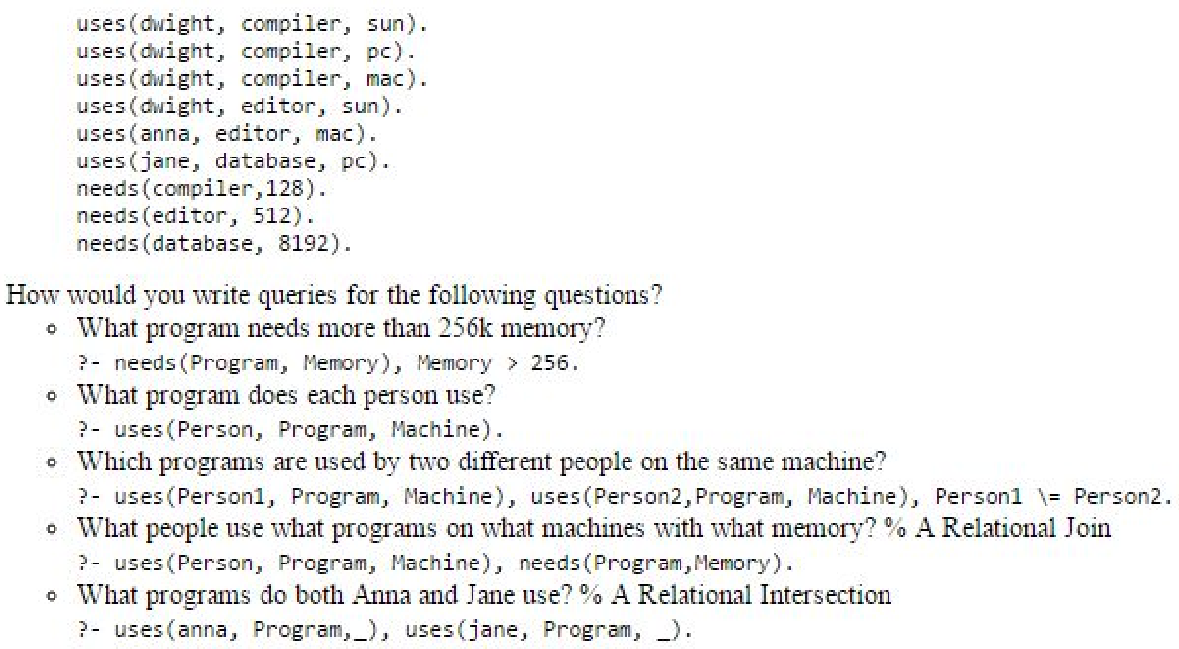
- The staff of an office run a coffee club, and they have set up a database containing the following relations:

```
manager(mary).
manager(john).
manager(bob).


%bill(Manager,ID,Amount).
bill(john,1,14).
bill(mary,2,25).
bill(mary,3,12).
bill(bob,4,5).
bill(john,5,17).

%paid(ID,Amount).
paid(1,15).
paid(2,20).
paid(3,12).
paid(5,17).
```

- How would you write queries for the following questions?
  - Which managers have been sent a bill for less than 10 dollars?
    ```
    ?- manager(NAME), bill(NAME,NUMBER, AMOUNT), AMOUNT < 10.
    ```
  - Who has been sent more than one bill?
    ```
    ?- bill(NAME, NUMBER1, AMOUNT1), bill(NAME, NUMBER2, AMOUNT2), NUMBER1 \= NUMBER2.
    ```
  - Who has made a payment that is less than the amount of their bill?
    ```
    ?- bill(NAME, NUMBER, AMOUNT1), paid(NUMBER, AMOUNT2), AMOUNT2 < AMOUNT1.
    ```
  - Who has received a bill and not paid it at all?
    ```
    ?- bill(NAME, NUMBER, AMOUNT1), not(paid(NUMBER,AMOUNT2)).
    ```



```
uses(dwight, compiler, sun).
uses(dwight, compiler, pc).
uses(dwight, compiler, mac).
uses(dwight, editor, sun).
uses(anna, editor, mac).
uses(jane, database, pc).
needs(compiler,128).
needs(editor, 512).
needs(database, 8192).
```

How would you write queries for the following questions?
- What program needs more than 256k memory?
  ```
  ?- needs(Program, Memory), Memory > 256.
  ```
- What program does each person use?
  ```
  ?- uses(Person, Program, Machine).
  ```
- Which programs are used by two different people on the same machine?
  ```
  ?- uses(Person1, Program, Machine), uses(Person2,Program, Machine), Person1 \= Person2.
  ```
- What people use what programs on what machines with what memory? % A Relational Join
  ```
  ?- uses(Person, Program, Machine), needs(Program,Memory).
  ```
- What programs do both Anna and Jane use? % A Relational Intersection
  ```
  ?- uses(anna, Program,_), uses(jane, Program, _).
  ```

Operators

Equality

X = Y Succeeds if X and Y unify

X \= Y Succeeds if X cannot unify to Y

X < Y, X > Y, X =< Y, X >= Y Numbers only, succeeds if comparison is true

Arithmetic

X + y * z can be expressed as +(x, *(y,z))

X = 3 + 4 -> X = 3+ 4
Standard operators: X + Y, X * Y, X / Y, X mod Y

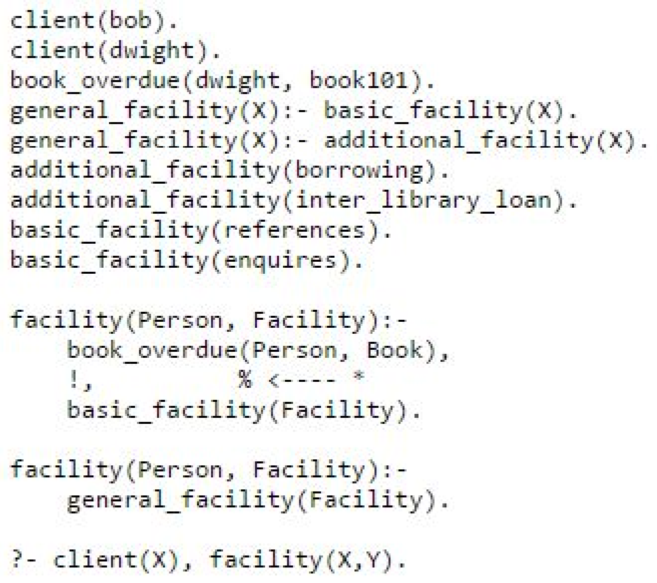
Backtracking can be done using % sign. For example:
likes(mary,food)
likes(mary,wine)
likes(john,wine)
likes(john,mary)

?- likes(mary,X), likes(john,X)
X = food
Fails because john doesn't like food so backtrack to re-satisfy goal
john(likes,wine) #t

Prolog Cuts:
How to stop Prolog from succeeding?
Use **cut,** commits system to every choice it has made since it chose the rule
1. Faster execution, don't spend time satisfying goals that can't be satisfied
2. Fewer backtrack points need to be consider, therefore less memory

Common uses of cut include telling Prolog it has found the appropriate goal and thus not to go farther, terminate backtracking, fail without trying alternatives

Example



```
client(bob).
client(dwight).
book_overdue(dwight, book101).
general_facility(X):- basic_facility(X).
general_facility(X):- additional_facility(X).
additional_facility(borrowing).
additional_facility(inter_library_loan).
basic_facility(references).
basic_facility(enquires).

facility(Person, Facility):-
    book_overdue(Person, Book),
    !,          % <---- *
    basic_facility(Facility).

facility(Person, Facility):-
    general_facility(Facility).

?- client(X), facility(X,Y).
```

If a client is found to have an overdue book, then only allow the client the basic facilities of the library. Don't bother going through all the client overdue books and don't consider any rule about facilities.

Example 2

```
%people database
person(alice, 20).
person(bob, 21).
person(charlie, 23).
person(dave, 20).

hobby(alice, birdwatching).
hobby(bob, larping).
hobby(charlie, larping).
hobby(charlie, birdwatching).
hobby(dave, birdwatching).

%true if anyone could be consider a friend of Person1
has_friend(Person1):-
        hobby(Person1,Hobby),
        hobby(Person2,Hobby),
        Person1 \= Person2,
        AgeDiff is abs(Age2-Age1),
        person(Person1,Age1),
        person(Person2,Age2),
        AgeDiff =< 2,!. %got a match, so stop looking
```

Example 3

- **Idea**: Fail and prevent backtracking from taking place
- **Format**: !, fail

```
    factorial(X, _):- X < 0, !, fail. %fail and do not backtrack
    factorial(0, 1):- !.
    factorial(X, Y):- Z is X - 1, factorial(Z, W), Y is X*W.
```

- We can implement NOT in terms of cut and fail

```
    not(P):- P, !, fail.
    not(P).
```

- The first rule above states that if we can satisfy P then !, fail. Therefore, do not try to prove not(P).
- The fact will only be reached if P cannot be satisfied.

Green cuts

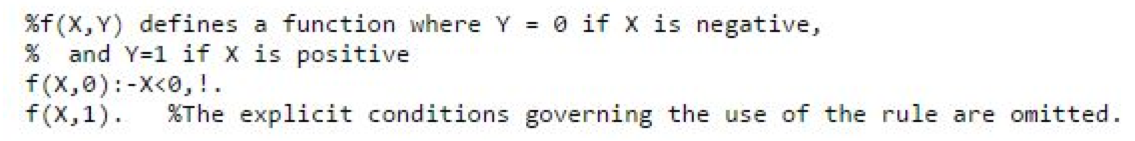
- Addition and removal cuts from a program do not affect the program's meanings. Green cuts only prune computational paths that do not lead to new solutions

```
%f(X,Y) defines a function where Y = 0 if X is negative,
%  and Y=1 if X is positive
f(X,0):-X<0,!.
f(X,1):-X>=0.
```

- If you solve one way, there's no need to attempt the other way.

Red cuts

- Cuts whose presence in a program changes the meaning of the program, e.g., the removal of the cut changes its meaning
- A standard Prolog programming technique using red cuts is the omission of explicit conditions



```
%f(X,Y) defines a function where Y = 0 if X is negative,
%  and Y=1 if X is positive
f(X,0):-X<0,!.
f(X,1).   %The explicit conditions governing the use of the rule are omitted.
```

Assignment material/Contour Diagram Examples:

Assignment 1

Question 2

A function f is defined by the rules:
f(n) = n, if n<4
f(n) = 4f(n-1) + 3f(n-2) + 2f(n-3) + 1f(n-4), otherwise

1. [5 marks] Write a procedure that computes f by means of a *recursive process*. Illustrate that your answer is recursive by showing the substitution model for (f 5) in comments below your code.
2. [10 marks] Write a procedure that computes f by means of an *iterative process* (i.e., a tail-recursive program). Illustrate that your answer is iterative by showing the substitution model for (f 5) in comments below your code.

Question 4

[10 marks] Consider the definition of general summation of numbers between two values as describe in lecture:

```
(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
       (sum term (next a) next b))))
```

Given helper functions inc and identity one can, for example, define a function that sums all integers from a to b:

```
(define (inc x) (+ x 1))
(define (identity x) x)
(define (sum-integers a b)
         (sum identity a inc b))
```

The general sum procedure above generates a linear recursive process.
Rewrite it as an iterative process.

Assignment 2

Question 1 [7 marks]

An interval is defined by an upper and lower bound. You are required to write the procedures: **add-interval, subtract-interval, multiply-interval, and divide-interval** that add, subtract, multiply, and divide two intervals respectively. You should create a **make-interval** procedure, along with procedures to access the upper and lower bounds. You must deal with intervals that span zero.
Addition: [a,b] + [c,d] = [a+c,b+d]
Subtract: [a,b] - [c,d] = [a-d,b-c]

Multiply: [a,b] * [c,d] = [min(ac,ad,bc,bd), max(ac,ad,bc,bd)]
Divide: [a,b] / [c,d] = [a,b] * [1/d,1/c] if [c,d] does not contain 0, otherwise error

Question 3 [7 marks total]
Here is an alternative procedure representation of pairs:

```
(define (special-cons x y)
  (lambda (m) (m x y)))
```

(a) [4 marks] What are the corresponding definitions of **special-car** and **special-cdr**? (Note: do not use names that conflict with the existing cons and car). For this representation, verify that (special-car (special-cons x y)) yields x for any objects x and y and (special-cdr (special-cons x y)) yields y.
(special-car (special-cons 1 2))→ 1
(special-cdr (special-cons 1 2))→ 2

(b) [3 marks] Create a procedure (triple x y z) that constructs a triplet. You may not use cons, car, cdr or lists in the triplet procedures. Next, write procedures first, second, third that return the first, second, third element respectively. E.g.:
(define a (triple 1 2 3))
(first a) → 1
(second a) → 2
(third a) → 3

Assignment 3
Question 3
[10 marks] Given the following code:

```
01|(define (outer z)
02|  (define x 2)
03|  (define (in1)
04|    (define z (+ 50 x))
05|    (in2))
06|  (define (in2)
07|    (set! z (* z 10))
08|    z)
09|  in1)
10|
11|(define closure (outer 30))
12|(closure)
```

1. [/6 marks] Draw a contour diagram at the start of line 5 (after calling the closure function on line 12, before calling in2 on line 5).
2. [/1 mark] What is the output of this code? (Using lexical scoping)
3. [/3 marks] Would this code work using dynamic scope as taught in lecture? If yes, provide the output. If no, explain why not.

Question 4

[20 marks] Add the special form **let** to the metacircular interpreter. Be sure to clearly label the changes you made. Hint: Remember let is just syntactic sugar for a lambda expression, so all that needs to be done is converting the let into a corresponding lambda.

Assignment 4
Question 2
[15 marks total] Given the following database of facts:
actor(jonny, depp, gender(male)).
actor(bruce, willis, gender(male)).
actor(glenn, close, gender(female)).
actor(orlando, bloom, gender(male)).
actor(jennifer, lawrence, gender(female)).
actor(sean, bean, gender(male)).
actor(angelina, jolie, gender(female)).
actor(keira, knightley, gender(female)).
actor(benedict, cumberbatch, gender(male)).

movie(year(2003), title([pirates,of,the,carribean]), cast([actor(jonny, depp), actor(keira, knightley), actor(orlando, bloom)])).
movie(year(2001), title([lord,of,the,rings]), cast([actor(orlando, bloom), actor(sean, bean)])).
movie(year(1988), title([die,hard]), cast([actor(bruce, willis)])).
movie(year(2014), title([the,imitation,game]), cast([actor(benedict, cumberbatch), actor(keira, knightley)])).
movie(year(2012), title([the,hunger,games]), cast([actor(jennifer,lawrence)])).

Write prolog queries that answer the following questions:
1. [1 mark] What movie(s) contain the word "of" in the title?
2. [1 mark] What movies were released in or after 2001?
3. [2 marks] What movies share one or more common words in their titles?
4. [1 mark] What are the names of the female actors?
5. [2 marks] In what movies is Orlando Bloom a member of the cast?
6. [2 marks] What actor(s) are in the cast of more than 1 movie?
7. [2 marks] What is the title of the oldest movie?
8. [2 marks] What actor(s) are not in the cast of any movie?

Hint: You may want to use the member/2 predicate in some of your answers. (Called contains in the notes).
For submission, include your queries as comments in the database, together with the output generated when you tested them.
Note: your solutions should be able to answer the provided questions even if the database of facts were changed.

Question 4
[15 marks total]
1. [3 marks] Write a recursive predicate to find the last element of a list. You may *not* use the built-in last predicate in your answer. E.g.,

2. ?- lastEle(X,[how,are,you,today]).
   X=today.
3. [3 marks] Write a predicate after(X,List,Result) that returns everything in a list after any occurrence of the given element X. E.g.,
4. ?- after(a, [b,a,x,d,a,f,g], R).
    R = [x,d,a,f,g] ;
    R = [f,g].

5. [4 marks] Write a predicate nextto(X, Y, L), that succeeds when elements X and Y are immediately consecutive elements of a list L. E.g.,
6. ?- nextto(a,b, [c,a,b,d]).
   True.
   ?- nextto(a,b, [c,a,d,b]).
   False

7. [5 marks] Write the predicate occurs_at_position(Element, List, Position), that allows access to the nth element in a list. E.g.,
8. ?- occurs_at_position(x, [a,b,c,x,x,d,e], 4).
   True
   ?- occurs_at_position(x, [a,b,c,x,x,d,e],Pos).
   Pos = 3;
   Pos = 4;
   ?- occurs_at_position(X, [a,b,c,x,x,d,e],2).
   X = c

Contour Diagram Examples

Problem 1:

```
line 1: (define b 5)
line 2: (define (square num)
line 3:      (* num num))
line 4: (define result (square 5))
```

Problem 2:

```
Line 1: (define z 2)
Line 2: (define (fun x)
Line 3:     (lambda (y) (* x y z)))
Line 4:
Line 5: (define closure1 (fun 3))
Line 6: (define closure2 (fun 4))
Line 7:
Line 8: (define r1 (closure1 5))
Line 9: (define r2 (closure2 5))
```

IMPORTANT Direct Questions from Fall 2016 COMP 3007

1. What is the definition of linear recursive programming?
2. What is the definition of linear iterative programming?
3. Implement stream-cons, stream-car, and stream-cdr
4. Implement a Stack using cons pairs. Ensure that the stack has the operations push, pop, peek, and size.
5. Using the following facts:
   a. male(X) % X is male
   b. female(X) % X is female
   c. father(X,Y) %X is the father of Y
   d. mother(X,Y) %X is the mother of Y
   e. married(X,Y) %X is married to Y

   Write prolog rules to define the following 11 relationships:
   1. parent(X,Y) %X is the parent of Y
   2. different(X,Y) %X and Y are different
   3. is_mother(X) % X is a mother
   4. is_father(X) % X is a father
   5. aunt(X,Y) % X is an aunt of Y
   6. uncle(X,Y) % X is an uncle of Y
   7. sister(X,Y) % X is a sister of Y
   8. brother(X,Y) % X is a brother of Y
   9. grandfather(X,Y) % X is a grandfather of Y
   10. grandmother(X,Y) % X is a grandmother of Y
   11. Ancestor(X,Y) % X is ancestor of Y

6. Consider this database of facts, describing what something is made up of.

```
has(bicycle,wheel,2).
has(bicycle,handlebar,1).
has(bicycle,brake,2).
has(wheel,hub,1).
has(bicycle,frame,1).
has(car,steering_wheel,1).
has(car,stereo,1).
```

Write a predicate partof(X,Y) that succeeds if Y is part of X.

```
?- partof(wheel,spoke).
True.
?- partof(bicycle,spoke).
True.
?- partof(car,spoke).
False.
```

partof(X,Y) can also be used to enu

merate the parts that make up an object or of which an object is part of.

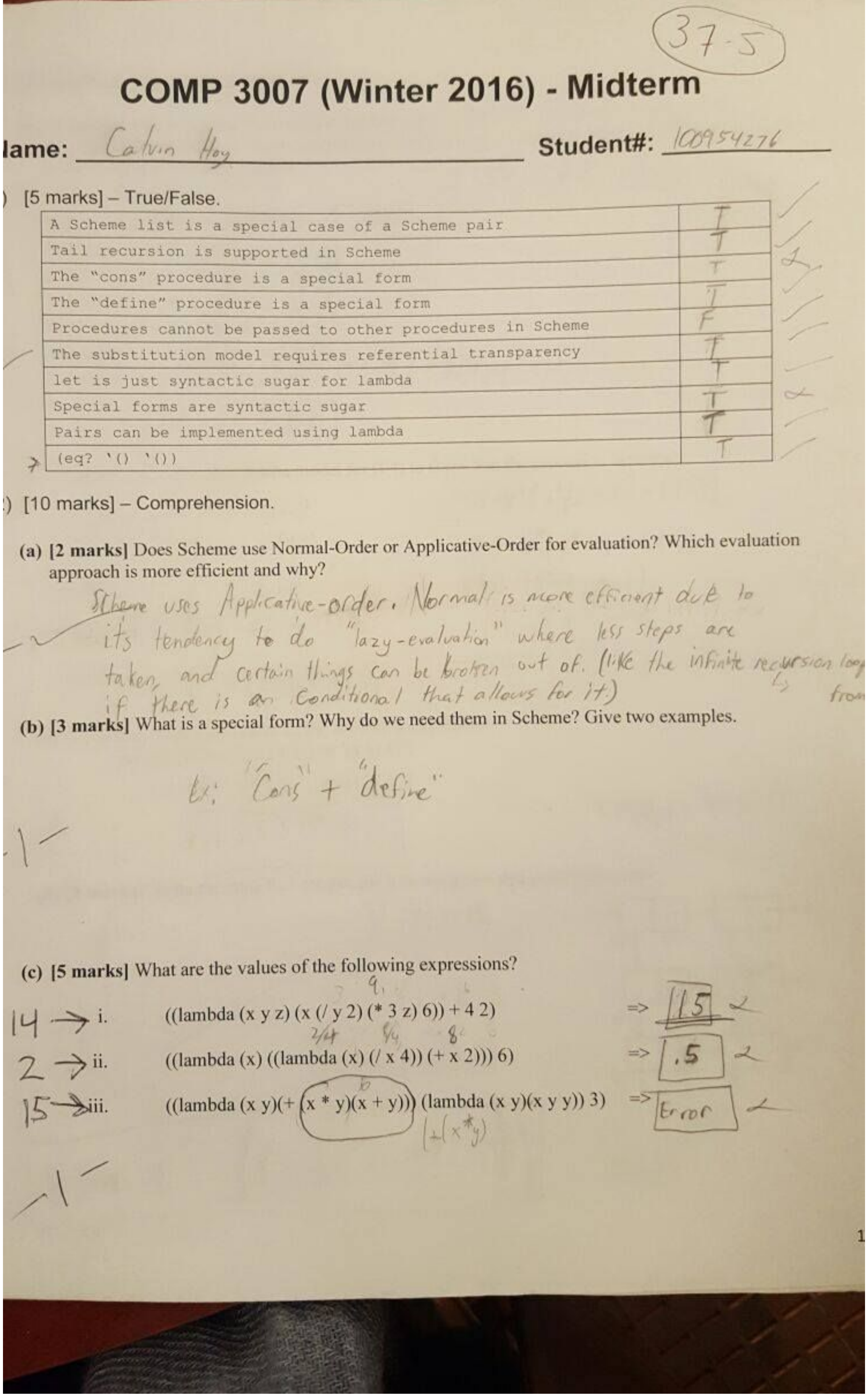
```
?- partof(bicycle,X).
X = wheel;
X = handlebar;
X = break;
X = frame;
X = hub;

?- partof(X,spoke).
X = wheel;
X = bicycle.
```

7. Missing lines from the meta-circular interpreter and you fill it in.

8. Create infinite streams with force and delay.

APPENDIX



37.5

COMP 3007 (Winter 2016) - Midterm

lame: Calvin Hoy **Student#:** 100954276

) [5 marks] – True/False.

A Scheme list is a special case of a Scheme pair	T
Tail recursion is supported in Scheme	T
The "cons" procedure is a special form	T
The "define" procedure is a special form	T
Procedures cannot be passed to other procedures in Scheme	F
The substitution model requires referential transparency	T
let is just syntactic sugar for lambda	T
Special forms are syntactic sugar	T
Pairs can be implemented using lambda	T
(eq? '() '())	T

) [10 marks] – Comprehension.

(a) **[2 marks]** Does Scheme use Normal-Order or Applicative-Order for evaluation? Which evaluation approach is more efficient and why?

Scheme uses Applicative-order. Normal is more efficient due to its tendency to do "lazy-evaluation" where less steps are taken, and certain things can be broken out of. (like the infinite recursion loop from if there is an Conditional that allows for it)

(b) **[3 marks]** What is a special form? Why do we need them in Scheme? Give two examples.

Ex: "Cons" + "define"

-1

(c) **[5 marks]** What are the values of the following expressions?

14 → i. ((lambda (x y z) (x (/ y 2) (* 3 z) 6)) + 4 2) => $\boxed{15}$

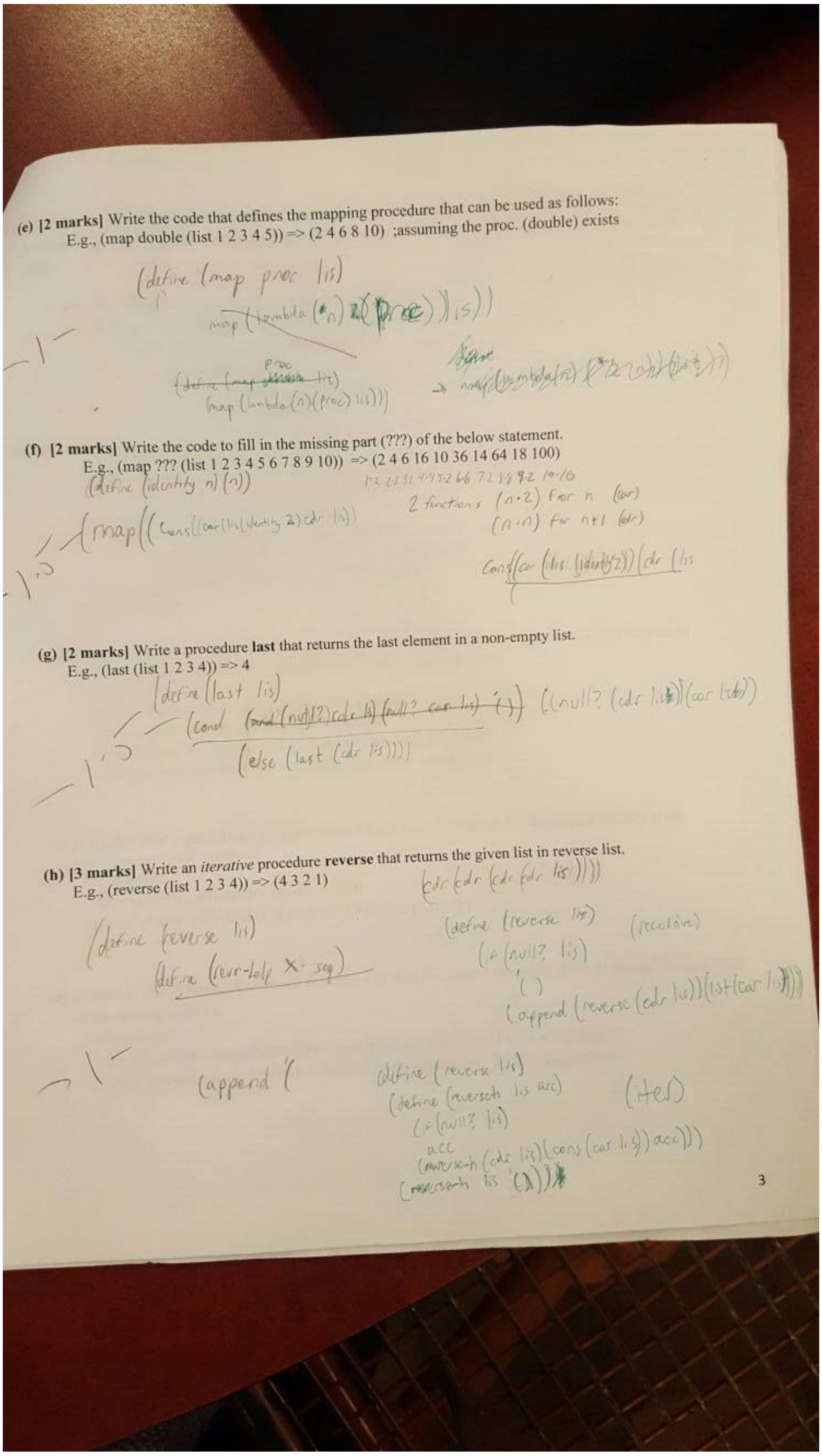
2 → ii. ((lambda (x) ((lambda (x) (/ x 4)) (+ x 2))) 6) => $\boxed{.5}$

15 → iii. ((lambda (x y)(+ (x * y)(x + y))) (lambda (x y)(x y y)) 3) => $\boxed{\text{Error}}$

(+(x*y)

-1

1



(e) [2 marks] Write the code that defines the mapping procedure that can be used as follows:
E.g., (map double (list 1 2 3 4 5)) => (2 4 6 8 10) ;assuming the proc. (double) exists

```
(define (map proc lis)
  ~~map (lambda (n) (proc)) lis))~~

-1

(~~define (map [illegible] lis)~~
 proc
(map (lambda (n)(proc) lis)))

→ map (lambda (n) (* 2 n)) (list))
```

(f) [2 marks] Write the code to fill in the missing part (???) of the below statement.
E.g., (map ??? (list 1 2 3 4 5 6 7 8 9 10)) => (2 4 6 16 10 36 14 64 18 100)

```
(define (identity n) (n))
                                1·2 2·2 3·2 4·4 5·2 6·6 7·2 8·8 9·2 10·10
                                2 functions (n·2) for n (car)
                                            (n·n) for n+1 (cdr)
(map ((cons (car (lis (identity 2) cdr lis))

-1.5
                                cons ((car (lis (identity 2)) (cdr (lis
                                (
```

(g) [2 marks] Write a procedure **last** that returns the last element in a non-empty list.
E.g., (last (list 1 2 3 4)) => 4

```
(define (last lis)
  (cond ~~(and (null? cdr lis) (null? car lis) '()~~)) ((null? (cdr lis)) (car lis))
        (else (last (cdr lis))))

-1.5
```

(h) [3 marks] Write an *iterative* procedure **reverse** that returns the given list in reverse list.
E.g., (reverse (list 1 2 3 4)) => (4 3 2 1)

```
                                (cdr (cdr (cdr (cdr lis))))

(define (reverse lis)           (define (reverse lis)        (recursive)
  (define (revr-help x seq)       (if (null? lis)
                                    '()
                                    (append (reverse (cdr lis)) (list (car lis)))))

-1

   (append '(                   (define (reverse lis)
                                  (define (reverse-h lis acc)       (iter)
                                    (if (null? lis)
                                      acc
                                      (reverse-h (cdr lis) (cons (car lis)) acc)))
                                  (reverse-h lis '()))
```

3) [20 marks] – Short coding questions.

(a) [5 marks] You are given the recursive function, f(n) = 2*f(n-1) + f(n-3), for n>2 and where f(0)=0, f(1)=1 and f(2)=2. Write a procedure called (f n) that computes the function for all values of n > 0 using a linear iterative process

```
(define (fn n)
  (define (iter-help x  sequ)
    (cond ~~((= x 0) 0)~~
          ((= x 1) 1)
          ((= x 2) 2)
          ((= x 0) 0)
          ((else (+ (*2 (- x 1)) (- x 3))(+ sequ 1)))
  (iter-help n 0)))
```

(+ (*2 (- x 1)) (x 3))

x-3

-3.5

(b) [2 marks] Use the substitution model for (f 4) to show that the process is iterative.

2

f(n) = 2* f(n-1) + f(n-3)
f(4) = 2* f(4-1) + f(4-3)
f(4) = 2* f(3) + f(1)
f(4) = 2*(2* (f(3-1) + f(3-3) + f(1))
f(4) = 2*(2*(f(2) + f(0)) + f(1))

f(4) = 2*(2*(2 + 0)) + f(1))
f(4) = (2*(2*2) + f(1))
f(4) = (2*4 + 1)
f(4) = 8 + 1
f(4) = 9

(c) [2 marks] What is the value of the following expression
(let ((x 5)) (list '(x x) x '(cons '(x x) x)))

~~list~~ (5 5) 5 'cons(5 5) 5

-2

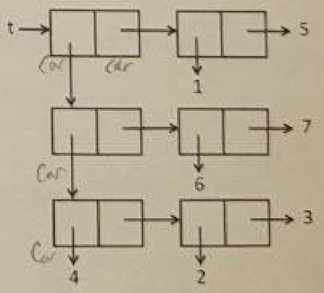
$\to$ $\boxed{(5\ 5\ 5\ ((5 . 5) 5))}$

$\to$ '((x x) 5 (cons '(x x) x))

(d) [2 marks] Write the code that creates the following box and pointer diagram:

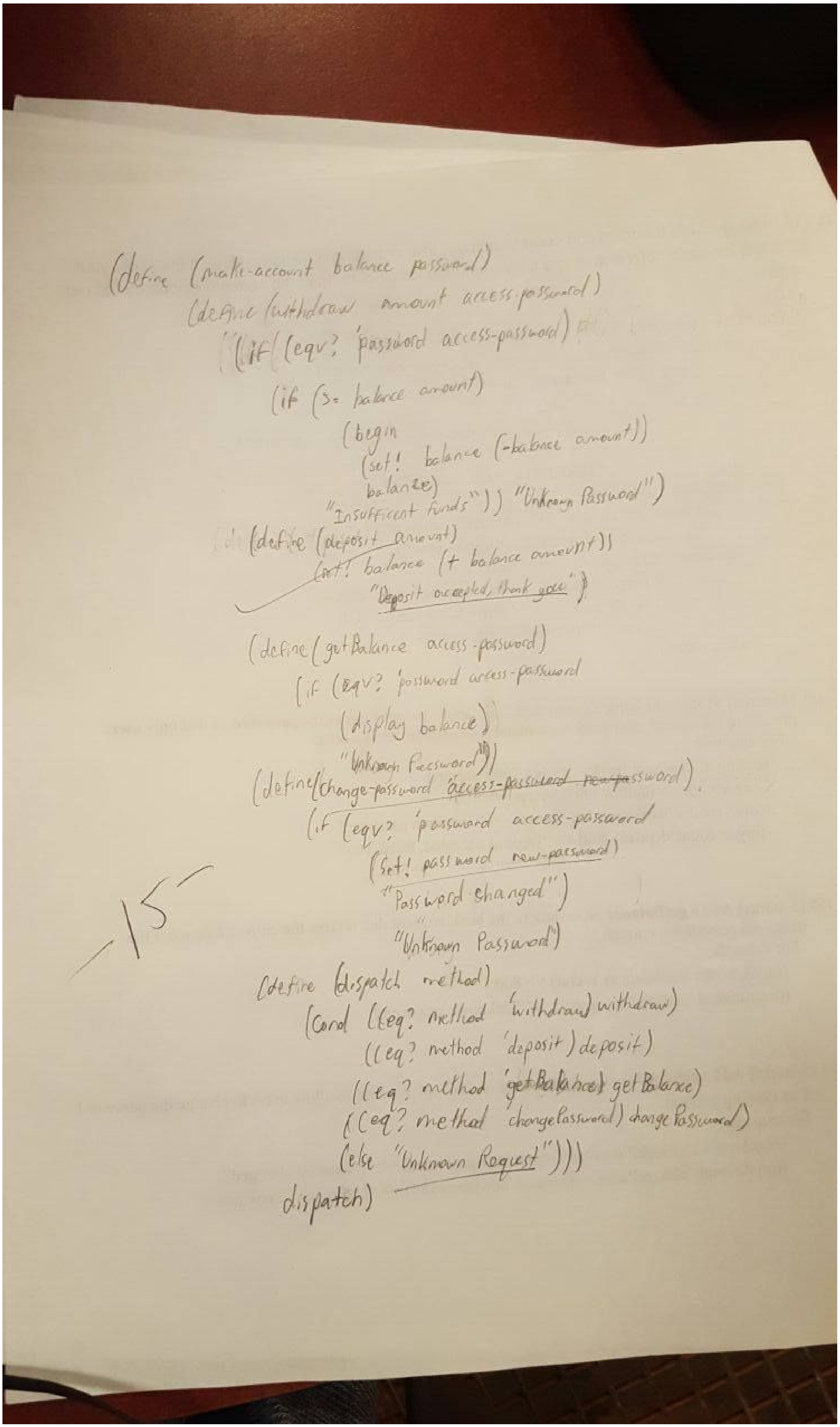
( )t(() (1 5))
↓
(( ) (6 7))
↓
(4 (2 3))

-2



$\boxed{\text{(define t (((4(2 3)) (6 7)) (1 5)))}}$

(4 2 3)



```
(define (make-account balance password)
    (define (withdraw amount access-password)
      ((if ((eqv? 'password access-password)
          (if (>= balance amount)
                (begin
                  (set! balance (-balance amount))
                  balanze)
                "Insufficent funds")) "Unknown Password")
    (define (deposit amount)
        (set! balance (+ balance amount))
              "Deposit accepted, thank you")

    (define(getBalance access-password)
        (if (eqv? 'password access-password
              (display balance)
              "Unknown Password"))
    (define(change-password ~~'access-password new~~password)
        (if (eqv? 'password access-password
              (set! password new-password)
              "Password changed")
              "Unknown Password")
    (define (dispatch method)
        (cond ((eq? method 'withdraw) withdraw)
              ((eq? method 'deposit) deposit)
              ((eq? method 'getBalance) getBalance)
              ((eq? method 'changePassword) changePassword)
              (else "Unknown Request")))
    dispatch)
```

-15

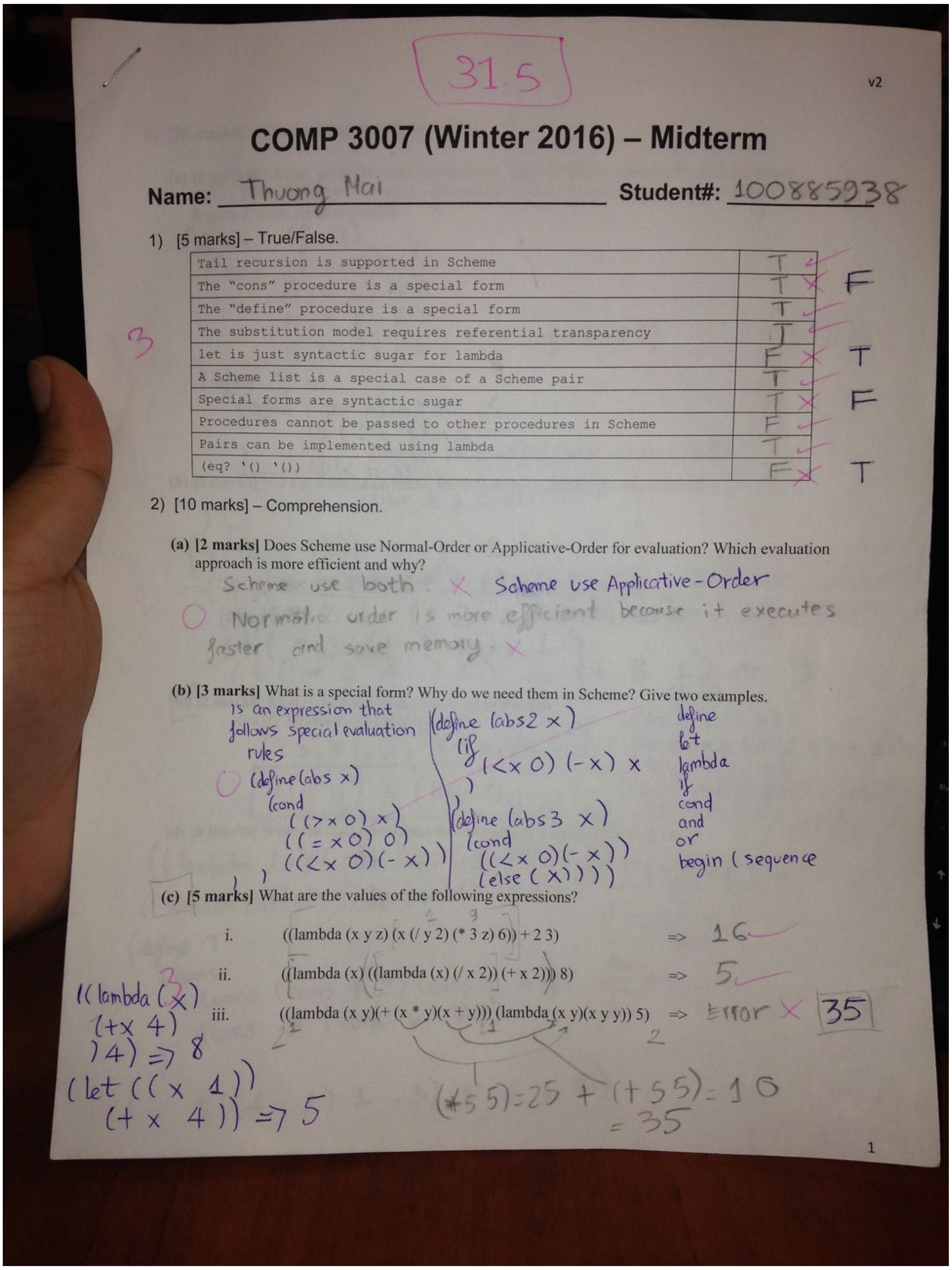
4) [15 marks] – Use the following code to answer the remaining questions. Note, think about your answers for each before writing a single solution for all of them in the space provided on the next page.

```
(define (make-account balance password)
    (define (withdraw amount access-password)
         (if (>= balance amount)
                (begin
                    (set! balance (- balance amount))
                    balance)
                "Insufficient funds"))
    (define (deposit  amount)
        (set! balance (+ balance amount))
        "Deposit accepted, thank you")
    (define (dispatch  method)
      (cond ((eq? method  'withdraw) withdraw)
            ((eq? method  'deposit) deposit)
            (else "Unknown Request")))
    dispatch)
```

(a) [5 marks] Modify the bank account definition above to make use of the password so that only users providing the correct password can withdraw funds from the account.
For example:
(define myAccount (make-account 1000 'secret))
((myAccount 'withdraw) 600 'secret) => 400
((myAccount 'withdraw) 400 'guess) => "Unknown password"
((myAccount 'deposit) 400) => "Deposit accepted, thank you"

(b) [5 marks] Add a **getBalance** procedure to the bank account that returns the current balance if the password provided is correct.
For example:
((myAccount 'getBalance) 'secret) => 800
((myAccount 'getBalance) 'guess) => "Unknown password"

(c) [5 marks] Add a **changePassword** procedure to the bank account to allow users to change the password of an existing account.
For example:
((myAccount 'changePassword) 'secret 'newpassword) => "Password changed"
((myAccount 'changePassword) 'guess 'newpassword) => "Unknown password"



31.5

v2

COMP 3007 (Winter 2016) – Midterm

Name: Thuong Mai **Student#:** 100885938

1) [5 marks] – True/False.

3

Tail recursion is supported in Scheme	T	✓	
The "cons" procedure is a special form	T	✗	F
The "define" procedure is a special form	T	✓	
The substitution model requires referential transparency	T	✓	
let is just syntactic sugar for lambda	F	✗	T
A Scheme list is a special case of a Scheme pair	T	✓	
Special forms are syntactic sugar	T	✗	F
Procedures cannot be passed to other procedures in Scheme	F	✓	
Pairs can be implemented using lambda	T	✓	
(eq? '() '())	F	✗	T

2) [10 marks] – Comprehension.

(a) [2 marks] Does Scheme use Normal-Order or Applicative-Order for evaluation? Which evaluation approach is more efficient and why?

Scheme use both. ✗ Scheme use Applicative-Order

0 Normal order is more efficient because it executes faster and save memory. ✗

(b) [3 marks] What is a special form? Why do we need them in Scheme? Give two examples.

Is an expression that follows special evaluation rules

0

```
(define (abs x)
  (cond
    ((> x 0) x)
    ((= x 0) 0)
    ((< x 0) (- x))
  )
)
```

```
(define (abs2 x)
  (if
    (< x 0) (- x) x
  )
)
(define (abs3 x)
  (cond
    ((< x 0) (- x))
    (else (x))))
```

define
let
lambda
if
cond
and
or
begin (sequence

(c) [5 marks] What are the values of the following expressions?

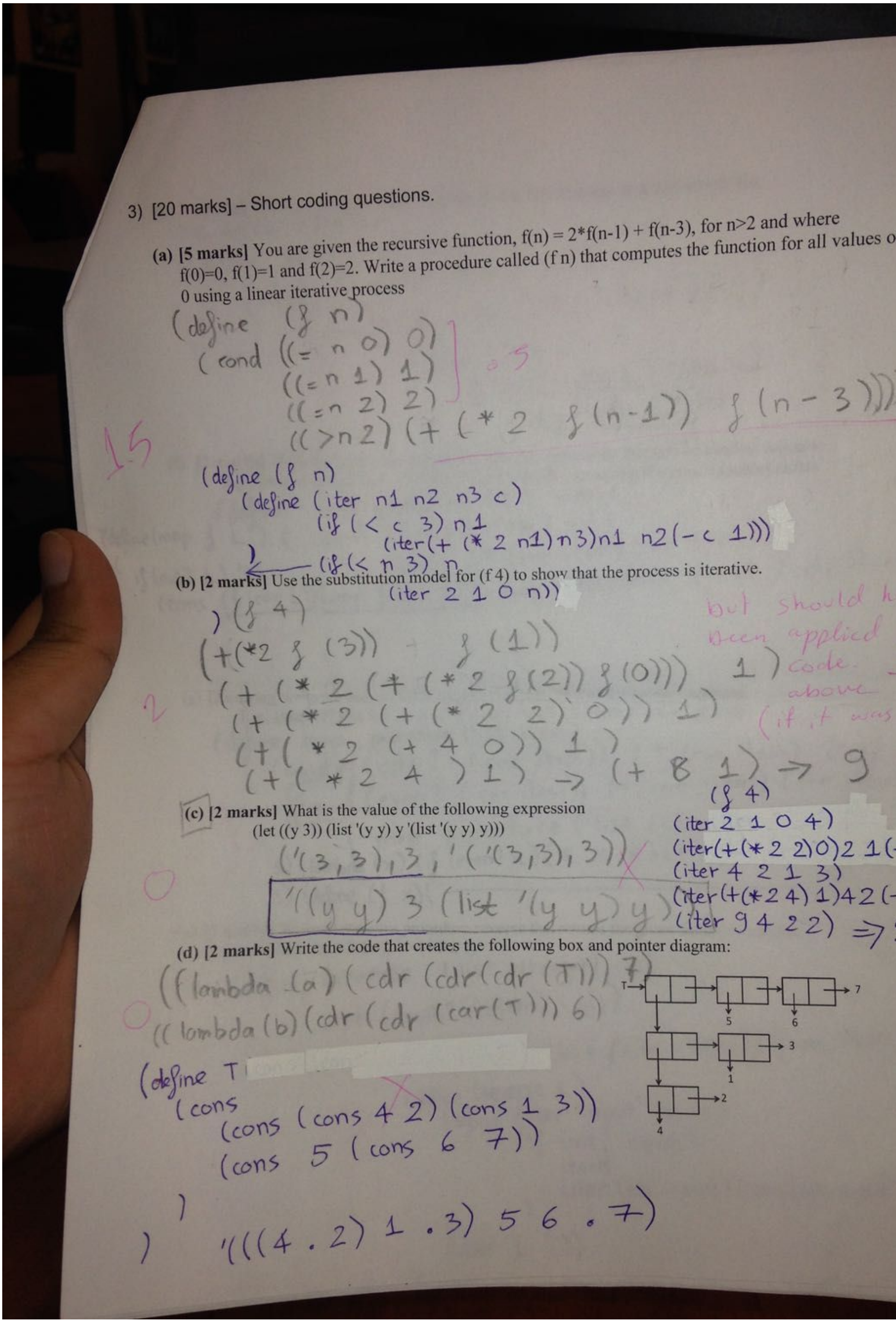
3

i.	((lambda (x y z) (x (/ y 2) (* 3 z) 6)) + 2 3)	=>	16 ✓
ii.	((lambda (x) ((lambda (x) (/ x 2)) (+ x 2))) 8)	=>	5 ✓
iii.	((lambda (x y)(+ (x * y)(x + y))) (lambda (x y)(x y y)) 5)	=>	Error ✗ $\boxed{35}$

(* 5 5) = 25 + (+ 5 5) = 10
= 35

```
((lambda (x)
  (+ x 4)
  ) 4) => 8
(let ((x 1))
  (+ x 4)) => 5
```

1



3) [20 marks] – Short coding questions.

(a) [5 marks] You are given the recursive function, f(n) = 2*f(n-1) + f(n-3), for n>2 and where f(0)=0, f(1)=1 and f(2)=2. Write a procedure called (f n) that computes the function for all values o
0 using a linear iterative process

```
(define (f n)
 (cond ((= n 0) 0)
       ((= n 1) 1)
       ((= n 2) 2)
       ((> n 2) (+ (* 2 f(n-1)) f(n-3)))
```

0.5

15

```
(define (f n)
  (define (iter n1 n2 n3 c)
     (if (< c 3) n1
          (iter(+ (* 2 n1) n3) n1 n2 (- c 1)))
  )
  (if (< n 3) n
      (iter 2 1 0 n))
)
```

(b) [2 marks] Use the substitution model for (f 4) to show that the process is iterative.

```
(f 4)
(+(*2 f (3))        f (1))
(+ (* 2 (+ (* 2 f(2)) f(0)))   1)
(+ (* 2 (+ (* 2 2) 0)) 1)
(+( * 2 (+ 4 0)) 1 )
(+ ( * 2 4 ) 1) -> (+ 8 1) -> 9
```

2

but should h
been applied
code
above
(if it was

```
(f 4)
(iter 2 1 0 4)
(iter(+(* 2 2)0)2 1(-
(iter 4 2 1 3)
(iter(+(*2 4) 1)4 2(-
(iter 9 4 2 2) =>
```

(c) [2 marks] What is the value of the following expression
(let ((y 3)) (list '(y y) y '(list '(y y) y)))

('(3,3), 3, '('(3,3), 3))

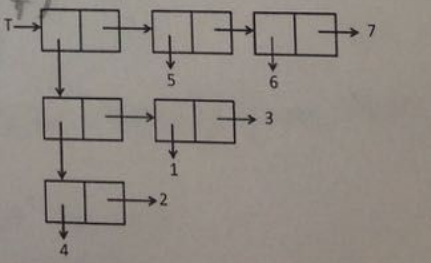
$\boxed{\text{'((y y) 3 (list '(y y) y))}}$

0

(d) [2 marks] Write the code that creates the following box and pointer diagram:

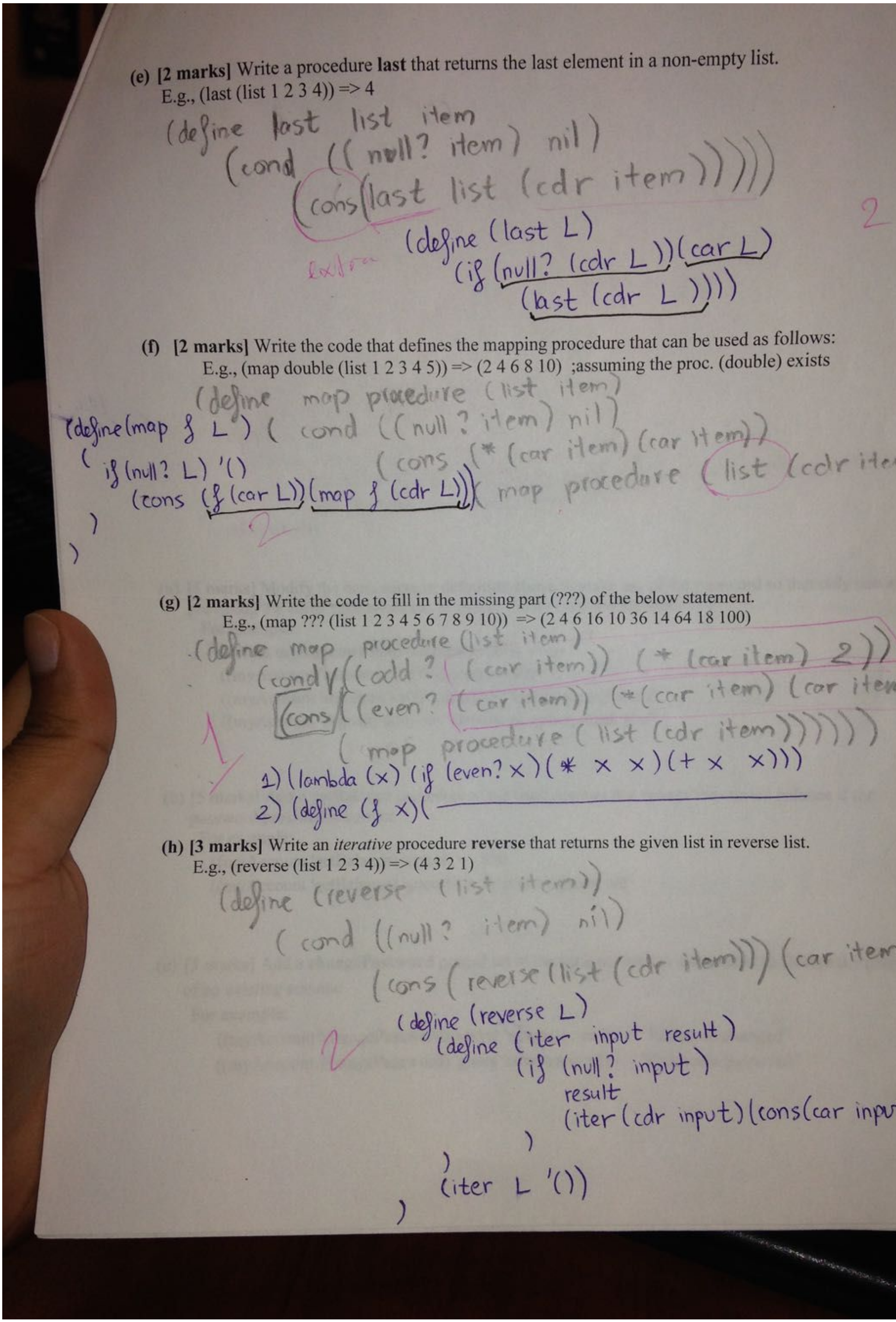
((lambda (a) (cdr (cdr(cdr (T))) 7)
((lambda (b) (cdr (cdr (car(T))) 6)

0



```
(define T
  (cons
    (cons (cons 4 2) (cons 1 3))
    (cons 5 (cons 6 7))
  )
)
```

'(((4 . 2) 1 . 3) 5 6 . 7)



(e) [2 marks] Write a procedure **last** that returns the last element in a non-empty list.
E.g., (last (list 1 2 3 4)) => 4

```
(define last list item
   (cond ((null? item) nil)
      (cons (last list (cdr item)))))
```

2

extra

```
(define (last L)
   (if (null? (cdr L)) (car L)
      (last (cdr L))))
```

(f) [2 marks] Write the code that defines the mapping procedure that can be used as follows:
E.g., (map double (list 1 2 3 4 5)) => (2 4 6 8 10) ;assuming the proc. (double) exists

```
(define map procedure (list item)
   (cond ((null ? item) nil)
      (cons (* (car item) (car item))
         (map procedure (list (cdr ite
```

```
(define (map f L)
 (
  if (null? L) '()
   (cons (f (car L)) (map f (cdr L)))
 )
)
```

2

(g) [2 marks] Write the code to fill in the missing part (???) of the below statement.
E.g., (map ??? (list 1 2 3 4 5 6 7 8 9 10)) => (2 4 6 16 10 36 14 64 18 100)

```
(define map procedure (list item)
   (cond ((odd ? ((car item)) (* (car item) 2))
      (cons ((even? ((car item)) (*(car item) (car iten
         (map procedure (list (cdr item))))))
```

1

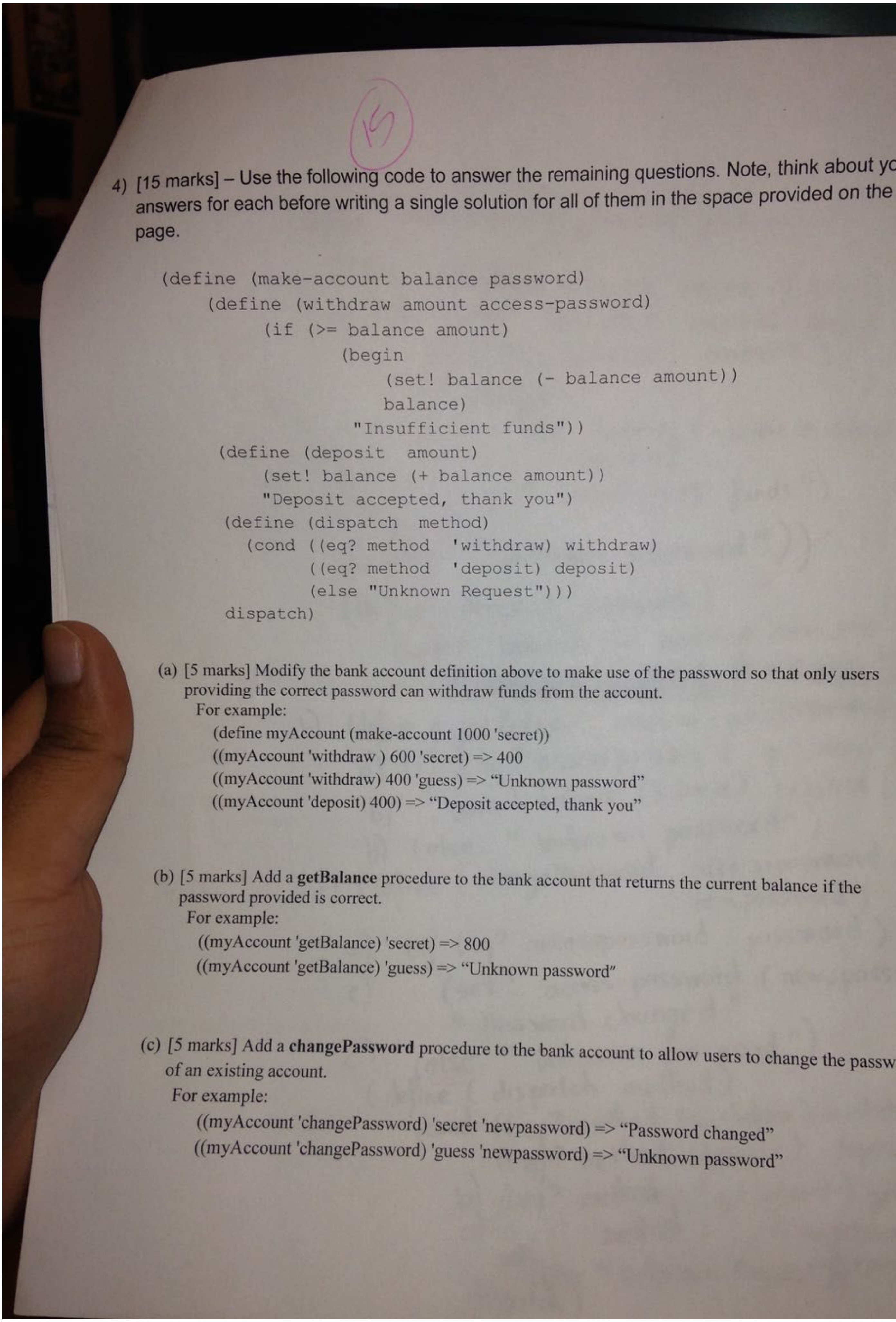
```
1) (lambda (x) (if (even? x) (* x x) (+ x x)))
2) (define (f x) (
```

(h) [3 marks] Write an *iterative* procedure **reverse** that returns the given list in reverse list.
E.g., (reverse (list 1 2 3 4)) => (4 3 2 1)

```
(define (reverse (list item))
   ( cond ((null ? item) nil)
      (cons (reverse (list (cdr item))) (car item
```

2

```
(define (reverse L)
   (define (iter input result)
      (if (null? input)
         result
         (iter (cdr input) (cons(car inpu
      )
   )
   (iter L '())
)
```



15

4) [15 marks] – Use the following code to answer the remaining questions. Note, think about yo
answers for each before writing a single solution for all of them in the space provided on the
page.

```
(define (make-account balance password)
    (define (withdraw amount access-password)
         (if (>= balance amount)
                (begin
                    (set! balance (- balance amount))
                    balance)
                 "Insufficient funds"))
     (define (deposit  amount)
         (set! balance (+ balance amount))
         "Deposit accepted, thank you")
     (define (dispatch  method)
       (cond ((eq? method  'withdraw) withdraw)
             ((eq? method  'deposit) deposit)
             (else "Unknown Request")))
     dispatch)
```

(a) [5 marks] Modify the bank account definition above to make use of the password so that only users providing the correct password can withdraw funds from the account.
For example:
(define myAccount (make-account 1000 'secret))
((myAccount 'withdraw) 600 'secret) => 400
((myAccount 'withdraw) 400 'guess) => "Unknown password"
((myAccount 'deposit) 400) => "Deposit accepted, thank you"

(b) [5 marks] Add a **getBalance** procedure to the bank account that returns the current balance if the password provided is correct.
For example:
((myAccount 'getBalance) 'secret) => 800
((myAccount 'getBalance) 'guess) => "Unknown password"

(c) [5 marks] Add a **changePassword** procedure to the bank account to allow users to change the passw
of an existing account.
For example:
((myAccount 'changePassword) 'secret 'newpassword) => "Password changed"
((myAccount 'changePassword) 'guess 'newpassword) => "Unknown password"

```
(define (make-account balance password)

   (define (withdraw amount access-password)
  a) (if (eq? password access-password)
         (if (>= balance amount)
            (begin
              (set! balance (-balance amount))
                 balance)
              "Insufficient funds")

        )cond
  a) (else "Unknown password"))
   (define (deposit amount)
      (set! balance (+ balance amount))
           "Deposit accepted, thank you")
b) (define (getBalance access-password)
   b) (if (eq? accesspassword password)
   b)    (set! balance (balance) balance))
   b) (else "Unknown password")
c) (define (changePassword access-password
                            new-password)
   c) (if (eq? accesspassword password)
   c)     (set! access-password (new-passwo
        "Password changed"
   c)  (else "Unknown password")
   (define (dispatch method)
      (cond ((eq? method 'withdraw) withdraw
            ((eq? method 'deposit) deposit
        b) ((eq? method 'getBalance) getBa
        c) ((eq? method 'changePassword
            (else "Unknown Request")))changePa
dispatch)
```

-end-