

SOEN 331: Introduction to Formal Methods for  
Software Engineering

**Assignment 1**

Propositional and Predicate Logic, Structures,  
Binary Relations, Functions, Orderings and  
Construction Techniques

Anh Thien Nguyen - 40122030

Paul Touma- 40210678

*Gina Cody School of Computer Science and Software Engineering  
Concordia University, Montreal, QC, Canada*

Winter 2024

# Contents

<b>1</b>	<b>Problem 1 (5 pts)</b>	<b>3</b>
1.1	Answer: . . . . .	3
<b>2</b>	<b>Problem 2 (14 pts)</b>	<b>4</b>
2.1	Part 1 . . . . .	4
2.2	Part 2: Categorical propositions . . . . .	6
2.3	Part 3: Categorical propositions . . . . .	7
<b>3</b>	<b>Problem 3 (22 pts)</b>	<b>8</b>
3.1	Part 1 . . . . .	8
3.2	Part 2 . . . . .	9
<b>4</b>	<b>Problem 4 (14 pts)</b>	<b>11</b>
<b>5</b>	<b>Problem 5 (15 pts)</b>	<b>13</b>
<b>6</b>	<b>Problem 6 (20 pts)</b>	<b>16</b>
6.1	Part 1: . . . . .	16
6.2	Part 2: . . . . .	17
6.3	Part 3: . . . . .	19
<b>7</b>	<b>Problem 7 (10 pts)</b>	<b>20</b>

# 1 Problem 1 (5 pts)

## 1.1 Answer:

Assume that  $p$  represents the statement "The face of a card is **blue**" and  $q$  represent "It has a **prime** on the other side". The proposition could be expressed as  $p \rightarrow q$

We must turn over the card shows the color **blue**, and expect to see prime. Because this card shows the color blue, to validate the truth of the proposition, this card must be turned over and check if its back has a prime number. In other words, the statement  $p$  is true. In order to validate the proposition  $p \rightarrow q$ ,  $q$  has to be true. Therefore the turned card's back is expected to have prime number.

Taking the contrapositive of the implication statement,  $\neg q \rightarrow \neg p$  ( $\neg prime \rightarrow \neg blue$ ). We must turn over the card shows number **9** - not a prime number, and expect to not see blue color in the back, in this case, we expect to see yellow.

We cannot make any claim based on the card shows number **11**. In this case, the statement  $q$  is true, the card has prime number). The proposition  $p \rightarrow q$  will be true regardless the truth value of  $p$  - the another side of the card could be blue or yellow.

We cannot make any claim based on the card shows **yellow**. Because the statement  $p$  is false in this case ( $\neg blue$ ), as this card has yellow color on one side. Therefore, no matter the back of the card is prime or not prime, this will not affect the truth of the proposition.

In conclusion, the cards with visible sides blue and 9 must be turned over.

## 2 Problem 2 (14 pts)

### 2.1 Part 1

1. Construct a formula in predicate logic to define a planet, where planet is defined as an object whose mass is greater than  $0.33 \times 10^{24}KG$ , and which it orbits around the sun. For all practical purposes, you may ignore the  $10^{24}KG$  factor.

Formula for Planet:

$$Planet(object) \equiv \exists mass (Mass(object, mass) \wedge mass > 0.33) \wedge Orbits(object, sun)$$

Formula for *is\_satellite\_of*:

$$is\_satellite\_of \equiv Orbits(satellite, object) \wedge Planet(object)$$

2. Map your formulas to Prolog rules *is\_planet*/1, and *is\_satellite\_of*/2, and demonstrate how it works by executing both ground- and non-ground queries. Identify the type of each query.

Here we have the 2 main rules *is\_planet* and *is\_satellite\_of*.

```
is_planet(P) :-  
    object(P),  
    mass(P, Mass),  
    Mass >= 0.33,  
    orbits(P, sun).
```

```
is_satellite_of(S, P) :-  
    object(S),  
    is_planet(P),  
    orbits(S, P).
```

Below you can see the interaction with the Prolog interpreter(to be able to see the multiple results, you need to press *space*).

```

pault@DESKTOP-QG7B519 MINGW64 ~/desktop
$ swipl solar.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

1 ?- is_planet(pluto).
false.

2 ?- is_planet(mars).
true.

3 ?- is_planet(P).
P = mercury ;
P = venus ;
P = earth ;
P = mars ;
P = jupiter ;
P = saturn ;
P = uranus ;
P = neptune ;
false.

4 ?- is_satellite_of(moon, Planet).
Planet = earth ;
false.

5 ?- is_satellite_of(S, mars).
S = deimos ;
S = phobos ;
false.

6 ?- halt.

```

Figure 1: Prolog interaction for problem 2 part 2

3. Construct a Prolog rule obtain all satellites/2 that succeeds by returning a collection of all satellites of a given planet.

Here we have the 2 main rules *is\_planet* and *is\_satellite\_of*.

```

is_planet(P) :-
    object(P),
    mass(P, Mass),
    Mass >= 0.33,
    orbits(P, sun).

```

```

is_satellite_of(S, P) :-
    object(S),
    is_planet(P),
    orbits(S, P).

```

```

obtain_all_satellites(Planet, Satellites) :-
    findall(Satellite, is_satellite_of(Satellite, Planet),
    Satellites).

```

```

pault@DESKTOP-Q67B5I9 MINGW64 ~/desktop
$ swipl solar.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

1 ?- obtain_all_satellites(earth, L).
L = [moon].

2 ?- obtain_all_satellites(mars, L).
L = [deimos, phobos].

3 ?- obtain_all_satellites(venus, L).
L = [].

4 ?- obtain_all_satellites(saturn, L).
L = [atlas, calypso, helene].

5 ?- halt.

```

Figure 2: Prolog interaction for problem 2 part 3

## 2.2 Part 2: Categorical propositions

1. "Some numbers are not composite"

Answer:

This is:  $\exists x(\text{number}(x) \wedge \neg \text{composite}(x))$ . Type O.

2. "No numbers are prime"

Answer:

Rephrased as: "All numbers are composite".  $\forall x(\text{number}(x) \wedge \text{composite}(x))$ . Type A.

3. "Some numbers are not prime"

Answer:

Rephrased as: "Some numbers are composite".  $\exists x(\text{number}(x) \wedge \text{composite}(x))$ . Type I.

4. "All numbers are prime"

Answer:

Rephrased as: "No numbers are composite".  $\forall x(\text{number}(x) \wedge \neg \text{composite}(x))$ . Type E.

## 2.3 Part 3: Categorical propositions

1. Prove formally that negating A is logically equivalent to obtaining O (and vice versa).

Let A be the statement  $\forall s : S \mid (s \in P)$ .

The negation of A,  $\neg A$  is  $\neg(\forall s : S \mid (s \in P))$ .

By De Morgan's,  $\neg A$  is equivalent to  $\exists s : S \mid (\neg (s \in P))$ .

Therefore, negating A yields O, since  $\exists s : S \mid (\neg (s \in P))$  is the definition of an O proposition.

2. Prove formally that negating E is logically equivalent to obtaining I (and vice versa).

Let E be the statement  $\neg(\exists s : S (s \in P))$ .

The negation of E,  $\neg E$  is  $\neg(\neg(\exists s : S (s \in P)))$ .

By the law of double negation,  $\neg E$  is equivalent to  $\exists s : S (s \in P)$ .

Therefore, negating E yields I, since  $\exists s : S (s \in P)$  is the definition of an I proposition.

### 3 Problem 3 (22 pts)

#### 3.1 Part 1

1. Visualize all models of behavior.

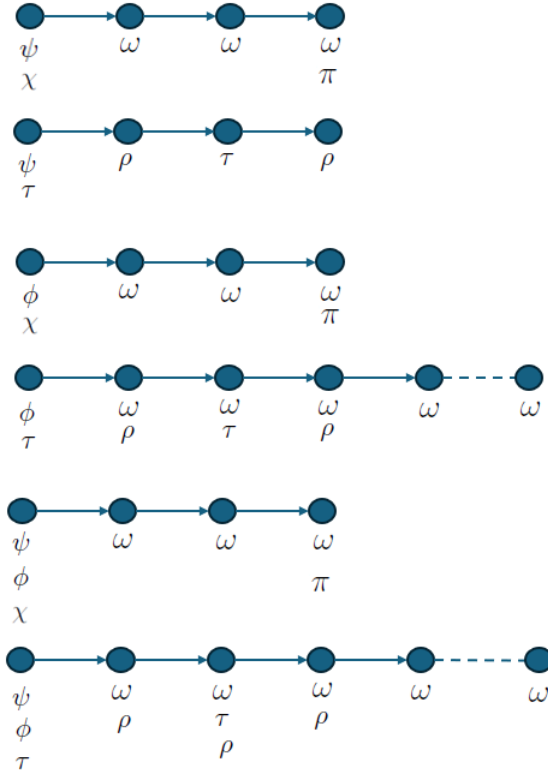


Figure 3: Models of behavior for Problem 3.1.1

2. Make observations on the visualization by specifying exact conditions about termination, non-termination and consistency (or lack thereof), if any exist.

**Termination:** As you can see with the image above, we have some models that will terminate. For example, the first one will terminate because as soon as  $\pi$  is true, the other condition  $(\pi \mathcal{R} \omega)$  will stop.

**Non-termination:** The examples 4 and 6 represent non-termination models since  $\omega$  will stay true until and including the state at which  $\pi$  first becomes true.

**Consistency:** They are not paths that are consistent with the temporal formula. The closest one is the last one that adheres to 6 out of the 7 of the temporal constraints.



### 3.2 Part 2

1. Formalize and visualize the following requirement: “If none of  $\phi$  or  $\psi$  are invariants, then starting from time  $i + 2$ ,  $\chi$  will eventually become true and it will remain true up to and including the moment  $\tau$  first becomes true. Note that there exists no guarantee that  $\tau$  ever becomes true.”

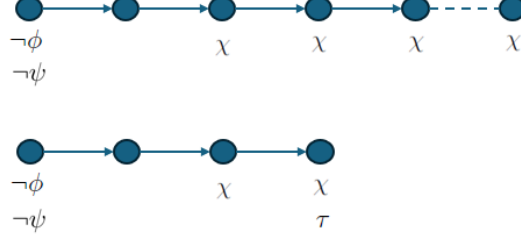


Figure 4: Models of behavior for Problem 3.2.1

This requirement can be formalize as follows:

$$(\neg\phi \vee \neg\psi) \rightarrow \bigcirc^2(\Box\chi \wedge (\chi\mathcal{U}\tau))$$

2. Describe the visualize the following requirement:

$$(\neg\alpha \vee \neg\beta) \rightarrow (\bigcirc \diamond (\gamma\mathcal{U}\delta))$$

$(\neg\alpha \wedge \neg\beta)$  signifies that either  $\alpha$  is not true, or  $\beta$  is not true, or both are not true.  $(\bigcirc \diamond (\gamma\mathcal{U}\delta))$  means that there will eventually be a time from which point forward ( $\Box$ ) it will always be the case that  $\gamma$  holds until  $\delta$  becomes true. Note that in this case,  $\delta$  is guaranteed to become true at some future time. This is the definition of the  $\mathcal{U}$  operation.

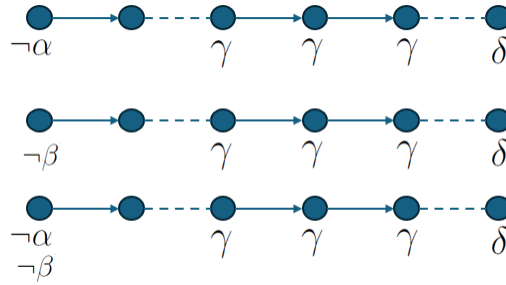


Figure 5: Models of behavior for Problem 3.2.2

3. Describe the visualize the following requirement:

$$(\bigcirc\tau \wedge \bigcirc\Box\diamond\chi) \rightarrow \bigcirc^2(\phi\mathcal{W}\psi)$$

This requirement signifies that at iteration  $i + 1$   $\tau$  will become true and eventually  $\chi$  will always be true (not necessarily at  $i + 1$ ). At  $i + 2$   $\psi$  will be true until  $\phi$  becomes true.

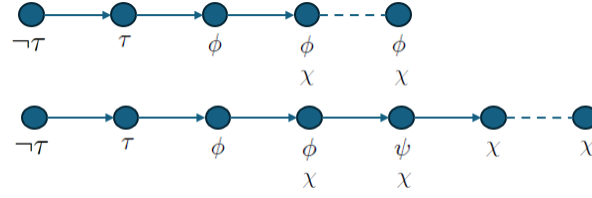


Figure 6: Models of behavior for Problem 3.2.3

## 4 Problem 4 (14 pts)

- (a) Provide a description (in plain English) on what is meant by *P Languages* :

*P Languages* is the power set of *Languages* set, which contains all subsets of the given set. In this case, it contains all subsets of *Languages* set.

- (b) Describe in detail (a) what the expression *Favourites: P Languages* signifies and (b) how it should be interpreted. (c) What could be any legitimate value for variable *Favorites*?

This expression is a **variable declaration**, in which the LHS specifies a variable and the RHS defines a type. In this case, this expression can be expressed as "The variable *Favourites* can assume any value supported by the power set of *Languages*. It means any subset that can be produced from set *Languages* can be a legitimate value of *Favourites*.

- (c) What does the expression *Favourites = P Languages* signify and (b) Is semantically equivalent to *Favourites: P Languages*

This expression is an **assignment statement**, it assigns the value of the RHS (which is a set) to the variable on the LHS. In this case, variable *Favourites* is assigned to the power set of *Languages*. Meanwhile, in the variable declaration, a variable is associated to a type. Therefore, the two expressions are *not semantically equivalent*.

- (d) Is  $\{Lua, Groovy, C\} \in P Languages$ ? Explain in detail.

No, because  $\{Lua, Groovy, C\}$  is not an element of the power set *P Languages*.

- (e) Is  $\{\{Lua, Groovy, C\}\} \subset P Languages$ ? Explain in detail.

Yes, because  $\{\{Lua, Groovy, C\}\}$  is a set of sets.

- (f) What is the difference between the following two variable declarations (*Languages*, *P Languages*) and are the two variables atomic and non-atomic?

*Languages* is a variable, it is a set contains atomic variables and another set. Therefore, it is non-atomic.

*P Languages* is the power set of *Languages* set, it contains all of the subsets of the given set. Therefore, it is also non-atomic.

- (g) In the expression  $Library = \{C, Ruby, Go\}$ , where the variable is used to hold any collection that can be constructed from *Languages*, what is the type of the variable?

The variable *Library* is a **composite**. It is a set.

(h) Is  $\{\emptyset\} \in P \text{ Languages}$ ? Explain.

No, There is no set that contains the empty set in the power set of *Languages*.

(i) **Programming** using Commom LISP

Common Lisp contains some build-in functions such as:

**member item list:** returns true if item is an element of list. Otherwise, it returns false.

**equal obj1 obj2:** returns true if two objects are equal. Otherwise, it returns false. Using the given build-in functions, the implementation using Common Lisp is as follows:

```
(defun is-memberp (set1 set2)
  (if (null set1)
      t
      (if (member (car set1) set2)
          (is-memberp (cdr set1) set2)
          nil)
      ))

(defun equal-setsp (set1 set2)
  (and (is-memberp set1 set2) (is-memberp set2 set1)))
```

Example runs are as follows:

```
> (is-memberp '() '(a, b, c, d))
T
> (is-memberp '(a, c) '(a, b, c, d))
T
> (is-memberp '(a, e) '(a, b, c, d))
NIL
> (is-memberp '(a, d) '(a, b, c, d))
T

> (equal-setsp '() '(a, b, c, d))
NIL
> (equal-setsp '(a, b, c) '(a, c, b))
T
> (equal-setsp '(a, b) '(a, b, c, d))
NIL
> (equal-setsp '() '())
T
```

## 5 Problem 5 (15 pts)

(a) Define operations  $Enqueue(Q, T)$  and  $Dequeue(Q)$ .

- $Enqueue(Q, T)$  is an operation that is used to insert an element to the rear end of the queue. In this case, the queue ADT is defined using two stack  $\Sigma_1$  and  $\Sigma_2$ . In order to implement the enqueue operation, it can use the  $push()$  operation of the Stack ADT to complete. In addition, the Stack ADT is implemented using List  $\Lambda$ , so the  $push()$  operation could be implemented using the  $cons()$  operation of the List  $\Lambda$ , it adds an element at the top of the Stack. Therefore, using  $cons()$  operation of the List to implement  $push()$  operation of Stack, and then implement  $Enqueue()$  operation of Queue ADT. This allows to insert an element at the rear end of the queue (enqueue).

$push(\Lambda, T)$  is  
 $cons(T, \Lambda)$

$enqueue(Q, T)$  is  
 $push(Q, T)$

- $Dequeue(Q)$  is an operation of Queue ADT that removes an element from the front end of the queue. Because the Queue ADT is implemented using two Stacks  $\Sigma_1$  and  $\Sigma_2$ , the elements will be moved from a stack to another stack, e.g from  $\Sigma_1$  to  $\Sigma_2$ , by using  $push()$  operation to gradually add to the top of  $\Sigma_2$  the element of the top of  $\Sigma_1$ . Eventually, the top of  $\Sigma_2$  is the front end of the Queue ADT. Then this front end can be removed using  $pop()$  operation of Stack, in which this  $pop()$  operation is implemented using  $head()$  and  $tail()$  operations ( $car, cdr$ ).

$pop(stack)$  is  
if not( $isEmpty(\Lambda)$ )  
     $el = g()$   
     $\Lambda' = f(\Lambda)$   
return  $el$

$dequeue(Q)$   
if ( $isEmpty(\Sigma_1) \ \&\& \ isEmpty(\Sigma_2)$ )  
    nil  
if ( $isEmpty(\Sigma_2)$ )  
    while ( $not(isEmpty(\Sigma_1))$ )  
         $push(\Sigma_2, pop(\Sigma_1))$   
return  $pop((\Sigma_2))$

- (b) Use Common LISP to define two stacks (`stack1` and `stack2`) as global variables to hold the collection, and implement functions `enqueue` and `dequeue`. Place your code in file `queue-adt.lisp`, and include your interaction with the language environment to demonstrate the behavior of your code.

The Code for the implementations of Enqueue and Dequeue:

```
(defun push (stack element)
  (cons element stack))

(defun pop (stack)
  (if (null stack)
      (error "Stack is empty")
      (values (car stack) (cdr stack))))

(defun stack-empty? (stack)
  (null stack))

(defun enqueue (queue element)
  (setf (symbol-value 's1) (push (symbol-value 's1) element)))

(defun dequeue (queue)
  (if (stack-empty? (symbol-value 's2))
      (progn
        (while (not (stack-empty? (symbol-value 's1)))
          (multiple-value-bind (element remaining-stack)
            (pop (symbol-value 's1))
            (setf (symbol-value 's2)
                  (push (symbol-value 's2) element)))))
      (if (stack-empty? (symbol-value 's2))
          (error "Queue is empty")))
    nil)
  (multiple-value-bind (element remaining-stack)
    (pop (symbol-value 's2))
    (setf (symbol-value 's2) remaining-stack)
    element))
```

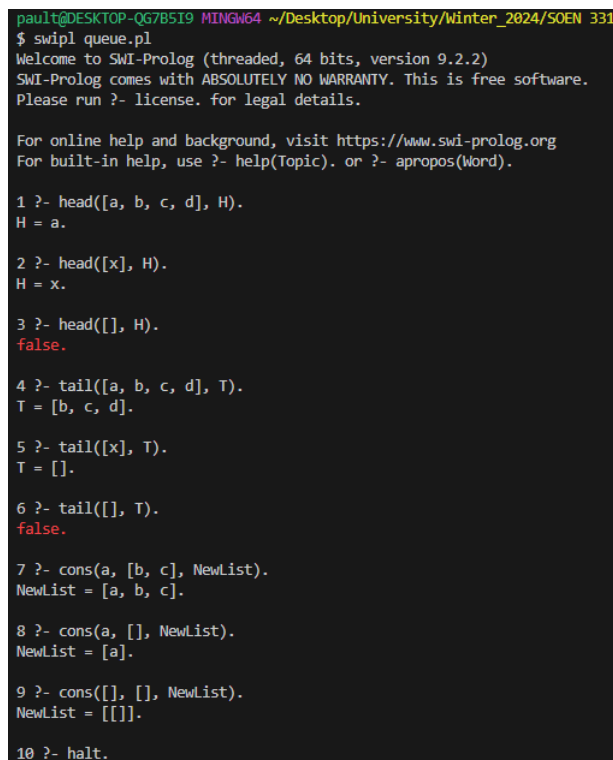
The interaction with Queue ADT:

```
(setq s1 nil) ;Initialize  $\Sigma_1$  stack  
(setq s2 nil) ;Initialize  $\Sigma_2$  Stack
```

```
(enqueue 'q 'a)  
(enqueue 'q 'b)  
(enqueue 'q 'c)
```

```
(print (dequeue 'q))  
>A  
(print (dequeue 'q))  
>B  
(print (dequeue 'q))  
>C
```

- (c) Implement operations *head*, *tail* and *cons* in Prolog and demonstrate their usage (include your interaction with the language environment in your submission).



```
pault@DESKTOP-QG7B5I9 MINGW64 ~/Desktop/University/Winter_2024/SOEN_331  
$ swipl queue.pl  
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.2)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit https://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
1 ?- head([a, b, c, d], H).  
H = a.  
  
2 ?- head([x], H).  
H = x.  
  
3 ?- head([], H).  
false.  
  
4 ?- tail([a, b, c, d], T).  
T = [b, c, d].  
  
5 ?- tail([x], T).  
T = [].  
  
6 ?- tail([], T).  
false.  
  
7 ?- cons(a, [b, c], NewList).  
NewList = [a, b, c].  
  
8 ?- cons(a, [], NewList).  
NewList = [a].  
  
9 ?- cons([], [], NewList).  
NewList = [[]].  
  
10 ?- halt.
```

Figure 7: Prolog interaction for problem 5 part 3

## 6 Problem 6 (20 pts)

### 6.1 Part 1:

- (a) In order to prove that the binary relation  $R$ : "*is of type*" in the domain of types in the Java API is a *partial order*, it is necessary to prove that binary relation  $R$  satisfies three properties: reflexivity, antisymmetry, and transitivity. Consider set  $A$  is the domain of types in the Java API.

- **Reflexivity:** When  $\forall a \in A: aRa$ . In this case, for any type  $a$  in  $A$ ,  $a$  is type of  $a$ . This is true since every type is inherently of its own type.
- **Antisymmetry:** When  $\forall a, b \in A: (aRb \wedge bRa) \rightarrow a = b$ . If  $a$  is of type  $b$  and  $b$  is of type  $a$ , then  $a$  must be equal to  $b$ . This property is satisfied as in Java, two types are of type to each other only if they are equal.
- **Transitivity:** When  $\forall a, b, c \in A: (aRb \wedge bRc) \rightarrow aRc$ . If type  $a$  is type of  $b$  and type  $b$  is type of  $c$ , then type  $a$  is type of  $c$ . This property is true because if a class or interface or extends another type, and that type implements or extends another type (is type of another type), then the first type is also of the last type.

Therefore, the binary relation  $R$  is a partial order as it satisfies three properties: reflexivity, antisymmetry, and transitivity.

- (b) To prove  $(V_1, R)$  is a poset, it is necessary to prove that the binary relation  $R$  is a partial order over  $V_1$ . It means that the relation must satisfy three properties: reflexivity, antisymmetry, transitivity.

- **Reflexivity:** When  $\forall a \in V_1: aRa$ . In this case, for any type  $a$  in  $V_1$ ,  $a$  is type of  $a$ . This is true since every type is of its own type. Which means that the provided edges have self-loops.
- **Antisymmetry:** When  $\forall a, b \in V_1: (aRb \wedge bRa) \rightarrow a = b$ . If  $a$  is of type  $b$  and  $b$  is of type  $a$ , then  $a$  must be equal to  $b$ . From the provided edges, it is observed that there are no two different vertices  $a, b$  such that  $aRb$  and  $bRa$  hold. E.g: There are no  $(HashMap, AbstractMap)$  and  $(AbstractMap, HashMap)$  hold. Therefore, antisymmetry is satisfied.
- **Transitivity:** When  $\forall a, b, c \in V_1: (aRb \wedge bRc) \rightarrow aRc$ . If type  $a$  is type of  $b$  and type  $b$  is type of  $c$ , then type  $a$  is type of  $c$ . According to the provided edges, there are no vertices  $a, b, c$  such that  $aRb$  and  $bRc$ . E.g: There are no sets of edges such that  $(LinkedHashMap, HashMap)$ ,  $(HashMap, AbstractMap)$ , and  $(LinkedHashMap, AbstractMap)$ . Therefore, the relation is transitive.

Therefore,  $(V_1, R)$  is a poset as it satisfies three properties: reflexivity, antisymmetry, and transitivity.



(c) Hasse Diagram

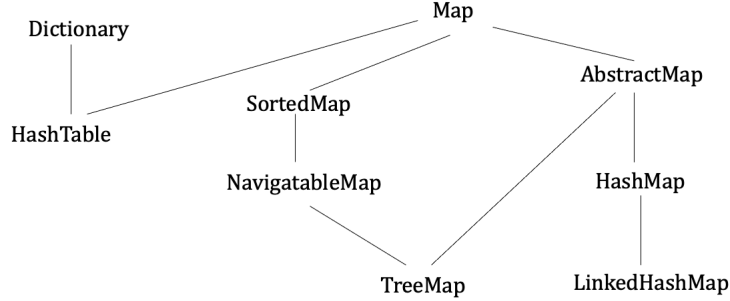


Figure 8: Hasse Diagram for Problem 6.1.3

From Figure 8, there is three minimal elements (HashTable, TreeMap, and LinkedHashMap), and two maximal elements (Dictionary, Map).

## 6.2 Part 2:

(a) To prove that  $\subseteq$  (is subset of) is a partial order, it is needed to indicate that  $\subseteq$  satisfies three properties: reflexivity, antisymmetry, and transitivity. Consider  $A$  is the domain of sets.

- **Reflexivity:** When  $\forall a \in A: a \subseteq a$ . This property is satisfied since every set is a subset of itself.
- **Antisymmetry:** When  $\forall a, b \in A: (a \subseteq b \wedge b \subseteq a) \rightarrow a = b$ . This property is true because if two sets are subsets of each other, they must be equal.
- **Transitivity:** When  $\forall a, b, c \in A: (a \subseteq b \wedge b \subseteq c) \rightarrow a \subseteq c$ . This property holds its validity, because if  $a$  is a subset of  $b$  and  $b$  is a subset of  $c$ ,  $a$  is a subset of  $c$ .

Therefore,  $\subseteq$  is a partial order as it satisfies three properties: reflexivity, antisymmetry, transitivity.

(b)  $P(V_2)$  is the power set of  $V_2 = \{a, b, c\}$ . This is the set containing all subsets of the set  $V_2$ . It is define as follows:

$$P\{a, b, c\} = \{ \emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\} \}$$

$$\begin{aligned} &\{a, b\}, \{a, c\}, \{b, c\}, \\ &\{a, b, c\} \\ &\} \end{aligned}$$

To prove that  $(P(V_2), \subseteq)$  is a poset, it is required to prove that the binary relation  $\subseteq$  is a partial order over the power set  $P$ . It means that the binary relation  $\subseteq$  must satisfy three properties: reflexivity, antisymmetry, transitivity.

- **Reflexivity:** When  $\forall a \in P: a \subseteq a$ . This property is satisfied because for every subset in  $P$ , it is a subset of itself.
- **Antisymmetry:** When  $\forall a, b \in P: (a \subseteq b \wedge b \subseteq a) \rightarrow a = b$ . This property is true because if two sets are subsets of each other, they must be equal. In this case, it is observed that there are not two subsets such as  $\{a, b\}$ ,  $\{b, a\}$  in  $P$ . Therefore, this property is true.
- **Transitivity:** When  $\forall a, b, c \in P: (a \subseteq b \wedge b \subseteq c) \rightarrow a \subseteq c$ . This property is true, because if  $a$  is a subset of  $b$  and  $b$  is a subset of  $c$ ,  $a$  is a subset of  $c$ . E.g:  $\{a\}$  is a subset of  $\{a, b\}$ , and  $\{a, b\}$  is a subset of  $\{a, b, c\}$ , then  $\{a\}$  is a subset of  $\{a, b, c\}$ .  $\{a\}$ ,  $\{a, b\}$ ,  $\{a, b, c\}$  are subsets of  $P$ .

Therefore,  $(P(V_2), \subseteq)$  is a poset as it satisfies three properties: reflexivity, antisymmetry, transitivity.

(c) Hasse Diagram

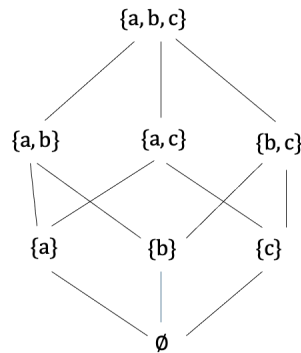


Figure 9: Hasse Diagram for Problem 6.2.3

From Figure 9 There is one minimal element,  $\emptyset$  and one maximal element,  $\{a, b, c\}$ .

### 6.3 Part 3:

**Mapping:** Mapping map is a *partial function* as it is a function defined for some subset of  $V_1$ . It does not force the mapping of every element of  $V_1$  to an element of  $P(V_2)$

**Injectivity:** The function is not injective because it does not satisfy the condition of injectivity such that  $\forall a, b(a \neq b \rightarrow f(a) \neq f(b))$ . In this case, Dictionary and AbstractMap are mapped to the same element  $(\{a, b, c\})$ .

**Surjectivity:** The function is surjective as it is the case that  $\forall b \exists a(f(a) = b)$ . It can be observed that for each element of the codomain, it is mapped with at least one element in the domain.

**Bijection:** By definition, the function is not bijective as it is not the case that both injectivity and surjectivity are satisfied.

**Order preserving:** This function is not order-preserving as it is not the case that  $x \prec y$  in domain of map, implies  $f(x) \prec f(y)$  in codomain of map. In this case, TreeMap is a predecessor of AbstractMap, but  $\emptyset$  is not a predecessor of  $\{a, b, c\}$

**Order reflecting:** : This function is order reflecting because the predecessor relationship between elements in the codomain is reflected by their pre-images in the domain.

**Order embedding:** By definition, the function is not order embedding as it is not the case that the function is both order preserving and order reflecting.

**Isomorphism:** By definition, the function is not isomorphic as it is not the case that the function is order embedding and surjective.

## 7 Problem 7 (10 pts)

- (a) Transform the definition into a computable function

$$\begin{aligned}\text{compress}(\langle a, a, b, b, b, c, c, a, b \rangle) &= \langle a, b, c, a, b \rangle \\ &= \text{cons}(\text{head}(\Lambda), \text{compress}(\text{tail}(\Lambda)))\end{aligned}$$

- (b) Define f recursively:

$$\begin{aligned}\text{compress}(\Lambda) &= \langle \rangle \text{ if } \Lambda = \langle \rangle \\ \text{compress}(\Lambda) &= \Lambda \text{ if } \text{tail}(\Lambda) = \langle \rangle \\ \text{compress}(\Lambda) &= \text{compress}(\text{tail}(\Lambda)) \text{ if } \text{head}(\Lambda) = \text{head}(\text{tail}(\Lambda)) \\ \text{compress}(\Lambda) &= \text{cons}(\text{head}(\Lambda), \text{compress}(\text{tail}(\Lambda)))\end{aligned}$$

- (c) Unfold the definition for  $\text{compress}(\langle a, a, b, b, c, a \rangle)$

$$\begin{aligned}\text{compress}(\langle a, a, b, b, c, a \rangle) &= \text{compress}(\langle a, b, b, c, a \rangle) \\ &= \text{cons}(a, \text{compress}(\langle b, b, c, a \rangle)) \\ &= \text{cons}(a, \text{compress}(\langle b, c, a \rangle)) \\ &= \text{cons}(a, \text{cons}(b, \text{compress}(\langle c, a \rangle))) \\ &= \text{cons}(a, \text{cons}(b, \text{cons}(c, \text{compress}(\langle a \rangle)))) \\ &= \text{cons}(a, \text{cons}(b, \text{cons}(c, \langle a \rangle))) \\ &= \text{cons}(a, \text{cons}(b, \langle c, a \rangle)) \\ &= \text{cons}(a, \langle b, c, a \rangle) \\ &= \langle a, b, c, a \rangle\end{aligned}$$

- (d) Map definition into Common LISP function

Please find the code attached in compress.lisp file.

Here is the code:

```
(defun compress (list)
  (cond
    ((null list) nil)
    ((null (tail list)) list)
    ((eql (head list) (head (tail list)))
     (compress (tail list)))
    (t (cons (head list) (compress (tail list)))))
)
```