SOEN 331 - S: Formal Methods for Software Engineering

Assignment 1

Nirav Patel, Nathan Grenier

March 7, 2024

PROBLEM 1: Propositional logic (5 pts)

You are shown a set of four cards placed on a table, each of which has a **number** on one side and a **color** on the other side. The visible faces of the cards show the numbers **9** and **11**, and the colors **blue**, and **yellow**.

Which card(s) must you turn over in order to test the truth of the proposition that "*If the face of a card is **blue**, then it has a **prime** number on the other side*"? Explain your reasoning by deciding for each card whether or not it should be turned over and why.

Solution: The proposition states: "If the face of a card is **blue**, then it has a **prime** number on the other side."

Given the cards with visible faces showing **9**, **11**, **blue**, and **yellow**, we need to determine which cards to turn over to test this proposition.

We will apply propositional logic to decide which cards to flip:

1. If a card has a **prime** number on one face, we do not need to turn it over. The face having a **prime** number is not exclusive to blue cards. The "Denying the Antecedent" fallacy should be avoided.

2. If a card has a **non-prime** number on one face, and it is **blue**, we must turn it over to ensure it contradicts the proposition. This would verify the "Modus Tollens" rule of inference.

3. If a card has a **blue** color on one face, we must turn it over to ensure it has a **prime** number on the other side. This would verify the "Modus Ponens" rule of inference.

4. If a card has a **non-blue** color on one face, we do not need to turn it over, as it does not contradict the proposition. The "Affirming the Antecedent" should be avoided.

Now, let's analyze each card:

1. The card showing **9** should be turned over since it is a non-prime number. If it's flipped side is blue, it would violate the proposition.

2. The card showing **11** does not need to be turned over, as the proposition makes no claim regarding non-prime numbers.

3. The card showing **blue** must be turned over to verify if it has a prime number on the other side, as per the proposition.

4. The card showing **yellow** does not need to be turned over because the proposition only concerns cards with blue faces.

Therefore, we must turn over the card with the **blue** and **non-prime** number on the other side to test the proposition. $\therefore$

PROBLEM 2: Predicate logic (14 pts)

Part 1 (8 pts)

Consider types *Object*, and the binary relation Orbits over the domain of all celestial objects, which is codified by clause `object/1` in Prolog (available in `solar.pl`):

1. (2 pts) Construct a formula in predicate logic to define a planet, where planet is defined as an object whose mass is greater than or equal to 0.33×10^{24} KG, and which it orbits around the sun. For all practical purposes, you may ignore the 10^{24} KG factor.

   Solution: To define a planet in predicate logic based on its mass and orbit properties, we can construct the following formula:

   - Let $P(x)$ represent "x is a planet."
   - Let $M(x)$ represent "x has a mass greater than 0.33 KG."
   - Let $O(x, y)$ represent "x orbits around y."
   - Let S represent the sun.

   The formula to define a planet can be expressed as:

   $$P(x) \equiv (M(x) \land O(x, S))$$

   This formula states that an object x is considered a planet if it has a mass greater than 0.33 KG and orbits around the sun. $\therefore$

   Use the formula for *Planet* to construct a formula that defines the binary relation *is_satellite_of* in terms of the binary relation **Orbits**. A satellite is an object that orbits around a planet.

   Solution: To define the binary relation "is satellite of" in terms of the binary relation "Orbits," where a satellite is an object that orbits around a planet, we can construct

the following formula:

- Let $S(x, y)$ represent "x is a satellite of y."
- Let $P(x)$ represent "x is a planet."
- Let $O(x, y)$ represent "x orbits around y."

The formula to define a satellite in terms of the Orbits relation can be expressed as:

$$S(x, y) \equiv O(x, y) \land P(y)$$

2. (3 pts) (**PROGRAMMING**) 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**.

Solution: We can define the following Prolog rules based on the given predicates:

```
% Rule to define a planet
is_planet(X) :- mass(X, Mass), Mass >= 0.33, orbits(X, sun).

% Rule to define a satellite of a planet
is_satellite_of(Satellite, Planet) :- is_planet(Planet), orbits(
    Satellite, Planet).

```

Now, we demonstrate how these rules work using both ground and non-ground queries:

Ground Queries:

(a) Query `?- is_planet(pluto).`

- Type: Ground Query
- Result: ‘`false`‘

(b) Query `?- is_planet(mars).`

- Type: Ground Query
- Result: ‘`true`‘

(c) Query `?- is_satellite_of(moon, earth).`

- Type: Ground Query
- Result: ‘`true`‘

Non-Ground Queries:

(a) Query `?- is_planet(P).`

- Type: Non-Ground Query
- Result:

  `P = mercury;`

  `P = venus;`

  `P = earth;`

  `P = mars;`

  `P = jupiter;`

  `P = saturn;`

  `P = uranus;`

  `P = neptune;`

  `false`

(b) Query `?- is_satellite_of(S, mars).`

- Type: Non-Ground Query
- Result:

  `S=deimos;`

  `S=phobos;`

  `false`

(c) Query `?- is_satellite_of(moon, Planet).`

- Type: Non-Ground Query
- Result:

```
Planet = earth;

false
```

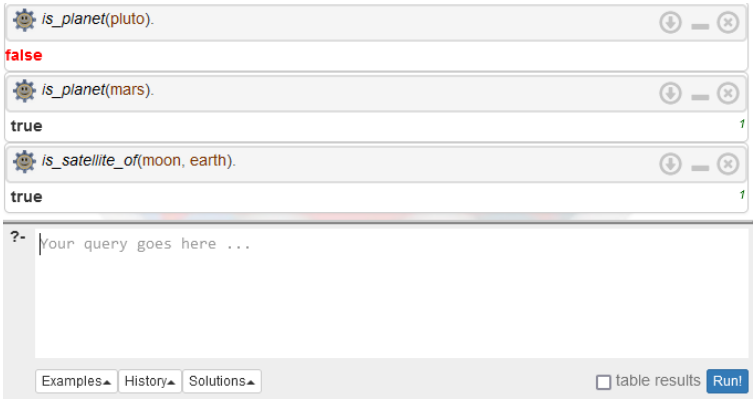


Figure 1: Interaction for Ground Queries

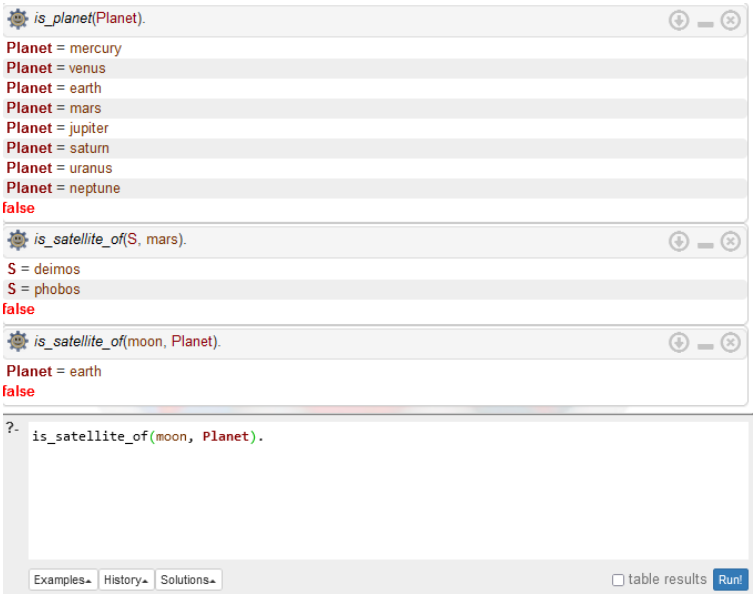


Figure 2: Interaction for Non-Ground Queries

3. (3 pts) (**PROGRAMMING**) Construct a Prolog rule `obtain_all_satellites/2` that succeeds by returning a collection of all satellites of a given planet.

Solution: We can define a rule named `all_satellites/2` that collects all the satellites of a specified planet. The following describes an example of how the rule can be implemented:

```
% Rule to obtain all satellites of a given planet
all_satellites(Planet, Satellites) :- findall(Satellite,
    is_satellite_of(Satellite, Planet), Satellites).

```

In this rule:

- `Planet` is the input parameter representing the planet for which we want to find its satellites.
- `Satellites` is the output parameter that will contain a collection of all satellites of the specified planet.
- `findall/3` is used to collect all solutions for `Satellite` that satisfy the condition `is_satellite_of(Satellite, Planet)`.

We can test this rule using the following query:

Query:

```
?- all_satellites(jupiter, Satellites).
```

Result:

```
Satellites = [arche, callisto, europa, io, themisto]
```

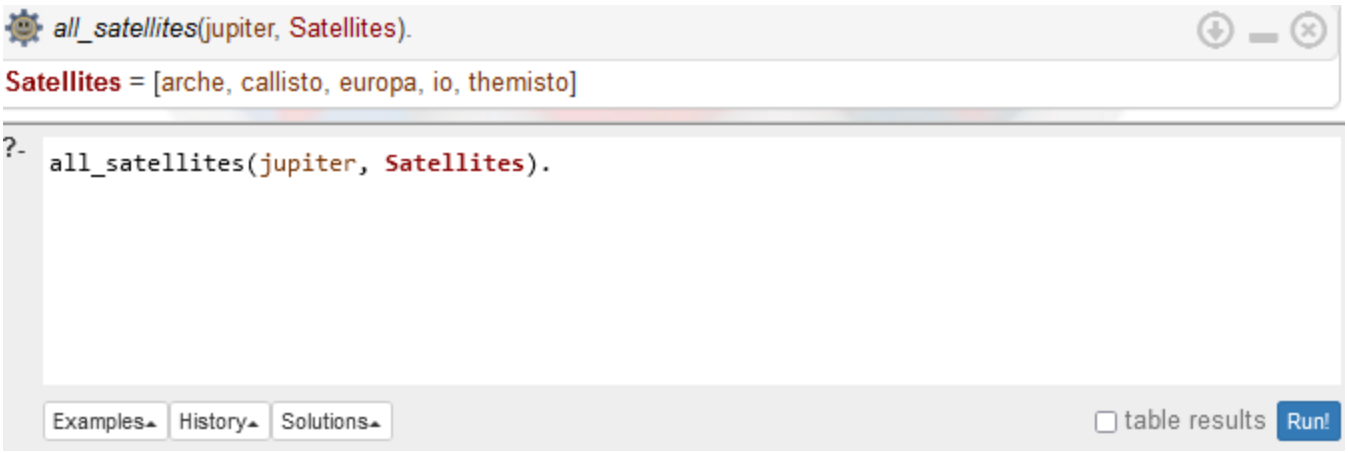


Figure 3: Interaction for Non-Ground Queries

Part 2: Categorical propositions (2 pts)

In the domain of all integers, let $number(x)$ denote the statement "x is a number", and $composite(x)$ denote the statement "x is a composite." Formalize the following sentences and indicate their corresponding formal type.

Solution: We will use the predicates number(x) and composite(x), and their corresponding formal types (A, E, I, O) to formalize the sentences.

1. "Some numbers are not composite."
   - Formalization: $\exists\, x(number(x) \wedge \neg composite(x))$
   - Corresponding Formal Type: O (Particular Negative)
2. "No numbers are prime."
   - Formalization: $\forall\, x(number(x) \rightarrow composite(x))$
   - Corresponding Formal Type: A (Universal Affirmative)
3. "Some numbers are not prime."
   - Formalization: $\exists\, x(number(x) \wedge composite(x))$
   - Corresponding Formal Type: I (Particular Affirmative)

4. "All numbers are prime."

   - Formalization: $\forall\, x(number(x) \rightarrow \neg composite(x))$
   - Corresponding Formal Type: E (Universal Negative)

Part 3: Categorical propositions (4 pts)

For subject S and predicate P, we can express the Type A categorical proposition as

$$\forall\, s : S \mid s \in P$$

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

   Solution: We first determine the negation of A ($\neg A$):

   $$\neg A = \neg(\forall\, s : S \mid s \in P) = \exists\, s : S \mid s \notin P$$

   The equivalent expression obtained from determining the negation of A is logically equivalent to the formulation of O. Similarly, we can obtain an A-type proposition using the negation of O:

   $$\neg(\exists\, s : S \mid s \notin P) = \forall\, s : S \mid s \in P$$

   We have therefore proven that negating A is logically equivalent to obtaining O, and vice-versa.

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

   Solution: We first determine the negation of E ($\neg E$):

   $$\neg(\forall\, s : S \mid s \notin P) = \exists\, s : S \mid s \in P$$

The equivalent expression obtained from determining the negation of E is logically equivalent to the formulation of I. Similarly, we can obtain an E-type proposition using the negation of I:

$$\neg(\exists\, s : S \mid s \in P) = \forall\, s : S \mid s \notin P$$

We have therefore proven that negating E is logically equivalent to obtaining I, and vice-versa.

PROBLEM 3: Temporal logic (22 pts)

Part 1 (16 pts)

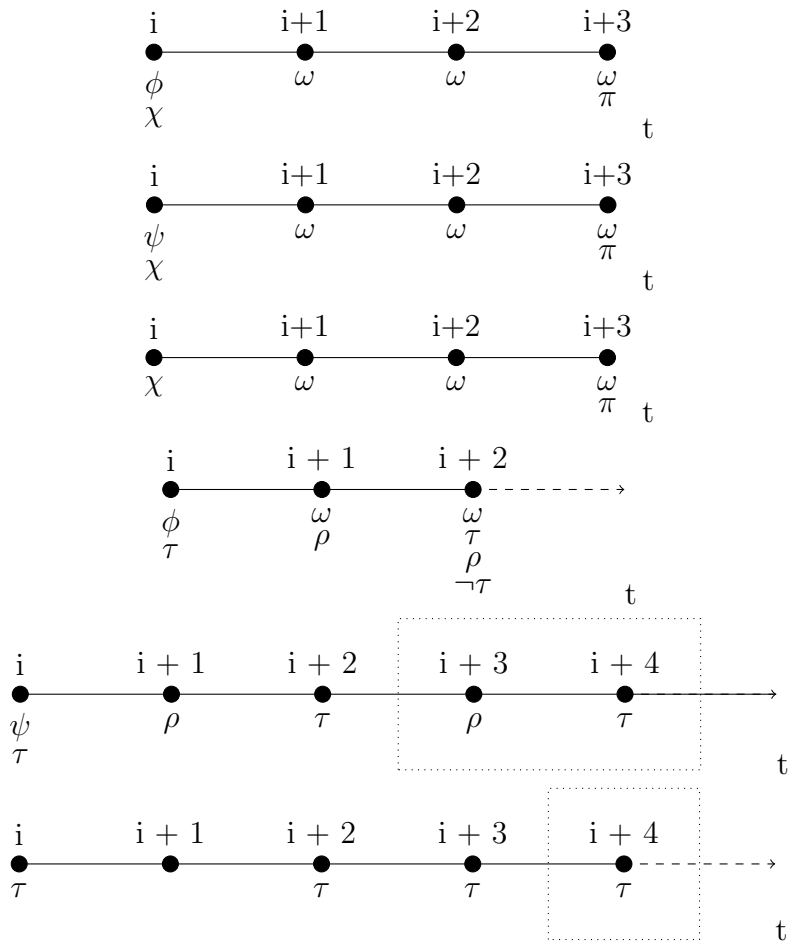
The behavior of a program is expressed by the following temporal formula:

$$\Box \left[\begin{array}{c} \textbf{start} \rightarrow (\neg\phi \vee \neg\psi) \\ \\ \textbf{start} \rightarrow \chi \oplus \tau \\ \\ \phi \vee \chi \rightarrow \bigcirc(\pi \mathcal{R} \omega) \\ \\ x \wedge \bigcirc\omega \rightarrow \bigcirc^3\pi \\ \\ \tau \rightarrow \bigcirc^2(\tau \mathcal{W} \rho) \\ \\ \omega \wedge \bigcirc\rho \rightarrow \bigcirc^2\rho \\ \\ \psi \wedge \tau \rightarrow \bigcirc(\rho \mathcal{U} \tau) \\ \\ \rho \wedge \bigcirc\tau \rightarrow \bigcirc^2\rho \\ \\ \phi \wedge \tau \rightarrow \bigcirc(\tau \mathcal{R} \rho) \end{array} \right]$$

Figure 4: Temporal Formula

1. (9 pts) Visualize all models of behavior.

   Solution:



2. (7 pts) Make observations on the visualization by specifying exact conditions about termination, non-termination and consistency (or lack thereof), if any exist.

   Solution: In regards to the first three models, we can observe that they are terminating. In other words, they can be described through the following expressions:

$$\mathbf{1} : \langle(\phi \wedge \chi), (\omega), (\omega), (\omega \wedge \pi)\rangle$$

$$\mathbf{2} : \langle(\psi \wedge \chi), (\omega), (\omega), (\omega \wedge \pi)\rangle$$

$$\mathbf{3} : \langle(\chi), (\omega), (\omega), (\omega \wedge \pi)\rangle$$

In regards to the fourth model, given the initial conditions $(\phi \wedge \tau)$, there is a contradiction between two specific propositions, namely $(\tau \rightarrow \bigcirc^2(\tau \mathcal{W} \rho))$ and $(\rho \wedge \bigcirc\tau \rightarrow \bigcirc^2\rho)$ which cause the time $i + 3$ to contain the atomic proposition τ in both its true and false form, which contradicts the argument and falsifies this specific model.

Finally, for the remaining two models, we can determine that the last two steps of the visual figure are infinitely repeating, therefore representing non-terminating models. Specifically, the global set of propositions contained in the argument does not possess any conditions enabling the last three models to terminate.

In the last two models, the set of repeating propositions are as follows:

$$\mathbf{5} : (\tau), (\tau \wedge \rho) \text{ if beginning with } (\psi \wedge \tau)$$

$$\mathbf{6} : (\tau) \text{ if beginning with } (\tau)$$

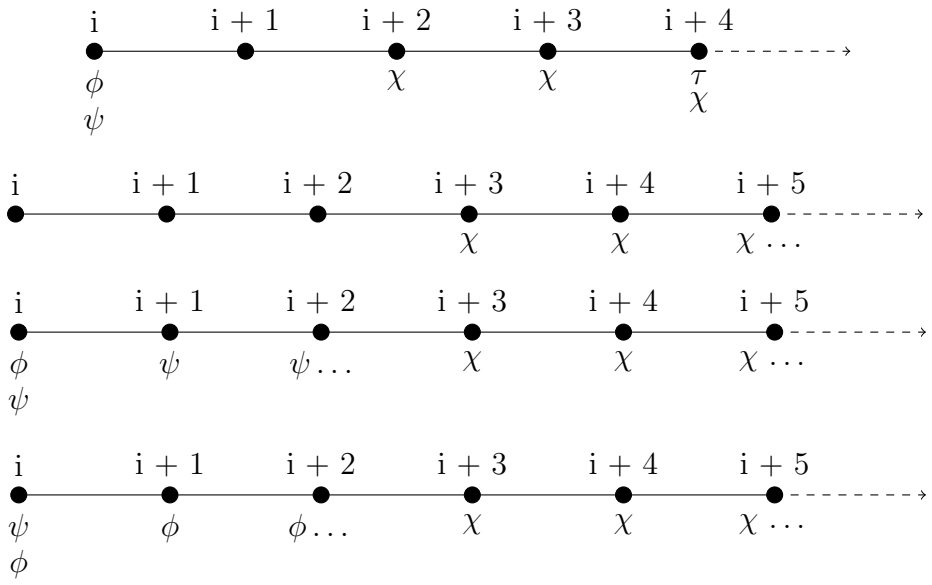
Part 2 (6 pts)

1. (2 pts) Formalize and visualize the following requirement: "If none of ϕ or ψ are invariants, then starting from time $= i + 2$, χ will eventually become true and it will remain true up to and including the moment τ first becomes true. Note that there exists no guarantee that τ ever becomes true."

   Solution: We can express it as follows:

$$\neg(\Box\phi \wedge \Box\psi) \rightarrow (\bigcirc^2(\Diamond\chi \wedge (\tau R \chi)))$$

and, among the many ways to visualize the requirement, we can discuss the different variations using the following two models:



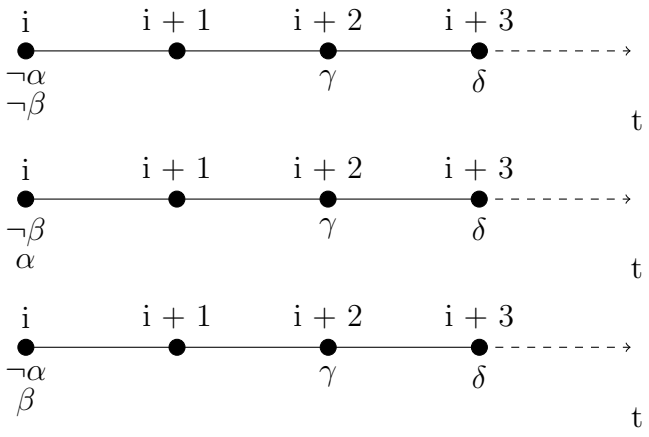
It should be noted that the requirement "none of ϕ or ψ are invariants" introduces the idea that both atomic propositions ϕ and ψ can be true at some point a finite number of times, and can also be false in other times as well.

Additionally, with the lack of guarantee that the atomic proposition τ will ever become true, we can visualize the requirement as having the proposition become true at some point in time or simply never become true, letting the atomic proposition χ remain true.

2. (2 pts) Describe then visualize the following requirement: $(\neg\alpha \vee \neg\beta) \rightarrow \bigcirc \diamond (\gamma \, \mathcal{U} \, \delta)$

Solution: We can describe and visualize the above requirement as follows:

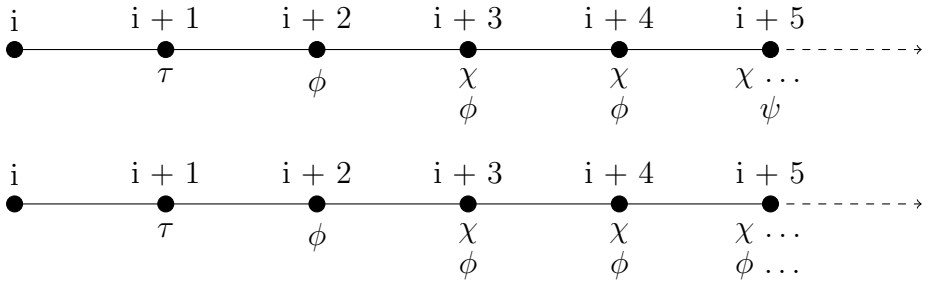
- If neither α nor β is true, then at the next time instant, there exists a path where γ holds until δ becomes true.



3. (2 pts) Describe then visualize the following requirement: $(\bigcirc\tau \wedge \bigcirc\diamond\Box\chi) \rightarrow \bigcirc^2(\phi \mathcal{W} \psi)$

Solution: We can describe and visualize the above requirement as follows:

- If at the next time instant τ holds and at some future time there exists a path where χ is invariant, then two time steps later, ϕ will remain true until ψ becomes true. There is no guarantee that ψ will become true.



PROBLEM 4: Unordered Structures (14 pts)

Consider the following set:

$$Languages = Ruby, Go, Lisp, Rust, C, Groovy, Python, Clojure, \{Lua, Groovy, C\}$$

Answer the following questions:

1. (1 pt) Provide a description (in plain English) on what is meant by $\mathcal{P}$ *Languages*.

   Answer: The set $\mathcal{P}Languages$ refers to the power set of the set "Languages." In plain English, the power set of a set is the set of all possible subsets that can be formed from the original set. In this context, $\mathcal{P}Languages$ would represent all possible combinations of subsets that can be created from the set of languages provided, including the empty set, individual languages, and combinations of multiple languages.

2. (1 pt) Describe it in detail (a) what the expression *Favorites* : $\mathcal{P}$ *Languages* signifies and (b) how it should be interpreted. (c) What would be any legitimate value for variable *Favorites*?

   (a) The expression *Favorites* : $\mathcal{P}Languages$ signifies that the variable *Favorites* is associated with the power set of the set "Languages." This means that *Favorites* can take on values that are subsets of the set of languages provided, including all possible combinations of languages within the set.

   (b) The expression should be interpreted as allowing the variable *Favorites* to represent any subset of languages from the given set "Languages." It provides flexibility for *Favorites* to encompass various combinations of programming languages, including individual languages, multiple languages, or even an empty set.

   (c) Legitimate values for the variable *Favorites* could include any subset of the languages listed in the set "Languages," such as individual languages like Ruby or Python, combinations like Ruby, Go, or even the empty set if no language is selected as a favorite.

3. (1 pt) What does the expression $Favorites = \mathcal{P}\ Languages$ signify and (b) Is it semantically equivalent to $Favorites : \mathcal{P}\ Languages$?

   (a) The expression $Favorites = \mathcal{P}\ Languages$ signifies that the variable *Favorites* is assigned the value of the power set of the set "Languages." This means that *Favorites* represents all possible subsets that can be formed from the original set of languages provided.

   (b) Semantically, $Favorites = \mathcal{P}\ Languages$ and $Favorites : \mathcal{P}\ Languages$ are not equivalent. The former indicates that the variable *Favorites* is set to be equal to the power set of languages, implying a specific assignment of values. On the other hand, the latter notation suggests a relationship or association between *Favorites* and the power set of languages without explicitly assigning a specific value.

4. (1 pt) Is $\{Lua, Groovy, C\} \in \mathcal{P}\ Languages$? Explain in detail.

   Answer: The expression $\{Lua, Groovy, C\} \in \mathcal{P}\ Languages$ signifies whether the set $\{Lua, Groovy, C\}$ is an element of the power set of the set "Languages." In detail, the power set of a set contains all possible subsets of that set, including the empty set and the set itself. Therefore, for $\{Lua, Groovy, C\}$ to be in the power set of "Languages," it must be a valid subset that can be formed from the original set of languages provided.

   In this case, since $\{Lua, Groovy, C\}$ is a specific combination of languages from the set "Languages," it is indeed an element of the power set. This is because any subset that can be formed from the original set, including combinations of languages like $\{Lua, Groovy, C\}$, is considered part of the power set.

5. (1 pt) Is $\{\{Lua, Groovy, C\}\} \subset \mathcal{P}\ Languages$? Explain in detail.

   Answer: The expression $\{\{Lua, Groovy, C\}\} \subset \mathcal{P}\ Languages$ signifies whether the set $\{\{Lua, Groovy, C\}\}$ is a subset of the power set of the set "Languages." In detail, a subset is a set that contains only elements that are also in another set. The

power set of a set contains all possible subsets that can be formed from the original set.

In this case, the set $\{\{Lua, Groovy, C\}\}$ is not a subset of the power set of "Languages" because the power set includes all possible combinations of subsets of the original set. The element $\{\{Lua, Groovy, C\}\}$ is a nested set within itself and does not directly match any subset that can be formed from the original set of languages provided.

6. (1 pt) What is the difference between the following two variable declarations and are the two variables atomic or non-atomic?

   (a) $\lambda_1 : Languages$

   Answer: This variable declaration λ_1 signifies that it is associated with the set "Languages." Here, λ_1 can take on values directly from the set of languages provided. The variable λ_1 is atomic as it directly refers to individual elements within the set "Languages."

   (b) $\lambda_2 : \mathcal{P}\ Languages$

   Answer: In contrast, the variable declaration λ_2 indicates that it is linked to the power set of the set "Languages." This means that λ_2 can represent any subset that can be formed from the original set of languages. The variable λ_2 is non-atomic as it encompasses various combinations and subsets of languages rather than individual elements directly from the set.

7. (1 pt) 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?

   Answer: The type of the variable in the expression $Library = C, Ruby, Go$, where it is used to hold any collection that can be constructed from *Languages*, would typically be an array or a list in programming languages like Ruby. The variable *Library* is intended to store a collection of programming languages, and using an array or list

data structure would allow for easy manipulation and access to these languages within the collection.

8. (1 pt) Is $\{\varnothing\} \in \mathcal{P}\ Languages$? Explain.

   Answer: The set $\{\varnothing\}$ is indeed an element of the power set of the set "Languages." This is because the power set includes all possible subsets that can be formed from the original set, including the empty set. Since $\{\varnothing\}$ represents a subset that contains only the empty set, it is a valid element of the power set of "Languages."

9. (6 pts) (**PROGRAMMING**) Use Common LISP to implement the functions below and demonstrate the behavior of each function with a listing of your interaction with the language environment. **Do not use imperative constructs**.

   - **is-memberp** Returns true if its first argument is found to be a member in the set provided by the second argument. Returns nil (false) otherwise.

```
;; Define is-memberp function
(defun is-memberp (element set)
  "Check if ELEMENT is a member of SET."
  (cond
    ((null set) nil)                    ; If empty set, return nil.
    ((equal element (car set)) t)       ; If the first element of the set equals ELEMENT, return true.
    (t (is-memberp element (cdr set)))))) ; Otherwise, recursively check the rest of the set.

;; Demonstrate the behavior of the functions
;; Test is-memberp function
(print (is-memberp 'Ruby '(Python Ruby Lisp))) ; This will output T
(print (is-memberp 'Go '(Python Ruby Lisp)))   ; This will output NIL
```

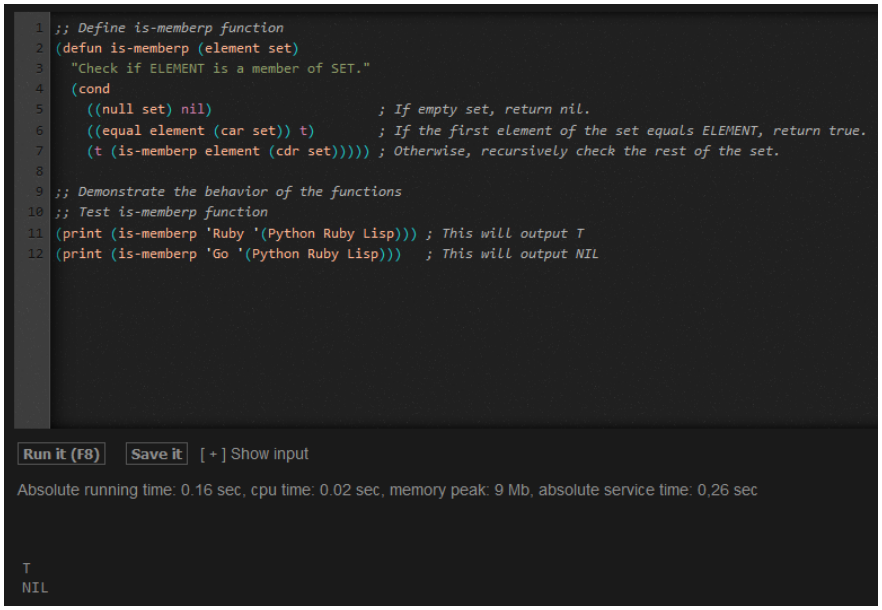


Figure 5: Interaction for Ground Queries

- **equal-setsp** Returns true if its two arguments are two identical sets. Returns nil (false) otherwise. You may assume that neither argument (set) contains any redundancies.

```
;; Define equal-setsp function
(defun equal-setsp (set1 set2)
  "Check if SET1 and SET2 are equal sets."
  (and (subsetp-func set1 set2) (subsetp-func set2 set1)))

;; Define subsetp-func helper function
(defun subsetp-func (subset set)
  "Check if SUBSET is a subset of SET using a functional approach."
  (cond
    ((null subset) t)                    ; If the subset is empty, it is a subset.
    ((is-memberp (car subset) set)       ; If the first element of the subset is in the set,
     (subsetp-func (cdr subset) set))    ; continue checking the rest of the subset.
    (t nil)))                            ; Otherwise, it's not a
```

```
    subset.
14
15 ;; Define is-memberp helper function
16 (defun is-memberp (element set)
17   "Check if ELEMENT is a member of SET."
18   (cond
19     ((null set) nil)                      ; If empty set, return
     nil.
20     ((equal element (car set)) t)         ; If the first element
     of the set equals ELEMENT, return true.
21     (t (is-memberp element (cdr set))))) ; Otherwise, recursively
      check the rest of the set.
22
23 ;; Test equal-setsp function
24 (print (equal-setsp '(Python Ruby Lisp) '(Lisp Python Ruby))) ;
    This will output T
25 (print (equal-setsp '(Python Ruby) '(Lisp Python Ruby)))      ;
    This will output NIL
```

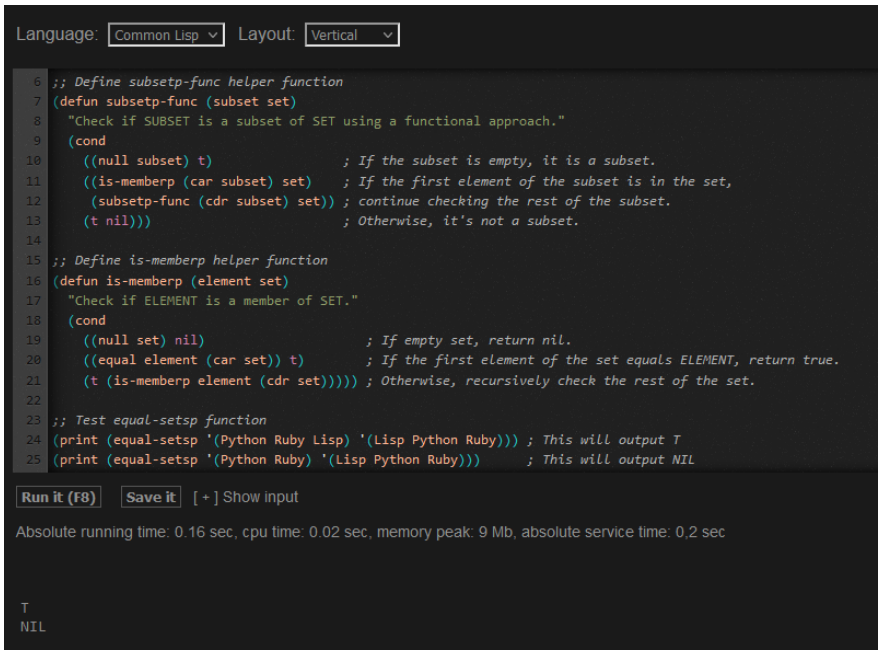


Figure 6: Interaction for Ground Queries

PROBLEM 5: Ordered Structures (15 pts)

Consider the Queue Abstract Data Type (ADT), $\mathbb{Q}$, defined over some generic type $\mathbb{T}$ and defined using two stacks, $\sum_1$ and $\sum_2$, where a Stack ADT is implemented as a list, Λ. (*Hint:* This description implies that the only available operations are those defined for a list.)

1. (10 pts) Define operations *Enqueue*($\mathbb{Q}, \mathbb{T}$) and *Dequeue*($\mathbb{Q}$)

   Solution: To enqueue an element of type $\mathbb{T}$ into the queue $\mathbb{Q}$:

   (a) We simply add the element to one of the stacks (let's say $\sum_1$).

   (b) There's no need to consider the contents of the other stack for this operation.

   (c) This operation corresponds to appending the element to the end of the list representing $\sum_1$.

   To dequeue an element from the queue $\mathbb{Q}$:

   (a) If $\sum_2$ is not empty, we pop and return the top element from $\sum_2$.

   (b) If $\sum_2$ is empty:

   - We transfer all elements from $\sum_1$ to $\sum_2$, effectively reversing their order.
   - Then, we pop and return the top element from $\sum_2$.

   (c) This operation involves transferring elements between stacks to maintain the FIFO (First-In-First-Out) order.

   (d) This corresponds to removing the last element from the list representing $\sum_2$.

   To better visualize the operations, we can imagine the following scenario:

   Let's imagine managing a queue at a ticket counter. Initially, there are two empty stacks, one for incoming tickets and the other for processing them. Enqueueing involves adding tickets to the incoming stack. For instance, when the first three tickets arrive (let's say labeled A, B, and C, in that particular order), they are placed in the incoming stack in reverse order, so A is at the bottom and C is at the top.

When it's time to serve a customer, we transfer tickets from the incoming stack to the processing stack, ensuring we serve the oldest ticket first. In this case, we transfer tickets from the incoming stack to the processing stack, so now the processing stack holds tickets A, B, and C in the desired, "queue-like", order (FIFO). We serve ticket A first, transferring it out of the processing stack.

Subsequent tickets are added to the incoming stack, maintaining the queue order. When we serve a ticket, we remove it from the processing stack, ensuring the next ticket served is the next in line. This process continues, ensuring that tickets are served in the order they were received. Tickets from the incoming stack are only sent to the processing stack when the processing stack is empty.

2. (3 pts) (**PROGRAMMING**) Use Common LISP to define two stacks (`stack1` and `stack2` as global variables to hold the collection, and implement functions `enqueue` and `dequeue`.

```
(defparameter stack1 '()) ; Define stack1 as an empty list
(defparameter stack2 '()) ; Define stack2 as an empty list

(defun enqueue (element) ; Define the enqueue function
  (push element stack1)) ; Push the element onto stack1

(defun dequeue () ; Define the dequeue function
  (if (null stack2) ; Check if stack2 is empty
      (progn ; If stack2 is empty
        (setf stack2 (reverse stack1)) ; Transfer elements from
   stack1 to stack2 and reverse their order
        (setf stack1 '())) ; Reset stack1 to an empty list
  (pop stack2)) ; Pop and return the top element from stack2

; Test enqueue and dequeue operations
(enqueue 1) ; Enqueue element 1
(enqueue 2) ; Enqueue element 2
(enqueue 3) ; Enqueue element 3

(format t "Dequeued element: ~a~%" (dequeue)) ; Dequeue and print 1
```

```
(format t "Dequeued element: ~a~%" (dequeue)) ; Dequeue and print 2
(enqueue 4) ; Enqueue element 4
(format t "Dequeued element: ~a~%" (dequeue)) ; Dequeue and print 3
```

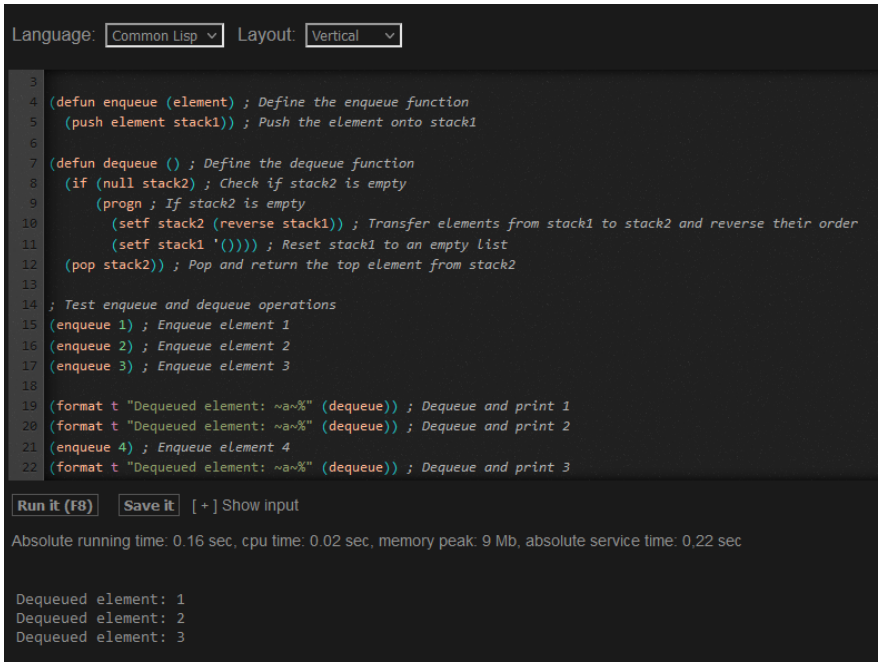


Figure 7: Interaction for enqueue/dequeue Functions

3. (2 pts) (**PROGRAMMING**) Implement operations *head*, *tail* and *cons* in Prolog and demonstrate their usage.

```
% head/2 returns the first element of a list
head([Head|_], Head).

% tail/2 returns the list without its first element
tail([_|Tail], Tail).

% cons/3 adds an element to the beginning of a list
cons(Element, List, [Element|List]).
```

(a) **head/2**: This predicate retrieves the first element of a list. It takes a list as its first argument and a variable to unify with the first element of the list. In Prolog, lists are represented as nested pairs [Head|Tail], where Head is the first element

and Tail is the rest of the list. The head/2 predicate pattern-matches the list [Head|_] and unifies Head with the first element of the list.

(b) **tail/2**: This predicate returns the list without its first element. Similarly to head/2, it also takes a list as its first argument and a variable to unify with the tail of the list. It pattern-matches the list [_|Tail], ignoring the first element and unifying Tail with the remaining elements of the list.

(c) **cons/3**: This predicate adds an element to the beginning of a list. It takes three arguments: an element to add, an existing list, and a variable to unify with the resulting list. It constructs a new list by prepending the given element to the existing list, resulting in [Element|List]. This predicate is commonly used to build lists recursively in Prolog.

Here are some queries that test the above Prolog predicates:

(a) Query to get the first element of a list:

- **Input:** `head([1,2,3,4], X).`
- **Output:** `X = 1`

(b) Query to get the tail of a list:

- **Input:** `tail([1,2,3,4], X).`
- **Output:** `X = [2, 3, 4]`

(c) Query to add an element to the beginning of a list:

- **Input:** `cons(5, [1,2,3,4], X).`
- **Output:** `X = [5, 1, 2, 3, 4]`

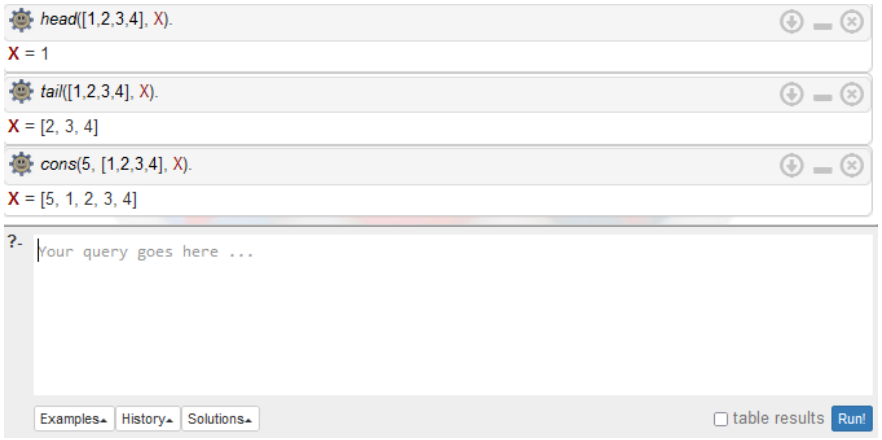


Figure 8: Interaction for Ground Queries

PROBLEM 6: Binary Relations, Functions and Orderings (20 pts)

Part 1 (6 pts)

1. (2 pts) Consider the binary relation R: "is of type" in the domain of types in the Java API. Prove that R is *a partial order*.

   Solution: To prove that the binary relation R: "is of type" in the domain of types in the Java API is a partial order, we need to show that it satisfies three properties: reflexivity, antisymmetry, and transitivity.

   (a) **Reflexivity:**

   - For any type T, T is of type T. This property holds as every type is of its own type.

   (b) **Antisymmetry:**

   - If T_1 is of type T_2 and T_2 is of type T_1, then $T_1 = T_2$. This property holds as types are uniquely defined.

   (c) **Transitivity:**

   - If T_1 is of type T_2 and T_2 is of type T_3, then T_1 is of type T_3. This property holds as types can be hierarchically related.

Therefore, the binary relation R: "is of type" in the domain of types in the Java API satisfies reflexivity, antisymmetry, and transitivity, making it a partial order.

2. (2 pts) Given the set of vertices

$$V_1 = \{Dictionary, TreeMap, SortedMap, AbstractMap, Hashtable, LinkedHashMap, Map, NavigableMap, HashMap\}$$

and the set of edges

$$E = \{(LinkedHashMap, HashMap), (HashMap, AbstractMap), (Hashtable, Dictionary), (TreeMap, AbstractMap), (TreeMap, NavigableMap), (NavigableMap, SortedMap), (Hashtable, Map), (AbstractMap, Map), (SortedMap, Map)\}$$

Prove that (V_1, R) is a *poset*.

Solution: To demonstrate that the set (V_1, R) is a poset, we first establish that the binary relation R ("is of type") is a partial order. This involves verifying three key properties: reflexivity, antisymmetry, and transitivity.

- Reflexivity: Every type is inherently of its own type, ensuring that the relation is reflexive.
- Antisymmetry: The relation between types is uniquely defined, preventing two distinct types from being of each other's type.
- Transitivity: Types can be hierarchically related, allowing for the transitive property to hold.

Having confirmed that R is a partial order, we proceed to analyze the set of vertices and edges provided. By constructing a directed acyclic graph based on these elements,

we observe that the graph maintains acyclicity. This acyclic nature aligns with the requirements of a partial order set, reinforcing the poset structure of (V_1, R).

Therefore, through the fulfillment of partial order properties and the absence of cycles in the directed graph representing (V_1, R), we can confidently assert that this set forms a poset.

3. (2 pts) Create a Hasse Diagram to visualize V, R and identify *maximal* and *minimal* elements.

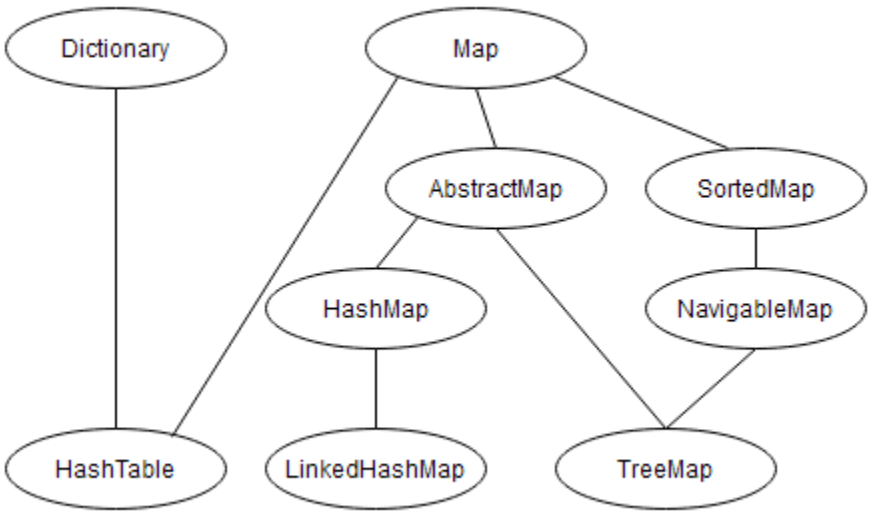


Figure 9: Hasse Diagram Depicting (V_1, R)

- **Minimal Elements**: *HashTable*, *LinkedHashMap* and *TreeMap*
- **Maximal Elements**: *Dictionary* and *Map*

Part 2 (6 pts)

1. (2 pts) Consider the binary relation $\subseteq$: "is subset of" in the domain of sets. Prove that $\subseteq$ is a partial order.

   To prove that the binary relation $\subseteq$ ("is subset of") in the domain of sets is a partial order, we need to demonstrate three key properties: reflexivity, antisymmetry, and transitivity.

- **Reflexivity**: Every set is a subset of itself, satisfying the reflexivity property.
- **Antisymmetry**: If set A is a subset of set B and B is a subset of A, then $A = B$, ensuring antisymmetry.
- **Transitivity**: If set A is a subset of set B and B is a subset of set C, then A is also a subset of C, validating transitivity.

Since $\subseteq$ fulfills these properties, it qualifies as a partial order.

2. (2 pts) Given the set $V_2 = \{a, b, c\}$, prove that $(\mathcal{P}(V_2), \subseteq)$ is a poset.

   Solution: To prove that the relation $\subseteq$ applied over the set $V_2 = \{a, b, c\}$ is a poset, we need to demonstrate three key properties: reflexivity, antisymmetry, and transitivity.

   - **Reflexivity:** For any set S in $\mathcal{P}(V_2)$, we have $S \subseteq S$ since every set is a subset of itself. Therefore, $(\mathcal{P}(V_2), \subseteq)$ is reflexive.
   - **Antisymmetry:** If $A \subseteq B$ and $B \subseteq A$ for sets A and B in $\mathcal{P}(V_2)$, then $A = B$. This is true because two sets are equal if and only if they contain the same elements. Therefore, $(\mathcal{P}(V_2), \subseteq)$ is antisymmetric.
   - **Transitivity:** If $A \subseteq B$ and $B \subseteq C$ for sets A, B, and C in $\mathcal{P}(V_2)$, then $A \subseteq C$. This is true because if every element of A is in B and every element of B is in C, then every element of A is also in C. Therefore, $(\mathcal{P}(V_2), \subseteq)$ is transitive.

   Since $(\mathcal{P}(V_2), \subseteq)$ satisfies reflexivity, antisymmetry, and transitivity, it is a poset.

3. (2 pts) Create a Hasse Diagram to visualize $(\mathcal{P}(V_2), \subseteq)$ and identify *maximal* and *minimal* elements.

   Solution: The maximal and minimal elements of the diagram are *a,b,c* and $\{\varnothing\}$, respectively.

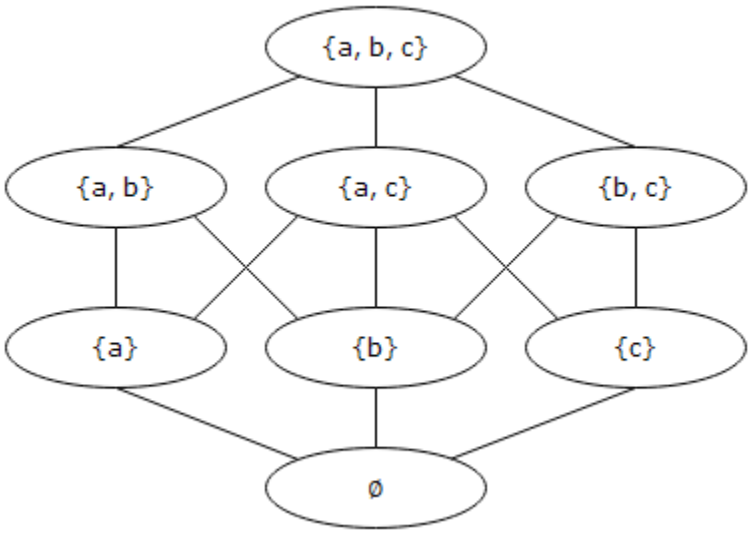


Figure 10: Hasse Diagram Depicting $(\mathcal{P}(V_2), \subseteq)$

Part 3 (8 pts)

Consider (V_1, R) and $(\mathcal{P}(V_2), \subseteq)$ from Parts 1 and 2 respectively. Assume the following mapping, captured by variable *map*:

$$\begin{aligned} map = \{&Dictionary \mapsto \{a, b, c\}, \\ &AbstractMap \mapsto \{a, b, c\}, \\ &NavigableMap \mapsto \{c\}, \\ &TreeMap \mapsto \varnothing\} \end{aligned}$$

Is *map* a function? Discuss **in detail** (in plain English and formally) all applicable properties (total vs. partial function, injectivity, surjectivity, bijection, order preserving, order reflecting, order embedding, isomorphism).

Solution: The variable *map* captures a mapping between elements in (V_1, R) and subsets in $(\mathcal{P}(V_2), \subseteq)$. Let's evaluate whether *map* qualifies as a function and explore various properties it exhibits.

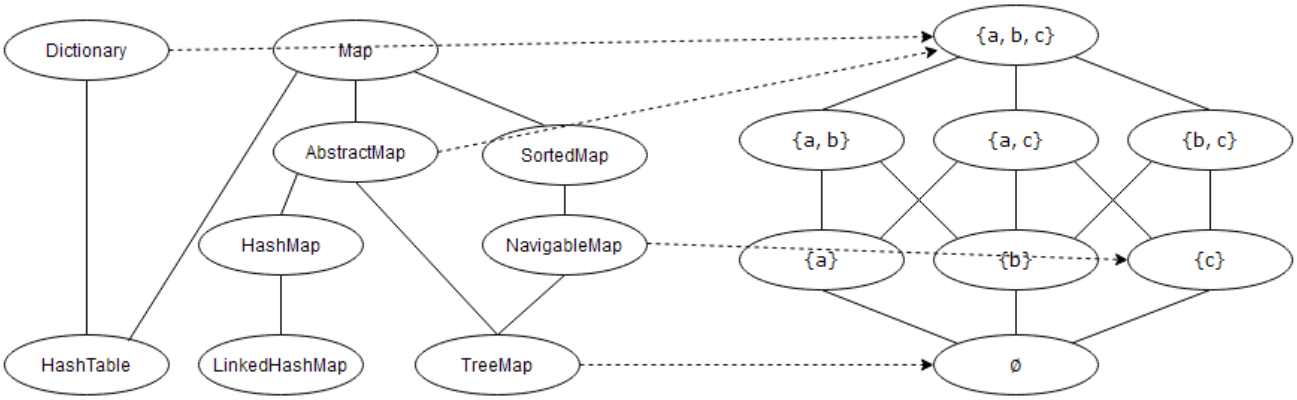


Figure 11: Function *map*

We begin by defining the various properties of a function in plain English:

- **Functionality**: A function assigns each element from the domain to exactly one element in the codomain.
- **Total vs. Partial Function**: A total function maps every element from the domain to an element in the codomain, while a partial function may not cover all elements.
- **Injectivity**: An injective function ensures distinct elements in the domain map to distinct elements in the codomain.
- **Surjectivity**: A surjective function covers every element in the codomain.
- **Bijection**: A bijective function is both injective and surjective, establishing a one-to-one correspondence.
- **Order Preservation**: Preserving order means that if $a < b$ in the domain, then $f(a) < f(b)$ in the codomain.
- **Order Reflection**: Reflecting order implies that if $f(a) < f(b)$ in the codomain, then $a < b$ in the domain.
- **Order Embedding**: A function is order-embedding if it is both order-preserving and order-reflecting.

- **Isomorphism**: A function that is order-embedding and surjective is called isomorphic.

We can now perform a formal analysis using the definitions of the properties that we are working with:

- **Functionality Check**: For each element in (V_1, R), *map* assigns a unique subset from $(\mathcal{P}(V_2), \subseteq)$, meeting the function criteria.
- **Total vs. Partial Function**: *map* is a partial function since not all elements from (V_1, R) are mapped to subsets in $(\mathcal{P}(V_2), \subseteq)$.
- **Injectivity**: Since multiple elements from (V_1, R) map to $\{a, b, c\}$, *map* is not injective.
- **Surjectivity**: Not every subset of the power set of $\{a, b, c\}$ are mapped to elements in the domain, (V_1, R). *map* is not surjective.
- **Bijection**: As *map* is not injective nor surjective, it cannot be bijective.
- **Order Preservation/Reflection/Embedding**: The mapping does not reflect or preserve any order relationships between elements of (V_1, R) or subsets of $(\mathcal{P}(V_2), \subseteq)$.
- **Isomorphism**: Since *map* is neither order-embedding nor surjective, it cannot be isomorphic.

In summary, while *map* functions as a partial function and exhibits surjectivity for certain subsets, it lacks properties like injectivity and order preservation/reflection necessary for bijection or isomorphism.

PROBLEM 7: Construction Techniques (10 pts)

Define a function *compress* : *lists*($\mathbb{T}$) $\rightarrow$ *lists*($\mathbb{T}$) that accepts a list argument (of some generic type $\mathbb{T}$) and returns a list without any *subsequent* redundancies.

1. (2 pts) Transform the definition into a computable function.

   We transform the definition of *compress* into a computable function using available operations on the underlying structure (list).

   We can use `cons` as follows:

$$\text{compress}(lst) = \begin{cases} [], & \text{if } lst = [] \\ [\text{x}], & \text{if length}(lst) = 1 \text{ where } lst = [\text{x}] \\ \text{cons}(x, \text{compress}(y)), & \text{otherwise where } lst = \text{x:y (head:tail) and } x \neq \text{head}(y) \\ \text{compress}(y), & \text{otherwise where } lst = \text{x:y (head:tail) and } x = \text{head}(y) \end{cases}$$

2. (3 pts) Define f recursively.

   To define the function f recursively for compressing a list without subsequent redundancies, we can express it as follows:

   - Let $L = [a_1, a_2, ..., a_n]$ be a list of elements of some generic type $\mathbb{T}$.
   - The function $f(n)$ recursively compresses the list to remove subsequent redundancies.

   The recursive definition of function f can be outlined as:

   (a) **Base Case**:

   - If the list is empty, then $f(n) = []$, as an empty list has no subsequent redundancies.
   - If $n = 1$, then $f(n) = [a_1]$, as a single-element list has no subsequent redundancies.

(b) **Recursive Steps**:

- If $a_n = a_{n-1}$, then $f(n) = f(n-1)$, as the current element is redundant.
- If $a_n \neq a_{n-1}$, then $f(n) = f(n-1) \cup [a_n]$, adding the current element to the compressed list.

3. (2 pts) Unfold your definition for $compress(\langle a, a, b, b, c, a\rangle)$.

Let's unfold the recursive definition of the function *compress* for the input list $\langle a, a, b, b, c, a\rangle$ step by step:

(a) Initial List: $\langle a, a, b, b, c, a\rangle$

(b) $f(1) = \langle a\rangle$ (Base case: First element is added to the compressed list)

(c) $f(2) = f(1) = \langle a\rangle$ (Skipping the subsequent redundant element)

(d) $f(3) = f(2) \cup \langle b\rangle = \langle a\rangle \cup \langle b\rangle = \langle a, b\rangle$ (Adding the non-redundant element)

(e) $f(4) = f(3) = \langle a, b\rangle$ (Skipping the subsequent redundant element)

(f) $f(5) = f(4) \cup \langle c\rangle = \langle a, b\rangle \cup \langle c\rangle = \langle a, b, c\rangle$ (Adding the non-redundant element)

(g) $f(6) = f(5) \cup \langle a\rangle = \langle a, b, c\rangle \cup \langle a\rangle = \langle a, b, c, a\rangle$ (Adding the last non-redundant element)

Alternatively, we can unfold the computable definition of the function as well,

(a) Initial List: $\langle a, a, b, b, c, a\rangle$

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

4. (3 pts) (**PROGRAMMING** Map your definition into a Common LISP function.

```
(defun compress (lst)
  (if (null lst) ; Base case: empty list
      nil
      (let ((rest (compress (cdr lst)))) ; Recursive call on the rest of the list
        (if (or (null rest) ; If the rest is empty or the current element is different
                (not (equal (car lst) (car rest))))
            (cons (car lst) rest) ; Include the current element in the result
            rest)))) ; Skip the current element and continue compressing the rest

; Test cases
(format t "Compressed list: ~s~%" (compress '())) ; Empty list
(format t "Compressed list: ~s~%" (compress '(1 1 2 2 3 3))) ; List with duplicates
(format t "Compressed list: ~s~%" (compress '(a a b c c c d d))) ; List with duplicates and different types
(format t "Compressed list: ~s~%" (compress '(1 2 3 4 5))) ; List with no duplicates
```

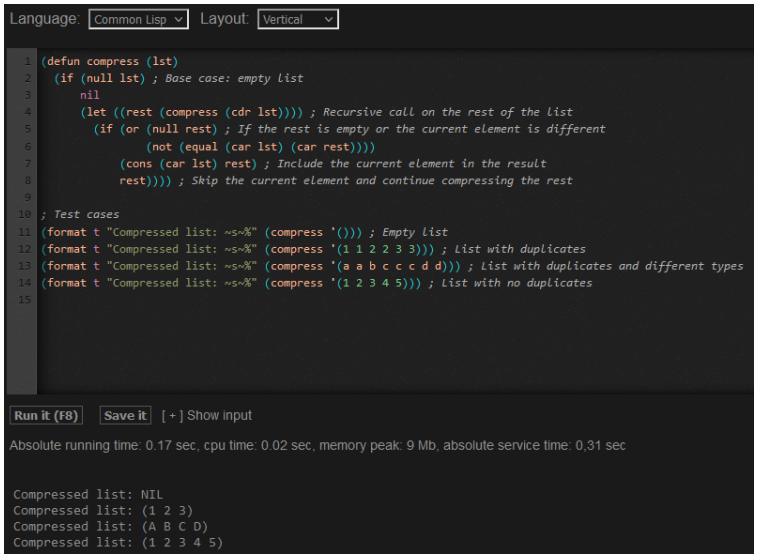


Figure 12: Interaction for compress Function