Software **A**nalysis and **F**ormal r**E**asoning

ThanhVu (Vu) Nguyen

March 18, 2025 (latest version available on nguyenthanhvuh.github.io/class-oo/safe.pdf)

Preface

Contents

Part I

Abstraction and Specification

In this part of the book, we will focus on program abstraction via specifications. Specifications allow developers to describe the behavior of a program without revealing its implementation details, thus making the program easier to understand and maintain and also more general, i.e., can use different algorithms or data structures.

We will start with *procedural abstraction* (§1), which is on the specification of functions and methods. We will then move on to *data abstraction* (§2), which is on the specification of abstract data types (ADTs) that encapsulate data and operations on that data.

Chapter 1

Procedural Abstraction and Specification

Abstraction is a key concept in software development that allows programmers to hide the implementation details and focus on the essential features. By decoupling the **what** (the behavior specification) from the **how** (the actual implementation), programmers could focus on higher-level design and reuse code more effectively.

Procedural abstraction hides implementation details from program functions or procedures. This makes the function more general and reusable, and the program easier to understand and maintain. The most common kind of abstraction that is familiar to all programmers is *parameterization*, which allows a function to take in different input values. For example, the `cal_area` function in Fig. 1.1 calculates the area of a rectangle given its length and width, which are passed as parameters.

```
def cal_area(length: int , width:int ) -> int;
    return length * width

# can be used with different values for length and width.
area1 = cal_area(5, 10)
area2 = cal_area(7, 3)
```

Fig. 1.1: Example: Abstraction by Parameterization

Another type of abstraction is through *specification*, which is less common but is crucial for creating and maintaining high-quality software (e.g., for API documentation). Specification defines what the function does (e.g., sorting), instead of how it does it (e.g., using quicksort or mergsort algorithms, implemented in C++). For example, from its specification in the comment, we know that the `exists` function in Fig. 1.2 returns true if the `target` item is found in a list of sorted `items`. We do not need to know or care about the search algorithm used.

By defining a function's behavior through specifications, developers can implement the function in different ways as long as it fulfills the specifications. Similarly, the user can use the function without knowing the implementation details. In this

```
def exists(items:List[int], target:int) -> bool:
    """
    Find an item in a list of sorted items.

    Requires: List of sorted items
    Effects: Returns True if the target is found, False otherwise.
    """
    ...
```

Fig. 1.2: Abstraction by Specification

chapter, we will focus on procedural specification and how construct and formalize specifications for functions and methods[1].

1.1 Specifications

The description of a function is captured through its *header* and *specification*. The header gives the signature of the function, including its name, parameters, and return type. The specification describes the function's behavior, including its preconditions and postconditions.

Header Function header, also called the *signature* of a function, is the first line of the function definition. It provides the *name* of the function, the number, order, and types of its *parameters* (inputs), and the type of its return value (output). For instance, the headers for functions `cal_area` in Fig. 1.1 and `exist` in Fig. 1.2 are:

```
def exists(items: list) -> bool: ...
def calc_area(length: float, width: float) -> float: ...
```

> **Type Hinting**: Python does not require specifying the types of parameters and return values. However, in more recent versions of Python, you can use *type hinting* to specify the types of parameters and return values as shown above. While type hinting is not enforced by Python, it is useful for documentation and code readability. We will use type hinting in this book to make the code more readable.

Note that in a language like Java, the header can also indicate *exceptions* that the method may throw, e.g.,

```
public boolean exists(List<Integer> items) throws Exception { ... }
```

However, function header alone is not sufficient to describe the behavior of a function. We need to provide a more detailed description of the function's behavior, which is done through the *specification*.

[1] We use the terms *function*, *method*, and *procedures* interchangeably.

1.1.1 Specification

The specification of a function defines the behavior of the function. It includes: *preconditions* (also called the "Requires" clause) and *postconditions* (also called the "Effects" clause). Preconditions describe the conditions that must be true before the function is called. Often these are constraints or assumptions about the input parameters. If there are no preconditions, the clause is often written as `None`.

Postconditions, under the assumption that the preconditions are satisfied, describe the conditions that will be true after the function is called. Postconditions state the expected results or outcomes of the function. Moreover, they often describe the relationship between the inputs and outputs.

```
def calc_area(length: float, width: float) -> float:
    """
    Calculates the area of a rectangle given its length and width.

    Reqires: None
    Effects: The area of the rectangle which is the product of the length and width.
    """
    ...
```

The clauses are usually written as *comments* above the function definition. For example, the specification of the `calc_area` function in Fig. 1.1 has (i) no preconditions and (ii) the postcondition that the function returns the area of a rectangle given its length and width. Similarly, the `exists` function in Fig. 1.2 has the specification that given a list of sorted items (precondition), it returns true if the item is found in the list, and false otherwise (postcondition). Notice how the specification is written in plain English, making it easy to understand for both developers and users of the function.

1.1.2 Modifies

Another clause that might appear in a function specification is *modifies*, which describes variables that the function can change. Often, these would be the input parameters, e.g., the function can modify the input list or data structure passed to it. However, this could also be other variables such as global variables like counters or flags. A main use of the *modifies* clause is to reveal and avoid *side effects*.

For example, the `add_to_list` function below modifies the input list and the `dirty_bit` flag.

```
dirty_bit = False
def add_to_list(input_list: List[int], value: int) -> None:
    """
    Adds a value to the input list.

    Requires: None
    Effects: Value is added to the input list.
    Modifies: the input list, dirty_bit
    """
    ...
```

1.1.3 Invariants and Assertions

Invariants A related concept to pre- and postconditions is *invariants*, which are conditions that must be true at all times during the execution of the function. For example, the `calc_area` function in Fig. 1.1 can have an invariant that the length and width are always positive.

Common invariants include *loop invariants*, which are conditions that must be true at the beginning and end of each iteration of a loop (§C), and *representation invariants* (rep invariants), which are conditions that must be true for the internal state of a class (§2). An example of repr invariants for a class representing a binary search tree is that the left child is less than the parent and the right child is greater than the parent.

> **Invariants Examples** Examples of loop invariants include a **bubble sort** implementation might have an invariant that *after each complete pass through the array, the last k elements are in their final sorted positions.* This shows that with every iteration, the largest unsorted element "bubbles up" to its correct position. For binary search, a loop invariant could be that *if the target value is in the list, it is in the range of the left and right indices.* This shows that after each iteration (which changes the left and right indices), the target value is still in the list.
>
> For repr invariants, a class representing a binary search tree (BST) might have an invariant that the left child is less than the parent and the right child is greater than the parent. This ensures that the BST is correctly structured and is not affected by any operations on the tree.

Unlike specifications, which should not reveal implementation details, invariants can involve implementation details. For example the condition $x == 0$ is True in line 7 of the file `safe.c` or that the array used to represent the binary search tree needs to have a certain property. Moreover, invariants are often checked during program execution, i.e., runtime checking, as assertions described next.

Assertions Assertions are a common way to check at the runtime constraints (code `expressions`) that should be true at a certain point in the program. Unlike pre and postconditions and invariants that are written as comments, assertions are often written as code and are checked during program execution. For example, the `calc_area` function in Fig. 1.1 can have several assertions as follows:

```python
def calc_area(length: float, width: float) -> float:
    """
    Requires: ...
    Effects: ...
    """
    ...
    assert length > 0 and width > 0  # check preconditions

    # calculate the area and store it in res
    ...
```

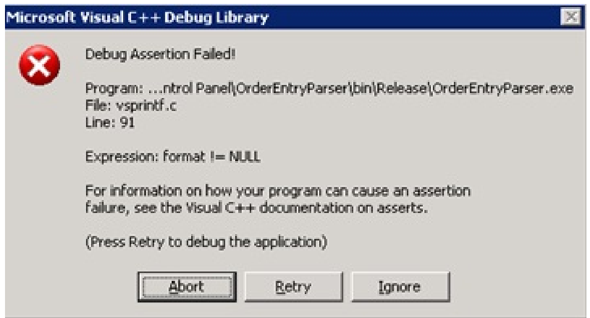

Fig. 1.3: Assertion violation. The expression `format!=NULL` fails on line 91 of the file `vsprintf.c`.

Fig. 1.4: The expression

```
assert res > 0
return res
```

Assertions can be used to check the preconditions, postconditions, invariants, or any other conditions that the programmer believes should be true. If an assertion is false, an exception is raised, indicating a bug in the program (e.g., see Fig. 1.3). Thus, assertions are useful for debugging and testing the program.

1.1.4 API

1.2 Designing Specifications

To have well-designed and effective specifications, it is important to consider several factors. These include the *strength* of the pre- and post-conditions, whether the function is *total* or *partial*, and *avoiding implementation details* in the specification.

1.2.1 Weakest Pre-conditions

For pre-conditions, we want as weak a constraint as possible to make the function more versatile, allowing it to handle a larger class of inputs. Logically, a condition x is weaker than another if it is *implied* by the other y, i.e., $y \Rightarrow x$, or that x's constraints are a superset of y's. For example, the condition $x \leq 10$ is weaker than $x \leq 5$ (because everything that is less than 5 is also less than 10). Thus a function that works with $x \leq 10$ is better than one that works with $x \leq 5$ because it can handle more inputs.

As another example the input list is not sorted is weaker than the list is sorted (which is weaker than the list that is both sorted and has no duplicates). Thus a function that can handle any list is better than one that can only handle sorted lists.

The *weakest* precondition is *True*, which indicates no constraints on the input.

1.2.2 Strongest Post-conditions

In contrast, for post-conditions, we want as strong a condition as possible to ensure that the function behaves as expected. A condition y is stronger than another condition x if y implies x, i.e., $y \Rightarrow x$, or that y's constraints are a strict subset of x's. For example, the condition $x \leq 5$ is stronger than $x \leq 10$ (because everything that is less than 5 is also less than 10) or that the input list is sorted is stronger than the list is not sorted. The reason stronger is better for postcondition is that it describes the expected behavior more precisely, e.g., a person who's under 5 feet is more precise than a person who's under 10 feet.

1.2.3 Total vs Partial Functions

A function is *total* if it is defined for all legal inputs; otherwise, it is *partial*. Thus a function with no precondition (weakest precondition) is total.

Total functions are *preferred* because they can be used in more situations, especially when the function is used publicly or in a library where the user may not know the input constraints. Partial functions can be used when the function is used internally, e.g., a helper or auxiliary function and the caller is knowledgeable and can ensure its preconditions are satisfied.

For example, functions `calc_area` in Fig. 1.1 and `add_to_list` in Fig. 1.2 are total because they can be called with any input. The `exists` function in Fig. 1.2 is partial because it only accepts sorted lists.

Turning Partial Functions into Total Functions We can often turn a partial function into a total function in two steps.

1. Move preconditions into postconditions and (in the postconditions) specify the expected behavior when the precondition is not satisfied, e.g., when invalid input `X` occurs, throws an exception `Y`

2. Modify the function to satisfy the new specification (specifically, to satisfy the new postcondition). In other words, add code to handle the cases when the preconditions are not satisfied.

For example, the `exists` function in Fig. 1.2 is turned into the total function shown in Fig. 1.5.

1.2.4 Avoid Implementation Details in Specifications

A specification should not include any implementation details, such as the algorithm used or the data structures employed. This improves flexibility as it allows the function to be implemented in different ways as long as it satisfies the specification. For example, the `exists` function in Fig. 1.2 does not specify the search algorithm used to find the item in the list.

```
def exists(items: List[int], target: int) -> bool:
    """
    Find an item in a list of sorted items.

    Pre: True
    Post: If the input items are not sorted, raise an exception.
          Return True if the item is found, False otherwise.


    """

    if not is_sorted(items):
        raise Exception(...)
```

Fig. 1.5: Total Specification for the program in Fig. 1.2

Some common examples to avoid include: the mentioning of specific data structures (e.g., arrays, indices), algorithms (e.g., quicksort or mergesort), and exceptions (e.g., related to IndexError). Also avoid specifications mentioning indices because this implies the use of arrays.

1.3 Under- vs Over-Specifications

From what we have seen, we prefer specifications that balance both precision (postcondition) and flexibility (e.g., precondition, no implementation details). We want to avoid over and under-specifications.

1.3.1 Over-Specification

An over-specified specification includes unnecessary constraints or implementation details that restrict the function's flexibility. For example, precondition that the input must be a list instead of iterable collections or include the use of specific algorithms or data structures. Over-specification makes the function less flesible and code less reusable.

Example: sorting function Taking as input a list of integers (precondition) and returning a list that is a permutation of the input list in non-decreasing order using quicksort (postcondition).

This is over-specified because it restricts the data structure to list and the sorting choice to quicksort (which has a worst case of $O(n^2)$). A better spc would be take as input a collection of comparable elements and return a collection that is a permutation of the input collection in non-decreasing order.

1.3.2 Under-Specification

An under-specified specification does not define the expected behavior of the function precisely. For example, failing to specify potential outputs or edge cases. Under-specification can cause incorrect implementation.

Example: Search for a number from a sorted list Taking as input a list of numbers and a target number and returning the location of the target number in the list. This is under-specified because it doesn't specify what happens if the target is not found or if there are duplicates. A better specification would be to return the location of the target if it is found, raise an exception if the target is not found, and return the first location if there are duplicates.

1.4 More Examples of Bad Specifications

- The function `find_max` takes a list of elements (precondition) and returns the maximum element by iterating over the indices from 0 to n-1 (postcondition).
- The function `sort_list` sorts a list of numbers (precondition) and uses quick-sort to sort the list (postcondition).
- The function `exists` returns true if the target is found or raise IndexError if the target is not found (postcondition)

1.5 Exercise

1.5.1 Specification for Sorting

Write the specification for the generic `ascending_sort` method below. The specification should include preconditions and postconditions.

```
def ascending_sort(my_list):
    # REQUIRES/PRE:
    # EFFECTS/POST:
    ...
```

1.5.2 Specification for Merging Two Sorted List of Numbers

Write the spec. for the function `merge_sorted_lists` that takes two sorted lists of (int) numbers and returns a single list in non-decreasing order that contains all the elements of the two input lists. The header of the function is given below.

```
def merge_sorted_lists(lst1: List[int], lst2: List[int]) -> List[int]:
    """
    PRE/REQUIRES:
    POST/EFFECTS:
    """
    ...
```

1.5.3 Specification of Binary Search

Come up with the specification for a *binary search* implementation whose header is given below. Remember for precondition you want something as *weak* as possible and for postcondition as *strong* as possible. Note that binary search returns the *location* (an non-neg integer) of the `target` value if found, and returns -1 if `target` is not found.

```
def binary_search(arr: List[int], target: int) -> int:
    """
    PRE/REQUIRES:
    POST/EFFECTS:
    """
    ...
```

1.5.4 Loan Calculator

Consider a function that calculates the number of months needed to pay off a loan of a given size at a fixed *annual* interest rate and a fixed *monthly* payment. For example, a $100,000 loan at an 8% annual rate would take 166 months to discharge at a monthly payment of $1,000, and 141 months to discharge at a monthly payment of $1,100. (In both cases, the final payment is smaller than the others; we round 165.34 up to 166 and 140.20 up to 141.) Continuing the example, the loan would never be paid off at a monthly payment of $100, since the principal would grow rather than shrink.

- Define a function satisfying the following specification:

```
def months(principal: int, rate: float, payment: int) -> int:
    """
    Calculate the number of months required to pay off a loan.

    param principal: Amount of the initial principal (in dollars)
    param rate: Annual interest rate (e.g., 0.08 for 8%)
    param payment: Amount of the monthly payment (in dollars)

    Requires/Pre: principal, rate, and payment all positive and
    payment is sufficiently large to drive the principal to zero.
    Effects/Post: return the number of months required to pay off the principal
    """
```

  - The precondition is quite strong, which makes implementing the method easy. The key step in your calculation is to change the principal on each iteration with the following formula (which amounts to monthly compounding):

```
new_principal = old_principal * (1 + monthly_interest_rate) - payment
```

  - To make sure you understand the point about preconditions, your code is required to be *minimal*. Specifically, if it is possible to delete parts of your implementation and still have it satisfy the requirements, you'll earn less than full credit.

- *Total* specification: Now change the specification to *total* in which the post-condition handles violations of the preconditions using *exceptions*. In addition, provide a new implementation `month` that satisfies the new specification.

1.5.5 Partial and Total Specifications for `tail`

Consider the following code:

```
def tail(my_list):
     result = my_list.copy()
     result.pop(0)
     return result
```

- What does the implementation of `tail` do in each of the following cases? You might want to see the Python document for `pop`. How do you know: Running the code or reading Python document?
  - `list = None`
  - `list = []`
  - `list = [1]`
  - `list = [1, 2, 3]`
- Write a *partial specification* that satisfies the given `tail` implementation
- Rewrite the specification to be *total*. Use *exceptions* as needed.

1.5.6 Partial and Total Functions

1. Write the *partial* specifications for the below two functions.
2. Modify the specifications to make the functions *total*.
3. Modify the *implementations* of the two functions to satisfy the total specifications.

Recall that specifications do not deal with types (which are specified by function signatures and enforced by the type system of compiler/interpreter). In other words, you do not need to worry about types here and can assume conditions about types are satisfied.

```
def divide(a:float, b:float) -> float:
    """
    PRE:
    POST:
    """
```

```python
class User:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        if not isinstance(other, User):
            return False
        return self.name == other.name
```

Fig. 1.6: User class

```python
    return a / b

def get_average(numbers: list[float]) -> float:
    """
    PRE:
    POST:
    """
    total = sum(numbers)
    return divide(total, len(numbers))
```

1.5.7 Problems with Inheritance in OOP: Equality

Inheritance, which allows a class to inherit properties and methods from another class, is a key feature of OOP. While this has many benefits, inheritance introduces certain unexpected kinds of issues for specifying desired properties and implementing them. This exercise will show you one of using the popular *equals* method in OOP.

1. First, look at the Javadoc to understand the behaviors `equals()` (while the specification is for Java, the idea is the same in Python).
   - Specifically, read carefully the *symmetric*, *reflexive*, and *transitive* properties of `equals()`.
   - Ignore *consistency*, which requires that if two objects are equal, they remain equal.
2. For the `User` class in Fig. 1.6, does `equals()` satisfy the three equivalence relation properties? If not, what is the problem?
   - Come up with several concrete test cases (e.g., create various `User` instances) to check the properties.
   - If there is a problem, show the test case that demonstrates the problem.
   - Explain why the problem occurs and come up with a fix.
3. Do the same analysis for the `SpecialUser` class in Fig. 1.7.

```python
class SpecialUser(User):
    """Don't do this until you've done with User"""

    def __init__(self, name, id):
        super().__init__(name)
        self.id = id

    def __eq__(self, other):
        if not isinstance(other, SpecialUser):
            return False
        return super().__eq__(other) and self.id == other.id
```

Fig. 1.7: SpecialUser class

Chapter 2

Abstract Data Type

In 1974, Barbara Liskov and Stephen N. Zilles introduced Abstract Data Types (*ADTs*) in their influential paper "Programming with Abstract Data Types" as part of their work on the CLU programming language at MIT. ADTs changed software design by separating the specification of a data type from its implementation. This allows developers to define operations on a data structure without exposing the detailed implementation of data (e.g., calling `pop` to remove data from a `Stack` without knowing the internal details on how stack stores data).

For her pioneering contributions to programming languages and system design, particularly on ADTs and CLU, Barbara Liskov was awarded the Turing Award in 2008. Today, ADTs are a cornerstone of all modern programming languages.

2.1 Specifications of an ADT

The *specification of an ADT* describe the operations and behaviors of the object, allowing users to interact with it only via methods, rather than accessing its internal representation. As with functions (§1), the specification for an ADT defines its behaviors without being tied to a specific implementation (e.g., the internal data of an ADT).

Structure of an ADT In a modern OOP language such as Python or Java, data abstractions are defined using *classes*. Each class defines a name for the data type, along with its constructors and methods.

Fig. 2.1 shows an ADT class template in Python. It consists of three main parts. The *Overview* describes the abstract data type in terms of well-understood concepts, like mathematical models or real-world entities. For example, a stack could be described using mathematical sequences. The Overview can also indicate whether the objects of this type are *mutable* (their state can change) or *immutable*. The *Constructor* initializes a new object, setting up any initial state required for the instance. Finally, *methods* define operations users can perform on the objects.

```
class DataType:
    """
    Overview: A brief description of the data type and its objects.
    """

    def __init__(self, ...):
        """
        Constructor to initialize a new object.
        """

    def method1(self, ...):
        """
        Method to perform an operation on the object.
        """
```

Fig. 2.1: Abstract Data Type template

These methods allow users to interact with the object without needing to know its internal representation. In Python, `self` is used to refer to the object itself, similar to `this` in Java or C++.

Note that as with procedural specification (§1), the specifications of constructors and methods of an ADT do not include implementation details. They only describe what the operation does, not how it is done. Moreover, they are written in plain English as code comment.

2.1.1 Example: `IntSet` ADT

Fig. 2.2 gives the specification for an `IntSet` ADT, which represents unbounded set of integers. `IntSet` includes a constructor to initialize an empty set, and methods to insert, remove, check membership, get the size, and choose an element from the set. `IntSet` is also mutable, as it allows elements to be added or removed. *mutator* `insert` and `remmove` are mutator methods and have a `MODIFIES` clause. In contrast, `is_in`, `size`, and `choose` are *observer* methods that do not modify the object.

2.2 Implementing ADT

To implement an ADT, we first choose a *representation* (**rep**) for its objects, then design constructors to initialize it correctly, and methods to interact with and modify the rep. For example, we can use a `list` (or vector) as the rep of `IntSet` in Fig. 2.2. We could use other data structures, such as a `set` or `dict`, as the rep, but a list is a simple choice for demonstration.

To understanding and reasoning of the rep of an ADT, we use two key concepts: *representation invariant* and *abstraction function*.

```python
class IntSet:
    """
    Overview: IntSets are unbounded, mutable sets of integers.
    This implementation uses a list to store the elements, ensuring no duplicates.
    """

    def __init__(self):
        """
        Constructor
        EFFECTS: Initializes this to be an empty set.
        """
        self.els = []  # the representation (list)

    def insert(self, x: int) -> None:
        """
        MODIFIES: self
        EFFECTS: Adds x to the elements of this set if not already present.
        """
        if not self.is_in(x): self.els.append(x)

    def remove(self, x: int) -> int:
        """
        MODIFIES: self
        EFFECTS: Removes x from this set if it exists. Also returns
        the index of x in the list.
        """
        i = self.find_idx(x)
        if i != -1:
            # Remove the element at index i
            self.els = self.els[:i] + self.els[i+1:]
        return i

    def is_in(self, x: int) -> (bool, int):
        """
        EFFECTS: If x is in this set, return True. Otherwise False.
        """
        return True if find_index(x) != -1 else False

    def find_idx(self, x:int)->int:
        """
        EFFECTS: If x is in this set, return its index. Otherwise returns -1.
        """
        for i, element in enumerate(self.els):
            if x == element:
                return i
        return -1

    def size(self) -> int:
        """
        EFFECTS: Returns the number of elements in this set (its cardinality).
        """
        return len(self.els)

    def choose(self) -> int:
        """
        EFFECTS: If this set is empty, raises an Exception.
        Otherwise, returns an arbitrary element of this set.
        """
        if len(self.els) == 0:
            raise Exception(...)
        return self.els[-1]  # Returns the last element arbitrarily

    def __str__(self) -> str:
        """
        Abstract function (AF) that returns a string representation of this set.
        EFFECTS: Returns a string representation of this set.
        """
        return str(self.els)
```

Fig. 2.2: The IntSet ADT

2.2.1 Representation Invariant (Rep-Inv)

Because the rep data might not be related to the ADT itself (e.g., the list has different properties compared to a set), we need to ensure that our use of the rep is consistent with the ADT's behavior. To do this, we use *representation invariant* (**rep-inv**) to specify the constraints for the rep to capture the behavior of the ADT.

For example, the rep-inv for a *stack* is that the last element added is the first to be removed and the rep-inv for a *binary search tree* (BST) is that the left child is less than the parent, and the right child is greater. The rep-inv for our `IntSet` ADT in Fig. 2.2 is that all elements in the list are unique integers.

```
# Rep-inv:
# els is not null, only contains integers and has no duplicates.
```

Rep-invs must be preserved by all methods (more precisely, *mutator* methods). It must hold true before and after the method is called. The rep-inv might be violated temporarily during the method execution, but it must be restored before the method returns. For `IntSet` Notice that the mutator `insert` method ensures that the element is not already in the list before adding it.

`repOK()` method The rep-inv is decided by the designer and specified in the ADT documentation as part of the specification (just like pre/post conditions) so that it is ensured at the end of each method (like the postcondition). Moreover, because rep-inv is so important, it is not only documented in comments like specification but should be checked at runtime. This is done by defining a bool `repOK` to check the rep-inv, and invoking `repOK` at the start and end of each method (more specifically, methods that modify the rep).

For example, for the `IntSet` ADT, we define a `repOK` method as follows:

```
def repOK(self) -> bool:
    """
    Check rep-inv els is not null, only contains integers and has no duplicates.
    """
    if self.els is None:
        raise Exception("Rep-inv violated: elements are None.")
    if not all(isinstance(x, int) for x in self.els):
        raise Exception("Rep-inv violated: elements are not integers.")
    if len(self.els) != len(set(self.els)):
        raise Exception("Rep-inv violated: duplicates in elements.")
```

Now we can invoke `repOK` at the start and end of each method to ensure that the rep-inv is maintained. For example,

```
def insert(self, x: int) -> None:
    """
    MODIFIES: self
    EFFECTS: Adds x to the elements of this set if not already present.
    """
    self.repOK()
    if not self.is_in(x): self.els.append(x)
    self.repOK()
```

2.2.2 In-Class Exercise: Checking Rep-Invs

```python
class Members:
    """
    Overview: Members is a mutable record of organization membership.
    AF: Collect the list as a set.

    Rep-Inv:
        - rep-inv1: members != None
        - rep-inv2: members != None and no duplicates in members.
        For simplicity, assume None can be a member.
    """

    def __init__(self):
        """Constructor: Initializes the membership list."""
        self.members = []  # The representation

    def join(self, person):
        """
        MODIFIES: self
        EFFECTS: Adds a person to the membership list.
        """
        self.members.append(person)

    def leave(self, person):
        """
        MODIFIES: self
        EFFECTS: Removes a person from the membership list.
        """
        self.members.remove(person)
```

1. Analyze these four questions for *rep-inv 1*.
    - Does `join()` maintain rep-inv?
    - Does `join()` satisfy its specification?
    - Does `leave()` maintain rep-inv?
    - Does `leave()` satisfy its specification?
2. Repeat for *rep-inv 2*.
3. Recode `join()` to make the verification go through. Which rep-invariant do you use?
4. Recode `leave()` to make the verification go through. Which rep-invariant do you use?

2.2.3 Abstraction Function (AF)

It can be difficult to understand the ADT by looking at its representation data directly. For example, we might not be see a binary tree or a graph ADT that uses lists or vectors as rep data, or a telephone number from a string (e.g., ‘‘11234567890’’).

To aid understanding, *abstraction function* (**AF**) provides a mapping between the rep and the ADT. Specifically, an AF maps from a *concrete state* (i.e., the `els` rep in Fig. 2.2 or ``11234567890'' to an *abstract state* (i.e., the set of integers or a telephone number 1-123-456-7890).

Another common property of AF is that it is a *many-to-one* mapping. This allows multiple concrete states map to the same abstract state, e.g., the list `[1, 2, 3]` and `[3, 2, 1]` both map to the same set `{1, 2, 3}`.

`__str__()` method Just as with rep-inv (§2.2.1), the AF is documented in the class specification and also through implementation, typically in a method that returns a *string representation* of the object (the rep). In modern OOP languages, AF methods are often implemented by overriding `__str__` in Python or `toString` in Java. For example, the `__str__` method in Fig. 2.2 returns a string representation of the set. Another example is shown below that shows the telephone number as a string.

```
class PhoneNumber:
    def __str__(self): -> str:
        """
        AF that returns a properly formatted phone number.

        Assuming that the rep is a string of 10 digits (e.g., "11234567890").
        EFFECTS: Returns the phone number in the format 1-123-456-7890.
        """
        return f"{self.rep[:1]}-{self.rep[1:4]}-{self.rep[4:7]}-{self.rep[7:]}"
```

2.3 Algebraic Specifications

Algebraic specifications provide a formal way to define ADTs using *equations* and *axioms*, rather than procedural descriptions, i.e., how the ADT behaves rather than how it is implemented. This allows for precise reasoning about the behavior of ADTs *independently of implementation.*

An algebraic spec. consists of three main components:

1. Sorts (Types): Define the ADT (e.g., `IntSet`, `Stack`, `Queue`).

2. Operations: Defines the methods (e.g., `push`, `pop`, `enqueue`, `insert`).

3. *Equational Axioms*: Defines rules that specify behavior using equations, (e.g., `top(push(x, stack)) = x`).

The first two components, sorts and operations, are similar to normal ADT specification. However, the axioms are unique to algebraic specifications. These axioms describe the interactions between operations using equations. For example, for a Stack ADT, the axiom `top(push(x, stack)) = x`, gives a relationship between `push` and `top` operations.

Example A *stack*, which follows Last-In, First-Out (LIFO) behavior, can be specified algebraically as follows:

- Sort: Stack
- Operations: `push`, `pop`, `top`, `is_empty`, `is_in`
- Axioms (notations: x is some element, s is some stack, $\emptyset$ is an empty stack):
    1. $top(push(x, S)) = x$
    2. $pop(push(x, S)) = S$
    3. $is\_empty(push(x, S)) = False$
    4. $is\_empty(\emptyset) = True$
    5. $x = y \;\Rightarrow\; is\_in(x, push(y, S)) = True$
    6. $x \neq y \;\Rightarrow\; is\_in(x, push(y, S)) = is\_in(x, S)$
    7. ...

These axioms specify the behavior of the stack ADT through its operations. This is not a complete list of axioms, but is sufficient to illustrate the concept. Notice the last two axioms use conditional equations (i.e., involving implications), which are useful for specifying behavior based on different cases.

Benefits Using algebraic specifications to define ADT has several benefits. First, it provides *mathematical precision*, avoiding ambiguities in natural language descriptions. Second, it is *implementation independent* and focuses on what an operation should do rather than how it is implemented. Finally, it allows for *formal reasoning* about ADTs, enabling formal proofs of correctness.

2.3.1 Algebraic Axioms vs. Rep-Invs

Tab. 2.1: Comparison Between Algebraic Axioms and Representation Invariants

Feature	**Algebraic Axioms**	**Rep Invs**
Define	Eqts laws specifying ADT	Properties of a valid ADT rep
Scope	Ext. spec (what the ADT does)	Internal Constraints (valid internal states)
Focus	Functional correctness	Internal consistency
Implementation	Implementation Agnostic	Depend on chosen rep
Example (Set)	$contains(x, insert(x, S)) = \text{True}$	Ensuring no duplicate elements exist.

Tab. 2.1 shows the differences between algebraic axioms and rep-invs. We use algebraic axioms to specify an ADT before implementing it, and use rep invariants to check the implementation (specifically the internal rep). They complement each other, with axioms focusing on the behavior of the ADT and rep-invs focusing on the internal correctness of the rep.

2.3.2 Testing Algebraic Specifications

We can use algebraic specifications, especially the axioms, to check the ADT implementation. If the implementation does not satisfy the axioms, then it has a bug.

For example, to check the algebraic axioms of the `IntSet` implementation in Fig. 2.2, we can write some tests as follows:

```
def test_axioms():
    s = IntSet()
    x = 1
    assert(s.insert(x).is_in(x) == True)
    assert(s.remove(x).is_in(x) == False)

    size_orig = s.size()
    assert(s.insert(x).size() == size_orig + 1)
```

Notice we use `size_orig` to store the size of the set before inserting an element. This is because the implementation has side effects, and the size of the set changes after inserting an element.

2.4 Mutability vs. Immutability

An ADT can be either mutable or immutable, depending on whether the states of their object instances can change after creation. An ADT should be immutable if it models objects that remain constant once created. For example, mathematical objects like integers, polynomials (Polys), or complex numbers are typically immutable, as their values do not change once set. Similarly, data structures like tuples or strings are immutable.

On the other hand, an ADT should be mutable if it models things that can change over time. For example, an ADT representing a bank account would be mutable, as the account balance changes with deposits and withdrawals. Similarly, data structures like arrays or lists are typically mutable, allowing for dynamic updates and modifications.

Immutability is beneficial because it offers greater safety and allows sharing of subparts without the risk of unexpected changes. Moreover, immutability can simplify the design by ensuring the object's state is fixed once created. However, immutable objects can be less efficient, as creating a new object for each change can be costly in terms of memory and time.

2.4.1 Converting from mutable to immutable

Given a mutable ADT, it is possible to convert it to an immutable one by ensuring that the rep is not modified by any method. This can be achieved by making the rep private and only allowing read-only access to it. In Python, this can be done by using the `@property` decorator to create read-only properties. For example, the `els` list in Fig. 2.2 can be made read-only by defining a property method `elements` that returns a copy of the list.

```
class IntSet:
    def __init__(self):
        self.__els = []    # Private rep
    @property
    def self.els(self):
        return self.__els
```

Moreover, we need to convert mutator methods into observer methods, which make a copy of the rep, modify it, and return the modified rep object.

```
def insert_immutable(self, x: int) -> IntSet:
    new_set = self.els.copy()
    if not self.is_in(x):
        new_set = new_set.append(x)
    return new_set
```

If the mutator returns a value v, then our new method returns a tuple consisting of (i) the new rep object and the return the value v.

```
def remove_immutable(self, x: int): -> (IntSet, int):
    i = self.find_idx(x)
    new_set = self.els.copy()
    if i != -1:
        # Remove the element at index i
        new_set = self.els[:i] + self.els[i+1:]
    return (new_set, i)
```

If you do not want to return multiple values (e.g., like in Java), then you can create two methods, one for returning the value and the other for returning the new rep object. For example, a mutator `pop` method of a `Stack` would result into two methods: `pop2` returns the top element and `pop3` returns the new stack with the top element removed.

Finally, it is important that while it is possible to convert a mutable ADT to an immutable one as shown, mutability or immutability should be the property of the ADT type itself, not its implementation. That is, the decision to make an ADT mutable or immutable should be made at the design stage and documented in the ADT specification.

2.5 Exercise

2.5.1 Stack ADT

In this exercise, you will implement a `Stack` ADT. A stack is a common data structure that follows the Last-In-First-Out (LIFO) principle. You will:

1. Choose a Representation (rep) to represent your stack
2. What would be the *representation invariant* (rep-inv) for this rep?
3. Implement the rep-inv in a `repOK` method.
4. Provide the specifications of basic stack operations (`push`, `pop`, `is_empty`) and implement these methods accordingly (pseudo code is fine).

5. What would be the abstraction function (AF) for this ADT?
6. Implement the AF in a `__str__()` method (returns a string representation of the stack based on the AF)

2.5.2 Polynomial ADT

Use the Poly ADT in Fig. 2.3 to answer the following questions. Use the `Stack` ADT in Fig. D.1 as an example.

1. Part 1
   (a) Write an Overview that describes what `Poly` does. You must provide some examples to demonstrate (e.g., `Poly(2,3)` means what?).
   (b) Provide the specifications for all methods in the ADT.
   (c) Write the **rep** used in this code. Describe how this rep represents `Poly`.
   (d) Provide the **rep-inv** for the ADT. Note, this would be the constraints over the rep variable(s).
   (e) Write a `repOK` method that checks the rep-inv.
   (f) Describe the AF in this code. Use `__str__`.
2. Part 2
   (a) Introduce a fault (i.e. "bug") that breaks the **rep-inv**. Try to do this with a small (conceptual) change to the code. Show that the rep-invariant is broken with a concrete test case.
   (b) Analyzed your bug with respect to the method specifications of `Poly`. Are all/some/none of the specification violated?
   (c) Do you think your fault is realistic? Why or why not?

2.5.3 Algebraic Axioms for IntSet

Define the algebraic axioms for the `IntSet` ADT in Fig. 2.2. You can look at Fig. 2.2 to get ideas on how these operations work (e.g., `find_idx` returns an index of a given element). **Do not** include any implementation details in your algebraic axioms. For example, do not assume that `choose()` always return the last element in the list (even though the implementation in Fig. 2.2 does that).

- Sort: `IntSet`
- Operations: `insert`, `remove`, `is_in`, `find_idx`, `size`, `choose`
- Axioms: find equalities to define the behavior of the given operations. .

```
class Poly:
    def __init__(self, c=0, n=0):
        if n < 0:
            raise ValueError("Poly(int, int) constructor: n must be >= 0")
        self.trms = {}
        if c != 0:
            self.trms[n] = c

    def degree(self):
        if len(self.trms) > 0:
            return next(reversed(self.trms.keys()))
        return 0

    def coeff(self, d):
        if d < 0:
            raise ValueError("Poly.coeff: d must be >= 0")
        return self.trms.get(d, 0)

    def sub(self, q):
        if q is None:
            raise ValueError("Poly.sub: q is None")
        return self.add(q.minus())

    def minus(self):
        result = Poly()
        for n, c in self.trms.items():
            result.trms[n] = -c
        return result

    def add(self, q):
        if q is None:
            raise ValueError("Poly.add: q is None")

        non_zero = set(self.trms.keys()).union(q.trms.keys())
        result = Poly()
        for n in non_zero:
            new_coeff = self.coeff(n) + q.coeff(n)
            if new_coeff != 0:
                result.trms[n] = new_coeff
        return result

    def mul(self, q):
        if q is None:
            raise ValueError("Poly.mul: q is None")

        result = Poly()
        for n1, c1 in self.trms.items():
            for n2, c2 in q.trms.items():
                result = result.add(Poly(c1 * c2, n1 + n2))
        return result

    def __str__(self):
        r = "Poly:"
        if len(self.trms) == 0:
            r += " 0"
        for n, c in self.trms.items():
            if c < 0:
                r += f" - {-c}x^{n}"
            else:
                r += f" + {c}x^{n}"
        return r
```

Fig. 2.3: Polynomial ADT

1. Memberships: properties of `is_in`
2. Size: properties of `size`
3. Index: properties of `find_idx`
4. Choice: properties of `choose`
5. Miscs: other interesting properties of IntSet

2.5.4 Algebraic Axioms for BankAccount

`BankAccount` is an ADT that models a real-world financial system where users can deposit, withdraw, transfer money, and check their account balance.

- Sort: `BankAccount`
- Operations: `BankAccount` has the operations
  - `balance(account)` → float: return the balance of the account.
  - `deposit(account:BankAccount, amount:float)` → BankAccount: deposit amount into the account and return the updated account.
  - `withdraw(account:BankAccount, amount:float)` → BankAccount: if there is not sufficient funds, return the original account, otherwise return the updated account.
  - `transfer(account1, account2, amount)` → (BankAccount, BankAccount): transfer amount from account1 to account2. If there is not sufficient funds, return the original accounts, otherwise return the updated accounts.

Define the algebraic axioms for these operations. You can invent shortcuts to make the axioms more readable. For example, `balance(transfer(A1, A2, amount))` means that we first `transfer amount` from `A1` to `A2` and get the updated accounts `A1'` and `A2'`, and then take the `balance` of `A1'` and the `balance` of `A2'`, i.e., `balance(transfer(A1, A2, amount)) = (balance(A1'), balance(A2'))`.

2.5.5 Dictionary ADT

A *dictionary ADT* stores key-value pairs, where each key is unique. It has the following operations:

- **put(D, k, v)** → Dictionary
  Insert a key-value pair (k, v) into dictionary D. If k already exists, update its value. Return the updated dictionary.
- **remove(D, k)** → (Dictionary, v)
  Remove key k from dictionary D and returns the updated dictionary and the value v associated with k. If k is not found, return the original dictionary.

- **get(D, k)** → v
  Return the value associated with key k. If k does not exist, return an error.

- **is_in(D, k)** → bool
  Returns `True` if and only if key k is present in D.

- **size(D)** → int
  Returns the number of key-value pairs in D.

Your tasks

- Find algebraic axioms for this dictionary ADT over the given operations.
  - Try to be as complete as possible.
  - Hilight the interesting and complex axioms and describe them.
- Implement the dictionary ADT in Python using the following *mutable* template (you will convert it to an immutable ADT later).

```python
class Dictionary:
    def __init__(self):
        # to be implemented
        # define a rep to store key-value pairs

    def repOK() -> bool:
        # to be implemented
        # check the rep-inv

    def put(self, k:int, v:float) -> None:
        # specs
        # to be implemented

    def get(self, k:int) -> float:
        # to be implemented

    def remove(self, k) -> float:
        # to be implemented

    def is_in(self, k) -> bool:
        # to be implemented

    def size(self) -> int:
        # to be implemented

    def __str__(self) -> str:
        # to be implemented
        # abstract function
```

  - Clearly *put the specifications as comments* as you have learned and done in class. Also put side effects for mutator methods.
  - Describe the rep you use to represent the ADT.
  - Describe *rep invariant(s)* you use in `repOK` as a logical formula over the rep variable.

  - Describe the *abstraction function* you use as implemented in code(__str__)

- Create an *immutable dictionary ADT* by converting the mutable version to an immutable one as learned in class (§2.4).
  - Remember that when changing mutator methods (mutable) to observer methods (immutable), you also also return a new rep of the ADT or a new ADT (your choice)-the point is so that the original rep is not modified. For example, `remove` should return a tuple of (new dictionary, return value).
  - Clearly write the specs, reps, rep-invs, af, of the new methods. Note that most are just copied over from the mutable version, e.g., reps, rep-invs, af should be unchanged.

2.5.6 Immutable Queue

Rewrite the mutable `Queue` implementation in Fig. 2.4 so that it becomes *immutable*. Keep the **rep** variables `elements` and `size`.

2.5.7 Immutability 1

The below class `Immutable` is supposed to be immutable. However, it is not. Identify the issues and fix them.

1. Which of the lines (A–F) has a problem with immutability? Explain why by showing code example, i.e., show code involving problematic lines; show how that breaks immutability.
2. For each line that has a problem. Write code to fix it so that the class is immutable.

Notes:

1. Python or Java, immutable types include `int`, `float`, `str`, `tuple`. and mutable types include `list` and `dict`.
2. In Python, you can use `copy` method to create a copy of a list and `deepcopy` for more complicated data structures like `dict`.

```
class Immutable:
    def __init__(self, mstr: str, mint: int, mlist: list[str]):
        self._mstr = mstr                       # Line A
        self._mint = mint                       # Line B
        self._mlist = mlist.copy()              # Line C


    def get_mstr(self) -> str: return self._mstr        # Line D
    def get_mint(self) -> int: return self._mint        # Line E
    def get_mlist(self) -> list[str]: return self._mlist # Line F
```

```python
class Queue:
    """
    A generic Queue implementation using a list.
    """

    def __init__(self):
        """
        Constructor
        Initializes an empty queue.
        """
        self.elements = []
        self.size = 0

    def enqueue(self, e):
        """
        MODIFIES: self
        EFFECTS: Adds element e to the end of the queue.
        """
        self.elements.append(e)
        self.size += 1

    def dequeue(self):
        """
        MODIFIES: self
        EFFECTS: Removes and returns the element at the front of the queue.
        If the queue is empty, raises an IllegalStateException.
        """
        if self.size == 0:
            raise Exception(...)

        result = self.elements.pop(0)  # Removes and returns the first element
        self.size -= 1
        return result

    def is_empty(self):
        """
        EFFECTS: Returns True if the queue is empty, False otherwise.
        """
        return self.size == 0
```

Fig. 2.4: Mutable Queue

2.5.8 Immutability 2

Do the same with the previous exercise (§2.5.7) but now with the below class `Immutable2`.

```
class Immutable2:
    def __init__(self, username: str, user_id: int, data1: list[str], data2: dict):
        self._username = username  # Line A
        self._user_id = user_id    # Line B
        self._data1 = data1        # Line C
        self._data2 = data2        # Line D

    def get_username(self) -> str: return self._username
    def get_user_id(self) -> int: return self._user_id
    def get_data1(self) -> list[str]: return self._data1  # Line E
    def get_data2(self) -> dict:  return self._data2      # Line F
```

Chapter 3

Types

A *type* specifies the set of possible values that a variable or expression can hold, and defines the operations that are valid for those values. In OOP, types are used to define ADTs through classes and interfaces. A well-designed type system can detect many errors at compile time and allow the compiler to generate optimized code.

This chapter covers key concepts in the type system of OOP languages. We will review topics like polymorphism, inheritance, dynamic dispatching, and explore Lispov's principle of substitution, which is essential for understanding how types work in OOP.

> In 1999, NASA's Mars Climate Orbiter mission ended in failure due to a simple yet catastrophic software error. The spacecraft, which costs $125 million to build and launch, was launched on December 11, 1998 to study Mars. After a 9-month journey, the spacecraft approached Mars on September 23, 1999, and was supposed to enter a stable orbit around Mars at an altitude of about 226 kilometers (140 miles) above the planet's surface. However, the spacecraft instead plunged much deeper into the Martian atmosphere, to an estimated altitude of 57 kilometers (35 miles), causing it to either burn up or crash on the surface and resulting in a complete loss of the mission.
>
> The cause of the failure was a software error involving typing mismatch between imperial units (pounds-force) and metric units (newtons) in the software that controlled the spacecraft's thrusters. The software expected data in metric units, but the thruster data was provided in imperial units, leading to the incorrect trajectory calculations. This mismatch was not caught during testing. This failure not only cost NASA a significant financial investment but also set back the Mars exploration program.

3.1 Type as Specification

Programming languages leverage typing not only to prevent errors but also to encode expected behavior properties. In a sense typing serves a lightweight form of specifi-

```python
from abc import ABC, abstractmethod

class Mammal(ABC):
    """
    Abstract class
    """

    @abstractmethod
    def make_sound(self):
        raise NotImplementedError("Subclasses should implement this!")

class Dog(Mammal):
    def make_sound(self):
        return "Woof!"

    def bark(self):
        return "Bark!"

class Cat(Mammal):
    def make_sound(self): return "Meow!"

# Using polymorphism
def mammal_make_sound(mammal: Mammal): return mammal.make_sound()

mammals = [Dog(), Cat()]
for m in mammals:
    print(mammal_make_sound(m))
```

Fig. 3.1: Polymorphism

cation that is explicitly supported by the language and checked by the compiler.

Consider the function `f(x: int, y: int) -> int`. The input types `int` are precondition requiring that the inputs are integers. The output type `int` is a postcondition ensuring that the output is an integer. A type checker would verify that `f` adheres to these type specifications and returns an error if `f`'s uage violates its specification, e.g., `f(3,4) + ‘‘hello’’` would be an error.

3.2 Fundamental OOP Concepts

3.2.1 Polymorphism

Polymorphism is a cornerstone of OOP that allows objects of different classes to be treated as objects of a common superclass. Polymorphism allows for flexibility and extensibility in the design of software systems.

Fig. 3.1 shows an example of polymorphism, where the class `Mammal` has two subclasses, `Dog` and `Cat`. Since a `Dog` and a `Cat` are both `Mammals`, they can be treated as `Mammals` when needed. For example, they both can `make_sound`, even though they make different sounds. This ability to treat objects of different classes in a uniform way is the core of polymorphism.

3.2.2 Inheritance

Inheritance creates a hierarchical relationship between classes and allows a class to be a *subclass* (or subtype) of one other class (the *superclass* or supertype) In Fig. 3.1, `Mammal` is the superclass of `Dog` and `Cat`. `Dog` and `Cat` are the subclasses of `Mammal`, which is the superclass of both `Dog` and `Cat`.

Subclasses can override methods defined in the superclass and can also define new methods. For example, `Dog` overrides the `make_sound` to provide a specific implementation, and defines a new method `bark` that is specific to dogs.

This is an example of single inheritance, where a subclass can inherit from only one superclass. Python also supports multiple inheritance, where a subclass can inherit from multiple superclasses. For example, an `HybridVehicle` class could inherit from both `Car` and `BatteryVehicle` classes. However, multiple inheritance can lead to complex hierarchies and potential conflicts, so it should be used judiciously.

The difference between inheritence and polymorphism is that inheritance is a mechanism for code reuse and defining relationships between classes, while polymorphism is a mechanism for treating objects of different classes in a uniform way.

3.2.3 Abstract Class

OOP has two types of classes: *concrete* and *abstract* classes. Concrete classes provide a full implementation of the type while abstract classes provide at most a partial implementation of the type. Abstract classes cannot be instantiated (no objects) since some of their methods are not yet implemented (abstract methods). Thus, abstract classes act as a specification (that any subclass must adhere to) while concrete classes act as an implementation.

In Python abstract classes are defined using the `abc` module, which provides the `ABC` class and the `abstractmethod` decorator. The `ABC` class is used as a base class for abstract classes, and the `abstractmethod` decorator is used to mark methods as abstract. In Fig. 3.1, `Mammal` is an abstract class and contains an abstract method `make_sound` that its subclasses must implement. In Java, abstract classes and methods are defined using the `abstract` keyword, e.g., `public abstract class Mammal` and `public abstract void make_sound();`.

Interface Interface is a special type of abstract classes that contains only abstract methods (no concrete methods). They define a specification that classes must adhere to, providing the methods that must be implemented by any class that implements the interface. Multiple classes can implement the same interface, allowing for polymorphism and flexibility in the design.

In Python, interfaces are not explicitly defined, but the concept can be implemented using abstract classes with only abstract methods. For example, the abstract class `Mammal` in Fig. 3.1 acts as an interface that specifies the `make_sound` method that all mammals must implement. In Java, interfaces are explicitly defined using the `interface` keyword, e.g., `interface Mammal`, and methods are declared without a body, e.g., `public void make_sound();`. A class can implement multiple interfaces, allowing for more flexibility in defining contracts between classes.

Example Interface: Comparable A good example of an interface is `Comparable`, which defines a single method `compare_to` that allows objects to be compared to each other. Any class that implements `Comparable` can be compared to other objects of the same type, enabling sorting and other operations that require comparison.

The code below demonstrates the use of the `Comparable` interface in Python. The `Number` class implements the `Comparable` interface by defining the `compare_to` method, which compares two `Number` objects based on their values. The `sort` function uses the `compare_to` method to sort a list of `Number` objects.

```python
from abc import ABC, abstractmethod
from typing import List

# Define a Comparable interface using ABC
class Comparable(ABC):
    @abstractmethod
    def compare_to(self, other: "Comparable") -> int:
        """Compares this object with another."""
        pass

# Implement Comparable in a concrete class
class Number(Comparable):
    def __init__(self, value: int):
        self.value = value

    def compare_to(self, other: "Number") -> int:
        if self.value < other.value:
            return -1
        elif self.value > other.value:
            return 1
        else:
            return 0

# Polymorphic sorting function that relies on the compare_to method
def sort(items: List[Comparable]) -> List[Comparable]:
    return sorted(items, key=lambda x: x.value)

# Usage
numbers = [Number(3), Number(1), Number(4), Number(2)]
sorted_numbers = sort(numbers)
print(sorted_numbers)  # Output: [1, 2, 3, 4]
```

3.2.4 Element Subtype vs Related Subtype

There are two types of subtypes: *element subtype* and *related subtype*. They differ in how they define the relationship between types.

Element subtype relies on a common interface or abstract class, e.g., `Number` is an subtype of `Comparable`. While this common approach allows for polymorphism, it requires all potential types must be pre-planned to fit the hierarchy.

On the other hand, a *related subtype* does not directly rely on a common interface or abstract class (which might be designed much later). Instead, this approach creates a related subtype that implement the desired interface and then adapts it to the existing hierarchy. The code below demonstrates the use of a related subtype, where `Price` is adapted to `PriceComparable`, which implements `Comparable`, to allow sorting of `Price` objects.

```
class Price:
    def __init__(self, amount: float):
        self.amount = amount


class PriceComparable(Comparable):
    def __init__(self, price: Price):
        self.price = price
    def compare_to(self, other: "PriceComparable") -> int:
        if self.price.amount < other.price.amount:
            return -1
        elif self.price.amount > other.price.amount:
            return 1
        else:
            return 0

# sorting using related subtype
prices = [Price(3.0), Price(1.0), Price(4.0), Price(2.0)]
price_comparators = [PriceComparable(p) for p in prices]
sorted_prices = sort(price_comparators)
```

3.2.5 Dynamic Dispatching

Dynamic dispatching is the fundamental technique that enables polymorphism in OOP. It refers to how a program selects which method to invoke when a method is called on an object. It allows the correct method to be invoked based on the *runtime type* of the object, even if the reference to the object is of a more general (superclass) type. This is particularly useful when working with inheritance and polymorphism, where subclasses override methods from a superclass. The distinction between dynamic dispatching and static dispatching lies in when the decision about which method to invoke is made—either at runtime (dynamic) or compile-time (static).

In Fig. 3.1 the `mammal_make_sound` method will invoke the `make_sound` method of the correct subclass based on the runtime type of the object. This is dynamic dispatching in action, where the method `make_sound` to be called is determined at runtime based on the actual type of the object. However, if we explicitly create a `Dog` instance and call `make_sound` on it, the method is statically dispatched, as the compiler knows the type of the object at compile-time and can directly call the correct method.

The code below demonstrates the difference between static and dynamic dispatching. The `Dog` object `d` is statically dispatched, while the `Mammal` object `m` is dynamically dispatched.

```
Dog d = Dog();
d.make_sound();  # Static dispatching

Mammal m = Dog();
m.make_sound();  # Dynamic dispatching
```

3.2.6 Encapsulation

Encapsulation is a major OOP idea that allows the class to control access to its data and methods. It helps ensure that the internal representation of the class is not exposed to the outside world and prevents unintended modifications to the internal state of the class.

Encapsulation is achieved through the use of access modifiers, which specify the level of access to class members. In Java, access modifiers are enforced by the language, and there are four levels of access: *private*, *protected*, *package-private* (default), and *public*. In Python, access modifiers are not enforced by the language, but conventions are used to indicate the intended level of access. For example, underscore (_) is used to indicate private or protected attribute (variable).

Encapsulation avoids direct access to the internal representation of a class, e.g., rep-invariants, which can lead to unintended side effects and break the class's invariants. Instead, access to the class's data should be controlled through methods, such as `getters` and `setters` methods.

In the `BankAccount` class in Fig. 3.2, the `_balance` attribute is a private member, and access to it is controlled through the `deposit` and `withdraw`. The `BonusBankAccount` class extends `BankAccount` and adds a `_bonus_interest` attribute, which is also a private member that is not exposed directly.

3.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) is a fundamental concept of object-oriented design. The idea that if S is a subclass of T then objects (or instances) of S can be used in place of objects of T without affecting the correctness of the program. In other words, a subclass *is-a* superclass and can do everything the superclass can do. For example, a `Dog` is a `Mammal` and can make sound like any mammal, but it can also bark, which is specific to dogs.

LSP promotes proper design and enforces correct use of inheritance. Violating LSP can lead to unexpected behavior and errors in the program, as the assumptions made about the superclass may no longer hold for the subclass.

3.3.1 Rules

If `S` is a subtype of `T`, then objects of type `T` may be replaced with objects of type `S` without altering any of the desirable properties of the program. This means whenever you use `T`, you can use `S` instead. To achieve this, we must follow the following rules:

Signature Rule The signatures of methods of `S` must strengthen methods of `T`. In other words, the methods of `S` are a superset of the methods of `T`. Thus, if `T` has `n` methods, `S` also has `n` methods and additional ones (methods specific to `S`).

For the example in Fig. 3.1, `Dog` and `Cat` have the same `make_sound` method as `Mammal`, but they also have additional methods specific to dogs and cats. In Fig. 3.2, `BonusBankAccount` has the same methods as `BankAccount`.

Method Rule If `f` be a method of `T` and `f'` be a method of `S` that overrides `f`, then the specification of `f'` is the same or strengthen the specification of `f`. This means that the preconditions of `f'` must be weaker or equal to the preconditions of `f`, i.e., `f'` accepts more inputs than `f`. The postconditions of `f'` must be stronger or equal to that of `f`. This means that `f'` is more precise than `f`.

In Fig. 3.2 the `deposit` of `BonusBankAccount` has a stronger postcondition than `deposit` of `BankAccount` as it adds bonus interest to the deposit.

Property Rule The subtype must preserve all properties of the supertype. For example, the rep-invariant of the subtpe `S` must be stronger or equal to that of the supertype `T`. This means `S` should maintain or strengthen the properties (including rep invariants) of `T`

The rep-inv, captured in the `repOK` method, of `BonusBankAccount` in Fig. 3.2 is stronger than that of `BankAccount` as it includes the rep-inv of `BankAccount` *and* the constraint on the bonus interest.

Note that LSP is meant to be a *guideline* for designing classes and inheritance hierarchies, and it is not always possible to strictly adhere to it in all cases. The compiler cannot enforce LSP, so it is up to the programmer to ensure that the principle is followed.

3.4 Exercise

3.4.1 LSP: Market

Determine whether the `LowBidMarket` and `LowOfferMarket` classes in Fig. 3.3 are proper subtypes of `Market`. Specifically, for each method, list whether the precondition is weaker, the postcondition is stronger, and conclude whether LSP holds.

Note that this is purely a "paper and pencil" exercise. No code is required. Write your answer so that it is easily understandable by someone with only a passing knowledge of LSP.

3.4.2 Polymorphism concepts: Vehicle

You will design a system that models different types of vehicles (e.g., cars, bicycles). Each vehicle has the ability to start, stop, and display its details. Vehicles should

```python
class BankAccount:
    def __init__(self, balance: float):
        self._balance = balance if balance >= 0 else 0

    def repOK(self):
        return self._balance >= 0

    def deposit(self, amount: float) -> bool:
        """
        REQUIRES: amount must be positive
        EFFECTS: balance is the original balance plus deposited amount
        """
        if amount < 0:
            return False
        self._balance += amount
        # check_repOK()
        return True

    def withdraw(self, amount: float) -> bool:
        # REQUIRES: amount must be positive and less than or equal to balance
        # EFFECTS: balance is the original balance minus withdrawn amount

        if amount < 0 or amount > self._balance:
            return False
        self._balance -= amount
        self.check_repOK()
        # check_repOK()
        return True

class BonusBankAccount(BankAccount):
    def __init__(self, balance: float, bonus_interest: float):
        super().__init__(balance)
        self._bonus_interest = bonus_interest

    def deposit(self, amount: float) -> str:
        # REQUIRES: (same) amount must be positive
        # EFFECTS: (stronger) same post as deposit of BankAccount and also add bonus interest

        stats = super().deposit(amount)
        if stats:
            # deposit successful, add interest
            self._balance += self._bonus_interest * amount

        # check_repOK()
        return stats


    def withdraw(self, amount: float) -> bool:
        """
        REQUIRES: (weaker) allow zero withdrawals, which are ignored
        EFFECTS: (same) balance is the original balance minus withdrawn amount
        """
        if amount == 0:
            return True  # Zero withdrawal is considered a no-op
        ret =  super().withdraw(amount)
        # check_repOK()
        return ret


    def repOK(self):
        """
        Stronger Rep-inv: balance and bonus interest must be non-negative
        """
        return super().repOK() and self._bonus_interest >= 0
```

Fig. 3.2: Liskov Substitution Principle demonstration

```python
class Market:
    def __init__(self):
        self.wanted = set()  # items for which prices are of interest
        self.offers = {}     # offers to sell items at specific prices

    def offer(self, item, price):
        """
        Requires: item is an element of wanted.
        Effects: Adds (item, price) to offers.
        """
        if item in self.wanted:
            if item not in self.offers:
                self.offers[item] = []
            self.offers[item].append(price)

    def buy(self, item):
        """
        Requires: item is an element of the domain of offers.
        Effects: Chooses and removes some (arbitrary) pair (item, price) from
                    offers and returns the chosen price.
        """
        if item in self.offers and self.offers[item]:
            return self.offers[item].pop(0)  # Removes and returns the first price
        return None

class LowBidMarket(Market):
    def offer(self, item, price):
        """
        Requires: item is an element of wanted.
        Effects: If (item, price) is not cheaper than any existing pair
                    (item, existing_price) in offers, do nothing.
                    Else add (item, price) to offers.
        """
        if item in self.wanted:
            if item not in self.offers:
                self.offers[item] = []
            # Only add if price is lower than existing prices
            if not self.offers[item] or price < min(self.offers[item]):
                self.offers[item].append(price)

class LowOfferMarket(Market):
    def buy(self, item):
        """
        Requires: item is an element of the domain of offers.
        Effects: Chooses and removes the pair (item, price) with the
                    lowest price from offers and returns the chosen price.
        """
        if item in self.offers and self.offers[item]:
            # Find and remove the lowest price from the list
            lowest_price = min(self.offers[item])
            self.offers[item].remove(lowest_price)
            return lowest_price
        return None
```

Fig. 3.3: LSP Market Exercise

differ in their implementation of these behaviors. You will use abstract classes and interfaces to define the basic structure and ensure that your system adheres to OOP principles.

1. Create an abstract class `Vehicle` that has
   (a) An encapsulated attribute for `speed`.
   (b) Abstract methods: `start()`, `stop()`, and `display()`.
2. Define an interface called `Refuelable`, with a method `refuel(amount:int)`
3. Create concrete subclasses
   (a) Create `Car` and `Bicycle` classes that inherit from `Vehicle`.
   (b) Car also implements the `Refuelable` interface (because it uses fuel).
   (c) Implement methods to `start`, `stop`, `display`, and `refuel` if applicable.
   (d) Ensure each class encapsulates its specific properties (e.g., `fuel_level` for cars).
4. Demonstrate Polymorphism and other OOP principles
   (a) Create a function `operate_vehicle(vehicle:Vehicle)` that accepts any vehicle type and calls its `start`, `stop`, and `display` methods. This function demonstrates polymorphism and dynamic dispatching.
   (b) Create test cases to demonstrate LSP by substituting instances of `Car` and `Bicycle` for `Vehicle` in the `operate_vehicle` function.
   (c) Protect rep data and other attributes and access them through setters and getters methods.
   (d) Provide proper document and specifications for your code (e.g., class Overview, rep-invs, method specifications, AF, `repOK`).

3.4.3 LSP: Reducer

For the classes `A`, `B`, and `C` in Fig. 3.4, determine whether LSP holds in the following cases. Specifically, for each case, list whether the precondition is weaker, the postcondition is stronger, and conclude whether LSP holds.

1. `B` extends `A`.
2. `C` extends `A`
3. `A` extends `B`
4. `C` extends `B`
5. `A` extends `C`

```python
class A:
    def reduce(self, x):
        """
        Effects: if x is None, raise ValueError;
                 if x is not appropriate, raise TypeError;
                 else, reduce this by x.
        """

class B:
    def reduce(self, x):
        """
        Requires: x is not None.
        Effects: if x is not appropriate, raise TypeError;
                 else, reduce this by x.
        """

class C:
    def reduce(self, x):
        """
        Effects: if x is None, return normally with no change;
                 if x is not appropriate, raise TypeError;
                 else, reduce this by x.
        """
```

Fig. 3.4: LSP Exercise: Update

3.4.4 LSP Analysis

Consider the following classes with their specifications for the `update()` method:

```python
class A:
    def update(self, value):
        """
        Effects/Post: If value is not valid, do nothing;
                 otherwise, update this with value.
        """

class B:
    def update(self, value):
        """
        Requires/Pre: value must be an integer.
        Effects/Post: If value is valid, update this with value;
                 otherwise, do nothing.
        """

class C:
    def update(self, value):
        """
        Effects/Post: If value is invalid, set default update;
                 otherwise, update this with value.
        """
```

For each case below, determine if LSP holds by checking whether the preconditions are weaker and the postconditions are stronger, and conclude whether LSP holds. Note that as soon as one rule is violated, LSP does not hold. Also note that if it is not possible to directly compare the pre/post conditions, make assumptions based on your preference, and determine if LSP holds based on those assumptions.

1. `B` extends `A`
2. `C` extends `A`
3. `A` extends `B`
4. `C` extends `B`
5. `A` extends `C`

Chapter 4

Iterators

Iterators and generators are powerful concepts in OOP that enable efficient traversal and on-the-fly computation of sequences of data. They allow developers to handle large datasets, abstract complex data traversal patterns, and create custom iterators for any type of object.

> **History** The idea of iterators in OOP was pioneered by the CLU language in the 1970s, developed by Barbara Liskov. CLU introduced iterators as a core language feature, allowing traversal of collections without exposing internal structures. This innovation laid the foundation for modern iterator designs and showed how encapsulating traversal could lead to cleaner, more maintainable code. C++ in the 1980s introduces iterators through its STL. Iterators was further solidified by the Design Patterns book by the Gang of Four (GoF) in 1994, which formalized iterator patterns, emphasizing the separation of traversal from data structure.
>
> Java, released in 1995, built on these ideas through its Iterator interface, standardizing the way collections were traversed across the language. Java's approach unified data traversal, promoting encapsulation and abstraction in OO. Python introduces generators in 2001 and allowed functions to produce values lazily, one at a time, without storing the entire sequence in memory. This enables efficient data processing for large or infinite sequences and emphasizes efficient iteration over data in modern languages.

4.1 Motivation

Let's consider a scenario where you need to generate Fibonacci numbers. A common but inefficient approach is to generate Fibonacci numbers up to a certain limit and store all them in a list, which consumes a lot of memory.

```python
def generate_fib_list(n: int) -> list[int]:
    fib_sequence = [0, 1]
    for _ in range(2, n):
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
```

```
    return fib_sequence

# Create the first 100K Fibs; consume lots of memory for storing all numbers
fib_numbers = generate_fib_list(10**6)
print(fib_numbers[:10])  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34] # only use first 10
```

This approach is inefficient because it generates all Fibonacci numbers up to a certain limit and stores them in a list, which consumes a lot of memory, especially for large sequences. Also, this approach is wasteful because it generates all Fibonacci numbers at once, even if only a few are needed. A more efficient approach is to use an iterator or generator to produce Fibonacci numbers on the fly, only when needed.

```
# Efficient generator function that yields Fibonacci numbers on demand
def fib_generator(n: int):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

print(list(fib_generator(10))) # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Using generator functions, we can efficiently generate Fibonacci numbers on demand, reducing memory consumption and improving performance. Generators produce values one at a time, only when needed, making them ideal for large datasets or infinite sequences.

4.2 Iterators

An iterator is an ADT that allows you to traverse through all the elements of a collection, such as a list, tuple, or custom data structure, without exposing the underlying details of the collection (i.e., encapsulation).

Key Concepts of Iterators:

- Iteration Methods: An iterator object implements two key methods: `__iter__()` and `__next__()`.
  - `__iter__()`: Returns the iterator object itself and is implicitly called at the start of loops.
  - `__next__()`: Returns the next element in the sequence and raises a StopIteration exception when there are no more elements.
- State Management: Iterators manage their own state, allowing them to keep track of the current position in the collection.

```
class Countdown:
    def __init__(self, start: int):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current \le 0:
            raise StopIteration
        else:
            current_value = self.current
```

```
            self.current -= 1
            return current_value

# Usage of the custom iterator
countdown = Countdown(5)
for number in countdown:
    print(number)

# Output: 5, 4, 3, 2, 1
```

In the example above, the `Countdown` class implements iteration by defining the `__iter__()` and `__next__()` methods. The `__iter__()` method returns the iterator object itself, while `__next__()` manages the countdown state by returning the next element in the countdown sequence and stopping the iteration by raising the `StopIteration` exception when the countdown reaches zero.

Benefits of Iterators

- Memory Efficiency: Iterators retrieve elements one at a time, reducing memory usage compared to loading all elements at once.
- Encapsulation: Iterators hide the internal structure of the collection, providing a clean, consistent interface for traversal.
- Flexibility: Custom iterators can be defined for any object, making them adaptable to a wide range of data structures.

4.3 Generator

A generator is a special iterator that uses the `yield` keyword. Generators allow you to turn a method into one that behaves like an iterator, without having to create a separate iterator class. Generator thus has the same benefits as iterators, such as memory efficiency and encapsulation, and does not require the explicit implementation of the `__iter__()` and `__next__()` methods.

```
# Generator function for a countdown
def countdown(start: int):
    while start > 0:
        yield start
        start -= 1

# Usage of the generator
for number in countdown(5):
    print(number)
# Output: 5, 4, 3, 2, 1
```

Instead of defining the `Countdown` class as an iterator, the `countdown` function is defined as a generator that yields the countdown sequence. Each call to `yield` returns the current value of `start` and saves the function's state, allowing it to resume where it left off when called again.

Benefits of Generators

- Conciseness: Generators provide a more straightforward syntax for creating iterators.
- Performance: They generate values on demand, reducing memory consumption compared to traditional lists.
- Enhanced Readability: Generator functions are typically easier to understand and maintain compared to an iterator class.

4.4 Exercise

4.4.1 Prime Number

A *prime number* is a natural number greater than 1 that has no positive divisors other than 1 and itself. In this exercise, you will implement three different approaches to generate prime numbers: a non-iterator method, a custom iterator class, and a generator function. You will compare the performance of these approaches and observe the benefits of using iterators and generators.

1. Write a non-iterator and non-generator method `gen_prime` that generates prime numbers up to a specified limit.
   (a) Test the iterator by printing all prime numbers that is less than 50.
   (b) Measure the performance of the iterator by generating all prime numbers that your computer can handle (in Python, use `time(...)`). Try various limits and measure the time.
2. Write a custom iterator called `PrimeNumberIterator` that generates prime numbers up to a specified limit.
   (a) The class needs to have `__iter__()` and `__next__()` methods.
   (b) Use a helper function to check for prime numbers (reuse the code in `gen_prime`).
   (c) Raise `StopIteration` when the current number exceeds the limit.
   (d) Test the iterator by printing all prime numbers that is less than 50.
   (e) Measure the performance of the iterator by generating all prime numbers that your computer can handle like before. Try various limits and measure the time.
3. Write a generator function called `gen_prime_generator` that yields prime numbers up to a specified limit (this means using the `yield` keyword).

(a) Test the generator by printing all prime numbers that is less than 50.

(b) Measure the performance of the generator by generating all prime numbers that your computer can handle like before. Try various limits and measure the time.

4.4.2 Perfect Number Generation

A *perfect number* is a positive integer that is equal to the sum of its proper divisors, excluding itself (e.g., 6, 28). You will implement three different approaches to find perfect numbers up to a given limit, comparing their performance and resource usage.

For this exercise, you can use either Python or Java. You need to submit your code with *clear documentation* on how to run and test your code. That is, you must explicitly state the commands to run your code and the expected output. You will also need to provide screenshots or logs of the execution results, including the time taken and memory usage.

If you do not provide clear documentation, you will not receive credit. If we cannot run your code, you will not receive credit. If we do not see the results you claim, you will not receive credit.

1. Part 1: Generate Perfect Numbers Without Iterators or Generators. Write a method `gen_perfect` that generates perfect numbers up to a given positive value `n`, i.e., generate perfect numbers less than or equal to `n`. You will not use iterators or generators and store all perfect numbers in a list.

   (a) Play around with different `n` (e.g., 10,000, 100,000) to see how the program performs. Aim for about 20 seconds of execution time.

   (b) Print out the first 5 perfect numbers generated. Note that if this takes too long, print out the first `n` numbers that seems to take reasonable time. Be sure to document and explain your choice of `n`.

   (c) Measure execution time and memory usage, which should be relatively high due to high computational demands and storage of all perfect numbers. For Python, use `timeit` and `tracemalloc` modules to measure time and memory usage.

2. Part 2: Implement a custom iterator called `PowerNumberIterator` for perfect numbers. You can reuse the code from part 1. After that, do exactly the analysis that you did in Part 1, i.e., play with different `n` values, print out the first 5 numbers generated, and measure the performance of the iterator. You should see a significant improvement in memory usage and execution time compared to the non-iterator approach.

3. Part 3: Use a generator function `gen_power_generator` to yield perfect numbers. Reuse the code from part 1 and make changes to it to use generator. Then do the same analysis as in Part 1 and Part 2.

4. Part 4: Write a short report comparing the performance of the three approaches. Include the time taken, memory usage, and ease of implementation. Discuss the benefits of using iterators and generators over the non-iterator approach.

4.4.3 Iterator and Generator Multiple Choice

1. What does this class represent?

```python
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

   (a) A list that can be iterated through once.
   (b) An infinite loop.
   (c) An iterator that generates numbers between `start` and `end`, inclusive.
   (d) A generator that yields values on demand.

2. What is main advantage of using a generator in this example?

```python
def count_down(n):
    while n > 0:
        yield n
        n -= 1
```

   (a) It stores all the countdown numbers in mem at once.
   (b) It allows for lazy evaluation, producing numbers one at a time without storing in memory.

3. What is returned by `fibonacci`?

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
```

   (a) The sum of all Fib numbers up to `n`.
   (b) Fib numbers up to `n`, one by one, using lazy evaluation.

(c) The first `n` Fib numbers.

(d) The Fibonacci sequence stored as a tuple.

4. What happens if you try to convert the generator `generate_squares` to a list?

```
def generate_squares(limit):
    for i in range(limit):
        yield i ** 2
```

(a) It yields values 1-by-1 instead of storing in memory.

(b) It returns an error.

(c) It gets exhausted and returns an empty list.

(d) It will create a list of square numbers up to `limit - 1`.

5. What is the purpose of this generator?

```
def infinite_numbers():
    num = 0
    while True:
        yield num
        num += 1
```

(a) It generates numbers up to a fixed limit.

(b) It produces numbers starting from 0, but stops after a certain point.

(c) It generates an infinite sequence of numbers, one at a time.

(d) It returns numbers in ascending order.

Chapter 5

First-Class Functions

In modern OOP, functions are treated as *first-class* citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

```
def greet(name):
    return f"Hello, {name}!"

# Assigning the function to a variable
greeting = greet

# `greeting` can now be used like the function `greet`
print(greeting("Alice"))  # Output: Hello, Alice!
```

In this example, the `greet` function is assigned to a variable `greeting`, which can then be called like a regular function.

```
def apply(op, a:int, b:int) -> int: return op(a, b)
def add(x:int, y:int) -> int: return x + y
def subtract(x:int, y:int) -> int: return x - y

# Passing functions as arguments
result_add = apply_op(add, 10, 5)       # Output: 15
result_subtract = apply_op(subtract, 10, 5)  # Output: 5
```

For this example, `apply_op` takes another function `op` as an argument and applies it to the given arguments. This allows for dynamic behavior based on the function passed to `apply_op`.

> **History** Lisp, developed by John McCarthy in the late 1950s, was one of the first languages to treat functions as first-class citizens. Lisp's approach to functions was heavily influenced by *lambda calculus*, developed by Alonzo Church in the 1930s, which formalized functions as mathematical expressions. Lisp's support for first-class functions allows for powerful programming techniques, such as higher-order functions (§5.2). Modern programming languages including Python, JavaScript, and Ruby all treat functions as first-class citizens.

5.1 Anonymous and Lambda Functions

A popular use for first-class functions is to create *anonymous* or *lambda* functions, which are unnamed functions defined on the fly. Lambda functions are useful for short, simple operations that do not require a full function definition.

```
# Lambda function to square a number
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

In the example above, a lambda function is used to define a function that squares a number. The lambda function is assigned to the variable `square` and can be called like a regular function. Lambda functions are often used in conjunction with higher-order functions like `map`, `filter`, and `reduce`, described in §5.2, to perform operations on collections of data.

5.2 Higher-Order Functions

In the world of first-class functions, functions that operate on other functions are called *higher-order functions*. More specifically, a higher-order function is a function that takes one or more functions as arguments or returns a function as its result.

```
def square(x):
    return x * x

def cube(x):
    return x * x * x

def apply_to_list(func, numbers):
    return [func(number) for number in numbers]

numbers = [1, 2, 3, 4, 5]
print(apply_to_list(square, numbers))  # Output: [1, 4, 9, 16, 25]
print(apply_to_list(cube, numbers))    # Output: [1, 8, 27, 64, 125]
```

In this example, the higher-order function `apply_to_list` takes a function and a list of numbers as inputs and applies the function to each number in the list, returning a new list with the results.

5.2.1 Popular Higher-Order Functions

Higher-order functions are commonly used in functional programming and are available in many programming languages. Three popular higher-order functions include:

- `map(f, iterable)`: Applies a function `f` to each item in an iterable (e.g., list, tuple) and returns a new iterable with the results.
  Example: `list(map(square, [1,2,3,4,5]))` returns `[1, 4, 9, 16, 25]`.
- `filter(f, iterable)`: Filters elements in an iterable based on a predicate `f` (i.e., a function that returns a boolean value).

Example: `list(filter(lambda x: x % 2 == 0, [1,2,3,4,5]))` returns `[2, 4]`. Lambda functions are discussed in the next section (§5.1).

- `reduce(f, iterable)`: Applies a binary function `f` to the first two items of an iterable, then to the result and the next item, and so on. It returns a single value.
  Example: `reduce(lambda x, y: x + y, [1,2,3,4,5])` returns `15`.

Fun Fact While `reduce` is well-known in functional languages such as Haskell and Ocaml, the Python community believes that list comprehensions and generator expressions made the code more readable than `reduce`. Thus, in Python 3, `reduce` was moved to the `functools` module to emphasize its specialized use case.
For example, compare the following code snippets that calculate the sum of a list of numbers using `reduce` and list comprehension:

```python
# Calculate the sum of a list of numbers using reduce
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers)
print(total)  # Output: 15

# using list comprehension
total = sum(numbers)
print(total)  # Output: 15

# using generator expression
total = sum(x for x in numbers)
print(total)  # Output: 15
```

Fun Fact The MapReduce framework, introduced by Google in 2004, was inspired by `map` and `reduce` ("map" distributes work across multiple nodes and the "reduce" aggregates the results). It revolutionizes large-scale data processing and allows Google to index the web efficiently. It influences current web technologies such as Apache Hadoop and Apache Spark.

5.3 Closures

Closures are a higher-order function that returns a function. It is a powerful feature of first-class functions and allows functions to retain access to variables from their enclosing scope even after the scope has finished executing.

Fun fact Closures are used extensively in Javascript, introduced in the Netscape browser in 1995 by Bredan Eich. Javascript supports closures and first-class functions and enables the development of dynamic and interactive web applications, leading to its widespread adoption and popularity.

```python
def make_multiplier(factor):
    def multiplier(x):   # a closure
        return x * factor
    return multiplier  # Return the closure

# Create a function that multiplies by 3
times_three = make_multiplier(3)
print(times_three(5))  # Output: 15

# Use with higher-order functions
numbers = [1, 2, 3, 4, 5]
multiplied_numbers = list(map(make_multiplier(2), numbers))
print(multiplied_numbers)  # Output: [2, 4, 6, 8, 10]
```

Fig. 5.1: Closure example. Note that this example also illustrate curry, a form of closure (§5.4)

Examples The above example demonstrates a closure where the `make_multiplier` function returns the *closure* inner function `multiplier` that multiplies a number by a given factor. The `times_three` function is created by calling `make_multiplier(3)`, which returns a function that multiplies by 3. The closure allows the `multiplier` function to retain access to the `factor` variable even after `make_multiplier` has finished executing.

```python
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager

avg = make_averager()
print(avg(10))  # Output: 10.0
print(avg(11))  # Output: 10.5
print(avg(12))  # Output: 11.0
print(avg(13))  # Output: 11.5
```

In the example above, the `make_averager` function creates a closure that calculates the average of a series of numbers. The `averager` function retains access to the `series` list, allowing it to accumulate values and calculate the average over time.

5.4 Currying

Currying is a special form of closure. The curried function takes one argument at a time and returns a new function that takes the next argument. In other words, it transforms a function of arity n to n functions of arity 1.

The `make_multiplier` function in Fig. 5.1 is an example of currying. The function needs 2 arguments, but it is transformed into a series of 2 function calls where each take 1 argument. For example, `make_multiplier(2)(3)` is equivalent to `2*3`.

> **History** Currying was introduced by Haskell Curry in the 1930s. Currying and higher-order functions (§5.2) are widely-used in functional programming languages such as Ocaml and Haskell (named after Haskell Curry).

5.5 Exercise

5.6 Functions First

In this exercise you will demonstrate the concepts of higher-order functions, lambda functions, and closure. Example code are written in Python but you can use Python or any other language that supports these features.

1. Part 1: Create a *higher-order* function that applies different operations (addition, subtraction, multiplication) to two numbers.
   (a) Create a function called `operate_on_numbers (operation: function, a: int, b: int) -> int` that takes another function (operation) as an argument and applies that function to two numbers.
   (b) Create multiple simple functions `add`, `subtract`, `multiply` that can be passed as arguments to `operate_on_numbers`.
   (c) Test the function by applying each operation to two numbers and printing the results.
   ```
   print(operate_on_numbers(add, 5, 3))        # Output: 8
   print(operate_on_numbers(subtract, 5, 3))   # Output: 2
   print(operate_on_numbers(multiply, 5, 3))   # Output: 15
   ```
2. Part 2: Modify the code from Part 1 to use *lambda functions*
   (a) Replace `add`, `subtract`, and `multiply` with lambda expressions.
   (b) Test the function by applying each operation to two numbers and printing the results.
   (c) Discuss when you would want to use lambda functions? When would you want to use a name function?
3. Part 3: Using higher-order functions
   (a) For each higher-order function `map`, `filter`, and `reduce`, create some code to apply each to a list of `str`.
   (b) Clearly explain what each function does and print several examples to demonstrate each function.

(c) `reduce` also takes a third input called an *accumulator*. Explain how `reduce` works with the accumulator ? e.g., `reduce(f,[1,2,3,...,n],acc)` does what?

(d) Create some code to demonstrate the use of the accumulator in `reduce`. DO NOT use the example in the lectures (eg.., sum, product, subtract).

4. Part 4: Write a function `make_max_tracker()` that returns a closure that tracks and returns the highest number seen so far. In Python, to access a variable that is not in scope, you might need to use the `nonlocal` keyword, e.g., `nonlocal var_name`.

```
def make_max_tracker():
    ...
    def tracker(v):
       ...

    return tracker

max_tracker = make_max_tracker()

# Test closer, notice how it "memorizes" what it has seen so far.
print(max_tracker(5))   # Output: 5
print(max_tracker(3))   # Output: 5
print(max_tracker(8))   # Output: 8
print(max_tracker(7))   # Output: 8
```

5.6.1 E1

1. Explain the difference between a *higher-order* function and a *closure*. Provide an example of each.

2. When would you use a *lambda function* over a regular function? Provide an example.

3. Write a function `make_min_tracker()` that returns a closure which tracks and returns the lowest number seen so far.

```
def make_min_tracker():
    ...
    def tracker(v):
       ...

    return tracker

min_tracker = make_min_tracker()
print(min_tracker(5))   #  5
print(min_tracker(3))   #  3
print(min_tracker(8))   #  3
print(min_tracker(-1)) # -1
print(min_tracker(0))   # -1
```

Part II

Testing and Fault Localization

Chapter 6

Testing

The terms *validation*, *verification*, and *testing* are commonly used in software development for quality assurance. **Validation** is a process typically achieved by verification and validation to ensure the program behaves as expected. **Verification** ensures that the program works on *all possible inputs*. Verification provides better guarantee but is expensive or impossible for large programs.

In contrast, **testing** checks that the program behaves as expected over *some inputs*. Testing only shows the program works on the test inputs, but it is usually cheaper to do (comparing to verification). Software developers are often more familiar with testing, e.g., by running the program with various inputs. We focus on testing in this chapter.

6.1 Black-box Testing

Black-box approach tests the program using its specifications (e.g., type of inputs, expected outputs) *without* any knowledge of its internal implementation. In fact, blackbox testing does not even require the program code (hence the name blackbox). The approach is efficient and easy to use, but can miss certain bugs.

```
class MathStuff:
    def square(self, x:int) -> int:
        if x == 123:
            return -1 # bug
        else:
            return x*x

    def div(self, x:int, y:int) -> int:
        if y == 0:
            raise ValueError("Cannot divide by 0")
        else:
            return x // y


""" Only test on integer inputs and check that the outputs are as expected"""
ms = MathStuff()
assert ms.square(0) == 0
```

```
assert ms.square(1) == 1
...
assert ms.square(12) == 144
assert ms.square(-5) == 25

assert ms.div(10, 2) == 5
assert ms.div(10, 3) == 3

try:
    ms.div(10, 0)
except ValueError:
    # raise an exception is expected
    pass
else:
    print "Error: Should have raised an exception"
```

For these functions (`square, div`) we simply test them with various numbers as inputs and check that the outputs are as expected. We do not need to know how the functions (e.g., `square`) were implemented. Observe that because of this, we do not know about the special "buggy" case of `123` in `square` and thus do not test for it. This is a limitation of blackbox testing.

6.1.1 Unit Testing

Modern OOP languages often have built-in capability or library to help with testing. *Unit testing* is a popular and supported by most languages to test individual *units* (e.g., functions, classes) of the program. Below is a small example of using Python's `unittest` library to test the `MathStuff` class (§6.1).

```
import unittest
class TestCalculator(unittest.TestCase):
    ## setup unit tests. This is run before each test
    def setUp(self):
        self.ms = MathStuff()

    # Basic Unit Tests
    def test_square(self):
        self.assertEqual(self.ms.square(0), 0)
        self.assertEqual(self.ms.square(1), 1)
        self.assertEqual(self.ms.square(12), 144)
        self.assertEqual(self.ms.square(-5), 25)



    def test_div(self):
        self.assertEqual(self.ms.div(10, 2), 5)
        self.assertEqual(self.ms.div(10, 3), 3)

        with self.assertRaises(ValueError):
            self.ms.div(10, 0)

if __name__ == "__main__":
    unittest.main()
```

6.1.2 Special/Edge Cases Testing

This testing runs the program on special or edge cases to find bugs that are not caught by regular inputs.

For example, a program like `MathStuff.square` in §6.1.1 should be tested with negative numbers, zero, and positive numbers. Similarly, for a program that takes a list of numbers as input, special cases could include an empty list, with one element, with all 0's, with all negative numbers, etc.

6.1.3 Fuzz Testing

This testing generates random and *invalid* inputs to test the program. For example, a program expects a number is tested with a string or a dict. It has the similar purpose as special cases testing (§6.1.2), but instead of using specific valid inputs, it generates many random and invalid inputs. Fuzz testing is often used to find security vulnerabilities. Many advanced fuzzing techniques generate new inputs from existing or *seed* inputs, e.g., by flipping bits or changing values slightly.

```
#generate 100 random numbers
for i in range(100):
    x = random.randint(-1000, 1000)
    assert square(x) == x*x

#invalid inputs
for x in [["hello", [1,2,3], {"a":1}]]:
    try:
        square(x)
        assert False, "Should have raised an exception"
    except:
        # raise an exception is expected
        pass

#generate inputs from existing ones
for x in [1,2,3]:
    x2 = x + random.randint(-10, 10)
    assert square(x) == square(x2)
```

6.1.4 Combinatorial Testing

This technique combines different inputs to generate tests. The combination is typically done using *Cartesian* products, i.e., all possible combinations of inputs are tested. Combinatorial testing is useful for finding issues that occur when combining different inputs. For example, a program that takes two numbers as input could be tested with all combinations of positive, negative, and 0 numbers.

```
from parameterized import parameterized
...

xs = [11, 12, -11, -12, 0]
ys = [1, 2, -1, -2, 0]

@parameterized.expand(product(xs, ys))
```

```
def test_div(self, x, y):
    if y == 0:
        with self.assertRaises(ValueError):
            self.ms.div(x, y)
    else:
        expect = x // y
        self.assertEqual(self.ms.div(x, y), expect)
```

For the example above, the `test_div` function is run with all combinations numbers in `xs` and `ys`. The `product` function generates all 25 combinations of the numbers in the lists `xs, ys` (Cartesian product). The `@parameterized.expand` runs the test with each input. Note that while this is illustrated using Python, the concept of combinatorial testing is used in other languages and testing frameworks.

6.1.5 Property-Based Testing

Property-based testing generates random inputs to check specific *properties* of the program. For example, square of a negative number is positive and addition and multiplication being commutative (e.g., $x + y \equiv y + x$). Property-based testing is a convenient way to generate and test desirable behaviors with many inputs.

Assertions Property-based tests often use *assertions* to check the properties. Most languages have the function `assert(c)` or similar that raises an exception if the condition `c` is false.

```
from hypothesis import given
from hypothesis.strategies import integers

@given(integers(), integers()) # create random integers
def test_square(x, y):
    assert square(x) == x*x
    assert square(y) == y*y
    assert square(x) == square(-x) # square of neg is positive

@given(integers(), integers()) # create random integers
def test_add(x, y):
    assert add(x, y) == add(y, x) # commutative
```

This example tests various properties of `square` and `add` with randomly generated integers `x, y`. In Python, you can use the `hypothesis` library, which generates random inputs and runs the tests with them. In Java, you can use the `jqwik` library for property-based testing.

6.2 In-class Exercise: GCD

You are given two implementations computing the GCD (Greatest Common Divisor) of two numbers. One of them is correct and the other has a bug. You will write combinatorial and property tests to find the bug. Recall that the GCD of two numbers is the largest number that divides both of them. For example, `gcd(8,12)=4`.

```python
def gcd_correct(a, b):
    while b != 0:
        a, b = b, a % b
    return abs(a)
```

```python
def gcd_buggy(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Part 1: Using Combinatorial Testing to Find Bugs

- Write code to perform combinatorial testing on `gcd_correct` and `gcd_buggy`. In Python, these would be done by importing the `parameterized` module (§6.1.4)
- Create tests with several positive, negative, and zero numbers.
- Run the tests and show the bug in `gcd_buggy`.
- Explain how combinatorial testing helped find the bug.

Part 2: Using Property Testing to Find Bugs

- Identify several properties of GCD (use Wikipedia if you have to). One of these properties should help you detect the bug in `gcd_buggy`.
- Write code to perform property-based testing on `gcd_correct` and `gcd_buggy`. In Python, these would be done by importing the `hypothesis` module (§6.1.5)
- Run the tests and show the bug in `gcd_buggy`.
- Explain how property-based testing helped find the bug.

6.2.1 Search-Based Software Testing (SBST)

SBST searches for inputs to optimize some objective. Examples include maximizing code coverage, causing a crash, or satisfying a specific property.

6.2.2 Genetic Algorithm

Genetic Algorithm (GA) is an SBST technique that uses biological evolution (Darwin's theory of evolution) to generate test inputs. GA starts with an initial set or *population* of random inputs (*individuals*) and iteratively evolves them to find the best one that achieves some objective. GA uses a *fitness function* to evaluate the quality of the individuals and *selects* the best ones to survive and reproduce (i.e., survival of the fittest). GA then applies genetic operators to create individuals representing the new population in the next *generation*. This process continues until a *stopping criterion* is met.

The main genetic operators in a GA are

1. *Crossover (xover)*: combines two individuals or *parents* to create new ones. Common xover methods include single-point, two-point, and uniform crossover. xover rate is the probability of applying xover, and typically is high (e.g., 0.8 or 80% chance).

2. *Mutation*: randomly changes some elements of an individual. Common mutation methods include creating a random element, swapping two element, and flipping (e.g., negative to positive, 0 to 1, etc). Mutation rate is the probability of doing mutation, and typically is low (e.g., lower than 0.1 or 10% chance).

GA Template The following is a template for GA:

```python
def ga(...):
    # Initialize the population
    pop = gen_pop(...) # generate a random population

    # Evaluate the fitness of each individual
    fitness = eval_fitness(...)

    # Repeat until stopping criterion is met
    while not stopping_criterion(...):
        # Select the best individuals
        parents = select(...)

        # Apply genetic operators to create new individuals
        offspring = crossover(...)
        offspring = mutate(...)

        # Replace the old population with the new one
        pop = offspring

        # Evaluate the fitness of the new population
        fitness = eval_fitness(...)

    # Return the best individual
    best = select_best(...)
    return best
```

6.3 In-class Exercise: GA list sum

In this assignment you have two tasks. First, you will *implement a GA* that evolves a population of lists of integers to find a list whose sum is a given target sum. Next, you will write a *short report* that explains your GA and how you tested it.

Task 1: GA implementation You can use the GA template in §6.2.2 for this task. You can also use the following GA code for counting 0's as example. You will likely need to modify this code to fit your needs as the problem and objective are very different.

Specifically, you will implement the following GA components. The *signatures* below for the functions are just suggestions. You can modify them as needed.

1. *Generate* an initial, random population of lists of integers. The length of the popular and individual lists are given as input. The integers in the list should be between a specified range (e.g., -100,100)
   `gen_pop(pop_size:int, indv_size:int, min_val:int, max_val:int) -> list[list[int]]`

2. Write a *fitness function* that computes the fitness score based on how close the sum of the list is to the target sum. Closer is better (e.g., if the target is 99, then a list whose sum is 99 should have the "perfect score" while a list whose sum is 90 has a better score than a list whose sum is 50). Note that you must also take account of negative numbers and sums.
   `get_fitness(indiv:list[int], target_sum:int)`

3. Write a *selection function* that selects the best individuals based on their fitness scores. You can use any selection method you like (e.g., roulette wheel, tournament selection). You should look these up to understand how they work.
   `select(pop:list[list[int]], fitness:list[int], pop_size:int) -> list[list[int]]`

4. Write a *crossover* function that takes two parents and creates two offsprings using *single-point* crossover (i.e., pick a random point and swap).
   `def crossover(parent1: list[int], parent2: list[int], rate:float) -> tuple(list[int], list[int])`

5. Write a *mutation* function that randomly changes a few elements of an individual based on a mutation rate.
   `def mutate(indiv:list[int], rate:float, min_val:int, max_val:int) -> list[int]`

6. Write a *stopping criterion* function that stops the GA when it found an individual whose sum is the target number.
   `def stopping_criterion(best_fitness) -> bool`

7. Write the *main* genetic algorithm that uses all the above functions and returns the best individual and its fitness.
   `def ga(pop_size:int, indv_size:int, xover_rate:float, mut_rate:float, min_val:int, max_val:int, target:int): -> (list[int], float)`

8. Your GA should print out the best individual, its sum, and its fitness score at each generation (iteration).

9. Your GA has the various parameters (e.g., inputs to the `ga`). You should play with them to find values that work best. You can start with these values: `pop_size=100, indv_size=10, xover_rate=0.8, mut_rate=0.1, min_val=-100, max_val=100, target=1000`.

10. Time your GA. You can use Python's `time` module for this.

11. Submit your code with a clear `README` instruction on how to run your GA and test it. You should also submit screenshots of your GA running (you don't need to show all the generations, just a few to show that it is working).

Task 2: Write and submit a short report

1. Write a report explaining your GA. More specifically for each of the above task, explain that you did (e.g., how do you generate the population, how do you compute the fitness, etc).

2. Explain the parameters used and how they affect the performance of the GA (e.g., the time it took). For example, how does the population size affect the performance? crossover and mutation rates? etc.

6.4 Whitebox Testing with Symbolic Execution

In contrast to black-box testing (§6.1) that does not look at the code, *white-box testing* reasons about the program using its source code, allowing it to find bugs that escape black-box testing. For example, in the `square` function in §6.1, by analyzing the code we can see that the program has a bug on input 123 because it returns -1 instead of 123^2.

Symbolic execution is a white-box testing technique to find inputs causing the program to take some interesting paths (e.g., that result in a bug). Symbolic execution runs the program with *symbolic inputs* instead of concrete ones (e.g., x instead of 5) and tracks program's state. It uses constraints or logical formulae to represent the program's *path conditions* (PCs) over symbolic inputs that would reach the interesting paths or locations. It then uses a constraint solver, e.g., a SAT or SMT solver, to find the concrete input that satisfies the path condition (and thus reach the desired path or loc).

```
void foo(int a, int b, int c){
    // l0
    int x=0, y=0, z=0;
    // l1
    if(a) {
        x = -2;
        // l2
    }
    // l3
    if (b < 5) {
        // l4
        if (!a && c) {
            y = 1;
            // l5
        }
        z = 2;
        // l6
```

```
    }
    // l7
    assert(x + y + z != 3);
}
```

Example We execute this program with symbolic inputs a, b, c. At each location l, we keep track of two things: the path condition (PC) to reach l and the program state (PS), consisting values of variables at l.

At $l0$, the PC is always true (i.e., T) and the PS is $\{\}$, i.e., nothing yet. At $l1$, PC is T and PS is $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$. The PC for $l2$ is a with PS $\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$.

At $l3$ we have two paths reaching it. The PC for the first path is a with PS $\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$. The PC for the second path is $\neg a$ with PS $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$. At $l4$ we have two paths: 1st path has PC $a \wedge b < 5$ with PS $\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$, and 2nd path has PC $\neg a \wedge b < 5$ with PS $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$. At $l5$ we have 2 paths, at $l6$ we have 4 paths, and so on as shown in Tab. 6.1

Constraint Solving After obtaining the PCs, we can use a constraint solver like Microsoft Z3 solver to find the concrete inputs reaching to a specific location by solving the corresponding PC. For example, a solution to the PC $a \wedge b < 5$ of $l4$ is $a = 1, b = 3$, which means the program reaches $l4$ with $a = 1, b = 3$.

Assertions Assertions indicate what the programmer believes to be true at a certain point in the program. If an assertion fails, it indicates a bug in the program. For example, the assertion in this example would fail when we have $x + y + z = 3$.

To make the reasoning easier, we can convert the statement `assert(c)` to

```
if(!c){
    // failure loc
    assert(0);
}
```

This allows us to use symbolic execution as usual compute the PC to reach the failure location.

None-Symbolic Values Observe our assertion here involves the non-symbolic values x, y, z, which we keep track in the program state PS. It is common in symbolic execution where we have to reason both symbolic and non-symbolic values (hence we keep track of both PC and PS).

Thus we essentially want to check if any of the paths can reach the assertion location has $x + y + z = 3$. In this example, according to Tab. 6.1, we see that the path reaching $l7$ with PC $\neg a \wedge b < 5 \wedge (\neg a \wedge c)$ with PS $\{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$ would satisfy $x + y + z = 3$. Using a constraint solver, we can find the a concrete input $(a = 0, b = 3, c = 1)$ that would reach this path and fail the assertion.

Tab. 6.1: Symbolic Execution Example

Loc (l)	**Path Condition** (PC)	**Program State** (PS)
$l0$	T	$\{\}$
$l1$	T	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$
$l2$	a	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l3$	a	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l3$	$\neg a$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$
$l4$	$a \wedge b < 5$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l4$	$\neg a \wedge b < 5$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$
$l5$	$a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto -2, y \mapsto 1, z \mapsto 0\}$
$l5$	$\neg a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto 0, y \mapsto 1, z \mapsto 0\}$
$l6$	$a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto -2, y \mapsto 1, z \mapsto 2\}$
$l6$	$a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 2\}$
$l6$	$\neg a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$
$l6$	$\neg a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 2\}$
$l7$	$a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto -2, y \mapsto 1, z \mapsto 2\}$
$l7$	$a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 2\}$
$l7$	$\neg a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$
$l7$	$\neg a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 2\}$
$l7$	$a \wedge b \geq 5$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l7$	$\neg a \wedge b \geq 5$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$

Chapter 7

Fault Localization

Fault localization is a debugging process of isolating the bug in the program. It is crucial for developers to understand and fix the bug. Programmers use various techniques, including `printf` debugging where they output variable values to analyze the bug. Professional developers use built-in debugger tools in IDEs to step through and pause code execution to inspect variables and program states.

Here we will discuss two popular fault localization techniques: *statistical debugging* and *delta debugging* to localize code and inputs that likely contain the bug.

7.1 Statistical Debugging

Statistical debugging is a white-box technique that uses statistics to find bugs in code, i.e., *fault localization*. It collects program execution traces, e.g., which lines of code were executed, how many times, etc, and uses this data to find the lines that likely contain the bug. For example, if a line l is executed many times when the program fails but not when it runs correctly, then l is likely the bug.

Tarantula is a popular statistical debugging technique that computes a suspicious score for each line of code based on the number of times it was executed when the program failed and when it passes (gives expected behavior). The formula is:

$$\text{Suspiciousness}(l) = \frac{\text{Failed}(l)/\text{TotalFailed}}{\text{Failed}(l)/\text{TotalFailed} + \text{Passed}(l)/\text{TotalPassed}},$$

where $\text{Failed}(l)$ is the number of times line l was executed when the program failed, $\text{Passed}(l)$ is the number of times line l was executed.

Consider the following code:

```python
def median(x, y, z):
    print("input ", x, y, z)    # line 1
    m = z                       # line 2
    if y < z:                   # line 3
        if x < y:               # line 4
```

Tab. 7.1: Statistical Debugging with Tarantula scoring metrics. ‘x‘ means the line was hit (executed) and ‘-‘ means it was skipped (not executed).

line	1	2	3	4	5	6	7	8	9	10	11	12	13	Pass/Fail
t1 (3,3,5)	x	x	x	x	-	x	x	-	-	-	-	-	x	P
t2 (1,2,3)	x	x	x	x	x	-	-	-	-	-	-	-	x	P
t3 (3,2,2)	x	x	x	-	-	-	-	x	x	x	-	-	x	P
t4 (5,5,5)	x	x	x	-	-	-	-	x	x	-	x	-	x	P
t5 (1,1,4)	x	x	x	x	-	x	x	-	-	-	-	-	x	P
t6 (5,3,4)	x	x	x	x	-	x	-	-	-	-	-	-	x	P
t7 (3,2,1)	x	x	x	-	-	-	-	x	x	x	-	-	x	F
t8 (2,1,3)	x	x	x	x	-	x	x	-	-	-	-	-	x	F
t9 (5,4,2)	x	x	x	-	-	-	-	x	x	x	-	-	x	F
t10 (5,2,6)	x	x	x	x	-	x	x	-	-	-	-	-	x	F
Score (Tara)	0.5	0.5	0.5	0.42	0.	0.5	0.6	0.6	0.6	0.75	0.	0.	0.5	

```
            m = y                # line 5
        else if x < z:           # line 6
            m = y                # line 7, %bug, should be z
    else:                        # line 8
        if x > y:                # line 9
            m = z                # line 10, %bug, should be y
        else if (x > z):         # line 11
            m = x                # line 12
    print("median is ", m)       # line 13
```

We now run the program on tests and collect the number of times each line was executed when the program failed and when it passed. For example, for a test `t1` with input `(3,3,5)`, the program passes (shows median is 3) and hits lines 1, 2, 3, 4, 6, 7, 13 and skips lines 5, 8, 9, 10, 11, 12. For a test `t9` with input `(5,4,2)`, the program fails (shows median is 2 instead of 4) and hits lines 1, 2, 3, 8, 9, 10, 13 and skips lines 4, 5, 6, 7, 11, 12. It is easy to see that every test will hit lines 1 and 13.

After we do this for all tests (e.g., 10 tests `t1, t2, ..., t10`), we can compute the suspiciousness score for each line using the Tarantula formula. The higher the score, the more likely the line contains the bug. The following table shows the number of times each line was executed when the program failed and when it ran correctly over several test runs.

We can now compute the suspiciousness score for each line using the Tarantula formula. Here we have 10 tests with 6 passing and 4 failing. For example, the suspiciousness score for line 4 is: $2/4/(2/4 + 4/6) = 0.42$. The score for line 5 is $0/4/(0/4 + 1/6) = 0$, i.e., this line is definitely not buggy. The score for line 7 is $2/4/(2/4 + 2/6) = 0.6$, line 10 is $2/4/(2/4 + 1/6) = 0.75$. Note that scores for lines 1 and line 13, which are always executed, are $4/4/(4/4 + 6/6) = 0.5$. The score for line 12, which was not executed in any test runs, is 0. (if it never runs, it should not be responsible for any issue).

Tab. 7.2: Delta Debugging Example 1.

Failing Input	Split	Remove 1st	Remove 2nd
abcdef*h	abcd/ef*h	abcd: F	-
ef*h	ef/*h	ef: F	-
*h	*/h	*: P	h: F
*	-	-	-

7.2 Delta Debugging (DD)

While statistical debugging (§7.1) aims to localize faults in the code, DD focuses on finding the smallest input that triggers the issue. DD aims to *minimize* a *failing* input (e.g., causing the program to crash or producing some interesting behavior). It is useful for debugging and finding a simpler input that is still interesting. DD works by repeatedly splitting the input into smaller parts and checking if they still trigger the issue. When using DD, you will need to provide an *oracle* that checks if the input P is interesting (e.g., causing a crash).

Example 1 *Oracle*: program fails (is interesting) whenever input contains an asterisk (*).

Tab. 7.2 shows the steps of DD. First, we start with the original input `abcdef*h` and split it into two parts `abcd/ef*h`. Removing the first part `abcd` still fails, so we remove it. We then repeat DD on the new input `ef*h` and split it into two parts `ef/*h`. Removing `ef` still fails, so we remove it and have the new input `*h`, which is then split into `*/h`. Removing the first part `*` passes, so we keep it and remove the second part `h`, which fails and is removed. The new input is now `*`, which cannot be split further and is the smallest failing input.

This example does not show the case when the split results in all parts passing. In that case, DD would increase the split size (e.g., split into 4 parts instead of 2) and repeat the process.

Example 2 *Oracle*: program fails whenever input contains two asterisks (**).

Tab. 7.3 shows the steps. We first split the input `*abcdef*` into 2 parts, which both pass. Thus, we increase the granularity and split the input into 4 parts `*a`, `bc`, `de`, `f*`. Removing the 1st part result in a pass and so we keep it and try removing the 2nd part `bc`, which fails and thus we can remove `bc` to get the new input `*adef*`. This keeps going until we find the smallest failing input `**`. Note even when our failing input is `**`, we still continue applying DD to it as we did not know that it would be the smallest failing input.

Tab. 7.3: Delta Debugging Example 2.

Failing Input	Split	Remove 1st	Remove 2nd
abcdef	*abc / def*	*abc: P	def*: P
abcdef	*a / bc / de / f*	*a: P	bc: F
adef	*ad / ef*	*a: P	ef*: P
adef	*a / de / f / *	*a: P	de: F
af	*a / f*	*a: P	f*: P
af	* / a / f / *	*: P	a: F
f	*f / *	*f: P	*: P
f	* / f / *	*: P	f: F
**	* / *	*: P	*: P
**	-	-	-

7.3 Exercises

7.3.1 Statistical Debugging: Tarantula vs. Ochiai

Ochiai is another popular metrics for statistical debugging. Its formula is

$$\text{Suspiciousness}(l) = \frac{\text{Failed}(l)/\text{TotalFailed}}{\sqrt{\text{Failed}(l)/\text{TotalFailed} + \text{Passed}(l)/\text{TotalPassed}}}$$

1. Compute the Ochiai score for the lines in the table above.
2. Explain the differences between Tarantula and Ochiai scores. Which one do you think is better? Why?

7.3.2 Statistical Debugging: M Metrics

The **M** metrics to compute the suspiciousness of a line l is calculated as follows:

$$\text{Suspiciousness}(l) = \frac{\text{Failed}(l)}{\text{Failed}(l) + \text{Passed}(l)}$$

Apply this metrics to compute the suspiciousness scores for the lines in Tab. 7.1.

7.3.3 Delta Debugging Practice

- Apply DD to the input string `"*hello*world*"` to find the smallest failing input. The oracle is that the input fails whenever it contains `"oo"`.
- What is best-case complexity of DD? Give an example of an input that would take the most steps to find the smallest failing input.

- What is the worst-case complexity of DD? Give an example of an input that would take the most steps to find the smallest failing input.

7.3.4 Delta Debugging (DD) Implementation

In this exercise you will implement the DD technique. Your DD will take an input string and an oracle that decides if the input is interesting (e.g., has a bug) or not. More specifically, `oracle(s:str)->bool` takes in a string `s` and returns T (`s` has a bug) or F (`s` has no bug). The goal of your DD is to reduce the original input to become *minimal* but is still *interesting* (i.e., a minimal input that still has the bug).

- You can be as creative as you want with your DD, however it must not run for too long (e.g., try to split the input in to parts that are power of 2 as discussed in class to reduce the input).
- You must provide *two* examples (inputs) to demonstrate your DD: 1 example where the DD takes a reasonably short time (few iterations) and 1 example where the DD takes a long time (e.g., 50 or more iterations).
- At each iteration, your DD should output the current input, its size, and whether it is interesting or not. You can also output the split number and parts, etc (the same way we did in class).
- **Code submission**: submit your DD code along with the examples you provided with a clear instructions and *screenshots* on how to run your DD and expected output.
- **Short Report**: write a short report describing your DD implementation. Explain how it works, how you tested it, and the results. Also briefly explain the examples you provided and why some take longer than others.

7.3.5 Hello SWE419

- Assume you're given a simple `hello SWE419` program in C

  ```c
  #include <stdio.h>
  int main() {
      printf("Hello SWE419!\n");
      return 0;
  }
  ```

  and an *oracle* that checks if the input C program is valid (compiles) and contains the word `SWE419`. If so, the oracle returns 0 (Fail) and 1 otherwise (Pass).

- Describe *in words* how you would apply DD to obtain the minimal C program that fails the oracle.

- Show what you get at the end. That is, show the minimal C code that DD would return.
- You **do not need** to show step by step like we did in class. Instead, just describe the steps you (the DD algorithm) would take to reduce the input, e.g., *"in the first step you split the program into two parts, then you remove the first part, run the oracle which fails/passes because ..., etc"*.
- Pay attention to the additional requirement of being a *valid* C program (e.g., needs the `#include <stdio.h>`, `int main()...` and `return 0` statements).

7.3.6 Symbolic Execution

Consider a simple function `f` below

```
void f(int y) {
  int z = y * 2;
  if (z == 12) {
    // L
    fail();
  } else {
    printf(''OK'');
  }
}
```

Use symbolic execution to compute the path condition (PC) and program state (PS) at location `L` (where the program fails). Then give an input that causes the program to fail at `L`.

7.3.7 Using the Z3 SMT Solver

Z3 is a theorem prover or SMT constraint solver developed at Microsoft. It has been employed in various software testing and reasoning tasks. Major tech companies including MS, Google, Amazon (AWS), NASA, etc use Z3 for a wide-range of projects to solve problems in software, security, and AI. For example, Amazon AWS runs *billions* of Z3 queries *everyday*[1]. In this exercise you will be introduced to Z3 and use it for various reasoning tasks.

Installation and Setup To have Z3 to work with Python, you can install it various methods including `pip` or `homebrew` (Mac) or `apt-get` (Linux). You can search online for the installation method that works best for your system.

To ensure Z3 is installed correctly, you can try to `import z3` in Python. If you do not get an error, then Z3 is installed correctly.

[1]https://www.amazon.science/blog/a-billion-smt-queries-a-day

Your Tasks You write Python code using Z3 for various problems below. You can use the Z3 API or use Google to find the relevant information like Z3's method names, e.g., `z3.solve(...)` for satisfiability checking and `z3.prove(...)` for proving or validy checking.

1. Boolean Logic
   (a) Create boolean variables `p, q, r`
   (b) Check if the formula $p \lor q$ is *satisfiable*.
   (c) Prove transitivity, i.e., show that $(p = q \ \land \ q = r) \Rightarrow p = r$.
   (d) Show that $(p \land q) \Rightarrow p$ is a *tautology* (valid).
   (e) Prove that $p \Rightarrow q$ is equivalent to $\neg p \lor q$.
2. First-Order Logic over the Integers
   (a) Create integer variables `x, y, z`.
   (b) Show that $x > 3 \Rightarrow x > 2$
   (c) Prove that $x > 3 \land y > 3 \Rightarrow x + y > 6$
   (d) Solve for x, y such that $x + y = 10$ and $x < 0$
   (e) Show the transitive $x > y \land y > z \Rightarrow x > z$
   (f) Confirm that $x > 2$ and $x < 2$ is *unsatisfiable*.
   (g) Prove that $x \leq y$ and $x \geq y$ is equivalent to $x = y$.
3. Using symbolic execution with Z3 to find inputs leading to each location in the program below. For this you will do 2 things: (i) create a table like Tab. 6.1 to show the path conditions and program states, and (ii) provide the Python Z3 code to solve for the inputs (e.g., if the PC is $x > 0$ and $y < 0$, you can just use `z3.solve(PC)` to get the inputs and show the values of x, y leading to that location.

```
void bar(int x, int y) {
int a = 0, b = 0;
int z = x + y + a;

// L1
if (x > 5) {
    int w = z - x + b;
    // L2
    if (w < 3) {
        // L3
    } else {
        // L4
    }
} else {
    int v = z + 2;
    // L5
    if (v > y) {
```

```
            // L6
        } else {
            // L7
        }
    }
}
```

What to submit You will submit a Python file with the Z3 code for the above problems. As usual be sure to include instructions (could be comments in the Python code) and screenshots on how to run the code.

You will also submit a text (or doc or pdf) file showing the symbolic execution table and the inputs leading to each location in the program.

Part III

Program Verification

In §6 we focus on testing to find bugs and find counterexample inputs to show the presence of bugs in programs. Here we look at *verification* techniques to *prove the absense of bugs*. Like whitebox approaches (§6.4), verification techniques are a form of static analysis that reason about program behaviors without actually running the program (e.g., testing). Verification complements testing and can do more thorough analysis as it considers all possible program inputs.

Chapter 8

Hoare Logic

Hoare Logic is a formal system used to reason about the correctness of programs. It was introduced by C.A.R. Hoare in the late 1960s and is a fundamental technique in program verification. Specifically, given a program S and its specification consisting of a precondition P and postcondition Q, we can use Hoare Logic to prove that the program satisfies its specification.

In Hoare logic, we will learn fundamental concepts such as *Hoare Tripples*, *weakest precondition*, and *verification conditions*. In particular, we will learn about *program invariants* such as loop invariants that are crucial for proving the correctness of programs with loops.

8.1 Hoare Tripple

Given a program S and its specification consisting of a precondition P and postcondition Q, we first represent them as a *Hoare Tripple*:

$$\{P\}\ \texttt{S}\ \{Q\},$$

which reads: assuming P holds before executing S, and S is executed successfully, then Q holds. If the Hoare Triple is *valid* (true), then the program S satisfies its specification P, Q.

> **Formatting**: Note that S is represented as a single statement or a sequence of statements, and P, Q are logical expressions or formulae. We differentiate between program statements (code) and logical expressions (which include formulae, conditions, constraints and evaluate to true) by using different fonts and styles. For example, `z:=x+y;`, `if (z >= 9)`, `a && b` are program code, while $z \leq 9$ and $a \wedge b$ are expressions. We also use T and F to represent true and false, respectively.

Examples

1. the Hoare Triple $\{x = 5 \wedge y > 2\}$ `z := x + y; z := z + 2` $\{z > 9\}$ is valid because assume $x = 5$ and $y > 2$, then after executing `z := x + y; z := z + 2`, we do have $z > 9$.

2. Consider a program with a single assignment statement `x:= 5;`.

   - The The Hoare triple $\{T\}$ `x := 5` $\{x > 6\}$ is *not* valid because the postcondition $x > 6$ is not satisfied after executing `x := 5`.
   - These tripples are valid

     (a) $\{T\}$ `x := 5` $\{x = 5$ or $x = 6$ or $x > 6\}$
     (b) $\{T\}$ `x := 5` $\{x > 1\}$
     (c) $\{T\}$ `x := 5` $\{x = 5\}$

     Moreover, the postcondition $x = 5$ is **strongest** because it is more precise than $x > 1$ and $x = 5$ or $x = 6$ or $x > 6$. In general, we aim for the strongest postcondition (see §1.2.2).

3. Consider another program `z := x / y`.

   - These are valid Hoare triples:

     (a) $\{x = 1 \wedge y = 2\}$ `z := x / y` $\{z < 1\}$
     (b) $\{x = 2 \wedge y = 4\}$ `z := x / y` $\{z < 1\}$
     (c) $\{0 < x < y \wedge y \neq 0\}$ `z := x / y` $\{z < 1\}$

     Moreover, the precondition $0 < x < y \wedge y \neq 0$ is the **weakest** precondition (i.e., it imposes the least constraints). In general, we seek the weakest precondition (see §1.2.1).

   - These are *invalid*:

     (a) $\{x < y\}$ `z := x / y` $\{z < 1\}$
     (Counterexample: $x = -1$, $y = 0$. Executing `z := x / y` results in a division-by-zero exception, so $z < 1$ does not hold.)
     (b) $\{x = 0\}$ `z := x / y` $\{z < 1\}$
     (Counterexample: $x = 0$, $y = 0$.)
     (c) $\{y \neq 0\}$ `z := x / y` $\{z < 1\}$
     (Counterexample: $x = 2$, $y = 1$.)
     (d) $\{x < y \wedge y \neq 0\}$ `z := x / y` $\{z < 1\}$
     (Counterexample: $x = -2$, $y = -1$.)

Partial and Total Correctness

A Hoare Tripple only requires the program to satisfy its post condition *only when* that program executes successfully, i.e., it *terminates.* This is known as *partial correctness*, which assumes the program terminates. In contrast, *total correctness*, which *requires* the program to terminate. Total correctness is much harder to achieve because proving program termination, i.e., the *halting problem* is in general an undecidable problem. Hoare logic and most other verification approaches use partial correctness as it is easier to prove.

8.2 Verifying Programs using Hoare Logic

So far we have manually checked the validity of Hoare triples. However, for more complex programs, we need a systematic way to do it automatically. Hoare logic provides such an automatic way to verify programs using logical reasoning.

To prove the Hoare Tripple `{P} S {Q}` is valid, we first compute the *weakest precondition* (WP) of the program `S` with respect to its postcondition `Q` (§8.2.1). We then form a *verification condition* to check that given precondition P implies the computed WP (§8.3). If the VC is valid, the Hoare Triple is valid and the program satisfies its specification.

8.2.1 Computing Weakest Preconditions

The weakest precondition (WP) of a program S with respect to a postcondition Q is the *least restrictive* or weakest condition that ensures Q after executing S. The WP is computed by working backwards from Q using the program S.

Tab. 8.1 defines the function $\texttt{wp}(\texttt{S}, Q)$ that computes the WP, represented as a logical formula, of diferent program statements S with respect to a postcondition Q.

Assignment The *assignment* `x :=` E statement assigns the expression E to a variable `x`. The WP of an assignment `wp(x:=E,Q)` is obtained by substituting all occurrences of `x` in `Q` with the expression `E`.

$$\texttt{wp(x:=E, Q)} = Q[x/E] \tag{8.1}$$

Examples

1.
$$\begin{aligned}&\texttt{WP}(x := 3, x + y = 10)\\&= 3 + y = 10\\&= y = 7\end{aligned}$$

Thus, we have $\{y = 7\}\ x := 3\ \{x + y = 10\}$

Algorithm 1: Weakest Precondition Computation

Input : A program statement S and a postcondition Q
Output: The weakest precondition $WP(S, Q)$

switch S **do**
  case S *is* ***skip*** **do**
    return Q
  case S *is an assignment* $x := E$ **do**
    return $Q[x/E]$
  case S *is a sequence* $S_1; S_2$ **do**
    return `WP` $(S_1,$ `WP` $(S_2, Q))$
  case S *is a conditional statement* ***if*** B ***then*** S_1 ***else*** S_2 **do**
    return $B \Rightarrow \texttt{WP}(S_1, Q) \wedge (\neg B \Rightarrow \texttt{WP}(S_2, Q))$
  case S *is a loop* ***while*** B ***do*** S **do**
    `// Require a loop invariant` I
    return $I \wedge (I \wedge B \Rightarrow \texttt{WP}(S, I) \wedge (\neg I \wedge B \Rightarrow Q)$
  case *otherwise* **do**
    return error: unsupported statement

Tab. 8.1: Weakest Precondition Rules.

Statement	S	$\texttt{wp}(S, Q)$	Notes
Assignment	`x:=E`	`Q[E/x]`	Replace all `x`'s with `E` in `Q`
Sequence	`S1; S2`	$\texttt{wp}(\texttt{S1}, \texttt{wp}(\texttt{S2}, Q))$ $\texttt{wp}(\texttt{[]}, Q) = Q$	Recursively compute the WP Base case when the seq is empty
Conditional	`if` b `then S1 else S2`	$b \Rightarrow \texttt{wp}(\texttt{S1}, Q) \wedge$ $\overline{b} \Rightarrow \texttt{wp}(\texttt{S2}, Q)$	Produce disjunction representing the two branches
While	`while` b `do S`	$I \wedge$ $I \wedge b \Rightarrow \texttt{wp}(S, I) \wedge$ $I \wedge \overline{b} \Rightarrow Q$	User supplied loop inv I

2.

$$\begin{aligned}\texttt{WP}(x := 3, x + y > 0)& \\ &= 3 + y > 0 \\ &= y > -3\end{aligned}$$

Thus, we have $\{y > -3\}$ $x := 3$ $\{x + y > 0\}$

3.

$$\texttt{wp}(x := 3, y > 0) = y > 0 \text{ \# no x in y > 0, so result is just y > 0}$$

Thus, we have $\{y > 0\}$ `x := 3` $\{y > 0\}$

List of Statements The WP for a list (sequence or block) of statements is defined *recursively* as the WP of the first statement followed by the WP of the rest of the statements.

$$\texttt{wp}(\texttt{[S1; S2;]}, Q) = \texttt{wp}(\texttt{S1}, \texttt{wp}(\texttt{S2}, Q)) \tag{8.2}$$

$$\texttt{wp}(\texttt{[]}, Q) = Q \tag{8.3}$$

Examples

$$\begin{aligned}&\texttt{wp}(\texttt{[x:=x+1; y:=y*x]}, y = 2z) \\ &= \texttt{wp}(x := x + 1, \texttt{wp}(\texttt{[y:=y*x]}, y = 2z)) \\ &= \texttt{wp}(x := x + 1, yx = 2z) \\ &= y(x + 1) = 2z\end{aligned}$$

Thus, we have $\{y(x + 1) = 2z\}$ `x:=x+1; y:=y*x;` $\{y = 2z\}$

Conditional

The WP of a conditional statement `if-then-else` combines the WPs of its two branches.

$$\texttt{wp}(\texttt{if } b \texttt{ then } S1 \texttt{ else } S2, Q) = (b \Rightarrow \texttt{wp}(S1, Q)) \wedge (\neg b \Rightarrow \texttt{wp}(S2, Q)) \tag{8.4}$$

Examples

- $$
\begin{aligned}
&\texttt{wp}(\text{if } x > 0 \texttt{ then } y := x + 2 \texttt{ else } y := y + 1, y > x)\\
&= (x > 0 \Rightarrow \texttt{wp}(\texttt{y := x + 2}, y > x)) \ \wedge \ (x \leq 0 \Rightarrow \texttt{wp}(y := y + 1, y > x))\\
&= (x > 0 \Rightarrow x + 2 > x) \ \wedge \ (x \leq 0 \Rightarrow y + 1 > x)\\
&= (x > 0 \Rightarrow 2 > 0) \ \wedge \ (x \leq 0 \Rightarrow y + 1 > x)\\
&= T \ \wedge \ (x \leq 0 \Rightarrow y + 1 > x)\\
&= x \leq 0 \Rightarrow y + 1 > x
\end{aligned}
$$

- $$
\begin{aligned}
&\texttt{wp}(\texttt{if } x > 0 \texttt{ then } y := x \texttt{ else } y := 0, y > 0)\\
&= (x > 0 \Rightarrow \texttt{wp}(\texttt{y:=x;}, y > 0)) \wedge (x \leq 0 \Rightarrow \texttt{wp}(\texttt{y:=0;}, y > 0))\\
&= (x > 0 \Rightarrow x > 0) \wedge (x \leq 0 \Rightarrow 0 > 0)\\
&= \neg(x > 0) \vee (x > 0) \wedge \neg(x \leq 0) \vee F\\
&= T \wedge x > 0\\
&= x > 0
\end{aligned}
$$

Note that instead of using implication $\Rightarrow$, which might be confusing to some, we can use $\neg$ and $\vee$. This is because $a \Rightarrow b$ is equivalent to $\neg a \vee b$ (implication rule, Tab. B.2). Thus, the above can be written as:

$$
\texttt{wp}(\texttt{if } b \texttt{ then } S1 \texttt{ else } S2, Q) = (\neg b \vee \texttt{wp}(S1, Q)) \ \wedge \ (b \vee \texttt{wp}(S2, Q)) \tag{8.5}
$$

Loop Unlike other statements where we have rules to compute WPs *automatically*, to obtain the WP of loop, we need to *manually* supply a **loop invariant** I (§C). Moreover, I must be strong enough to prove postcondition `Q` holds when the loop terminates. Finding sufficiently strong loop invariants is a challenging problem with many research works dedicated to it.

Assume that a loop invariant I is given, the WP of a loop[1] `while` b `do S` with the postcondition `Q` is

$$
\begin{aligned}
&\texttt{wp}(\texttt{while } [I] \ \ b \texttt{ do } S, Q)\\
&= I \ \wedge \ (I \wedge b) \Rightarrow \texttt{wp}(S, I) \ \wedge \ (I \wedge \neg b) \Rightarrow Q
\end{aligned} \tag{8.6}
$$

Observe that the WP for loop consists of 3 conjuncts:

[1] A `while` loop is often used because other kinds of loops can be converted to it.

```
while(True){
  # [I]: loop invariants here
  if(!(i < N)) break;
  i := N;
}
{i == N}   // postcondition Q
```

Fig. 8.1: A simple `while` loop

1. I : the loop invariant (should hold the first time the loop is entered)
2. $(I \wedge b) \Rightarrow I$: (entering the loop body because b is true) and I is preserved after each loop body execution
3. $(I \wedge \neg b) \Rightarrow Q$: (exiting the loop because b is false), when exiting the loop, the post condition holds

The first two ensure that I holds when entering the loop and is preserved after each loop body execution. The last conjunct ensures that the the postcondition Q holds when the loop terminates.

Example Let's compute the wp of the loop in Fig. 8.1 using these loop invariants:

1. $i \leq N$
2. $N \geq 0$
3. T (true) (always a loop invariant, though very weak and likely useless)

Using loop invariant $i \leq N$

$$
\begin{aligned}
&\texttt{wp}(\texttt{while}[i \leq N](i < N)\ \texttt{i := N}, i = N) \\
&= i \leq N \wedge \\
&\quad (i \leq N \wedge i < N) \Rightarrow \texttt{wp}(\texttt{i := N}, i \leq N) \wedge \\
&\quad (i \leq N \wedge \neg(i < N)) \Rightarrow i = N
\end{aligned}
$$

Let's deal with these 3 conjuncts one by one, easier that way

1. By definition I must hold when first entering the loop

$$i \leq N$$

2. This is the case when we assume the loop invariant holds before the loop body *and* the loop guard is true (allowing the program to enter the loop), then after executing the loop body, the loop invariant should be preserved.

$$\begin{aligned}(i \leq N \land i < N) &\Rightarrow \texttt{wp}(\texttt{i := N}, i \leq N)\\ &= (i \leq N \land i < N) \Rightarrow N \leq N\\ &= i < N \Rightarrow T\\ &= T\end{aligned}$$

   This becomes T because assuming the loop invariant $i \leq N$ and the loop guard $i < N$ are true, the body of the loop sets i to N thus preserves $I : i \leq N$ (i.e., $i = N$ satisfies $i \leq N$).

3. This is the case when we assume the loop invariant $i \leq N$ holds before the loop body *but* the loop guard $i < N$ is false, i.e., $i \geq N$, then we do not enter the loop and should get the postcondition $i = N$.

$$\begin{aligned}(i \leq N \land \neg(i < N)) &\Rightarrow i = N\\ &= (i \leq N \land i \geq N) \Rightarrow i = N\\ &= i = N \Rightarrow i = N\\ &= T\end{aligned}$$

   This becomes T because $i \leq N$ and $i \geq N$ is equivalent to $i = N$, which is the postcondition $i = N$.

Thus, the WP when using the loop invariant $i \leq N$ is itself

$$i \leq N.$$

Using loop invariant $N \geq 0$

$$\begin{aligned}&\texttt{wp}(\texttt{while}[N \geq 0](i < N)\ \texttt{i := N}, i = N)\\ &= N \geq 0 \land\\ &\quad (N \geq 0 \land i < N) \Rightarrow \texttt{wp}(\texttt{i := N}, N \geq 0) \land\\ &\quad (N \geq 0 \land \neg(i < N)) \Rightarrow i = N\end{aligned}$$

Again, let's reason through these three conjuncts one by one.

1. First time we enter the loop

$$N \geq 0$$

2. When we enter the loop body

$$\begin{aligned} &(N \geq 0 \wedge i < N) \Rightarrow \texttt{wp}(\texttt{i := N}, N \geq 0) \\ =\ &(N \geq 0 \wedge i < N) \Rightarrow N \geq 0 \\ =\ &T \end{aligned}$$

   This becomes T because the body of the loop sets i to N and does not modify N, thus the loop invariant $N \geq 0$ is preserved because we already assume that it and the loop guard $i < N$ holds.

3. When we do not enter the loop body

$$\begin{aligned} &(N \geq 0 \wedge \neg(i < N)) \Rightarrow i = N \\ =\ &(N \geq 0 \wedge i \geq N) \Rightarrow i = N \end{aligned}$$

   This looks a bit hard to simplify so we leave it as is.

Thus, the WP when using the loop invariant $N \geq 0$ is

$$N \geq 0 \ \wedge \ (N \geq 0 \wedge i \geq N) \Rightarrow i = N.$$

Simplification When computing the WP, we can simplify the expressions using the logic laws given in §B.2 so that they are easier to understand and reason about (e.g., $a \Rightarrow T$ is T through implication $\neg a \vee T$ and domination $a \vee T = T$). However, be careful when doing this as we can make mistakes and invalidate our entire reasoning. When in doubt, it's better to leave the expression as is.

Using loop invariant T

$$\begin{aligned} &\texttt{wp}(\texttt{while}[T](i < N)\ \texttt{i := N}, i = N) \\ =\ &T\ \wedge \\ &\quad (T \wedge i < N) \Rightarrow \texttt{wp}(\texttt{i := N}, T)\ \wedge \\ &\quad (T \wedge \neg(i < N)) \Rightarrow i = N \\ =\ &(i < N) \Rightarrow T \ \wedge \ i \geq N \Rightarrow i = N \\ =\ &i \geq N \Rightarrow i = N \end{aligned}$$

Thus, when using different loop invariants, we get different WPs. As we will see in §8.3, this will affect the validity of the Hoare triple. If we pick a good or strong invariant, we can prove the Hoare triple and thus the program. But if we pick a weak one, we might not be able to do so. This is why finding good loop invariants is a challenging problem.

8.3 Verification Condition

After computing the WP of a program S with respect to a postcondition Q, we check that the given precondition P implies the WP. This is known as the *verification condition* (VC) and written as

$$P \Rightarrow \texttt{wp}(\texttt{S}, Q) \tag{8.7}$$

If the VC is valid, the Hoare triple is valid and the program satisfies its specification. If the VC is invalid, then the Hoare triple is invalid and we cannot prove that the program is correct.

It is important to emphasize that not being able to prove the program is correct (i.e., the VC is invalid) *does not* mean the program is incorrect. It simply means we cannot prove it. Hoare logic and verification in general only prove program correctness, not incorrectness. This is in contrast to testing (§6) which can prove program incorrectness, i.e., by finding a counterexample input causing the bug, but not correctness (because we can't test all inputs).

Example

1. The VC of $\{x = 2\}$ `y := x + 1` $\{y > 2\}$ is $x = 2 \Rightarrow \texttt{wp}(\texttt{x := x + 1}, y > 2)$, which is $x = 2 \Rightarrow x > 1$. This VC is valid because $x = 2$ implies $x > 1$. Notice that we can use any precondition that is stronger than $x > 1$ as it is the weakest precondition. For example, $x = 100$, $x > 2$, etc. as preconditions will also make the VC (and thus Hoare triple) valid.

2. For the Hoare triple $\{x \geq 0\}$ `x := x + 1; y := 2 * x` $\{y \geq 2\}$, the WP is $\texttt{wp}(x := x+1, \texttt{wp}(y := 2*x, y \geq 2)) = \texttt{wp}(x := x+1, x \geq 1) = x \geq 0$. The VC $x \geq 0 \Rightarrow x \geq 0$ is valid.

3. The VC of $\{x = 1\}$ `y := x + 1` $\{y > 2\}$ is $x = 1 \Rightarrow x > 1$. This VC is *invalid* because $x = 1$ does not imply $x > 1$.

   Note that in this case the program is indeed incorrect because at the end $y = 2$ and therefore not greater than 2. However, we cannot make this conclusion because the VC is invalid.

4. Let's now verify the program with the while loop in Fig. 8.1. using the precondition $i = 0 \land N > 0$. Convince yourself that the program is correct with respect to this precondition and postcondition $i = N$. We will reuse the WPs computed using the given loop invariants $i \leq N$, $N \geq 0$, and T.

   (a) When using $I = i \leq N$, the computed WP is $i \leq N$. The VC $(i = 0 \land N > 0) \Rightarrow i \leq N$ is valid.

(b) When using $I = N \geq 0$, the computed WP is $N \geq 0 \wedge (N \geq 0 \wedge i \geq N) \Rightarrow i = N$. The VC $(i = 0 \wedge N > 0) \Rightarrow N \geq 0 \ \wedge \ (N \geq 0 \wedge i \geq N) \Rightarrow i = N$ is valid (because $N > 0 \Rightarrow N \geq 0$ and $N \geq 0 \wedge 0 \geq N \Rightarrow 0 = N$).

(c) When using $I = T$, the computed WP is $i \geq N \Rightarrow i = N$. The VC $(i = 0 \wedge N > 0) \Rightarrow (i \geq N \Rightarrow i = N)$ is valid (because $i \geq N$ is F, $F \Rightarrow \alpha$ is T (for any expression α), and $\alpha \Rightarrow T$ is T).

8.4 Summary

Hoare logic, espressed as Hoare triples, allows us to prove program correctness by computing the WP and forming and checking the VC. The WP is computed by working backwards from the postcondition using the program statements. The VC, which can be checked by SMT solver like Z3, shows that the given precondition implies the computed WP. If the VC is valid, the Hoare triple is valid and the program satisfies its specification. If the VC is invalid, then the Hoare triple is invalid and we cannot prove that the program is correct.

The main limitation of Hoare logic is that it requires the user to provide the correct loop invariants, which can be very difficult. Another limitation is that the computational cost of checking the VC, which can be expensive for complex programs. In the next chapter, we will learn about abstract interpretation, an alternative static analysis technique that use approximation to automatically and efficiently computes loop invariants to prove program correctness.

8.5 Exercises

8.5.1 Hoare Triples

Fill in `P,S,Q` to make the following Hoare Triples valid. Remember that we want the strongest postcondition `Q` and weakest precondition `P`.

1. $\{P\}$ `x:=3` $\{x = 8\}$
2. $\{P\}$ `x:= y - 3` $\{x = 8\}$
3. $\{x = y\}$ `S` $\{x = y\}$
4. $\{x = 10 \wedge y = 10\}$ `if x > 0 then y := x + 2 else y := x - 1` $\{Q\}$

8.5.2 Weakest Precondition and Verification Condition

Prove the following program using the *two* loop invariants $I_1 : i \leq 10$ and $I_2 : i \geq 0$. You must show your work including computing the weakest precondition, forming and checking verification condition, and explaining your reasoning (e.g., simplification).

```
{T}
int i = 0;
int j = 0;
while (i < 10) {
    i = i + 1;
    j = i;
}
{j = 10}
```

8.5.3 Prove a program with conditional

Prove the following program. You must show your work including computing the weakest precondition, forming and checking verification conditions, and explaining your reasoning (e.g., simplification).

```
{x >= 0}   // P
if (x > 0){
    y := x + 1
}
else{
    y := -x
}
{y >= 0}  // Q
```

Chapter 9

Abstract Interpretation

Abstract interpretation is another popular static analysis for proving program correctness. It computes an overapproximation (an abstraction) of the program behavior to prove that the program satisfies its specification.

The key difference between Hoare logic and abstract interpretation is that Hoare logic requires the user to provide the loop invariants, which can be challenging and error-prone. In contrast, abstract interpretation is fully *automatically* computes the loop invariants. However, abstract interpretation is an approximation, i.e., it might compute invariants that are too weak, and may not always be able to prove the program as we will see in this chapter. Nonetheless, abstract interpretation is powerful, automatic, and efficient, and is widely used in practice.

9.1 Abstraction Domains

To use abstract interpretation, we need to determine which *abstraction domain* is appropriate for our verification task. An abstract domain is a set of abstract values that represent the concrete values of the program. For example, to prove that a program does not have a division by zero error, we can use the abstraction of the sign of the divisor, in which we map the divisor to one of three values: positive, negative, or zero. If the divisor is positive or negative, then we can prove that there is no division by zero error.

9.1.1 The Parity Domain (Zero, Odd, Even)

Consider a simple abstraction over the natural numbers that maps each number to one of three abstract values:

$$\texttt{Concrete} \to \{\texttt{0}, \texttt{odd}, \texttt{even}\}.$$

For example:

$$5 \mapsto \texttt{odd}, \quad 4 \mapsto \texttt{even}, \quad 0 \mapsto \texttt{0}.$$

Abstract Transformer for Arithmetic Operations

In the Zero/Odd/Even domain, operations are reinterpreted over abstract values. For instance, consider addition:

$$5 + 2 = 7 \quad \text{transforms to} \quad \texttt{odd} + \texttt{even} = \texttt{odd}.$$

The abstract addition rules can be summarized in a table:

+	0	odd	even
0	0	odd	even
odd	odd	even	odd
even	even	odd	even

Tab. 9.1: Abstract Addition in the Zero/Odd/Even Domain

Similarly, for the greater-than relation (>) and integer division (//), abstract transformers are defined with partial information. For example:

>	0	odd	even
0	false	false	false
odd	true	indet.	indet.
even	true	indet.	indet.

Tab. 9.2: Abstract Greater-Than Operation

9.1.2 A Running Example: Multiplication via Abstract Interpretation

Consider the following Python-like pseudocode for multiplying two natural numbers A and B:

Listing 9.1: Multiplication via Repeated Addition

```
def mult(A, B):
    # Precondition: A >= 0 and B >= 0
    x = A
    y = B
    z = 0
    # L1: Initialization
    while True:
        # L2: Loop condition (x > 0)
        if not (x > 0):
            break
        if x % 2 == 1:  # when x is odd
            z = z + y
        x = x // 2
        y = y * 2
        # L3: Loop iteration update
    # L4: Program termination, determine abstract values of x, y, and z
    return z
```

In this example, we wish to determine the abstract values (over the Zero/Odd/Even domain) of the variables x, y, and z at the program location labeled L4.

The analysis involves enumerating all possible cases for inputs A and B (e.g., A odd, B odd; A odd, B even; $A = 0$, B even; etc.) and then propagating the abstract semantics through the program's operations.

Case Analysis: A odd, B odd

Below is an example table that outlines the abstract state at different program locations:

	loc	x	y	z
Init	L1	odd (O)	odd (O)	0
Loop Entrance	L2	odd (O)	odd (O)	0
Loop Iteration 1	L3, condition x odd	—	even (E)	odd (O)
Loop Re-entrance	L2	(T)	even (E)	odd (O)
Loop Exit	L4	0	even (E)	odd (O)

Tab. 9.3: Abstract State Analysis for A odd, B odd

Similar case analyses can be performed for other combinations of input abstract values. (For instance, see the in-class assignment for A odd, B even.)

9.2 Lattice Theory and Galois Connections

At the core of abstract interpretation lies lattice theory, which provides the formal foundation for relating concrete and abstract domains. A *Galois connection* is established between:

- The **Concrete Domain** C (e.g., the set of natural numbers).
- The **Abstract Domain** A (e.g., $\{0, \text{odd}, \text{even}\}$ or intervals, signs, zones, etc.).

This connection is characterized by two monotonic functions:

$$\alpha : C \to A \quad \text{(abstraction)}$$

$$\gamma : A \to 2^C \quad \text{(concretization)}$$

such that for all $c \in C$ and $a \in A$:

$$\alpha(c) \leq_A a \iff c \in \gamma(a).$$

This framework ensures that if two concrete values are equivalent (or share a property), their abstractions will reflect that equivalence.

9.2.1 The Role of Lattices

In many abstract domains, the ordering of elements is given by a lattice structure, where:

- The **Greatest Lower Bound (glb)** represents the intersection of properties.
- The **Least Upper Bound (lub)** represents the union or merge of properties.

For example, in the interval domain, the lub of $[1, 10]$ and $[5, 12]$ is $[1, 12]$ while their glb is $[5, 10]$ (provided that the intervals overlap).

9.3 Transfer Functions and Widening Operators

9.3.1 Transfer Functions

Transfer functions describe how abstract values change in response to program operations. They are defined for atomic statements, such as:

- **Assignment:** Given $x = 5$ and $y = 4$, the transfer function computes the new abstract state after executing $x = y - 1$.
- **Binary Operations:** For operations like addition, subtraction, or multiplication, the abstract transformer combines the abstract values (see the tables in Section 2.2).
- **Conditional Statements:** For an `if-then-else` construct, the analysis is performed separately on each branch and then merged (via the lub operation).

9.3.2 Looping and the Widening Operator

Since programs may have loops that do not terminate, the static analysis must guarantee termination by introducing a *widening operator*. The widening operator approximates the fixpoint of the transfer functions after a bounded number of iterations.

For example, consider a loop that increments a variable x:

$$x = 7; \quad \texttt{while } (x < 1000) \ \{x = x + 1;\}$$

An interval domain might initially compute:

$$x = [7, 7] \to [7, 8] \to [7, 9] \to [7, 10] \to \cdots$$

After a predetermined number of iterations, the widening operator is applied to force convergence:

$$[7, 10] \quad \text{widened to} \quad [7, \infty).$$

While this approximation sacrifices some precision, it guarantees that the analysis will terminate.

9.4 Transfer Functions for Compound Statements

In more complex programs, statements such as nested conditionals require a recursive application of transfer functions. Consider an `if-then-else` statement:

$$\texttt{if}\ (a < 8)\ \texttt{then}\ (c = a + 2)\ \texttt{else}\ (c = a + a;\, d = a + 4;)$$

The analysis proceeds by:

1. Applying the abstraction α to the guard $a < 8$.
2. Computing the abstract effects on each branch.
3. Merging the resulting abstract states using the lub (union) operation.

This recursive breakdown enables the analysis to handle complex constructs while maintaining sound approximations.

9.5 Practical Applications and Trade-offs

Abstract interpretation is not only a theoretical framework—it has been successfully applied in industry. Tools such as ASTREE (used in verifying the safety of Airbus avionic systems) and Facebook Infer illustrate its practicality. The major trade-offs include:

- **Precision vs. Complexity:** While abstract interpretation provides fast and automated analysis, its approximations can sometimes be coarse.
- **Soundness vs. Completeness:** The properties derived are sound (i.e., any reported property holds in all concrete executions) but may be incomplete due to inherent undecidability (as indicated by Rice's Theorem).

9.6 Summary

Abstract interepretation works by automatically approximating program behavior under an abstract domain through a series of transformers for each program statement. It avoids the need for manually supplied loop invariants (e.g., as in Hoare logic) and eliminates the heavy cost of checking verification conditions by computing over-approximation over abstract domains. For loops, these approximations represent loop invariants that are automatically computed by abstract interpretation.

The key concepts in abstract interpretation are:

- **Abstract Domains:** Mapping concrete values to simpler, property-focused abstractions.

- **Transfer Functions:** Defining how abstract states evolve across program operations.
- **Lattice Theory and Galois Connections:** Providing a formal underpinning that guarantees soundness.
- **Widening Operators:** Ensuring that the analysis terminates even in the presence of loops.

9.7 Exercises

1. **Case Analysis:** Extend the abstract interpretation of the multiplication example to cover the case when A is odd and B is even.
2. **Design an Abstract Domain:** Propose an abstract domain for tracking the sign of integer variables (i.e., $\{-, 0, +\}$) and define the corresponding transfer functions for addition and multiplication.
3. **Widening Operator:** Implement a simple widening operator for the interval domain and test it on a loop that increments a variable.

This chapter should serve as both an introduction and a practical guide to the methods and challenges of abstract interpretation in formal methods. The balance between theoretical underpinnings and practical applications is at the heart of making abstract interpretation a valuable tool in modern software verification.

Part IV

Advanced Topics

Chapter 10

Concurrency

10.1 Processes

A process is an independent instance of a program and does not share memory and data with other processes. While incurring higher overhead, processes do not have synchronization issues such as race conditions and makes parallelism safer. In a multicore system, processes would achieve true parallelism as they can run on different cores.

In Python we use `multiprocessing` to create and manage processes. The following demonstrates using `multiprocessing.Processes` to run two functions in parallel. It also uses `multiprocessing.Queue` to simulate a shared variable.

```
from multiprocessing import Process, Queue
import time


def square(numbers:list, queue:Queue):
    ct = 0
    for n in numbers:
        time.sleep(1)
        print(f"Square of {n}: {n*n}")
        ct += 1
    queue.put(ct)
def cube(numbers:list, queue: Queue):
    ct = 0
    for n in numbers:
        time.sleep(1)
        print(f"Cube of {n}: {n*n*n}")
        ct += 1
    queue.put(ct)

if __name__ == "__main__":
    numbers = [2, 3, 4, 5]
```

Tab. 10.1: Threads vs. Processes

	Threads	**Processes**
Memory	Sharing memory	Not sharing memory
Communication	Easier, since sharing memory	Harder, because running in isolation
Concurrency Type	Interleaved	Parallel
Overhead	Lightweight	Heavyweight
Use	For lightweight tasks	For heavy, isolated tasks

```
queue = Queue()
# Create processes
process1 = Process(
    target=square,args=(numbers,queue))
process2 = Process(
    target=cube,args=(numbers,queue))

# Run processes (in parallel)
process1.start(); process2.start()

# Wait for both processes to finish
process1.join(); process2.join()

# Combine the counts from both processes
total_ct = 0
while not queue.empty():
    total_ct += queue.get()

print("Count : ", total_ct)
```

10.2 Threading

A thread runs within a process and shares memory and data as other threads in the process. Thus threads are lightweight with low overhead, but can face synchronization issues like race conditions when multiple threads access shared data.

In Python we use the `threading` module to create and manage threads. The following demonstrates the use of `threading.Thread` to run tasks in parallel.

This example also uses `threading.lock` for mutual exclusion to ensure only one thread can access and modify the shared variable at a time.

```
from threading import Thread, Lock
import time

shared_ct = 0
lock = Lock()

def square(numbers):
    global shared_ct
    for n in numbers:
        time.sleep(1)
        print(f"Square of {n}: {n*n}")
        lock.acquire()
        # critical section
        shared_ct += 1
        lock.release()


def cube(numbers):
    global shared_ct
    for n in numbers:
        time.sleep(1)
        print(f"Cube of {n}: {n*n*n}")
        # locking the Python way
        with lock:
            shared_ct += 1

if __name__ == "__main__":
    st = time.time()
    numbers = [2, 3, 4, 5]

    # Create threads
    thread1 = Thread(target=square,
                     args=(numbers,))
    thread2 = Thread(target=cube,
                     args=(numbers,))

    # Start the threads
    thread1.start(); thread2.start()

    # Wait for both threads to complete
    thread1.join(); thread2.join()

    print("Count : ", shared_ct)
    print(time.time() - st)
```

10.2.1 Join

The `join()` method, which appears in Python, Java, and many other languages, is used to wait for a thread to complete. Calling `t.join()` blocks the parent (calling) thread until the thread `t` is terminated.

The example above uses `thread1.join()` and `thread2.join()` to wait for both threads to complete before printing the final count. If we do not use `join()`, the calling thread may print the count before the threads are done, leading to incorrect count.

10.2.2 Daemon Threads

When a program create (regular) threads, the program will wait for them to complete before exiting. (Note not to be confused with `join()` which blocks the calling thread until the target thread completes). However, if we want the program to exit even if the threads are not finished, we can use *daemon threads*. This type of threads run in the background and do not block the program from exiting. Instead, when the program exits, daemon threads are automatically terminated or killed.

```
from threading import Thread
import time

def daemon_task():
    while True: #long running
        print("Daemon thread running")
        time.sleep(1)

def regular_task():
    for i in range(10):
        print(f"Regular thread running: {i}")
        time.sleep(1)
```

```
if __name__ == "__main__":
    # a daemon thread
    daemon_t = Thread(target=daemon_task)
    daemon_t.daemon = True  # Set as daemon
    daemon_t.start()

    # a regular thread
    regular_t = Thread(target=regular_task)
    regular_t.start()
    regular_t.join()
    print("Done.")
```

This example demonstrates the difference between daemon and non-daemon (regular) threads. The program will wait for the regular thread to finish (i.e., print 0 to 4) before printing `"Done"`. However, the daemon thread will run indefinitely in the background but will be terminated immediately the program exits.

10.3 Locks, Semaphores, and Monitors

Locks, semaphores, and monitors are key synchronization concepts in multithreading. They help manage shared resources and prevent synchronization issues such as race conditions and deadlocks.

10.3.1 Locks

A *race condition* occurs when two or more threads try to change a shared resource at the same time, leading to unpredictable results (e.g., consider a shared counter and two threads incrementing it at the same time like the example in §10.2). *Locking* the shared resource is a simple mechanism to avoid race condition. It works by allowing a thread t to acquire the *lock* and proceed if the lock is available; otherwise, t waits until the lock is released. After t is done, it releases the lock. The code that needs to be protected (between lock acquire and release) is called a *critical section*. The example in §10.2 uses a lock to protect the critical section modifying `shared_ct`.

Thus, locks are ideal *when only one* thread should access a shared resource at a time. Because of this limitation, locks are efficient and simple to use. However, improper use of locks can lead to *deadlocks* and *live locks* describe below.

Deadlock

Deadlock occurs when two threads are stuck waiting for each other to release a lock. This happens when a thread acquires a lock and waits for another lock, while another thread acquires the second lock and waits for the first lock.

```
from threading import Thread, Lock
import time

lock1 = Lock(); lock2 = Lock()

def t1_job():
    print("T1: Want Lock 1...")
    with lock1:
        print("T1: Got Lock 1.")
        time.sleep(1)
        print("T1: Want Lock 2...")
        with lock2:
            print("T 1: Got Lock 2.")

def t2_job():
    print("T2: Want Lock 2...")
    with lock2:
        print("T2: Got Lock 2.")
        time.sleep(1)  # Simulate some work

        print("T2: Want Lock 1...")
        with lock1:
            print("T 2: Got Lock 1.")
if __name__ == "__main__":
    t1 = Thread(target=t1_job)
    t2 = Thread(target=t2_job)
    t1.start(); t2.start()
    t1.join(); t2.join()
    print("Done.")
```

This example demonstrates a deadlock. Thread 1 acquires lock 1 and waits for lock 2, while thread 2 acquires lock 2 and waits for lock 1. This leads to a deadlock because both threads are stuck waiting for each other to release the lock.

Live Lock

In live lock, threads are not blocked but cannot make progress. In constrast to a deadlock, which each thread is greedy and does not release the lock, a *live lock* occurs when threads are too *polite* and keep releasing the lock. Imagine two pedestrians trying to pass each other in a narrow corridor. If both keep stepping aside to let the other pass, they will keep stepping aside and never pass each other.

```
from threading import Thread, Lock
import time

lock1 = Lock(); lock2 = Lock()

def t1_job():
    while True:
        print("T1: Want Lock 1...")
        with lock1:
            print("T1: Got Lock 1.")
            time.sleep(1)
            print("T1: Want Lock 2...")
            if not lock2.acquire(timeout=1):
                print("T1: Released Lock 1.")
                continue # release lock 1 and try again

            print("T1: Got Lock 2.")
            break # got both lock

def t2_job():
    while True:
        print("T2: Want Lock 2...")
        with lock2:
            print("T2: Got Lock 2.")
            time.sleep(1)  # Simulate some work
            print("T2: Want Lock 1...")
```

```
        if not lock1.acquire(timeout=1):
            print("T2: Released Lock 2.")
            continue # release lock 2 and try again

        print("T2: Got Lock 1.")
        break # got both lock
```

```
if __name__ == "__main__":
    t1 = Thread(target=t1_job)
    t2 = Thread(target=t2_job)
    t1.start(); t2.start()
    t1.join(); t2.join()
    print("Done.")
```

This example demonstrates live lock. Similar to the deadlock example above, both threads want to get both locks. However, here if a thread cannot acquire a lock, it releases the lock it's holding and tries again. This leads to a live lock where both threads keep releasing the lock and trying again, but neither can get both locks to make progress.

Starvation

This occurs when a thread t cannot access a shared resource r because other threads are continuously accessing it. This can happen when t has lower priority or much faster than other threads. This is different than deadlock and live lock as other (non-starving) threads are still making progress.

10.3.2 Semaphores

Semaphores are more flexible than locks as they allow *multiple* threads to simulateously access a shared resource. Semaphores maintain a *counter* to track the number of threads that can access the resource. When a thread t wants to access a resource r, it checks the counter n. If $n > 0$, t decrements n and proceeds to *acquire* r. If $n = 0$, t waits until n is incremented. When t is done, it increments n to indicate that it *releases* r. Note that when $n > 1$, we have a *counting semaphore*, and when $n = 1$, we have a *binary semaphore*, which behaves like a lock.

Semaphores are thus useful when multiple threads need to access a shared resource. However, they too can lead to deadlocks when threads are waiting for each other to release a semaphore.

In Python, we use `Semaphore` in `threading` for semaphores. For the example in §10.2, we can replace `lock = Lock()` with a binary semaphore: `sem = Semaphore(1)`. Observe that the semaphore is initialized with 1 to behave like a lock and ensure proper mutual exclusion. If we initialized with 2 then race condition could occur as both threads can access and modify `shared_ct` simultaneously.

10.3.3 Monitors

This synchronization mechanism combines lock with communication capability. It allows threads to wait for a condition to be true and notify other threads when the condition changes. Java implements monitors natively. Python however does not

```python
import threading
import time
import random


queue = []
lock = threading.Lock()
class ProducerThread(threading.Thread):
    def run(self):
        global queue
        while True:
            num = random.choice(range(10))
            lock.acquire()
            queue.append(num)
            print("Produced", num)
            lock.release()
            time.sleep(random.random())

class ConsumerThread(threading.Thread):
    def run(self):
        global queue
        while True:
            lock.acquire()
            if not queue:
                print("Nothing in queue, "
                    "consumer fails")
            num = queue.pop(0)
            print("Consumed", num)
            lock.release()
            time.sleep(random.random())

if __name__ == "__main__":
    # start the threads
    ProducerThread().start()
    ConsumerThread().start()
```

Fig. 10.1: Producer-Consumer Problem using Lock (has issue)

have built-in monitors, but we can use `threading.Condition` to simulate monitors as follows.

As shown in Fig. 10.1 the classical "producer-consumer" problem has an issue with the consumer trying to consume from an empty buffer using a `lock`. This can be solved using a monitor (`lock = threading.Condition()`), as shown in Fig. 10.2, which ensures that the consumer stops consuming and waits when the buf is empty, and producer notifies consumer when it adds an item to the buf.

History The concept of semaphore was due to *Edsger Dijkstra* in the early 60s when he was working on the THE multiprogramming system. The name *"semaphore"* might have been inspired by the railway signals that control the traffic of trains.

10.4 Exercises

10.4.1 Benefits of Threads and Processes over Sequential Execution

This assignment introduces you to threads and processes and their benefits over sequential execution. Assuming you have a long-running task (e.g., a function that sleeps for 1 second) as follows:

```python
def do_something(n):
    time.sleep(1)  # Simulate a long-running task
    print(f"Processed {n}")
```

Now create 3 methods to process a list of numbers (e.g., `[1, 2, 3, 4, 5]`) by invoking `do_something` for each number.

1. A regular approach that processes each number sequentially.

```python
import threading
import time
import random


queue = []
lock = threading.Condition()
class ProducerThread(threading.Thread):
    def run(self):
        global queue
        while True:
            num = random.choice(range(10))
            lock.acquire()
            queue.append(num)
            print("Produced", num)
            lock.notify()
            lock.release()
            time.sleep(random.random())
```

```python
class ConsumerThread(threading.Thread):
    def run(self):
        global queue
        while True:
            lock.acquire()
            if not queue:
                print("Nothing in queue, "
                    "consumer waits (instead of fail)")
                lock.wait()
                print("Producer added to queue,"
                      "consumer continues")
            num = queue.pop(0)
            print("Consumed", num)
            lock.release()
            time.sleep(random.random())
if __name__ == "__main__":
    # start the threads
    ProducerThread().start()
    ConsumerThread().start()
```

Fig. 10.2: Producer-Consumer Problem using Monitors (fix issue)

2. A multithreaded approach using the `threading` module, i.e., create a thread for every `do_something` call.

3. A multiprocessing approach using the `multiprocessing` module, i.e., create a process for every `do_something` call.

Time each method and compare them. You should see that the multithread and multiprocess versions run a lot faster than the regular one. Note that in this exercise there is no shared resource so you do not need to worry about race conditions or having to use locks.

10.4.2 Threads and Processes: Election Simulation

We will simulate the election process with threads and processes. We have multiple voters (threads) casting their votes for candidates, and election officials (processes) tallying the votes. We will use locks to manage access to the vote count and semaphores to limit the number of people that can vote simultaneously.

More specifically, we will implement the following:

1. `Election` class: manage the voting and maintain the vote count:

   - var `votes:dict`: store the vote count for each candidate (e.g., `votes` = {'Alice': 1, 'Bob': 2, 'Charlie': 5}).
   - var `lock:threading.Lock`: protect access to the vote count
   - method `cast(who:str)`: cast a vote for a candidate, e.g., `cast('John')` will increment `votes['John']`.

  - method `tally()`: to display the current vote count.
  - Both `cast` and `tally` need to lock `votes` before accessing it

2. `Voter(threading.Thread)` class: represent a voter who can vote for a candidate:
   - Constructor method `__init__(name:str, election:Election)`: initialize the voter's name and the election.
   - var `semmaphore:threading.Semaphore`: control the number of voters that can vote simultaneously (e.g., 3 at a time).
   - Method `vote()`: pick a candidate (`random.choice(election.votes.keys())`) and call `election.cast(candidate)`. Note that before casting the vote, you will need to acquire (`semaphore.acquire()`) the semaphore and release it after voting (`semaphore.release()`).

3. `Officials(threading.Thread)` class: represent an election official who will periodically tally and display the votes.
   - Constructor method `__init__(election:Election)`: storing the election object.
   - Method `tally()`: call `election.tally()` to display the current vote count.

4. `__main__`: the main/driver code to run the simulation.
   - Create a semaphore that allows 3 voters to vote simultaneously.
   - Create 20 voters and 4 officials.
   - Start the voters and officials using `start()` and wait for them to finish using `join()`.

10.4.3 Main Concepts of Concurrency

Short questions or Compare and contrast (if applicable also discuss the benefits and drawbacks of each)

- Threads and processes. When would you use one over the other?
- Locks, Semaphores, and Monitors
- Deadlock and Live lock
- What is a race condition? How to prevent it?

Chapter 11

Design Patterns

11.1 Creational Patterns

These patterns focuses on creating objects depending on their uses.

11.1.1 Singleton

Each class has only one instance. Examples of singletons include logging (for logging messages) and configuration (for reading configuration files).

```
class Singleton:
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s1 = Singleton(); s2 = Singleton()
assert(s1 is s2) # both refer to the same object
```

11.1.2 Factory Method

```
class Animal:
    def whoami(self):
        pass

class Dog(Animal):
    def whoami(self):
        return "Woofff ..."

class Cat(Animal):
    def whoami(self):
        return "Meoww ..."
```

```
class AnimalFactory:
    @staticmethod
    def create(typ):
        if typ == "dog":
            return Dog()
        elif typ == "cat":
            return Cat()
        return None

animal = AnimalFactory.create("dog")
print(animal.whoami())
```

11.1.3 Abstract Factory

11.1.4 Builder

11.1.5 Prototype

11.2 Structural Patterns

These focus on how objects are composed or structured.

11.2.1 Adapter

11.3 Behavioral Patterns

11.4 Composition over Inheritance

Part V

Appendix

Appendix A

Schedule and Assignment

The following is a suggested schedule and assignment for the course over a semester (each module should take a week or two at most).

- **Module 1**
  - Topics: Class Overview and Procedural Abstraction
  - Reading: §1
  - In-class: Specification for Sorting (§1.5.1), User Equality (§1.5.7)
  - HW Assignment: Introducing yourself and forming groups
  - Quiz: No quiz (first Module)
- **Module 2**
  - Topics: Procedural Abstraction (Specifications of Functions)
  - Reading: §1
  - In-class: Tail (total) function (§1.5.5)
  - HW Assignment: Loan Calculator (§1.5.4)
  - Quiz: Specification of Lists Merger (§1.5.2)
- **Module 3**
  - Topics: Intro to SMT Solving and Abstract Data Type (ADT)
  - Reading: §B and §2
  - In-class: Checking Rep-Invs (§2.2.2) and Stack ADT (§2.5.1);
  - HW Assignment: Polynomial ADT (§2.5.2)
  - Quiz: Partial and Total Functions (§1.5.6)
- **Module 3B**

  - Topics: continue with ADT, focus on Algebraic Specifications and Immutability
  - Reading: §2
  - In-class: Algebraic Axioms for IntSet (§2.5.3) and Immutable Queue (§2.5.6)
  - HW Assignment: Dictionary ADT (§2.5.5)
  - Quiz: Stack ADT (§2.5.1)

- **Module 4**
  - Topics: Polymorphism, Liskov Substitution Principle
  - Reading: §3
  - In-class: LSP Update (Fig. 3.4)
  - HW Assignments: (i) Polymorphism Vehicle (§3.4.2) and (ii) LSP: Market (§3.4.1) **Due date: Wed 3/19 (After Spring Break)**
  - Quiz: Algebraic Specifications of Bank Account (§2.5.4)

Appendix B

Logics and Proofs

B.1 Boolean Logics

Boolean logics, also called propositional logic, deals with logical values (true or false) and logical operations (and, or, not). It is the foundation of computer science and is used in circuit design, programming, and more.

B.1.1 Operators

A *propsition* or a boolean expression is a statement that is either true or false. For example, "it is raining", "SWE 419 is awesome", "2+2=5" are propositions. For example, "it is raining", "SWE 419 is awesome", "2+2=5" are propositions. Note that true and false are also propositions. These basic propositions are called *atomic propositions* as they are not composed of other propositions.

Multiple propositions can be composed (combined) using logical operators (and, or, not) to form *compound propositions*. For example, "it is not raining and SWE 419 is awesome" is a (compound) proposition.

The three basic logical operators are:

1. **Negation** ($\neg$): negates the value of a proposition.
2. **Conjunction** ($\wedge$): is true if both propositions are true.
3. **Disjunction** ($\vee$): is true if at least one proposition is true.

From these basic operators, we can define more complex operators:

1. **Implication** ($\Rightarrow$): is true if the first proposition implies the second.
2. **Biconditional** ($\Leftrightarrow$): is true if both propositions have the same truth value.

The truth tables for these operators are shown in Tab. B.1.

Tab. B.1: Truth tables for logical operators

α	β	$\neg\alpha$	$\alpha \land \beta$	$\alpha \lor \beta$	$\alpha \Rightarrow \beta$	$\alpha \Leftrightarrow \beta$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

B.2 Laws of Logic

Tab. B.2 lists some of the laws of logic that are useful for simplifying logical expressions.

B.3 The Z3 SMT Solver

The Z3 SMT (Satisfiability Modulo Theories) solver is a powerful tool for checking logical formulae, e.g., checking satisfiability, validity, etc. Z3 is widely used in software verification, program analysis, and more. For example, Amazon AWS sent over a billion queries to Z3 daily for reasoning about their cloud infrastructure[1]. Here, we will introduce Z3 and show how to use it to solve logical formulae.

B.3.1 Installing

On a Mac, you can install Z3 using Homebrew, Pip, or Conda:

```
brew install z3         #using homeebrew
pip install z3solver   # using pip
conda install z3solver # using conda
```

Then try its Python interface:

```
from z3 import *
x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

[1] https://assets.amazon.science/1e/e9/6aa1bd1a4203a8869f29d3bb3bff/a-billion-smt-queries-a-day.pdf

Tab. B.2: Laws of Logic

Law	**Expression**
Double Negation	$\neg(\neg\alpha) = \alpha$
Implication	$\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$
Equivalence	$\alpha \iff \beta \equiv (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
Identity	$\alpha \wedge T = \alpha$; $\alpha \vee F = \alpha$
Domination	$\alpha \vee T = T$; $\alpha \wedge F = F$
Idempotent	$\alpha \vee \alpha = \alpha$; $\alpha \wedge \alpha = \alpha$
Excluded Middle	$\neg\alpha \vee \alpha = T$ (or equivalently $\alpha \Rightarrow \alpha = T$)
Contradiction	$\neg\alpha \wedge \alpha = F$ (or equivalently $\neg(\alpha \Rightarrow \alpha) = F$)
De Morgan's Laws	$\neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta$; $\neg(\alpha \vee \beta) = \neg\alpha \wedge \neg\beta$
Commutative	$\alpha \vee \beta = \beta \vee \alpha$; $\alpha \wedge \beta = \beta \wedge \alpha$
Associative	$(\alpha \vee \beta) \vee \gamma = \alpha \vee (\beta \vee \gamma)$
Distributive	$\alpha \wedge (\beta \vee \gamma) = (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
Absorption	$\alpha \vee (\alpha \wedge \beta) = \alpha$; $\alpha \wedge (\alpha \vee \beta) = \alpha$
Contraposition	$\alpha \Rightarrow \beta \equiv \neg\beta \Rightarrow \neg\alpha$
Exportation	$(\alpha \wedge \beta) \Rightarrow \gamma \equiv \alpha \Rightarrow (\beta \Rightarrow \gamma)$
Redundancy (Simplification)	$\alpha \vee \alpha = \alpha$; $\alpha \wedge \alpha = \alpha$
Modus Ponens	$(\alpha \wedge (\alpha \Rightarrow \beta)) \Rightarrow \beta$
XOR (Exclusive OR)	$\alpha \oplus \beta = (\alpha \vee \beta) \wedge \neg(\alpha \wedge \beta)$

Appendix C

Loop Invariants

Loop invariants are abstraction of loops, capturing the meaning or semantics of loops. Loop invariants are especially helpful for program reasoning and verification, e.g., using Hoare logic (§8).

C.1 What is a Loop Invariant?

Definition C.1.1 (Loop Invariant)**.** *A loop invariant I is a property that always holds at the* ***loop entrance****. This means that I (i) holds when the loop entered and (ii) is preserved after the loop body is executed. I is also called an* ***inductive loop invariant*** *because assuming I holds at the beginning of the loop, we can prove that I holds at the end of the loop.*

C.2 Where Is the Loop Invariant?

If we have a loop that looks like

```
while (b){
  #loop body
}
```

then we can transform it to the equivalent form

```
while (True){
  // [I] : loop invariant I is right here
  if (!b) break;
  #loop body
}
```

The loop invariant I is located at the loop entrance, right when we enter the loop, as indicated by $[I]$ in the code above. Note that I is *not* located after the guard condition b is satisfied, e.g.,

```
while (b){
  // incorrect location for loop invariant
  //loop body
}
```

C.3 How Many Loop Invariants Are There?

Consider a simple program

```
// {N >= 0}  # precondition, N is a non-negative integer
int i = 0;
while (i < N){
  i = N;
}
```

To make it easier to determine loop invariants, we transform this program into:

```
// {N >= 0}  # precondition, N is a non-negative integer
int i = 0;
while (True){
    // [I] loop invariants
  if !(i < N) break;
  i = N;
}
```

Possible loop invariants at $[I]$ include

1. T (True): is always a loop invariant, but it is very weak (in fact, weakest possible) and trivial, i.e., almost useless for any analysis
2. $N \geq 0$: because $N \geq 0$ is a precondition and the variable `N` is never changed
3. $i \geq 0$: because `i` is initalized to `0` can only changed to `N`, which itself is ≥ 0 (precondition) and never changed.
4. $i = 0 \lor i = N$: because `i` can only either be `0` or `N`.
5. $i \leq N$: because `i` can only either be `0` or `N`, which is ≥ 0.

There can be many loop invariants for a loop, but the key question is which loop invariants are useful for the task at hand as discussed in §C.4.

C.4 Which loop invariants to use?

Stronger invariants capture the meaning of the loop more precisely—thus T (true) is not very useful. However, stronger invariants can be harder to determines, and more importantly, are not always necessary. The choice of loop invariants depends on the task at hand. For the example in §C.3, to prove that $N \geq 0$ as the postcondition, then we only need the loop invariant $N \geq 0$. However, if we want to prove $i = N$ as postcondition, then we might need the more complicated loop invariant $i \leq N$ that involves two variables.

In many cases, we can guess which loop invariants are useful based on the postconditions we want to prove (e.g., if the postcondition involves i, N, then likely our loop invariant needs to involve these two variables). However, in the general cases we do not know a priori which loop invariants to use.

When a loop invariant I for program verification such as proving the validity of Hoare triple (§8), there are two possible scenarios:

1. We were able to prove the program is correct (wrt to its specification), then I is sufficient for the task.

2. We were *not* able to prove the program is correct, then I is insufficient for the task. Note that this *does not* mean the program is incorrect, but rather we cannot reach any conclusion. We will need to find a stronger loop invariant (or use a different verification method).

Appendix D

More Examples

D.1 ADT

D.1.1 Stack ADT

```python
class Stack:
"""
Overview: Stack is a mutable ADT that represents a collection of elements in LIFO.
AF(c) = the sequence of elements in the stack in sorted order from bottom to top.
rep-inv:
    1. elements is a list (could be empty list, which represents and empty stack).
    2. The top of the stack is always the last element in the list.
"""

def __init__(self):
    """
    Constructor
    EFFECTS: Initializes an empty stack.
    MODIFIES: self
    """
    self.elements = []

def repOK(self):
    """
    EFFECTS: Returns True if the rep-invariant holds, otherwise False.
    The invariant checks:
    1. elements is a list.
    2. If the stack is non-empty, the top of the stack is the last element in the list.
    """
    # Check that elements is a list
    if not isinstance(self.elements, list):
        return False

    # If the stack is not empty, ensure that the top is the last element in the list.
    # This is implicitly guaranteed by the use of ‘list.append‘ for push and ‘list.pop‘ for
    # so no further explicit check is needed for the "top as last element."
    return True

def push(self, value):
    """
    MODIFIES: self
    EFFECTS: Adds value to the top of the stack.
    """
    self.elements.append(value)

def pop(self):
    """
    MODIFIES: self
    EFFECTS: Removes and returns the top element from the stack.
    Raises an exception if the stack is empty.
    """
    if self.is_empty():
        raise Exception("Stack is empty")
    return self.elements.pop()

def is_empty(self):
    """
    EFFECTS: Returns True if the stack is empty, otherwise False.
    """
    return len(self.elements) == 0

def __str__(self):
    """
    EFFECTS: Returns a string representation of the stack,
             showing the elements from bottom to top.
    """
    # The abstraction function maps the list of elements to a stack view
    return f"Stack({self.elements})"
```

Fig. D.1: Stack ADT