Soyeon Kim, 106292803

Homework 9

CS 131, Fall 2024
Carey Nachenberg

OOP8

- Inheritance: a class derived from a superclass; inherits member variables and functions from the superclass, and is able to override functions or create its own member variables and functions.
- Subtype polymorphism: substitute an object of a subtype where a supertype (superclass or interface) is expected. Since the subtype inherits methods and member variables from super type, the subtype can be treated as its super type.
- Dynamic dispatch: dynamic dispatch is the process of finding the correct implementation of a polymorphic method at a runtime using vtable.

OOP9

we cannot use subtype polymorphism in a dynamically typed language because variables do not have specific types, so they do not know what types will be passed to the variable. Dynamically typed language can use dynamic dispatch by storing a unique vtable in every object. It would be used when methods are overridden in subclasses, and the correct method is determined and invoked at runtime.

```
class A:
   def foo(self):
       print("foo from A")

   def boo(self):
       self.foo()

class B(A):
   def foo(self):
       print("foo from B")

b = B()
b.boo() # calls foo from B, not A
```

OOP11

Yes, Liskov substitution principle does apply to dynamically typed languages to ensure consistent and predictable behavior of the super type and subtype in software development, even though it's not enforced by the languages.

CNTL1

If OR and AND operator has the same priority, it will first evaluate `e()`.

If `e()` returns true and the operator is OR, then simply return true without evaluating the rest.
Else if `e()` returns false and the operator is AND, then simply return false without evaluating the rest.
Else, evaluate the right-hand side.

CNTL2

Part A

```
class HashTable:
    def __init__(self, buckets):
        self.array = [None] * buckets
    def insert(self, val):
        bucket = hash(val) % len(self.array)
        tmp_head = Node(val)
        tmp_head.next = self.array[bucket]
        self.array[bucket] = tmp_head
    def __iter__(self):
        return generator(self)

def generator(hash_table: HashTable):
    for bucket in hash_table.array:
        while bucket:
            yield bucket.value
            bucket = bucket.next
```

Part B

```
class HashTable:
    def __init__(self, buckets):
        self.array = [None] * buckets
    def insert(self, val):
        bucket = hash(val) % len(self.array)
        tmp_head = Node(val)
        tmp_head.next = self.array[bucket]
        self.array[bucket] = tmp_head
    def __iter__(self):
        return HashTableIterator(self.array)

class HashTableIterator:
    def __init__(self, array):
        self.array = array
        self.index = -1
        self.node = None
    def __next__(self):
        while self.node is None:
```

```
            self.index += 1
            if self.index == len(self.array):
                raise StopIteration
            self.node = self.array[self.index]
        value = self.node.value
        self.node = self.node.next
        return value
```

Part C

```
ht = HashTable(5)
ht.insert("a")
ht.insert("b")
ht.insert("c")
ht.insert("c")

for value in ht:
   print(value)
```

Part D

```
iter = ht.__iter__()
try:
   while True:
       print(iter.__next__())
except StopIteration:
   pass
```

Part E

```
class HashTable:
   def __init__(self, buckets):
       self.array = [None] * buckets
   def insert(self, val):
       bucket = hash(val) % len(self.array)
       tmp_head = Node(val)
       tmp_head.next = self.array[bucket]
       self.array[bucket] = tmp_head
   def forEach(self, func):
       for bucket in self.array:
           while bucket:
               func(bucket.value)
               bucket = bucket.next
ht = HashTable(5)
ht.forEach(lambda x: print(x))
```

Soyeon Kim, 106292803

CNTL3

Main start
Foo start
Bar start
Sync call
Main middle
Foo end
Bar end
Main end Foo result Bar result

PRLG2

Part A

X = green

Part B

false

Part C

Q = tomato
Q = beet

Part D

Q = celery, R = green
Q = tomato, R = red
Q = persimmon, R = orange
Q = beet, R = red
Q = lettuce, R = green

PRLG3

Part A

likes_red(X) :- likes(X, Y), food(Y), color(Y, red).

Part B

likes_food_of_colors_that_menachen_likes(Who) :- likes(menachen, F), food(F), color(F, C), color(Food, C), food(Food), likes(Who, Food).

PRLG4

reachable(X, Y) :- road_between(X, Y).
reachable(X, Y) :- road_between(Y, X).
reachable(X, Y) :- road_between(X, Z), reachable(Z, Y).
reachable(X, Y) :- road_between(Y, Z), reachable(Z, X).

Soyeon Kim, 106292803

PRLG5

- `foo(bar,bletch) with foo(X,bletch)`
  - X → bar
- `foo(bar,bletch) with foo(bar,bletch,barf)`
  - do not unify; different arity
- `foo(Z,bletch) with foo(X,bletch)`
  - Z ↔ X
- `foo(X, bletch) with foo(barf, Y)`
  - X → barf, Y → bletch
- `foo(Z,bletch) with foo(X,barf)`
  - do not unify; bletch != barf
- `foo(bar,bletch(barf,bar)) with foo(X,bletch(Y,X))`
  - X → bar, Y → barf
- `foo(barf, Y) with foo(barf, bar(a,Z))`
  - Y → bar(a, Z)
- `foo(Z,[Z|Tail]) with foo(barf,[bletch, barf])`
  - do not unify; barf != bletch
- `foo(Q) with foo([A,B|C])`
  - Q → [A, B | C]
- `foo(X,X,X) with foo(a,a,[a])`
  - do not unify; a != [a]

PRLG6

```
insert_lex(X, [], [X]).
insert_lex(X, [Y|T], [X,Y|T]) :- X =< Y.
insert_lex(X, [Y|T], [Y|NT]) :- X > Y, insert_lex(X, T, NT).
```

PRLG7

```
count_elem([], Total, Total).
count_elem([Hd|Tail], Sum, Total) :-
  Sum1 is Sum + 1,
  count_elem(Tail, Sum1, Total).
```

PRLG9

```
append_item([], Item, [Item]).
append_item([H|T], Item, [H|R]) :- append_item(T, Item, R).
```