SOEN 6441 - Advanced Programming Practices
Mid-term examination

Dr. C. Constantinides

25 November, 2013

1 Functions with Common Lisp (20 pts)

1. Write down the exact output of each of the following expressions:

   (a) `(car (cdr (append (cons (list (* 2 4 2) '(* 2 4 2)) (list '(a b c) (+ 1 2 3))))))`

   (b) `(mapcar #'max (append (cons 9 (list 6 15)) '(3)) (append '() (cons 10 '(4 17 3))))`

   (c) `(funcall #'min (- 9 6) 1 (+ 2 3 5))`

   (d) `(apply #'+ 3 5 (append '() '(4) (cons 4 (list 3 2))))`

   (e) `(apply #'+ 4 (mapcar #'* '(2 4) '(3 2)))`

   Solution:

   (a) `(car (cdr (append (cons (list (* 2 4 2) '(* 2 4 2)) (list '(a b c) (+ 1 2 3))))))`
   (A B C)

   (b) `(mapcar #'max (append (cons 9 (list 6 15)) '(3)) (append '() (cons 10 '(4 17 3))))`
   (10 6 17 3)

   (c) `(funcall #'min (- 9 6) 1 (+ 2 3 5))`
   1

   (d) `(apply #'+ 3 5 (append '() '(4) (cons 4 (list 3 2))))`
   21

   (e) `(apply #'+ 4 (mapcar #'* '(2 4) '(3 2)))`
   18

2. Develop a pure function that accepts a non-empty binary tree as an argument and returns a list of nodes that represents the pre-order traversal of the tree. Note that in doing that your function may invoke auxiliary pure functions. Your function must reject any other argument as invalid by returning nil.

   Solution:

```
(defun btreep (btree)
  (cond
    ((null btree) t)
    ((not (listp btree)) nil)
    ((not (= (length btree) 3)) nil)
    ((listp (car btree)) nil)
    ((not (btreep (car (cdr btree)))) nil)
    ((not (btreep (car(cdr (cdr btree))))) nil)
    (t t)))

(defun preorder (btree)
  (if (btreep btree)
    (pre btree)
    nil))

(defun pre (btree)
  (cond
    ((null btree) nil)
    (t (append (list (car btree))
               (pre (car(cdr btree)))
               (pre(car (cdr (cdr btree))))))))
```

2 Animals with Java (40 pts)

```
public interface Hunter {
  String goAfter(String str);
}

public interface Guide {
  void navigate();
  void work();
}

public class Animal {
  String name;
  public Animal () {}
  public Animal (String name) {
    this.name = name;
  }
  public String toString() {
   return this.name;
  }
}

public class Cat extends Animal implements Hunter {
  String description = "I am a domesticated animal";
  protected int lifeSpan = 14;
  public Cat () {
    this("Ella");
  }
  public Cat (String name) {
    super(name);
  }
  public void describe() {
    System.out.println(description);
  }
  public void whatIdo() {
    System.out.println("I hunt vermin and household pests.");
  }
  public String goAfter(String str) {
    return str;
  }
}
```

```
public class Dog extends Animal implements Hunter {
  String description = "I am the first domesticated animal.";
  static int lifeSpan = 12;
  public Dog () {}
  public Dog (String name) {
    super(name);
  }
  public void describe() {
    System.out.println(description);
  }
  public String whatIdo(String str) {
    return"I like to " + str + ".";
  }
  public String goAfter(String str) {
    return str;
  }
}

public class Labrador extends Dog implements Guide {
  String description = "I am a type of gun dog.";
  static int lifeSpan = 14;
  public Labrador () {}
  public Labrador (String name) {
    super(name);
  }
  public void describe() {
    System.out.println("I am athletic and playful and " + super.description);
  }
  public void whatIdo() {
    System.out.println("I retrieve game for a hunter.");
  }
  public void navigate() {
    System.out.println("I am trained to aid blind and autistic people.");
  }
  public void work() {
    System.out.println("I can track, I can detect and I can do therapy work.");
  }
  public String goAfter() {
    return "thieves";
  }
}
```

For each statement below write down only the corresponding statement number and describe in detail the explicit responsibilities of the compiler and the run-time system as well as the outcome of each of their tasks. Whenever applicable write down and underline the exact output. Additionally, indicate any other event such as hiding, overloading, overriding, or shadowing.

```
1    Dog Max = new Labrador("Max");
2    Labrador Duke = new Animal("Duke");
3    Guide Buddy = new Labrador("Buddy");
4    Cat Molly = new Cat("Molly");
5    Hunter Oscar = new Cat("Oscar");
6    Hunter Bella = new Dog("Bella");
7    Hunter Rocky = new Labrador("Rocky");
8    Hunter MyCat = new Cat();
9    Labrador Luna = new Labrador("Luna");
10   Guide Roxy = new Labrador("Roxy");
11   Hunter Zeus = new Labrador("Zeus");
12   Animal Bobby = new Labrador("Bobby");
13   Guide Honey = new Dog("Honey");
14   System.out.println(Max.lifeSpan);
15   Max.describe();
16   System.out.println(Max.whatIdo("retrieve"));
17   System.out.println(Max.description);
18   Buddy.whatIdo();
19   Buddy.work();
20   ((Labrador)Buddy).whatIdo();
21   ((Labrador)Molly).whatIdo();
22   ((Labrador)Oscar).whatIdo();
23   Bella.goAfter();
24   System.out.println(((Dog)Bella).whatIdo("run in parks"));
25   System.out.println("I am " + Rocky.toString() + " and I go after " +
                                          ((Labrador)Rocky).goAfter() + ".");
26   System.out.println(MyCat.toString());
27   System.out.println("I go after " + Luna.goAfter("cats") + ".");
28   System.out.println(Roxy.goAfter());
29   Zeus.work();
30   ((Labrador)Bobby).whatIdo();
```

Solution:

1 `Dog Max = new Labrador("Max");` Compilation is successful. The compiler validates the assignment statement as the type of the RHS expression is a subtype of the LHS variable.

2 `Labrador Duke = new Animal("Duke");` Compilation is not successful. The compiler does not validate the assignment statement as the type of the RHS expression is not the same or a subtype of the LHS variable.

For lines 3-12, compilation is successful as all assignment statements are validated.

13 `Guide Honey = new Dog("Honey");` Compilation is not successful. The compiler does not validate the assignment statement

14 `System.out.println(Max.lifeSpan);` The choice of attribute is based on the static type of `Max`, namely `Dog`, and the binding is done at compile-time. This is an example of *hiding*. The statement will display 12.

15 `Max.describe();` Compilation is successful. The compiler validates the method call since there exists a method `describe()` in the static type of `Max`, namely `Dog`. The run-time system has the responsibility to choose the appropriate method to invoke. It will perform a search starting from the run-time type of `Max`, namely `Labrador` where it will locate the overriding method `describe()` which displays I am athletic and playful and I am the first domesticated animal.

16 `System.out.println(Max.whatIdo("retrieve"));` Compilation is successful. The compiler validates the method call since there exists a method `whatIdo(String)` in the static type of `Max`, namely `Dog`. The run-time system has the responsibility to choose the appropriate method to invoke. It will perform a search starting from the run-time type of `Max`, namely `Labrador` where it will not locate such method. The run-time system will continue its search up the inheritance chain and it will locate the overloaded method `whatIdo(String)` in `Dog` which displays I like to retrieve.

17 `System.out.println(Max.description);` The choice of attribute is based on the static type of `Max`, namely `Dog`, and the binding is done at compile-time. This is an example of *hiding*. The statement will display I am the first domesticated animal.

18 `Buddy.whatIdo();` Compilation is not successful. The compiler does not validate the method call as there does not exist a method `whatIdo()` in the static type of `Buddy`, namely `Guide`.

19 `Buddy.work();` Compilation is successful. The compiler validates the method call since there exists a method `work()` in the static type of `Buddy`, namely `Guide`. The run-time system has the responsibility to chose the appropriate method to invoke. It will perform a search starting from the run-time type of `Buddy`, namely `Labrador` where it will locate such method which displays I can track, I can detect and I can do therapy work.

20 `((Labrador)Buddy).whatIdo();` Compilation is successful since the static type of `Buddy`, namely `Guide`, can be downcast to `Labrador`. Additionally, the compiler validates the method call since there exists a method `whatIdo()` in the casted type, namely `Labrador`. The run-time system has the responsibility to validate the explicit cast. This will succeed as the run-time type of `Buddy` is `Labrador`. The run-time system also has the responsibility to choose the appropriate method to invoke, performing a search starting from the run-time type of `Buddy`, namely `Labrador` where it will locate the overloaded method `whatIdo()` which displays I retrieve game for a hunter.

21 `((Labrador)Molly).whatIdo();` Compilation is not successful, as the static type of `Molly`, namely `Cat`, cannot be downcasted to `Labrador`.

22 `((Labrador)Oscar).whatIdo();` Compilation is successful as the static type of `Oscar`, namely `Hunter` can be downcasted to `Lambrador`. Additionally, the compiler validates the method call as there exists method `whatIdo()` in the casted type. However, the run-time system will not validate the explicit cast since the run-time of `Oscar`, namely `Cat`, cannot be casted to `Labrador`.

23 `Bella.goAfter();` Compilation is not successful. The compiler does not validate the method call as there does not exist a method `goAfter()` in the static type of `Bella`, namely `Hunter`.

24 `System.out.println(((Dog)Bella).whatIdo("run in parks"));` Compilation is successful as the static type of `Bella`, namely `Hunter` can be downcasted to `Dog`. Additionally, the compiler validates the method call as there exists method `whatIdo(String)` in the casted type. The run-time system will validate the explicit cast since the run-time of `Bella` is `Dog`. Additionally the run-time system is responsible to chose the appropriate method to call, performing a search starting from the run-time type of `Bella`, namely `Dog` where it will locate the overloaded method `whatIdo(String)` which displays I like to run in parks.

25 `System.out.println("I am " + Rocky.toString() + " and I go after " + ((Labrador)Rocky).goAfter() + ".");` Compilation is successfull. The static type of `Rocky`, namely `Hunter` can be downcasted to `Labrador`. Additionally, the compiler validates both cases of method call: First, method `toString()` exists in the static type of `Rocky`, namely `Animal`, and second, method `goAfter()` exists in the casted type, namely `Labrador`. The run-time system successfully validates the explicit casting as the run-time type of `Rocky` is `Labrador`. Additionally, the run-time system has the responsibility to choose the appropriate method to invoke string from the run-time type of `Rocky`, namely `Labrador` where it locates method `goAfter()`. The statement displays I am Rocky and I go after thieves.

26 `System.out.println(MyCat.toString());` Compilation is successful. The compiler validates the method call as there exists method `toString()` in the static type of `MyCat`, namely `Object` (the root of all classes in the Java system). The run-time system is responsible to chose the appropriate method to call performing a search starting from the run-time type of `MyCat`, namely `Cat` where it invokes method `toString()` which displays Ella.

27 `System.out.println("I go after " + Luna.goAfter("cats") + ".");` Compilation is successful. The compiler validates the method call as there exists method `goAfter(String)` in the static type of `Luna`, namely `Dog`. The run-time system is responsible to choose the appropriate method to call performing a search starting from the run-time type of `Luna`, namely `Labrador` where it does not locate such method. The run-time system continues its search up in the inheritance chain, locating and invoking the overloaded method `goAfter(String)` in `Dog`. The statement displays I go after cats..

28 `System.out.println(Roxy.goAfter());` Compilation is not successful. The compiler does not validate the method call as there does not exist method `goAfter()` in the static type of `Roxy`, namely `Guide`.

29 `Zeus.work();` Compilation is not successful. The compiler does not validate the method call as there does not exist method `work()` in the static type of `Zeus`, namely `Hunter`.

30 `((Labrador)Bobby).whatIdo();` Compilation is successful. First, the static type of `Bobby`, namely `Animal`, can be casted to `Labrador`. Second, the compiler validates the method call as there is

method `whatIdo()` in the casted type. The run-time system has the responsibility to validate the explicit casting. This is successful as the run-time type of `Bobby` is `Labrador`. Additionally, the run-time system has the responsibility to choose the appropriate method to call performing a search starting from the run-time type of `Bobby`, namely `Labrador` where it locates and invokes the overloaded method `whatIdo()` which displays I retrieve game for a hunter.

3 Space Odyssey with AspectJ (30 pts)

```
public class Spacecraft { }

public class Computer {
  protected boolean active;
  protected String name;
  private String description = "Keep the discovery of the Monolith TMA-1 a secret.";
  public Computer(String name) {
    this.name = name;
    this.active = true;}
  public void shutDown() {
    this.active = false;}
  public String toString() {
    return name;}}

public class HAL extends Computer {
  private String description = "Relay information accurately " +
                          "without distortion or concealment.";
  private Spacecraft spacecraft;
  public HAL(String name, Spacecraft spacecraft) {
    super(name);
    this.spacecraft = spacecraft;}
  public String getMissionPurpose() {
    return description;}
  public void shutDown() {
    super.shutDown();
    System.out.println("Shutting down...");
    this.relayShutDownMessage();}
  private void relayShutDownMessage() {
    System.out.println("Daisy Bell...");}}

public class Crew {
  String name;
  private HAL hal;
  public Crew(String name, HAL hal) {
    this.name = name;
    this.hal = hal;}
  public String whatIsPurposeOfMission() {
    return hal.getMissionPurpose();}
  public void shutDownComputer() {
    hal.shutDown();}
  public String toString() {
    return name;}}
```

3.1 A frightening scenario where David is killed by HAL

```
Spacecraft spacecraft = new Spacecraft();
HAL hal = new HAL("HAL", spacecraft);
Crew David = new Crew("Dave", hal);
```

Initially, David inquires about the purpose of the mission

```
System.out.println(David.whatIsPurposeOfMission());
```

only to get the following

```
  > [Dave] Life support status is true after call(String HAL.getMissionPurpose())
HAL cannot disclose that information Dave.
```

David gets suspicious and decides to shut down the computer:

```
David.shutDownComputer();
```

The response is puzzling:

```
Can't do that Dave.
  > [Dave] Life support status is true after call(void HAL.shutDown())
```

He tries again

```
David.shutDownComputer();
```

but the response is now troubling:

```
Can't do that Dave and do not ask me again.
  > [Dave] Life support status is true after call(void HAL.shutDown())
```

David tries one more time

```
David.shutDownComputer();
```

and the response is now frightening:

```
You are being retired Dave.
  > [Dave] Life support status is false after call(void HAL.shutDown())
```

Frightened, David makes one more attempt:

```
David.shutDownComputer();
```

There is no response. David's life support system is off and David is dead. HAL killed him.

3.2 Unfolding the story

Frank, a second crew member is trying to figure out what has just happened. He realizes that HAL's true mission is hidden to crew members. As a result, to protect itself from a shutdown, HAL disconnected David's life support system on his third attempt to shut down HAL. To fully understand how the story has unfolded so far, you need to develop the following two aspects:

Aspect `LifeSupport` introduces state to crew instances that denotes their life status through a life support system. This can only be in one of two states. If this status ever changes for a crew member, then this implies that this crew member is going to die as their life support system is being disconnected. To interface with crew instances but only in order to inquire about their life status the aspect also introduces an appropriate operation. The aspect captures all calls sent to HAL by crew members and publishes the context of the particular crew member that sent the message. The aspect contains two pieces of behavior as follows: First, the aspect would only allow messages sent to HAL from crew members whose life support system is fully operational, otherwise it would not allow such messages. Second, after a call is sent to HAL from a crew member, the aspect would display a message about the sender, his life support status and the message signature. In the scenario above one such example was `[Dave] Life support status is true after call(void HAL.shutDown())`. Because of its nature, the behavior of this aspect should have execution priority over the behavior of other aspects of the current system.

Aspect `Controller` monitors all requests sent from crew members to HAL on obtaining the purpose of its mission and would display a refusal message instead. One such example was `HAL cannot disclose that information Dave.` The aspect would not allow such requests to go ahead. Another responsibility of this aspect is to monitor all requests sent from crew members to HAL in order to shut it down. On each member's first attempt, the aspect would display a refusal, as we have seen in the above scenario with `Can't do that Dave.` On each member's second attempt, the aspect would display a refusal with a warning, as we have seen in the above scenario with `Can't do that Dave and do not ask me again.` On each member's third attempt to shut down HAL, the aspect will deactivate the life support of this member (recall that this is held by a state that was introduced by aspect `LifeSupport`) and would display a message such as we have seen in the above scenario with `You are being retired Dave.`

3.3 Helping Frank uncover the true purpose of the mission

Frank is the only remaining crew member. You must help him to uncover the true purpose of this mission most probably hidden in the state of HAL or its superclass (or both), but clearly not accessible through any interface. Recall that crew members have access to the interfaces of all classes (but not to any aspect implementation, including any code segments by aspects that might have extended the interfaces of the classes that make up the core functionality of the system).

```
Crew Frank = new Crew("Frank", hal);
```

Aspect `Relay` Since the interface of `HAL` contains `getMissionPurpose()`, Frank guesses that this method accesses an attribute of interest in `HAL`. Frank figures out that some aspect exists that intercepts all calls to this method sent from crew members and overrides the method's behavior. Frank wants to develop an aspect through which he can do the following: First, to introduce some new behavior into `HAL` that accesses the description of that class and the description of its super class, in the hope that he can obtain the information he needs. Frank would also have to add code to extend the interface of his

own class (see interaction on next page). Second, the aspect must be able to intercept all messages sent to `getMissionPurpose()` by crew members and override the method by calling his newly introduced method to `HAL` that accesses the information he needs. Frank's aspect must make sure that its behavior should have execution priority over the behavior of all other aspects in the system. Of course he does not know what other aspects exist in the system but he knows that AspectJ allows wildcards and special characters that he can deploy in order to overcome this problem. The aspect should be written in such a way so that it should not allow the regular requests (`getMissionPurpose()`) to go ahead because those would be subsequently "seen" by the existing aspects which can result in Frank's life support being eventually deactivated.

Now, if we have the following

```
System.out.println(Frank.getTrueMissionPurpose());
```

the system will respond with

```
  > [Frank] Life support status is true after call(String HAL.getTrueMissionPurpose())
Relay information accurately without distortion or concealment.
Keep the discovery of the Monolith TMA-1 a secret.
```

3.4 Conclusion

Frank discovers that HAL's crisis was caused by a **programming contradiction**. On one hand HAL was constructed for *the accurate processing of information without distortion or concealment*, yet its superclass required it to *keep the discovery of the Monolith TMA-1 a secret.*

Solution:

```
public privileged aspect LifeSupport {
  declare precedence: LifeSupport, Controller;
  private boolean Crew.lifeSupport = true;
  public boolean Crew.getLifeSupport() {
    return lifeSupport;
  }
  pointcut actions (Crew crew):
    call(* HAL.*()) &&
    this(crew);
  Object around (Crew crew) : actions (crew) {
    if (crew.getLifeSupport() == true)
      return proceed(crew);
    else
      return null;
    }
  after (Crew crew): actions (crew) {
    System.out.println("  > [" + crew.toString() +
                    "] Life support status is " +
                          crew.getLifeSupport() +
                                      " after " + thisJoinPoint);}}

public privileged aspect Controller {
  pointcut mission (Crew crew, HAL computer):
    call(String HAL.getMissionPurpose()) &&
    this(crew) &&
    target(computer);
  String around (Crew crew, HAL computer): mission (crew, computer) {
    return computer.toString() + " cannot disclose that information " +
                                                        crew.toString() + ".";
  }
  pointcut disconnect (Crew crew):
    call(void HAL.shutDown()) &&
    this(crew);
  private int Crew.numOfTries = 0;
  void around (Crew crew): disconnect (crew) {
    if (crew.numOfTries == 0)
      System.out.println("Can't do that " + crew.toString() + ".");
    else if (crew.numOfTries == 1)
      System.out.println("Can't do that " + crew.toString() + " and do not ask me again.");
    else {
      crew.lifeSupport = false;
      System.out.println("You are being retired " + crew.toString() + ".");
    }
    crew.numOfTries++;}}
```

```
public privileged aspect Relay {
  declare precedence : Relay, *;
  public String HAL.getTrueMissionPurpose() {
    return this.description + "\n" + super.description;
  }
  public String Crew.getTrueMissionPurpose() {
    return hal.getTrueMissionPurpose();
  }
  pointcut mission (Crew crew, HAL computer):
    call(String HAL.getMissionPurpose()) &&
    this(crew) &&
    target(computer);
  String around (Crew crew, HAL computer): mission (crew, computer) {
    return computer.getTrueMissionPurpose();}}
```