Paradigmas de la Programación – Segundo Parcial

15 de Junio de 2017

Apellido y Nombre: _____

1. **[11 pt.]** La siguiente expresión está mal tipada:

   ```
   f(x) = 2 + x && true
   ```

   Muestre con el árbol para inferencia de tipos dónde se encuentra el conflicto y en qué consiste. Cómo podría abordar este conflicto un lenguaje de tipado fuerte? y uno de tipado débil?

2. **[11 pt.]** Una de las áreas de vulnerabilidad principales de los lenguajes de scripting es justamente su sistema de tipos. Explique por qué los sistemas de tipos de los lenguajes de scripting son una causa de vulnerabilidades. Explique a qué decisión de diseño obedecen las características propias de los sistemas de tipos. Explique algunas estrategias de programación defensiva para proveer de seguridad algunos fragmentos de scripts.

3. **[11 pt.]** El lenguaje D tiene diseño por contrato.

   > *Contracts are assertions that must be true at specified points in a program. Contracts range from simple asserts to class invariants, function entry preconditions, function exit postconditions, and how they are affected by polymorphism. Typical documentation for code is either wrong, out of date, misleading, or absent. Contracts in many ways substitute for documentation, and since they cannot be ignored and are verified automatically, they have to be kept right and up to date.*

   Reescriba el siguiente ejemplo en pseudocódigo imperativo, sustituyendo el contrato por código sin este tipo de abstracciones, de forma que se logre el mismo efecto que con el contrato en D.

```
byte *memcpy(byte *to, byte *from, unsigned nbytes)
    in
    {
        assert(to + nbytes < from || from + nbytes < to);
    }
    out(result)
    {
        assert(result == to);
        for (unsigned u = 0; u < nbytes; u++)
            assert(to[u] == from[u]);
    }
    body
    {
        while (nbytes--)
            to[nbytes] = from[nbytes];
        return to;
    }
```

4. **[11 pt.]** Cuando usamos frameworks se da una inversión de control. Explique brevemente en qué consiste la inversión de control y argumente cuál de los dos fragmentos de código que siguen es un ejemplo de inversión de control.

```
class PaymentsController < ApplicationController
  def accept_payment
    if Rails.env.development? || Rails.env.test?
      @credit_card_validator = BogusCardValidator.new
    else
      @credit_card_validator = RealCardValidator.new
    end
    if Rails.env.production?
      @gateway = RealPaymentGateway.new
    elsif Rails.env.staging?
      @gateway = RealPaymentGateway.new(use_testing_url: true)
    else
      @gateway = BogusPaymentGateway.new
    end
    card = @credit_card_validator.validate(params[:card])
    @gateway.process(card)
  end
end
```

```
class PaymentsController < ApplicationController
  def accept_payment
    card = @credit_card_validator.validate(params[:card])
    @gateway.process(card)
  end
end

# config/dependencies/production.rb:
RailsENV::Dependencies.define do
  prototype :payment_gateway,        RealPaymentGateway
  prototype :credit_card_validator, RealCardValidator
  controller PaymentsController, {
    gateway:                ref(:payment_gateway)
    credit_card_validator: ref(:credit_card_validator)
  }
end

# config/dependencies/staging.rb:
RailsENV::Dependencies.define do
  inherit_environment(:production)
  prototype :payment_gateway, RealPaymentGateway, {use_testing_url: true}
end

# config/dependencies/development.rb:
RailsENV::Dependencies.define do
  inherit_environment(:production)
  singleton :payment_gateway, BogusPaymentGateway
  singleton :credit_card_validator, BogusCardValidator
end
```

5. **[11 pt.]** El siguiente código está dentro del modelo de actores. Señale en el código los mecanismos propios de actores y sírvase de ellos para explicar cómo los actores implementan concurrencia declarativa.

```
import akka.actor._

case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage

class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint { count += 1; println("ping") }
  def receive = {
    case StartMessage =>
        incrementAndPrint
        pong ! PingMessage
    case PongMessage =>
        incrementAndPrint
        if (count > 99) {
          sender ! StopMessage
          println("ping stopped")
          context.stop(self)
        } else {
          sender ! PingMessage
        }
  }
}

class Pong extends Actor {
  def receive = {
    case PingMessage =>
        println("  pong")
        sender ! PongMessage
    case StopMessage =>
        println("pong stopped")
        context.stop(self)
  }
}

object PingPongTest extends App {
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "ping")
  // start them going
  ping ! StartMessage
}
```

6. **[11 pt.]** En el siguiente lenguaje de programación, la palabra "`local`" se usa para declarar variables en un alcance, y el operador "$\rightarrow$" se usa para ligar la variable de la derecha a la de la izquierda, de forma que el valor de la variable de la izquierda será una referencia a la de la derecha. Todas las variables se representan en la pila como punteros a una estructura de datos en el heap.

   Diagrame los diferentes estados por los que pasa la pila de ejecución y muestre en qué momento se

puede recolectar cada variable. Señale también si hay casos de variables que podrían ser recolectadas antes de que queden sintácticamente disponibles para que el recolector de basura las recolecte.

```
{ local bli
  local bla
  { local ble
    local blu
    local bla
    bla <- ble
    { local blo
      ble <- blo
    }
    bli = 3
  }
}
```

7. **[11 pt.]** Dada la siguiente base de conocimiento en Prolog, grafique o explique verbalmente cuál sería la secuencia de predicados por los que se realizaría la búsqueda para verificar si es cierto `recomendar(pedro,clara)`.

```
amigo(pedro,maria).      amigo(pedro,juan).
amigo(maria,pedro).      amigo(maria,clara).
amigo(maria,romina).     amigo(juan,pedro).
amigo(juan,clara).       amigo(clara,maria).
amigo(clara,juan).       amigo(romina,maria).

recomendar(X,NuevoAmigo) :-
  amigo(X,Amigo), amigo(Amigo,NuevoAmigo),
  not(amigo(X,NuevoAmigo)), not(X = NuevoAmigo).
```

8. **[11 pt.]** En Go existen interfaces con características semejantes a las de Java, pero con una diferencia:

   > *With how Go interfaces work, you don't need to declare an interface implementation. If you implement the proper methods, you implement the interface.*

   Relacione esta propiedad con las políticas de subtipado de diferentes lenguajes orientados a objetos, que pueden estar implementadas en herencia o en inclusión entre interfaces.

9. **[11 pt.]** Ordene los siguientes fragmentos de código de más de scripting a menos de scripting, siempre justificando por qué.

```
#!/bin/bash
for jpg; do
    png="${jpg%.jpg}.png"
    echo converting "$jpg" ...
    if convert "$jpg" jpg.to.png ; then
        mv jpg.to.png "$png"
    else
        echo 'jpg2png: error: failed output saved in "jpg.to.png".' >&2
        exit 1
    fi
done
echo all conversions successful
exit 0
```

```
proc for {initCmd testExpr advanceCmd bodyScript} {
    uplevel 1 $initCmd
    set testCmd [list expr $testExpr]
    while {[uplevel 1 $testCmd]} {
        uplevel 1 $bodyScript
        uplevel 1 $advanceCmd
    }
}
```

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::Handle;

my ( $remaining, $total );

$remaining = $total = shift(@ARGV);

STDOUT->autoflush(1);

while ( $remaining ) {
    printf ( "Remaining %s/%s \r", $remaining--, $total );
    sleep 1;
}

print "\n";
```

```
class classname : public superclassname {
 protected:
 // instance variables

 public:
 // Class (static) functions
 static void * classMethod1();
 static return_type classMethod2();
 static return_type classMethod3(param1_type param1_varName);

 // Instance (member) functions
 return_type instanceMethod1With1Parameter (param1_type param1_varName);
 return_type instanceMethod2With2Parameters (param1_type param1_varName, param2_typ
};
```