Observaciones a partir de errores encontrados en la corrección del 1er Parcial de Computación 1 – Setiembre de 2009

Observación 1 – Comparación de vectores en Matlab

Tipo: Error sintáctico.
Gravedad: Leve.

Correcto	**Incorrecto**

`if length(v) == 0`
`  disp('VACIO');`
`end`
	`if v == []`
	`  disp('VACIO');`
	`end`

Observación 2 – Asignación redundante

Tipo: Estilo de programación.
Gravedad: Leve.

Los argumentos de entrada (y los de salida) son variables locales a la función. Esto significa que cuando la función termina, estas variables se destruyen y no pueden volver a ser consultadas. No obstante, observamos con relativa frecuencia una tendencia a reasignar los argumentos de entrada a nuevas variables, para utilizarlos directamente.

Por ejemplo:

```
function d = ejemplo(a, b, c)
  aux1 = a;
  aux2 = b;
  aux3 = c;
  d = aux1*2+aux2*3+aux3*4;
```

Operar de esta manera es conceptualmente erróneo, debido a que no existe ninguna ventaja respecto al uso directo de los argumentos. Además, no hay diferencias entre los significados de **a** y **aux1**, de **b** y **aux2** y de **c** y **aux3**. Por lo tanto, lo correcto sería:

```
function d = ejemplo(a, b, c)
  d = a*2+b*3+c*4;
```

Es distinto el caso en el cual se utiliza uno de los argumentos de entrada para inicializar una variable. Por ejemplo:

```
function d = ejemplo(a, b, c)
  i = a;
  d = 0;
  while i*b < c
    d = d + i + a;
    i = i + 1;
  end
```

En este caso, la variable **i** se incrementará dentro del **while** y toma inicialmente el valor de **a**. Ambas variables deben ser independizadas porque se necesita mantener el valor de **a**.

Observación 3 – Condiciones superfluas

Tipo: Desempeño de la solución.
Gravedad: Leve.

Si la condición del **while** comprende un caso límite, no es necesario agregar condiciones (**IF-ELSE**) preventivas que lo incluyan. Por ejemplo:

Correcto

```
function z = darDistanciaCam(M, v)
z = 0;
m = length(v);
salir = 0;
i = 1;
while i <= m-1 & ~salir
  y = darDistancia(M, v(i), v(i+1));
  if y == -1
    salir = 1;
  else
    z = z + y;
  end
  i = i+1;
end
if salir
  z = -1;
end
```

Incorrecto

```
function z = darDistanciaCam(M, v)
z = 0;
m = length(v);
if  m > 1
  salir = 0;
  i = 1;
  while i <= m-1 & ~salir
    y = darDistancia(M, v(i), v(i+1));
    if y == -1
      salir = 1;
    else
      z = z + y;
    end
    i = i+1;
  end
end
if salir
  z = -1;
end
```

Si el largo del vector (asignado a **m**) fuera igual a **0** o **1**, la condición del **while** no se cumpliría debido a que inicialmente la variable **i** es igual a **1**.

Observación 4 – Cuándo usar for, cuándo usar while

Tipo: Error conceptual respecto a conocimientos básicos de programación.
Gravedad: Alta.

Cuando conocemos de antemano cuántas iteraciones se realizarán, debemos utilizar la estructura de control **for**. Cuando no conocemos de antemano cuántas iteraciones se realizarán, debemos usar la estructura **while**. Por

ejemplo, si queremos recorrer un vector de números enteros y sumarlos usamos un **for** porque sabemos *a priori* cuál es la cantidad de elementos que tiene el vector. Por otro lado, si queremos recorrer un vector de números enteros y sumarlos hasta que encontremos un **-1** usamos un **while** porque no sabemos *a priori* en qué parte del vector se encuentra el **-1**, si es que se encuentra.

Para el segundo ejemplo, el problema se podría resolver de la siguiente manera:

```
function suma = sumarHastaMenosUno(v)

  encontre = 0;
  i = 1;
  suma = 0;
  while i <= length(v) & ~encontre
    if v(i) == -1
      encontre = 1;
    else
      suma = suma + v(i);
    end
    i = i + 1;
  end
```

En este caso, sería incorrecto utilizar un **for** en lugar del **while** debido a que **es necesario salir del ciclo una vez que se encuentra el valor -1** (denominado *centinela*). De lo contrario, la computadora realizará iteraciones innecesarias. En caso de un vector con pocos elementos, el costo es bajo. Sin embargo, si el vector tuviera millones de elementos y el centinela se encontrara entre los primeros lugares, el desperdicio computacional sería muy grande.

Por esta razón, el siguiente código es considerado erróneo **a pesar de retornar una solución válida**:

```
function suma = sumarHastaMenosUno(v)

  encontre = 0;
  suma = 0;
  for i = 1:length(v)
    if ~encontre
      if v(i) == -1
        encontre = 1;
      else
        suma = suma + v(i);
      end
    end
  end
```

Observación 5 – Pérdida del valor del acumulador

Tipo: Error conceptual respecto a conocimientos básicos de programación.
Gravedad: Muy alta.

Con frecuencia, la confusión en el uso del **while** y el **for** viene acompañada de otro error: sobrescribir el acumulador en cada iteración:

```
function suma = sumarHastaMenosUno(v)

  encontre = 0;
  suma = 0;
  for i = 1:length(v)
      if v(i) ~= -1
        suma = suma + v(i);
      end
  end
```

Este error es más grave que el anterior, ya que el programa no garantiza la devolución de una solución válida.

Observación 6 – Inicialización de variables

Tipo: Error conceptual respecto a conocimientos básicos de programación.
Gravedad: Media-Alta.

Observamos una tendencia a confundir la frontera entre inicialización de variables y modificación de variables dentro del ciclo de repetición (**while** o **for**). La práctica consiste en acumular parte de los resultados fuera del ciclo, cuando deberían acumularse dentro de él.

Por ejemplo, supongamos que debemos resolver el siguiente problema: *Escribir una función en Matlab que tome dos vectores **v** y **w** del mismo tamaño y devuelva la posición **i** en que la suma de **v(i)** y **w(i)** sea igual a **15**. Si esto no ocurre, la función debe devolver **0**.*

Propuesta de solución (incorrecta):

```
function y = suma15(v, w)

  n = length(v);
  i = 1;
  suma = v(1) + w(1);
  while suma ~= 15 & i <= n
    suma = v(i) + w(i);
    i = i+1;
  end
  if suma==15
    y = i-1;
  else
    y = 0;
  end
```

Cuando i vale 1, `suma = v(1) + w(1)`. Esta condición ya se evaluó.

La solución propuesta no es correcta, debido a que la primera vez que se ejecuta el ciclo se vuelve a calcular un valor de **suma** que ya fue calculado. Además, no considera el caso en que el largo del vector fuera **0**.

Un refinamiento demasiado apresurado para solucionar este problema consiste en incrementar la variable `i` antes de calcular la suma.

Refinamiento (incorrecto):

```
function y = suma15(v, w)

  n = length(v);
  i = 1;
  suma = v(1) + w(1);
  while suma ~= 15 & i <= n
    i = i+1;
    suma = v(i) + w(i);
  end
  if suma==15
    y = i-1;
  else
    y = 0;
  end
```

Cuando `i` llega a `n`, el intérprete ingresa en el **while**, ingrementa `i` y calcula `suma = v(n+1) + w(n+1)`.

Esta solución no es correcta debido a que no contempla un caso límite. El programa falla cuando se llega al final del vector. Por lo tanto, habría que cambiar la condición del **while** para que se detenga cuando `i < n`. De todas formas, seguiría sin contemplarse la posibilidad de que el vector fuera vacío.

Una tercera alternativa sería iniciar con `i = 2` y `suma = v(1) + w(1)`. Sin embargo, no escapamos al problema de que estamos colocando parte del cálculo fuera del **while** (¿por qué no colocar el caso de `i = 3`, `i = 4` y así sucesivamente, fuera del **while**?).

Todo se simplifica si consideramos el caso "cero": definimos que para un hipotético `i = 0`, suma va a ser igual a 0. Como queremos que el **while** jamás entre en este caso, tenemos que elegir un valor para suma que sea distinto de **15**. En caso de que el vector esté vacío, la condición del **while** dará falso y, finalmente, el valor a retornar va a ser 0.

Solución correcta:

```
function y = suma15(v, w)

  n = length(v);
  suma = 0;
  i = 1;
  while suma ~= 15 & i <= n
    suma = v(i) + w(i);
    i = i+1;
  end
  if suma==15
    y = i-1;
  else
    y = 0;
  end
```

Observación 7 – Evaluación condicional de funciones

Tipo: Desempeño de la solución.
Gravedad: Leve.

Ocasionalmente, la decisión de evaluar una función depende de un determinado resultado. Si el resultado cumple determinada condición, la función debe ser evaluada. Si no cumple la condición, la evaluación no debe realizarse.

Para el problema 1 (c) del parcial[1], observamos con mucha frecuencia la siguiente solución:

```
function l = buscarLineaConDefectos(M, d_cerca, d_lejos)

    [m n] = size(M);
    i = 1;
    l = 0;
    encontre = 0;
    while i <= m & ~encontre
        v = paradasCercanas(M(i,:),d_cerca);
        w = paradasLejanas(M(i,:),d_lejos);
        if length(v) ~= 0 | length(w) ~= 0
            encontre = 1;
            l = i;
        end
        i = i+1;
    end
```

Cada vez que se repite el ciclo, se evalúan las funciones **paradasCercanas** y **paradasLejanas**. Si **paradasCercanas** no fuera vacío, ¿para qué evaluar **paradasLejanas**? Evaluar una función es relativamente costoso; deberían evitarse todas las evaluaciones innecesarias.

[1] Setiembre de 2009, Primer Parcial.

Solución mejorada:

```
function l = buscarLineaConDefectos(M, d_cerca, d_lejos)

    [m n] = size(M);
    i = 1;
    l = 0;
    encontre = 0;
    while i <= m & ~encontre
        w = paradasCercanas(M(i,:),d_cerca);
        if length(w) ~= 0
            encontre = 1;
            l = i;
        else
            w = paradasLejanas(M(i,:),d_lejos);
            if length(w) ~= 0
                encontre = 1;
                l = i;
            end
        end
        i = i+1;
    end
```

Esto debe generalizarse para cualquier porción del código que deba ejecutarse si y solo si se cumple una determinada condición. Por ejemplo, la siguiente solución al Problema 2 (a) siempre ejecuta el segundo **while**. Si en el primer **while** no se encontró el origen, ¿para qué buscar el destino en el segundo **while**?

Propuesta (incorrecta):

```
function d = darDistancia(M, origen, destino)

  [n,m] = size(M);
  i = 2;
  d = -1;
  while i <= n & M(i,1) ~= origen
      i = i+1;
  end
  j = 2;
  while j <= m & M(1,j) ~= destino
      j = j+1;
  end
  if i <= n && j <= m
      d = M(i,j);
  end
```

Propuesta mejorada:

```
function d = darDistancia(M, origen, destino)

  [n,m] = size(M);
  i = 2;
  d = -1;
  while i <= n & M(i,1) ~= origen
      i = i+1;
  end
  if i <= n
      j = 2;
      while j <= m & M(1,j) ~= destino
          j = j+1;
      end
      if j <= m
          d = M(i,j);
      end
  end
```

Otra alternativa (basada en que la matriz M es cuadrada):

```
function y = darDistancia(M, origen, destino)

  [n,m] = size(M);
  c = 0;
  d = 0;
  i = 2;
  while i <= n & (c == 0 | d == 0)
      if M(i,1) == origen
          c = i;
      end
      if M(1,i) == destino
          d = i;
      end
      i = i+1;
  end
  if (c == 0 | d == 0)
      y = -1;
  else
      y = M(c,d);
  end
```

Observación 8 – Múltiples invocaciones a una misma función

Tipo:	Desempeño de la solución.
Gravedad:	Leve.

Cuando la condición del **while** depende de la invocación de una función que, a su vez, se utilizará en el propio cuerpo del **while** debemos utilizar una variable auxiliar para evitar llamar a la función dos (o más) veces.

Incorrecto:

```
function d = darDistanciaCam(M, camino)

  n = length(camino);
  d = 0;
  i = 1;

  while i < n & darDistancia(M,camino(i),camino(i+1)) ~= -1
      d = d + darDistancia(M,camino(i),camino(i+1));
      i = i+1;
  end

  if i ~= n
      d = -1;
  end
```

Correcto:

```
function d = darDistanciaCam(M, camino)

  n = length(camino);
  d = 0;
  i = 1;
  aux = 0;

  while i < n & aux ~= -1
      aux = darDistancia(M,camino(i),camino(i+1));
      d = d+aux;
      i = i+1;
  end

  if aux == -1
```

```
        d = -1;
    end
```

Observación 9 – Indentación del código

Tipo: Legibilidad del código generado.
Gravedad: Media.

Una vez que logramos entender el problema y estamos en condiciones de proponer un programa para resolverlo debemos enfrentarnos a dos restricciones:

El código debe ser legible por el intérprete (por ejemplo, Matlab), o el compilador.

El código debe ser legible por un humano.

Para que el programa funcione tenemos que respetar la primera restricción, pero eso no significa que la segunda no sea igual de importante. A veces escribimos programas y los volvemos a consultar mucho tiempo después. Si el programa no es legible, el esfuerzo por interpretarlo puede llegar a ser muy grande. Es importante, por lo tanto, indentar las líneas de código y, por supuesto, documentarlas adecuadamente.