

# Enunciado.pdf



Juanma21\_



Teoría de Los Lenguajes de Programación



4º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingeniería Informática  
Universidad de Málaga



MÁSTER EN

## Inteligencia Artificial & Data Management

MADRID

Formamos  
**talento** para un futuro  
**Sostenible**

saber más



Esto no son apuntes pero **tiene un 10 asegurado** (y lo vas a disfrutar igual).

Abre la Cuenta NoCuenta con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Me interesa

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en [ing.es](https://www.ing.es)



UNIVERSIDAD  
DE MÁLAGA

E.T.S.I. INFORMÁTICA

## Lenguajes de Programación

### 4.º del Grado en Ingeniería Informática (Computación)

7 de septiembre de 2020

APELLIDOS, NOMBRE: \_\_\_\_\_

- Para la evaluación del examen se valorarán la **corrección** y la **claridad** de las soluciones
- Solo se modifican y entregan los ficheros `NaturalSemantics2020.hs` y `StructuralSemantics2020.hs`
- Las definiciones semánticas y la implementación Haskell deben hacerse en las **secciones** señaladas mediante **comentarios** en el fichero correspondiente.

Tomaremos como base el lenguaje WHILE. El fichero `While2020.hs` contiene los tipos algebraicos para representar la sintaxis abstracta, así como las funciones semánticas `aVal` y `bVal` para evaluar expresiones aritméticas (**Aexp**) y Booleanas (**Bexp**), respectivamente.

El fichero `NaturalSemantics2020.hs` contiene la definición de la semántica natural de WHILE. El fichero `StructuralSemantics2020.hs` contiene la definición de la semántica estructural operacional de WHILE.

**Problema 1.** (1.0 + 1.0 + 0.5 ptos.) Una posible optimización al evaluar expresiones aritméticas del lenguaje **Aexp** consiste en reemplazar las subexpresiones constantes (es decir, aquellas en las que todos los operandos son constantes) por su valor. Por ejemplo, la expresión  $x + 5 * 3$  puede ser reemplazada por la expresión equivalente  $x + 15$ . De la misma forma, podemos reemplazar la expresión  $8 * y + (3 * 2 + 5)$  por la expresión  $8 * y + 11$ .

- Define formalmente una función **reduce**: **Aexp**  $\rightarrow$  **Aexp** que dada una expresión *a* devuelva una expresión *a'* equivalente en la que las subexpresiones constantes han sido reemplazadas por su valor.
- Implementa la función **reduce** en Haskell (en el fichero `NaturalSemantics2020.hs`).
- Enuncia y demuestra formalmente que la optimización propuesta por **reduce** es correcta; es decir, que preserva la semántica de **Aexp**. Solo es necesario demostrar los casos base y uno de los casos inductivos.

**Problema 2.** (1.0 + 1.0 ptos.) Añade al lenguaje WHILE la sentencia `swap`, que intercambia el valor de dos variables. La sintaxis de la sentencia `swap` es:

$$S ::= \text{swap } x \ y$$

- Define e implementa la semántica natural para `swap`.
- Define e implementa la semántica estructural operacional para `swap`.

**Problema 3.** (1.0 + 1.0 ptos.) Añade al lenguaje WHILE una sentencia iterativa `for` semejante al `for` de Java. La sintaxis de la sentencia `for` es:

$$S ::= \text{for}(S_1; b; S_2) S_3$$

donde *S*<sub>1</sub> es una sentencia que se evalúa siempre una sola vez antes de la ejecución del bucle; *b* es la guarda pre-comprobada del bucle, *S*<sub>3</sub> es el cuerpo del bucle y *S*<sub>2</sub> es una sentencia que se ejecuta después de cada iteración del bucle.

Consulta  
condiciones aquí



do your thing

WUOLAH

- Define e implementa la semántica natural para `for`.
- Define e implementa la semántica estructural operacional para `for`.

**Problema 4.** (1.0 + 1.0 *ptos.*) Demuestra que las sentencias `WHILE`:

```
if b then (S1; S) else (S2; S)
```

y

```
if b then S1 else S2; S
```

son semánticamente equivalentes según:

- la semántica natural
- la semántica estructural operacional

Puedes suponer que las sentencias `S1`, `S2` y `S` terminan en ambos casos.

**Problema 5.** (1.5 *ptos.*) Se desea añadir a `WHILE` una nueva sentencia iterativa no determinista cuya sintaxis es:

$$\begin{aligned} S &::= \text{do } G_S \text{ od} \mid \dots \\ G_S &::= b \rightarrow S; G_S \mid \varepsilon \end{aligned}$$

donde  $b \rightarrow S$  es una sentencia guardada:  $S$  solo se puede ejecutar si la guarda  $b$  que la precede es cierta. Para ejecutar una iteración no determinista se procede de la siguiente manera:

- Si hay varias sentencias guardadas cuya guarda sea cierta, se selecciona una de ellas de manera no determinista y se ejecuta esta sentencia; el resto de sentencias guardadas se ignora y se vuelve a ejecutar el bucle
- Si no hay ninguna sentencia guardada cuya guarda sea cierta, el bucle termina su ejecución.

Por ejemplo, dada la siguiente iteración no determinista:

```
do
  x > y  -> y := y + 1;
  x > y  -> y := y + 3;
od
```

Si  $x = 3, y = 0$ , son posibles tres resultados:  $y = 3$ ,  $y = 4$ , o  $y = 5$

Define formalmente la semántica natural de la iteración no determinista en el comentario apropiado del fichero `NaturalSemanticsWhile2020.hs`. No es necesario implementar la regla en Haskell.