TEMA 3: SEMÁNTICA AXIOMÁTICA

1. Semántica axiomática de un lenguaje de programación
2. Tipos abstractos de datos

1. Semántica axiomática de un lenguaje

- Para poder verificar un programa es necesario conocer la semántica del lenguaje en el que está escrito .
- Vamos a ver la semántica de un lenguaje inventado muy simple, el *lenguaje while (D. Gries, "The Science of programming")*.
- Para cada instrucción del lenguaje veremos cuál es su pmd y cuando dicha instrucción será correcta en función de su precondición y postcondición.

1. Semántica axiomática de un lenguaje

- **Instrucción nula:**
  - Instrucción que no hace nada.
  - Existe en la mayoría de los lenguajes.
  - En el lenguaje *while* se llama **seguir**.
- **Semántica de seguir:**

  pmd(seguir, A) ≡ A
- **Por tanto, el siguiente programa será correcto:**

```
{A}
  seguir
{A}
```


1. Semántica axiomática de un lenguaje

- **Ejemplo:**

```
{x >= 1}
  seguir
{x >= 1}
```

- **Si tenemos un aserto A1 más fuerte que A, también será correcto:**

```
{A1}
  seguir
{A}
```

Ejemplo:

```
{x = 1}
  seguir
{x >= 1}
```


1. Semántica axiomática de un lenguaje

- **Instrucción composición:**
  - Es la composición secuencial de instrucciones (estructura secuencial).
  - Sirve para unir dos instrucciones en una sola.
  - Lo representaremos mediante el operador ';'.

  Ejemplo:

```
I1;
I2
```

- **Semántica:**

  pmd("I1;I2", A3) ≡ pmd(I1, pmd(I2, A3))

1. Semántica axiomática de un lenguaje

- **Verificación de la instrucción composición:**

```
{A1}
  I1;
  I2
{A3}
```

  El programa es correcto si existe un aserto A2 tal que ({A1}I1{A2}) ∧ ({A2}I2{A3})
- **Ejemplo:**

```
{x = 1}
 seguir;
 seguir
{x = 1}
```

  ¿Es correcto? Demostrarlo

1. Semántica axiomática de un lenguaje

- **La instrucción composición cumple la propiedad asociativa:**

```
{A1}(I1;I2);I3{A4}
```

Es lo mismo que:

```
{A1}I1;(I2;I3){A4}
```

Por tanto, no es necesario el uso de paréntesis:

```
{A1}
 I1;
 I2;
 I3
{A4}
```


1. Semántica axiomática de un lenguaje

- **Asignación:**
  - La representaremos mediante el símbolo '**:=**'.

  Ejemplo:

```
x := Expresion
```

  El tipo de la variable y de la expresión han de coincidir.
- **Semántica:**
  - Expresión sin operaciones parciales:

    pmd("x := E", A) ≡ A[x ← E]
  - Expresión con operaciones parciales:

    pmd("x := E", A) ≡ A[x ← E] ∧ Cond_de_dominio

1. Semántica axiomática de un lenguaje

- **Verificación de la instrucción asignación:**

```
{A1}
  x := E;
{A2}
```

El programa es correcto si cumple

A1 ⇒ A2 [x ← E] ∧ Cond_de_dominio

- **Ejemplo:**

```
{x >= 0}
 x := x + 1
{x > 1}
```

¿Es correcto? Demostrarlo

1. Semántica axiomática de un lenguaje

Def (condiciones de dominio): Condiciones que garantizan que una expresión con operaciones parciales se puede evaluar.

Ejemplo:

```
¿Pre?
  x := x/b;
{x > 1}
```

Hallar la pmd que garantiza que el programa sea correcto.

Ejemplo 2:

```
¿Pre?
  x := x + 1;
{y > 1}
```

Ejercicio: Escribir la especificación de un programa para intercambiar los valores de las variables *x* e *y*. A continuación escribir un programa que intercambie las variables utilizando una variable auxiliar y verificar que es correcto.

1. Semántica axiomática de un lenguaje

- **Asignación múltiple:**
  - Realiza varias asignaciones simultáneas.

  Ejemplo:

```
<x1,...,xn> := <E1,...,En>
```

- **Semántica:**

pmd("<x1,..,xn> := <E1,...,En>", A) ≡

A[x1,...,xn ← E1,...,En] ∧ Cond_de_dominio

Ejemplo:

```
¿Pre?
  <x,y> := <7, 2*x>;
{x < y}
```


1. Semántica axiomática de un lenguaje

- **Llamada a función con especificación Pre/Post:**
  - Es un caso especial de asignación, donde la expresión es una función.
  - Para realizar la llamada, habrá que garantizar que los parámetros de entrada cumplen la precondición de la función.
- **Semántica:**

  Sea la especificación de la función:

```
func f(p1, ...,pk) dev r1, ..., rm
{Pre: A1(p1, ..., pk)}
{Post: A2(p1, ..., pk, r1, ..., rm}
```

  La llamada a la función tendrá la siguiente precondición y postcondición:

```
{A1[p1, ..., pk ← e1, ..., ek]}
  <x1, ..., xm> := f(e1, ..., ek)
{A2[p1, ..., pk, r1, ..., rm ← e1,..., ek, x1, ..., xm]}
```

1. Semántica axiomática de un lenguaje

- **Ejemplo:**

```
func dividir(a, b: Nat) dev c, r: Nat
{Pre: b > 0}
{Post: (a = b * c + r) ∧ (r < b) }
```

Precondición y postcondición de una llamada a la función dividir.

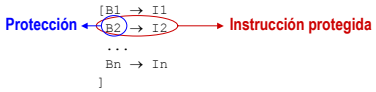
```
{i+1 > 0}
  <k, l> := dividir(2*p, i+1)
{(2*p = (i+1) * k + l) ∧ (l < i+1) }
```

Si la postcondición posee expresiones que no dependen de las variables incluidas en la asignación, estas se mantienen en la precondición.

```
{i+1 > 0 ∧ (p > i)}
  <k, l> := dividir(2*p, i+1)
{(2*p = (i+1) * k + l) ∧ (l < i+1) ∧ (p > i)}
```


1. Semántica axiomática de un lenguaje

- **Instrucción alternativa:**
  - Permite elegir una instrucción de entre varias. Similar a la instrucción *switch* de C++.
  - Sintaxis:



  - Ejecución de la instrucción:
    a) Se evalúan todas las protecciones.
    b) Si todas están cerradas, el programa aborta.
    c) Si hay alguna abierta, se selecciona una de ellas y se ejecuta la instrucción asociada.

1. Semántica axiomática de un lenguaje

- **Semántica:**

pmd("[B1 → I1,..., Bn → In]", A) ≡

(∃ i:1 ≤ i ≤ n:Bi) ∧ cond_dominio(Bi) ∧ (∀ i:1 ≤ i ≤ n:Bi ⇒ pmd(Ii, A))

- **Verificación de la instrucción alternativa:**

```
{A1}
    [B1 → I1
     ...
     Bn → In
    ]
{A2}
```

1. Todas las protecciones se pueden evaluar: A1 ⇒ cond_dominio(Bi)
2. Al menos una de las protecciones está abierta: A1 ⇒ ∃ i:1 ≤ i ≤ n:Bi
3. Cuando se ejecuta una rama cualquiera, se acaba cumpliendo la post.: ∀i:1 ≤ i ≤ n: {A1 ∧ Bi} Ii {A2}

1. Semántica axiomática de un lenguaje

- **Verificación instrucción alternativa con dos ramas:**

```
{A1}
    [B1 → I1
     B2 → I2
    ]
{A2}
```

1. Todas las protecciones se pueden evaluar:
   A1 ⇒ cond_dominio(Bi)
2. Al menos una de las protecciones está abierta:
   A1 ⇒ (B1 ∨ B2)
3. Cuando se ejecuta una rama cualquiera, se acaba cumpliendo la post.:
   {A1 ∧ B1} I1 {A2}
   {A1 ∧ B2} I2 {A2}

1. Semántica axiomática de un lenguaje

- **Ejercicio:** Verificar el siguiente programa en C++. Para ello se deberá traducir antes al lenguaje *while* estudiado.

```
{Pre: a = A}
    if(x == 0)
      a = a + 1;
    else
      a = a + x;
{Post: (x=0 ∧ a=A+1) ∨ (x≠0 ∧ a=A+x)}
```


1.1. Derivación de algoritmos

- **Derivación de algoritmos:**
  - Consiste en deducir el programa a partir de la especificación del algoritmo.
  - La derivación se realiza desde el final hacia el principio del programa, es decir, partiendo de la postcondición vamos calculando instrucciones hasta llegar a la precondición.
  - Para elegir la instrucción hay que fijarse en la forma de la postcondición:
    - Igualdades → Asignaciones
    - Disyunciones → Instrucción alternativa
  - Si obtenemos como precondición falso, significa que nos hemos equivocado al elegir alguna instrucción.

1.1. Derivación de algoritmos

- **Ejercicio 1:** Mantenimiento de un valor constante.

```
{Pre: (x + y = T) ∧ z = Z}
{Post: (x + y = T) ∧ (x = z) ∧ z = Z}
```

- **Ejercicio 2:** Intercambio de valores mediante asignaciones simples.

```
{Pre: (x = X) ∧ (y = Y)}
{Post: (x = Y) ∧ (y = X)}
```

- **Ejercicio 3:** Máximo de dos valores.

```
{Pre: cierto}
{Post: (z = x ∨ z = y) ∧ (z >= x) ∧ (z >= y)}
```