Algo 1 - Teórico 2020-10-20

Repaso: Programación Imperativa

- Programación funcional: un programa es una expresión, y la ejecución de un programa es la reducción de esa expresión a una forma canónica.
- Programación imperativa: un programa es una “receta” o “serie de pasos”, y la ejecución de un programa parte de un **estado** inicial y lo va transformando hasta llegar a un **estado** final.

- Estado: es una asignación de valores a un conjunto fijo y predeterminado de variables. En un programa, el estado se define al principio en lo que se llama la “declaración de constantes y variables”.

Programa de ejemplo:

```
Var x, y : Int        ← declaración de variables
x := 3;
y := x + 3
```

Ejemplo de ejecución del programa a partir de un estado concreto (en forma de texto):

|[x → 4, y → 8]| → notación de texto para hablar del estado
`x := 3 ;`
|[x → 3, y → 8]|
`y := x + 3`
|[x → 3, y → 6]|

Observación: Acá el estado final va a ser siempre x→ 3, y → 6 sin importar cuál sea el estado final.

Otro ejemplo:

|[x → 3, y → 8]|
`y := (x + 3) * y`
|[x → 3, y → 48]|

Otro ejemplo:

|[n → 10]|
`n := n + 1`
|[n → 11]|

Otro ejemplo (asignación múltiple):

|[x → 4, y → 8]|
```
 x, y := 3, x + 3;
```
|[x → 3, y → 7]|

En la asignación múltiple, primero se calculan todos los valores a asignar **usando el estado anterior**. No es equivalente a: x := 3; y := x + 3 . Observación: en C la asignación múltiple no existe.

Sintaxis vs. Semántica

Sintaxis: ¿Cómo se escriben los programas? Un programa es un texto (una secuencia de letras). La sintaxis de un lenguaje me dice qué textos son programas válidos.

Semántica: ¿Qué significan? ¿Qué hacen? Un programa se puede ejecutar, y esa ejecución tiene efecto en un "mundo semántico". En el caso de los programas imperativos, este mundo semántico es el **estado**.

Programación Imperativa

Sintaxis de un programa. Dos secciones principales:

1. Declaración de constantes y variables
2. Sentencia(s) del programa

Semántica de un programa:

- Estado: variables y constantes definidas por las declaraciones.
- Ejecución: Dado un estado inicial, ejecutar las sentencias del programa, modificando el estado hasta llegar a un estado final.

Diferencias con funcional: En imperativo (del teórico) no hay funciones y **no hay un valor resultado**. El **resultado es el estado final entero**.

Diferencias con C: En imperativo (del teórico) no hay funciones ni procedimientos. Sólo estamos tomando de C lo que se llama programación "in the small" (en pequeña escala): Los algoritmos pequeños, sin la superestructura de los programas que podemos definir en C.

Declaración de constantes y variables

Especificador: `Var` o `Const`. (para avisar si declaro variable o constante)

Nombres (identificadores): Palabras con letras y números (pero empezando con una letra).

Tipos: `Nat, Int, Real, Bool, Char,` arreglos. **No hay listas**.

Sintaxis:

```
Var/Const   nombre1, nombre2, ... , nombren  :  Tipo;
   ...
Var/Const   nombre1, nombre2, ... , nombren  :  Tipo;
```

Ejemplo:

```
Var x, y : Int;
Var resultado : Bool;
```

Otro ejemplo:

```
Const divisor, dividendo : Int;
Var cociente, resto : Int;
```

Observación: El valor de las constantes no estará determinado en los programas (no lo escribiremos como parte de la sintaxis), si no que podrá ser cualquier valor (en principio). Luego veremos cómo se podrá restringir los valores posibles. (por ejemplo: me interesa que el divisor no sea cero).

Otro ejemplo (con variables de varios tipos en la misma línea):

```
Var x : Int, b : Bool;
```

Sentencias

Dos tipos:

- Elementales: sentencias básicas que no necesitan de otras sentencias.
- Compuestas: sentencias que se arman a partir de otras sentencias (o sea, sentencias que tienen adentro otras sentencias).

Skip

Es una sentencia que no hace nada.

Ejemplo:
|[x → 3, y → 6]|

`skip`
|[x → 3, y → 6]|

Sintaxis:

`skip`

Semántica: No modifica el estado.

Asignación (:=)

Asignar valores a variables.

Sintaxis:

`v1, v2, …, vn := E1, E2, …, En`

a donde `v1, v2, …, vn` son variables declaradas del programa, y `E1, E2, ..., En` son expresiones válidas en lenguaje y del tipo que se corresponde con las variables.

Semántica:

1. Se calculan valores-resultado para todas las expresiones E1, E2, …, En. (O sea, se hace usando el estado actual, no el nuevo estado.)
2. Se modifica el estado, asignando a cada variable vi el resultado de calcular la expresión Ei.

Expresiones válidas en programación imperativa:

- valores constantes
- variables y constantes declarados
- operaciones básicas (+, -, *, /, div, mod, max, min, ∧, ∨, ¬, =, ≠, ≤, <, >, etc.)
- **NO: cuantificadores ni cosas que esconden cuantificadores (factorial, exponenciación, etc.)**

Secuenciación (;)

Primer sentencia compuesta que vemos. La secuenciación me permite ejecutar una sentencia y después otra.

Ejemplo (ya visto):

|[x→ 4, y → 8]|

`x := 3 ; y := x + 3`

|[x→ 3, y→ 6]|

Se arma a partir de las dos sentencias: `x := 3, y := x + 3.`

Sintaxis:

`S1 ; S2`

a donde S1 y S2 son dos sentencias.

Semántica:

1. Ejecutar S1 a partir del estado inicial.
2. Ejecutar S2 a partir del estado que resulta del paso 1. (el estado queda como lo deja la ejecución de S2).

Observaciones:

- La secuenciación es **asociativa**:

Es lo mismo (semánticamente)

$$(S1 ; S2) ; S3$$

que

$$S1 ; (S2 ; S3)$$

Luego, podremos escribir directamente:

$$S1 ; S2 ; S3$$

- Un programa es en realidad siempre **una sola sentencia**. Cuando hablamos de varias sentencias en realidad éstas forman una sola sentencia a través de la secuenciación.

Condicional (if)

Sentencia compuesta. El condicional me permite ejecutar diferentes sentencias dependiendo de una condición booleana.

Ejemplo (valor absoluto):

```
|[ x → -10 ]|
  if x ≥ 0 →
    skip
  □ x < 0 →
    x := -x
  fi
|[ x→ 10 ]|
```

Sintaxis:

```
if B1 → S1
□ B2 → S2
…
□ Bn → Sn
fi
```

a donde B1, B2, …, Bn (llamadas **guardas**) son expresiones de tipo booleano, y S1, S2, …, Sn son sentencias.

Semántica:

1. Se evalúan todas las guardas B1, …, Bn usando el estado actual. (se obtienen valores booleanos).
2. Se elige alguna de las guardas que da True. Si ninguna da True, el programa termina con error (“se rompe”). Supongamos que elegimos la k-ésima guarda (Bk = True).
3. Ejecutar esa rama del if: Sk. (el estado final es el que resulta de esta ejecución).

Observación: **No determinismo:** Puede haber varias guardas que den True. En ese caso, la semántica dicta que se puede ejecutar cualquiera de esas (pero siempre una y sólo una). La semántica no me dice cuál va a ser (no necesariamente es la primera). El programador no elige.

Ejemplo de no determinismo:
|[x → 0]|
```
if x ≥ 0 →
  x := 7
□ x ≤ 0 →
  x := x + 1
if
```
No puedo saber exacto el estado final. Puede ser |[x → 7]| o bien |[x→ 1]|

Repetición, ciclo o bucle (do)

Sentencia compuesta. Me permite repetir la ejecución de una sentencia mientras se cumpla una condición.

Ejemplo:
```
  |[ x → 3 ]|
  do x ≠ 0 →
     |[ t0: x → 3 ]|  |[ t2: x → 2 ]|  |[ t4: x → 1 ]|
      x := x - 1
     |[ t1: x → 2 ]|  |[ t3: x → 1 ]|  |[ t5: x → 0 ]|
  od
|[ t6: x → 0 ]|
```
Semántica: “Mientras valga que x ≠ 0, ejecutar x := x - 1.”

Sintaxis:

do B → S od

a donde B es una expresión booleana (**guarda**), y S es una sentencia (**cuerpo del ciclo**).

Semántica:
1. Se evalúa la guarda B en el estado actual.
2. Si da False, termina la ejecución.
   Si da True, modifica el estado ejecutando S, y luego vuelve al punto 1.

Observación: Si la guarda nunca llega a dar False, la repetición se ejecuta infinitamente, cosa que es un error.

Ejemplo: “cuántos p”

Dado un predicado p : Int → Bool y un número N > 0, queremos contar cuántos números

entre 0 y N satisfacen p. (observación: podemos usar p en las expresiones. p puede ser, por ejemplo, un predicado que me dice si un número es par, o si es un número primo, etc.)

Idea del algoritmo: Recorrer usando una repetición, todos los números desde el 0 hasta el N. Para cada valor, fijarme si se cumple p. En caso afirmativo, sumar 1 a una variable en la que voy a guardar el resultado.

```
Const N : Int;
Var resultado : Int;
Var numero_actual : Int;

resultado, numero_actual := 0, 0 ;
do numero_actual ≤ N ->
   if p.numero_actual →
     resultado := resultado + 1
   □ ¬ p.numero_actual →
     skip
   fi  ;
   numero_actual := numero_actual + 1
od
```