FILMINAS COMENTADAS
SINTÁXIS Y SEMÁNTICA DE LENGUAJES

INTRODUCCIÓN

El presente documento, toma como hilo conductor las filminas de clase e incluye notas y ejemplos que pretenden explicar sintéticamente los principales contenidos de la asignatura. Es una guía, que debe ser ampliada luego con la lectura bibliográfica, el material digital complementario propuesto, la resolución de trabajos prácticos y la indagación personal y autónoma por parte de los alumnos.

SEMÁNTICA DE ENUNCIADOS

La mayor parte de los lenguajes de programación definen estructuras sintácticas para construir:

- ✓ enunciados declarativos
- ✓ enunciados de asignación
- ✓ enunciados de control de flujo (bifurcaciones o estructuras de decisión, repeticiones o estructuras iterativas, llamadas a funciones, procedimientos o corrutinas, saltos explícitos a otras áreas de código)
- ✓ enunciados de Input/Output que manejan la entrada y salida desde y hacia el programa

En los apartados que siguen abordaremos en detalle los aspectos semánticos de cada uno de estos tipos de enunciado y/o bloques de enunciados.

Declaraciones

Una declaración es un enunciado de programa del tipo

<tipo_de_dato><identificador> ['('<lista_argumentos>')']

Las declaraciones sirven para comunicar al traductor información sobre el nombre y tipo de dato de las variables, constantes o funciones/procedimientos/operaciones que se van a utilizar.

Por su posición en el programa *(por ej.: dentro de una función)* indican también el ámbito o scope, el alcance y el tiempo de vida deseado para la variable, constante u operación.

Ejemplo

```
void main() {
        int X1;
        X1 = X1 + 1;
        …
        Z = 1;              MAL: no es accesible desde aquí
        if (X1 > 0){
                int Z;
                Z = X1;
        }
}
```

La información proporcionada en las declaraciones es incorporada en el proceso de traducción a la ***tabla de símbolos.***

Identificador	***Tipo***	***Dir. Memoria***	***Scope***	***¿Puedo Modificarlo?***
X1	*Int*	*FFA1*	*main*	*true*
Z	*Int*	*FF00*	*12*	*true*

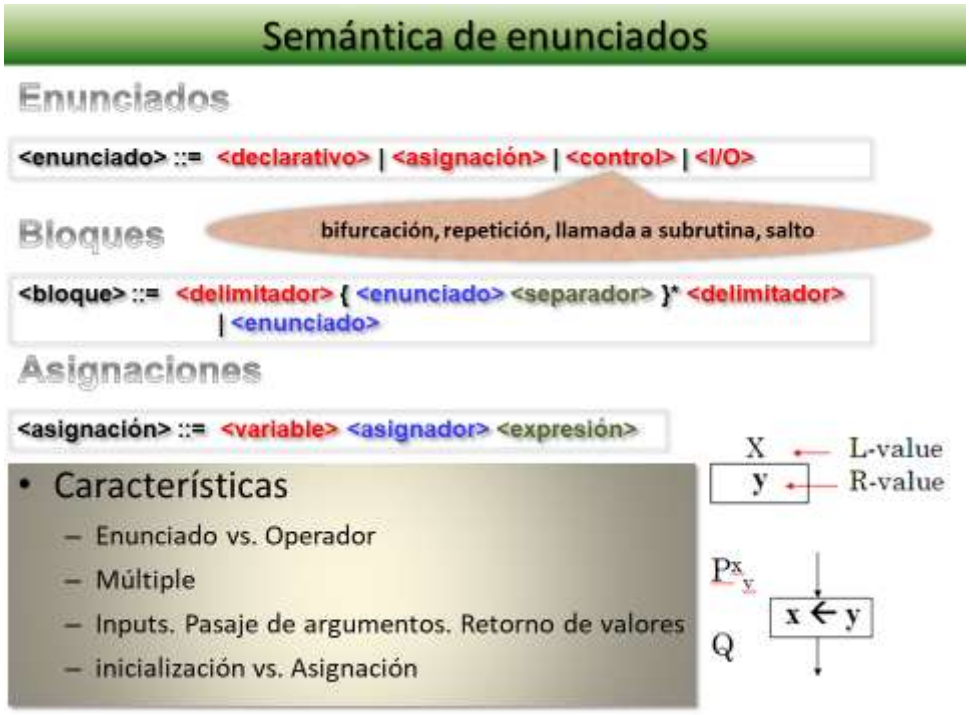
Luego el analizador semántico se vale de esta información para producir código objeto que haga una manipulación correcta de los datos almacenados.

Usando la tabla de símbolos, el analizador semántico verifica el cumplimiento de una variedad de reglas de buena formación de los programas que no pueden ser expresadas en las estructuras sintácticas definidas por gramáticas independientes del contexto y los árboles de sintaxis, tales como:

- Que cada identificador sea declarado antes de ser usado (si el lenguaje lo exige)
- Que los identificadores no sean utilizados en ámbitos inadecuados.
- La verificación de tipos en las expresiones y pasaje de argumentos a funciones
- etc.

Asignación

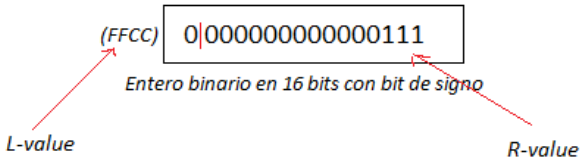
Toda instrucción de asignación presenta una parte izquierda (Left) relativa a ubicaciones de memoria y expresiones a la derecha (Right) relativas a valores.



La semántica de un enunciado de asignación vincula un **L-value** (valor a la izquierda) a un **R-value** (valor a la derecha). El L-value es siempre una dirección de memoria que referencia a la celda o bloque de memoria donde se almacenará el valor representado por el R-value. El R-value es un valor contante, una variable o una expresión cuya evaluación dará lugar a un único valor (simple o estructurado) que será alojado en el bloque de memoria referenciado por el L-value.

Ejemplo

X = 7



En un lenguaje particular, deberemos diferenciar si la asignación se implementa mediante un operador (=), mediante un enunciado o en ambas formas.

Javascript	*X=7;*
Cobol	*SET X TO 7;*

Algunos lenguajes soportan asignación múltiple

X=W=Z=3; // se asigna a X,W y Z el valor 3

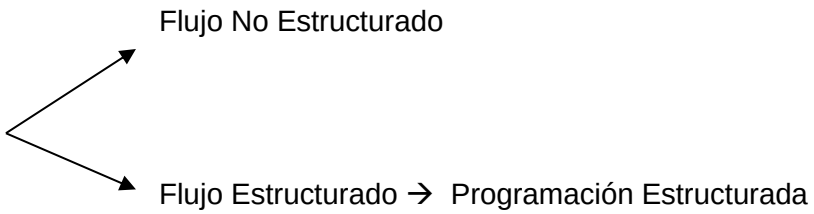
Aunque no veamos explícitamente el operador o el enunciado de asignación, hay asignación en enunciados de Entrada (input), en el pasaje de argumentos a funciones o procedimientos o en el retorno de valores.

Control de flujo de ejecución

La semántica de control de flujo se ocupa de determinar en qué orden deben ejecutarse los enunciados y bloques de enunciados de un programa.

Los lenguajes que vemos en esta materia son lenguajes **imperativos** y caen dentro de las arquitecturas de Von Newman. Esto implica que llegan a la solución de un problema a través de un proceso de transformación de los datos o estados de la memoria. Es por esto que la asignación es la operación por excelencia. De hecho, la forma más elemental de escribir un programa es como una secuencia de asignaciones.

Para este tipo de lenguajes imperativos existen, en principio, 2 formas de organizar el flujo de ejecución de los enunciados o sentencias.

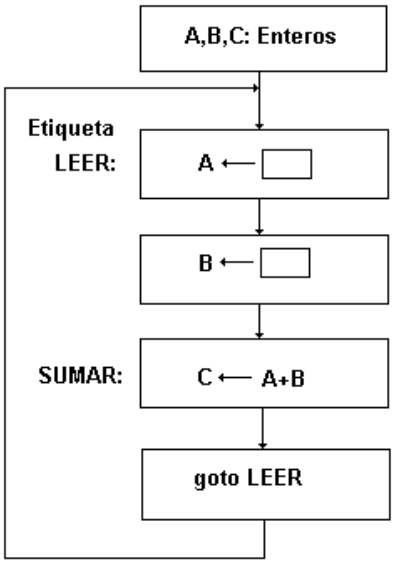


Flujo no estructurado (Control Explícito de Secuencia)

Este tipo de control de flujo, muy utilizado por los primeros lenguajes (Fortran, Algol), supone que los enunciados se ejecutan en secuencia, y que el programador controla explícitamente las bifurcaciones o saltos en el orden de ejecución a través de enunciados goto (ir a) y etiquetas de enunciado.

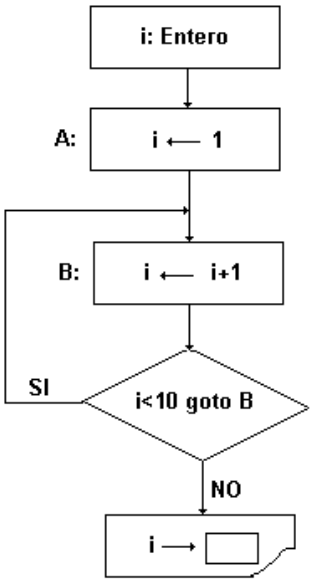
Ejemplo

Los dos tipos básicos de sentencias goto que implementaban los primeros lenguajes eran:



goto Incondicional

El enunciado goto transfiere el control al enunciado etiquetado



goto Condicional

El enunciado goto transfiere el control al enunciado B: solo si se cumple la condicion.

Observar que este ejemplo es una forma de implementar el variar desde i=1 hast i=10.

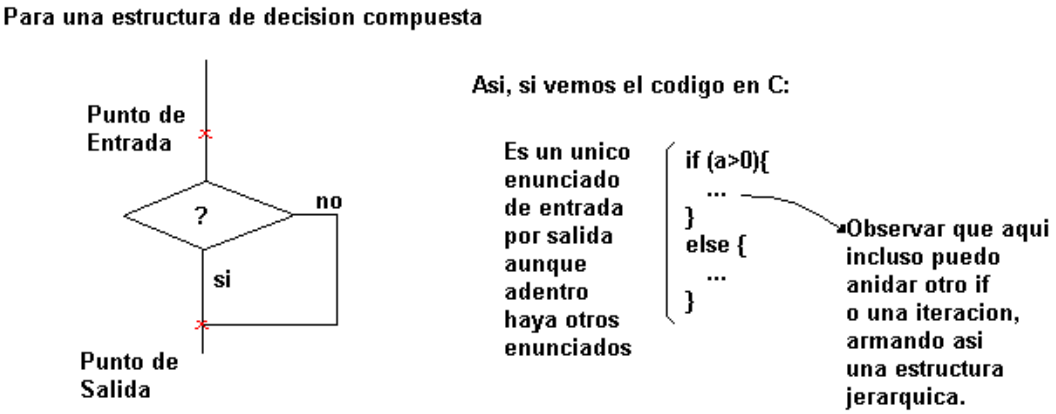
Flujo estructurado

Con la evolución de los lenguajes, en la década del ’70 y ’80, aparecieron las “estructuras” de control de flujo (o secuencia) tales como estructuras de decisión (o alternancia), de iteración o el concepto de subrutinas y corrutinas; y excepciones.

El concepto o idea fundamental de la programación estructurada supone que al construir programas el programador se ocupa de combinar apropiadamente estructuras de decisión, de iteración o llamadas a subrutinas (composición).

Un concepto importante a tener en cuenta es que estas estructuras constituyen **enunciados básicos de entrada por salida**. Es decir, enunciados que tienen un único punto de entrada y un único punto de salida.

Ejemplo



Así, si uno de estos enunciados se coloca en serie con algunos otros enunciados, entonces el flujo de ejecución avanzará necesariamente desde el enunciado precedente al interior del enunciado de entrada por salida, a través del mismo y saliendo hacia el enunciado siguiente.

De esta forma, al leer un programa construído exclusivamente con enunciados de entrada por salida organizados **jerárquicamente** (estructuras anidables), el flujo de ejecución del programa deberá coincidir con el orden de los enunciados en el texto del programa.

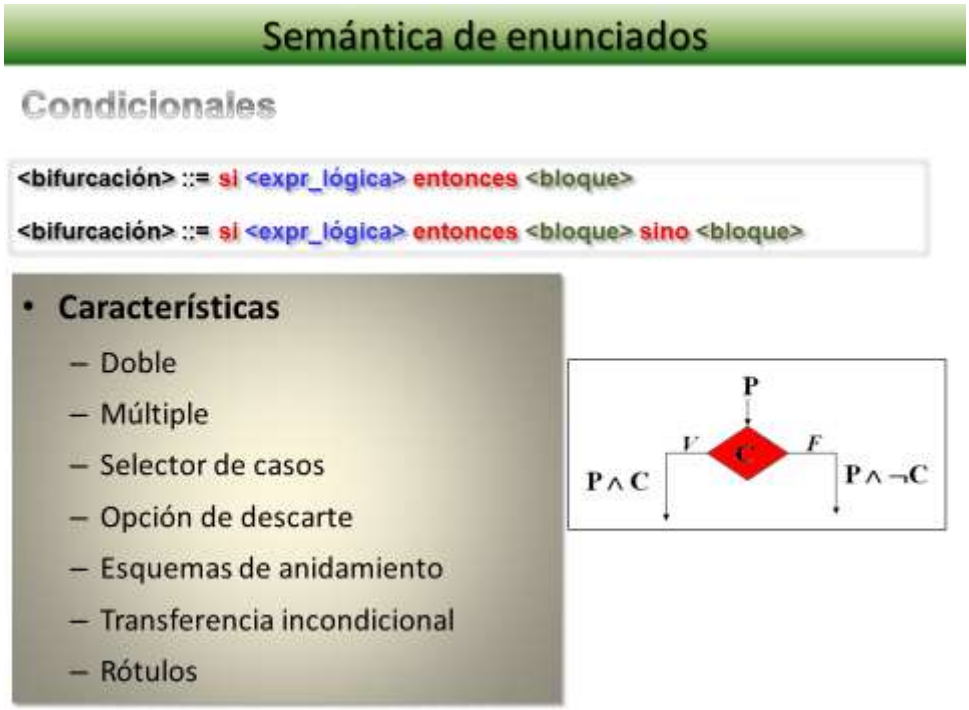
Cada enunciado de control de entrada por salida puede incluir bifurcaciones e iteraciones internas, pero el control sólo sale del enunciado a través de su único punto de salida. (En este caso, estamos frente a estructuras de control de **secuencia implícitas** o por omisión, es decir que los “enunciados básicos” se ejecutan en secuencia).

Flujo estructurado vs. Flujo no estructurado

El uso del *goto* es atractivo en programas muy sencillos, con pocas líneas, sin embargo en programas extensos se arma un laberinto de saltos (o bifurcaciones) que hace que el program se vuelva ilegible y que sea muy difícil que otros programadorres entiendan su lógica; ya que hay carencia de una estructura jerárquica en el programa que permita analizar bloques de código como unidades que resuelven una parte del problema y que se integran en esta estructura jerárquica para resolver todo. Esto último es la propuesta conceptual de la **Programación Estructurada**: combinar apropiadamente grupos de enunciados básicos de entrada por salida (estructuras de decisión, iteración, subrutinas, corrutinas y excepciones); donde cada uno sirve para un solo propósito (resuelve una parte del problema) dentro de la estructura global del programa.

Veamos ahora en detalle las Estructuras de Control de Flujo.

Bifurcaciones

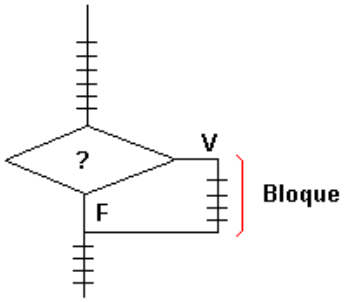


Alternan el flujo de ejecución decidiendo si se ejecuta **alternativamente uno u otro** bloque de enunciados en función de una condición.

1. Bifurcación Simple

Expresa la ejecución condicional de un enunciado.

if (<condicion>) then {<enunciado>}



Recordar que desde el punto de vista semantico esto se ve como una unica sentencia o enunciado basico.

Concepto de Bloque de Enunciados

Es muy importante que el programador maneje adecuadamente el concepto de bloque de enunciados o sentencias y que sepa cómo se maneja en cada lenguaje que utiliza para no cometer errores.

Ejemplo

En C y en Javascript, los delimitadores de bloque son las {}

```
if (<condicion>) {
   ...
   ...                          (bloque de sentencias)
   ...
}
```

Sin embargo, no es un delimitador de bloque específico. Esto tiene el problema de que hay distintas formas de definir un bloque y, si no se interpreta adecuadamente, puede dar lugar a errores. Vamos algunos casos:

```
if (a>b);  // No hay error sintáctico, pero no hace nada.
```

```
if(a>b) a=1;b=2;        /* esto es semánticamente equivalente a if(a>b){a=1;}, b se
                           asigna siempre en 2, se cumpla o no la condición.  */
```

```
if (++a); //a se incrementa en uno.
```

Observar que tengo más posibilidades pero con más posibilidad de error. Todas son sintácticamente válidas.

Esta es una característica que define a C como un lenguaje ORTOGONAL.

Nota: ALGOL también es un lenguaje ortogonal. En ADA, COBOL y FORTRAN no existe un delimitador genérico de bloque, porque esto está definido en cada una de las estructuras:

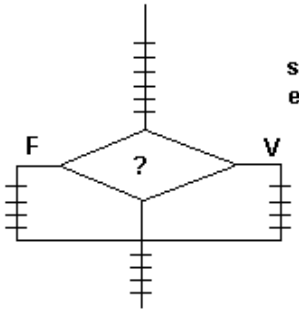
Ejemplo

```
IF a IS GREATER THAN b THEN {
   ...
}
[ELSE {statement}]   END-IF
```

2. Bifurcación Doble

Expresa la alternancia entre la ejecución de un bloque u otro, pero no ambos, ante la evaluación de una condición.

if (<condicion>) then {<bloque1>} else {<bloque2>}



semanticamente hablando esto tambien es una unica sentencia.

Ejemplo

También vinculado a la definición de bloque

Esto es una única sentencia que se ejecuta si se cumple la primera condición

```
if (a>b)
  if (c>=d){
    a=1;
    b=2;
  }
else
  d=1;
```

3. Bifurcación Múltiple

No todos los lenguajes la proveen y no todos la proveen de la misma forma. Una forma de lograr bifurcaciones múltiples es anidar bifurcaciones dobles.

```
if (a>b) {
   ...
   ...
   ...
}
else {
     if (a!=c){
           ...
           ...
     }
     else {
           if (c==d){... }
      }
}
```

Otra forma de selección múltiple es el **selector de casos** que sería el equivalente a la selección doble anidada de la izquierda.

```
if (a+b==1 || a+b==0) {
   ...
   ...
}
else if (a+b==2){
   ...
   ...
}
else if (a+b==3){
  ...
  ...
}
else {
   ...
}
```

```
switch (a+b) {
   case 0: case 1:
                    ...
                     ...
                    break;
     case 2:
                    ...
                     ...
                    break;

     case 3:
                    ...
                     ...
                    break;
     default:

}
```

El **rótulo** (valor de cada caso) tiene que ser un valor integral. Es decir entero de cualquier tipo o carácter (no puede ser punto flotante). Las últimas versiones de algunos lenguajes han incorporado la posibilidad de aceptar cadenas.

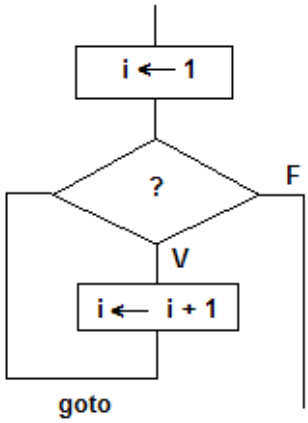
Si no uso el break, lo que estoy haciendo es implementar el concepto de rótulos múltiples.

La opción de descarte *default*, es opcional y de usarse debe colocarse al final para que sólo entre en ella cuando no se satisfaga ninguno de los casos previos.

Iteraciones

Los primeros lenguajes implementaban las iteraciones con sentencias de control explícito ***goto*** (ir a una línea o párrafo específico en el código)

Ejemplo



Luego, aparecieron las estructuras iterativas al estilo de las que se han estudiado en algoritmos.

Con cantidad indeterminada de ciclos

1. Pre-test / Evaluación Inicial / Mientras

mientras condición hacer bloque

Ejemplo

```
while (a>b) {
   …  //bloque de sentencias
}
```

2. Post-test / Evaluación Final / Repetir-Hasta

Repetir bloque hasta condición

Ejemplo

```
do {
   …  //bloque de sentencias          es un repetir-mientras
} while (a>b);
```

3. Midle-test

Evalúan la condición en el medio del bloque repetitivo.

Ejemplo

```
while(1) {

   …
   …          (n+1) veces

   …


   if (a>b) break;

   …
   …           (n) veces

   …
}
```

Otra alternativa sería:

```
while(a>b) {

   …
   …          (n+k) veces

   …


   if (a==++b) continue;

   …
   …           (n) veces

   …
}
```

Una tercer alternativa sería usando goto y rótulos de párrafo.

```
:acm   //rótulo

while(a>b) {

   …
   …          (n+k) veces

   …


   if (a!=c) goto acm;   //va al rótulo.

   …
   …           (n) veces

   …
}
```

Importante: Observar, que en realidad estos ejemplos de estructuras de midle-test no están implementadas como tal en C/C++. En el código provisto como ejemplo se ha utilizado ***transferencia de control explícito*** con sentencias del tipo goto. Su uso, obviamente no es recomendado ya que des-estructura el programa.

Con cantidad determinada de ciclos

1. Enumerativa / Variar

En C/C++ no podemos decir que se implemente exactamente el Variar. La estructura iterativa que nos permite repetir una cantidad determinada de veces un bloque de enunciados es el ***for***, cuya estructura sintaxis general es:

```
for (<inicialización>; <condición>;<incremento>) {
        <bloque>
}
```

y su comportamiento o semántica sería:

inicialización

condición

bloque

incremento

condición

bloque

incremento

condición

bloque

incremento

El problema con C es que tiene muchísimas variantes para el ***for*** y esto da lugar a errores.

Ejemplos

```
for (i=1; i<10; i++) {…}  //observe que sucede si se post-incrementa

for (i=1; i<10;++i) {…} // y qué si se pre-incrementa. Hay diferencia? Justificar.

for (i=1; i==10; i++) {...}  //no se ejecuta nunca. Por qué?

for (i=1; 10; i++) {...}  //se ejecuta indefinidamente, la condición es siempre true.

for (;i<=10;i++){…} //omitir una parte de la estructura es válido.

for(i=0, k=1; (c=getche())!=13; i+=k, k++); /* cdo. sale la variable i tiene la  acumulación
                                                de k hasta que se presione enter.*/
```