Asignatura: Entornos de programación
Entornos orientados a estructura

Características y ejemplos

1. Entornos basados en editores de estructura

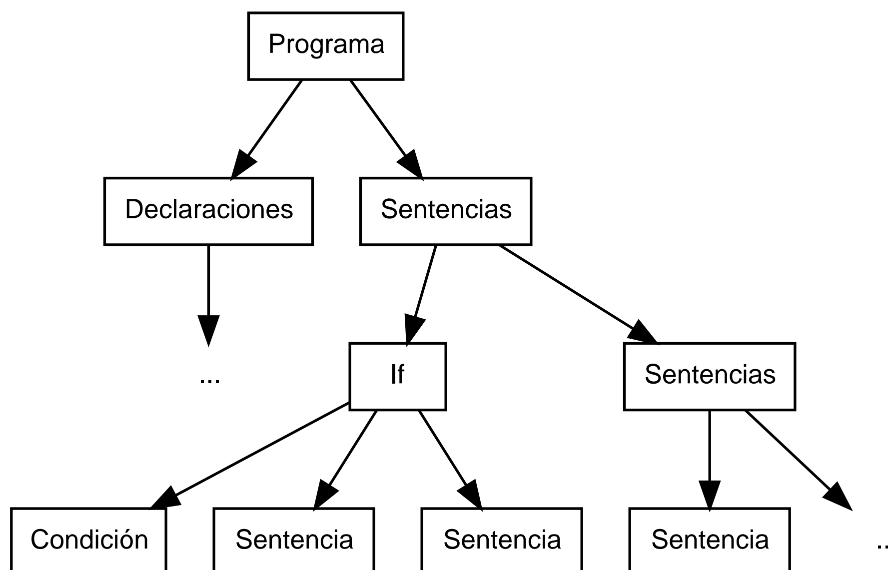
La idea de que un programa no es equivalente al texto de su código fuente, sino que lo esencial es la estructura lógica del cómputo que describe, ha llevado a la creación de los llamados *entornos de programación orientados a estructura*, en los que se manipula directamente la estructura lógica del código y no su representación como texto. La edición de la estructura se hace sobre elementos sintácticos tales como expresiones, sentencias o funciones y no sobre elementos textuales tales como caracteres, palabras o líneas.

Incluso hay casos en los que el código del programa no se representa como texto sino en forma gráfica, como ocurre en los llamados *lenguajes visuales*.

1.1 Editores de estructura

Los editores de estructura de código, llamados también editores sintácticos o más frecuentemente editores dirigidos por sintaxis (*sintax-directed editors*), permiten editar el código fuente manipulando directamente una representación interna de su estructura.

La representación habitual de la estructura del código es la de su árbol de sintaxis abstracta (AST). Ejemplo:



1.2 Características de un entorno orientado a estructura

En general, los entornos orientados a estructura suelen ser específicos para un lenguaje de programación. Por lo tanto son un caso particular de *entornos centrados en un lenguaje* y comparten sus características generales, pero están concebidos de manera diferente:

- Normalmente soportan un único lenguaje de programación.

- El editor de código fuente no es un editor de texto, sino un editor de estructura (editor sintáctico)
- Se basan en representar internamente el código fuente como una estructura:
    - Árbol de sintaxis abstracta: AST
- La presentación externa del código es en forma de texto, mediante:
    - Plantillas (para elementos sintácticos no terminales)
    - Texto simple (para elementos terminales - a veces para expresiones)
- Garantizan que el código es sintácticamente correcto.
- La compilación se realiza de manera incremental, a medida que se edita el código.
- Permite la ejecución inmediata del código editado, incluso aunque esté incompleto.
- Soportan el desarrollo de software a nivel individual, pero no el desarrollo en equipo a gran escala.

1.3 Historia

La historia de estos entornos es relativamente reducida en el tiempo.

- La mayoría se desarrollaron a finales del años 70 y a lo largo de los 80
- Los desarrollos fueron fundamentalmente académicos
- No se han difundido en la práctica profesional. Han quedaron en desuso
- Los entornos convencionales modernos proporcionan muchas de sus funciones (aunque no usen editores de estructura)

En la actualidad los lenguajes de marcado (tal como XML) pueden ser una buena forma de representar la estructura del código fuente con vistas a su manipulación. Existen editores y procesadores XML que podrían ser la base de nuevos entornos de programación orientados a estructura.

1.4 Ejemplos

Algunos ejemplos de entornos orientados a estructura son:

- El **Cornell Program Synthesizer (CPS)**. Es quizá el ejemplo más conocido y el más referenciado en la literatura. Ha servido de ejemplo para desarrollar otros. Permite programar en un subconjunto del lenguaje PL/I denominado PL/CS.
- **Mentor** es un entorno de programación en Pascal.
- **Gandalf** es un conjunto de varios subproyectos. Su objetivo principal fue crear un entorno completo de desarrollo de software, y no sólo un entorno de programación.
- **Alice Pascal** es otro entorno de programación en lenguaje Pascal compatible con TurboPascal. Sigue las ideas del CPS.
- **SDS** es un entorno de programación en Modula-2. Es un producto comercial desarrollado por Interface Technologies. Ha desaparecido.

Como complemento se han llegado a desarrollar también generadores de entornos similares a los generadores de compiladores. En particular existe el **Synthesizer Generator**, capaz de generar entornos similares al sintetizador de Cornell para otros lenguajes de

programación a partir de una descripción de la sintaxis y semántica del lenguaje mediante una gramática de atributos.

1.5 Lenguajes y entornos visuales

Este es un caso especial de entornos orientados a estructura. La representación externa del código fuente no es en forma de texto, sino gráfica. El editor permite ir construyendo el grafo que representa la estructura del código. El programa construido de esta manera se ejecuta directamente mediante un intérprete, o bien se exporta como texto en un lenguaje formal para ser compilado o interpretado externamente. Algunos ejemplos de este tipo de entornos son:

- **Prograph**
- **Projector** (parte del meta-CASE DOME)
- **VFPE**

Tanto Prograph como Projector son lenguajes de flujo de datos. Un programa basado en flujo de datos se representa como un grafo en el que los nodos son operadores y los arcos son flujos de datos que conectan la salida de ciertos operadores con las entradas de otros. Una operación se ejecuta cuando hay datos presentes en todas las entradas requeridas. En ese momento se producen resultados que se transmiten por los arcos de salida, pudiendo entonces ejecutarse otras operaciones.

VFPE es un editor gráfico de la estructura (árbol sintáctico) de un programa funcional. El programa editado puede ejecutarse directamente o exportarse como código Haskell.

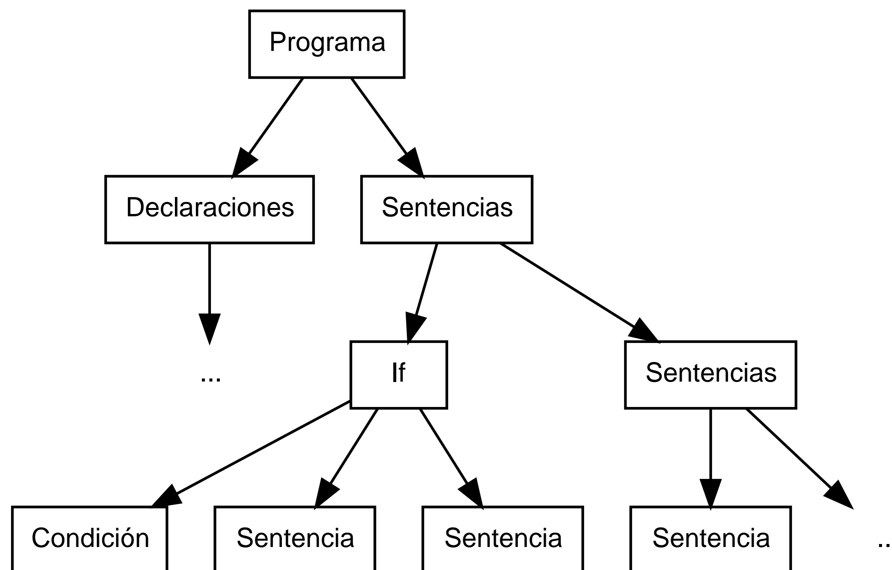
2. El sintetizador de programas de Cornell

Es el ejemplo más conocido y el más referenciado en la literatura. Sus características principales son:

- Orientado a la enseñanza de la programación
    - Desarrollo por refinamiento progresivo
- Para el lenguaje PL/I (subconjunto PL/CS)
- Entorno orientado a estructura
    - Editor de estructura, plantillas
- Ejecución directa
    - Compilación incremental
    - Ejecución interpretada
    - Ejecuta programas incompletos
    - Facilidades de traza y depuración

2.1 Edición de la estructura

- Representación interna: AST
    - (Abstract Syntax Tree)



2.2 Presentación como texto

- Plantillas (elementos no terminales)

```
IF (condición)
   THEN sentencia
   ELSE sentencia

PUT SKIP LIST (lista-de-expresiones)
```

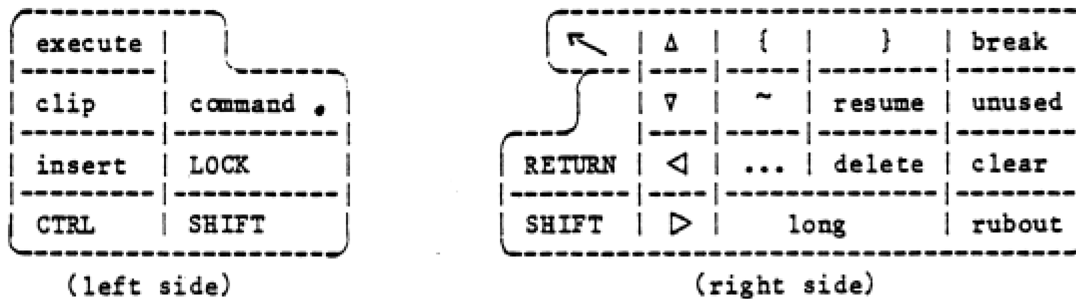
- Frases (elementos terminales = texto)

```
k > 0

'no positivo'
```

2.3 Microcomputador TERAK

- Pantalla de texto
- Teclado con teclas especiales



2.4 Movimiento del cursor

- Señalar inicio de plantilla o frase (en preorden)
  - Cursores ↑ ↓ (y también ← →)

```
IF (k > 0)
   THEN sentencia
   ELSE PUT SKIP LIST ('no positivo');
```

- Señalar inicio de plantilla o cada carácter de una frase
  - Cursores ← →

```
IF (k > 0)
   THEN sentencia
   ELSE PUT SKIP LIST ('no positivo')
```

- Avances rápidos
  - Prefijo **long**, movimiento **diagonal** (ir al padre)

2.5 Insertar elementos (frases)

```
IF (condición)
   THEN sentencia
   ELSE sentencia

 → k
```

```
IF (k )
   THEN sentencia
   ELSE sentencia

 > 0

IF (k > 0 )
   THEN sentencia
   ELSE sentencia
```

2.6 Insertar elementos (plantillas)

```
IF (k > 0)
   THEN sentencia
   ELSE sentencia

 ↓↓↓

IF (k > 0)
   THEN sentencia
   ELSE sentencia

 .p  return

IF (k > 0)
   THEN sentencia
   ELSE PUT SKIP LIST (lista-de-exp);
```

2.7 Errores sintácticos (en frases)

- Compilación incremental

```
IF (k > 0)
   THEN sentencia
   ELSE PUT SKIP LIST (lista-de-exp);

 no positivo' ↑

⇒ ERROR: comillas desajustadas

IF (k > 0)
   THEN sentencia
   ELSE PUT SKIP LIST (no positivo');

 ' ↑

IF (k > 0)
```

```
   THEN sentencia
   ELSE PUT SKIP LIST ('no positivo');
```

2.8 Transformación de la estructura

- Problema: transformar

```
IF (k > 0)
   THEN PUT SKIP LIST ('positivo');
   ELSE PUT SKIP LIST ('no positivo');
```

en

```
DO WHILE (condición);
   IF (k > 0)
      THEN PUT SKIP LIST ('positivo');
      ELSE PUT SKIP LIST ('no positivo');
END;
```

2.9 Transformar mediante cortar y pegar

```
IF (k > 0)
  THEN PUT SKIP LIST ('positivo');
  ELSE PUT SKIP LIST ('no positivo');

 clip

sentencia

 .dw  return

DO WHILE (condición);
  {sentencia}
END;

 ↓  insert

DO WHILE (condición);
  IF (k > 0)
    THEN PUT SKIP LIST ('positivo');
    ELSE PUT SKIP LIST ('no positivo');
END;
```

2.10 Errores semánticos

- Compilación incremental

```
DECLARE ( temp ) FIXED;
DECLARE ( m, n ) FLOAT;
temp = m;
m = n;
```

```
n = temp;

 borrar declaración de temp

DECLARE ( m, n ) FLOAT;
temp = m;
m = n;
n = temp;

 reinsertar temp en la otra declaración

DECLARE ( m, n, temp ) FLOAT;
temp = m;
m = n;
n = temp;
```

2.11 Edición de listas

- Iteración sintáctica

```
{elemento} → 0 o más elementos
```

- Movimiento de cursor con **return**

```
x = 0;
y = 0;
               return
x = 0;
{sentencia}
y = 0;
               return
x = 0;
y = 0;
               return
x = 0;
y = 0;
{sentencia}
```

2.12 Plantilla de comentario

- Refinamiento progresivo

Plantilla:

```
/* comentario */
   {sentencia}
```

Equivale a una sentencia. Ejemplo:

```
/* Intercambiar m y n */
   temp = m;
   m = n;
```

```
   n = temp;
PUT SKIP LIST (m, n);

 ellipsis (folding)

/* Intercambiar m y n */
   ...
PUT SKIP LIST (m, n);
```

2.13 Ejecución de programas

- Ejecución inmediata (programa ya compilado)
- Ejecución rápida, temporizada o paso a paso (*debug*)
  - elemento = plantilla o frase
  - elipsis = elemento único
- **resume** un paso
- **long resume** todo el elemento
- **return** completar lista
- **diagonal** completar plantilla
- **long diagonal** completar PROCEDURE

Seguimiento sobre el texto del programa

2.14 Irregularidad texto / estructura (I)

- Elementos distintos con presentación común
  - Comparación (frase) `x = 0`
  - Asignación (plantilla) `x = 0`
  - Declaración de argumentos y variables locales

```
nombre: PROCEDURE ( j );
   DECLARE ( j ) FIXED;
   DECLARE ( k ) FIXED;
   {sentencia}
END nombre;
```

2.15 Irregularidad texto / estructura (II)

- AST inconsistente con la presentación. Situación inicial

```
IF (condición)
   THEN sentencia
   ELSE PUT LIST ('¿Qué ELSE soy?');
```

- Tras insertar IF-THEN

```
IF (condición)
   THEN IF (condición)
```

```
          THEN sentencia
   ELSE PUT LIST ('¿Qué ELSE soy?');
```

2.16 Implementación. Representación de un nodo

- Código de un nodo

TIPO	... código ...	continuación

- Ejemplo (if-then-else)

```
IF (condición)
   THEN sentencia
   ELSE sentencia
```

- Ejemplo (código interno)

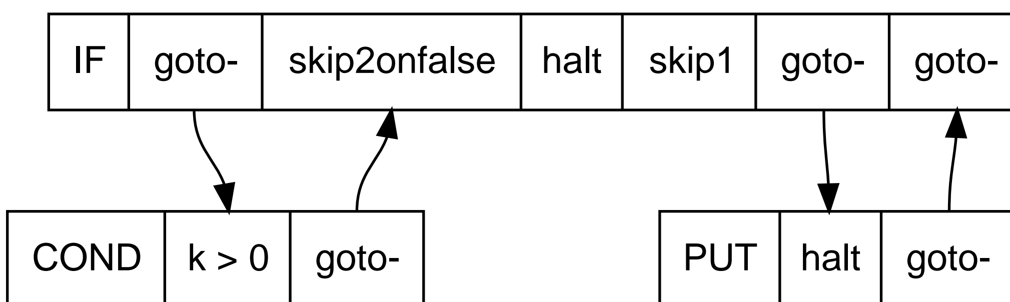
IF	halt	skip2onfalse	halt	skip1	halt	goto-

2.17 Implementación. Representación de un árbol

- Enlaces entre nodos. Ejemplo (if-then-else)

```
IF (k > 0)
   THEN sentencia
   ELSE PUT LIST (lista-de-expr);
```

- Código interno



2.18 Implementación. Presentación como texto

- Codificación de las plantillas de presentación como texto

```
"IF"
condición
") \{\nTHEN"
sentencia 1
"ELSE"
```

```
sentencia 2
“\}\r”
```

- `\{` aumentar margen
- `\n` salto de línea, mismo margen
- `\}` disminuir margen
- `\r` retroceder al margen izquierdo

3. Discusión

- Los entornos orientados a estructura presentan innegables ventajas respecto a los entornos basados en la edición del texto fuente. A continuación se enumeran las más importantes.
- Sin embargo estos entornos, tal como se concibieron inicialmente, no han llegado a utilizarse en la práctica habitual de desarrollo de software, ya que presentaban claros inconvenientes
- Muchos de los productos citados como ejemplo han desaparecido. Su interés es fundamentalmente histórico.
- No obstante, la idea de trabajar directamente sobre la estructura del código sigue resultando atractiva, y parece posible aplicarla de nuevo siendo conscientes de sus dificultades y aprovechando los avances tecnológicos actuales.

3.1 Ventajas

- Evitan los errores sintácticos
- Evitan tener que escribir los elementos fijos del código: Palabras clave, puntuación, etc.
- Presentan el código con un estilo uniforme, bien encolumnado (formato automático)
- Guían al programador indicando qué elementos pueden insertarse en cada punto y recordándole la sintaxis de cada sentencia estructurada.
- Facilita el tratamiento incremental
  - compilación al editar
  - ejecución inmediata (incluso de programas incompletos o con errores semánticos)
- Facilita la construcción mediante refinamientos
- Facilitan la reorganización del código al permitir la selección directa de secciones de código: funciones, bucles, etc.
- Facilita la refactorización del código mediante transformaciones programadas
- Permiten trabajar con estructuras lógicas no contempladas directamente en el lenguaje de programación.

3.2 Inconvenientes

- Es difícil editar la estructura a nivel de grano fino (expresiones: operadores, operandos, ...)
- Es difícil realizar algunas operaciones de edición que resultan triviales sobre el texto
  - Insertar/eliminar elementos intermedios anidados
  - Insertar/eliminar elementos en listas
  - Inhibir (comentar) fragmentos del programa
  - Transformar una plantilla en otra
- Dificulta el uso generalizado de comentarios

- Se pueden construir programas incomprensibles
    - Algún AST no corresponde al texto del programa
- Exigen un cambio en la mentalidad del programador.
- Los productos citados como ejemplo no permitían el trabajo en equipo. En muchos casos sólo trabajaban con programas monolíticos.