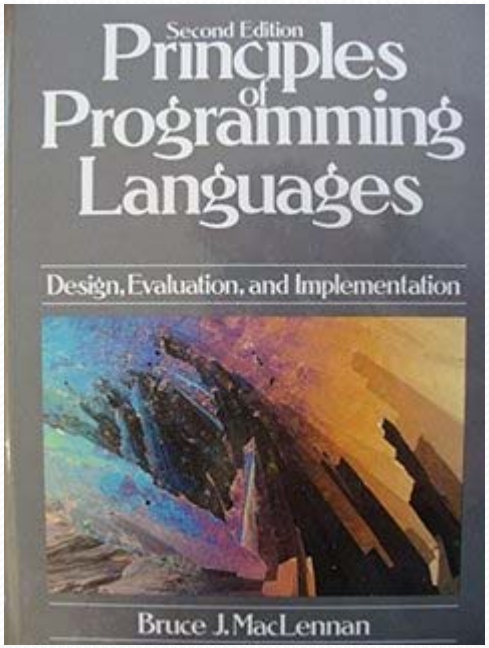
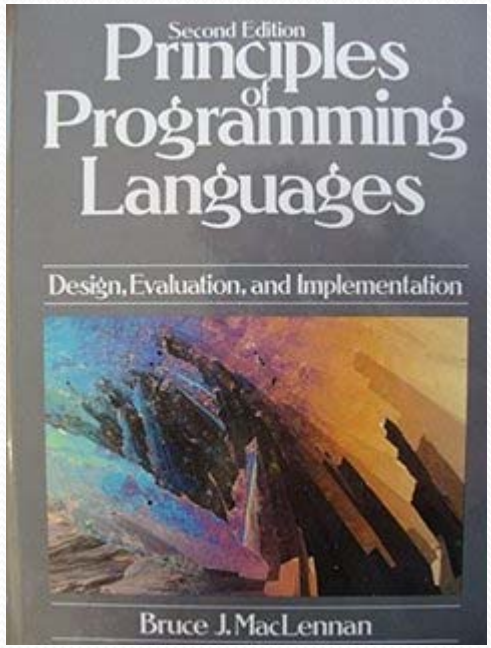
Lenguajes de Programación

Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx



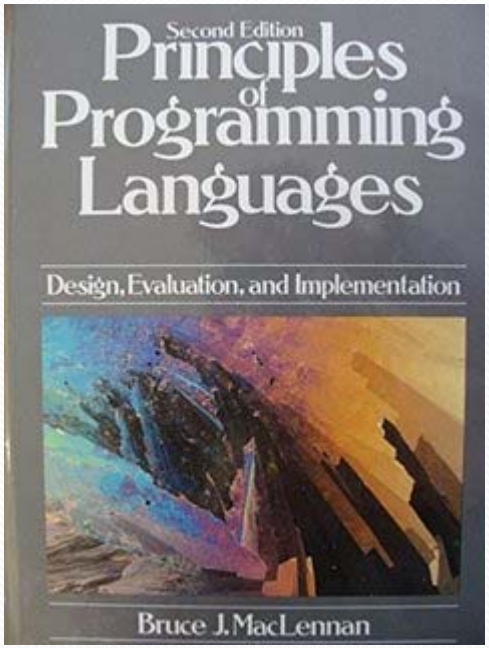
- Comenzaremos por analizar 3 principios más (de los sugeridos por MacLennan) que se cubrieron en el tema anterior (intérpretes de pseudo-código), pero que no alcanzamos a enunciar.

Principio de Seguridad



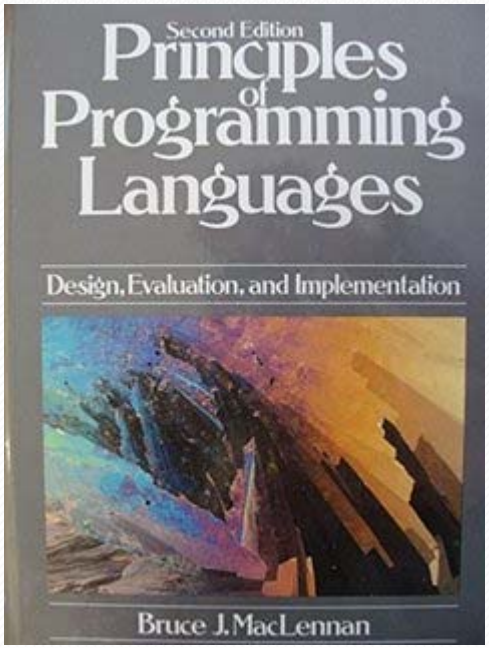
Ningún programa que viole la definición del lenguaje, o su propia estructura original, debe escapar detección.

Principio de Regularidad



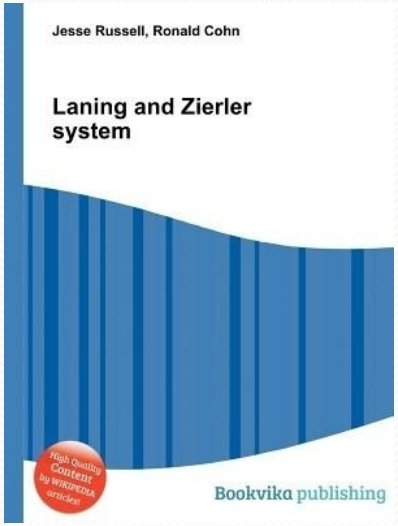
Evite requerir que la misma cosa se tenga que definir más de una vez; factorice el patrón recurrente.

Principio de Etiquetado



No requiera que el usuario se tenga que saber la posición absoluta de un elemento en una lista. En vez de eso, asocie etiquetas con cualquier posición que debe ser referenciada más adelante.

Historia de FORTRAN



- Estrictamente hablando, FORTRAN no fue el primer lenguaje de programación que existió (Laning & Zierler, del MIT, ya tenían un compilador de un lenguaje algebraico en 1952), pero sí fue el primero en atraer la atención de una parte importante de la reducida comunidad de usuarios de computadoras de la época.

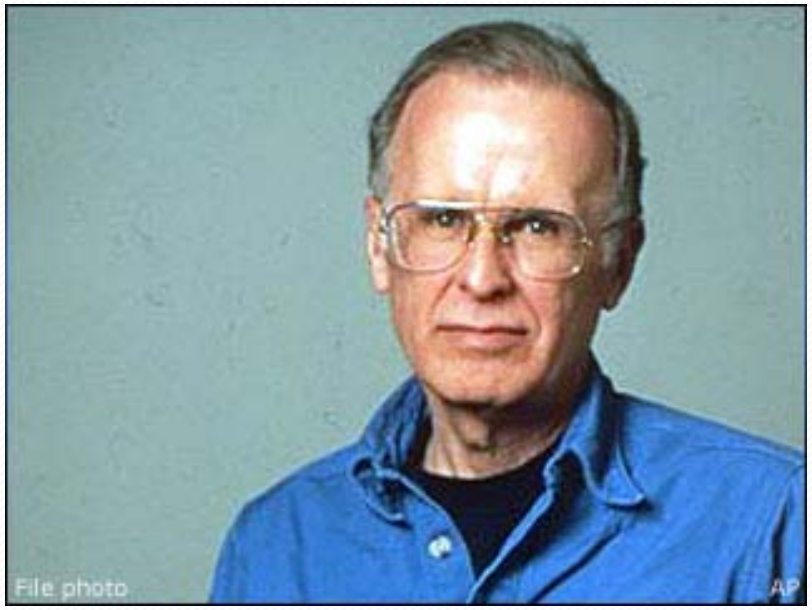
Historia de FORTRAN

```
C FORTRAN IV program to
C find sum of integers 1-100
      INTEGER I, SUM
      SUM=0
      DO 1 I=1, 100
    1 SUM = SUM + I
      WRITE (6,2) SUM
    2 FORMAT(1X,I5)
      STOP
      END
```

FIGURE 114. FORTRAN IV program

- El desarrollo de FORTRAN comenzó en 1955 y el lenguaje se liberó finalmente en abril de 1957, después de 18 años-hombre de trabajo.

Historia de FORTRAN



- FORTRAN (*FORmula TRANslating system*) fue desarrollado principalmente por John Backus en IBM. Backus recibió el *Turing Award* en 1977 y falleció en 2007.

Historia de FORTRAN

- El principal énfasis de FORTRAN fue la eficiencia. Su diseño se basó en un intérprete llamado "*Speedcoding*", que fue desarrollado por Backus para la IBM 701.

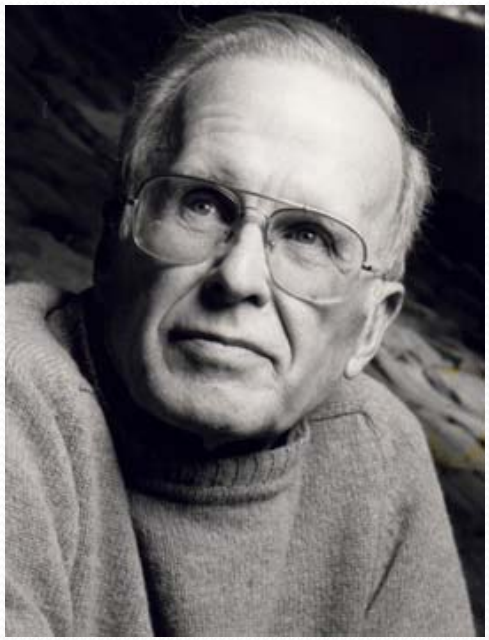
Historia de FORTRAN

- FORTRAN fue desarrollado inicialmente en una IBM 704, aprovechando sus rutinas de punto flotante (proporcionadas en hardware).

Historia de FORTRAN

- La especificación preliminar de FORTRAN fue recibida con frialdad y escepticismo. Pocos creían que pudiera diseñarse un compilador que fuese equiparable (en eficiencia) a un humano.

Historia de FORTRAN



- John Backus fue fuertemente criticado incluso por luminarias de la época como John von Neumann, quienes creían que el desarrollo de FORTRAN sólo era un desperdicio de dinero (el proyecto estuvo a punto de ser cancelado).

Historia de FORTRAN

- El secreto del éxito de FORTRAN se atribuye a su excelente documentación y a las sofisticadas técnicas de optimización que se usaron en el diseño de su compilador. De acuerdo a John Backus, las técnicas utilizadas en FORTRAN no fueron superadas sino hasta bien entrados los 1960s.

Cronología de FORTRAN

- En 1958 se liberó FORTRAN II.
- Otro dialecto, llamado FORTRAN III, fue liberado también en 1958, pero no resultó muy exitoso debido a su enorme dependencia de la IBM 704.
- En 1962 se liberó FORTRAN IV, que ha sido quizás la versión más popular del compilador.

Cronología de FORTRAN

- En 1958, el *American National Standards Institute* (ANSI) estandarizó FORTRAN (a esta versión se le llama ANS FORTRAN).
- Un nuevo estándar fue liberado en 1977 (se le conoce como FORTRAN 77).
- La versión más reciente de FORTRAN (FORTRAN 2008), se liberó en 2010 y difiere mucho del lenguaje original.

Estructura de los Programas en FORTRAN

- Los programas en FORTRAN se dividen en 2 partes:
  - 1) Una parte **declarativa**, la cual describe las áreas de datos, sus longitudes (tipos) y sus valores iniciales. Las instrucciones declarativas existen en la mayor parte de los lenguajes estructurados, aunque se les llama de maneras distintas. En FORTRAN se les denomina “sentencias no ejecutables”.
  - 2) Una parte **imperativa** que contiene los comandos a ser ejecutados en el momento en que corre un programa. A las instrucciones imperativas de FORTRAN se les denomina “sentencias ejecutables”.

Parte Declarativa de un Programa

- La parte declarativa del programa realiza 3 funciones principales:

  - 1) Asignar un área de memoria de un tamaño especificado. Los tipos de datos disponibles en FORTRAN son: **INTEGER**, **REAL**, **DOUBLE PRECISION** y arreglos (definidos con **DIMENSION**). El tipo de las variables numéricas suele considerarse **REAL** por omisión. Versiones modernas de FORTRAN cuentan también con los tipos **COMPLEX**, **LOGICAL** y **CHARACTER**.

Parte Declarativa de un Programa

- 2) Asociar un nombre simbólico a un área específica de memoria (a esto se le llama “binding” o declaración de variables).

- 3) Inicializar el contenido de esa área de memoria. En el caso de FORTRAN, las variables no tienen que ser inicializadas. Esto suele ser una fuente común de errores en FORTRAN.

Parte Declarativa de un Programa

- Ejemplo de una declaración:

  DIMENSION MIARREGLO(100)

- Esto inicializa el arreglo MIARREGLO a 100 posiciones de memoria (se presuponen números reales por omisión).

Parte Declarativa de un Programa

- Ejemplo de inicialización:

  DATA MIARREGLO, MIVAR/100*0.0,5.0

- Esto inicializa las 100 posiciones del arreglo MIARREGLO a 0.0. Adicionalmente, inicializamos la variable MIVAR a 5.0.

Parte Imperativa de un Programa

- Las sentencias imperativas son de 3 tipos diferentes:
  - 1) **Sentencias computacionales**: Son análogas a las operaciones aritméticas y de movimiento de datos del pseudo-código que vimos anteriormente. La asignación es la sentencia computacional más importante de FORTRAN.

Parte Imperativa de un Programa

- 2) **Sentencias de control de flujo**: Comparaciones y ciclos. FORTRAN proporciona instrucciones condicionales (sentencias **IF**), ciclos (usando **DO**) y saltos incondicionales (usando **GOTO**). El lenguaje tiene sentencias muy primitivas para flujo de datos.

- 3) **Sentencias de entrada/salida:** FORTRAN I tenía 12 sentencias de entrada/salida de un total de 26 del lenguaje. La razón era la enorme variedad de dispositivos de E/S que debía ser capaz de manejar el compilador. Las versiones más recientes de FORTRAN han reducido y/o simplificado la necesidad de usar tantas sentencias de E/S.

Parte Imperativa de un Programa

- Ejemplo de asignación:

  RESULTADO = TOTAL/**FLOAT**(N)

- Nótese que en este caso, FLOAT se usa para convertir N (presumiblemente una variable entera) a un valor real, de manera que el resultado de la operación sea un número de punto flotante.

Parte Imperativa de un Programa

- ¿Qué otro lenguaje conoce donde deba hacerse una reducción de tipo similar a ésta?

- Es importante recordar que el uso de notación algebraica convencional fue una de las principales contribuciones de FORTRAN.

Notación Algebraica

- Un ejemplo de notación algebraica en FORTRAN es el siguiente:

```
RESULTADO = A*B+C-9.0/D**3.4
```

Notación Algebraica

Operación	Operador (FORTRAN)
Suma	+
Resta	-
Multiplicación	*
División	/
Exponenciación	**

Funciones Matemáticas

```
 1          K=1
 2    6     IF (K.EQ.11) GO TO 8
 3          READ,I,J
 4          IF (J.GT.I) GO TO 65
 5          GO TO 66
 6    65    WRITE(6,6002)J,I
 7    6002  FORMAT(' ',I3,' IS GREATER THAN ',I3)
 8          K=K+1
 9          GO TO 6
10    66    WRITE(6,6001)I,J
11    6001  FORMAT(' ',I3,' IS GREATER THAN ',I3)
12          K=K+1
13          GO TO 6
14    8     CALL EXIT
15          END
```

- FORTRAN tiene también una cantidad considerablemente grande de rutinas matemáticas disponibles para el cálculo de logaritmos, funciones trigonométricas, números complejos, etc.

Jerarquías de los Operadores

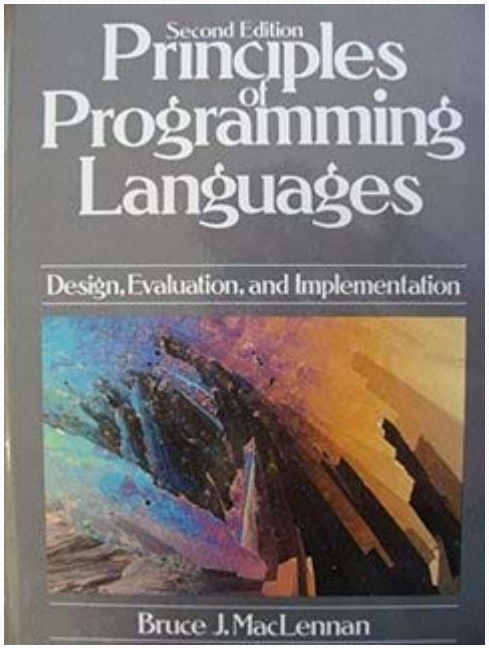
- Los operadores aritméticos también tienen prioridades:
  - 1) Exponenciación
  - 2) Multiplicación y división
  - 3) Suma y resta
- Estas mismas prioridades han sido adoptadas en la mayoría de los lenguajes de programación modernos.

Ciclos con **DO**

- La sentencia **DO** es la única estructura de control de alto nivel que proporciona FORTRAN, puesto que les permite a los programadores expresar lo que quieren que se haga y no cómo quieren que se haga.

- La sentencia **DO** ilustra el principio de Automatización.

Principio de Automatización



Automatiza las actividades mecánicas, tediosas o propensas a errores.

Ciclos con **DO**

- La sentencia **DO** permite realizar iteraciones definidas (llamadas en inglés "*counted loops*"):

```
      DO 100 I=1,N
            haz lo que quieras
100   CONTINUE
```

Ciclos con **DO**

- En el ejemplo anterior, 100 es una etiqueta (sólo se permiten etiquetas numéricas en FORTRAN) que indica lo que está contenido dentro del ciclo. **I** es el contador del ciclo en este caso y toma un valor inicial de 1, llegando hasta **N**. **CONTINUE** no tiene ningún efecto y sólo se usa para indicar la finalización del ciclo (como la sentencia **NEXT** en BASIC).

Ciclos con **DO**

```
c$doacross if ( jmax > 1000) share ( pi2, jmax, imax, a, b, c )
c$local ( i, j, x ) lastlocal ( y )
    do 35 j = 1 , jmax
    do 30 i = 1, imax
    x = a ( i , j ) + b ( i , j )
    y = pi2 * x
    c ( i , j ) = y
35  continue
30  continue
C Example of a directive based parallel Fortran 77 loop.
```

- Los ciclos con **DO** también pueden ser anidados, lo que proporciona mucha flexibilidad.

Ciclos con **DO** (Ejemplo)

```
DO 100 I=1,M
                .
          DO 200 J=1,N
                .
200       CONTINUE
                .
100   CONTINUE
```

Ciclos con **DO**

- FORTRAN tiene, en general, una estructura “lineal” y no “jerárquica”, dado que la mayoría de sus instrucciones no permiten ningún tipo de anidamiento.

- Por lo tanto, la sentencia **DO** resulta ser más bien una excepción que una función más del lenguaje.

Ciclos con **DO**

- Adicionalmente, la seguridad del lenguaje se ve comprometida por el hecho de que FORTRAN permite saltos incondicionales (GOTOs) dentro y fuera de los ciclos DO bajo ciertas circunstancias.

- Esta es una de las razones por las cuales la sentencia GOTO se volvió obsoleta en la segunda generación de lenguajes de programación.

Ciclos con **DO**

- Finalmente, el ciclo DO está altamente optimizado, lo que lo hace altamente útil y práctico desde la perspectiva de la meta principal del FORTRAN: la eficiencia.

- Aunque parezca extraño, suele darse el caso de que las estructuras de más alto nivel (como el **DO** en este caso) resultan más fáciles de optimizar que las de bajo nivel.

Estructura Básica de un Programa en FORTRAN

Declarations

Main program

Subprogram 1

Subprogram 2

.

Subprogram *n*

Subprogramas

- Los subprogramas son realmente una adición posterior al FORTRAN, ya que no aparecieron sino hasta en el FORTRAN II.

- Aunque FORTRAN I proporcionaba bibliotecas con funciones matemáticas y de E/S, no permitía que los programadores definieran sus propios subprogramas.

Subprogramas

```
C Main loop
C ---------
100    CONTINUE
C
C Check for motion - Let's not go there!
C
       IF (LNK:MOTEN) THEN
           STOP 'FLY_DB not allowed with motion !!!'
       ENDIF
C
       CALL FLY_FR
C
C End of loop
C -----------
C      End of loop - suspend till next pass
       CALL X:SUSP(TASK,TIMEOUT,*90)          ! Concept
C
```

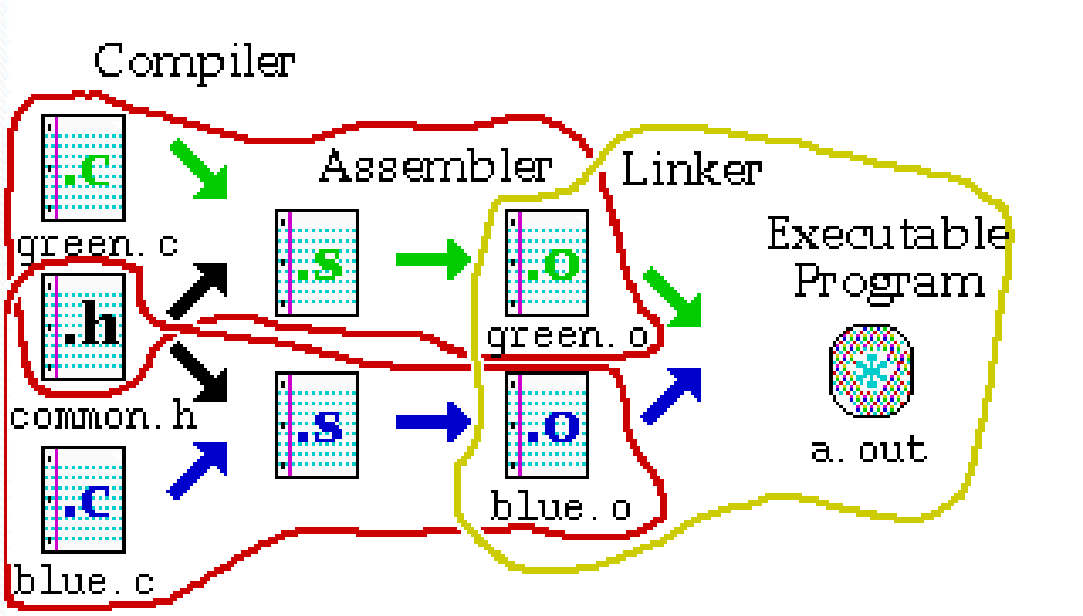
- FORTRAN II resolvió el asunto de los subprogramas al agregar las declaraciones **SUBROUTINE** y **FUNCTION**.

Eficiencia de la Compilación

- Puesto que una de las metas primordiales de FORTRAN durante su diseño fue la eficiencia, el enfoque tomado para generar un archivo ejecutable estaba altamente optimizado:

- 1) **Compilación:** Traducir subprogramas individuales en FORTRAN a código objeto relocalizable (código que usa direcciones relativas en vez de direcciones absolutas de memoria). Estas direcciones relativas se resolverán (o sea, se convertirán a direcciones absolutas) en tiempo de cargado.

Eficiencia de la Compilación



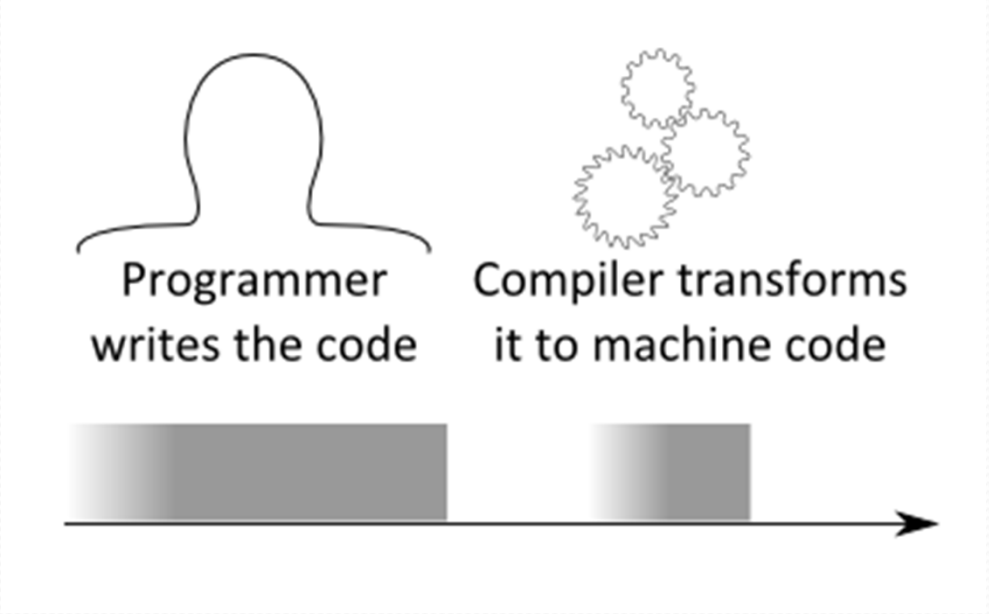
- La mayor parte de los errores de sintaxis se interceptarán en esta etapa (p.ej., sentencias inexistentes, uso erróneo de variables u operadores, etc.)

Eficiencia de la Compilación

- 2) **Ligado (*Linking*):** Incorporar bibliotecas de subprogramas que han sido previamente escritos, depurados y compilados. Los programas en FORTRAN contienen referencias externas que deben resolverse durante esta etapa. Cualquier referencia errónea a funciones externas se resuelve en esta etapa.

- 3) **Cargado:** Es el proceso en el cual se coloca un programa en la memoria de la computadora. Las direcciones relativas (o direcciones relocalizables) se traducen a direcciones absolutas de memoria.

Eficiencia de la Compilación



- 4) **<u>Ejecución</u>:** El control de la computadora se pasa al programa que se encuentra en memoria. Puesto que el programa se ejecuta directamente y no a través de un intérprete, tiene el potencial de correr significativamente más rápido.

Pasos de la Compilación

- La compilación de un programa en FORTRAN se realiza normalmente en 3 pasos:
  - 1) **Análisis sintáctico:** El compilador debe clasificar las sentencias y constructores de FORTRAN y extraer sus partes.

Pasos de la Compilación

- 2) **Optimización:** Los compiladores originales de FORTRAN incluían un optimizador muy sofisticado cuya meta era producir código tan bueno como el de un buen programador humano. Hasta la fecha, la mayor parte de los compiladores de FORTRAN que existen realizan algún tipo de optimización.

- 3) **Síntesis de código:** Juntar las porciones de las instrucciones en código objeto en un formato relocalizable.

Estructuras de Control

- En sus orígenes, FORTRAN fue un lenguaje muy dependiente de la máquina en la que se desarrolló (la IBM 704). Esa es la razón por la que algunas de sus instrucciones lucen un tanto extrañas, redundantes y hasta tontas.

- Algunas de estas sentencias fueron removidas en versiones posteriores del lenguaje, pero la mayor parte de ellas han permanecido ahí durante casi 60 años.

Estructuras de Control

- **<u>La sentencia IF aritmética</u>:**
  - **IF** (e) n1, n2, n3

evalúa la expresión “e” y salta a n1,n2 o n3, dependiendo de si el resultado de la evaluación es <0, =0 o >0.

Estructuras de Control

- La sentencia **IF** aritmética corresponde de manera exacta a una instrucción en lenguaje máquina de la IBM 704 y es un buen ejemplo de cómo NO diseñar una sentencia IF.

- Llevar el control de 3 etiquetas es algo difícil y el hecho de que en muchos casos dos de estas etiquetas pueden fusionarse (por ser idénticas) hacen de ésta una instrucción muy primitiva.

Estructuras de Control

- **<u>La sentencia IF lógica</u>:**

IF (X .**EQ**. 5) M=A+3

.LT., **.GT.**, **.GE.**, **.LE.**, **.AND.**, **.OR.**, **.NOT.**, **.EQV.**, **.NEQV.**

Esta sentencia reemplaza a la primitiva sentencia **IF** aritmética y la vuelve más fácil de usar y más regular.

Estructuras de Control

- **<u>Sentencias Condicionales</u>**
  - **GOTO** era el caballito de batalla del control de flujo en FORTRAN, ya que no existía estructura **IF-THEN-ELSE** como en la mayoría de los lenguajes de programación modernos.

Estructuras de Control

IF (se cumple) **GOTO** 100

. . . caso para condición falsa . . .

GOTO 200

100 . . . caso para condición cierta . . .

200 . . .

Estructuras de Control

- El uso de GOTO puede llevarnos (cuando no se aplica adecuadamente) al desarrollo del denominado "código spaghetti", lo cual genera programas difíciles de leer y modificar.

- **Sentencias para casos:**

  - Como no había CASE como en los lenguajes de programación modernos, se utilizaba en su lugar el denominado "GOTO calculado" (*computed GOTO*).

Estructuras de Control

```
GOTO (10, 20, 30, 40), I
10      . . . manejar caso 1 . . .
        GOTO 100
20      . . . manejar caso 2 . . .
        GOTO 100
30      . . . manejar caso 3 . . .
        GOTO 100
40      . . . manejar caso 4 . . .
100     . . .
```

Aquí se salta a la línea 10, 20, 30 o 40, cuando I tome los valores 1, 2, 3 o 4.

Estructuras de Control

- Ciclos con la decisión al final (*trailing-decision loops*):

  Los ciclos en los cuales la decisión se toma al final, pueden implementarse usando:

  100 . . . Haz lo que quieras . . .
  IF (no se cumple) **GOTO** 100

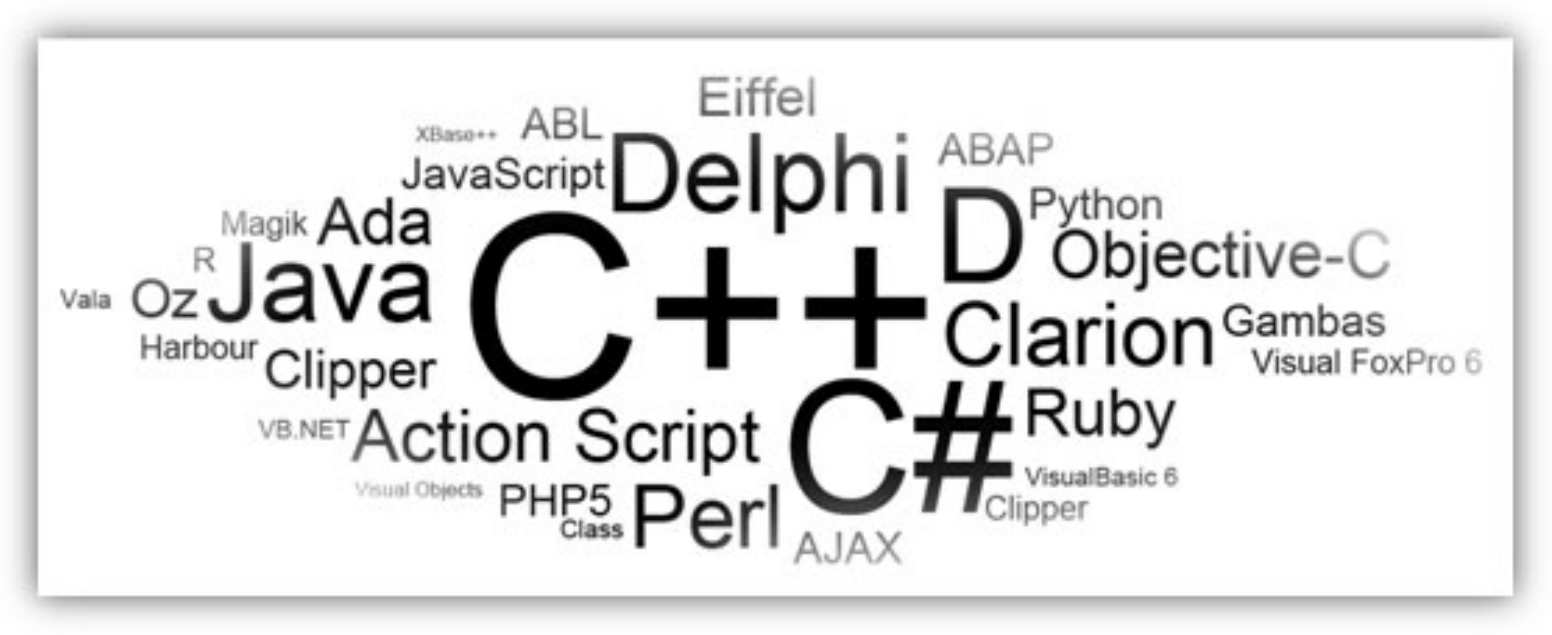
Estructuras de Control

- Ciclos con la decisión al inicio (*leading-decision loops):*

  Los ciclos en los cuales la decisión se evalúa al inicio, pueden implementarse usando:

```
100    IF (se cumple) GOTO 200
       . . . Haz lo que quieras . . .
       GOTO 100
200    . . .
```

Estructuras de Control



- ¿Puede proporcionar ejemplos de ciclos de los 2 tipos anteriores en los lenguajes de programación modernos?

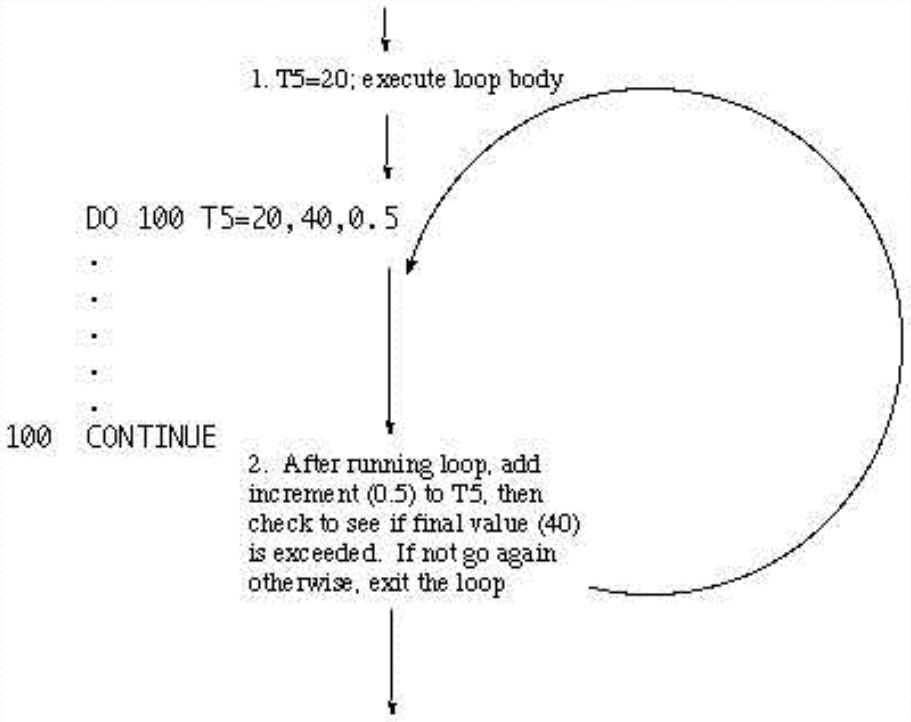
Estructuras de Control

- Ciclos con la decisión en medio (*mid-decision loops*):

  Los ciclos en los cuales la decisión se toma a la mitad de la estructura pueden implementarse usando:

```
100     . . . primera mitad del ciclo . . .
        IF (se cumple) GOTO 200
        . . . segunda mitad del ciclo . . .
        GOTO 100
200     . . .
```

Estructuras de Control



- ¿Cuándo usaríamos normalmente un ciclo donde la decisión se tome a la mitad de la estructura?

Estructuras de Control

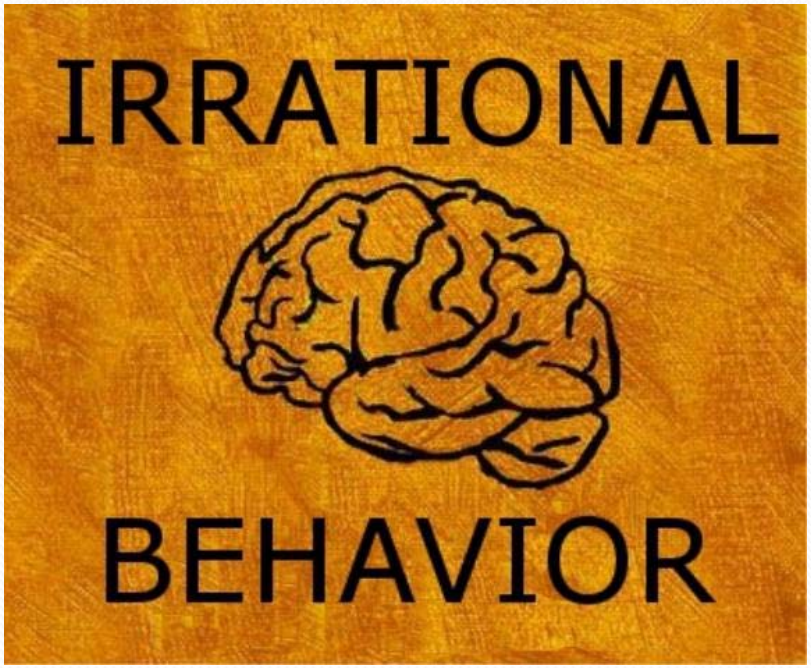
Control by branching - FORTRAN

```
    IF(X .EQ. Y) X=X+1

    IF(X .EQ. Y) GO TO 100
     << 'else' statements >>
    GO TO 200
100 CONTINUE   (dummy statement)
    << 'then' statements >>
200 CONTINUE
    << rest of program >>
```

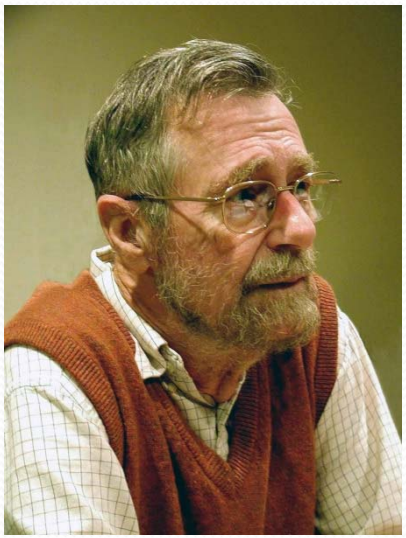
- La sentencia **GOTO** es muy poderosa, pero al mismo tiempo es muy primitiva y potencialmente peligrosa. ¿Por qué?

Estructuras de Control



- Aunque la sentencia **GOTO** nos permite implementar cualquier tipo de estructura de control, su uso irracional puede conducirnos a la elaboración de programas difíciles de leer y actualizar.

Estructuras de Control



- Además, el uso extensivo de la sentencia **GOTO** nos hace violar el **Principio Estructural** (de Dijkstra):

Debe ser posible visualizar el comportamiento del programa fácilmente a partir de su expresión escrita.

Estructuras de Control

- ¿Puede proporcionar ejemplos de ciclos con la decisión en medio en los lenguajes de programación modernos?

Estructuras de Control

- El abuso de la sentencia **IF** en FORTRAN es un error desde la perspectiva del diseñador de lenguajes, puesto que hace más difícil que un programador pueda identificar el propósito de una sentencia dentro de un segmento de código, a menos que se examine el comportamiento de las sentencias que la rodean.

Estructuras de Control

- La confusión entre el **GOTO** calculado (*computed*) y el **GOTO** asignado (*assigned*):

GOTO calculado: **GOTO**(L1, . . ., Ln),I

GOTO asignado: **GOTO** I, (L1, . . ., Ln)

Estructuras de Control

- Estos 2 **GOTO**s son muy similares, pero hacen cosas diferentes. El **GOTO** calculado es una sentencia de casos, mientras que el GOTO asignado salta a las sentencias cuya dirección está en la variable I (es decir, es un **GOTO** indirecto).

- Las etiquetas del ejemplo anterior son innecesarias y sólo se proporcionan para fines de documentación.

Estructuras de Control

- Problema # 1: El programador pone un **GOTO** calculado en vez de un **GOTO** asignado:

ASSIGN 20 TO N (La dirección de la sentencia 20 se almacena en N)

.

.

.

GOTO (20, 30, 40, 50), N

Estructuras de Control

- En este caso, la sentencia **ASSIGN** asignará la dirección de la sentencia 20 (p.ej., 156) a N. El **GOTO** calculado intentará usar este valor como índice a la tabla de saltos (20, 30, 40, 50).

- Puesto que el índice (156 en este caso) está fuera de rango, el compilador saltará a una dirección de memoria impredecible (a menos que el compilador detecte el error). Esto producirá un error muy difícil de detectar.

Estructuras de Control

- Problema # 2: El programador usa un **GOTO** asignado en vez de uno calculado:

```
I=5
.
.
.
GOTO I, (20, 30, 40, 50)
```

Estructuras de Control

- Puesto que el GOTO espera que la variable I contenga la dirección de la sentencia, se transferirá a la dirección de memoria 5, produciendo errores inesperados (p.ej., el sistema operativo puede colapsarse).

- El problema de los GOTOs constituye una violación al **Principio de Consistencia Sintáctica** (de MacLennan):

> Cosas que se ven similares deben hacer cosas parecidas y cosas que se ven distintas deben hacer cosas diferentes.

Sistema de Tipos

- FORTRAN tiene un sistema de tipos débil (al igual que C). Esto significa que no existe un mecanismo de chequeo estricto de tipos al momento de compilación.

- Otros lenguajes, tales como Pascal, tienen un sistema de tipos fuerte, que es capaz de detectar la mayoría de los errores sintácticos en tiempo de compilación y no de ejecución.

Sistema de Tipos

- Debido a su sistema de tipos débil, FORTRAN viola el **Principio de la Defensa en Profundidad** (de MacLennan):

> Si un error logra escapar una línea de defensa (chequeo sintáctico en este caso), entonces debiera ser atrapado en la siguiente línea de defensa (chequeo de tipos en este caso).

Subprogramas

- Los subprogramas son abstracciones procedurales (o abstracciones de control), lo que significa que nos ayudan a factorizar patrones comunes en nuestro algoritmo y reducen consecuentemente la cantidad de código necesario y generan soluciones más elegantes a un problema.

- Las subrutinas en FORTRAN toman parámetros que pueden o no regresar valores de la subrutina y constituyen una característica muy útil del lenguaje.

Subprogramas

```
SUBROUTINE MAXIMO(X, Y, M)
    M=X
    IF (X .LT. Y) M=Y
    RETURN
    END
```

Subprogramas

- Para llamar esta subrutina desde el programa principal (o desde otro subprograma), necesitamos usar:

CALL MAXIMO(VAL1, VAL2, RESULTADO)