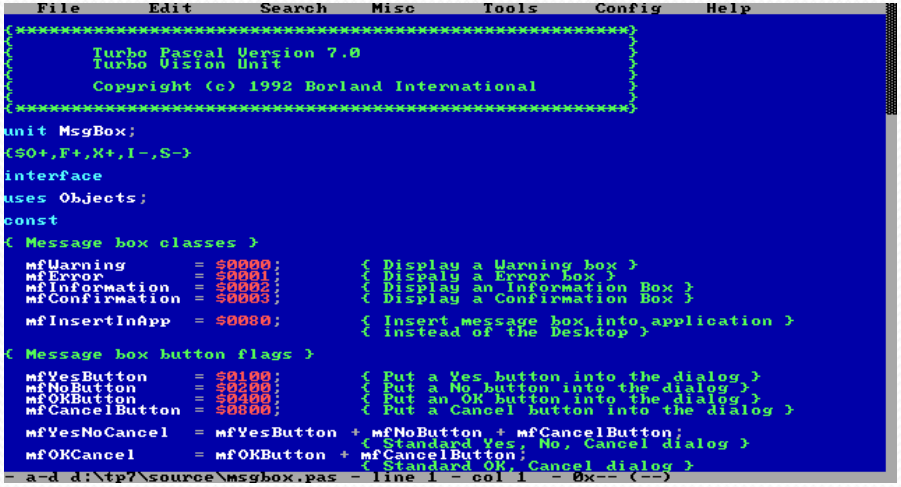
Lenguajes de Programación

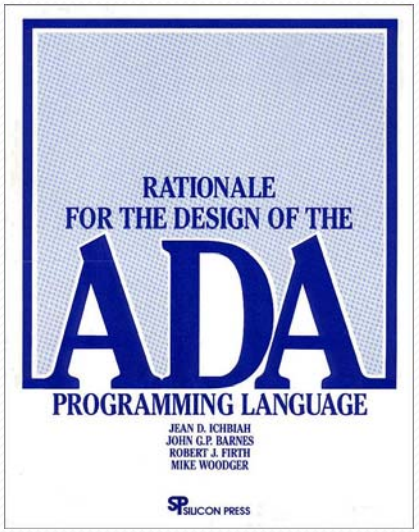
Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx

Evolución de los Lenguajes de Programación



- Pascal fue diseñado originalmente para escribir compiladores, pero su mayor éxito rebasó en mucho las expectativas de su diseñador, convirtiéndose en uno de los lenguajes de programación más importantes de todos los tiempos. A lo largo de los años, se desarrollaron muchas versiones y dialectos distintos de Pascal alrededor del mundo, si bien el lenguaje C acabó por desplazarlo hace unos 20 años.

Evolución de los Lenguajes de Programación



- **Cuarta Generación:** Ada (1980). Este lenguaje fue diseñado por un comité del Departamento de Defensa de los Estados Unidos. Se volvió una contribución importante a los lenguajes estructurados puesto que resuelve la mayor parte de los problemas presentes en las generaciones previas y resultó ser en cierta forma, el futuro de los lenguajes estructurados.

Evolución de los Lenguajes de Programación

Structure of a Concurrent Program

```
package Point_Pkg is
   procedure Assign(X, Y : in Float);
   -- Assigns X and Y as the new coordinates

   procedure Fetch(X, Y : out Float);
   -- Copies the current coordinates to X and Y
end Point_Pkg;
```

```
with Point_Pkg;
procedure Tasking_Prog is
  task T1;
  task body T1 is
  begin
   Point_Pkg.Assign(1.0, 2.0);
  end T1;

  task T2;
  task body T2 is
     X, Y : Float;
  begin
   Point_Pkg.Fetch(X, Y);
  end T2;
begin
  -- T1, T2 activated here
  -- Assign, Fetch executed with
  --  mutual exclusion
end Tasking_Prog;
```

```
package body Point_Pkg is
   -- Implementation for tasking programs

   protected Point is
      procedure Assign(X, Y : in Float);
      procedure Fetch(X, Y : out Float);
   private
      X, Y : Float;
   end Point;
   -- Invoking Point.Assign or Point.Fetch involves
   --  obtaining a "lock" on the Point object
   -- These operations are thus performed
   --  under mutual exclusion

   protected body Point is
      procedure Assign(X, Y : in Float) is
      begin
         Point.X := X;
         Point.Y := Y;
      end Assign;

      procedure Fetch(X, Y : out Float) is
      begin
         X := Point.X;
         Y := Point.Y;
      end Fetch;
   end Point;

   procedure Assign(X, Y : in Float) is
   begin
      Point.Assign(X, Y);
   end Assign;

   procedure Fetch(X, Y : out Float) is
   begin
      Point.Fetch(X, Y);
   end Fetch;
end Point_Pkg;
```

Figure 2

This shows an example of Ada tasking with a protected object. Notice that the interface (the specification of package Point_Pkg) is the same as in Figure 1, but here the implementation is safe for usage in a tasking program.

- Aunque es gigantesco, Ada se considera uno de los mejores lenguajes de programación jamás diseñados y su similitud con Pascal lo hacen relativamente fácil de aprender, aunque es difícil de dominar debido a la gran cantidad de instrucciones que posee.

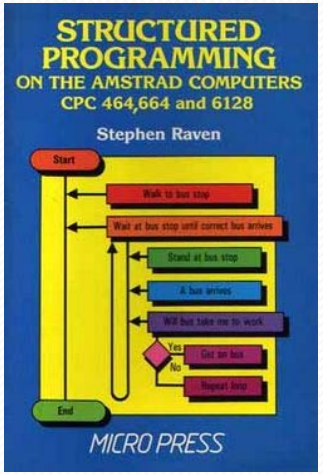
Evolución de los Lenguajes de Programación

- **Quinta Generación:** ??????. Todavía no sabemos cuál será la siguiente generación de lenguajes de programación. Los candidatos principales son: lenguajes paralelos, lenguajes de flujo de datos, lenguajes restringidos, lenguajes de dominio específico, lenguajes para programación móvil, funcionales, orientados a objetos y lógicos.

Paradigmas de Programación

- Programación Estructurada
- Programación Orientada a Objetos
- Programación Orientada a las Funciones
- Programación Orientada a la Lógica

Programación Estructurada

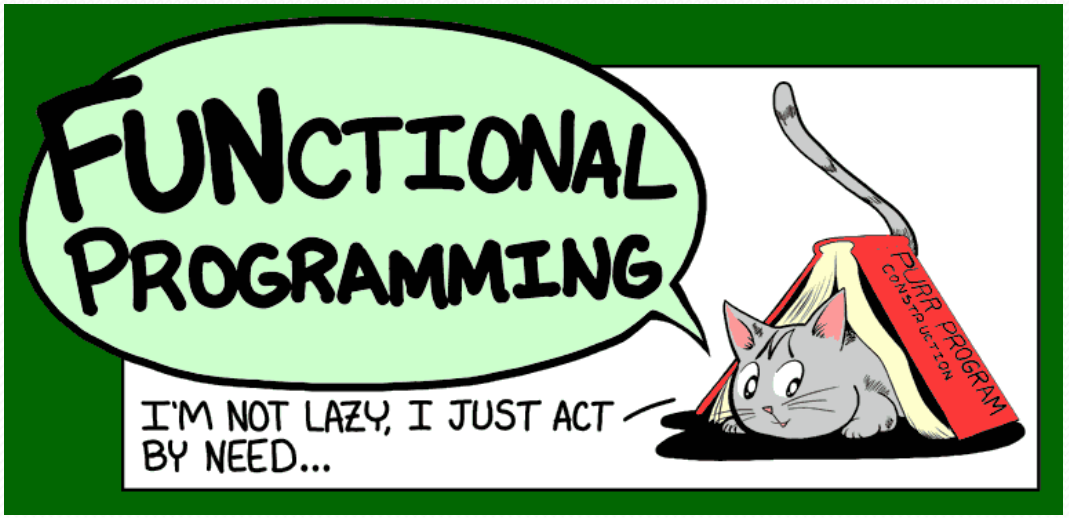


- Ejemplos: ALGOL-60, Ada, Pascal, C, Modula-2, BASIC, FORTRAN. En este caso, el bloque es la estructura principal del lenguaje. Normalmente, se usan dos símbolos (tales como "{" y "}" en C) o palabras reservadas (tales como "BEGIN" y "END" en Pascal) para indicar el principio y el final de un bloque. Estos lenguajes son descendientes de ALGOL-60.

Programación Orientada a Objetos

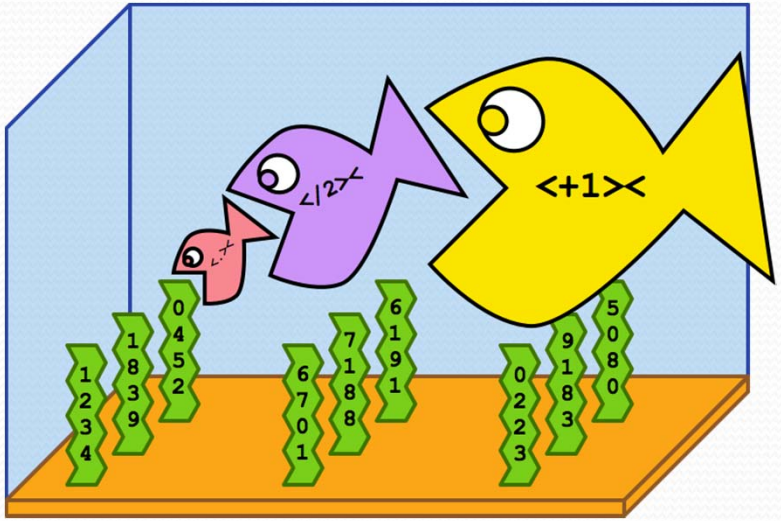
- Ejemplos: C++, Smalltalk, Eiffel. En estos lenguajes, la unidad básica de codificación es el “objeto”, que es una entidad que tiene su propia identidad y oculta información innecesaria del resto del mundo. El concepto de clase se originó con el Simula 60 y fue adoptado posteriormente en lenguajes como C++.

Programación Orientada a las Funciones



- Ejemplos: Scheme, LISP, Haskell, ML. En estos lenguajes, las funciones son la principal entidad del lenguaje, y éstas son tratadas como ciudadanos de primera clase, lo que significa que pueden ser regresadas por una función, asignadas a otra función y pasadas como argumento a cualquier función.

Programación Orientada a las Funciones

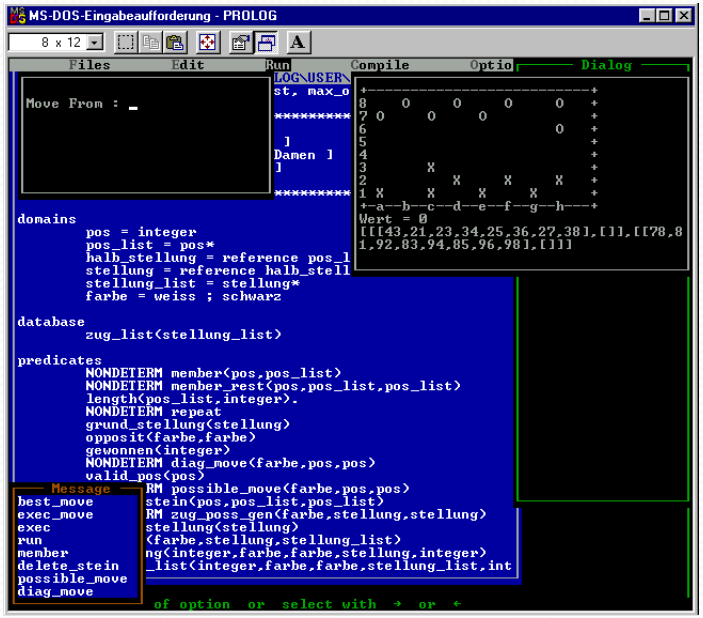


- Los lenguajes funcionales puros no tienen variables, saltos incondicionales (goto's), ni ciclos. En lugar de eso, se usan acoplamientos estáticos (*static bindings*) y la recursión se usa para las iteraciones. Scheme y LISP no tienen tipos y escribir un compilador para ellos resulta, por tanto, una tarea difícil.

Programación Orientada a la Lógica

- Ejemplo: PROLOG. En estos lenguajes es posible alcanzar el más alto nivel de abstracción (al menos teóricamente), puesto que el estilo de programación es declarativo.

Programación Orientada a la Lógica



- Esto significa que en vez de tener que escribir programas, el usuario escribe cláusulas con aserciones y reglas, y el lenguaje se encarga de decidir cuál es la respuesta (o respuestas) a una cierta consulta. Los lenguajes orientados a la lógica todavía distan bastante de la perfección y se requiere mucho trabajo para volverlos útiles en la práctica.

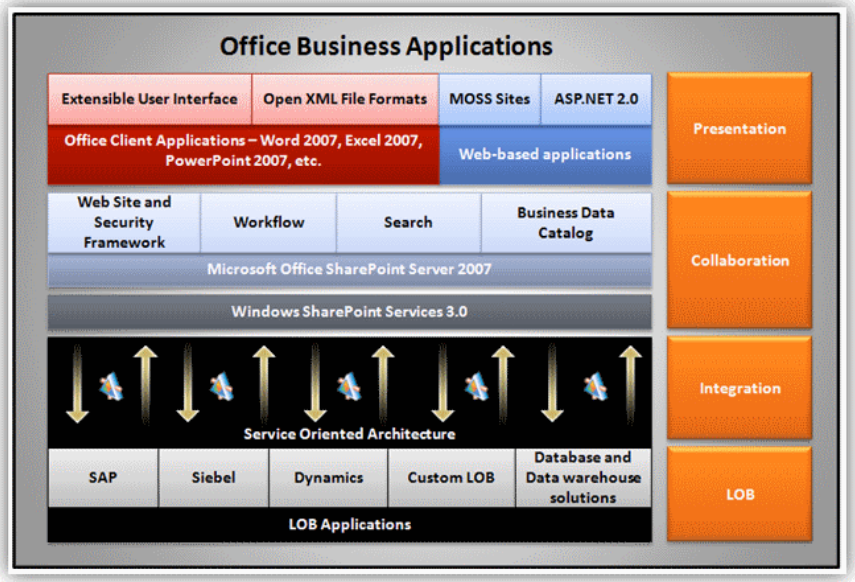
Dominios de la Programación

- Aplicaciones Científicas
- Aplicaciones de Negocios
- Inteligencia Artificial
- Lenguajes para Programación de Sistemas
- Lenguajes de Propósito Especial

Aplicaciones Científicas

- FORTRAN, ALGOL-60 y la mayor parte de sus descendientes han sido usados para este propósito. La eficiencia es la mayor motivación detrás de estos lenguajes. Deben proporcionarse en la versión estándar de estos lenguajes operaciones de punto flotante, así como buenas bibliotecas matemáticas.

Aplicaciones de Negocios

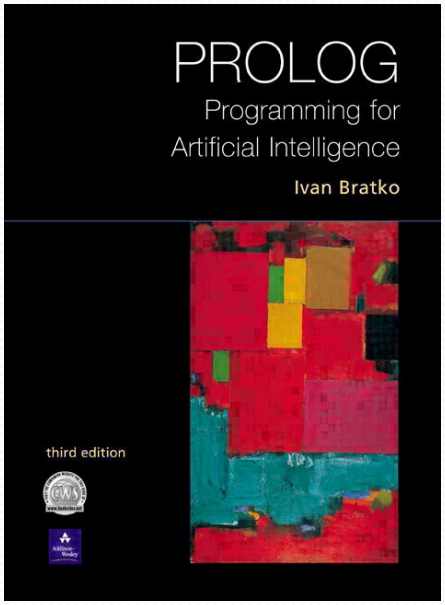


- COBOL y muchas otras herramientas desarrolladas en años recientes tales como Power House, FoxBase, CLIPPER, Oracle, Paradox, DB2, Microsoft SQL Server, etc. Los sistemas manejadores de bases de datos y muchas otras herramientas han sido ampliamente utilizadas por los usuarios finales recientemente para escribir sus propias aplicaciones.

Aplicaciones de Negocios

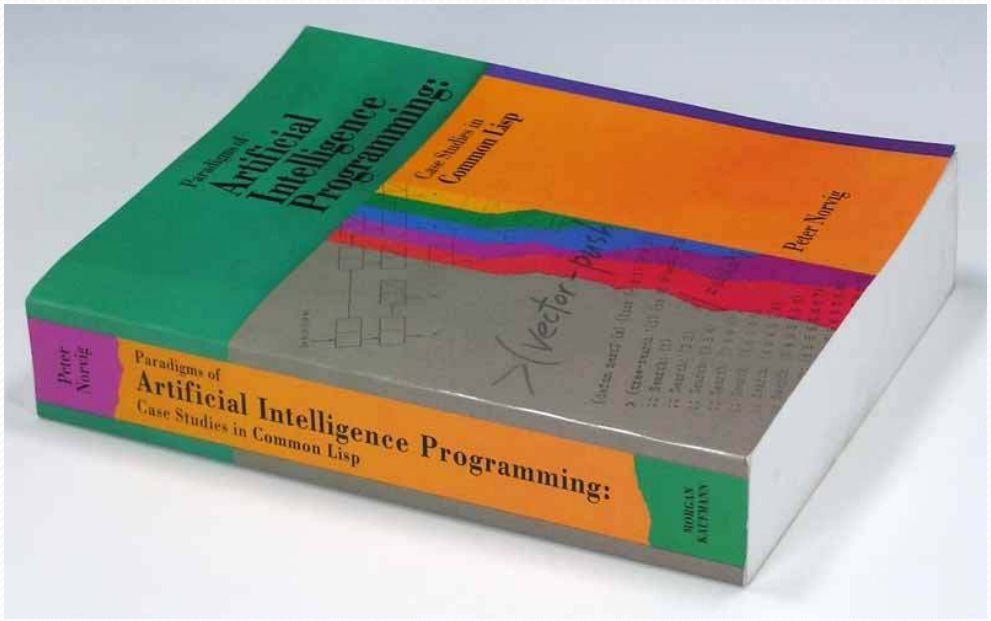
- El énfasis de este tipo de lenguajes es su facilidad de uso. Normalmente cuentan con muchas funciones orientadas al manejo de bases de datos, funciones muy elaboradas para entrada y salida y manejo de precisión fija (muy importante para contabilidad).

Inteligencia Artificial



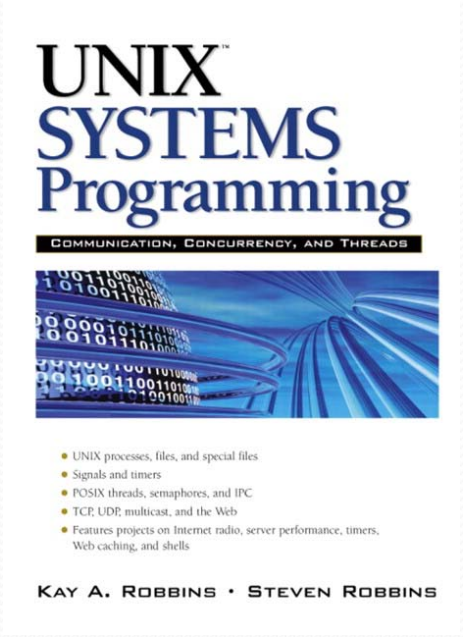
- LISP y PROLOG son los principales lenguajes de programación usados en esta área. El objetivo de tratar de escribir programas “inteligentes” es muy ambicioso, pero los lenguajes que para ello se utilizan son de muy alto nivel y permiten un gran nivel de abstracción.

Inteligencia Artificial



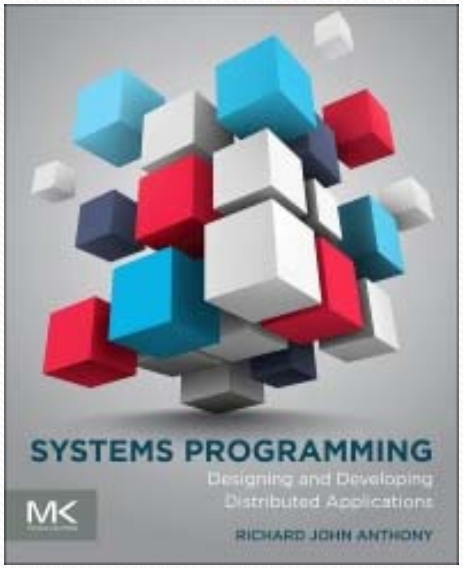
- El procesamiento simbólico es el principal énfasis de estos lenguajes, en vez del procesamiento numérico, el cual se considera secundario en este caso. El uso de abstracción y capacidad de incorporar conocimiento fácilmente es la principal motivación detrás del diseño de estos lenguajes.

Lenguajes para Programación de Sistemas



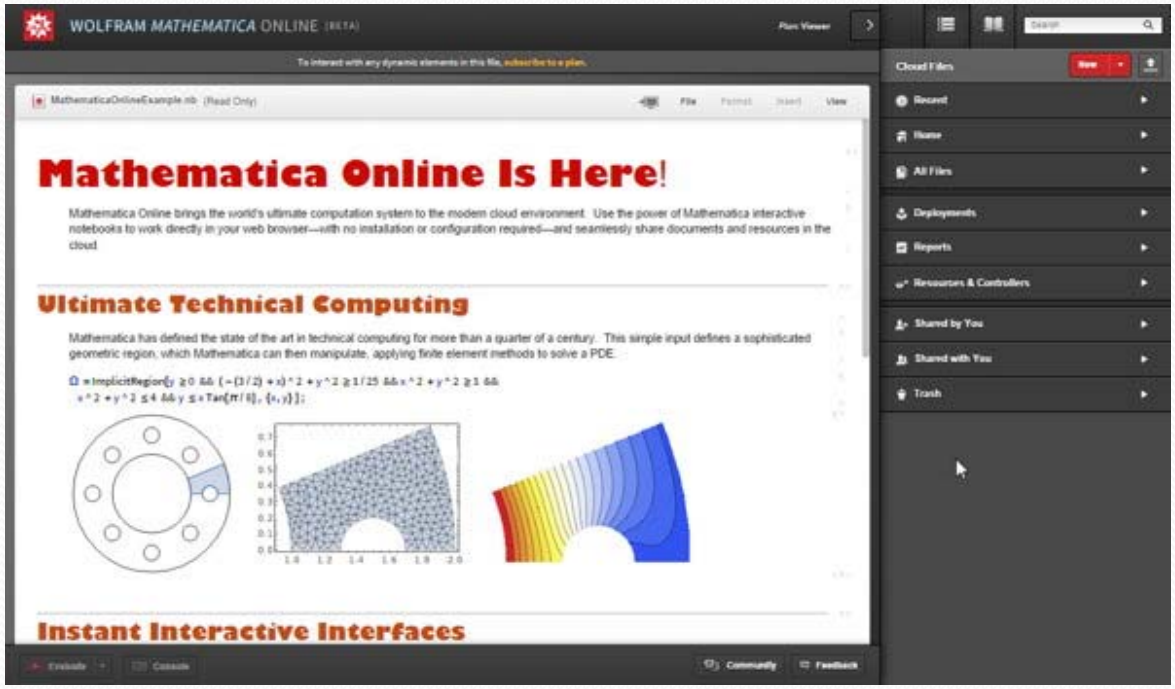
- Ensamblador, C, Modula-2, etc. Estos lenguajes están destinados a la escritura de sistemas operativos y software relacionado con ellos (p.ej., editores, depuradores, etc.). En este caso se requiere tener acceso a operaciones de bajo nivel de una forma eficiente y confiable.

Lenguajes para Programación de Sistemas



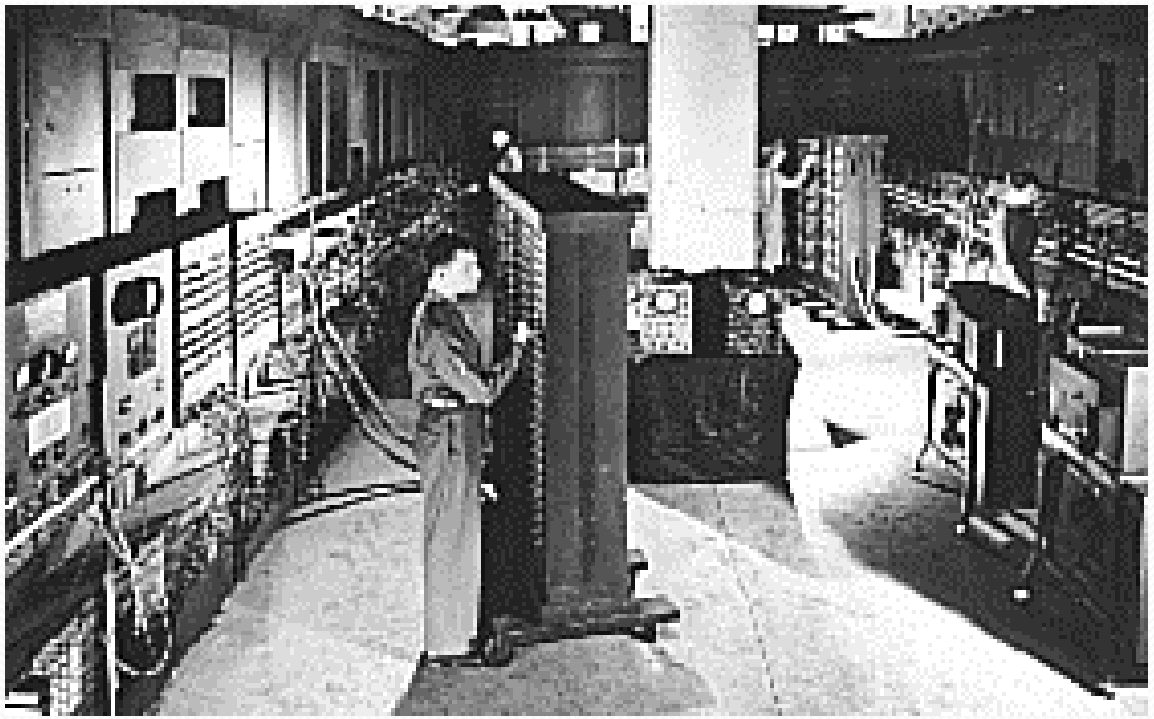
- Los lenguajes de este grupo son normalmente más difíciles de aprender y extremadamente difíciles de dominar, dado el alto número de inconsistencias y excepciones contenidas en sus gramáticas, las cuales dan pie a sofisticados “trucos” de programación que lenguajes de más alto nivel no permiten.

Lenguajes de Propósito Especial



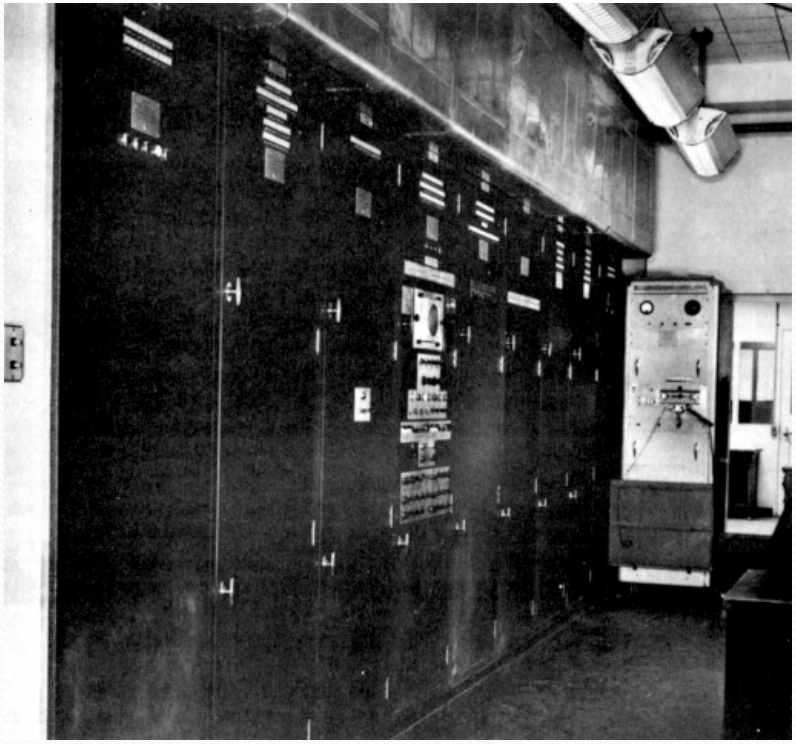
- Mathematica, Auto-LISP, RPG, GPSS, MatLab, etc. Lenguajes destinados a una tarea específica. Normalmente son pequeños y simples, y la motivación detrás de ellos es permitir que usuarios finales diseñen sus propios programas dentro de un cierto ambiente de software.

Intérpretes de Pseudo-Código



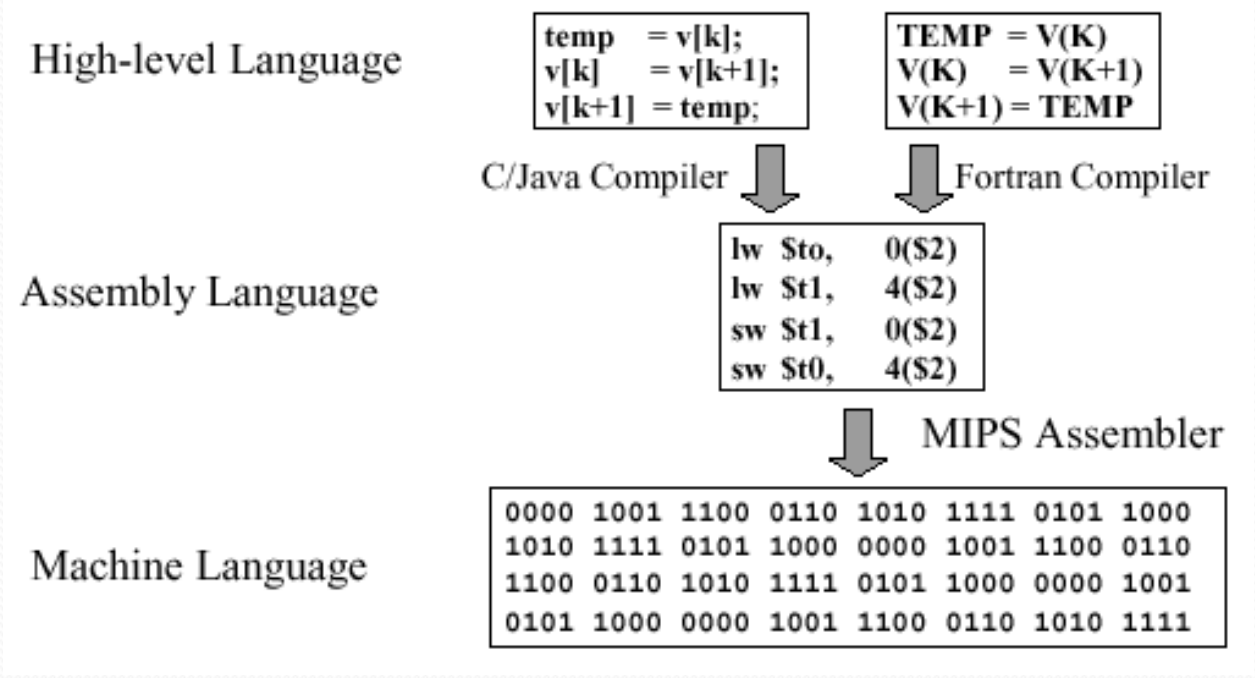
- A fines de los 1940s, las computadoras tenían memorias muy pequeñas (del orden de 1K), y por tanto se requería código muy compacto.

Intérpretes de Pseudo-Código



- Las computadoras de esa época también eran muy lentas, por lo que se ponía gran énfasis en la eficiencia en tiempo de ejecución.

Intérpretes de Pseudo-Código



- La programación en ese entonces se realizaba sin ningún tipo de herramientas de software (ni siquiera ensambladores). Se programaba en lenguaje máquina (o sea, en binario).

Lenguaje Máquina

- Extremadamente simple y de bajo nivel.
- Su conjunto de instrucciones era extremadamente irregular y por lo tanto era difícil recordarlo.
- Cero portabilidad. Las instrucciones eran específicas a cada computadora en particular.

Lenguaje Máquina

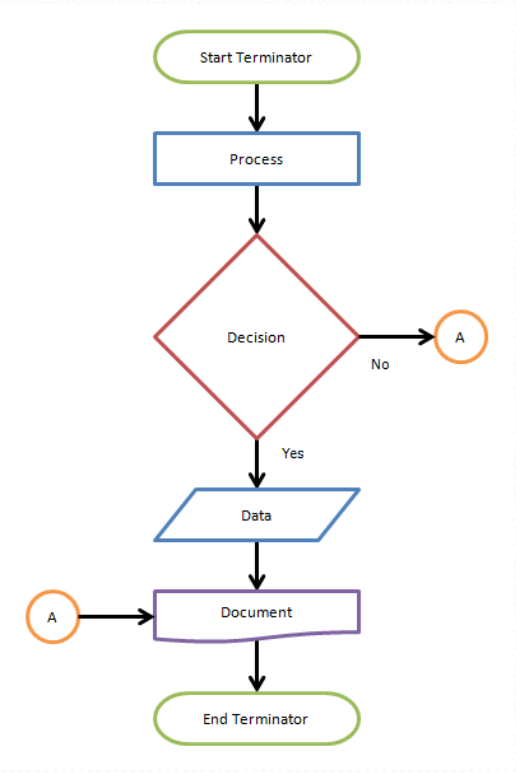
- Los problemas numéricos requerían de un escalamiento manual de los números.
- Las operaciones de punto flotante y la indización no se proporcionaban en hardware.
- Las instrucciones no estaban diseñadas para ser leídas o escritas por humanos (resultaban muy difíciles de entender).

Notaciones de Diseño

- La programación en esta época era sumamente tediosa.

- Eso motivó el desarrollo de herramientas (llamadas notaciones de diseño), cuyo propósito era facilitar la codificación de programas.

Notaciones de Diseño



- John von Neumann y Herman Goldstine inventaron los **diagramas de flujo**. Estas ayudas eran representaciones gráficas de los algoritmos que facilitaban en cierta medida la programación.

Notaciones de Diseño

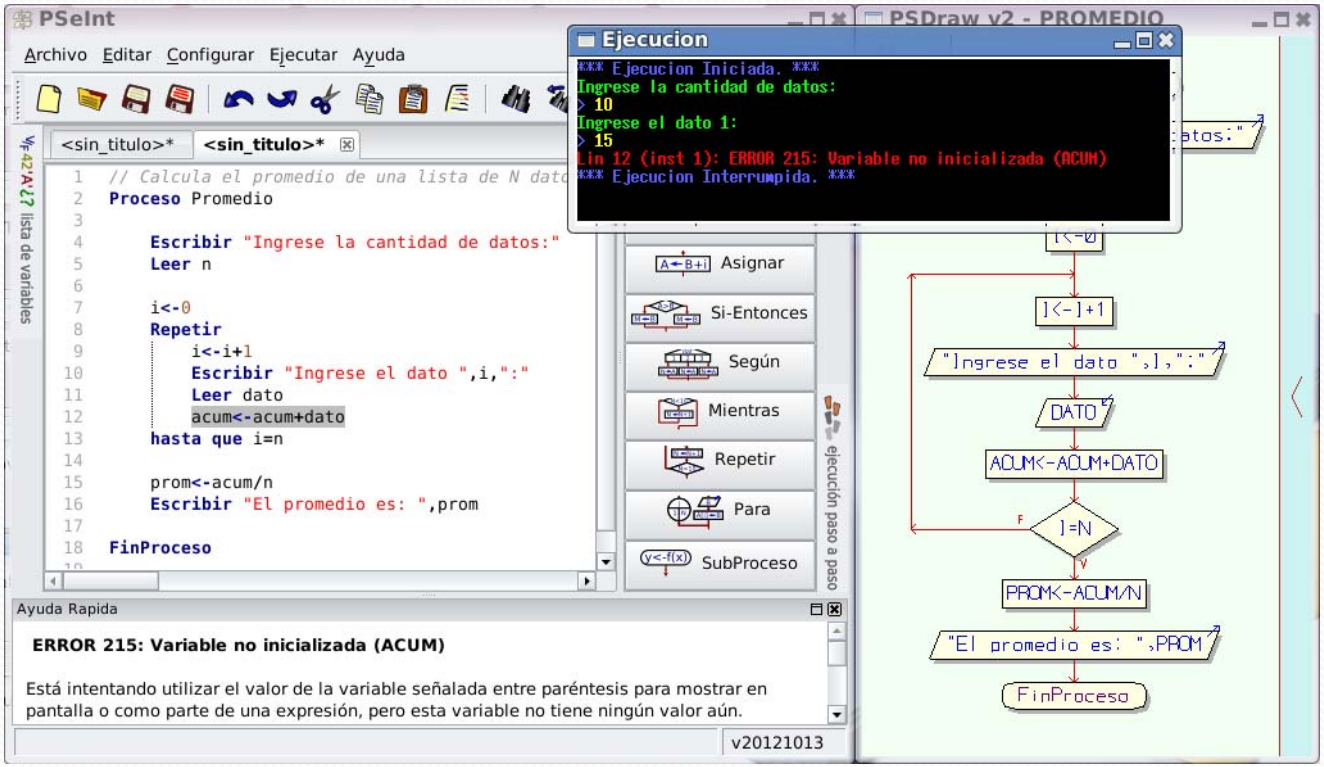
```
mult.asm        Wed Feb 02 15:42:09 2000        1
; Impliments a simple 32 bit integer multiply by successive addition
; R. H. Klenke, Sun Jan 31 10:45:14 EST 1999
;

Multp:  .equ    32              ; multiplicand
Multd:  .equ    64              ; multiplier
        .org    4096            ; 0x1000
        lar     r30, Done       ; Load address of Done for branch
        lar     r31, Loop       ; Load address of Loop for branch
        la      r1, Multd       ; Load multiplier
        la      r2, Multp       ; Load multiplicand
        andi    r3, r3, #0      ; clear r3
        andi    r4, r4, #0      ; clear r4
        addi    r5, r3, #1      ; place 1 in r5
        neg     r3, r5          ; place -1 in r3
Loop:   add     r4, r4, r2      ; add multiplicand to running sum
        add     r1, r1, r3      ; start loop, decrement multiplier
        brzr    r30,r1          ; jump to Done if multiplier = 0
        br      r31             ; jump back to Loop
Done:   st      r4, Result      ; store result
        stop
        .org    8192            ; 0x2000
Result: .dw     1               ; storage for result
```

- **Ensambladores:** Usan letras en vez de números para denotar instrucciones. Por ejemplo: add 01 34.

- En ensamblador, los comandos son mnemónicos y cada instrucción en ensamblador genera exactamente una instrucción en lenguaje máquina.

Notaciones de Diseño



- **Intérpretes de pseudo-código:** En este caso, se proporciona una abstracción de más alto nivel que con los ensambladores. La abstracción pertenece a una computadora más regular y útil: una máquina virtual.

Notaciones de Diseño

- Los primeros pseudo-códigos aparecieron en 1951 (en un libro seminal de Maurice Wilkes sobre programación).

Notaciones de Diseño

```
[(m n) (ack)
 ((ack  (if (= m 0) done next))
  (next (if (= n 0) ack0 ack1))
  (done (return (+ n 1)))
  (ack0 (n := 1)
        (goto ack2))
  (ack1 (n := (- n 1))
        (n := (call ack m n))
        (goto ack2))
  (ack2 (m := (- m 1))
        (n := (call ack m n))
        (return n)) ))
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } n = 0 \\ A(m-1,A(m,n-1)) & \text{otherwise} \end{cases}$$

- Los primeros intérpretes de pseudo-código fueron inspirados sobre todo por el hecho de que las rutinas de punto flotante y de indización simplificaban considerablemente el proceso de programación.

Evolución del Pseudo-Código

Store Doubleword with Update DS-form

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

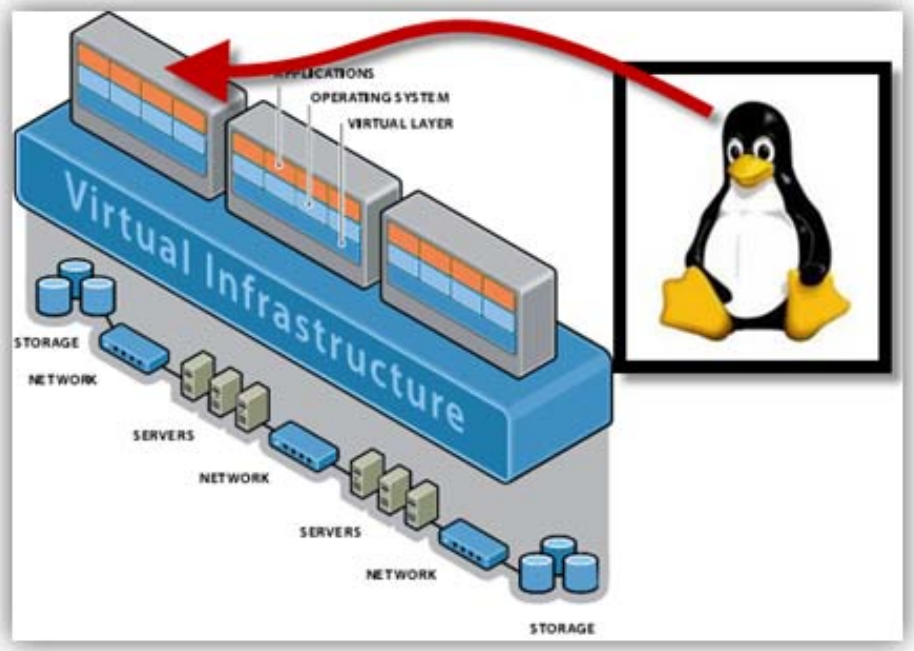
None

- En consecuencia, se popularizaron pequeñas rutinas desarrolladas para realizar operaciones de punto flotante e indización.

Evolución del Pseudo-Código

- La manera de usar estas rutinas era la siguiente:
  - Llamarlas como subrutinas.
  - El programa pasa a ser entonces una lista de estas subrutinas y sus datos.
  - Insertar el código de las subrutinas como si fuese una expansión de macros (compilación).

Evolución del Pseudo-Código



- La principal ventaja del pseudo-código era que proporcionaba una computadora virtual, la cual era más regular y de más alto nivel que las computadoras verdaderas que existían en aquellos días.

Evolución del Pseudo-Código

- Otra ventaja era que el pseudo-código decrementaba las oportunidades de equivocarse, al quitarle al programador la tarea de repetir procesos tediosos y propensos a errores.

Evolución del Pseudo-Código

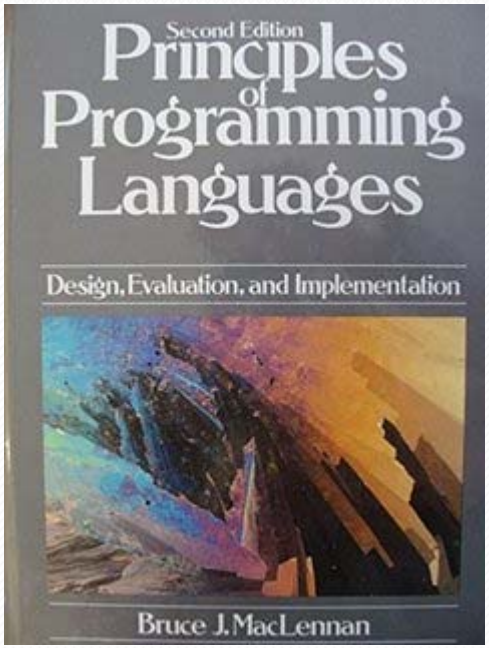
- También se incrementaba la seguridad al permitir chequeo de errores de, por ejemplo, variables no declaradas y referencias a posiciones de un arreglo que estuviesen fuera de su límite de definición.

Evolución del Pseudo-Código

- Simplificaba la depuración al proporcionar funciones tales como el rastreo (*trace*) de una ejecución.

- La principal desventaja del pseudo-código es que hacía más lenta la ejecución de un programa.

Principio de la Automatización



Automatiza las actividades mecánicas, tediosas o propensas a errores.

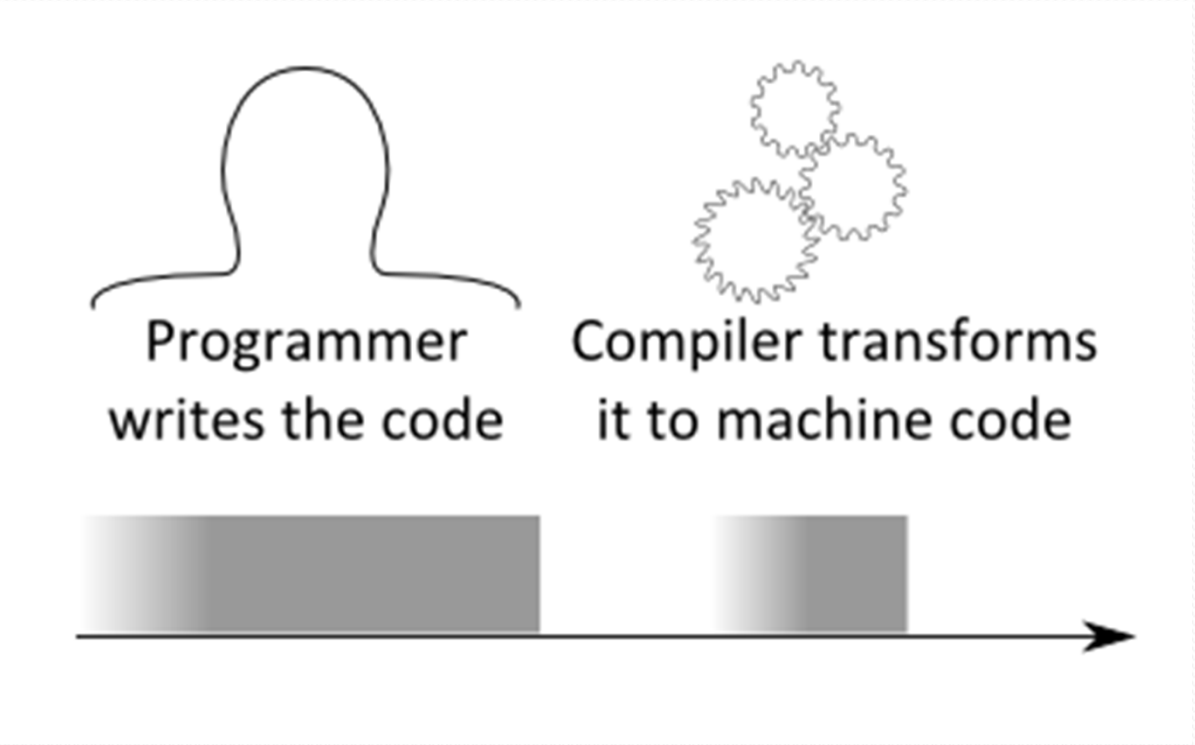
Evolución del Pseudo-Código

- El surgimiento de computadoras que implementaban en hardware las operaciones de punto flotante y las indizaciones hizo que el pseudo-código se volviera obsoleto.

Evolución del Pseudo-Código

- En su lugar, se volvió popular la idea de “compilar” programas a partir de bibliotecas de subrutinas.

Evolución del Pseudo-Código



- Sin embargo, la compilación seguía siendo considerada como ineficiente y se usaba solamente para programas cortos que se usarían muchas veces.

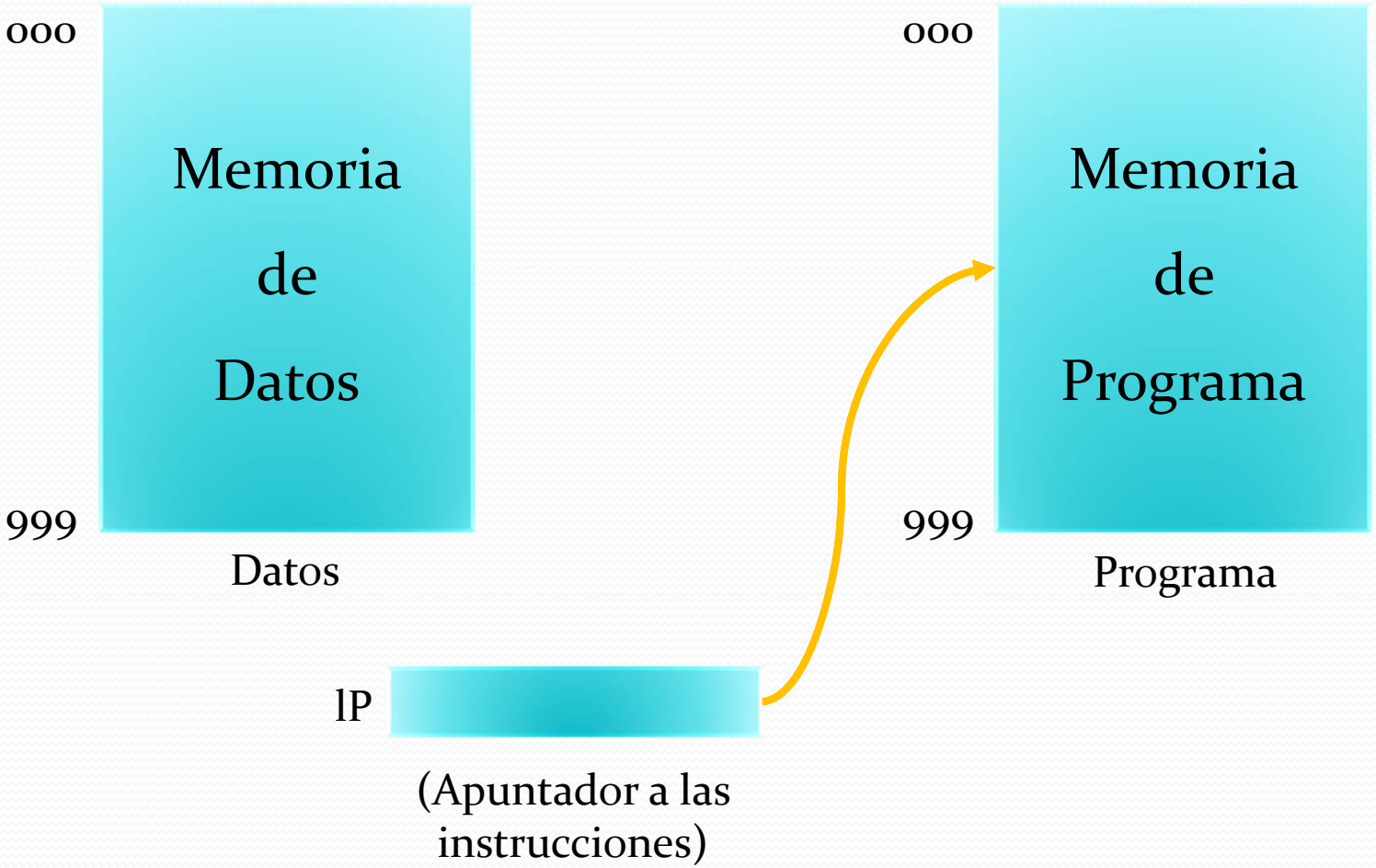
Implementación de Intérpretes de Pseudo-Código

- ¿Qué es lo que necesitamos para construir un intérprete de pseudo-código?
  - **Estructuras de datos:** Memoria de datos, memoria de programa y un apuntador de instrucciones.
  - **Representación:** Tanto la memoria de programa como la de datos son arreglos.

Implementación de Intérpretes de Pseudo-Código

- **Ciclo Lee-Ejecuta** (esto implica que nuestro intérprete es iterativo):
  - Leer una instrucción en memoria (PI).
  - Avanzar el PI.
  - Decodificar la instrucción.
  - Obtener operandos (de ser necesario) y ejecutar la operación.
  - Ir al primer paso

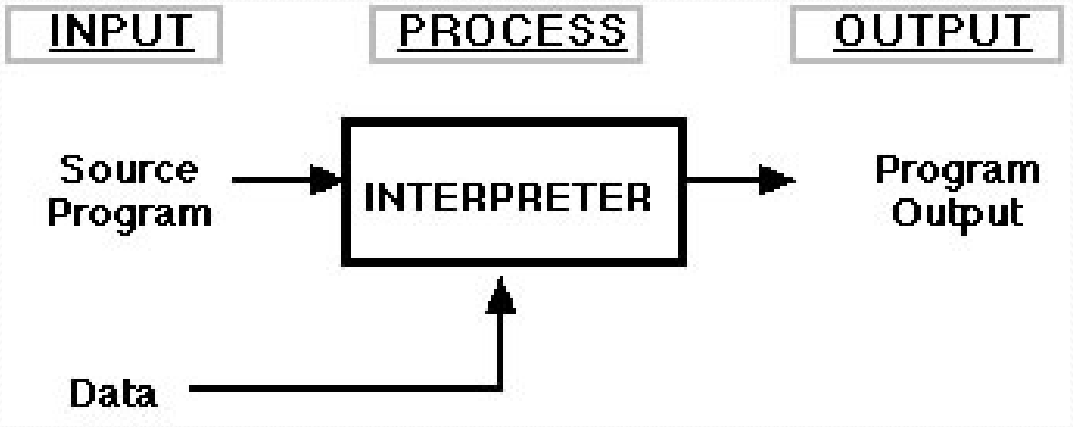
Implementación de Intérpretes de Pseudo-Código



Implementación de Intérpretes de Pseudo-Código

- La decodificación consiste básicamente de una extracción de campos (a partir de una cadena numérica).
- La operación de obtención (*fetch*) es manejada mediante subíndices de un arreglo.

Implementación de Intérpretes de Pseudo-Código



- Los cálculos se realizan en el lenguaje huésped (lo cual hace que el proceso sea más lento).
- El flujo de control se realiza cambiando el PI.

Implementación de Intérpretes de Pseudo-Código

- El proceso de **depuración** abarca lo siguiente:
  - Rastreo de ejecución (*trace*), el cual puede activarse o desactivarse.
  - Puede rastrearse la salida del programa usando "*pretty printing*".
  - Pueden colocarse puntos de quiebra y operaciones de evaluación del contenido de memoria.

Mejoras al Intérprete

- **Etiquetas simbólicas:**
  - Etiquetas para las sentencias (instrucciones). Su uso hace mucho más fácil el poder sumar posiciones de memoria. Sin embargo, su uso requiere de una operación previa de definición de etiquetas.

Mejoras al Intérprete

- **Etiquetas para las variables.** Se requiere una operación para declarar variables, así como sus tipos (arreglos contra punto flotante).

- **Asociación (binding).** Las declaraciones asocian una cierta etiqueta con una posición de memoria (del programa o de datos).

Mejoras al Intérprete

- **Implementación de etiquetas simbólicas:**
  - Recorrer el listado del programa buscando una etiqueta.
  - Construir una tabla (simbólica) que traduzca etiquetas a posiciones absolutas de memoria. Esta tabla puede checar que las etiquetas sean únicas, así como también el que se hayan definido antes de usarse.

Mejoras al Intérprete

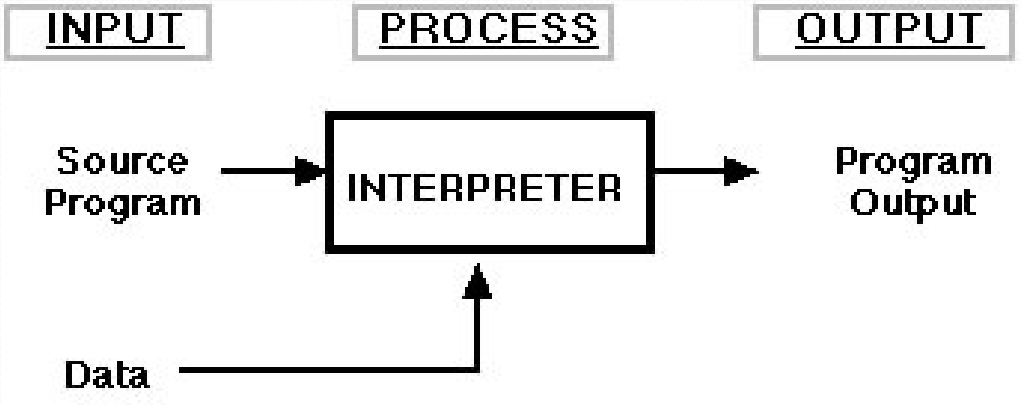
- En la primera pasada, se inicializa la tabla a “no definida”. Si se encuentra una referencia antes de una definición, entonces cambiar el valor a “referenciada”.

- Si se encuentra una definición de una etiqueta, entonces cambiar el valor a “definida”. Una vez que termine el proceso de recorrido del programa, checar la existencia de los valores “referenciados”.

Mejoras al Intérprete

- **Formato (sintaxis):**
  - **Uso de caracteres.** Esto es un mero cambio de la sintaxis que, sin embargo, mejora considerablemente la legibilidad de los programas.
  - **Formato fijo.** Este formato facilita el proceso de evaluación de expresiones (*parsing*).

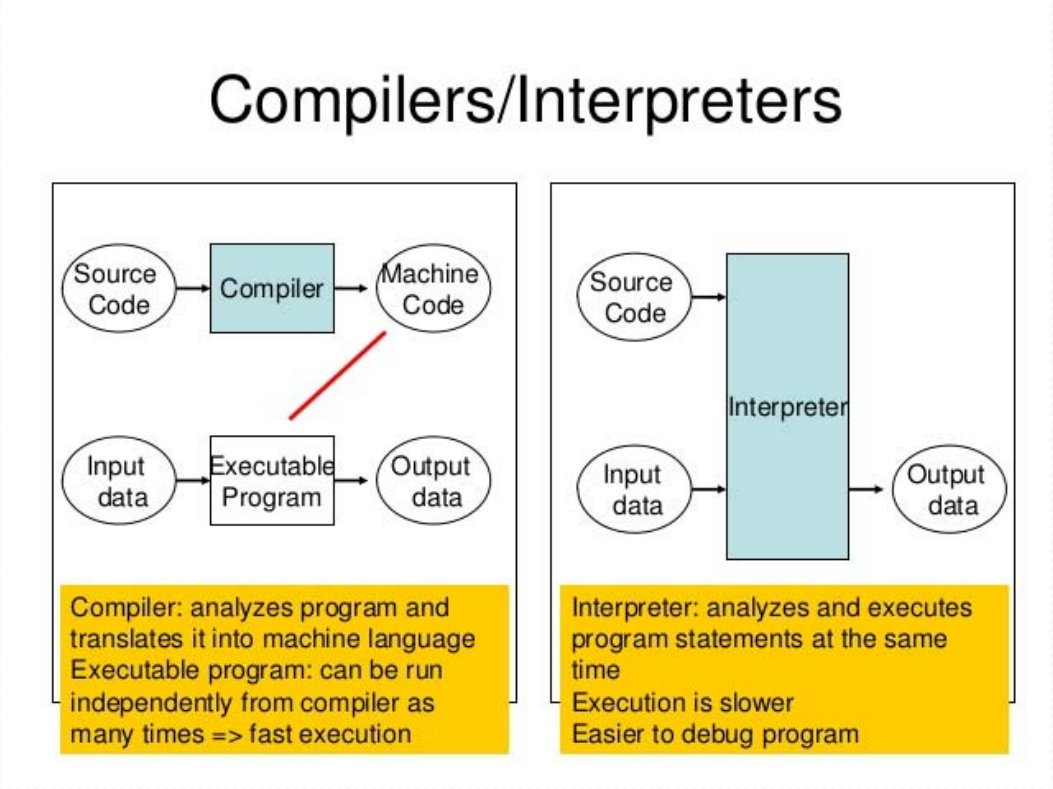
Mejoras al Intérprete



- **Formato variable (libre).** El ignorar los espacios en blanco da más flexibilidad al programador.

- **Implementación.** El cargador traduce los caracteres en el formato interno anterior, usando la tabla de símbolos para las operaciones, variables y etiquetas. La vieja sintaxis está ahora en un formato “interno”.

Mejoras al Intérprete



- **Comparación con los compiladores.** El pseudo-código tiene el mismo principio básico que los compiladores: usa una tabla de nombres y efectúa asignación de espacios de almacenamiento.

Ejemplo de Diseño de un Pseudo-Código

- ¿Qué tipos de datos necesitamos?
  - Operaciones de punto flotante: aritméticas, comparaciones, funciones de entrada/salida.
  - Arreglos: inicialización, indización.
  - Flujo de control (basado en resultados de comparaciones), ciclos.

Ejemplo de Diseño de un Pseudo-Código

- Operaciones de punto flotante:
  - Aritméticas
    - identidad (mover).
    - suma, resta, multiplicación, división, elevar al cuadrado, raíz cuadrada, logaritmo.

Ejemplo de Diseño de un Pseudo-Código

- Comparación y flujo de control
  - Saltar si los 2 argumentos son iguales; saltar si no lo son.
  - Saltar si argumento1 es mayor o igual que argumento2; saltar si argumento1 es menor que argumento2.
- Entrada/Salida
  - Leer, imprimir

Ejemplo de Diseño de un Pseudo-Código

- Operaciones con arreglos:
  - Inicialización (no se hace en tiempo de ejecución).
  - Indización (asignación de un elemento, extracción de elementos).

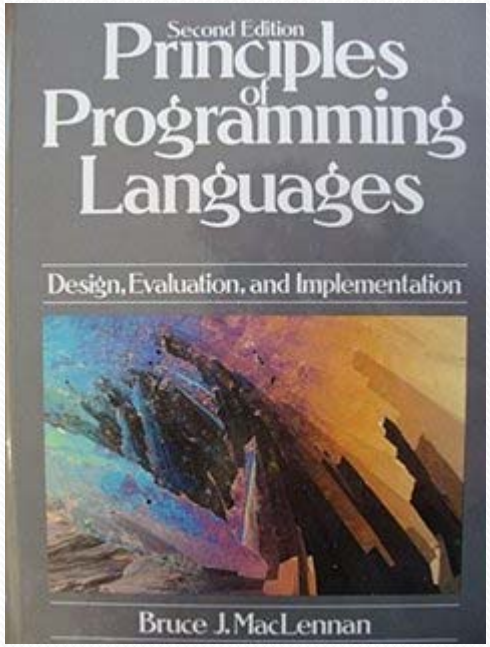
Ejemplo de Diseño de un Pseudo-Código

- Flujo de control:
  - Comparaciones (como antes).
  - Ciclos. Inicialización. Incrementar y evaluar y quizás decrementar y evaluar.
  - Finalizar (programa).

Ejemplo de Diseño de un Pseudo-Código

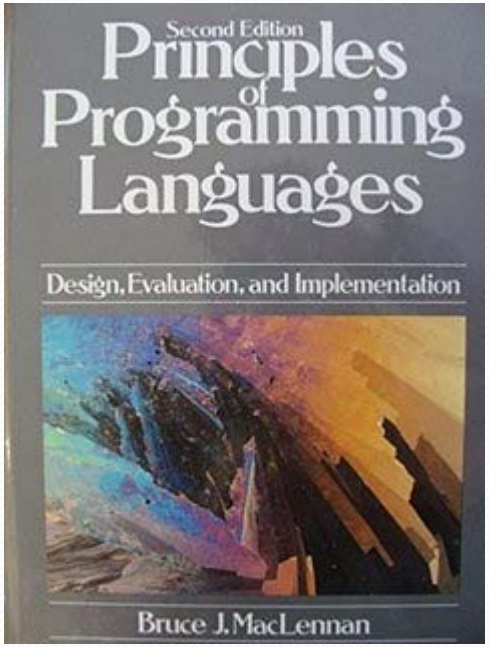
- ¿Por qué usar ciclos?
  - Abstracción. Evitamos repetir el cuerpo del ciclo muchas veces.
  - Pueden implementarse eficientemente.
  - La aritmética sólo aplica a las operaciones de punto flotante.

Principio de Regularidad



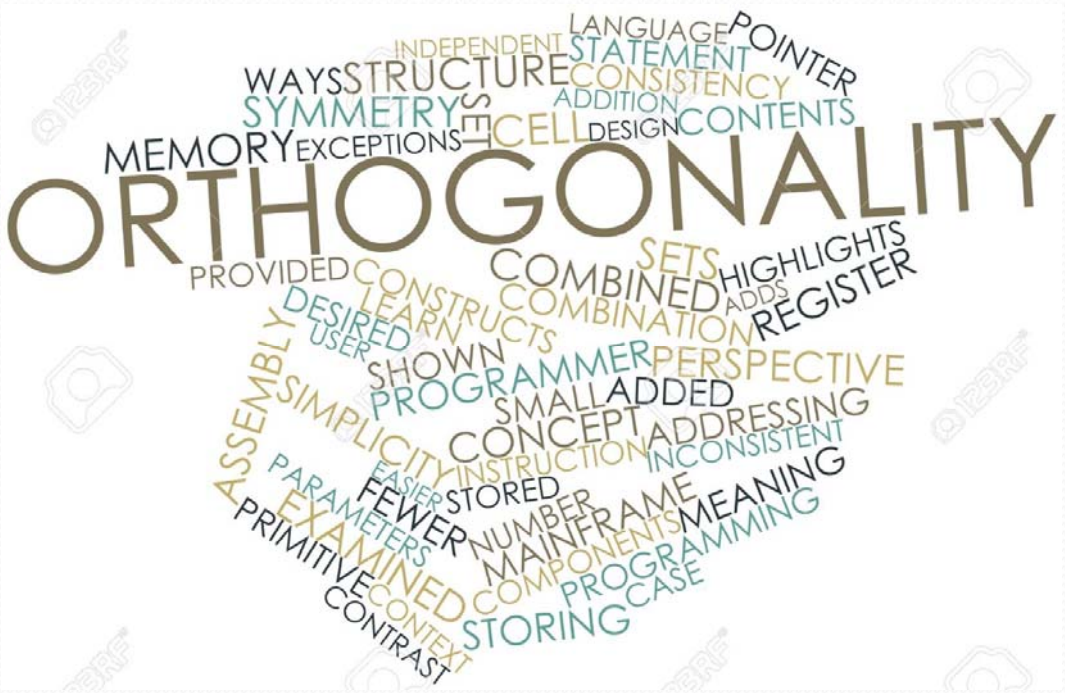
Las reglas regulares, sin excepciones, son más fáciles de aprender, usar, describir e implementar.

Principio de Ortogonalidad



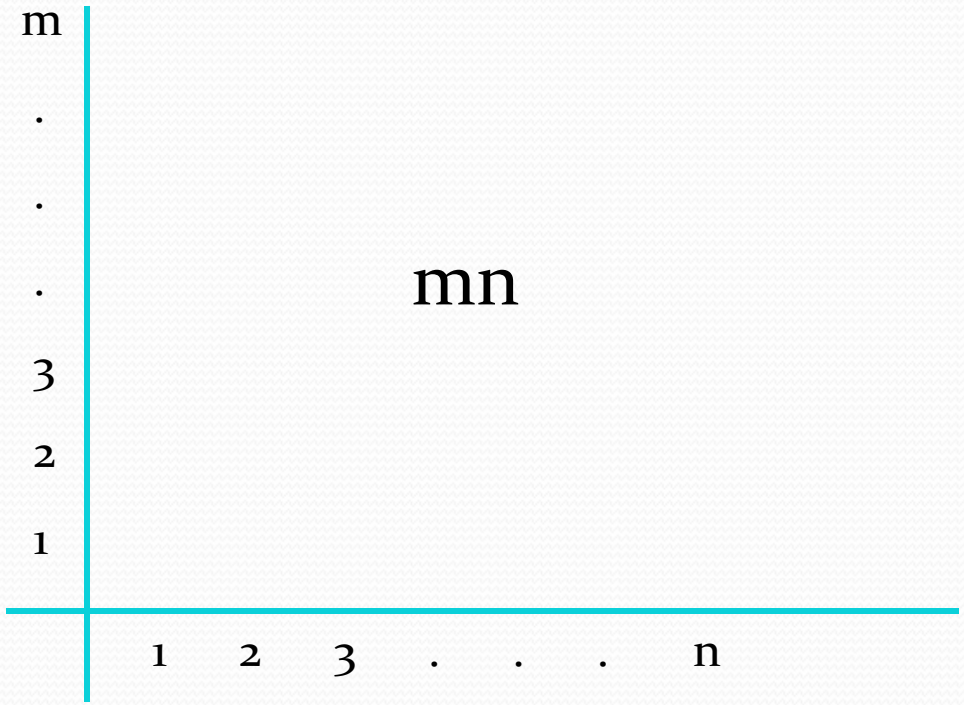
Funciones independientes deben ser controladas por mecanismos independientes.

Principio de Ortogonalidad



- El pseudo-código diseñado presenta **ortogonalidad**: funciones independientes son controladas por mecanismos independientes. Este principio nos permite realizar m*n cosas, pero teniendo que memorizar sólo m+n.

Principio de Ortogonalidad



Principio de Ortogonalidad

- Debe tenerse en cuenta, sin embargo, que el exceso de ortogonalidad también es malo. Podemos llegar a tener demasiadas operaciones, algunas de las cuales sean inútiles o difíciles de implementar. De hecho, algunas de ellas podrían incluso ser ilegales (en cuyo caso deberán recordarse como excepciones).

- Si “e” es el número de excepciones, entonces podemos decir que el principio de ortogonalidad es útil sólo si: m+n+e <mn-e

Ejemplos de instrucciones

- Formato de las instrucciones:

S	OP	OPN1	OPN2	DEST

Donde **S** es el *signo*, **OP** es la *operación*, **OPN1** y **OPN2** son los *operandos* y **DEST** es el *destino*.

Ejemplos de instrucciones

- Si suponemos ahora que **+1** significa **adición**:

+ 1 010 150 200

Esta instrucción agrega el contenido de la dirección **010** al contenido de la dirección **150** y el resultado se almacenará en la dirección **200**.

Ejemplos de instrucciones

	+	-
0	Mover	(No usado)
1	+	-
2	x	÷
3	Elevar al cuadrado	Raíz cuadrada
4	if = goto	if ≠goto
5	if ≥ goto	if < goto
6	(y) ⟶ z	x ⟶ y(z)
7	Incrementar y evaluar	(No usado)
8	Leer	Imprimir
9	Detenerse	(No usado)

Ejemplos de instrucciones

- Un ejemplo de un salto condicional:

+ 4 200 201 035

Significa: si el contenido de la dirección 200 es igual al contenido de la dirección 201, entonces saltar a la dirección 035.

Ejemplos de instrucciones

- Un ejemplo de índices. Supongamos que hay un arreglo de 100 elementos que comienza en la dirección 250. Supongamos también que la dirección 050 contiene 17:

+6 250 050 803

Mueve el contenido de la dirección 267 (250+17) a la dirección 803.

Ejemplos de instrucciones

- Otro ejemplo:

-6 722 250 050

Mueve el contenido de la dirección 722 a la dirección 267.