Lenguajes de Programación

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

Los Bloques

- Como en Pascal, el procedimiento es el único constructor que define entornos, éste es el único constructor que agrega o elimina registros de activación a la pila.

- Otros lenguajes de programación como Algol y Ada, tiene otro constructor que define entornos: el bloque.

Los Bloques

- Puesto que los registros de activación representan contextos y los bloques definen contextos, debe resultar obvio que los bloques también necesitan registros de activación.

- Estos registros de activación tendrán que ser creados cuando el bloque es *activado* (o sea, cuando se entra a él) y destruído cuando el bloque es *desactivado* (o sea, cuando salimos de él).

Los Bloques

- ¿Cómo se implementa la entrada-salida de un bloque?
- Podemos resolver el problema de la implementación de los bloques reduciéndolo a otro problema que ya hemos resuelto anteriormente: la implementación de procedimientos.
- Existe una clara similitud entre ambos: cuando se entra a un procedimiento, se crea un registro de activación, igual que como se hace con un bloque. Cuando salimos de un procedimiento, su registro de activación es destruído, igual que como se hace con un bloque.

Los Bloques

- Veremos en clase un segmento de código en Algol (usando bloques) que se traducirá en un programa en Pascal.

- Se hace notar que cada bloque se ha convertido en un procedimiento que se invoca exactamente en un lugar (el sitio de anidamiento del bloque correspondiente).

- Una implementación correcta de los bloques consistiría en traducirlos a procedimientos definidos de esta forma. Esto, sin embargo, es un tanto ineficiente, y puede mejorarse, como veremos más adelante.

Los Bloques

- Derivaremos los pasos para entrar a un bloque a partir de los pasos que vimos antes para invocar un procedimiento.

- Las diferencias cruciales estriban en el hecho de que un bloque corresponde a un procedimiento que es invocado desde exactamente un lugar en el programa.

- Otra diferencia es que los bloques no admiten parámetros.

Los Bloques

- El primer paso para entrar a un bloque es establecer la liga dinámica (esto es siguiendo el método de la cadena estática, puesto que en este caso es más simple que el de los displays):

M[SP].DL := EP;

- El siguiente paso en la invocación de un procedimiento es encontrar el ambiente de ejecución del procedimiento y usarlo para inicializar la liga estática.

Los Bloques

- El ambiente de ejecución en Algol es siempre el ambiente de definición, y el ambiente de definición para un bloque es siempre el bloque inmediatamente circundante. Esto es, la liga estática de un bloque siempre apunta al bloque inmediatamente circundante, el cual está contenido en el registro del EP (apuntador al ambiente). Por tanto, la liga estática del nuevo bloque se establece usando:

```
M[SP].SL := EP;
```

- Se hace notar que tanto la liga estática como la dinámica son la misma en este caso.

Los Bloques

- Los siguientes pasos en la invocación a un procedimiento instalan el nuevo registro de activación y asignan su espacio en la pila. Esto también se requiere para un bloque:

  EP := SP;

  SP := SP + tamaño(RA);

- El paso final de una invocación, que es saltar al inicio del procedimiento, no se requiere para un bloque puesto que la primera instrucción de un bloque aparece inmediatamente después del código para entrar a éste (o sea, está después del **begin**).

Los Bloques

- La salida de un bloque es también muy similar a la salida de un procedimiento. En este caso, omitimos el salto al punto donde continuaremos la ejecución tras la invocación del procedimiento, puesto que la instrucción a ejecutarse en este caso será la que esté justo después del **end**.

- ¿Cómo podemos mejorar este código? Dado que las ligas estática y dinámica son siempre las mismas para un bloque, no hay razón para tener ambas en el registro de activación. Por tanto, eliminaremos la liga dinámica.

Los Bloques

- El resultado es que el registro de activación de un bloque tiene una estructura muy simple:

- **LV** variables locales
- **IP** dirección de continuación
- **SL** liga estática

- Se requiere una dirección de continuación (IP) porque un bloque puede invocar a un procedimiento. Por tanto, necesitamos un lugar para almacenar el estado de la ejecución.

Los Bloques

- En clase se mostrará el código para entrar y salir de un bloque. Se hace notar que este código requieren sólo dos referencias de memoria para entrar y salir de un bloque.

- ¿Por qué los bloques requieren sólo una liga, pero los procedimientos requieren dos? La *liga estática* nos lleva al contexto estáticamente anterior y la *liga dinámica* nos lleva al contexto dinámicamente anterior.

Los Bloques

- Para un bloque, los contextos estáticos y dinámicos anteriores son siempre el mismo, por lo cual se requiere sólo una liga.

- Dicho de otra manera, un procedimiento puede ser activado en un contexto diferente de aquel donde fue definido. Por lo tanto, distinguiremos entre el *ambiente de definición* y el *ambiente de invocación*.

- Esta potencial activación remota de un procedimiento es posible debido al hecho de que un procedimiento tiene un nombre y, por tanto, puede activarse en cualquier parte donde su nombre sea visible (o cuando se pasa como parámetro).

Los Bloques

- En contraste, un bloque no tiene nombre; es anónimo. Como resultado de esto, un bloque puede ser activado en sólo un contexto (aquel en el cual está textualmente anidado).

- Por lo tanto, para un bloque, el ambiente de definición y el ambiente de activación son el mismo (el bloque inmediatamente circundante).

- De hecho, para ser preciso, la liga al registro de activación de un bloque no debiera llamarse ni estática ni dinámica, porque, de hecho, es ambas a la vez.

Los Bloques

- La entrada y salida de un bloque es también muy simple usando el método de los displays.

- Primero consideremos la entrada a un bloque. En clase veremos una figura que ilustra cómo lucen los displays y la pila antes y después de entrar a un bloque, al nivel *snl*.

- En esa figura podemos ver que el espacio para el registro de activación del bloque ha sido asignado en la parte superior de la pila y que la entrada al display D[*snl*] debe almacenarse en el registro de activación del bloque porque puede contener un apuntador a un registro de activación válido.

Los Bloques

- Por lo tanto, el código para entrar a un bloque es el siguiente:

M[SP].EP := D[*snl*];

D[*snl*] := SP;

SP := SP + tamaño(locales);

Esto requiere tres referencias de memoria (para la actualización del display).

Los Bloques

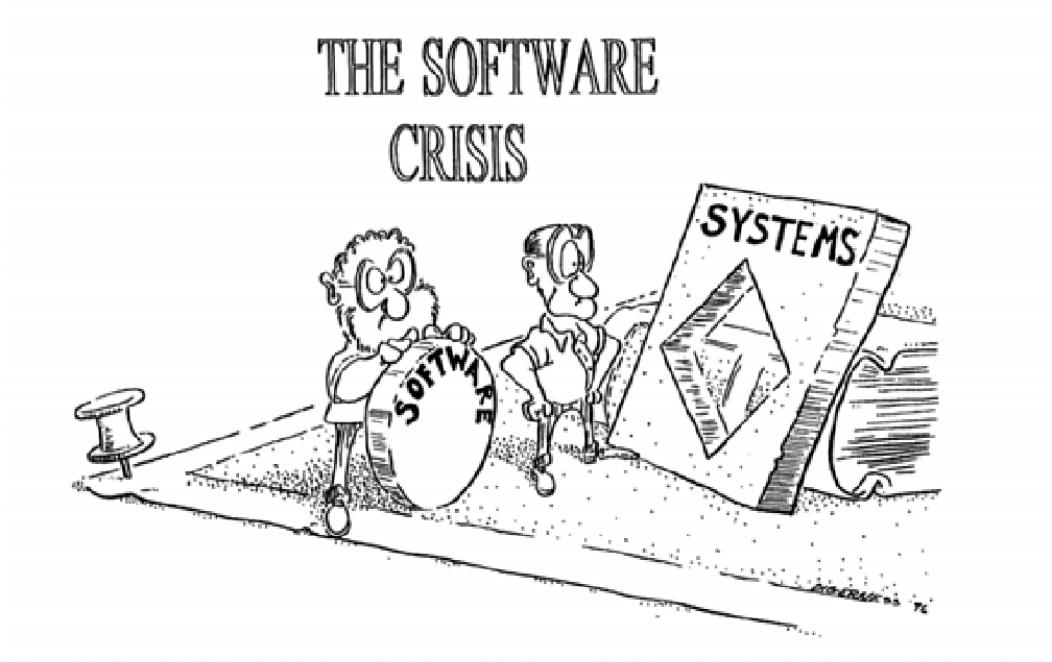
- La salida de un bloque es todavía más simple: todo lo que se requiere es liberar el registro de activación de un bloque y restaurar el display:

SP := SP – tamaño(locales);

D[snl] := M[SP].EP;

Se requieren dos referencias de memoria para restaurar el display.

Ada

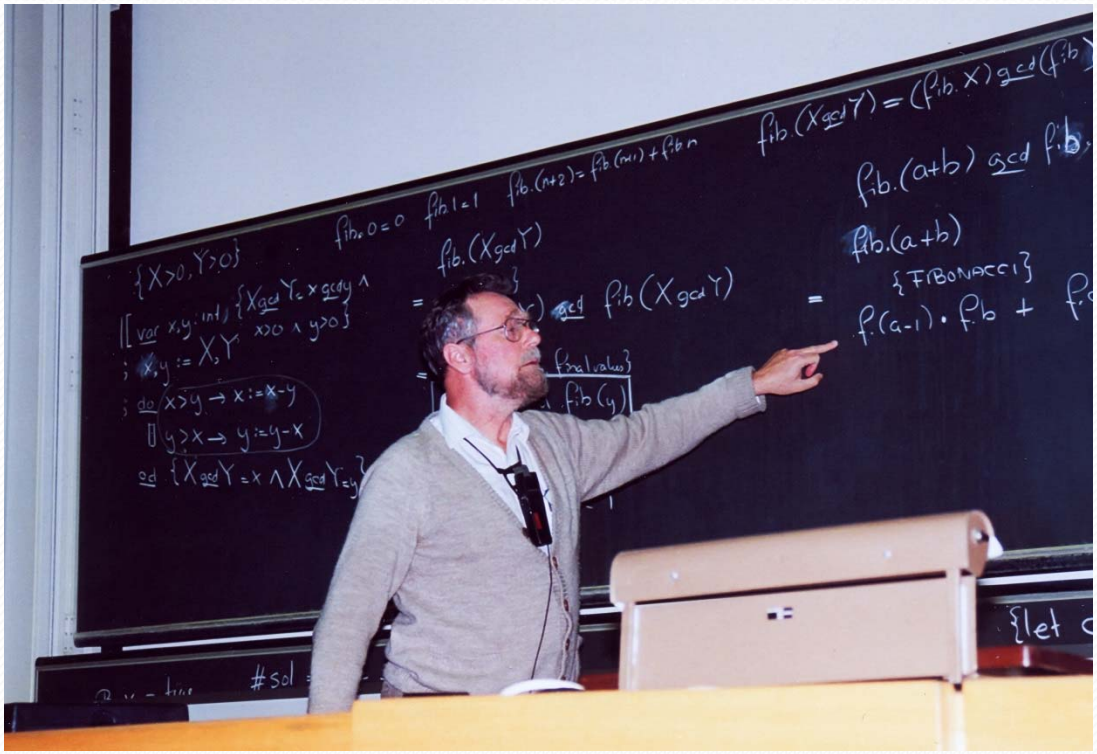


- En los 1970s, los expertos en computación reconocieron que los costos de producción de software estaban incrementándose de manera exponencial (la llamada "**crisis del software**") y que, por lo tanto, era vital encontrar una solución rápida a este problema.

Ada

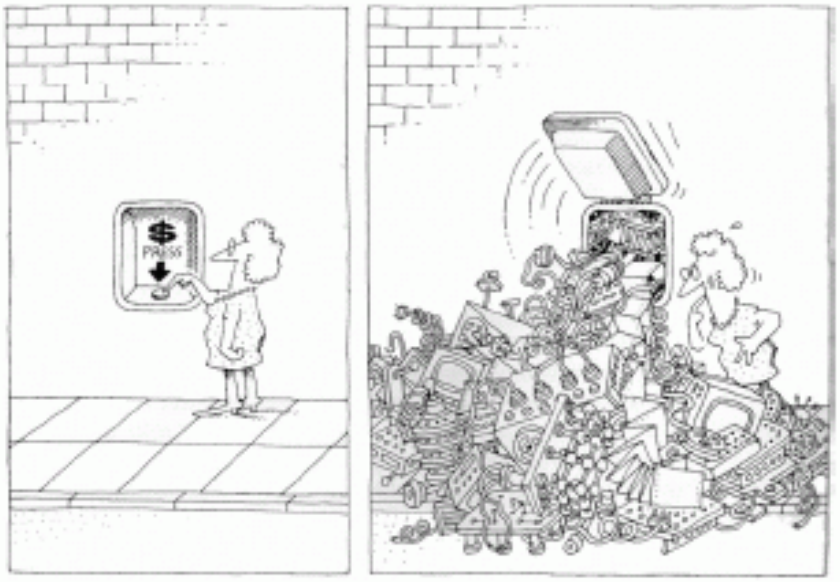
- Este problema motivó la creación de la **ingeniería de software** como una disciplina independiente dentro de las Ciencias de la Computación en la cual la gente estudiaría cómo diseñar software de grandes dimensiones y los diferentes problemas asociados con cada una de las etapas de la producción de software.

Ada



- Edsger Dijkstra y algunos otros investigadores se percataron de que el costo de producir un programa parecía incrementarse con el cuadrado de la longitud del mismo.

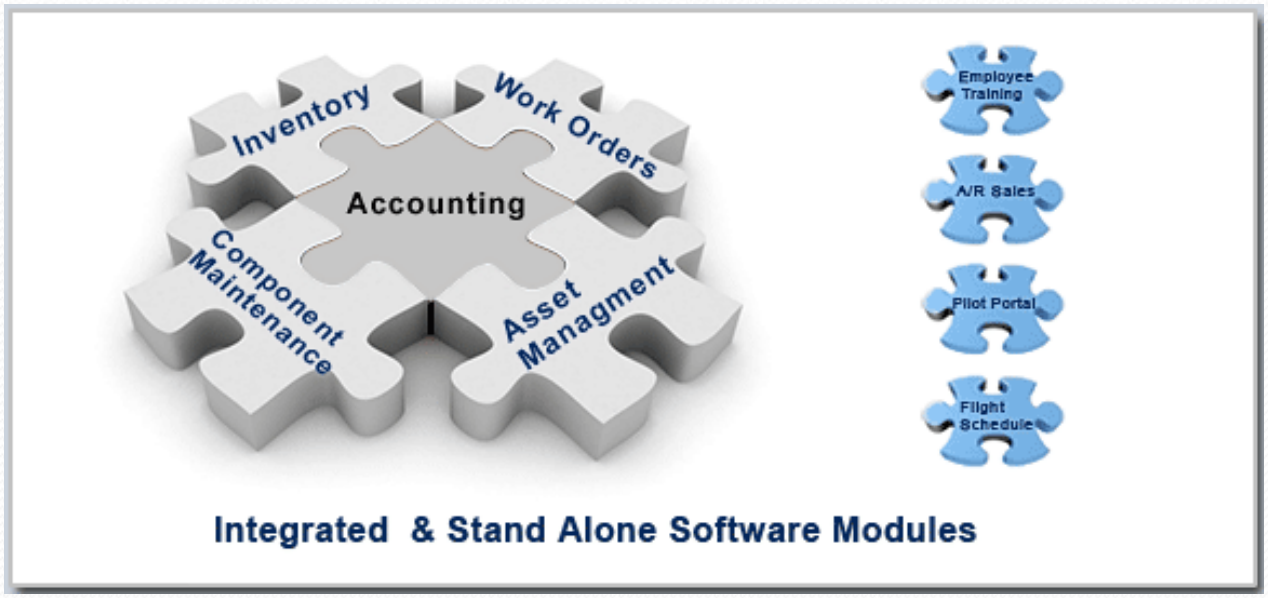
Ada



The task of the software development team
is to engineer the illusion of simplicity.

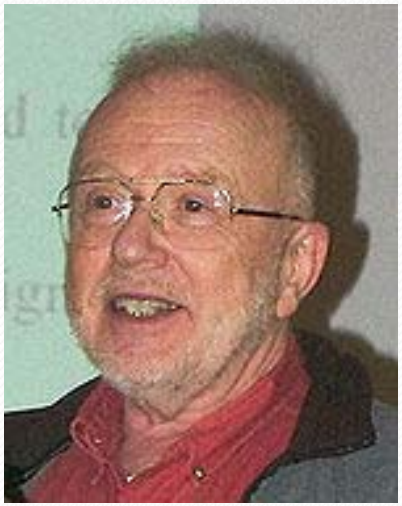
- Esto hacía infactible la creación de piezas de software de gran tamaño.
- El objetivo de la ingeniería de software sería, por tanto, bajar este costo a una función lineal.

Ada



- Uno de los métodos más tradicionales desarrollados para controlar la complejidad de un programa era la "modularización", que consistía en dividir el programa en un cierto número de módulos independientes que pudiesen ser modificados y actualizados sin afectar el resto del programa.

Ada



- David L. Parnas estableció en 1971 y 1972, algunos de los principios más importantes respecto al diseño modular, indicando, por ejemplo, que cada módulo debía representar una decisión difícil en el programa.

Ada

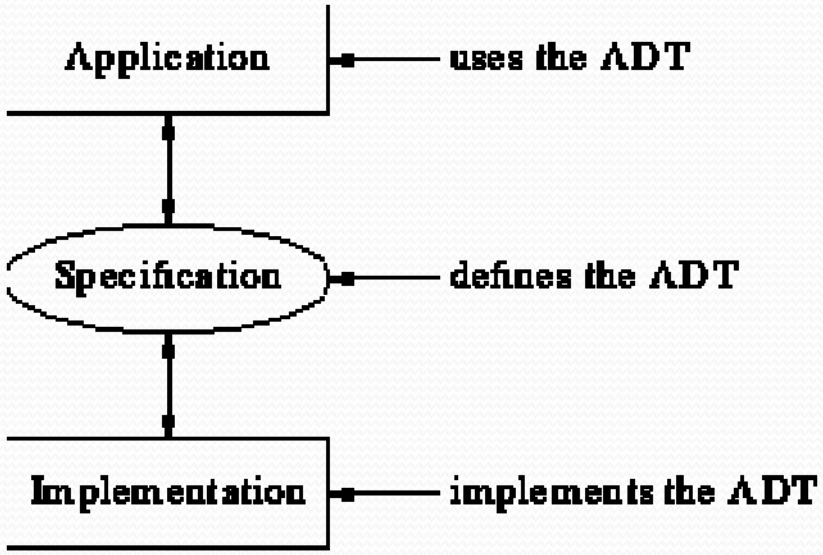
- Esto significa que los resultados de cada decisión pueden ocultarse en el módulo correspondiente.
- Si después esa decisión cambiaba, entonces sólo ese módulo debía ser modificado.
- A esto se le llama *ocultamiento de información*.

Ada

- Los **Tipos de Datos Abstractos** jugaron otro papel importante en la tarea de diseñar software a gran escala.

- Un tipo de datos abstracto (ADT por sus siglas en inglés) representa una abstracción de una estructura usada en un programa, en la cual las operaciones han sido separadas de los datos.

Ada



- El uso de los ADTs proporciona una gran flexibilidad en un programa y nos conduce a pensar más en términos del algoritmo (p.ej., las operaciones de una pila) que de los detalles de la implementación (p.ej., el uso de un arreglo para implementar una pila).

Ada

- En 1973, se desarrollaron varios lenguajes de programación inspirados en la noción de ADT y de módulo.
- Por ejemplo: Alphard, CLU, Mesa, Euclid, MODULA y Tartan.

Ada

- Muchos de estos lenguajes fueron influenciados por la idea de **clase** propuesta originalmente en el Simula 67.

- Esta experiencia fue adoptada más tarde en el desarrollo de lenguajes de producción de ADTs tales como Ada.

Ada

- A mediados de los 1970s, el Departamento de Defensa de los Estados Unidos (DoD, por sus siglas en inglés) identificó la necesidad de contar con un lenguaje de programación que constituyera el estado del arte y que se usaría para todos los servicios militares en aplicaciones de cómputo “embebidas” (*embedded*), las cuales son críticas para una misión.

Ada

- Estas son misiones en las cuales una computadora se inserta y se integra dentro de un sistema más grande.

- Por ejemplo, un sistema de misiles o un control de comandos y un sistema de comunicación.

Ada

- El DoD estaba gastando alrededor de $3,000 millones de dólares al año en software, la mayor parte del cual era para sistemas “embebidos”.

Ada

- Una de las razones principales para estos altos costos era la falta de un lenguaje de programación estándar dentro del DoD, lo cual limitaba fuertemente la portabilidad y el reuso de software, puesto que habían más de 400 lenguajes y dialectos de programación en uso dentro del DoD en aquella época.

Ada

- Reconociendo que sus necesidades no serían satisfechas por ese alto número de lenguajes de programación y en un intento por ahorrarse entre $12 mil y $24 mil millones de dólares en los 17 años siguientes, el DoD estableció un comité a cargo de estudiar el desarrollo de un solo lenguaje de programación para todas sus aplicaciones.

Ada

- De 1975 a 1979, este comité (llamado "*Higher-Order Language Working Group*", o simplemente HOLWG) publicó una serie de especificaciones que debía cumplir el nuevo lenguaje, dando en cada ocasión, mayor detalle de lo que querían.

Ada

- Los nombres de estas especificaciones reflejaban, de alguna manera, su nivel de detalle:

- 1975 Strawman
- 1975 Woodenman
- 1976 Tinman
- 1978 Ironman
- 1979 Steelman

Ada

- La legibilidad y la simplicidad fueron parte de las especificaciones generales, pero también se incluyó una ayuda para desarrollo de módulos y mecanismos que permitieran efectuar programación concurrente.

- Asimismo, se planteó un diseño que facilitara la verificación de los programas.

Ada

- En 1977, HOLWG estudió 26 lenguajes de programación en uso (ignorando C, por razones que se desconocen hasta la fecha), y concluyeron que ninguno de ellos cumplía con las especificaciones establecidas.

- De tal forma, se organizó una competencia internacional para diseñar un nuevo lenguaje de programación.

Ada

- El concurso duró de 1977 a 1979 y dio como resultado 16 propuestas originales, las cuales se redujeron a 4, luego a 2 y finalmente a una.

- El lenguaje ganador fue diseñado por un equipo de *CII-Honeywell-Bull*, lidereado por **Jean Ichbiach** (1940-2007)

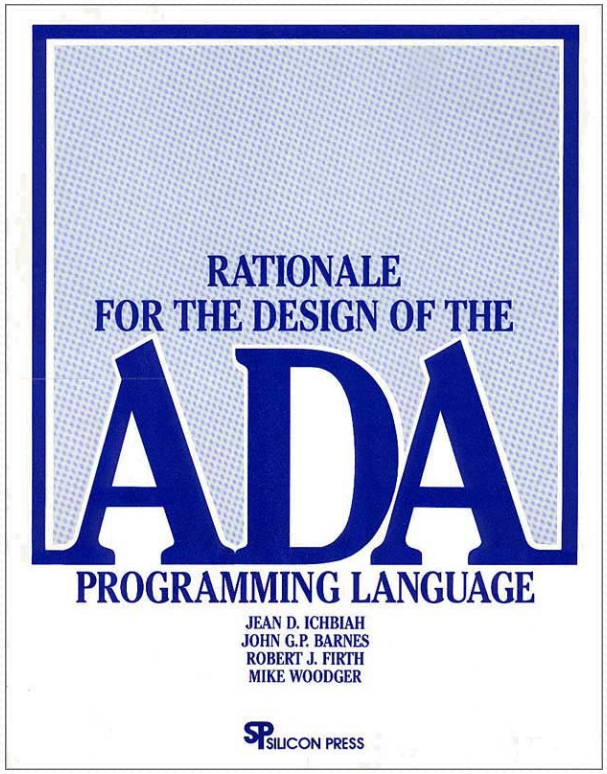
Ada

- Esta propuesta estuvo influenciada por el lenguaje de programación LIS (*Language d'Implementation de Systèmes*) que Ichbiach y su grupo desarrollaron en los 1970s.

Ada

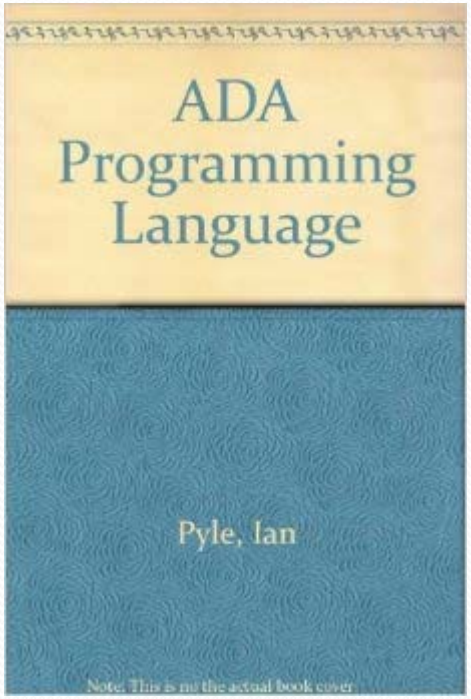
- En mayo de 1979, el HOLWG renombró este lenguaje como “Ada” en honor de Augusta Ada, condesa de Lovelace (1815-1852), quien es considerada la primera programadora de la historia.

Ada



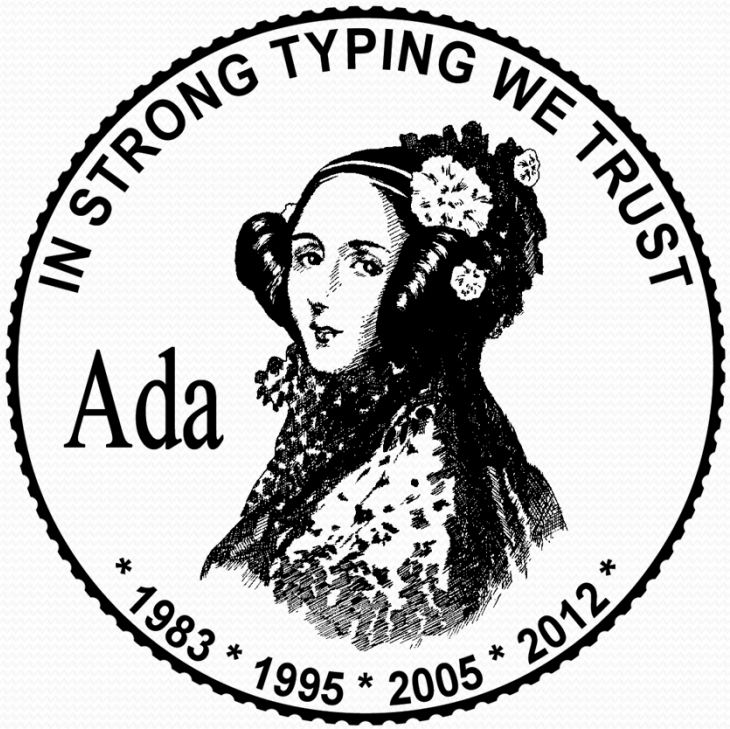
- En respuesta a los más de 7,000 comentarios y sugerencias de los expertos en diseño de lenguajes de más de 15 países, Ada fue revisado y alcanzó su forma final en septiembre de 1980.

Ada



- Ada se volvió un estándar militar de los Estados Unidos en enero de 1983, y se volvió obligatorio para todo el software de misiones críticas (embebido) en julio de 1984.

Ada



- Con el objetivo en mente de preservar la portabilidad del lenguaje, el DoD emprendió la acción sin precedentes de registrar el nombre “Ada” como una marca propia.

Ada

- Esto les dio el apoyo legal necesario para controlar la evolución del lenguaje, ya que el DoD sólo autoriza el uso del nombre “Ada” para un lenguaje de programación que siga por completo el estándar definido por ellos.

Ada

- El DoD no permite el desarrollo de subconjuntos ni superconjuntos de Ada.
- Para validar que un compilador de Ada sigue el estándar del DoD, se diseñaron más de 2,500 pruebas que intentan asegurar que el compilador implementa exactamente el lenguaje.
- Esto ha creado una fuerte controversia dentro de la comunidad de lenguajes de programación.

Ada

- El manual de referencia preliminar de Ada fue publicado en la revista *ACM SIGPLAN Notices* en junio de 1979.

- El manual de referencia estándar para los militares fue aprobado el 10 de diciembre de 1980 (justo la fecha de cumpleaños de Ada Lovelace), y se le otorgó el número MIL-STD-1815 en honor del año de nacimiento de Ada Lovelace.

Ada

- Debido a sus funciones de soporte de actividades donde la seguridad es crítica, Ada se usa actualmente no sólo para aplicaciones militares, sino también para proyectos comerciales en los cuales los errores tendrían consecuencias muy graves (p.ej., en diseño aeronáutico, en control de tráfico aéreo, en sistemas ferroviarios y en aplicaciones bancarias.

- Por ejemplo, el sistema de manejo de información aérea y el sistema fly-by-wire usados en el Boeing 777, fueron escritos en Ada.

Ada

- El sistema canadiense de control de tráfico aéreo, que cuenta con 1 millón de líneas de código, fue escrito en Ada.
- Ada se ha usado también para escribir los sistemas de control de varios trenes de alta velocidad y del metro de París, Londres, Hong Kong y Nueva York.

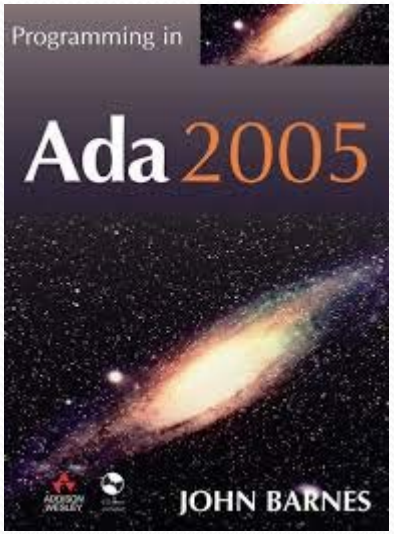
Ada

- El lenguaje se volvió un estándar de ANSI en 1983 (ANSI/MIL-STD 1815A), y sin ningún otro cambio posterior, se volvió un estándar de ISO en 1987 (ISO-8652:1987). A esta versión del lenguaje se le conoce como Ada 83 (su fecha de adopción por ANSI), aunque algunos autores le llaman también Ada 87 (la fecha de su adopción por ISO).

- Ada 95, que es un estándar tanto ISO como ANSI, se publicó en febrero de 1995. Este fue el primer lenguaje de programación orientado a objetos que se estandarizó.

- La versión más reciente del lenguaje es Ada 2012, que fue anunciada en un congreso europeo realizado en Estocolmo en diciembre de 2012.

Organización Estructural



- La sintaxis de Ada es similar a la de Pascal, con la diferencia de que en el primero las palabras clave tienen que escribirse en minúsculas y todas las demás palabras en mayúsculas.

Organización Estructural

- Los constructores de Ada pueden dividirse en cuatro categorías:

  1) Declaraciones

  2) Expresiones

  3) Sentencias

  4) Tipos

Organización Estructural

- Las expresiones y las sentencias son muy similares a las de Pascal.
- Los tipos también son similares, excepto por el hecho de que son más flexibles y algunos de los problemas con el sistema de tipos de Pascal han sido corregidos.

Organización Estructural

- Sin embargo, las diferencias más significativas se presentan en las declaraciones. Las declaraciones en Ada pueden clasificarse en:

  1) Objeto
  2) Tipo
  3) Subprograma
  4) Paquete
  5) Tarea

Organización Estructural

- Las declaraciones de **objetos** son similares a las declaraciones de constantes y variables en Pascal.
- Las declaraciones de **subprogramas** son similares a las declaraciones de funciones y procedimientos en Pascal.

Organización Estructural

- Sin embargo, en el caso de Ada, las declaraciones de subprogramas permiten usar como nombres operadores internos del lenguaje (+, -, =>, etc.).

- Esto permite sobrecargar los operadores internos del lenguaje con nuevos tipos de datos definidos por el usuario.

Organización Estructural

- Por ejemplo, '+' normalmente se aplica tanto a enteros como reales.
- En Ada, este operador se puede sobrecargar, de tal manera que actúe sobre tipos definidos por el usuario (p.ej., números complejos y matrices).

Organización Estructural

- Dos de las declaraciones más importantes de Ada son los **paquetes** y las **tareas**.
- Ambas declaran módulos, pero se distinguen por el hecho de que las tareas tienen la capacidad de ejecutarse en paralelo.
- Los módulos son los componentes básicos con que se construyen los programas en Ada.

Organización Estructural

- Cada módulo forma un ambiente disjunto que se comunica con otros módulos a través de interfaces bien definidas.

- La declaración de un módulo se divide en dos partes: una **especificación** que describe la interfaz a ese módulo, y un **cuerpo** o **definición**, que describe la forma en la que el módulo es implementado.

- La especificación de un paquete contiene las especificaciones de aquellas cosas (p.ej., procedimientos, tipos, etc.) que suministra el paquete.

Organización Estructural

- El cuerpo del paquete contiene los cuerpos o definiciones de estas cosas.
- El propósito de este tipo de estructura es implementar los principios de ocultamiento de información previamente estudiados.

Organización Estructural

- Ada está diseñado para permitir la implementación de un compilador convencional.

- Típicamente, un compilador de Ada se dividiría en 4 subsistemas:

  1) Analizador sintáctico
  2) Analizador semántico
  3) Optimizador
  4) Generador de código

Organización Estructural

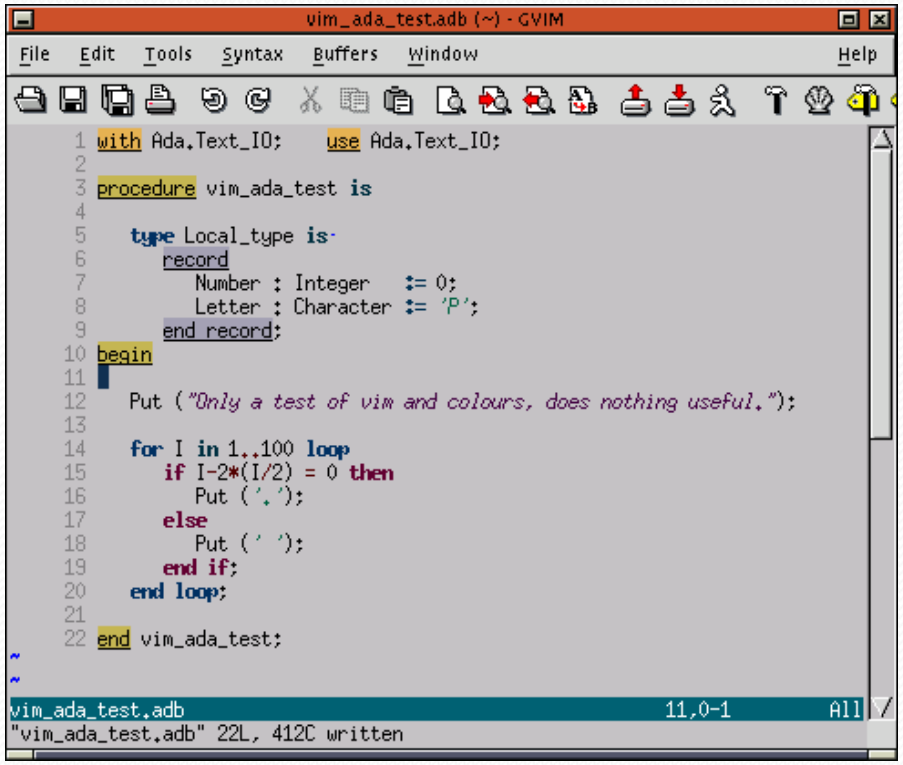
- El “analizador sintáctico” (*parser*) es convencional y sólo moderadamente más complicado que un analizador para Pascal.
- Algunos sistemas interactivos de Ada reemplazan el analizador sintáctico por un editor dirigido por sintaxis, el cual genera directamente el árbol de evaluación a ser usado por el analizador semántico.

Organización Estructural

- El “analizador semántico” efectúa chequeo de tipos, como en Pascal, y procesa algunas de las funciones más complejas de Ada, tales como las declaraciones genéricas y los operadores sobrecargados.

- La complejidad de estas funciones es lo que hace que el analizador semántico de Ada sea mucho más grande y complejo que el de Pascal.

Organización Estructural



- El resultado del analizador semántico es el árbol de un programa que se pasa a un optimizador convencional y a un generador de código.

Estructuras de Diseño y Sistema de Tipos

- Los tipos de Ada son esencialmente como los de Pascal, incluyendo la capacidad de incluir una restricción de rango (*range constraint*) para limitar el conjunto de valores permisibles. Por ejemplo:

type NUMERO **is range** -30..50;

Estructuras de Diseño y Sistema de Tipos

- La aritmética con enteros es exacta y esencialmente la misma que en FORTRAN, Algol-60 y Pascal.
- Ada va mucho más allá de la simplicidad del tipo real de Pascal, al proporcionar tipos de punto flotante (*floating point*) y de punto fijo (*fixed point*).

Estructuras de Diseño y Sistema de Tipos

- La declaración:

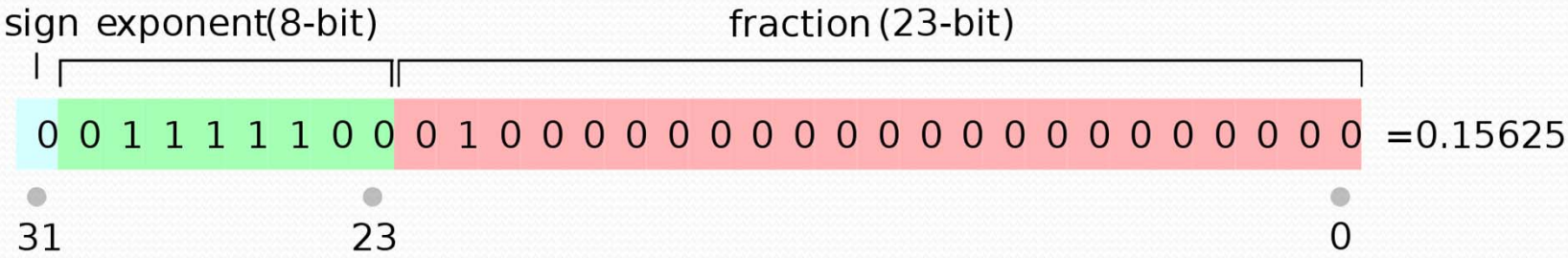
type VALOR **is digits** 10 **range** -1.0e10..2.0e10;

- define a VALOR como un tipo de punto flotante con al menos 10 dígitos de precisión y capaz de acomodar números en el rango especificado.

Estructuras de Diseño y Sistema de Tipos

- Si la computadora donde se está usando Ada proporciona aritmética con diferentes tipos de precisión (p.ej., simple precisión, doble precisión, cuádruple precisión, etc.), el compilador deberá decidir cuál usar de acuerdo a lo que el programador esté pidiendo.

Estructuras de Diseño y Sistema de Tipos



- Ada también especifica que toda implementación debe proporcionar un tipo predefinido FLOAT, que corresponda con la precisión usual de la computadora donde corra el compilador.

Estructuras de Diseño y Sistema de Tipos

- También pueden pre-definirse los tipos SHORT_FLOAT y LONG_FLOAT en caso de que los soporte la implementación, aunque su uso limita la portabilidad de los programas y contradice la meta del HOLWG de no tener ningún dialecto de Ada.

Estructuras de Diseño y Sistema de Tipos

- Claro que se motiva a los programadores a usar las definiciones de punto flotante como la antes mostrada (usando **digits**), en vez de los tipos pre-definidos, a fin de que sus programas sean más independientes de la máquina donde corren.

Estructuras de Diseño y Sistema de Tipos

- Al agregar la especificación “digits” antes mencionada, los programadores especifican la precisión que *quieren* y dejan al compilador la determinación de la representación interna que *necesitan*.

- Esto es imposible si se usa FLOAT, LONG_FLOAT y cualquier otro tipo que sea dependiente de la computadora.

Estructuras de Diseño y Sistema de Tipos

- El único problema de este esquema en la práctica es que los programadores no suelen saber qué precisión necesitan.

- Además, un número significativo de programadores suele escribir la especificación de precisión que saben les dará una cierta representación en una implementación en particular.

Estructuras de Diseño y Sistema de Tipos

- Esto ya había ocurrido antes en PL/I, en el cual muchos programadores usaban “BINARY FIXED(31)”, no porque quisieran números de esa precisión, sino porque con esta instrucción se representarían los números usando palabras de 32 bits en una IBM-360.

- Esto inutiliza por completo el propósito original de las especificaciones independientes de la máquina.

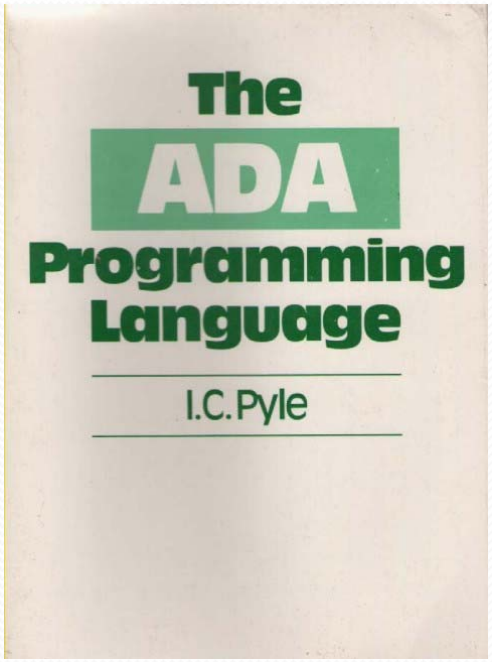
Estructuras de Diseño y Sistema de Tipos

- Las restricciones de punto flotante en Ada ilustran el **Principio de la Preservación de Información**, porque si el usuario sabe cuáles son sus requerimientos a un nivel más abstracto (el número de dígitos requerido), no debiera requerir definirlos a un nivel más concreto (el número de palabras requeridas).

Principio de la Preservación de la Información

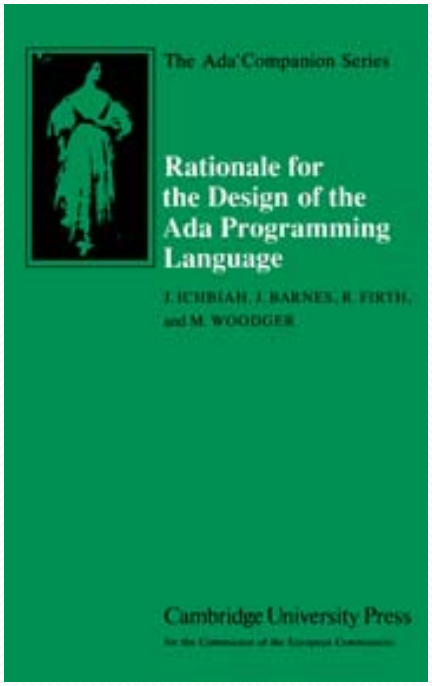
El lenguaje debiera permitir la representación de información que el usuario podría saber y que el compilador podría necesitar.

Estructuras de Diseño y Sistema de Tipos



- Esto evita que un programa incluya decisiones que son dependientes de la máquina y que mejor deben dejarse al compilador.

Estructuras de Diseño y Sistema de Tipos



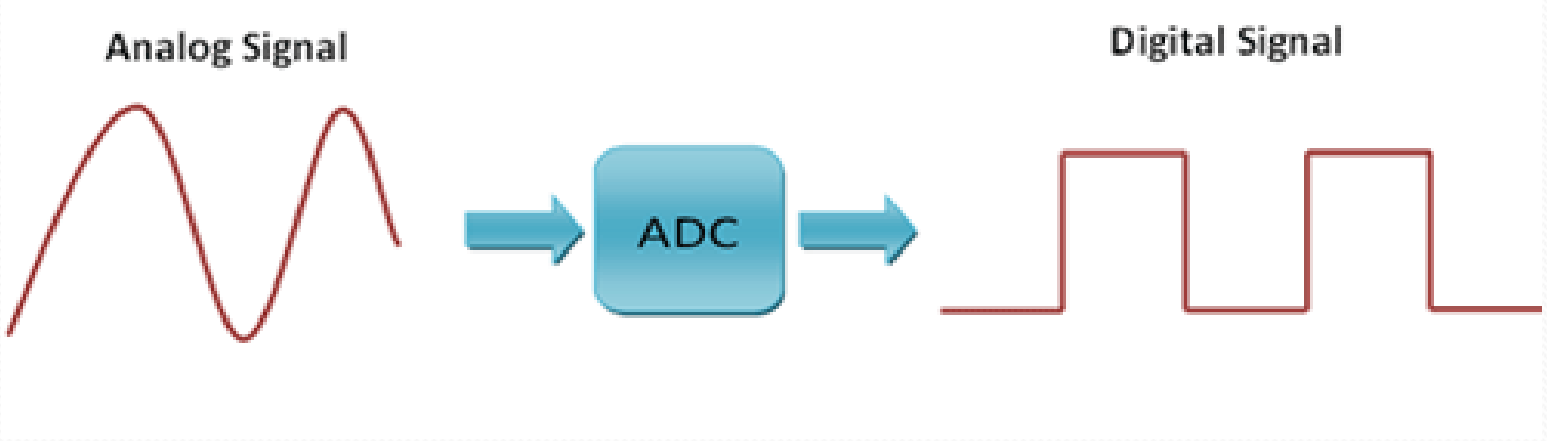
- La aritmética con los números de punto flotante es convencional, porque las operaciones realmente se realizan en la precisión máxima disponible y luego se redondean a la precisión de los operandos.

Estructuras de Diseño y Sistema de Tipos

- Además de los números de punto flotante, Ada proporciona también números de punto fijo, los cuales tienen un límite de error absoluto.

- Los números de punto fijo son comunes en los lenguajes comerciales (p.ej., COBOL), donde se requiere acotar el límite de error, a fin de evitar errores de redondeo (por ejemplo, en una nómina).

Estructuras de Diseño y Sistema de Tipos



- La razón por la que se incorporaron en Ada los números de punto fijo es porque son utilizados por muchos de los periféricos incorporados en los sistemas de cómputo embebidos (p.ej., convertidores de analógico a digital).

Estructuras de Diseño y Sistema de Tipos

- Los tipos de punto fijo se especifican usando definiciones como la siguiente:

type PRECIO **is delta** 0.01 **range** 0.00..1_000_000.00;

donde “delta” se refiere al límite de error absoluto deseado.

Estructuras de Diseño y Sistema de Tipos

- En este caso, los valores del tipo PRECIO serán múltiplos exactos de 0.01.

- Por ejemplo, el número 16.75 se almacenaría como el equivalente binario de 1675, puesto que 16.75 = 1675 × 0.01.

Estructuras de Diseño y Sistema de Tipos

- El número mínimo de bits requeridos para almacenar un tipo de punto fijo es simplemente el logaritmo del número de valores a ser representados. Por ejemplo:

$$\log_2[1+(1000000-0)/0.01] \approx \log_2 10^8 \approx 26.6$$

- Por lo tanto, se requieren 27 bits.

Estructuras de Diseño y Sistema de Tipos

- Convertir un valor entero a uno de punto fijo requiere dividir entre el valor de "delta".

- Por ejemplo: PRECIO(2.0) resultará en la representación binaria de 2/0.01 = 200. Si "delta" es una potencia de 2, entonces resulta trivial reemplazar esta operación por un desplazamiento (*shift*) a la izquierda.

Estructuras de Diseño y Sistema de Tipos

- Por ejemplo, consideremos la siguiente definición de VOLT:

type VOLT **is delta** 0.125 **range** 0.0..255.0;

Estructuras de Diseño y Sistema de Tipos

- En este caso, la conversión VOLT(20) se traduce como:

$$20 \div 1/8 = 20 \times 8 = 000010100 \times 2^3 = 010100000\ (160)$$

- y la operación puede realizarse desplazando tantas veces a la izquierda, como la potencia de 2 lo indique.

Estructuras de Diseño y Sistema de Tipos

- Es por ello que la definición de Ada permite al compilador escoger un valor de "delta" que sea menor que el especificado por el usuario, pero el cual sea una potencia de 2.

- Esto permite hacer más eficiente la conversión correspondiente.