CAPÍTULO I – Conceptos Preliminares

RAZONES PARA ESTUDIAR CONCEPTOS DE LP:

- Incrementar la capacidad para expresar ideas
- Mejorar la calidad y cantidad de conocimiento previo para la apropiada selección de lenguajes
- Aumentar la habilidad para aprender y diseñar nuevos lenguajes
- Mejor comprensión de la importancia de la implementación
- Inclusión de los avances en las formas de cómputo

Un **LENGUAJE,** en general, es el medio del que se vale cualquier individuo para comunicar ideas y experiencias a otros individuos. Estas ideas y experiencias están "almacenadas" de alguna manera en el individuo emisor, y "almacenadas" tal vez con otra representación en el receptor.

El **LENGUAJE DE PROGRAMACIÓN**, en general, tiene la misma idea, sólo que el receptor es una computadora. Por ello, es una de las principales herramientas en el proceso de desarrollo del SW. ~~Surge de la vinculación entre las diferentes metodologías de diseño y las arquitecturas de las computadoras~~. Otro de los factores que tienen especial relevancia es el *dominio de aplicación* pretendido:

- APLICACIONES CIENTÍFICAS: Gran cantidad de cómputos sobre números grandes en punto flotante. ED simples (arreglos y matrices), ciclos y selecciones. Buscan *EFICIENCIA.* Ej: FORTRAN, Algol60, Lenguajes imperativos.

- APLICACIONES DE NEGOCIOS: Gran producción de reportes elaborados. Requiere modos precisos de almacenar y describir números decimales y caracteres. Hojas de cálculo y BD. Requieren *PORTABILIDAD*. Ej: COBOL.

- APLICACIONES DE IA: Computación simbólica en vez de numérica. Manejo de listas. Requiere *FLEXIBILIDAD*. Ej: LISP (funcional), Prolog (lógico).

- PROGRAMACIÓN DE SISTEMAS (BD): Requiere EFICIENCIA por uso continuo. Debe tener ejecución veloz, brindar soporte de bajo nivel para manejar dispositivos. Ej: PL/S, C.

- LENGUAJES SCRIPTING: Una lista de comandos (script) se guarda en un archivo para ser ejecutado. Inicialmente *Shell*, luego añadido variables, sentencias de flujo de control, etc. Ej: Perl, Javascript.

- LENGUAJES DE PROPÓSITO ESPECIAL: Reportes de negocios, herramientas de máquinas programables, simulación de sistemas, etc. Ej: 2PG, APT, GPSS.

Una de las **CLASIFICACIONES** para lenguajes de programación sigue este esquema:

1. PRIMERA GENERACIÓN (1951-1957): Esencialmente lenguajes máquina

2. SEGUNDA GENERACIÓN (1958-1963): Lenguajes de alto nivel, no estructurados.
   Ej: Lengauje Ensamblador.

3. TERCERA GENERACIÓN (1964-1969): Lenguajes que incorporan la mayoría de las habilidades procedurales actuales. Ej: Pascal, Modula2, Ada.

4. CUARTA GENERACIÓN (1970-1990): Lenguajes con las siguientes características:
   - Estructura y programación de BD
   - Diccionario de datos centralizado con información sobre las componentes del sistema
   - Programación visual
   - Capacidad de definir interfaces para el usuario
   - Entorno de programación multifuncional, integrado e interactivo.

   Ej: SQL (Structured Query Language)

5. QUINTA GENERACIÓN (1990-Actualidad): Lenguajes de muy alto nivel. Basados en lógica y matemática.
   Ej: ML, Miranda.

CARACTERÍSTICAS DESEABLES EN EL SW: De cualquier SW se espera confiabilidad, extensibilidad, reusabilidad, robustez, eficiencia, etc.
Los LP también son SW pero al tener otro tipo de propósito se espera que posean otras características: {Criterios de evaluación de LP y las características que los afectan}

- FACILIDAD DE LECTURA (Readability): La facilidad para leer e interpretar programas es fundamental. Características que contribuyen/afectan a la legibilidad:

  - **Simplicidad:** Se obtiene en gran medida de combinar un número pequeño de constructores primitivos y un uso limitado (ni mucho ni poco) del concepto de *ortogonalidad*. Cuestiones que complican la legibilidad: cuando una operación se puede hacer de más de una forma (cont++, ++cont en C); sobrecarga de operadores definidos por el usuario; simplicidad excesiva (Assembler).

  - **Ortogonalidad:** Básicamente significa que un conjunto pequeño de constructores primitivos puede ser combinado en un número relativamente pequeño de formas para construir estructuras de control y datos. Cada combinación es legal y con sentido.
    La *falta de ortogonalidad* lleva a excepciones en las reglas del lenguaje. Cuantas menos excepciones, más regular y más fácil de aprender, leer y entender es el lenguaje. El *exceso de ortogonalidad* también trae problemas, pudiendo construir cosas muy complejas o posibilidades exponenciales de combinaciones.
    La ortogonalidad está muy relacionada con la simplicidad: cuanto más ortogonal es el diseño de un lenguaje, requiere menos excepciones en las reglas del mismo.
    Ej: Falta de ortogonalidad en C- las fcs pueden devolver *struct* pero no *array*.

- **Sentencias de Control:** Un programa que puede ser leído de arriba hacia abajo es más fácil de entender que un programa que requiere saltar de una sentencia a otra no adyacente. No hacer uso indiscriminado del GOTO/JUMP. Contar con estructuras legibles.

- **Tipos de Datos y Estructuras:** Es importante que se brinden las herramientas/facilidades necesarias para definir tipos y ED.
  Ej: uso de 0/1 en C para indicar flags (poco claro); Falta de registros en FORTRAN (necesario usar arrays).

- **Consideraciones sobre la Sintaxis:** La sintaxis de los elementos de un lenguaje afecta la facilidad de lectura:
  - *Forma de los identificadores:* identificadores demasiado cortos es malo.
  - *Palabras reservadas:* si se permite su uso para nombres de variables, etc. Se complica la lectura. Ej: el uso de begin/end y {/} complica seguir los bloques de programas.

- FACILIDAD DE ESCRITURA (Writability): Es una medida acerca de cuán fácil es crear programas con el lenguaje para un dominio de aplicación determinado. La mayoría de las cosas que afectan a la lectura también afectan la escritura. Este criterio depende del dominio de aplicación del lenguaje.
  Factores que afectan la escritura:

  - **Simplicidad y Ortogonalidad:** Si un lenguaje tiene demasiados constructores (no es simple) es difícil familiarizarse con todos ellos, provocando el desuso de algunos. Por esto, un menor número de primitivas y constructores y un conjunto de reglas consistente para combinarlos (ortogonalidad) es mucho mejor que tener gran cantidad de primitivas.
    Demasiada ortogonalidad puede complicar la detección de errores (combinaciones que son válidas y se escribieron por error). [Ver Hoja 10!]

  - **Soporte para la Abstracción:** Capacidad de definir y usar estructuras en operaciones complicadas, de manera que sea posible ignorar muchos de los detalles. Ej: TDA, procedimientos, etc.

  - **Expresividad:** El lenguaje posee formas relativamente convenientes de expresar ciertas operaciones. Ej: cont++ en C en lugar de cont=cont+1; poder usar *for* en vez de *while*.

- CONFIABILIDAD (Reliability): Un programa se dice confiable si hace lo especificado bajo toda condición. Cuestiones que afectan a la confiabilidad de programas en un determinado lenguaje:

  - **Chequeo de Tipos:** Testeo por errores de tipos en un programa, durante compilación o durante ejecución. El chequeo en ejecución es más costoso → preferible en compilación (cuanto antes se encuentren errores menos costoso resulta realizar los arreglos). Ej: antes en C no se chequeaban parámetros *int* (actual) con *float* (formal), lo cual podía producir errores. Ahora se chequean los tipos de todo parámetro.

  - **Manejo de Excepciones:** La habilidad para interceptar errores en tiempo de ejecución, tomar medidas correctivas, y continuar. Ej: Java try/catch; C no tiene.

- **Aliasing:** Tener 2 o más formas de referenciar a la misma celda de memoria (dentro del mismo ambiente de referenciamiento). Es malo y peligroso para la confiabilidad.

- **Readability/Writability:** Cuanto más fácil es un programa de leer y escribir, más fácil es de corregir. Programas difíciles de leer son difíciles de modificar.

- COSTO: El costo total de un LP depende de muchas de sus características:

  - **Aprender y Usar:** Costo de entrenamiento a los programadores. Que aprendan a escribir programas en el lenguaje, cercanos a aplicaciones particulares.

  - **Compilar:** Costo de compilación (si no hay compiladores).

  - **Ejecutar:** El costo de ejecución de un programa es altamente influenciado por el diseño del lenguaje. Un lenguaje que requiere mucho chequeo de tipos en ejecución dificulta la ejecución rápida de código, sin importar la calidad del compilador.

  - **Sistema de Implementación del Lenguaje:** Un lenguaje cuyo sistema de implementación (compiladores/intérpretes) es costoso o corre sobre HW muy costoso tendrá menor posibilidad de ser globalmente usado.

  - **Confiabilidad:** Escasa confiabilidad conlleva altos costos. Si el HW falla en un sistema crítico el costo será muy elevado.

  - **Mantenimiento de Programas:** Corrección y modificaciones para agregar nuevas funcionalidades. El costo de mantenimiento de SW depende de gran cantidad de características del lenguaje, pero principalmente del a facilidad de lectura → Los que modifican el SW generalmente no son los que lo hicieron (si es difícil de leer se complica el mantenimiento).

CRITERIOS DE EVALUACIÓN DE LENGUAJES DE PROGRAMACIÓN Y CARACTERÍSTICAS QUE LOS AFECTAN:

	CRITERIO		
CARACTERÍSTICA	Readability	Writability	Reliability
Simplicidad/Ortogonalidad	X	X	X
Estructuras de Control	X	X	X
Tipos de Datos y ED	X	X	X
Diseño de Sintaxis	X	X	X
Soporte para Abstracción		X	X
Expresividad		X	X
Chequeo de Tipos			X
Manejo de Excepciones			X
Aliasing Restringido			X

EVOLUCIÓN EN LOS CONCEPTOS:

- *NOMBRES SIMBÓLICOS* con restricciones (al principio se utilizaban 0 y 1 en tarjetas perforadas).

- *EXPRESIONES* más cercanas a la realidad (la expresión se escribe en una línea, no como una secuencia de instrucciones).
  Introducción de los primeros chequeo: que exista la operación, que los operandos sean accesibles, etc.

- *SUBPROGRAMAS* pensados solo como una forma de agrupar código.

- *TIPOS DE DATOS*
  - Los símbolos indexan variables de memoria con semántica, ya no hay locaciones de memoria anónimas.
  - Cada celda está asociada a un tipo.
  - Avanza el nivel de confiabilidad (chequeo y seguridad).
  - Sólo tipos de datos que proveía el lenguaje.
  - En la siguiente generación de lenguajes se extendió a tipos adecuados a diversas áreas de aplicación (definidos por el usuario).
  - La siguiente generación brinda pocos tipos primitivos, pero también más constructores generales combinables entre sí (más ortogonal).

- *ESTRUCTURAS DE CONTROL:*

  A nivel instrucción – Programación Estructurada
  - Hasta ahora había secuencia y transferencia de control (condicional e incondicional) y venían ligadas a la arquitectura subyacente.
  - La programación estructurada propone utilizar: 1. Secuencia, 2. Iteración, 3. Condicional. {Evolucionan los lenguajes eliminando la transferencia de control - goto}.
  - En base a las estructuras de control surgen luego la ejecución simétrica (o concurrente) y manejo de excepciones.

  A nivel unidad – Diseño Top-Down { Capítulo 9 – Sebesta }
  - Utilización de subprogramas para partir el problema en subproblemas.
  - Funcionalidad de la unidad: para resolver subproblemas; para usarla en el macro-problema.
  - Esquemas de relación entre unidades:

    Jerárquico: (Ej: C)
    1. La unidad llamadora se suspende.
    2. El subprograma corre por completo.
    3. Árbol de llamadas donde cada unidad llama a otra, sin ciclos.

    Simétrico: (Ej: C# - foreach)
    1. Uso de corrutinas para simular concurrencia (vinculado a la aparición de lenguajes para simulación. Ej: SIMULA).
    2. Esquema de relación útil para varias aplicaciones. Ej: Juegos multijugador.
    3. Es indispensable sincronizar las variables globales o compartidas.

Paralelo: (Ej: Java - thread)

1. También simula concurrencia.
2. En este caso no se requiere la suspensión de una de las unidades, ya que no hay recursos compartidos.

Unidades Latentes:

1. Surgió por diversos motivos, entre ellos el manejo de excepciones (la idea es evitar ensuciar el código con casos excepcionales).
2. Esta altamente vinculado al *manejo de eventos*.

- *ENCAPSULAMIENTO*: Encapsular teniendo en cuenta las acciones. Teniendo en cuenta los datos (TDA, representación adecuada, conjunto de operaciones).
  - Para que el lenguaje sea bueno en abstracción debe tener encapsulamiento y mecanismo para crear instancias.
  - Encapsulamiento sin abstracción = agrupamiento sin ocultamiento.

- *POLIMORFISMO:*

  Ad-Hoc:
  - *Coerción:* Convierte el tipo del argumento de una función al tipo esperado por ella (el polimorfismo es aparente).
  - *Sobrecarga:* Inicialmente sólo disponible para identificadores predefinidos. Es una forma de polimorfismo, se usa un mismo identificador para variables de distintos tipos (Significó una evolución para algunos y una involución para otros).

  Universal:
  - *Genericidad:* Un TDA define comportamiento abstracto independientemente del tipo de sus componentes.
  - *Herencia:* Forma particular de clasificación. Balancea la trascendencia de datos y acciones. Evitar sobre-especificar, sub-especificar, y repetir código.

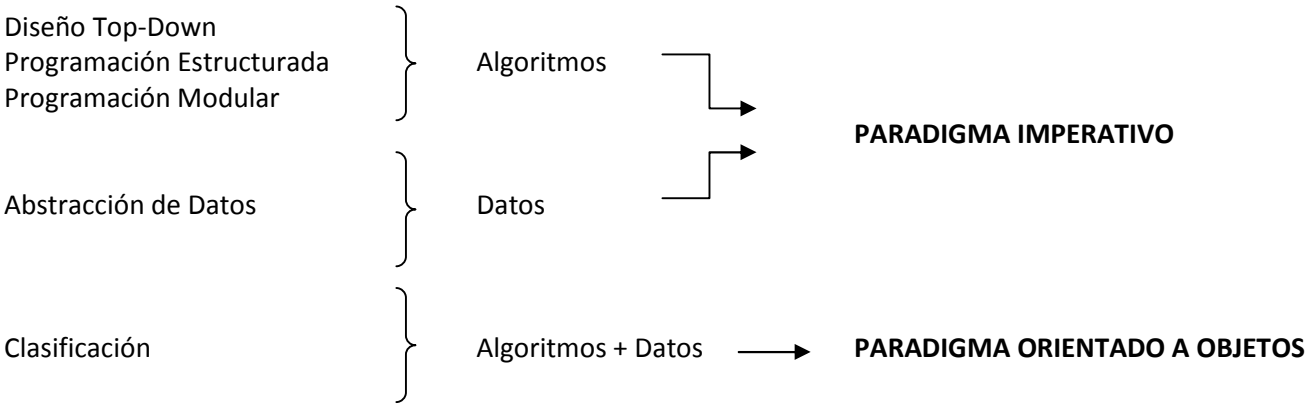
METODOLOGÍAS DE DISEÑO: Es un conjunto de métodos y pautas que guían el proceso de desarrollo.

PARADIGMA DE PROGRAMACIÓN: Conjunto coherente de métodos para el manejo de un tipo de problemas (dominio de aplicación). Tiene un principio básico. Guía el proceso de desarrollo del SW.
Colección de patrones conceptuales integrados que orientan el proceso de desarrollo de SW y determinan la estructura de un programa válido.

Un lenguaje *soporta* un paradigma si provee mecanismos que facilitan su implementación eficiente. Es decir, implementa mecanismos que permiten la sencilla aplicación del paradigma (impone restricciones para respetarlo).

Un lenguaje *admite* un paradigma si es posible escribir programas siguiendo los lineamientos del paradigma, pero hacerlo demanda un esfuerzo notable. El lenguaje no exige la aplicación del paradigma, permite programar de acuerdo al paradigma pero sin proveer facilidades.

LENGUAJES Y PARADIGMAS – FOCALIZADO EN ACCIONES O DATOS?



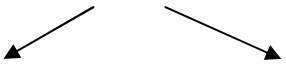
PARADIGMA IMPERATIVO: Un programa es una secuencia de instrucciones que indican el flujo de la ejecución. (Señal dada para que se realice el cambio de estado del autómata).

Las principales características son:

1. Ejecución secuencial de instrucciones
2. Uso de variables representando valores de locaciones de memoria
3. Uso de sentencias de asignación para cambiar los valores de las variables, permitiendo así al programa operar sobre las locaciones de memoria

Se busca estructurar el control realizando una programación estructurada y modular con abstracción de datos para fomentar la reusabilidad y extensibilidad.
Los programas se construyen siguiendo una aproximación:

- Top-Down — Sólo subprogramas
  Dividir para conquistar
- Modular — Existen abstracciones algorítmicas



Abstracción a nivel instrucción
(Agrupar instrucciones en unidades
Ej: *Procedure de Pascal*)

Abstracción de expresiones
(Ej: *Function de Pascal*)

Los programas se construyen siguiendo una aproximación *Modular:*

- *Programación Estructurada*: Estructuras de control a nivel instrucción. Inhibición del uso de estructuras de control.
- *Programación Modular*: Descomponer el problema priorizando recombinación y reutilización en otros problemas. El cómo descomponerlo es inherencia de la Ing. de SW.
  Módulo con independencia de significado y de implementación.
- *Abstracción de Datos:* Reconocer entidades abstractas (hallar representación para los datos). Reconocer operaciones lógicas (transformar en operaciones concretas).

Limitaciones del paradigma imperativo:

- *Dificultad para razonar*: alta dependencia de la arquitectura. Una variable es la abstracción de una celda de memoria.
- *Transparencia referencial*: asignaciones.
- *Efectos colaterales y Aliasing.*
- *Accesibilidad vulnerable:* desdibujan la lógica del programa creando problemas de mantenimiento y seguridad.

PARADIGMA ORIENTADO A OBJETOS: Se caracteriza por reconocer las entidades del problema (similar a la abstracción de datos). Caracterizado por atributos y comportamiento (de acuerdo a su propósito y habilidades). Entidad = *Objeto*. Comunicación por *mensajes*, diferente a la semántica de llamadas a procedimientos.

Objetivos: Mejorar la *reusabilidad* y *extensibilidad* del SW.

Los lenguajes OO deben proveer:

- Comunicación por *mensajes*.
- Concepto de *objeto* con estado interno y servicios.
- Soportar *clasificación*.

El mínimo que deben soportar (según algunos autores) es: *abstracción de datos* + *herencia* + *encapsulamiento* + *polimorfismo*. →Abstracción de datos y encapsulamiento deben ir juntos.
Para algunos autores polimorfismo es caso particular de herencia, y para otros, polimorfismo, herencia y sobrecarga son conceptos distintos.
Otros autores requieren además *ligadura dinámica* o *soporte de concurrencia* (no inherente al paradigma).

PARADIGMAS NO CONVENCIONALES: Las formalizaciones abstractas (MT, cálculo lambda, etc.) brindan formas de expresión y razonamiento que permiten obtener construcciones útiles y adecuadas para descubrir problemas. Redescubrir los formalismos como una forma de resolver problemas, utilizándolos en el diseño y la implementación.

Reciben el nombre de *no convencionales* porque su forma de especificar y ejecutar está "alejada" de la arquitectura. Están alejados totalmente del control del paradigma imperativo.

La lógica y la matemática brindan formas de expresión como:

PARADIGMA DECLARATIVO ⟶ **PARADIGMA LÓGICO**
PARADIGMA APLICATIVO ⟶ **PARADIGMA FUNCIONAL**

[Apunte ML]
Los modelos que se concentran en especificar "qué" se desea computar, en lugar de "cómo" la computación debe ser llevada a cabo se llaman *declarativos.* Podemos distinguir dos paradigmas que siguen esta filosofía: paradigma lógico y paradigma funcional.

PARADIGMA DECLARATIVO: Un programa establece un conjunto de propiedades que describen cómo alcanzar la solución. En ningún caso se indica cómo se computa. Un lenguaje declarativo brinda las facilidades para especificar estas propiedades y una forma de computar que sea transparente.
- Poca eficiencia
+ Programas elegantes y concisos

PARADIGMA LÓGICO: Basado en lógica de primer orden (lenguaje preciso para expresar conocimiento). Lenguaje representativo: *Prolog*.

Los fundamentos del paradigma son:
- Deducir consecuencias a partir de las premisas.
- Estudiar o decidir el valor de verdad de una sentencia a partir del valor de verdad de otras.
- Establecer la consistencia entre hechos y verificar la validez de argumentos.

Características de los lenguajes lógicos:
- Eliminación del *control*.
- El concepto de *variable* es más matemático, son nombres que retienen valores.
- Establecen "*qué*" es lo que se debe hacer sin dar ninguna especificación sobre el "cómo" hacerlo.

Características de los programas lógicos:
- Conjuntos de *axiomas* que establecen relaciones.
- Definen un conjunto de *consecuencias* que determinan el significado.
- Son *teoremas* y la ejecución es una *prueba* automática.

PARADIGMA APLICATIVO: Basado en el uso de funciones, por esto habitualmente se habla de este paradigma como el **paradigma funcional**. Muy popular en la resolución de problemas de inteligencia artificial, matemática, lógica, procesamiento paralelo.

+ Vista uniforme de programa y función.
+ Tratamiento de funciones como datos.
+ Liberación de efectos colaterales.
+ Manejo automático de memoria
- Ineficiencia de ejecución.

PARADIGMA FUNCIONAL: La esencia de esta metodología está en componer funciones para definir otras más complejas.

Una función es:

- Regla que mapea un valor del dominio en uno correspondiente al rango.
- Requiere establecer una signatura y una regla de mapeo.
- En la *definición* se establecen los parámetros. En la *aplicación* se habla de argumentos. El *retorno* de la función provee el resultado.

Características de los lenguajes funcionales:

- Define un conjunto de *datos*.
- Provee un conjunto de *funciones primitivas*.
- Provee un conjunto de *formas funcionales*.
- Requiere de un *operador de aplicación*.

Características de los programas funcionales:

- *Semántica* basada en *valores*.
- Transparencia referencial.
- Regla de mapeo basada en combinación o composición.
- Las funciones son "ciudadanos" de primer orden.

INTEGRACIÓN DE PARADIGMAS: Aparecen los lenguajes híbridos, principalmente debido a necesidades heterogéneas y la interoperabilidad disponible.

ORTOGONALIDAD: Atributo de poder combinar varias características de un lenguaje en todas las posibles combinaciones, con cada combinación teniendo significado.
Ej: Supongamos un lenguaje que provee expresiones que pueden producir un valor, y también provee un condicional que evalúa una expresión para obtener un valor trae o false. Estas dos características del lenguaje, expresión y sentencia condicional son ortogonales si TODAS las expresiones pueden usarse (y evaluarse) dentro de una sentencia condicional.

La SINTAXIS del lenguaje afecta a la facilidad de escritura, testeo y posterior entendimiento y modificación de los programas. Facilidad de Lectura vs. Facilidad de Escritura, ya que un lenguaje con sintaxis críptica puede escribirse fácil (en una línea puedo hacer muchas cosas) pero luego es imposible de entender lo que se hizo.
Otro problema con la sintaxis de los lenguajes se presenta en aquellos que presentan dos sentencias casi idénticas para hacer cosas radicalmente diferentes (puede traer problemas).

CAPÍTULO V – Nombres, Ligaduras, Chequeos de Tipo y Alcance

5.1 INTRODUCCIÓN

Los lenguajes de programación buscan la seguridad y facilidad en la programación. Los LP imperativos son abstracciones de la arquitectura de Von Neumann, cuyos componentes primarios son: memoria (almacena instrucciones y datos) y procesador (provee operaciones para modificar el contenido de la memoria).La abstracción en un LP para las celdas de memoria son las *variables*, cuyo atributo más importante es el *tipo*.

El *diseño de los tipos de datos de un lenguaje* requiere considerar una variedad de decisiones como son el *alcance* y el *tiempo de vida* de las variables. Relacionadas a estas están las decisiones de *chequeo de tipos* y la *inicialización*. Esto permite entender los lenguajes imperativos.

Hay dos alternativas extremas para llevar cualquier lenguaje a lenguaje máquina: *Interpretación* y *Traducción*.

5.2 NOMBRES

Cadena de caracteres utilizada para identificar entidades en un programa.
Los nombres o identificadores son uno de los atributos fundamentales de las *variables*. También son asociados con *etiquetas*, *subprogramas, parámetros formales*, etc.

5.2.1 Decisiones de Diseño (del LP con respecto a los nombres)

- Los nombres tienen *longitud* máxima?

Si la longitud es muy restringida afecta la legibilidad del programa. Ej: FORTRAN 6 chars – C++/Java sin límite

- El lenguaje es *case sensitive*?

El hecho de que los lenguajes sean sensibles a mayúsculas y minúsculas perjudica la legibilidad (nombres muy similares denotan entidades muy diferentes).

- Se permite el uso de *conectores*?

Muchos lenguajes no los permiten.

- Hay *palabras especiales (clave o reservadas)*?

5.2.3 Palabras Especiales

Son utilizadas para que los programas sean más legibles al nombrar las acciones a realizar. También son usadas para separar las entidades sintácticas de los programas. (Generalmente son llamadas palabras reservadas, pero algunas de ella son sólo palabras clave).

Las *palabras clave* aumentan la facilidad de lectura, ya que funcionan como separadores. Una palabra clave es un identificador que sólo es especial (en interpretación) en ciertos contextos.

Una *palabra reservada* es una palabra especial que no puede ser utilizada como nombre (no puede redefinirse por el usuario). Tiene su semántica claramente definida. Estas son una mejor elección ante las palabras clave, ya que la facilidad para redefinir palabras clave puede conducir a problemas de legibilidad.

Las *palabras predefinidas* tienen un significado predefinido, pero puede ser modificado por el programador.
El abuso de este tipo de palabras en un lenguaje perjudica su aprendizaje y evolución.

5.3 VARIABLES

Una variable es una abstracción de una celda de memoria o de una colección de celdas. No sólo es un nombre para una locación de memoria, sino que existe mucho más que el nombre para una variable.
Los atributos que caracterizan una variable son:

- **Nombre:** Los nombres de variables son los nombres más comunes en un programa, pero no todas las variables poseen un nombre.
- **Dirección:** También es llamada *i-valor*, y corresponde a la dirección de memoria al a que la variable está asociada.
  Una misma variable puede estar asociada a locaciones de memoria diferentes durante la ejecución de un programa, y en distintos lugares del programa.
  Cuando más de un nombre de variable puede utilizarse para acceder a la misma locación de memoria al mismo tiempo se produce ***aliasing***. Los alias se pueden crear a través del uso de punteros (variables por referencia).
  El *aliasing* perjudica la legibilidad de los programas, ya que el programador que lee el programa debe tener presente todos los alias de una locación (el valor de una variable puede ser modificado mediante otro nombre). Adicionalmente, se dificulta la verificación de los programas.
- **Valor:** también es conocido como *d-valor*, y es el contenido de la celda o celdas de memoria asociadas con la variable.
- **Tipo:** Determina el rango de valores para la variable, y las operaciones que están definidas para los valores de ese tipo.
- **Tiempo de Vida**
- **Alcance**

5.4 EL CONCEPTO DE LIGADURA

Los programas trabajan con *entidades* (variables, rutinas, sentencias), las cuales se asocian a atributos (nombre, tipo, valor, parámetros, etc).

Una ligadura es una asociación entre el valor de un atributo y una entidad, o entre una operación y un símbolo. La información sobre los atributos de una entidad se encuentra en un *descriptor*. El tiempo en que una ligadura toma lugar se llama *tiempo de ligadura.* Las ligaduras físicas de una variable a una celda de memoria son mantenidas por el HW y, el cambio es invisible para el programa.

Los lenguajes varían en la cantidad de entidades que pueden manejar, en el *tiempo de ligadura* y la *estabilidad de la ligadura* (fija o modificable):

TIEMPOS DE LIGADURA

Diseño del LP
Implementación del LP
Compilación
Vinculación
Carga

LIGADURA ESTÁTICA: Se realiza antes de la ejecución y no se modifica durante la ejecución del programa.

Ejecución ⟶ LIGADURA DINÁMICA: Se realiza durante la ejecución, o puede cambiar en el curso de la ejecución del programa.

{Ejemplos de Tiempos de Ligaduras}

Diseño del Lenguaje → Ligar símbolos de operadores a operaciones.
Implementación del Lenguaje → Ligar el tipo punto flotante a su representación; el tipo int a su rango de valores.
Compilación → Ligar una variable a su tipo en C o Java.
Carga → Ligar una variable estática de C a una celda de memoria.
Ejecución → Ligadura de variables no estáticas a celdas de memoria.

5.4.2 Ligaduras de Tipo
Antes que una variable pueda ser utilizada debe estar *ligada* a un tipo. Es relevante decidir entonces 2 aspectos: *cómo* se especifica el tipo, y *cuándo* la ligadura toma lugar.

LIGADURA ESTÁTICA: El tipo se determina a través de una declaración explícita o implícita.

- Una declaración explícita es una sentencia del programa en la cual se declara el tipo de la variable.
- Una declaración implícita es un mecanismo para especificar, por defecto, el tipo de una variable (la primera aparición de la variable en el programa). Ej: Fortran, Perl.
  + Facilidad de escritura
  - Perjudican la confiabilidad (impiden detectar errores tipográficos en compilación)
  → Algunos problemas pueden evitarse requiriendo que los nombres para ciertos tipos comiencen con caracteres especiales (Ej: $ para tipos escalares el Perl).

LIGADURA DINÁMICA: El tipo no se especifica en la sentencia de declaración, ni puede ser determinado por su identificador de nombre. La variable es ligada a un tipo cuando se le asigna un valor mediante una sentencia de asignación (se liga al tipo de la expresión en el extremo derecho de la asignación).
+ Flexibilidad.
- La capacidad detección de errores por parte del compilador disminuye. Tipos incorrectos del lado derecho de las asignaciones no son detectados como errores (se puede asignar un valor de cualquier tipo a las variables). Este error también puede ocurrir en lenguajes con ligadura estática pero con conversión automática de tipos.
- Alto costo: los chequeos de tipo deben realizarse en tiempo de ejecución; además, cada variable debe poseer un descriptor asociado que indique el tipo actual.

Los LP con ligadura dinámica de tipo deben ser implementados usando interpretación pura en vez de compilación. Los que tienen ligadura estática rara vez son interpretados, ya que pueden ser fácilmente traducidos a código máquina eficiente.

OBS: Es importante distinguir entre declaración (especifica tipos y otros atributos pero no causa la alocación de almacenamiento) y definición (especifica atributos *y* causa alocación de almacenamiento). Ej: para un determinado nombre, un programa en C puede tener cualquier número de declaraciones compatibles, pero una única definición.

INFERENCIA DE TIPO: El tipo de la mayoría de las expresiones puede ser determinado sin requerir que el programador especifique el tipo de las variables (es determinado por el contexto).

Ejemplos: ML

fun square(x) = x*x → ML infiere que el tipo del parámetro y del resultado son *int* (tipo default numérico).

Si se efectuara una llamada con un número real causaría un error (no hay coerción de *real* a *int*), en cuyo caso habría que reescribir la función:

fun square(x): real = x*x → como ML no permite sobrecarga de funciones, esta versión y la anterior no podrían coexistir.

5.4.3 Ligaduras al Almacenamiento y Tiempo de Vida

La celda de memoria a la cual una variable es ligada debe ser tomada de un pool de memoria disponible. Este proceso es llamado *asignación (allocation)*. El proceso de devolver una celda de memoria que ha sido desligada al pool de memoria es llamado *liberación (deallocation)*.

El tiempo de vida de una variable es el tiempo durante el cual la variable está ligada a una locación de memoria específica. Por lo tanto, comienza cuando es ligada a una celda y finaliza cuando es desligada de la misma.

CLASIFICACIÓN DE VARIABLES SEGÚN SU TIEMPO DE VIDA (según tiempo de ligadura al almacenamiento):

- ESTÁTICAS: Son ligadas a celdas de memoria antes de la ejecución del programa, y permanecen ligadas durante toda la ejecución.
  + Algunas variables en subprogramas son *sensibles a la historia*, es decir, retienen valores entre distintas ejecuciones de los subprogramas.
  + Eficiencia (direccionamiento/acceso directo)
  - Falta de flexibilidad (un LP con únicamente variables estáticas no permite soportar recursividad)
  - No permite almacenamiento compartido entre variables (Ej: Si 2 procedimientos usan arreglos grandes, y nunca son ejecutados concurrentemente, podrían compartir el almacenamiento)
  OJO!! El *static* de Java y C implican que es una variable de clase en vez de variable de instancia.

- DINÁMICAS BASADAS EN PILA: Son aquellas cuya ligadura al almacenamiento se crea cuando se *elabora* su sentencia de declaración, pero cuyos tipos son estáticamente ligados.
  La *elaboración* de la declaración se refiere a la asignación de almacenamiento y al proceso de ligadura indicado por la declaración, que ocurre cuando la ejecución alcanza el código al corresponde la declaración.
  + Admiten recursión
  + Los subprogramas comparten el mismo espacio de memoria para las variables locales
  - Overhead en el tiempo de ejecución, al momento de alocar y desalocar memoria
  - Las variables no pueden ser sensibles a la historia
  - Referencias ineficientes (direccionamiento indirecto)

- DINÁMICAS CON HEAP EXPLÍCITO: Son celdas de memoria anónimas (abstractas) que son alocadas y desalocadas a través de directivas explícitas, realizadas por el programador. Las variables sólo pueden ser referenciadas a través de punteros o referencias (Ej: variables dinámicas de C++ y objetos de Java). El *heap* es una colección de celdas de memoria desorganizadas, dada la imprevisibilidad de su uso.

+ Provee manejo dinámico del medio de almacenamiento (usadas para ED dinámicas: listas, árboles)
- Dificultad para el uso de punteros y referencias correctamente
- Ineficiencia (costo de referencias a variables, alocación y dealocación)

- DINÁMICAS CON HEAP IMPLÍCITO: Son ligadas al almacenamiento (heap) sólo cuando se les son asignados valores. Todos sus atributos son ligados cada vez que son reciben una asignación de valores: la alocación y dealocación se produce a través de las sentencias de asignación (son sólo nombres que se adaptan a cualquier uso).
  + Flexibilidad (permiten escribir código altamente genérico)
  - Ineficiencia (overead de ejecución para mantener todos los atributos, ya que son dinámicos)
  - Pérdida de poder de detección de errores por parte del compilador

5.5 CHEQUEO DE TIPOS

Es una actividad que permite asegurar que una operación y sus operandos son de tipos compatibles. Un tipo es compatible si es legal para la operación, o si está permitido por las reglas del lenguaje que se convierta implícitamente a un tipo legal (*coerción*).

Un error de tipo sucede cuando se aplica una operación a un operando de tipo inapropiado. Si todas las ligaduras de tipo son estáticas, entonces el chequeo de tipos se puede hacer casi completamente estático. Si la ligadura es dinámica, el chequeo deberá ser dinámico (en ejecución). Ej: Javascript y APL tienen chequeo dinámico (por la ligadura dinámica de tipos).

Si se realiza el *chequeo en compilación* se dice que el sistema de tipos es *estáticamente tipado.* Si el chequeo es realizado en ejecución se tiene un lenguaje con *tipado dinámico* (sistema dinámicamente tipado).

El chequeo de tipo se complica cuando un lenguaje permite que una celda de memoria almacene valores de diferentes tipos en distintos momentos durante la ejecución (Ej: registros variantes de Ada, unión de C y C++). En estos casos, el chequeo de tipos (si se efectúa) debe ser dinámico y requiere que el sistema en ejecución mantenga el tipo actual que se almacena en tales celdas de memoria.
Obs: Aunque todas las variables son estáticamente ligadas a los tipos en lenguajes como C++, no todos los errores de tipo pueden ser detectados mediante chequeo dinámico de tipos.

5.6 TIPADO FUERTE

[Sebesta] Un lenguaje es fuertemente tipado si los errores de tipo *siempre* son detectados. Esto requiere que los tipos de todos los operandos puedan ser determinados, ya sea en tiempo de compilación o de ejecución.

[Ghezzi] Si un lenguaje es fuertemente tipado, el compilador puede garantizar la ausencia de errores de tipo en los programas (los errores de tipo se detectan en compilación).

→ La importancia del tipado fuerte recae en su capacidad de detectar el mal uso de variables que resultan en errores de tipo. Un lenguaje fuertemente tipado también permite la detección, en tiempo de ejecución, del uso de valores de tipo incorrecto en variables que pueden almacenar valores de más de un tipo.
Las reglas de coerción de un lenguaje tienen un efecto importante en el chequeo de tipos. Sin embargo, la coerción también resulta en una pérdida del sentido de tener tipado fuerte (detección de errores).

Ej: *ML* es *fuertemente tipado*, pero en un sentido distinto que los lenguajes imperativos. ML tiene variables cuyos tipos se conocen estáticamente, ya sea por sus declaraciones, o por sus reglas de inferencia de tipos.

5.7 COMPATIBILIDAD DE TIPOS

La idea de compatibilidad de tipos se definió por necesidad para el chequeo de tipos.
Influencia el diseño de los tipos de datos y las operaciones provistas para ese tipo. Lo más importante sobre compatibilidad es saber si un tipo puede asignar su valor al otro.

Hay dos tipos de reglas de compatibilidad para variables de tipos estructurados (no escalares):

- *COMPATIBILIDAD POR NOMBRE:* Dos variables tienen tipos compatibles si están definidas en la misma declaración, o en declaraciones que usan el mismo tipo (mismo nombre de tipo).
  + Fácil de implementar (Sólo deben compararse los nombres de los tipos).
  - Demasiado restrictivo (Los subrangos no son compatibles con su tipo base; Los parámetros actuales deben ser los mismos que los formales).

- *COMPATIBILIDAD POR ESTRUCTURA:* Dos variables tienen tipos compatibles si sus tipos tienen estructuras idénticas.
  + Más flexible
  - Más difícil de implementar (Deben compararse las estructuras de los tipos por completo).
  - No permite diferenciar entre tipos con la misma estructura (los considera compatibles).

- *COMPATIBILIDAD POR DECLARACIÓN:* Dos variables son equivalentes por declaración si son equivalentes por nombre, o si el tipo de una de ellas está definido directamente a través del tipo de la otra.

~~Un tipo derivado es un tipo que se basa en otro tipo previamente definido con el cual es incompatible, aunque puedan tener la misma estructura. Los tipos derivados heredan todas las propiedades de sus tipos padres. Pueden agregar restricciones de rango sobre su tipo padre, heredando aún todas las operaciones de su padre.~~
~~Ej: *type* celsius *is new* float;~~

~~Un subtipo es posiblemente una restricción de rango de un tipo existente, y es compatible con su tipo padre.~~ λ

5.8 ALCANCE

El alcance de una variable es el rango de sentencias en el cual la variable es visible. Una variable es *visible* en una sentencia si puede ser referenciada en ella. Para determinar la visibilidad de una variable es necesario definir las *reglas de alcance.*

Las reglas de alcance determinan cómo una ocurrencia particular de un nombre es asociada con una variable. En particular, determinan cómo referencias a variables declaradas fuera del subprograma en ejecución o bloque son asociadas con sus declaraciones, y por lo tanto con sus atributos.

Una variable es local en un bloque de programa si es declarada allí, y es no local si es visible desde un bloque pero no está declarada allí.

Para establecer las reglas de alcance, es necesario conocer las *estructuras de los programas*:

MONOLÍTICA: El alcance de todo identificador es el programa entero. Existe un único ambiente de referenciamiento.

BLOQUES CHATOS: Todas las unidades están al mismo nivel: existe una unidad principal. Las unidades se consideran globales. Las variables en principio son locales, si se requiere que sea global hay que hacerlo en forma explícita a través del mecanismo que provea el lenguaje.

BLOQUES ANIDADOS: Las referencias se resuelven buscando la ligadura más profunda. Primero se busca en el ambiente local, si no se encuentra se va propagando la búsqueda hacia afuera.

Además de depender de la estructura del programa, el alcance puede pensarse vinculado a si es determinado antes de la ejecución o durante ella:

- *ALCANCE ESTÁTICO:* El alcance de una variable puede ser determinado estáticamente (antes del a ejecución), en base al anidamiento y definición de subprogramas (estructura del programa). Cuando una referencia a una variable es encontrada por el compilador, los atributos de la variable son determinados encontrando la sentencia donde ha sido declarada. Esto se hace de la siguiente manera: cuando se encuentra una referencia a la variable X en un subprograma, primero se busca su declaración en el subprograma que hizo la referencia. De no estar allí se busca la declaración en el subprograma donde se encuentra declarado el subprograma que hizo la referencia, y así sucesivamente hasta llegar al programa principal. Si se encuentra allí la declaración, se produce un error.
  Ventajas y Desventajas:
  + Provee un método de acceso no local que funciona bien en muchos casos.
  + Fácil de leer.
  + Ejecución rápida.
  - Todas las variables declaradas en el programa principal son visibles a todos los procedimientos, aunque esto no se desee.
  - Cambios en las especificaciones obligan a mover subprogramas anidados a un nivel más cercano al del programa principal.
  - Un programador puede equívocamente llamar a un subprograma que no debe poder ser llamado (este error no se detecta hasta la ejecución).

- *ALCANCE DINÁMICO:* Está basado en la secuencia de llamadas, no en la relación espacial entre subprogramas. De esta manera, el alcance sólo puede determinarse en tiempo de ejecución.
  Cuando la búsqueda de la declaración en el ámbito local falla, se procede a buscar en los *ancestros dinámicos*, que corresponden al subprograma que llamó al subprograma actual, y así sucesivamente con sus ancestros dinámicos, hasta encontrar la declaración, o producirse un error (en caso de no encontrarla).
  Desventajas:
  - Programas menos legibles (las referencias no locales requieren el conocimiento de la secuencia de llamadas).
  - El acceso a variables no locales toma mucho más tiempo que con alcance estático.
  - Una referencia a una variable no local en un subprograma puede corresponder a distintas variables no locales durante diferentes ejecuciones.
  - Los atributos correctos de las variables no son detectados estáticamente.
  - No es posible proteger el acceso a variables locales.

5.9 ALCANCE Y TIEMPO DE VIDA

Si bien los conceptos de *alcance* y *tiempo de vida* parecen estar relacionados, el concepto de alcance es espacial, mientras que el tiempo de vida es un concepto temporal.

Ej: Sea una variable que está declarada en un procedimiento en pascal, el cual no contiene llamadas a subprogramas. El alcance de esta variable es el cuerpo del procedimiento, mientras que su tiempo de vida es el período de tiempo entre que comienza y termina la ejecución del procedimiento.

5.10 AMBIENTES DE REFERENCIAMIENTO

El *ambiente de referenciamiento* de una sentencia es la colección de nombres que son visibles en la sentencia.

En lenguajes con reglas de alcance estático: el ambiente de referenciamiento está conformado por todos los nombres en su alcance local, más los nombres de sus alcances ancestros que son visibles (ambientes que lo contienen).
En lenguajes con reglas de alcance dinámico: el ambiente de referenciamiento está conformado por los nombres declarados localmente, más los identificadores de todos los subprogramas que están actualmente activos. Un subprograma está activo si su ejecución ha comenzado pero aún no ha finalizado.

5.11 CONSTANTES NOMBRADAS

Una *constante nombrada* es una variable que se liga a un valor sólo cuando se realiza la ligadura a su correspondiente locación de memoria (al almacenamiento). Su valor no puede cambiar por una asignación, o por una sentencia de entrada.
+ Son útiles para la confiabilidad y legibilidad del programa
+ Ahorra repetición de código
+ Facilita el mantenimiento

5.12 INICIALIZACIÓN DE VARIABLES

La ligadura de una variable a su valor en el momento que se liga al medio de almacenamiento es llamada *inicialización.* Generalmente toma lugar a través de una asignación.
Si la variable es estáticamente ligada al almacenamiento, la ligadura y la inicialización ocurren antes de la ejecución. Si la ligadura es dinámica, la inicialización también es dinámica.
Ejemplos:

Java/C++: int x=2;
Pascal y Modula2 no proveen un modo de inicializar variables (salvo por asignaciones en tiempo de ejecución).

CAPÍTULO VII – Expresiones y Sentencias de Asignación

7.1 INTRODUCCIÓN

Las expresiones son una forma fundamental de especificar computaciones en un lenguaje de programación.
Es esencial para un programador entender tanto la *sintaxis* (la forma - Ej: BNF) como la *semántica* (el significado) de las expresiones, la cual está determinada por su forma de evaluación.
Para entender la evaluación de las expresiones, es necesario tener en cuenta el *orden de evaluación de operandos y operadores*, el cual está determinado por las reglas de asociatividad y precedencia del lenguaje.

La esencia de los lenguajes imperativos es el rol dominante de las sentencias de asignación. El propósito de una sentencia de asignación es cambiar el valor de una variable.

7.2 EXPRESIONES ARITMÉTICAS

Las expresiones aritméticas constan de: operadores, operandos, paréntesis, llamadas a funciones. Los *operadores* pueden ser *unarios* (un solo operando), *binarios* (dos operandos), o *ternarios* (C-based languages).
La mayoría de los lenguajes imperativos adoptan una notación *infija* para los operandos binarios, pero existen otras opciones (*prefija, posfija*).

El *propósito* de una expresión aritmética es especificar una computación aritmética (cálculo aritmético). Una implementación de tal computación debe consistir de dos acciones: recuperar los operandos (generalmente de la memoria), y ejecutar las operaciones aritméticas sobre tales operandos.

ASPECTOS DE DISEÑO (para expresiones aritméticas):

- Reglas de *precedencia* de operadores
- Reglas de *asociatividad* de operadores

} Control

- *Orden de evaluación* de los operandos
- Evaluación de *efectos colaterales* de un operador
- *Sobrecarga* de un operador
- *Modo mixto* de expresiones
- *Errores* en las expresiones

Las *reglas de precedencia* para la evaluación de expresiones define el orden en el cual operadores adyacentes (con distintos niveles de precedencia) son evaluados. Estas reglas se basan en la jerarquía de prioridades de operadores, definidas por el diseñador del lenguaje.
Los niveles de precedencia típicos son: paréntesis, operadores unarios, potenciación, división y multiplicación, suma y resta.
Obs: En general, en todos los lenguajes imperativos las reglas de precedencia son las mismas, ya que están basadas en las reglas matemáticas.

Las *reglas de asociatividad* definen cómo se resuelve el orden de evaluación frente a operadores adyacentes del mismo nivel de precedencia. Un operador puede tener *asociatividad a izquierda* o *asociatividad a derecha* (generalmente los lenguajes imperativos utilizan asociatividad a izquierda).

El uso de *paréntesis* en las expresiones puede modificar las reglas de asociatividad y precedencia. Una parte parentizada de una expresión tiene precedencia sobre sus partes adyacentes no parentizadas.
Los LP podrían prescindir de las reglas de asociatividad y precedencia, especificándolas explícitamente en cada expresión mediante el uso de paréntesis → La programación es muy tediosa.

Los lenguajes basados en C (C, C++, Java) poseen un operador ternario ?:, que es utilizado para formar *expresiones condicionales*. La forma de las expresiones es *exp_bool ? exp1 : exp2*. Si la expresión booleana se evalúa como verdadera, el resultado de toda la expresión es el valor de *exp1*, en caso contrario el resultado es el valor de *exp2*.
Ej: if (cont==0) then average = 0; else average = sum/cont → average = (cont==0) ? 0 : sum/cont;

Orden de evaluación de los operandos:
- Las *variables* en las expresiones se evalúan recuperando sus valores desde memoria.
- Las *constantes* algunas veces son evaluadas de la misma forma que las variables. Otras veces, la constante puede ser parte de las instrucciones del lenguaje máquina (no requiere acceder a memoria).
- Si un operando es una *expresión parentizada*, entonces todos los operadores que contiene deben ser evaluados antes de que su valor pueda ser utilizado como un operando. → Se evalúan las expresiones de acuerdo al modo de evaluación determinado por los paréntesis.

Si ninguno de los operandos u operadores tiene *efectos colaterales*, entonces el orden de evaluación es irrelevante.

Un efecto colateral de una función, llamado *efecto colateral funcional*, ocurre cuando una función modifica un parámetro de entrada-salida o una variable global.
Ej: a + fun(a) → Si fun altera el valor de a el orden de evaluación es relevante, sino no afecta al resultado

Existen dos soluciones al problema de orden de evaluación de operandos:
1. El diseñador del lenguaje puede *deshabilitar los efectos colaterales funcionales*, evitando así que la evaluación de una función afecte el valor de una expresión:
   a) No hay parámetros de entrada-salida en las funciones
   b) No hay referencias a variables globales en las funciones
      + Es una solución
      - Es inflexible
2. Escribir la definición del lenguaje de manera que fije el orden de evaluación de los operandos, requiriendo que el programador lo respete.
   - Limita las posibilidades de optimizar código (compiladores que utilizan técnicas de reordenamiento de operandos), cuando hay llamadas a funciones involucradas en las expresiones.

7.3 OPERADORES SOBRECARGADOS
La *sobrecarga de operadores* tiene lugar cuando se utiliza el mismo operador para más de un propósito. Características:
- Pérdida de capacidad de detección de errores por parte del compilador (Ej: en C el operador & como binario es el *and*, y como unario es el acceso a memoria).
- Dificulta la lectura del código (pérdida de *readability).*
- La sobrecarga puede evitarse introduciendo nuevos operadores.
- Lenguajes como C++ y Ada permiten al usuario definir operadores sobrecargados (Desv: pueden definirse operadores sin sentido).

7.4 CONVERSIÓN DE TIPOS

Una *conversión* de tipo o bien *limita* el tipo o lo *expande*:

Una *conversión limitante* sucede cuando se convierte el tipo de un objeto a un tipo que no puede incluir todos los valores del tipo original. Ej: convertir un *float* a *int*.
→Este tipo de conversiones puede traer problemas (Ej: float a int, el valor queda truncado, lejos del valor original).

Una *conversión expansora* sucede cuando se convierte el tipo de un objeto a un tipo que puede incluir al menos aproximaciones de todos los valores del tipo original. Ej: convertir un *int* a *float*.
→Si bien este tipo de conversiones generalmente no trae problemas, pueden reducir la precisión (Ej: convertir de int a float, dado que ambos se almacenan en 32 bits, se pierde precisión para la parte entera – el float debe compartir los 32 bits entre parte entera y decimal).

Las *conversiones* de tipo, a su vez, pueden ser *explícitas o implícitas*.

7.4.1 Conversión Implícita – Coerción en Expresiones

Este tipo de conversiones son generalmente iniciadas por el compilador. La mayoría de los lenguajes provee coerciones expansoras para sus expresiones.

Una de las decisiones de diseño respectivas a las expresiones, es determinar si se van a permitir *expresiones mixtas* (expresiones en las que un operador puede tener operandos de diferentes tipos).

Los lenguajes que permiten expresiones mixtas deben definir convenciones para las conversiones implícitas de tipo de sus operandos, dado que las computadoras generalmente no cuentan con operadores binarios que aceptan operandos de diferentes tipos.
Para los *operadores sobrecargados* en un lenguaje con ligadura estática de tipo, el compilador elige el tipo correcto para la operación, basándose en los tipos de los operandos. Cuando los dos operandos de un operador no son del mismo tipo, y es legal en el lenguaje, el compilador debe elegir uno de ellos para aplicarle coerción, suministrando el código para tal coerción.
- Problemas de confiabilidad, porque eliminan los beneficios del chequeo de tipos
+ Aporta flexibilidad

7.4.2 Conversión Explícita

Las conversiones de tipo explícitas pueden ser tanto limitantes como expansoras. En algunos casos, mensajes de advertencia son generados por el compilador cuando una conversión explícita limitante da como resultado un cambio significativo en el valor del objeto convertido.

Ej: En C se hace mediante el cast → (int) prom
En Ada la sintaxis es parecida a la de llamadas a función → prom = float(suma)/float(cont)

7.4.3 Errores en Expresiones

Si el lenguaje requiere chequeo de tipos, no es posible que ocurran errores de tipo en los operandos.
Los errores que pueden ocurrir se deben a limitaciones inherentes a la aritmética (Ej: división por cero), o aquellos causados por limitaciones en la forma de computar la aritmética/limitaciones de la computadora (Ej: overflow-underflow en la representación). → Este tipo de errores ocurren en ejecución (*excepciones*)

7.5 EXPRESIONES RELACIONALES Y BOOLEANAS

Además de expresiones aritméticas, los lenguajes de programación también proveen expresiones relacionales y booleanas.

7.5.1 Expresiones Relacionales

Un *operador relacional* es un operador que compara los valores de sus dos operandos. Una *expresión relacional* usa un operador relacional sobre dos operandos. El valor obtenido de cualquier expresión relacional es *booleano*, excepto en los lenguajes en que este no es un tipo válido.

Los operadores relacionales generalmente están sobrecargados para una gran variedad de tipos (generalmente tipos numéricos, strings, tipos ordinales). Los lenguajes utilizan diferentes símbolos para representar los operadores relacionales, pero siempre tienen menor precedencia que los operadores aritméticos.

Ej:	OPERACIÓN	ADA	C, JAVA, C++
	Igual	=	==
	Distinto	/=	!=
	Mayor	>	>
	Menor o igual	<=	<=

7.5.2 Expresiones Booleanas

Este tipo de expresiones consisten de variables y constantes booleanas, expresiones relacionales, y operadores booleanos. Los resultados son valores booleanos.

Al igual que con los operadores relacionales, hay una variedad de representación entre los distintos lenguajes para los operadores booleanos.

Ej:	OPERACIÓN	PASCAL	JAVA
	AND	and	&
	NOT	not	!
	OR	or	|

La precedencia de los operadores booleanos y relacionales en general es la misma, pero hay lenguajes (Ej: Ada) donde se asocia mayor precedencia al AND que al OR, ya que el AND es asociado con la operación aritmética de multiplicación, y el OR con la suma (* tiene mayor precedencia que +).

Si dos o más operadores booleanos distintos aparecen en la misma expresión, deben utilizarse paréntesis para mostrar el orden de la evaluación.

Obs: C no posee el tipo booleano, sino que lo implementa mediante los valores enteros 0 y 1.
En C, la expresión a<b<c nunca compara los valores b y c. Dado que tiene asociatividad a izq, compara a con b, y su resultado con c.

7.6 MODOS DE EVALUACIÓN DE EXPRESIONES (posibles formas de evaluar expresiones)

- *ORDEN APLICATIVO, ESTRICTA O ANSIOSA*: Corresponde a la evaluación *bottom-up* del árbol que representa la expresión. Los operandos son evaluados en forma independiente, luego el operador es aplicado.

- *ORDEN NORMAL*: Corresponde a la evaluación de cada operando cuando es necesitado en la computación del resultado.
  Ej: En una llamada a procedimiento, si el parámetro es una expresión, esta no se evaluará hasta que sea estrictamente necesario.

  Considerando la llamada a la función sumar(2+3), con la siguiente definición:

```
sumar(int x)
int y;
{
    y = x+10
}
```

  El resultado se obtiene sustituyendo x por la expresión 2+3, la cual corresponde al valor de x como parámetro de la función (NO el valor 5).

- *CORTOCIRCUITO*: Se realiza sobre expresiones booleanas o lógicas. Tiene lugar cuando se determina el valor de la expresión, sin evaluar todas las sub-expresiones.

  La evaluación con cortocircuito de expresiones muestra los problemas de permitir efectos colaterales en expresiones.
  →Por ejemplo, supongamos que se utiliza evaluación con cortocircuito en una expresión, y parte de la expresión que contiene un efecto colateral no es evaluada; entonces, el efecto colateral solamente ocurrirá en evaluaciones completas de toda la expresión. Si la correctitud del programa depende del efecto colateral, la evaluación con cortocircuito puede resultar un grave error.

  Ej: *Ada* permite al programador utilizar esta forma de evaluación en sus expresiones *and then-or else*
  En los lenguajes *C-based* && y || evalúan las expresiones con cortocircuito

  La inclusión de los operadores con cortocircuito y los operadores ordinarios es el mejor diseño, ya que provee al programador mayor flexibilidad para elegir la evaluación con cortocircuito para una o todas las expresiones booleanas.

- *PEREZOSA*: Esta forma de evaluación es la que generalmente es adoptada por los *lenguajes funcionales*. Sigue las siguientes reglas:
  1. Pospone la evaluación de una expresión hasta que es necesaria (se evalúa cuando el evaluador es forzado a producir el valor de la expresión).
  2. Elimina la re-evaluación de la misma expresión más de una vez.

  Ej: En *ML* fun g(x,y,z) = if x<2 then y+3 else z+6

  → sólo el valor de *y* o el de *z* son requeridos para el cálculo del resultado, pero no ambos

  En *C* a==0 ? 0 : b/a

7.7 SENTENCIAS DE ASIGNACIÓN

La sentencia de asignación provee el mecanismo por el cual el programador puede cambiar dinámicamente las ligaduras de valores a variables. Es uno de los constructores centrales en los lenguajes imperativos.

Asignación Simple

La sintaxis general tiene la forma: *<variable destino> <operador de asignación> <expresión>*
El operador de asignación varía de acuerdo al lenguaje de programación.

Ejs: A := B (Ada, Pascal)
A = B (C, C++, Java – El operador es considerado binario y puede aparecer en expresiones)
MOVE B TO A (Cobol)
A <- B (APL)

Obs: En los lenguajes que se usa = para la asignación, se usa == para el operador relacional (p/ evitar conflictos).

Asignación como Expresión

La operación de asignación puede pensarse como una *instrucción* (Pascal, Ada, Fortran) que no retorna ningún valor, o como una *expresión* (C, Java, APL, ML).

La instrucción de asignación produce un resultado, el cual es el mismo que el valor asignado a un objetivo. Esto puede ser usado luego como una expresión y como un operando en otras expresiones. La asignación debe estar parentizada, y su precedencia es menor que la de los operadores relacionales.
Desventajas:
- Genera otro tipo de efectos colaterales
- Puede dificultar la legibilidad de expresiones
- Dificulta la detección de errores del compilador

Ej: C = (B=2) → Se le asigna el valor 2 a B, y el resultado de la expresión B=2, es decir 2, se le asigna a C.

Se puede definir la operación de asignación como sigue:

1. Computar el valor del lado izquierdo *(l-value = locación de memoria de un objeto de dato)*
2. Computar el valor de la expresión del lado derecho *(r-value = valor de un objeto de dato)*
3. Asignar el valor computado del lado derecho al computado como objeto de dato del lado derecho
4. Retornar el valor computado como valor del lado derecho como resultado de la asignación

La semántica de la operación de asignación puede pensarse como:
- Copia de la referencia
- Dereferenciamiento implícito

La asignación puede en algunas circunstancias darse en forma *implícita*. Generalmente esto sucede en la inicialización en las declaraciones (int a – float b), o en el pasaje de parámetros.

Asignación con Objetivos Condicionales
C++, Java y C# permiten objetivos condicionales en sentencias de asignación.

Ej: flag ? cont1 : cont2 =0 equivale a if (flag) then cont1=0 else cont2=0

Asignación con Múltiples Objetivos
Permite asignar el valor de la expresión a múltiples variables.
Ej: sum,total =0.

Operadores de Asignación Compuestos
Un operador de asignación compuesto es un método rápido de especificar una forma normalmente necesitada de asignación. Los operadores de asignación cmopuestos fueron introducidos por ALGOL68 siendo después adoptados en una forma ligeramente diferente por los lenguajes basados en C.

Ej: sum+= valor equivale a sum = sum + valor

Operadores de Asignación Unarios
Los lenguajes basados en C incluyen dos operadores aritméticos unarios especiales, que son abreviaciones de asignaciones. Combinan operaciones de incremento y decremento con asignaciones.
Son usados en expresiones, o en forma independiente en sentencias de asignación de un único operador.
Pueden aparecer como operadores *prefijos* (preceden a los operandos) o *posfijos* (siguen a los operandos).

Ej: sum = ++ cont → cont= cont+1 ; sum = cont; *(prefijo)*
Sum = cont ++ → sum = cont; cont= cont+1 ; *(posfijo)*

Cuando dos operadores unarios se aplican al mismo operando, la asociación es de derecha a izquierda.

7.8 ASIGNACIÓN EN MODO MIXTO
Frecuentemente, las asignaciones suelen ser en modo mixto. Algunos lenguajes (Fortran, C, C++) usan reglas de coerción para las asignaciones en modo mixto, que son similares a las usadas para expresiones en modo mixto. Es decir, muchas de las posibles mezclas de tipos son legales, con la coerción debidamente aplicada.

Ej: Pascal, Java, C, C++ permiten asignación en modo mixto si la coerción es expansora (asignar int a float).
Ada y Modula2 no permiten asignación en modo mixto.

En todos los lenguajes donde se permite la asignación en modo mixto, la coerción toma lugar sólo después que la expresión del lado derecho ha sido evaluada. Una alternativa podría ser convertir todos los operandos del lado derecho al tipo del objetivo de la asignación, antes de la evaluación.

Ej:
```
int a,b;
float c;
…
c = a/b;
```
Dado que c es de tipo float, los valores de a y b pueden ser convertidos a float antes de la división, lo que puede producir un valor distinto para c, que si la coerción hubiese sido demorada hasta luego de la evaluación (ej: si a = 2 y b=3).

CAPÍTULO III [Ghezzi] – VI [Sebesta] – Tipos de Datos y Sistemas de Tipos

INTRODUCCIÓN

La *abstracción de datos en los lenguajes de programación* trata con las componentes de un programa que son sujeto de computación. Está basada en las propiedades de los objetos de dato y las operaciones de dichos objetos.

- *Tipos de Datos*
- *Sistemas de Tipos*: Conjunto de reglas usadas por un lenguaje para estructurar y organizar su colección de tipos.
- *Encapsulamiento y Abstracción*: El uso de un tipo está separado de la representación y conjunto de operaciones sobre los valores del tipo.

TIPOS DE DATOS

Origen

El concepto de *tipo* surge inicialmente para mejorar la seguridad, porque permite establecer chequeos. El soporte de tipos contribuye a mejorar algunos de los criterios de calidad como seguridad, legibilidad, reusabilidad y extensibilidad.

Evolución

- Los primeros lenguajes contaban con pocos tipos muy cercanos al HW.
- Con la aparición de lenguajes orientados a los diferentes dominios de aplicación, aparecen tipos específicos.
- Tipos de propósito general (poca uniformidad).
- Aumentan los mecanismos de abstracción (pocos mecanismos pero generales y combinables).

Concepto

Los lenguajes de programación organizan los datos a través del concepto de *tipo*. Los tipos son usados como una forma de clasificar los datos de acuerdo a diferentes categorías. Los datos pertenecientes a un determinado tipo también comparten comportamiento semántico.

Definición general: Un *tipo de dato* define un patrón que permite factorizar las propiedades y comportamiento de un conjunto de entidades.
Definición más precisa: Un *tipo de dato* es un conjunto de valores y un conjunto de operaciones que pueden ser usadas para manipularlos.

Un tipo permite separar la definición de su uso. De esta manera se protege a los datos de accesos ilegales.

Los LP pueden especificar sus facilidades para soportar el concepto de tipo en:

- *Diseño del Lenguaje*
- *Implementación del Lenguaje*: Los tipos determinan un mapeo entre el área de almacenamiento y los valores.
- *Diseño de la Aplicación*: Los tipos particionan el dominio de valores de los datos en clases de equivalencia con atributos y operaciones comunes.
- *Implementación de la Aplicación*

- *Verificación de la Aplicación*: Los tipos determinan el comportamiento que sus instancias deben satisfacer.
- *Evolución de la Aplicación*: Los tipos son especificaciones de comportamiento y pueden ser compuestos en forma incremental para definir nuevas especificaciones.
- *Chequeo de la Aplicación*: Los tipos imponen restricciones sintácticas sobre las expresiones, de manera que sólo algunas combinaciones de operaciones y operandos resultan compatibles.
- *Ejecución de la Aplicación*: Los tipos son armaduras que protegen a la información de interpretaciones incorrectas.

Definición
La *sintaxis del lenguaje de programación* determina la forma de las declaraciones de tipo.
Las *declaraciones de tipo* permiten al compilador:

- Establecer ligaduras
- Ocultar la representación interna de los objetos del tipo
- Realizar chequeos (verificar correcto uso de variables)
- Elegir una representación e implementación adecuadas
- Manipulación del medio de almacenamiento
- Resolver problemas de sobrecarga de operaciones

Para *definir* un tipo de datos, es necesario brindar su *especificación* y su *implementación*.

La ESPECIFICACIÓN del tipo incluye:

- *Atributos*: Cualidades que distinguen a los objetos de datos de este tipo.
- *Valores* que cualquier objeto de datos de este tipo puede llegar a tomar (rango de valores válido).
- *Operaciones* (*): Definen las posibles manipulaciones que puede sufrir un objeto de datos de este tipo.

En la IMPLEMENTACIÓN se determina:

- La representación en el medio de almacenamiento
- Los algoritmos para manipular la representación

(*) Las *operaciones* pueden ser p*rimitivas* (definidas por el LP) o d*efinidas por el usuario* (en forma de subprogramas o declaración de métodos en clases).

La operación se define en términos de:
- *El mapeo* (o algoritmo) que especifica cómo se computa el resultado a partir de un conjunto de argumentos.
- *El dominio:* conjunto posible de argumentos de entrada.
- *El rango:* conjunto posible de resultados que puede producir.

A veces es difícil especificar la operación como una función matemática. Existen *factores que oscurecen su definición*:

- *Dominio Impreciso:* La operación está definida sobre un dominio, pero puede estar indefinida para ciertos valores pertenecientes a él. Puede resultar extremadamente difícil de determinar, por ejemplo, si queremos evitar los valores que provocan overflow o underflow de la representación.

- *Argumentos Implícitos:* La operación puede acceder a otros argumentos implícitos a través del uso de variables globales o referencias a identificadores no locales.

- *Sensibilidad a la Historia:* Una operación puede modificar su propia estructura interna, ya sea por retener datos locales entre ejecuciones o modificando su propio código (esto último poco frecuente, pero posible – Ej: LISP). El resultado producido para un conjunto de argumentos deja de depender sólo de ellos, para pasar a depender de la historia de ejecución previa (Ej: Generador de números aleatorios – uso de semilla).

- *Efectos Colaterales:* Una operación puede retornar un valor como resultado explícito, pero también generar modificaciones visibles en otros objetos de datos almacenados (tanto definidos por el programador como por el sistema). Esta característica es necesaria, particularmente para modificar estructuras de datos, pero hace difícil la exacta especificación del rango de la operación.

CLASIFICACIÓN DE TIPOS DE DATOS
Existen diversos criterios de clasificación para los Tipos de Datos.

De acuerdo a quién los define:
- *Predefinidos por el lenguaje*
- *Definidos por el programador*

De acuerdo a su estructura:
- ***Elementales/Simples***
- ***Estructurados/Compuestos***
- ***Recursivos***

De acuerdo a la naturaleza de sus valores:
- *Ordinales*
- *No ordinales*

Tipos de Datos Elementales (Primitivos/Simples)
Son aquellos que no están definidos en términos de otros tipos. Sus valores son atómicos y no pueden descomponerse en elementos más simples. Algunos son un simple reflejo del HW *(integer)* y otros sólo requieren un poco de soporte adicional para su implementación.
Obs: generalmente los tipos de datos primitivos coinciden con los predefinidos, pero no siempre (Ej: lenguajes que tienen el tipo String predefinido, pero no es elemental → se descompone en chars).

Mecanismos Predefinidos
- Numéricos
- Caracteres
- Booleanos

Mecanismos Para Definir Nuevos Tipos
- Enumerados
- Subrangos

Mecanismos Predefinidos (Tipos Elementales Predefinidos)

- **NUMÉRICOS**

- *Entero*: Conjunto finito y ordenado, con atributos *máximo* y *mínimo*.
  Un valor entero es representado en la computadora como una cadena de bits. Es soportado directamente por HW.

Operaciones
- Aritméticas
  - Binaria: int x int → int
  - Unaria: int → int
- Relacionales → Binaria: int x int → bool
- Asignación → Binaria: int x int → int

- *Punto Flotante*: Conjunto finito y ordenado, con atributos *rango* y *precisión*. Provee una aproximación para los números reales (por limitaciones de almacenamiento sólo es una aproximación).
  Los valores son almacenados en binario (notación científica: fracciones y exponentes), lo cual limita la precisión debido a la representación. Las operaciones aritméticas también provocan pérdida de precisión. Ej: Float 4 bytes – Double 8 bytes

- *Decimal*: Pensado para aplicaciones de negocios. Brinda una representación de números reales con un número fijo de decimales. La desventaja es que el rango de valores se encuentra restringido (no se permiten exponentes).

- **CARACTERES**: Conjunto finito y ordenado. Se almacena a través de una codificación matemática (Código ASCII, Unicode).

Operaciones
- Sucesor y Predecesor → Unaria: char →char
- Ordinal → Unaria: char →char
- Relacionales → Binaria: char x char → char
- Asignación → Binaria: char x char → char

- **BOOLEAN**: Sólo dos valores posibles: verdadero y falso. Puede ser representado utilizando un solo bit. Debido a la dificultad de acceder a un solo bit, se almacenan en la menor celda de memoria direccionable (byte).
  Obs: En C se implementa con números enteros: 0 = false, y cualquier otro número = true

Operaciones
- Sucesor y Predecesor → Unaria: char →char
- Lógicas
  - Binaria: int x int → int
  - Unaria: int → int
- Relacionales → Binaria: char x char → char
- Asignación → Binaria: char x char → char

Mecanismos Para Definir Nuevos Tipos (Elementales)

- **ENUMERADOS**: Formados por constantes simbólicas, las cuales se enumeran en su definición. Estas constantes conforman todos los valores posibles.

Operaciones:
- Sucesor y Predecesor → Unaria: char → char
- Ordinal → Unaria: char → char
- Relacionales → Binaria: char x char → char
- Asignación → Binaria: char x char → char

Los tipos enumerados se implementan como enteros.

Aspectos de diseño:
- Está permitido que una constante aparezca en más de una definición de tipos? Si lo está, cómo se chequea el tipo de la ocurrencia de una constante? (sobrecarga de identificadores) Ej: Pascal no permite, C y C++ sí.
- Los valores enumerados se pueden coercionar a enteros? Ej: C++ lo permite (hoy = lun; i=1; hoy++; i=hoy), pero C# y Ada no (mejor soporte para enumerados).
- Otros tipos pueden ser coercionados a un tipo enumerado?

Evaluación:
+ Incrementa la legibilidad – Fáciles de reconocer (Ej: no es necesario codificar colores como números)
+ Incrementa la confiabilidad (El compilador puede chequear operaciones, rango de valores, etc.)

- **SUBRANGOS**: Sub-secuencia contigua de un tipo ordinal

Operaciones:
- Sucesor y Predecesor → Unaria: char → char
- Relacionales → Binaria: char x char → char
- Asignación → Binaria: char x char → char

Los tipos subrangos son implementados como sus tipos padre con código insertado (por el compilador) para restringir la asignación sólo a variables de tipo subrango.

Evaluación:
+ Incrementa la legibilidad (hace más claro al que lector sobre variables de tipo subrango sólo se puede almacenar cierto rango de valores).
+ Incrementa la confiabilidad (la asignación de un valor fuera de los especificados en el subrango es detectado como un error).

Ej: Pascal →
```
Type días={lun,mar,mier,jue,vie,sab,dom}
dias_laborales = lun..vie
Type naturales = 1..Maxint;
```

Ada → Los subrangos son *subtipos*

Tipos de Datos Estructurados (Compuestos)
Las instancias de un tipo de datos estructurado son *estructuras de datos.* Una estructura de datos es un objeto de dato construido a partir de otros más simples. Es un conglomerado de datos más simples.

Estos tipos se definen también a través de:
SINTAXIS
ESPECIFICACIÓN
- *Atributos:*
    - Número de componentes
    - Tipo de cada componente
    - Nombres para las operaciones de selección
    - Máximo número de componentes
    - Organización de las componentes
- *Valores*
- *Operaciones:*
    - Selección de componentes
    - Operaciones sobre la estructura completa
    - Inserción y eliminación
    - Creación y destrucción de estructuras

IMPLEMENTACIÓN

El *chequeo de tipos* para los tipos estructurados es más complicado que para los tipos elementales, e involucra dos cuestiones fundamentales:

- Existencia de la componente seleccionada
- Tipo de la componente seleccionada

} Dependiendo de cuál sea el tipo estructurado, estas cuestiones son más o menos fáciles de resolver

Los ***constructores para tipos estructurados*** más frecuentes son:

- Mapeo Finito
- Producto Cartesiano
- Unión y Unión Discriminada
- Secuencia

- **MAPEO FINITO**: Un *mapeo* es una función que asocia valores de un conjunto (dominio) a valores de otro conjunto (rango). Para que se trate de un *mapeo finito*, la función debe definirse de un conjunto finito de valores de un dominio de tipo D en valores de un rango de tipo R.

  La función M: D → R puede definirse en los lenguajes de programación como:
  - *Definición Intensional*
  - *Definición Extensional:* El mapeo se define utilizando el constructor de arreglos. El constructor de tipo asocia a cada índice del arreglo el valor almacenado en esa posición del arreglo. El rango de índices es el dominio del mapeo, y los valores almacenados en el arreglo (asociados al tipo) conforman el rango.

  Sintaxis: El mapeo finito es representado como una colección de elementos de dato homogéneos (valores del mismo tipo) que son identificados por su posición relativa al primer elemento.
  El constructor provisto se conoce como *arreglo* (ED homogénea, lineal y ordenada), cuyos índices (explícitos o implícitos) conforman el dominio. La sintaxis para el acceso a los componentes es *Arreglo[indice]* o *Arreglo(índice)* ← Sobrecarga de los paréntesis.

  Diseño: Tipos legales para subíndices? Chequear rango de subíndices al hacer referencias? Tiempo de ligadura de rangos de subíndices? Cuántas dimensiones son permitidas? Cuando toma lugar la asignación de almacenamiento?

  Especificación – Atributos:
  - Cantidad máxima de dimensiones
  - Dominio del mapeo – Puede establecerse en forma:
    - *Estática:* Los índices son ligados estáticamente (el rango) y la alocación de almacenamiento es estática. Ej: Fortran, C, C++
      + Eficiente (no hay alocación dinámica)

    - *Semi-Estática*: Los índices son ligados estáticamente pero la alocación del almacenamiento se realiza en la declaración. Ej: Pascal, Ada, C, C++
      + Eficiente en el manejo de espacio

    - *Semi-Dinámica*: Los índices son ligados dinámicamente (en pila) y la alocación se resuelve en ejecución. Ej: Ada
      + Flexible (el tipo del arreglo no es necesario hasta que se vaya a utilizar)

    - *Dinámica sin Modificación:* Los índices son ligados dinámicamente pero quedan fijos luego de la alocación (en heap). Similar al anterior, pero con uso de heap. Ej: C, C++, Java, C#

    - *Dinámica con Modificación:* Los índices y la alocación de almacenamiento son ligados dinámicamente y pueden cambiar cuando se quiera. Ej: C#, Perl, Javascript
      + Muy flexible (los arreglos pueden crecer y achicarse durante la ejecución)

  - Rango del mapeo
  - Inicialización: Algunos lenguajes permiten la inicialización del mapeo finito al momento en el que se resuelve la ligadura al almacenamiento.
    Ej: int A[] = {4,7,234,90}
    char Nombre[] = “pepito”

Especificación – Operaciones: Es necesario contar con operaciones para:

- Seleccionar o subindicar
- Asignar
- Slicing – Un *slice* de un arreglo es una subestructura de ese arreglo. Si A es una matriz, una posible división (slice) sería una fila o columna de la matriz.
- Creación y destrucción
- Aritmética entre vectores

{ Ejemplos – Mapeo Finito }

Pascal – *Arreglos semi-estáticos* →

```
const n=10;
type Vector = array[1..n] of integer;
```

Ada – *Arreglos semi-estáticos y semi-dinámicos* →

```
m = integer:=.....;
type Vector is array (<integer range>) of float;
subtype Vec is Vector(1..20);

var
    a: Vector(1..10);
    b: Vector(1..m);
    c: Vec;
```

No se puede crear una instancia de Vector sin indicar su subrango. Por ejemplo, var v: Vector; ←

MAPEO FINITO – RANGO HETEROGÉNEO → Algunos lenguajes brindan facilidades para permitir que el dominio y el rango del mapeo, o sólo el rango, sean heterogéneos. Se trata de lenguajes dinámicamente tipados como SNOBOL4.

Ej:

```
A= ARRAY()
A<1> = 'Ana'
A<2> = 13
```

→ Rango Heterogéneo

```
T= TABLE()
T<'hijo'> = 'Juan'
T<'edad'>= 3
T<3> = 4.5
```

→ Rango y Dominio Heterogéneos

(Para el tipo table SNOBOL4 no requiere que el tipo dominio sea un subrango ordenado de un tipo discreto)

MAPEO FINITO – ARREGLOS ASOCIATIVOS → Son colecciones no ordenadas de elementos de datos que son indexadas por un número igual de valores llamados "llaves". No hay relación de orden entre los elementos. Este tipo de arreglos es provisto por el lenguaje Perl.

Ej:

```
%salarios = ("Juan" => 1500, "Luis" => 1200)
$salarios{"Luis"} = 1600
$salarios{"Marta"} = 1100
delete $salarios{"Juan"}
```

→ Los nombres de los arreglos comienzan con % y los literales son limitados por paréntesis

RECTANGULAR ARRAYS → Arreglo multidimensional en el cual todas las filas tienen la misma cantidad de elementos, todas las columnas tienen la misma cantidad de elementos, etc.

JAGGED ARRAYS → Arreglo en el cual las longitudes de las filas pueden ser distintas. También se aplica esto a las columnas u otras dimensiones.

- **PRODUCTO CARTESIANO:** El producto cartesiano de n conjuntos A_1, A_2, …, A_n (denotado $A_1 \times A_2 \times \dots \times A_n$) es un conjunto cuyos elementos están ordenados en n-tuplas (a_1, a_2, …, a_n) donde cada a_k pertenece al conjunto A_k.

  Ej: Un nodo en un grafo puede describirse con una letra (*caracter*) y el peso del nodo en el grafo (*entero*). El nodo puede entonces pensarse como un producto cartesiano de los tipos *caracter* y *entero*.

  <u>*Sintaxis*</u>: Los lenguajes ven al producto cartesiano como una composición de un número entero de nombres simbólicos (campos).
  El constructor provisto se conoce como *registro*/*tuplas*/*struct*, el cual representa una colección de datos heterogéneos que pueden ser accedidos individualmente según su nombre.

  <u>*Diseño*</u>: Cómo se referencian las componentes? → Los campos de un elemento de un producto cartesiano son seleccionados especificando sus nombres mediante la notación sintáctica correspondiente (La mayoría de los LP usan la notación de "." para acceder a los campos).
  Las referencias son totalmente *calificadas* o *elípticas*? → Una referencia totalmente calificada al campo de un registro es una en la cual se especifica todo el "path" hasta el acceso del campo. Una elíptica permite ignorar algún "paso" intermedio.
  Obs: En este tipo de datos, la representación interna no está oculta a los que utilizan objetos del tipo.

  <u>*Especificación – Atributos*</u>:
  - Número de conjuntos que participan en el producto cartesiano
  - Tipo de cada uno de los conjuntos intervinientes

  <u>*Especificación – Operaciones*</u>: Sólo se proveen las operaciones genéricas asociadas al tipo, para manipular la representación.
  - Selección de campos
  - Asignación de campos

  { Ejemplos – Producto Cartesiano }

  Pascal – *Record* →
  ```
  type T = record
             T1: TipoBase1;
             …
             Tn: TipoBasen;
           end;
  ```

  ML – *Tuplas* → $T = TipoBase_1 \times \dots \times TipoBase_n$

  ML – *Registros* → $T = \{ T_1: TipoBase_1, \dots , T_n: TipoBase_n \}$

C++ y C# – *Struct* → El constructor *struct* en estos lenguajes es más un mecanismo de encapsulamiento que una estructura de datos. Tiene una semántica similar al concepto de clase, su diferencia está en la alocación de memoria.

- **UNIÓN** (Registro Variante)**:** La *unión* o *unión discriminada* es muy similar al producto cartesiano, sólo que en este caso, los campos son mutuamente excluyentes. El *producto cartesiano* expresa *conjunción*, mientras que la *unión* expresa *disyunción*.

  Permite especificar una selección entre alternativas excluyentes, en general, de tipos diferentes $A_1 + A_2 + \ldots + A_n$, donde cada A_i (alternativa) es llamada *variante*.

  El problema de determinar cuál de los campos de la unión es el actual (y de qué tipo), puede solucionarse utilizando un campo especial: *discriminante*, que indica el valor del tipo actual del variante.
  → Si la unión utiliza discriminante se llama *unión discriminada.*

  Diseño: Debe requerirse el chequeo de tipo? (chequeo dinámico)
  Deben las uniones estar embebida en los registros?

  Especificación – Operaciones:
  - Chequeo del discriminante
  - Selección de los campos variantes

  Riesgos → Existe la posibilidad de acceder en forma errónea a la estructura.
  Debe efectuarse el chequeo de tipo en tiempo de ejecución tanto para los elementos de uniones o uniones discriminadas. Incluso en la unión discriminada, es posible manipular desde el programa un elemento como miembro de un tipo T, aunque sea de tipo S. La unión discriminada, sin embargo, es potencialmente más segura dado que permite al programador considerar explícitamente el campo discriminante antes de aplicar una operación a un elemento.
  Otro problema es que puede cambiarse el discriminante sin cambiar el variante, dejando el tipo del discriminante inconsistente con el del elemento en el variante.

  { Ejemplos – Uniones }

  Java y C# no proveen constructores para definir unión.

  Pascal – *Registro Variante* →

```
type
    tCargo = (ayudante, profesor);

    tDocente =      record
                        nombre: string;
                        case cargo:tCargo of
                            ayudante: (nivel: (A,B));
                            profesor: (grado: (adjunto, asociado); dedicacion: (simple, exclusiva));
                    end;
```

- **CONJUNTO POTENCIA:** El tipo *conjunto potencia* se define como un conjunto de todos los subconjuntos de elementos de un determinado tipo T. El tipo T generalmente es llamado el *tipo base* del conjunto potencia.

$$\text{powerset}(T) = \{S: S \subseteq T\}$$

Las variables de tipo *powerset(T)* representan conjuntos. Un tipo *conjunto* es aquel cuyas variables pueden almacenar colecciones no ordenadas de distintos valores de un tipo T ordinal.

Especificación – Operaciones: Las mismas operaciones soportadas por el tipo *conjunto*.

- Unión
- Intersección
- Diferencia
- Miembro
- Inclusión

{ Ejemplo – Conjunto Potencia }

Pascal →

```
type    colores = (rojo, azul, amarillo, blanco, negro);
        conjuntoDeColores = set of colores;

var
        conjunto1, conjunto2: conjuntoDeColores;

conjunto1 := [rojo,azul,blanco];
conjunto2 := [blanco,negro];
```

- **SECUENCIA:** La *secuencia* es una sucesión de elementos de un mismo tipo base.
Una de las propiedades importantes del constructor de secuencia, es que representan objetos de datos de tamaño arbitrario (el número de elementos del tipo base es arbitrario).

Las dos alternativas más comunes en los lenguajes de programación son:

- *Cadenas de Caracteres*
- *Archivos Secuenciales*

Otra alternativa utilizada para representar secuencias son los Arreglos y las Listas Recursivas

→ Si el tamaño de la secuencia no cambia dinámicamente, los arreglos proveen la mejor solución. Si el tamaño de la secuencia debe cambiar durante la ejecución del programa, los arreglos flexibles o listas deben ser utilizados. (Ej: Lisp y Prolog proveen listas como primitivas del lenguaje).

CADENA DE CARACTERES: Es un tipo en el cual sus valores son una secuencia de caracteres.

Alternativas sobre la Longitud de las Cadenas de Caracteres:

- Longitud Fija: Están siempre completas, se rellenan con blancos (Ej: Java – clase *String*, Ada, COBOL).
- Longitud Variable con Máximo Conocido: Pueden tener cualquier cantidad de caracteres entre 0 y el máximo establecido (Ej: C, C++, Ada).
- Longitud Variable: Sin restricciones. Requiere la sobrecarga de asignar y liberar almacenamiento dinámicamente, para proveer máxima flexibilidad (Ej: SNOBOL4, Perl, Javascript, Ada).

Sintaxis: La mayoría de los lenguajes lo provee como tipo predefinido. Hay lenguajes proveen una colección de operaciones para strings a través de librerías (Ej: C y C++ - string.h). Otros los aportan a través de la definición del programador como un arreglo de caracteres (Ej: C y C++ utilizan arreglos de caracteres para almacenar strings).

Especificación – Operaciones:

- Asignación
- Entrada – Salida
- Longitud, Concatenación, Sub-cadena
- Relacionales

ARCHIVOS SECUENCIALES: Estructura de datos lineal, potencialmente infinita, con componentes heterogéneas que sólo pueden ser accedidas en orden estrictamente secuencial. Modificaciones a la secuencia de elementos pueden ser realizadas añadiendo un nuevo valor al final del archivo (append).

Especificación – Operaciones:

- Abrir y Cerrar
- Leer y Escribir
- Fin de Archivo

ARCHIVO INDEXADO → Un *archivo indexado* es una estructura de datos lineal, con componentes homogéneas, que pueden ser accedidas en cualquier orden según una *clave* mantenida en un *índice ordenado*.

ARCHIVO DE ACCESO DIRECTO → Un *archivo de acceso directo* es una estructura de datos lineal, con componentes homogéneas, que pueden ser accedidas en cualquier orden según una *clave* mantenida en un *índice*.

Ej: Ada provee soporte para archivos secuenciales, indexados, y de acceso directo, a través de librerías estándar.

Tipos de Datos Recursivos

La *recursión* es un mecanismo de estructuración para definir agregaciones de datos cuyo tamaño puede crecer arbitrariamente, y cuya estructura puede tener complejidad arbitraria.
Un *tipo de dato* T se dice *recursivo* si es una estructura cuyos componentes son también de tipo T.

Alternativas de Implementación para Tipos de Datos Recursivos:

- *Constructores Genéricos*
- *Especificación Directa*
- *Punteros*

Los LP convencionales soportan la implementación de tipos de datos recursivos a través de punteros. Los lenguajes funcionales proveen una forma más abstracta para definir y manipular tipos recursivos, que oculta la representación basada en punteros subyacente.

- **CONSTRUCTORES GENÉRICOS:** int_list = list(int)
  int_tree = bin_tree(int)

- **ESPECIFICACIÓN DIRECTA:** Se define utilizando producto cartesiano. Así, las definiciones anteriores se ven de la siguientes manera:

  int_list = nil ∪ (int x int_list)
  int_tree = nil ∪ (int x int_tree x int_tree) → AB = (Elem, HI, HD)

  { Ejemplo – ML }

  datatype listaEnteros = nil|constante of int * listaEnteros

  datatype arbolEnteros = hoja of int | rama of arbolEnteros * arbolEnteros → No permite árbol nulo

  Ejs. listaEnteros:
  nil
  constante(11, nil)
  constante(2, constante(3, constante(5,nil)))

  Ejs. arbolEnteros:
  hoja 11
  rama(hoja 9, hoja 15)
  rama(rama(rama(hoja 5, hoja 7), hoja 9), rama(hoja 12, hoja 15))

- **PUNTEROS:** Un dato de tipo *puntero* contiene una referencia a la dirección de memoria de otro dato, o contiene la dirección nula.

  Diseño:
  - Alcance y tiempo de vida de la variable de tipo puntero
  - Tiempo de vida de la variable apuntada o referenciada
  - Restricciones de tipo (tienen las restricciones del tipo al que apuntan?)
  - Operaciones sobre variables de tipo puntero (son usados para manejo dinámico de almacenamiento?)
  - Incorporación de un tipo referencia (el LP debería soportar además de punteros el tipo referencia?)

  Especificación – Operaciones:
  - Creación de la variable apuntada
  - Destrucción
  - Asignación: Setea un valor útil a una variable de tipo puntero
  - Dereferenciamiento/Direccionamiento indirecto: Toma una variable a través de un nivel de indirección (el puntero se interpreta como una referencia al valor en la celda de memoria cuya dirección está en la celda de memoria a la cual la variable está ligada).
  - Aritmética de punteros

  Ventajas:
  + Provee direccionamiento indirecto
  + Provee una forma de manejo dinámico de memoria
  + Provee facilidades para trabajar con estructuras dinámicas

  Problemas:
  - Punteros colgantes/referencias colgadas (Dos punteros referenciaban a la misma variable, se liberó la memoria desde una variable, y el puntero de la otra variable quedó colgado. Si la celda de memoria fue asignada luego a una variable de tipo incompatible, y puede haber errores al efectuar chequeo de tipos).
  - Basura (Se retienen celdas de memoria marcadas como ocupadas pero sin referencias – El contenido de la celda no está accesible para el programa de usuario pero la celda de memoria no ha sido liberada).

  { Ejemplos – Punteros }

  C y C++ → - Extremadamente flexibles (pueden ser utilizados como las direcciones en los lenguajes ensambladores), por lo que deben ser usados cuidadosamente.
  - Son usados para el manejo de almacenamiento dinámico y direccionamiento.
  - La aritmética de punteros es posible.
  - Dereferenciamiento y direccionamiento explícito.
  - Pueden apuntar a locaciones anónimas y nombradas.
  - Notación para las operaciones: * = Dereferenciamiento & = dirección de la variable

  Ej: int *p;
  int list[10];
  p = list; ← Asigna la dirección de *list* a *p*, ya que un arreglo sin subíndice se interpreta como su dirección base.
  *(*p*+1) es igual a *list*[1]

- **REFERENCIAS:** C++ incluye una clase especial de tipo puntero llamada *tipo referencia*, que es utilizado principalmente para los parámetros formales.
  Una variable de tipo referencia en C++ es un puntero constante que está siempre implícitamente dereferenciado. Dado que es una constante, debe estar inicializada con la dirección de alguna variable en su definición y luego de la inicialización no puede cambiarse la referencia (no puede referenciar a otra variable).
  Las variables referencia son especificadas en las definiciones con un & precediendo su nombre.

  Java extiende las variables por referencia de C++ y permite que estas reemplacen totalmente a los punteros. → Las referencias tienen que ver con las instancias a las que se llama.

  C# incluye ambos: las referencias de Java y los punteros de C++.

SISTEMAS DE TIPOS

Un *sistema de tipos* es un conjunto de reglas que estructuran y organizan una colección de tipos de un LP. El objetivo de un sistema de tipos es lograr que los programas sean tan seguros como sea posible.
Restringe el conjunto de programas válidos de un lenguaje. Si las restricciones son excesivas muy pocos programas son válidos.

Con frecuencia, la *seguridad* compromete la *flexibilidad* del lenguaje. Cuál de las dos características es la preponderante o buscada por el sistema depende de a qué tipo de desarrollos se apunte con el LP.
Los lenguajes orientados al desarrollo de aplicaciones complejas buscan *mayor seguridad* y se caracterizan por tener un *sistema de tipos estricto*.
Los lenguajes script orientados al desarrollo rápido, buscan *mayor flexibilidad* de modo que se caracterizan por tener un *sistema de tipos relajado*.

***Especificación del Sistema de Tipos*:**

- Tipo y tiempo de chequeo
- Reglas de conversión
- Reglas de equivalencia
- Reglas de inferencia de tipo
- Nivel de polimorfismo del lenguaje

- **TIPO Y TIEMPO DE CHEQUEO:** El chequeo de tipo consiste en verificar que el tipo de una entidad corresponda al tipo esperado por el contexto. Si no coinciden, puede tratarse de un *error*, o se aplican *reglas de coerción* o *reglas de equivalencia*.

  Errores de Tipo
  Errores en el Uso del Lenguaje: Errores sintácticos o semánticos (Ej: Acceso ilegal inadvertido).
  Errores en la Aplicación: Desviaciones del comportamiento del programa con respecto a la especificación. (Ej: Error chequeado y notificado por el lenguaje, que permite su depuración).

  Idealmente en el sistema de tipos se establecen restricciones que aseguran que no se van a producir errores de tipos en la *aplicación*, gracias a que ya han sido detectados en el *lenguaje.*

Tipos de Ligadura
Tipado Estático: Ligaduras en compilación.
Tipado Dinámico: Ligaduras en ejecución, no es seguro.
Tipado Seguro: No es estático, ni inseguro.

Tiempo de Ligadura
Definición 1: El *tipado* es *estático* si cada *entidad* queda ligada a su tipo durante la *compilación*, sin necesidad de ejecutar el programa (Ghezzi – 138).
Definición 2: El *tipado* es *estático* si cada *variable* queda ligada a su tipo durante la *compilación*, sin necesidad de ejecutar el programa (Ghezzi – 54).

(Un sistema de tipos estático requiere que el tipo de todas las expresiones en el programa sea conocido en tiempo de compilación).

Chequeo de Tipos
El *chequeo dinámico* requiere que el programa sea ejecutado sobre una entrada de datos. El *chequeo estático* no lo requiere.
En general, si el chequeo puede realizarse estáticamente, es preferible hacerlo en lugar de retrasar el chequeo hasta tiempo de ejecución. Si bien es preferible es chequeo estático ante el dinámico, no cubre todos los errores del lenguaje (Hay ciertos errores que no pueden ser detectados durante compilación – Ej: división por cero).

Lenguajes Fuertemente Tipados
Definición 1: Un *lenguaje* se dice *fuertemente tipado* si el sistema de tipos impone restricciones que aseguran que no se producirán errores de tipo en tiempo de ejecución. El compilador puede garantizar la ausencia de errores de tipo en los programas (Ghezzi – 138).

Definición 2: Un *lenguaje* se dice *fuertemente tipado* si todos los errores de tipo se detectan (Sebesta).
→ En esta concepción, la intención es evitar los errores de *aplicación* y son tolerados los errores del *lenguaje*, detectados tan pronto como sea posible.

Observación - Inconsistencias:
- Un lenguaje con tipado estático es fuertemente tipado (Ghezzi – 139). Hay lenguajes fuertemente tipados que no son estáticamente tipados.
- Pascal es un lenguaje que tiene tipado estático (Ghezzi – 54).
- Pascal no es fuertemente tipado (Ghezzi – 154).

Un Sistema de Tipos puede ser Fuertemente Tipado requiriendo que:
- Sólo se puedan utilizar tipos predefinidos.
- Todas las variables son declaradas y asociadas a un tipo específico.
- Todas las operaciones son especificadas determinando con exactitud su signatura (los tipos requeridos por sus operandos y el tipo del resultado).

Teniendo en cuenta la segunda definición, se puede *alcanzar la posición de Fuertemente Tipado con requerimientos leves*. Es decir, especificando con precisión las reglas para:
- Conversión o coerción
- Equivalencia o Compatibilidad

- **REGLAS DE CONVERSIÓN:** Un sistema de tipos flexible brinda *reglas de conversión* o *coerción* que permiten aceptar datos de un tipo Q en un contexto en el que se espera un dato de tipo T.

  La conversión puede ser:
  - *Con pérdida (limitante)* → Insegura
  - *Sin pérdida (expansora)* → Segura

  Sea una operación definida por una función *f*: $T_1 \longrightarrow R_1$ (espera un parámetro de tipo T_1 y evalúa un resultado de tipo R_1).
  La operación *f* puede ser invocada con un argumento de tipo T_2, si el lenguaje brinda una regla que permita convertir los valores de tipo T_2 en valores de tipo T_1.
  La operación *f* puede ser invocada en un contexto en el que se espera un valor de R_2, si el lenguaje brinda reglas para convertir valores de tipo R_1 en valores de tipo R_2.

  En la mayoría de los lenguajes durante la asignación hay una operación implícita de *dereferenciamiento*.
  Ej: x:= x+1

  En algunos lenguajes, cualquier conversión permitida es aplicada automáticamente por el compilador. Tales conversiones automáticas son llamadas *coerciones.*

  Una *ventaja* de contar con reglas de coerción en un LP es que muchas conversiones deseables son automáticamente provistas por la implementación. La *desventaja* es que, dado que las transformaciones realizadas no son visibles al programador, el lenguaje se torna complicado y los programas "oscuros".

  Obs: Aún en lenguajes con reglas de conversión muy flexibles pueden producirse errores. Además, otros conflictos pueden producirse cuando se combina conversión con otras características del lenguaje (sobrecarga de operadores y rutinas).

  { Ejemplos - Reglas de conversión en los LP }

  Fortran → Resuelve todas las conversiones de acuerdo a una jerarquía de tipos:
  COMPLEX > DOUBLE PRECISION > REAL > INTEGER
  Ada → No provee coerciones. Exige que todas las conversiones se hagan explícitamente
  Pascal → Las conversiones no están especificadas con precisión y dependen de la implementación

  C → Provee un sistema de coerción simple.
  Ej:
  ```
  int x,y;
  real z;
  x = y+z;
  ```
  ← *y* se convierte a *real* antes de evaluar la suma, y luego el resultado se convierte a *int*

  Además, el programador puede indicar conversiones explícitas mediante el constructor de *cast*.
  Tal conversión explícita es semánticamente definida, asumiendo que la expresión a ser convertida es implícitamente asignada a una variable anónima del tipo especificado en el *cast*, utilizando las reglas de coerción del lenguaje.
  Ej: x = y+ (int) z;

  C++ → Similar a C
  Java → Tiene un tratamiento diferenciado para *tipos primitivos* y *tipos clase*

- **REGLAS DE EQUIVALENCIA:** Un *sistema de tipos estricto* requiere que si una operación espera un parámetro de tipo T, pueda ser invocada legalmente únicamente con un parámetro de tipo T. Sin embargo, los lenguajes generalmente permiten mayor *flexibilidad* definiendo condiciones bajo las cuales un parámetro de otro tipo Q, es también aceptable sin violar la seguridad de tipos. En tal caso, se dice que el lenguaje define que, en el contexto de una determinada operación, los *tipos* T y Q son *equivalentes/ compatibles*.

  Dos *tipos de datos* son *equivalentes* si cada uno de ellos puede aparecer en un contexto en el que se espera una aparición del otro (NO HAY CONVERSIÓN).

  <u>*Criterios de Equivalencia/Compatibilidad*</u>: (Idem sección 5.7)

  - *Equivalencia por Nombre:* Dos variables son de tipos equivalentes si han sido declaradas en una misma instrucción, o con exactamente el mismo nombre de tipo.
  - *Equivalencia por Estructura:* Dos tipos de datos son equivalentes si al reemplazar los identificadores por sus definiciones, los objetos de datos que definen tienen la misma estructura interna (la misma representación).
  - *Equivalencia por Declaración:* Dos variables son equivalentes si son equivalentes por nombre, o si el tipo de una de ellas está definido directamente a través del tipo de la otra.

  La *compatibilidad por nombre* es más simple de implementar, y también es más fuerte que la *compatibilidad por estructura*.

  Al momento de implementar la *equivalencia por estructura* es necesario plantear cómo resolver ciertos *problemas*:

  - Circularidad
  - Determinar si dos registros con diferentes nombres para sus campos son compatibles (En general, se consideran compatibles si los campos aparecen en el mismo orden)
  - Determinar si dos arreglos con la misma cardinalidad y rango son equivalentes

  { Ejemplos – Criterios de equivalencia en LP }

  Ada → Adopta compatibilidad por nombre. Sin embargo, dado que el lenguaje soporta el concepto de subtipo, los objetos pertenecientes a distintos subtipos del mismo tipo son considerados compatibles.

  C → Adopta compatibilidad por estructura para todos los tipos, excepto para *struct* (registros y uniones), que requieren compatibilidad por nombre.

  Java → La compatibilidad para las clases de equivalencia está dada por la jerarquía de herencia.

- **INFERENCIA DE TIPOS:** La *inferencia de tipos* permite que el tipo de una entidad declarada se "infiera", en lugar de ser declarado.

  La inferencia puede realizarse de acuerdo al tipo de:

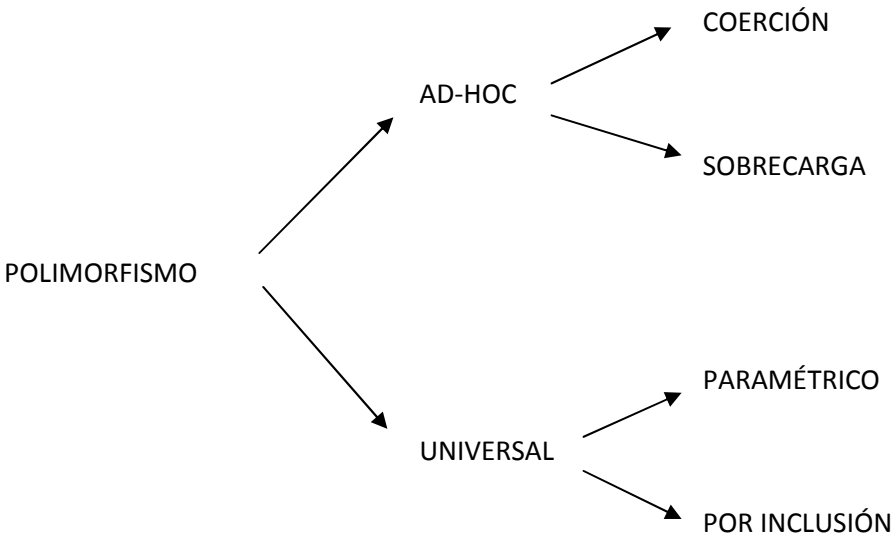
- Un operador predefinido	fun f1(n,m) = (n mod m=0)	→ n, m: int
- Un operando	fun f2(n) = (n*2)	→ n: int (numérico default)
- Un argumento	fun f3(n:int) = (n*n)	→ resultado: int
- El tipo del resultado	fun f4(n):int = (n*n)	→ n: int

- **NIVEL DE POLIMORFISMO:**
  Un *lenguaje* se dice *mono-mórfico* si cada entidad se liga a un único tipo. Cada entidad (variable, constante, función) se declara de un tipo específico. En un lenguaje mono-mórfico los chequeos y las ligaduras pueden establecerse en forma estática.

  Los LP mono-mórficos son seguros pero inflexibles – Ejemplos:
  - La función *disjuntos* deberá implementarse para cada tipo particular de conjuntos
  - Las funciones *read* y *write* son "polimórficas", pero no de forma pura (ya que el compilador infiere el tipo).

  Un *lenguaje* se dice *polimórfico* si las entidades pueden estar ligadas a más de un tipo.
  Las *variables polimórficas* pueden tomar valores de diferentes tipos. Las *operaciones polimórficas* son funciones que aceptan operandos de varios tipos. Los *tipos polimórficos* tienen operaciones polimórficas.

  Clasificación de Polimorfismo



Polimorfismo Ad-Hoc: El *polimorfismo ad-hoc* permite que una función se aplique a distintos tipos con un comportamiento sustancialmente diferente en cada caso.

Las funciones polimórficas ad-hoc se aplican a un conjunto pequeño de tipos y pueden comportarse diferente para cada tipo → Una función polimórfica ad-hoc puede ser vista como una abreviación sintáctica para un conjunto pequeño de diferentes funciones mono-mórficas.

Coerción: La *coerción* permite que un operador que espera un operando de un determinado tipo T pueda aplicarse de manera segura sobre un operando de tipo diferente al especificado (convierte el argumento de una función al tipo esperado por la función)→ *El polimorfismo es aparente*: la función realiza su tarea únicamente para el tipo definido, mientras que un argumento de tipo diferente puede ser utilizado, pero el argumento es automáticamente transformado al tipo requerido antes de la evaluación de la función.

Sobrecarga: El término *sobrecarga* se utiliza para referirse a conjuntos de abstracciones diferentes que están ligadas al mismo símbolo o identificador → El mismo nombre de función puede ser utilizado en diferentes contextos para denotar *diferentes funciones*, y en cada contexto la función es unívocamente determinada (hay implementaciones diferentes de funciones que utilizan el mismo nombre).

Cuando hay *sobrecarga* un mismo símbolo denota dos o más entidades diferentes, pero en cada llamada particular queda ligado a una función específica.

Criterios para resolver la sobrecarga:

- *Independientes del Contexto:* Los datos de entrada de la función son de tipos diferentes.
- *Dependientes del Contexto:* Los datos de entrada son del mismo tipo, pero los de salida no.

Obs: La inclusión de sobrecarga al lenguaje puede traer inconvenientes si se combina con otras características como el tipo unión, la utilización de parámetros por omisión y las expresiones mixtas.

{ Ejemplos – Sobrecarga en LP }

C++, C# y Java → Admiten expresiones mixtas pero la sobrecarga es siempre independiente del contexto

Ej: En C el operador aritmético + es una función polimórfica ad-hoc cuyo comportamiento es diferente dependiendo si se aplica a valores int o float. Si los 2 operandos son del mismo tipo, el operador se liga al tipo correspondiente, si son de distinto tipo invoca el operador + para float, luego de convertir el operando entero a real.

Ada → Soporta sobrecarga dependiente del contexto, pero no con expresiones mixtas.

Ej:
```
int f(int x);
float f(int x);

int A,B;          ← Si la operación + admite operandos mixtos no puede establecerse la ligadura
A = B + f(2);
```

Polimorfismo Universal: El *polimorfismo universal* permite que una única operación actúe uniformemente sobre un conjunto infinito de tipos, los cuales tienen alguna estructura común (tipos relacionados) →Una función polimórfica universal ejecuta el mismo código para argumentos de todos los tipos admisibles.

<u>*Paramétrico*</u>: Si la uniformidad de la estructura de tipos está dada a través de parámetros, hablamos de *polimorfismo paramétrico* (Genericidad) → En este caso, una función polimórfica actúa uniformemente sobre un rango de tipos. Un parámetro de tipo (implícito o explícito) determina el tipo de los argumentos para cada aplicación de la función.

Un *tipo parametrizado* es un tipo que tiene otros tipos como parámetros.

En un lenguaje hipotético podrían definirse tipos parametrizados de la siguiente forma:

Ej:

```
par(T) = T x T
lista(T) = T*
```

En *ML* el concepto de polimorfismo está soportado en el concepto de *politipo* (permite funciones genéricas)

Ej:

```
datatype T list = nil | cons of (T* T list)

longitud(L: T list) =   case L of
                        nil => 0
                        | cons(h,t) => 1+ longitud(t)
```

Los *lenguajes funcionales*, en general, proveen este tipo de polimorfismo. En este contexto, muchas veces aparece vinculado a algún mecanismo de inferencia de tipos.
Así se tienen los siguientes casos:

- Si el operador de una expresión es mono-mórfico la inferencia de tipos infiere un *monotipo* para la función
- Si el operador es polimórfico la inferencia de tipos producirá un *politipo*

<u>*Por Inclusión*</u>: El *polimorfismo por inclusión* es un mecanismo para soportar la *especialización* de un tipo T en otro tipo T' → Una función es aplicable tanto a un tipo como a sus "subtipos" (hay una única implementación de la función que puede ser utilizada por distintos tipos).

Existen formas limitadas de polimorfismo por inclusión:

- Subtipos
- Herencia (mecanismo sumamente poderoso – POO)

SUBTIPOS: Un *tipo* se define como un conjunto de valores y un conjunto de operaciones. Un *subtipo* T' de un tipo T puede definirse como un subconjunto de los valores de T y el mismo conjunto de operaciones. → T' es un *subtipo* de T, si T' $\subseteq$ T de modo que cualquier valor de T' puede ser usado de manera segura en cualquier expresión en que se espera un valor de tipo T.

Un lenguaje que soporta subtipos debe definir:

- Una forma de definir subconjuntos de un determinado tipo
- Reglas de compatibilidad entre un subtipo y su supertipo

{ Ejemplos – Subtipos en LP }

Pascal → subrangos
Ada → subtipos type Vector is array(integer <range>) of integer
subtype Vect is Vector(0..99)

HERENCIA: El mecanismo de *herencia* permite definir una nueva clase *derivada* a partir de una clase *base* ya existente. La clase derivada *hereda* los atributos y el comportamiento de la clase base, puede *agregar* atributos y comportamiento, y *redefinir* comportamiento de manera compatible → Se dice que la clase derivada es un *subtipo* de la clase base.

Cuando entre la clase derivada y la clase base existe una relación de tipo "is a", toda instancia de la clase derivada es también instancia de la clase base.

En el uso de la relación de herencia se puede dar el caso en el que la clase derivada oculta atributos o comportamiento, o redefine operaciones en forma no compatible. En estos casos, la relación no es de tipo "is a", y por lo tanto no estamos en un contexto en el que podamos hablar de polimorfismo por inclusión.

Una *variable polimórfica* puede estar asociada a objetos de diferentes clases dentro de la jerarquía de clases → Si cada objeto va a responder a un mensaje de acuerdo a la clase a la que pertenece se requiere *vinculación dinámica de código* (ejecuta el código definido en la clase de la cual es instancia).

[HERENCIA Y TIPADO FUERTE]

Restricciones para el correcto manejo de tipos:

- Una variable declarada de tipo B no puede vincularse a un objeto instancia de una superclase de B.
- No pueden redefinirse métodos o se debe conservar la signatura
- Pueden redefinirse signaturas de métodos pero respetando reglas de covarianza y contravarianza.

En un *lenguaje OO* con un *sistema fuertemente tipado*, el polimorfismo de las variables se ve limitado. El costo es la *integridad conceptual* del lenguaje, que se ve comprometida al imponer fuertes restricciones sobre la herencia.

→ Para evitar las violaciones de tipos se debe poder *sustituir* a un objeto de una clase derivada por un objeto de la clase base en cualquier contexto. Si se puede garantizar la sustitución de objetos, un tipo derivado puede verse como un subtipo del tipo base.

Imponiendo ciertas *restricciones* en el uso de la herencia se puede *garantizar la sustitución* (Un LP OO puede brindar polimorfismo y ser fuertemente tipado):

- *Extensión del Tipo*: La clase derivada sólo puede extender la funcionalidad de la clase base. No puede modificar o esconder las operaciones provistas por la clase base (Ej: Ada95).

- *Sobre-escritura de las Operaciones*: Permite que la clase derivada redefina una operación heredada. En este caso, para garantizar la sustitución de objetos se imponen las siguientes reglas:

    - CONTRAVARIANZA: Los *parámetros de entrada* deben ser *supertipos* de los tipos de los parámetros originales de la operación (Tipos entrada menos restrictivos que la fc. original).

    - COVARIANZA: Los *parámetros de salida* (resultado) deben ser *subtipos* de los tipos de los parámetros originales de la operación (Tipos salida más restrictivos que la fc. original).

{ Ejemplos – LP Orientados a Objetos / Tipado Fuerte }

C++ ~~y Java~~ → Soportan una versión limitada de la redefinición de operaciones, dado que requieren que la signatura de la operación redefinida coincida con la signatura de la operación dada en la clase base.

Eiffel y Ada → Dado que la noción de contravarianza es impráctica (los parámetros de una función redefinida no pueden imponer nuevas restricciones específicas) requieren covarianza tanto para los parámetros de entrada, como para los de salida.

[Sin embargo, utilizando la covarianza para el tipo resultado y la contravarianza para los parámetros se puede lograr un sistema de tipos que soporta la forma menos restringida de polimorfismo por inclusión, y que a la vez tiene tipado seguro]

CAPÍTULO VIII – Estructuras de Control a Nivel de Sentencias

SENTENCIAS DE CONTROL

Una *sentencia de control* permite seleccionar el flujo de control del programa, entre varios caminos alternativos. Además provee una forma de repetición para un conjunto de sentencias.

→ Contar con una gran cantidad de sentencias de control beneficia la facilidad de escritura de los programas. Por otra parte, el uso de demasiadas/muy pocas sentencias de control dificulta su lectura (deben conocerse todas las estructuras/es difícil seguir el programa).

ESTRUCTURAS DE CONTROL

Una *estructura de control* es una sentencia de control y una colección de sentencias cuya ejecución controla.

Las estructuras de control a nivel sentencia usualmente distinguidas son:

- *Composición:* Las sentencias es colocan en un orden textual, de manera que son ejecutadas en ese orden.
- *Alternación* (sentencias condicionales): Dos secuencias de sentencias pueden formar alternativas de manera que una u otra secuencia es ejecutada, pero no ambas.
- *Iteración:* Una secuencia de sentencias puede ser ejecutada repetidamente, cero o más veces.

Al construir programas se combinan las sentencias básicas en secuencias apropiadas que mediante el uso de *composición*, *alternación* e *iteración* logran el efecto deseado.
Las principales características a tener en cuenta son:

- Secuencia
- Transferencia de Control (condicional y no condicional)

Transferencia de Control Incondicional

Los primeros lenguajes cercanos al HW de la máquina trabajaban con tipos básicos y con etiquetas o rótulos y saltos como instrucciones. Las transferencias de control se realizaban a través de sentencias del tipo *goto.*
Este tipo de sentencia de salto incondicional es una de las sentencias más poderosas para controlar el flujo de control de los programas.

Aspectos de diseño: Los rótulos son de suma importancia, es necesario establecer las restricciones sobre ellos:
- Constantes
- Expresiones sin cómputo en ejecución
- Expresiones

El uso de las instrucciones como el *goto* conduce a programas con diseño no estructurado, haciendo muy difícil utilizar modelos de correctitud para los programas. En general, sus ventajas no amortizan sus grandes desventajas (dificultan la lectura de los programas, los hacen más difícil de mantener).

El uso de este tipo de instrucciones es superfluo, ya que puede fácilmente simularse con *secuencias de control estructuradas.*

Transferencia de Control Restringida
La idea es tener un solo punto de entrada y salida. En general, es implementada en los lenguajes a través de la instrucción *break*. Esta instrucción provoca que el programa avance hacia adelante a un punto explícito al final de una dada estructura de control.

→La sentencia *break* es utilizada para situar el control de loops en otro lugar que no sea ni el comienzo ni el fin del bucle. El *break* finaliza la ejecución del bucle más interno (En el *switch* se usa para ir al fin de la estructura).
La sentencia *continue* tiene un uso similar al *break*, con la diferencia de que finaliza la iteración actual, en lugar de la ejecución completa del bucle.

PROGRAMACIÓN ESTRUCTURADA
Al diseñar una programación más estructurada se tuvieron en cuenta los siguientes aspectos:

- *Estructura jerárquica de los programas:* Es más importante contar con un programa correcto que más eficiente → El concepto de *una entrada-una salida* condice a diseños más fáciles de comprender.

- *Reducir los programas con sentencias de ejecución irregular:* El uso de *goto* hace que el control del programa fluya en un orden que tiene poca conexión con el orden de las instrucciones.

- *Grupos de sentencias pueden ser simple-propósito:* El uso de *goto* fomenta grupos de sentencias multi-propósito haciendo más compleja la comprensión del programa.

Para obtener programas más fáciles de entender, verificar, corregir, modificar y re-verificar se debe enfatizar:

- Organización jerárquica de la estructura del programa (uso de composición, alternación e iteración).
- Uso de estructuras de control estructuradas (una entrada-una salida por instrucción).
- Correspondencia entre el orden del texto del programa y el orden de ejecución.
- Usar grupos de sentencias con propósito simple, aún cuando se requiera copia.

Las ***estructuras de control estructuradas*** son:
- ***Condicional***
- ***Iteración***
- ***Secuencia***

Secuencia de Control Estructurado
Una *sentencia compuesta* es una secuencia de instrucciones que pueden ser tratadas como una sola sentencia en la construcción de sentencias más grandes → Representa básicamente la *composición*.
El orden en el que aparecen en el texto del programa es el orden en el cual son ejecutadas.

Aspectos de Diseño:
- Delimitadores
- Separadores y terminadores
- Bloques

Sentencia Condicional

Una *setencia condicional* es aquella que expresa una alternativa entre dos o más flujos de ejecución. El control de las alternativas es controlado por una *condición*, usualmente expresada a través de una expresión booleana.

Aspectos de Diseño:

- Condición simple, doble, o múltiple
- Instrucción simple, compuesta, o secuencia
- Anidamiento
- Forma y tipo de las expresiones de control

- *Sentencia Condicional Simple*: Respeta la sintaxis *if <condición> <instrucción simple>* → Es malo para la legibilidad del programa.

- *Sentencia Condicional Doble*: La forma general que adopta el condicional doble en los lenguajes de programación es *If <condición> then <instrucción1> else <instrucción2>*

  La instrucción puede ser simple o compuesta.

  { Ejemplos – Sentencia Condicional Doble en LP }

  Ada → if <condición> then <instrucción> [else <instrucción>] endif

  Módula → if <condicion> then <instruccion> [else <instrucción>] end

  Algol68 → if <condicion> then <instruccion> [else <instrucción>] fi

  Java → if <condicion> <instrucción> [else <instrucción>]

  Anidamiento: La mayoría de los lenguajes permite anidar sentencias condicionales, lo que puede provocar algunos problemas semánticos (interpretación del programa resultante). Así, si se tienen dos *if* y un solo *else*, aparece el problema de determinar a qué *if* corresponde el *else*:

  C, C++, C# y Java → Asociar el *else* al *if* más cercano (puede haber diferencia entre lo que el programador quiso hacer y lo que asoció el compilador).

  Perl → Usar sentencias compuestas (con delimitadores especiales).
  Ada → Uso de terminador de sentencia (endif).

- *Sentencia Condicional Múltiple*: La sentencia condicional múltiple permite la selección de una sentencia o grupo de sentencias entre un grupo no fijo de sentencias.

  Aspectos de Diseño:
  - Alternativas excluyentes o no
  - Forma y tipo de las expresiones selectoras
  - Cómo se especifica el segmento seleccionado
  - Valores no representados
  - Instrucción simple, compuesta o secuencia
  - Flujo de ejecución cuando termina

La forma más moderna de selector múltiple es la sentencia *switch* de C/Java, o el *case* de Pascal:

switch (<expresión>){
    case <constante$_1$>: <sentencias$_1$>;
    …
    case <constante$_n$>: <sentencias$_n$>;
    [default: <sentencias$_{n+1}$>;]
}

case <expresión> of
<constante$_1$>: <sentencias$_1$>;
…
<constante$_n$>: <sentencias$_n$>;
 [else <sentencias$_{n+1}$>]
end;

Si las sentencias no son excluyentes, la ejecución no es determinista. En general, en estos casos el lenguaje establece reglas para determinar qué alternativa va a ejecutarse, y si es posible ejecutar más de una.

Ej: En el *case,* la expresión y las constantes pueden ser de cualquier tipo ordinal. Las alternativas deben ser excluyentes, y existe un salto implícito al final de cada segmento de código seleccionable.

Ej: Para el *switch*, las alternativas no son excluyentes, por lo que puede ser que la expresión *matchee* con más de una alternativa. Si se desea hacerlas excluyentes se puede incluir el *break* en cada uno de los casos.

La sentencia *switch* de C tiene tomadas varias decisiones de diseño importantes:
- La expresión que controla la sentencia y las constantes sólo pueden ser de tipo entero
- Los segmentos de código seleccionables pueden ser secuencias de sentencias, bloques, o sentencias compuestas.
- Cualquier cantidad de segmentos puede ser ejecutada en una ejecución del constructor (alternativas no excluyentes).
- No hay un salto implícito al final de cada segmento de código seleccionable.

Obs: El *switch* de C# difiere del de sus predecesores basados en C. Una regla estática de semántica no permite la ejecución implícita de más de un segmento. La regla es que cada segmento seleccionable debe terminar con una sentencia de salto incondicional explícita (*break* para salir del *switch*, o *goto* a un label dentro o fuera del *switch*).

Ej: La sentencia *case* de Ada es un descendiente de la sentencia de múltiple selección de Algol68.
La forma general del constructor es:

case <expresión> is
when <constante>* => <secuencia de sentencias>
…
when <constante>* => <secuencia de sentencias>
[when others => <secuencia de sentencias>]
end case

Esta versión es más confiable ya que al finalizar la secuencia elegida, el control es pasado a la siguiente instrucción luego de la sentencia *case*. La expresión puede ser de cualquier tipo ordinal, y la especificación de los casos debe ser exhaustiva.

- *Comandos con Guarda – Múltiple selección con if*: La forma general de este constructor es

  if <expresión booleana> → <sentencia>
  [[] <expresión booleana> → <sentencia>
  …
  [] <expresión booleana> → <sentencia>]
  fi

  Semántica del constructor – Cuando se alcanza el constructor durante la ejecución:

  1. Evaluar todas las expresiones booleanas.
  2. Si más de una es verdadera, elegir una en forma no determinística (la correctitud del programa no puede depender de cuál sentencia se elija).
  3. Si ninguna es verdadera se produce un error en ejecución (esto obliga al programador a considerar todas las posibilidades, como en el *case* de Ada).

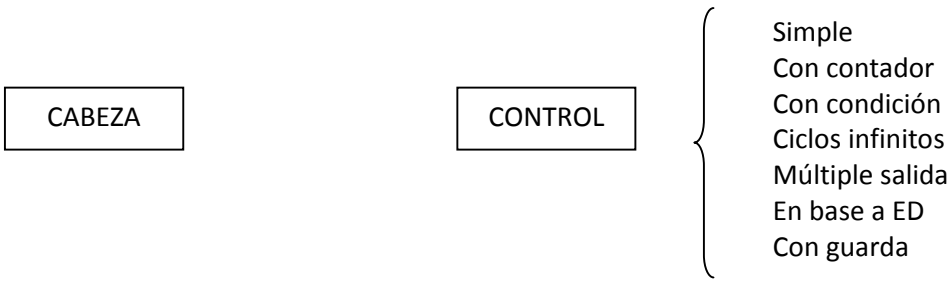
Obs: Cuando se desea implementar múltiple selección sobre una condición booleana, es posible utilizar sentencias condicionales anidadas, o en los casos en que sea provisto, el constructor *elsif*.

Sentencia Iterativa

Una *sentencia iterativa* es aquella que causa que una sentencia o una colección de sentencias se ejecute cero, una o más veces. Generalmente son llamadas *loop* o *ciclos.*

Los lenguajes que proveen facilidades para definir iteración conducen a programas menos extensos, flexibles, más simples de escribir y almacenar.

Aspectos de Diseño:
- Cómo se controla la iteración?
- Dónde debe aparecer el mecanismo de control en el constructor de ciclo?



El control puede efectuarse antes de la ejecución del cuerpo del constructor (pretest), o después (posttest)

- *Iteración Simple*: La iteración simple está presente en el lenguaje COBOL, en el constructor *perform*

  PERFORM <cuerpo> <variable> TIMES

  <cuerpo>: Es una instrucción simple o una secuencia de instrucciones.
  <variable>: Determina la cantidad de iteraciones. Esta variable puede modificarse en el <cuerpo>, afectando la cantidad de iteraciones.

- *Iteración con Contador*: Una sentencia de iteración con contador incluye una variable, llamada *variable de control*, en la cual se mantiene el valor del contador. La cabeza establece el valor inicial, final, y el incremento del contador.

  Aspectos de Diseño:
  - Cuándo se evalúan las expresiones que controlan el bucle? Antes de comenzar el bucle o en cada iteración?
  - Cuál es el tipo y alcance de la variable de control?
  - Cuál es el valor de la variable de control una vez que el bucle termina?
  - Puede modificarse el valor de la variable de control dentro del bucle? Altera la cantidad de iteraciones?

  { Ejemplos – Iteración con Contador en LP }

  Fortran – *DO* → DO <rótulo> <variable> = <expresión>, <expresión> [<instrucción>]*
  <rótulo> <instrucción>

  Pascal – *for* → for <variable> := <expresión> to|downto <expresión> do <instrucción>

  La variable de control debe ser de tipo ordinal, y no puede modificarse en el cuerpo del ciclo. Después del ciclo su valor queda indefinido.

  C-Based – *for* → for (<expresión1>; <expresión2>; <expresión3>) <cuerpo del ciclo>

  El cuerpo del ciclo puede ser una sentencia simple, una sentencia compuesta, o una sentencia nula.

  La primera expresión es para inicialización, y se evalúa una sola vez, cuando comienza la ejecución del *for*. La segunda expresión es el control del ciclo, y es evalúa antes de cada ejecución del cuerpo del ciclo (Si el valor de la segunda expresión es falso, el *for* termina). La última expresión es ejecutada luego de cada ejecución del cuerpo del ciclo, y se utiliza para incrementar el contador del ciclo.

  Todas las expresiones son opcionales (la ausencia de la segunda expresión es considerada como *true*, por lo que es un ciclo potencialmente infinito).

  Permite utilizar más de un contador, los cuales son separados por "," en las expresiones, y también permite la modificación de las variables dentro del cuerpo del ciclo.

- *Iteración con Condición*: En muchos casos, una colección de sentencias debe ser repetida, pero el control de la repetición está basado en una condición booleana en lugar de en un contador. Estos constructores son más generales que los que utilizan contador, y un loop con contador siempre puede ser modelado con un loop con condición booleana (pero no viceversa).

  Aspectos de Diseño:
  - Testeo de la condición antes del cuerpo del ciclo, después, o ambos?
  - Debe ser un mecanismo específico, o una forma especial de iteración con contador?

  { Ejemplos – Iteración con Condición en LP }

  C → Tiene loop pretest (while, for) y posttest (do while)
  Java → Al igual que C, pero sólo permite expresiones booleanas
  Pascal → Tiene loop pretest (while) y posttest (repeat until)

- *Iteración Contador-Condición*:

  Ada → *for* <variable> *in* [reverse] <rango discreto> [*while* <expresión>] *loop* <instrucción> *end loop*

  for k in index *while* A(k)/=0 *loop*
  …
  End loop

  El <rango discreto> es un subrango de los enteros o de un enumerado, y el alcance de la variable de ciclo es el ciclo mismo.

  Eiffel → *from* <expresión>
  invariant <expresión>
  variant <expresión>
  until <condición sobre la variable de control>
  loop <instrucción>
  end

- *Iteración infinita*: Utilizando bucles condicionales cuya condición es siempre verdadera (Ej: while true {}).

- *Iteración con Múltiple Salida*: (Puede salirse del bucle por más de un lugar)

  Aspectos de Diseño:
  - Es posible salir de un único bucle o de varios anidados?
  - La salida se establece con o sin condición?
  - La salida es etiquetada?

  { Ejemplos – Iteración con Múltiple Salida en LP }

  C y C++ → Salidas incondicionadas y sin etiqueta (break)

  Java, Pearl y C# → Salidas incondicionadas y con etiqueta (break <*label*> - a dónde se quiere salir)

- *Iteración en Base a ED*: Aspectos a tener en cuenta:

  - El número de elementos de la ED controla la cantidad de iteraciones del ciclo
  - El mecanismo de control es una llamada a una función *iterador* que retorna el próximo elemento en algún orden establecido (si es que hay un próximo elemento, sino termina).
  - Las instrucciones como el *foreach* de C# iteran sobre los elementos de arreglos y otras colecciones.

  { Ejemplos – Iteración basada en ED en LP }

```
C++ →   x.start()
        While x.more()
        {
                …
                y = x.next()
        }

Java →  for (ptr = root; ptr == null; traverse(ptr))
        {…}
```

- *Iteración con Guarda*: El *propósito* es soportar una nueva forma de programación que permita la verificación durante el desarrollo.
  La idea básica es que si el orden de evaluación no es importante, el programa no debería especificar uno. Resulta básico para la programación concurrente (Ada y CSP).

  La forma del ciclo con guarda es:

  do <expresión booleana> → <sentencia>
  [[] <expresión booleana> → <sentencia>
  …
  [] <expresión booleana> → <sentencia>]
  od

  La semántica del constructor es:

  1. Evaluar todas las expresiones booleanas
  2. Si más de una es verdadera, elegir una en forma no determinística y ejecutarla. Luego comenzar el ciclo nuevamente (evaluar las expresiones nuevamente).
  3. Si ninguna de las expresiones es verdadera (todas son simultáneamente falsas), termina el bucle.

Control de Secuencia { Capítulo 8 – Pratt }

Las estructuras de control de secuencia pueden categorizarse en cuatro grupos:

~~(*)~~

- *Expresiones*: Las expresiones son la forma básica de construir bloques y expresan cómo los datos son manipulados y modificados por el programa.
- *Sentencias o Instrucciones*: Determinan cómo el control fluye de un segmento de programa a otro.

- *Programación Declarativa*: Modelo de ejecución que no depende de sentencias, pero establece cómo la ejecución tiene lugar (Ej: Prolog).

- *Unidades – Subprogramas*: Indican una manera de transferir el control de un segmento de programa a otro.

Obs: Esta división no es precisa, ya que hay lenguajes que tienen más de una característica (Ej: LISP – Computa con expresiones pero usa mecanismos de control de sentencias).

Las estructuras de control de secuencia pueden ser:

- *Implícitas*: Son las definidas por el lenguaje para tener efecto a menso que sean modificadas por el programador a través de una estructura de control explícita.

- *Explícitas*: Son aquellas definidas por el programador para modificar el control implícito. Por ejemplo, el uso de los paréntesis en las expresiones, o el uso de *goto* <label> en un programa que sólo tiene secuencia.

(*) Ya se vio cómo se resuelve el *control de secuencia a través de expresiones e instrucciones*.

~~*Control de Concurrencia*: Es necesario contar con estructuras de control específicas.~~

~~Declare E evento~~
~~Read(x) Evento E~~ ~~←~~
~~Wait(E)~~

~~Cuando se alcanza la instrucción *read* se genera un nuevo proceso que ejecuta las instrucciones siguientes al *read*, y se interrumpe para que termine la lectura.~~
~~Mecanismo eficiente por ser de bajo nivel.~~

~~fork L~~
~~s2~~ ~~←~~
~~L: s3~~
~~join~~

~~Mecanismo de control de bajo nivel, por lo que tiene las mismas deficiencias que la transferencia de control.~~
~~Ejecuta paralelamente s2 y s3, y al hacer el *join* se termina la concurrencia.~~

parbegin
  s1
  …
  sN
parend

← Las instrucciones s1..sN pueden ejecutarse en cualquier orden. Si se ejecuta en un entorno multiprocesador, se pueden ejecutar en paralelo.
Mecanismo de alto nivel.

Los mecanismos de alto nivel expresan la concurrencia en alto nivel, pero requieren mecanismos de sincronización de bajo nivel para sincronizarla (Ej: semáforos).

- **PROGRAMACIÓN DECLARATIVA** – Control de Secuencia:

  Pattern Matching: Una operación tiene éxito si ajusta y asigna un conjunto de variables a un patrón predefinido. Es una operación crucial para lenguajes como ML, Prolog, Miranda, Pearl, Snobol.

  El Pattern Matching produce dos efectos:

  - Como estructura de control, permite decidir qué parte del código se va a ejecutar.
  - Permite establecer ligaduras entre el patrón de la definición y el de la instancia (instancia las variables).

  { Ejemplo – Pattern Matching en ML }

  Datatype Dias = lunes|martes|miercoles|jueves|viernes|sabado|domingo

  Ocupacion = libre|parcial|completo

  fun laborable(domingo) = libre | laborable(sábado) = parcial | laborable(_) = completo

  laborable(X) se vincula en función del valor ligado al parámetro actual (laborable: Dias → Ocupacion)

  Re-escritura de Términos: Es una forma restringida de pattern matching.

  { Ejemplo – Re-escritura de términos en ML }

  fun fact(1) = 1
  | fact(n: int) = n*fact(n-1)

  ← ML reemplazará la llamada a la función por su apropiada definición de acuerdo al valor del parámetro.

  fun factorial(n: int) = if n=1 then 1
  else n* factorial(n-1)

  ← No hay pattern matching ya que el condicional refleja una división del dominio (dividir no es lo mismo que definir por casos).

{ Pattern Matching en PROLOG }

El *pattern matching* en PROLOG establece la relación entre la consulta y un hecho o regla que aparece en la base de datos. La condición de la regla se transforma en una nueva consulta, luego de la *sustitución* adecuada de variables.

Sustitución: Es el principio general detrás del pasaje de parámetros y la expansión de macros → Es el resultado de aplicar nuevos valores a argumentos expresados como patrones o templates.

Unificación: Es una forma de pattern matching → Es el resultado de aplicar sustituciones simultáneas sobre un conjunto de templates, mostrando que son todas equivalentes bajo un conjunto de sustituciones simultáneas.
La determinación de un conjunto válido de sustituciones para una determinada consulta es la esencia de la unificación.

El mecanismo de pattern matching de Prolog es una combinación de unificación con sustitución de variables.
El control lo tiene el sistema. Si las cabezas de dos reglas son iguales idealmente debería resolverse con no determinismo pero es menos seguro, por lo que se elige un mecanismo de selección de reglas combinado con backtracking.

{ Ejemplo – Sustitución y Unificación en Prolog }

Patrón:	F(A,B) = G(A,B)
Ejemplo:	F(g(i),h(j))
Sustitución:	g(i) por A h(j) por B F(A,B) = G(g(i),h(j))

Patrón:	F(A,B) = G(A,B) M(C,D) = N(C,D)
Ejemplo:	F(Juan,M(h(v),7))
Sustitución:	Juan por A M(h(v),7) por B F(A,B) = G(Juan, M(h(v),7)) h(v) por C 7 por D M(C,D) por N(h(v),7)
Unificación:	F(Juan, N(h(v),7)) G(Juan, N(h(v),7))

- **UNIDADES [Capítulo 9 – Sebesta]**

CAPÍTULO IX – Unidades (Subprogramas)

INTRODUCCIÓN

Los LP imperativos y OO brindan mecanismos para dividir el programa en *unidades*, cada una de las cuales tiene cierta coherencia y lógica. Su incorporación fomenta:

- *Eficiencia:* Permiten factorizar el código (en los primeros lenguajes sólo se perseguía el incremento de eficiencia, principalmente de almacenamiento).
- *Legibilidad:* Metodología de diseño top-down (mecanismos más abstractos).
- *Verificación:* Una unidad puede pensarse como un mapeo entre dominios de valores. Se intenta que los chequeos sean lo más seguros posibles. Así evolucionaron los métodos de pasaje de parámetros.

Las unidades pueden ser *anónimas* (bloques) o *con nombre asociado* (subprogramas). Una unidad con nombre permite extender el lenguaje con una nueva primitiva.
Para que la abstracción sea poderosa, se debería poder usar una unidad conociendo sólo la interfaz. La implementación debería ser transparente → La unidad puede ser una abstracción a nivel expresión o a nivel instrucción.

En los lenguajes OO, cada unidad con nombre brinda un *servicio* que puede ser provisto por cualquiera de las instancias de la clase en la cual se define la unidad. En este tipo de lenguajes la clase es el mecanismo de abstracción. Un servicio puede acceder y hasta modificar el estado interno del objeto que recibe el mensaje que provoca la ejecución de ese servicio.

Aspectos de Diseño a tener en cuenta por el LP: (En general)
- Entidades
- Ligaduras
- Reglas de alcance
- Sistemas de tipos
- Soporte para definir unidades

ASPECTOS DE DISEÑO ESPECÍFICOS:
- Estructura estática
- Recursividad
- Control
- Métodos de pasaje de parámetros
- Correspondencia entre parámetros formales y actuales
- Sobrecarga y polimorfismo
- Tipos de los parámetros: datos y unidades
- Ambiente de referenciamiento de las unidades que son pasadas por parámetro (chequeo estático o dinámico de unidades)

ASPECTOS DE IMPLEMENTACIÓN:
- Chequeo de tipos entre parámetros
- Unidades de compilación
- Creación del ambiente de referenciamiento local

Atributos de la Unidad:

- ***Nombre**:* Algunos LP permiten definir unidades anónimas. Las unidades anónimas se ejecutan implícitamente y cuando se alcanzan se crea un nuevo ambiente de referenciamiento.
- ***Lista de parámetros**:* Permiten la comunicación de la unidad con el resto del programa. Si se admiten parámetros de diferentes tipos (en distintas llamadas) la *unidad* es *polimórfica*.
- ***Ambiente de referenciamiento:*** Se establece a qué otras unidades puede referenciar.
- ***Bloque ejecutable:*** Si la unidad tiene la capacidad de llamarse a sí misma, soporta recursividad.
- ***Alcance:*** Segmento de código dentro del cual la unidad puede invocarse (si incluye a la misma unidad acepta recursividad).

Obs: Salvo las corrutinas, las unidades tienen un único punto de entrada (al comienzo cuando son invocadas).

ESTRUCTURA ESTÁTICA

La unidad tiene un *encabezamiento* y un *cuerpo.*

El encabezamiento indica que la unidad sintáctica a continuación es la definición de un subprograma de un tipo determinado. Está formado por el nombre de la unidad, la lista de parámetros, y el tipo del resultado. En el encabezamiento se puede definir el *perfil de los parámetros,* el *protocolo* y la *signatura*.
El perfil de los parámetros de una unidad es el número, orden y tipo de los parámetros formales. El protocolo de una unidad es el perfil de los parámetros, más el tipo del resultado si es una función. La signatura de la unidad, está dada por el nombre de la unidad y su protocolo, y define la *interfaz* de la unidad.
El cuerpo está formado por las declaraciones locales y la sección ejecutable.

La unidad *estáticamente* posee:

- *Declaración*: provee el protocolo del subprograma, pero no incluye su cuerpo (el compilador debe chequear el tipo de los parámetros).
- *Definición:* Describe la interfaz y las acciones de la abstracción del subprograma.
- *Invocación:* Es el pedido explícito de que el subprograma llamado sea ejecutado. Tiene una vinculación bidireccional con la activación de la unidad y con su suspensión.

La unidad *dinámicamente* puede estar:

- *Pasiva*
- *Activa* (Luego de ser invocada, comenzó la ejecución pero todavía no la completó) – *Suspendida*

→ Su estado está determinado por las estructuras de control

COMUNICACIÓN ENTRE UNIDADES

La comunicación entre unidades puede darse a través de:

- *Ambiente Global*: Puede generar efectos colaterales (la ejecución de la unidad modifica el ambiente de referenciamiento de la unidad llamadora).
- *Ambiente Implícito*: Puede generar comportamiento sensible a la historia.
- *Parámetros y Resultado*

MECANISMOS PARA DEFINIR UNIDADES

- **Bloques:** Unidades anónimas
- **Rutinas:**
  - **Abstracciones Procedurales**: Abstracción a nivel instrucción. La manera de invocar a este tipo de rutinas es a través de una llamada en la cual aparece el nombre de la rutina (Ej: call <nombre proced>).

  - **Abstracciones Funcionales:** Abstracción a nivel expresión. La manera de invocar a este tipo de rutinas es en el contexto de una expresión.

Un *procedimiento* es una colección de sentencias que define computación parametrizada. Puede producir resultados en la unidad de programa llamadora mediante la modificación de variables globales/no locales (produce efectos colaterales), o mediante parámetros que permiten la transferencia de datos al llamador (parámetros de salida). Define nuevas sentencias (abstracción a nivel instrucción).
→Es una unidad que al ser invocada evalúa una instrucción y, en general, provoca un cambio en el estado del a computación (modifica los valores del ambiente de referenciamiento de la llamada provocando un efecto colateral). El usuario de un procedimiento observa únicamente las transformaciones en el estado de la memoria, pero no los pasos que dieron lugar a esa transformación.

Una *función* es estructuralmente similar a un procedimiento, pero es semánticamente modelada en funciones matemáticas. Si la función es un modelo fiel, no produce efectos colaterales (en la práctica muchas funciones tienen efectos colaterales). Las funciones son invocadas por la aparición de sus nombres en expresiones, junto con los parámetros actuales requeridos. El valor producido por la ejecución de una función es retornado al código llamador, efectivamente reemplazando la llamada misma. Define un nuevo operador definido por el usuario (abstracción a nivel expresión).
→ Es una unidad que evalúa una expresión y al ser invocada produce un resultado.

{ Ejemplos – Funciones en LP Imperativos y Funcionales }

LP Imperativos

```
function Pot(n,m:int):int
begin
if n=1   then Pot:=1
         else Pot:= Pot(n,m-1)*n
end;
```

LP Funcionales

```
function Pot(n,m:int):int
if n=1   then 1
         else Pot(n,m-1)*n
end;
```

En algunos lenguajes se hace el *return* del valor (el valor computado no se asigna a una variable).
Se utiliza el nombre de la función como área de almacenamiento. Así el nombre de la función tiene dos semánticas, dependiendo de dónde aparezca:

1. Área de almacenamiento auxiliar (a la izquierda)
2. Invocación recursiva (a la derecha)

En los LP funcionales se evita la doble semántica sobre el nombre de la función.

AMBIENTE DE REFERENCIAMIENTO LOCAL

Los subprogramas pueden definir sus propias variables, definiendo por lo tanto un ambiente de referenciamiento local. Las variables definidas dentro de los subprogramas son llamadas *variables locales,* porque su alcance es usualmente el cuerpo del subprograma en el que están definidas.

Si las *variables locales son dinámicas basadas en pila*, son ligadas al almacenamiento cuando el subprograma comienza a ejecutarse, y son desligadas cuando la ejecución termina.

+ Es esencial que los programas recursivos tengan variables dinámicas basadas en pila, dada la *flexibilidad* que brindan.

+ Otra ventaja que tienen las variables dinámicas basadas en pila, es que el *almacenamiento* para las variables locales en un subprograma activo puede ser *compartido* con las variables locales de todos los subprogramas inactivos.

- Requieren *costo de tiempo* para alocar, inicializar (cuando es necesario), y desalocar las variables en cada activación de un subprograma.

- Su *acceso* debe ser *indirecto*, mientras que el acceso a variables estáticas puede ser directo.

C, C++ → Las variables locales en los subprogramas son dinámicas basadas en pila, salvo que sean explícitamente declaradas como *static.*

Java, Pascal, Ada → Únicamente variables dinámicas basadas en pila.

PARÁMETROS

Hay dos maneras de hacerle llegar a una unidad los datos sobre los cuales debe computar:

- A través del acceso a variables no locales visibles para la unidad
- A través del pasaje de parámetros

Si los datos son accedidos a través de *variables no locales*, la única forma de que la computación pueda efectuarse sobre diferentes datos es asignando nuevos valores a tales variables no locales entre distintas llamadas al subprograma. El uso de variables no locales puede conducir a programas menos confiables.

Los datos que son pasados como *parámetros* son accedidos a través de nombres que son locales a la unidad. El pasaje de parámetros es más flexible, y define computación parametrizada. Puede realizar su computación sin importar qué datos reciba a través de sus parámetros (siempre que los tipos sean compatibles con los esperados por el subprograma).

Terminología – Conceptos

- *Argumento*: Datos que forman la entrada de una unidad: datos no locales, parámetros, y archivos externos (idealmente todos deberían ser parámetros).

- *Resultado*: Variables no locales o archivos externos que también pueden modificarse además de los parámetros (se mantienen en el área de almacenamiento temporal). Para las funciones no se retornan a través de los parámetros.

- *Parámetro Formal*: Dato particular dentro del área de referenciamiento de la unidad. Se declaran en el encabezado de la unidad, en general, aclarando su tipo. No son variables en el sentido usual: son ligados al almacenamiento únicamente cuando el subprograma es llamado, y la ligadura es usualmente a través de otras variables del programa.

- *Parámetro Actual*: En la invocación de una unidad debe especificarse el nombre de la misma, y una lista de parámetros a ser ligados con los parámetros formales del subprograma. Son variables, constantes, o expresiones que pueden ser locales, no locales, o a su vez parámetros. La elección de qué está permitido como parámetro actual afecta la implementación del lenguaje.

Correspondencia entre Parámetros Formales y Actuales

- *Posicional (Positional parameters)*: La ligadura de parámetros actuales con formales se hace por posición simple. El primer parámetro actual se liga al primer parámetro formal, y así sucesivamente. Es el más habitual.
  + Simple y confiable (bueno cuando la lista de parámetros es relativamente corta).

- *Por nombre (Keyword parameters)*: El nombre del parámetro formal al cual se quiere corresponder el parámetro actual se especifica junto con el parámetro actual → Se utiliza en general para librerías (Ej: Ada admite este tipo de correspondencia).
  + Se independiza del orden, los parámetros actuales pueden aparecer en cualquier orden (bueno cuando la lista de parámetros es larga).
  - El usuario de la rutina debe conocer los nombres de los parámetros formales.

La correspondencia por nombre está vinculada con el concepto de *inicialización implícita.* Así, en la invocación no se incluye la lista completa de parámetros, sino sólo la de aquellas variables que interesa que tengan un valor diferente al establecido en la definición *(por omisión)*.

Ej: procedure P(A,B: int=1; C:real=0) → llamada: P(A=100)

Algunos LP permiten correspondencia posicional y por nombre, pero no la combinación en una misma llamada. Otros lo permiten pero con una restricción: una vez que se especifica una correspondencia por nombre, todos los parámetros restantes en la lista de parámetros también deben ser especificados de este modo.
Otros lenguajes permiten posicional y por omisión → Riesgos si se combina con sobrecarga

Ej: llamada: P(100)→ Corresponde posicional, luego se agota la lista de parámetros, por se toman los valores de la definición para el resto.

Tipos de Parámetros
Los parámetros pueden ser *datos* o *unidades*. Los *datos* como parámetros, a su vez, pueden ser *argumentos* o *resultados*.

Modelos de Pasaje de Parámetros
Los métodos de pasaje de parámetros son las formas en las cuales los parámetros son transmitidos a/de los subprogramas llamados.

Los parámetros formales se caracterizan por uno de tres *modelos semánticos*:

- *in mode:* Pueden recibir datos del correspondiente parámetro actual.
- *out mode:* Pueden transmitir datos al parámetro actual.
- *in-out mode:* Combinación de los dos anteriores.

Métodos de Pasaje de Parámetros para Datos
Si se trata de datos, el pasaje de parámetros puede ser:

- *Por Referencia*
- *Por Copia*
  - *Valor*
  - *Resultado*
  - *Valor-Resultado*
- *Por Nombre*
- *Por Necesidad*

- *Por Referencia*: Es una implementación del modo *in-out*. El método de pasaje por referencia provee un camino de acceso (a veces sólo una dirección) al subprograma llamado, dando un camino de acceso a la celda que almacena el parámetro actual.
  El acceso y modificación del parámetro formal afecta directamente al parámetro actual, y por lo tanto al estado del ambiente de referenciamiento de la unidad llamadora.
  + Eficiente en términos de tiempo y espacio
  - Su uso puede provocar efectos colaterales y aliasing
  - Acceso más lento (se requiere un nivel adicional de indirección)

- *Por Copia*: Este tipo de pasaje de parámetros utiliza espacio de memoria adicional para los parámetros formales. Se requiere entonces almacenamiento extra, y el uso de operaciones de copia.

  Por Valor: Implementación del modo *in*. El parámetro formal se inicializa con el valor del parámetro actual en el momento de la llamada. Luego, el parámetro formal actúa como una variable local en el subprograma.
  + Acceso más eficiente (si se otorgara la dirección de la celda debería garantizarse que sea sólo lectura)
  - Operaciones de copia y almacenamiento en espacio extra (sobre todo si es un parámetro grande)

  Por Resultado: Implementación del modo *out*. No se transmite ningún valor al subprograma. El parámetro formal correspondiente actúa como una variable local. El parámetro actual recibe el valor del parámetro formal al terminar la ejecución normal de la unidad llamada (exactamente antes de que el control sea devuelto al llamador – cuando se hace el *return*).
  - Requiere operaciones de copia y almacenamiento en espacio extra
  - Puede haber colisión de parámetros actuales (Ej: llamada con el mismo parámetro– p(a,a))
  - Debe decidirse en qué momento se calcula la dirección de retorno para los parámetros (al momento de la llamada, o del retorno)

Por Valor-Resultado: Implementación del modo *in-out*, que corresponde a una combinación de los dos anteriores. El valor del parámetro actual es utilizado para inicializar el parámetro formal, el cual es tratado luego como una variable local. Cuando la unidad termina, se copia el valor del parámetro actual en el formal.
- Mismas desventajas que pasaje de parámetros por resultado.

- *Por Nombre*: Implementación del modo *in-out*. Toda aparición del parámetro formal en la unidad llamada, se reemplaza textualmente por el parámetro actual. El parámetro formal es ligado a un método de acceso al momento de la llamada al subprograma, pero la ligadura a un valor o una dirección es retrasada hasta que el parámetro formal sea referenciado → El parámetro actual se computa cada vez que el parámetro formal es referenciado, y el cómputo tiene lugar en el ambiente de referenciamiento de la unidad llamadora.
  - Muy costoso
  + Brinda flexibilidad

- *Por Necesidad*: Implementación del modo *in-out*. El parámetro actual se computa la primera vez que el parámetro formal es referenciado en una expresión. El cómputo se realiza en el ambiente de referenciamiento de la unidad llamadora.

Obs: Para el pasaje de parámetros por necesidad y por nombre, si se quiere utilizar en modo out, debe utilizarse como parámetro una variable.

Chequeo de Tipos de Parámetros

La consistencia entre parámetros formales y actuales puede resolverse en los LP *sin chequeo* o con *chequeo coherente* (aplicado a expresiones y asignaciones).
La confiabilidad del SW demanda que se efectúe el chequeo de tipos correspondiente, ya que sino pequeños errores de tipeo pueden causar grandes errores en el programa (y pasar inadvertidos por el compilador o en tiempo de corrida).

Unidades como Parámetros

A veces es conveniente pasar unidades como parámetro a otros subprogramas. Surgen en estos casos algunos aspectos de diseño que deben considerarse:
- Decidir si se *controlará el tipo de los parámetros.*
- Decidir cuál será el *ambiente de referenciamiento para el subprograma pasado como parámetro* (para lenguajes que admiten subprogramas anidados).

{ Ejemplos – Unidades como parámetros en LP }

Fortran → controla el tipo de los parámetros

C y C++ → No permiten pasar unidades como parámetro, pero sí punteros a unidades (en este caso se hace chequeo de tipos)

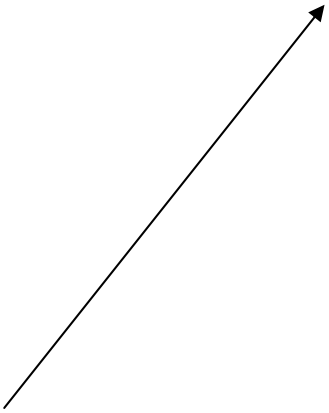
Ada → No permite pasar unidades como parámetros

Alternativas para el *ambiente de referenciamiento para la ejecución del subprograma pasado por parámetro*:

1. El ambiente de referenciamiento de la unidad que recibió como parámetro el subprograma (*Ligadura Superficial*).
2. El ambiente de referenciamiento de la definición de la unidad que fue pasada por parámetro (*Ligadura Profunda*).
3. El ambiente de referenciamiento de la sentencia de llamada que pasó la unidad como parámetro actual. Ambiente de la unidad que efectuó la llamada utilizando el subprograma como parámetro. (*Ligadura Ad-Hoc*).

Ejemplo:

```
function sub1()
{   var x;

    function sub2()
    {
       imprimir(x);
    }

    function sub3()
    {
       var x;
       x=3;
       sub4(sub2);
    }

    function sub4(subx)
    {
       var x;
       x=4;
       subx();
    }

    x=1;
    sub3();
}
```



Considerando la ejecución de sub2 desde sub4:

1. *Superficial*: El ambiente de referenciamiento es el de sub4, por lo tanto se imprime el valor 4.

2. *Profunda*: El ambiente de referenciamiento es el de sub1, por lo tanto está ligado a la variable local, imprimiendo 1.

3. *Ad-hoc*: El ambiente de referenciamiento corresponde al de sub3, y por lo tanto se imprime el valor 3.

Observaciones:

- La ligadura superficial no es buena para lenguajes con alcance estático con subprogramas anidados.
- En algunos casos, una unidad que declara a un subprograma también lo pasa como parámetro. En esos casos, la ligadura profunda y la ad-hoc son iguales.
- La ligadura ad-hoc no es utilizada, porque puede asumirse que el ambiente en el cual un subprograma es recibido como parámetro no tienen conexión natural con el subprograma pasado.

UNIDADES SOBRECARGADAS

Una *unidad sobrecargada* es aquella que tiene el mismo nombre que otra unidad en el mismo ambiente de referenciamiento. Cada versión de la unidad sobrecargada debe tener un protocolo único (debe ser diferente del resto en cuanto al número, orden o tipo de los parámetros, o en el tipo de retorno si es una función).

C++, Java, Ada y C# → Incluyen unidades predefinidas sobrecargadas (Ej: constructores sobrecargados en clases). Además permiten al usuario escribir múltiples versiones de unidades con el mismo nombre.

Ada → El valor retornado por una unidad funcional sobrecargada es utilizado para desambiguar las llamadas cuando tienen los mismos parámetros.

UNIDADES POLIMÓRFICAS/GENÉRICAS

Una *unidad polimórfica* o *genérica* es aquella que toma parámetros de diferentes tipos en diferentes activaciones. Las unidades sobrecargadas proveen polimorfismo ad-hoc.

El *polimorfismo paramétrico* es provisto por una unidad que toma un parámetro genérico que es utilizado en una expresión de tipo que describe el tipo de los parámetros de la unidad.

Ada y C++ → Proveen un tipo de polimorfismo paramétrico en tiempo de compilación.
ML → Politipos

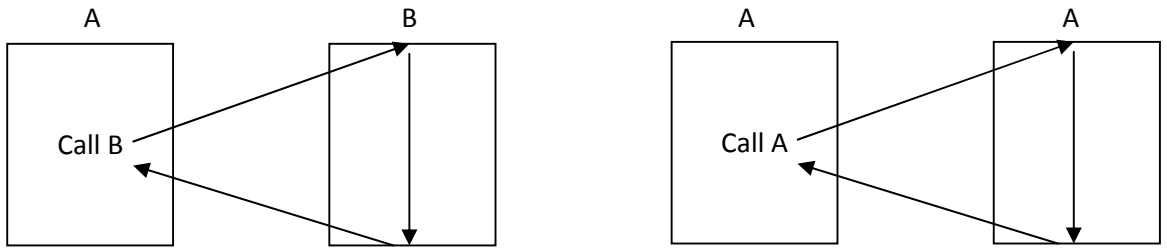
ESTRUCTURAS DE CONTROL

Las estructuras de control para la ejecución de unidades son:

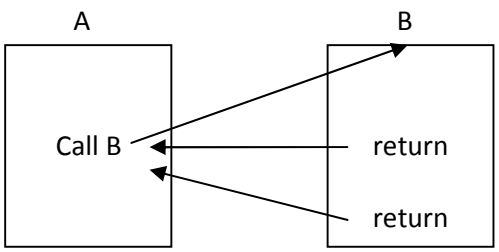
- <u>*Unidades Jerárquicas*</u>: *Invocación* a la unidad subordinada a partir de su nombre. *Retorno* a la unidad llamadora referenciada implícitamente. (Ej: C)

    1. La unidad llamadora se suspende
    2. El subprograma corre por completo
    3. Árbol de llamadas donde cada unidad llama a otra, sin ciclos

ABSTRACCIÓN PROCEDURAL

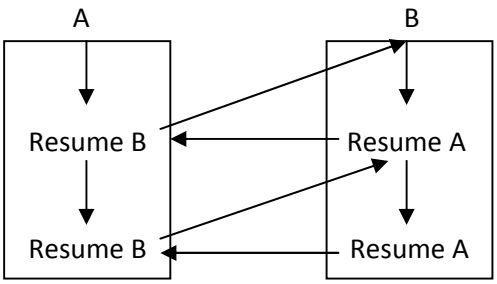


ABSTRACCIÓN FUNCIONAL



- *Unidades Simétricas*: Esquema de control mutuo. Ejecución entrelazada. Comportamiento sensible a la historia. (Ej: C# - foreach) → (Ver más abajo *Corrutinas*)

    1. Uso de *corrutinas* para simular concurrencia
    2. Esquema de relación útil para varias aplicaciones
    3. Indispensable para sincronizar las variables globales o compartidas



Comportamiento: Invocar – Suspender – Reanudar – Terminar

- *Excepciones*: Las excepciones son eventos inusuales que deberían controlarse sin oscurecer la legibilidad. Un SW que provee excepciones goza de la propiedad de confiabilidad y se dice tolerante a fallas.
  Las excepciones pueden ser:

- Detectadas por el HW o por el SW
- Situaciones de error o no previstas
- Controladas a través de *mecanismos generales* o específicos

MECANISMOS NO ESPECÍFICOS PARA MANEJAR EXCEPCIONES → La unidad detecta la situación y existen distintas alternativas para manejarla:

Por la misma unidad	Decidiendo qué acciones ejecutar
	Invocando a procedimientos que recibió como parámetros
Por la unidad llamadora	A través de un valor que le indica qué situación se produjo
	A través de la transferencia de control

MECANISMOS ESPECÍFICOS → Aspectos de diseño:

- Predefinidos o definidos por el programador
- Habilitar o deshabilitar
- Declaración y alcance
- Señalización
- Captura y ligadura al manejador
- Propagación
- Continuación
- Comunicación

{ Ejemplo - Mecanismos Específicos para Manejar Excepciones en LP }

Ada → El lenguaje provee cinco categorías de excepciones predefinidas (*constraint_error, numeric_error, program_error, storage_error, tasking_error*). El manejador de cada excepción puede ser redefinido.

Permite la definición de nuevas excepciones a través de declaraciones las cuales son tratadas al igual que las predefinidas, con la diferencia de que deben ser levantadas en forma explícita. La *declaración* determina el *alcance*, el *raise* la *señalización*.

Los manejadores de excepciones pueden ser incluidos en bloques o en el cuerpo de unidades, paquetes o tareas. Sin importar dónde aparezcan, los manejadores están agrupados en la cláusula *exception*, la cual debe estar ubicada al final del bloque o unidad. La cláusula *exception* resuelve la ligadura al manejador.

Si la unidad que levanta la excepción tiene un manejador para ella, el control se transfiere al manejador. Las acciones que se deberían haber ejecutado luego de la sentencia que provocó la excepción son salteadas. Se ejecuta el manejador y el programa continúa la ejecución como si la unidad que levantó la excepción hubiese terminado normalmente. Retoma la ejecución desde el *end* del bloque o unidad que causó la excepción. Si la unidad en ejecución no provee un manejador para la excepción, la unidad termina y la excepción es propagada al llamador. La propagación significa que la excepción es relanzada en otro contexto.

- *Eventos*: Por conveniencia, en algunos casos, los programas son estructurados como *sistemas reactivos*. Sistemas donde determinados eventos que ocurren en el entorno determinan el orden de ejecución (Por ejemplo: sistemas reactivos, interfaces gráficas, sistemas operativos). En la programación basada en eventos el control de ejecución se basa en acciones producidas por el usuario o por el entorno.

  Un *evento* es una notificación de que el usuario ha realizado una acción que requiere respuesta por parte del sistema. Un *manejador de eventos* es un segmento de código que se ejecuta en respuesta a un evento.
  Ej: *on* <event>
  when <condition>
  do <action>

  Una interfaz gráfica es una *colección de componentes* con una *representación gráfica* (botones, etc.) y capacidad para *percibir eventos* generados por el usuario. Una componente está asociada a un *objeto gráfico* que puede interactuar con el usuario. Un usuario realiza una *acción* que provoca un evento ante el cual una componente tiene una *reacción*.

  La *implementación* de una interfaz gráfica consiste en:
  - Crear un objeto gráfico para cada componente de la GUI e insertarlo en otras componentes contenedoras.
  - Definir el comportamiento de las componentes reactivas en respuesta a las acciones del usuario.

  El *desarrollo* de una interfaz gráfica está fuertemente ligado a los conceptos de *evento, herencia, polimorfismo y ligadura dinámica*.

- *Unidades Planificadas*: El concepto de *unidad planificada* aparece como una relajación de la máxima que establece que la ejecución de toda unidad debe comenzar inmediatamente a continuación de su llamada.

    1. Las unidades planificadas pueden ejecutarse antes o después que otras unidades.

        Ej: *call B after A* → Se planifica la ejecución de B para algún momento posterior a la culminación de la ejecución de A.

    2. Pueden ejecutarse cuando una expresión booleana arbitraria es verdadera.

        Ej: *call B when X=5 and Y*Z>24* → La ejecución de B toma lugar cuando se garantizan determinadas condiciones.

    3. Pueden ejecutarse sobre la base de una escala temporal simulada.

        Ej: *call B at time = 25* o *call B at time = CurrentTime+24*

    4. Pueden ejecutarse de acuerdo a un orden de prioridad.

        Ej: *call B with priority 5* → La activación de B tendrá lugar cuando no haya otra unidad con mayor prioridad planificada.

    Para programas con este tipo de unidades, ya no hay programa principal. Se cuenta con un programa planificador que va determinando qué unidades están planificadas para ejecutar, las mantiene en una especie de lista. La misma mantiene el orden en el cual las unidades se activarán o ejecutarán.

    Esta característica aparece en lenguajes diseñados para realizar simulación discreta de sistemas (Ej:SIMULA)

- *Unidades Paralelas*: En este tipo de control de ejecución de unidades se elimina la restricción de ejecutar en forma secuencial, ya que varias unidades pueden ejecutarse simultáneamente. Las unidades que pueden ejecutarse concurrentemente con otras unidades son llamadas *tareas* o *procesos*. (Ej: Java – thread).
  Hasta ahora las estructuras de control para unidades están basadas en la existencia de un hilo de ejecución predeterminado que describe la ejecución del programa.

  Un *sistema multiprocesador* tiene varias unidades de procesamiento (CPU) compartiendo memoria. Un *sistema computacional distribuido* o *paralelo* está conformado por varias computadoras, cada una con su propia CPU y memoria, conectadas entre sí de manera que todas pueden comunicarse.

  Principios de la programación paralela:
  1. *Definición de variables*: declaración sobre qué variables se requiere sincronización y sobre cuáles no.
  2. *Composición paralela*: agregado de la composición paralela que ocasiona la ejecución de más de un hilo.
  3. *Estructura del programa*: Transformacional (obtener una salida de un grupo de valores de entrada) o Reactiva (el sistema reacciona a múltiples eventos).
  4. *Comunicación*: Memoria compartida o Mensajes.
  5. *Sincronización*: en algunos sistemas se requiere un ordenamiento en la ejecución de los hilos.

Corrutinas – Unidades Simétricas(Sebesta)
Una *corrutina* es una clase especial de subprograma. En lugar de la relación *master-slave* entre el llamador y el subprograma llamado, las corrutinas llamadora y llamada están sobre la misma base. El mecanismo de control de las corrutinas es llamado *modelo de control de unidades simétrico*.

Las corrutinas tienen múltiples puntos de entrada, que son controlados por las mismas corrutinas. También tienen medios para mantener su estado entre activaciones, por lo tanto pueden ser sensibles a la historia y tener variables locales estáticas.

Ejecuciones secundarias de una corrutina usualmente comienzan en puntos que no son el comienzo. Por este motivo, la invocación de una corrutina es llamada *resume* en vez de *call.* Cuando se retoma una corrutina (resume), continúa su ejecución inmediatamente después de la instrucción usada para transferir el control a otra parte.

En general, las corrutinas son creadas en una aplicación por una unidad de programa llamada *master unit*, la cual no es una corrutina. Cuando son creadas, ejecutan su código de inicialización y retornan el control a la master unit. Cuando toda una familia de corrutinas es construida, el programa maestro retoma una de las corrutinas, y los miembros de la familia de corrutinas se retoman entre sí en algún orden, hasta que sus trabajos son completados (si pueden ser completados). Este es el mecanismo para finalizar la ejecución de la colección de corrutinas, cuando es deseado.

Cuando un conjunto de programas ejecutando en un sistema parecen estar corriendo concurrentemente pero comparten el procesador, se llama *quasi-concurrencia*.

CAPÍTULO XIII – Concurrencia

13.1 INTRODUCCIÓN

La *concurrencia* puede ocurrir a cuatro *niveles*:

- *Nivel Instrucción*: Ejecutar dos o más instrucciones máquina simultáneamente.
- ***Nivel Sentencia de Alto Nivel***: Ejecutar dos o más sentencias del lenguaje de programación simultáneamente.
- ***Nivel Unidad***: Ejecutar dos o más subprogramas-unidades simultáneamente.
- *Nivel Programa*: Ejecutar dos o más programas simultáneamente.

~~Como los lenguajes no proveen ítems para realizar concurrencia a nivel instrucción ni a nivel programa, sólo se analizarán los dos niveles restantes.~~

13.1.1 Arquitecturas Multiprocesador

Evolución de las Arquitecturas Multiprocesador:

- *Década del 50*: Un procesador de propósito general, y uno o más procesadores de propósito especial destinados a las operaciones de entrada-salida. Este esquema no requiere soporte en el LP.

- *Comienzo de la década del 60*: Múltiples procesadores completos, utilizados para concurrencia a nivel programa. Estos procesadores eran utilizados por un planificador de tareas del SO, quien distribuía trabajos de la cola en diferentes procesadores.

- *Mediados de la década del 60*: Múltiples procesadores parciales, utilizados para concurrencia a nivel instrucción. Estos procesadores eran alimentados con instrucciones de una misma secuencia. El compilador debe decidir qué instrucciones pueden ejecutarse simultáneamente, y planificarlas en forma adecuada.

Categorías de Computadoras Multiprocesador:

- *Arquitectura SIMD:* Instrucciones Simples – Múltiples Datos. Computadoras que tienen múltiples procesadores que ejecutan la misma instrucción simultáneamente, cada una sobre diferentes datos. Características:
  1. Cada procesador tiene su memoria local.
  2. Un procesador controla la operación del resto de los procesadores. Debido a que todos los procesadores, excepto el controlador, ejecutan la misma instrucción al mismo tiempo, no se requiere sincronización en SW.

- *Arquitectura MIMD:* Múltiples Instrucciones – Múltiples Datos. Computadoras con múltiples procesadores que operan independientemente, pero cuyas operaciones pueden ser sincronizadas.
  Características:
  1. Cada procesador ejecuta su propia secuencia de instrucciones.
  2. Aparecen en dos configuraciones diferentes: sistemas distribuidos, o sistemas de memoria compartida.
  3. Requiere sincronización, y soporta concurrencia a nivel unidad.

13.1.2 Categorías de Concurrencia (Categorías de control de unidad concurrente)

- *Concurrencia Física:* Asumiendo que hay más de un procesador disponible, varias unidades del mismo programa son ejecutadas simultáneamente.
  ~~Múltiples procesadores independientes, múltiples *hilos de control*. Un *hilo de control* en un programa es la secuencia de puntos del programa alcanzados mientras el control fluye por el programa.~~

- *Concurrencia Lógica:* Es una relajación del concepto de concurrencia física. Permite al programador y al SW de aplicación suponer la existencia de múltiples procesadores. La ejecución real del programa sucede sobre un solo procesador utilizando *interleaving*.

  → Desde el punto de vista del programador y del diseñador del lenguaje, la concurrencia lógica es igual que la física.

La apariencia de concurrencia física es presentada por tiempo compartido sobre un procesador (el SW se puede diseñar de manera que simule la presencia de múltiples hilos de control). Cada procesador se hace cargo de un solo hilo de control.

Cuando un programa con múltiples hilos se ejecuta en una máquina con un solo procesador, los hilos son mapeados a un solo hilo, convirtiéndose así en un programa multi-hilos virtual.

Los lenguajes que proveen corrutinas, son a veces llamados *quasi-concurrentes*, ya que sólo cuentan con un hilo de control.

13.2 CONCURRENCIA A NIVEL UNIDAD

Una *tarea* ~~o *proceso*~~ es una unidad que puede ser ejecutada en forma concurrente con otras unidades del mismo programa. Cada tarea en un programa puede proveer un hilo de control.

Características que diferencian una tarea de una unidad ordinaria:

- Una tarea puede ser comenzada en forma implícita, mientras que una unidad debe ser explícitamente invocada.
- Cuando una unidad comienza la ejecución de una tarea, no necesita esperar que la tarea complete su ejecución para poder continuar con la suya (no necesita suspenderse).
- Cuando se completa la ejecución de una tarea, el control puede o no retornar a la unidad que comenzó su ejecución.

Categorías de Tareas:

- *Heaviweight tasks (tareas pesadas):* Se ejecutan en su propio espacio de direcciones y tienen sus propias pilas de ejecución.
- *Lightweight tasks (tareas livianas):* Corren todas en el mismo espacio de direcciones y utilizan la misma pila de ejecución.

Una tarea se puede comunicar con otras a través de variables no locales compartidas, a través de pasaje de mensajes, o a través de parámetros. Si una tarea no se comunica con otras ni afecta su ejecución (de ninguna manera), se dice *disjunta*.

La ***sincronización*** es el mecanismo que controla el orden en el cual se ejecutan las tareas. Cuando las tareas comparten datos se requieren dos tipos de sincronización:

- *Sincronización Cooperativa:* La tarea A debe esperar que la tarea B complete una actividad específica, antes de que A pueda continuar su ejecución (Ej: Productor-Consumidor).

- *Sincronización Competitiva:* Dos o más tareas deben usar un recurso que no puede ser utilizado en forma simultánea (evita que dos tareas accedan a una estructura de datos compartida al mismo tiempo). Para proveer este tipo de sincronización debe garantizarse acceso mutuamente excluyente a los datos compartidos. (Ej: Contador compartido).

Condición de competencia (race condition): Dos o más tareas compiten por el uso de un recurso compartido, y el comportamiento del programa depende de qué tarea lo toma primero.

Al proveer sincronización se requieren mecanismos para demorar la ejecución de una tarea. El ***planificador*** controla la ejecución de las tareas, mapea la ejecución de tareas sobre los procesadores disponibles (maneja el compartimiento de procesador).

Las tareas pueden estar en diferentes estados:

1. *Nueva:* Tarea creada pero aún no comenzada.
2. *Ejecutable o Lista:* Lista para ejecutar pero no corriendo (el procesador no está disponible – esperando).
3. *Corriendo:* Ejecutando actualmente
4. *Bloqueado:* Estuvo ejecutándose, pero momentáneamente no puede continuar (usualmente está esperando por la ocurrencia de un evento).
5. *Muerta:* No está más activa. Una tarea muere cuando se completa su ejecución, o es matada por el programa.

Asociado con la ejecución concurrente de tareas y el uso compartido de recursos está el concepto de *vivacidad (liveness)*. Significa que si un evento (terminación completa del programa) se supone que suceda, eventualmente ocurrirá → Hay progreso continuo

En un entorno de *programación secuencial* significa que eventualmente la unidad se completa. En un entorno de *programación concurrente*, las tareas fácilmente pierden la propiedad de *liveness*. Si todas las tareas en un ambiente concurrente pierden su *vivacidad*, el sistema se encuentra en deadlock.

13.2.3 Aspectos de Diseño

- *Sincronización competitiva y cooperativa*
- *Cómo controlar la planificación de tareas?*
- *Cómo y cuándo realizar las tareas y terminar su ejecución?*
- *Cómo y cuándo son creadas las tareas?*

Algunas soluciones alternativas para los problemas de diseño para concurrencia son: *semáforos*, *monitores*, *y pasaje de mensajes* ← MECANISMOS DE SINCRONIZACIÓN

13.3 SEMÁFOROS

Un *semáforo* es una estructura de datos, que consiste de un contador y una cola para almacenar descriptores de tareas. Un *descriptor de tarea* es una estructura de datos que almacena toda la información relevante acerca del estado de ejecución de una tarea.

→ Es un mecanismo que puede ser utilizado para proveer *sincronización de tareas*.

Observaciones:

- Pueden utilizarse para implementar *guardas* sobre el código que accede a estructuras de datos compartidas (para garantizar el acceso exclusivo de las tareas a las ED compartidas).
- Las dos operaciones provistas para semáforos se conocen como *wait* y *signal* (release).
- Se pueden utilizar tanto para sincronización cooperativa como competitiva.

13.3.2 Sincronización Cooperativa – Ejemplo: Buffer Compartido (Productor-Consumidor)

El *buffer* es implementado como un TDA donde todo dato entra al buffer a través de la operación *DEPOSIT*, y sale a través de la operación *FETCH*. La operación *DEPOSIT* necesita chequear si el buffer tiene posiciones libres, y la operación *FETCH* necesita chequear si el buffer contiene al menos un elemento.

Se utilizarán dos *semáforos contadores* (*emptyspots* y *fullspots*) para llevar un recuento de los lugares vacíos y ocupados en el buffer. Las colas de espera de estos semáforos almacenan las tareas que fueron bloqueadas al intentar acceder al semáforo.

Operación DEPOSIT

- Chequea el semáforo *emptyspots* para detectar si hay lugar en el buffer.
  → Si hay lugar, el contador del semáforo *emptyspots* es decrementado y el elemento es insertado.
  → Si no hay lugar, el llamador es almacenado en la cola del semáforo *emptyspots*.
- Cuando la operación termina se debe incrementar el contador del semáforo *fullspots*.

Operación FETCH

- Chequea el semáforo *fullspots* para detectar si hay algún elemento almacenado en el buffer.
  → Si hay elementos, el contador del semáforo *fullspots* es decrementado y el elemento es removido.
  → Si no hay elementos, el llamador es almacenado en la cola del semáforo *fullspots*.
- Cuando la operación termina se debe incrementar el contador del semáforo *emptyspots*.

La interacción de las operaciones *DEPOSIT* y *FETCH* con los semáforos es realizada a través de las operaciones *wait* y *signal* de los semáforos.

Productor:

```
while (true) do
{
        Producir un elemento
        wait(emptyspot)
        DEPOSIT(elemento)
        signal(fullspot)
}
```

Consumidor:

```
while (true) do
{
        wait(fullspot)
        FETCH(elemento)
        signal(emptyspot)
        Consumir el elemento
}
```

13.3.3 Sincronización Competitiva

Los semáforos para sincronización competitiva están pensados para permitir el acceso de un solo proceso a los datos por los que compiten. Los semáforos utilizados en este caso son llamados *semáforos binarios*, ya que sólo toman los valores 0 y 1.
Es importante notar que las operaciones *wait* y *signal* deben ser atómicas.

Para el ejemplo del Productor-Consumidor con buffer compartido es necesario considerar un semáforo adicional binario (*Access*) para controlar el acceso al buffer (acceso exclusivo).

Productor:

```
while (true) do
{
        Producir un elemento
        wait(emptyspot)
        wait(access)
        DEPOSIT(elemento)
        signal(access)
        signal(fullspot)
}
```

Consumidor:

```
while (true) do
{
        wait(fullspot)
        wait(access)
        FETCH(elemento)
        signal(access)
        signal(emptyspot)
        Consumir el elemento
}
```

13.3.4 Evaluación

El mal uso de los semáforos puede causar fallas en la sincronización cooperativa. Puede producirse un *overflow* en el buffer si se omite accidentalmente el *wait* del semáforo *fullspots*.
De la misma manera pueden producirse fallas en la sincronización competitiva, conduciendo a deadlocks en caso de omitirse el *signal* del semáforo binario.

"Los semáforos son una herramienta de sincronización ideal para los programadores que no cometen errores"

13.4 MONITORES

Una solución a algunos problemas de los semáforos en un entorno concurrente es encapsular las ED compartidas y sus operaciones, y ocultar su representación, para así restringir el acceso (hacer un TDA).
Un *monitor* es un tipo de dato abstracto para datos compartidos. Muchos lenguajes proveen la facilidad de definir monitores, entre ellos: Concurrent Pascal, Modula, Ada, Java, C#.

13.4.2 Sincronización Competitiva

Los datos compartidos están ahora dentro del monitor, en lugar de en las unidades clientes. Todos los accesos a los datos dentro del monitor se realizan a través de sus operaciones:

- La implementación del monitor asegura el acceso sincronizado, permitiendo un solo acceso a la vez.
- Las llamadas a las unidades del monitor son implícitamente encoladas si el monitor está ocupado en el momento de realizar la llamada.

13.4.3 Sincronización Cooperativa

Si bien el acceso mutuamente exclusivo a los datos compartidos es intrínseco con un monitor, la cooperación entre procesos sigue siendo un trabajo de programación. El programador debe seguir garantizando, en la implementación, que no se produzca overflow o underflow del buffer (para el caso del productor-consumidor).

Los lenguajes proveen diferentes maneras de realizar sincronización cooperativa, las cuales terminan estando relacionadas con los semáforos.

13.4.5 Evaluación

- Brindan un mejor soporte para sincronización competitiva que los semáforos.
- Los semáforos pueden utilizarse para implementar monitores.
- Los monitores se pueden utilizar para implementar semáforos.
- El soporte para sincronización cooperativa es muy similar al que se provee con semáforos, por lo que tiene los mismos inconvenientes.

13.5 PASAJE DE MENSAJES

El pasaje de mensajes es un modelo general para concurrencia que permite modelar tanto semáforos como monitores, y que no es solo para sincronización competitiva. Puede ser tanto *sincrónico* como *asincrónico*.

Pasaje de Mensajes Sincrónico: Las tareas pueden ser diseñadas para que suspendan su ejecución en algún punto, ya sea porque están inactivas o porque necesitan información de otra unidad antes de continuar.
Si la tarea A quiere enviarle un mensaje a la tarea B, y B está esperando el mensaje, el mensaje puede ser transmitido. Esta transmisión se conoce como *rendezvous*.

→ Para que el *rendezvous* se produzca, el emisor y el receptor del mensaje deben querer que la comunicación se produzca: Si se envía un mensaje que aún no puede ser recibido (el receptor no desea recibirlo aún) el emisor se suspende hasta que se produzca el encuentro. Análogamente si una tarea desea recibir un mensaje que aún no ha sido enviado, el receptor se suspende.

- Existe un mecanismo que permite a una tarea indicar cuándo está disponible para recibir mensajes.
- Las tareas necesitan algún modo de recordar quién está esperando por sus mensajes, y una manera “justa” de elegir el próximo mensaje.
- El rendezvous se produce en la transmisión del mensaje (cuando el mensaje de la tarea emisora es aceptado por la tarea receptora).

{ Ejemplo Pasaje de Mensajes Sincrónico – Soporte de Concurrencia en Ada }

Las *tareas* en Ada tienen *especificación* y *cuerpo*. Al igual que los paquetes, la *especificación* tiene la interfaz, que es una colección de puntos de entrada. El *cuerpo* de la tarea describe las acciones que toman lugar cuando ocurre un rendezvous.
La tarea que envía el mensaje se suspende mientras espera el mensaje de aceptación, y durante el rendezvous.
Los puntos de entrada en la especificación son descriptos como cláusulas *accept* en el cuerpo. La tarea ejecuta desde el comienzo hasta la cláusula *accept* y espera por un mensaje. Durante la ejecución de la cláusula *accept* el emisor está suspendido. Los parámetros del *accept* pueden transmitir información en una o las dos direcciones. Cada cláusula *accept* tiene asociada una cola que almacena los mensajes en espera.

Una tarea que tiene cláusulas *accept*, pero no otro código, es llamada *tarea servidora.* Una tarea que no tiene cláusulas *accept* es llamada *tarea actora*, y puede enviar mensajes a otras tareas.

El emisor debe conocer el nombre de la entrada (*entry*) del receptor, pero no viceversa (*comunicación asimétrica*).
Las tareas pueden tener más de un punto de entrada, cada uno de los cuales requiere una cláusula *accept*. El cuerpo de la tarea tiene una cláusula *accept* para cada entrada colocada en una cláusula *select* dentro de un ciclo.

Semántica de mensajes con múltiples elecciones: Al constructor usualmente se lo llama *selective wait* y suele estar en bucles infinitos.

1. Si sólo hay una cola de *entry* y no está vacía, elegir un mensaje de ella.
2. Si hay más de una cola no vacía, elegir una en forma no determinística, y de ella aceptar un mensaje.
3. Si todas están vacías entonces esperar.

(Ghezzi) *Rendezvous* → El constructor puede ser un mecanismo de alto nivel que combina sincronización y comunicación.
Una entrada (*entry*) puede ser vista como un puerto mediante el cual la tarea puede intercambiar mensajes con otras tareas que pueden recibirlos o enviarlos.
La tarea puede indicar su voluntad de aceptar el mensaje mediante un accept, si la entrada está declarada en ella. En este punto, el emisor y el receptor se encuentran (rendezvous).
Si el emisor envía un mensaje antes que el receptor quiera recibirlo (aceptar), el emisor se suspende hasta que ocurra el encuentro. Algo similar ocurre si el receptor ejecuta un *accept* para recibir antes que el emisor envíe un mensaje.

13.5.4 Sincronización Cooperativa
Cada cláusula *accept* puede tener una guarda adosada, en la forma de cláusula *when*, que puede demorar el rendezvous.
Ej. when not Lleno(buffer) → *accept* Deposit(valor) do

Cada cláusula *accept* puede estar entonces:
- Abierta: si la expresión booleana es verdadera
- Cerrada: en caso de que la expresión sea falsa

→ Las cláusulas sin guarda siempre están abiertas.

La cláusula *select* chequea las guardas de todas las cláusulas:
- Si exactamente una está abierta, se controla la cola asociada por mensajes.
- Si más de una está abierta, se elige en forma no determinística una, y se chequea por mensajes.
- Si están todas cerradas, entonces se produce un error en ejecución.

→ La cláusula *select* puede incluir una cláusula *else* para evitar el error. Cuando la cláusula *else* se completa, se vuelve a repetir el ciclo.

13.5.5 Sincronización Competitiva

Si el acceso a una ED debe ser controlado por la tarea, entonces el acceso mutuamente excluyente puede conseguirse declarando la ED dentro de la tarea.

La semántica para la ejecución de una tarea usualmente garantiza el acceso exclusivo a la estructura, porque sólo una cláusula *accept* en la tarea puede estar activa en un determinado momento. Las únicas excepciones a esto ocurren cuando las tareas están anidadas en procedimientos u otras tareas.

13.5.6 Terminación de Tareas

La ejecución de una tarea está *completa* si el control alcanza el fin del código del cuerpo. Una tarea está *terminada* si la tarea no ha creado tareas dependientes y se completa. Si una tarea se completa pero ha creado tareas dependientes, no está terminada hasta que las tareas dependientes terminen.

La cláusula *terminate* en un *select* es sólo una sentencia de terminación, que sólo puede seleccionarse cuando no hay cláusulas abiertas en el *select*. La ejecución de la cláusula *terminate* completa la tarea, y la termina si sus dependientes están terminadas.

13.5.11 Evaluación

El modelo de concurrencia basado en pasaje de mensajes es un mecanismo general poderoso.
Con la ausencia de procesadores distribuidos con memorias independientes, la elección entre monitores y pasaje de mensajes como medio para proveer sincronización competitiva es cuestión de gustos.
La sincronización cooperativa en el pasaje de mensajes es menos dependiente de un uso correcto que los semáforos (los cuales son requeridos para la sincronización cooperativa con monitores).
Para sistemas distribuidos, el pasaje de mensajes es un mejor modelo de concurrencia, porque naturalmente soporta el concepto de procesos separados ejecutando en procesadores separados.

13.6 JAVA THREADS – CONCURRENCIA EN JAVA

El modelo de concurrencia en Java es simple pero efectivo, aunque no es tan poderoso como las tareas de Ada.
La *concurrencia* en Java está dada *a nivel unidad*, y son métodos llamados *run:*

- Un método *run* puede estar en ejecución concurrente con otros procesos *run*.
- El proceso en el cual está el método *run* se llama *thread*.

La clase *thread* tiene varios métodos para controlar la ejecución de los hilos:

- *yield*: es un requerimiento del hilo en ejecución a ceder el procesador en forma voluntaria.
- *sleep*: es un método que puede provocar que el llamador bloquee el hilo.
- *join*: es un método que se utiliza para forzar a un método a demorar su ejecución hasta que el método *run* de otro hilo complete su ejecución.

Sincronización Competitiva: Un método que incluye el *modificador synchronized* (en el encabezado del método) imposibilita que otro método se pueda ejecutar sobre el objeto mientras el primero esté en ejecución (se evita que interfieran entre sí).
Si se quiere sincronizar solo parte del método se puede utilizar el *atributo synchronized* para sentencias.

Sincronización Cooperativa: La sincronización cooperativa se logra utilizando los métodos:

- *wait:* Este método debe ser llamado en un ciclo.
- *notify:* Este método es llamado para indicar a un hilo en espera que el evento que estaba esperando sucedió.
- *notifyall:* Este método despierta a todos los hilos que estén en la lista de espera del objeto.

CONCURRENCIA A NIVEL SENTENCIA

El *objetivo* es proveer un mecanismo que el programador pueda utilizar para informar al compilador de qué maneras puede mapear el programa en una arquitectura multiprocesador. Se intenta minimizar la comunicación entre procesadores y las memorias de otros procesadores.

La instrucción *forall /foreach* es un ejemplo de concurrencia a nivel de sentencia. A través de ella se especifica una lista de sentencias que pueden ejecutarse en forma concurrente.

Ej: forall(index=1:1000) lis_1(index) = lis_2(index)

Fork-Join(Concurrencia a Nivel Sentencia)

La instrucción *fork L* produce dos ejecuciones concurrentes en el programa. Una ejecución comienza en la sentencia etiquetada con T, y la otra es la continuación de la ejecución en la sentencia que sigue al *fork*.
→Divide una computación simple en dos computaciones independientes.

La instrucción *join* provee los medios para recombinar las dos computaciones concurrentes en una. Cada una de las dos computaciones debe ser unida con la otra. Si una ejecuta el *join* antes que la otra, termina su ejecución, y la otra puede continuar. Debe ejecutarse en forma atómica.

Es necesario saber el número de computaciones que deben unirse, para poder terminar todas menos una. Para esto, la instrucción *join* tiene un parámetro para especificar el número de computaciones a unir (un contador que se decrementa y termina las tareas, excepto la última cuando el contador llega a 0).
→ Une varias ejecuciones concurrentes en una sola.

Parbegin-Parend(Concurrencia a Nivel Sentencia)

La sentencia *parbegin/parend* es un constructor de alto nivel del lenguaje para especificar concurrencia, y tiene la forma: *parbegin* S_1; S_2; …; S_n *parend;* donde cada S_i es una sentencia simple.

Las sentencias encerradas entre *parbegin* y *parend* pueden ser ejecutadas concurrentemente. La sentencia S_{n+1} puede ser ejecutada sólo cuando todas las S_i i=1,…,n hayan terminado.

MECANISMOS DE SINCRONIZACIÓN
- Semáforos
- Monitores
- Pasaje de Mensajes

CONSTRUCTORES DE CONCURRENCIA
- Fork-Join (bajo nivel)
- Parbegin-Parend (alto nivel)
- Threads/Procesos

CONSIDERACIONES GENERALES SOBRE CONCURRENCIA

Dificultades de Programación Concurrente:

- Inanición
- Deadlock
- Sincronización
- No determinismo

La concurrencia en un LP se ve limitada por:

- Exclusión mutua de recursos compartidos
- Precedencia entre eventos

CAPÍTULO XI – Encapsulamiento y Abstracción

ENCAPSULAMIENTO
En la construcción de programas grandes es inevitable tener que diseñar e implementar nuevos tipos de datos:

- La actividad de *diseño* puede considerarse como la *especificación* del tipo de dato abstracto (diseño de los atributos y operaciones requeridos.
- La *implementación* acontece luego. Se elige una representación específica utilizando los constructores provistos por el lenguaje u otros tipos de datos abstractos ya definidos.

En el diseño de lenguajes el objetivo buscado es hacer que la distinción entre las diversas formas de los datos sea transparente al programador que las usa.

Los mecanismos básicos que se le proveen al programador para crear nuevos tipos de datos y operaciones para ese tipo son:

- *Datos Estructurados:* Casi todos los lenguajes proveen mecanismos para crear objetos más complejos, a partir de objetos de datos elementales.
- *Unidades:* El programador puede hacer uso de ellas para definir las funcionalidades del nuevo tipo de dato.
- *Declaraciones de Tipo:* El lenguaje incluye la habilidad para definir nuevos tipos y operaciones para esos tipos (concepto de TDA).
- *Herencia:* Los conceptos de POO y herencia incrementan la habilidad del programador para crear nuevos tipos, construir operaciones para esos tipos, y hacer que el lenguaje de programación detecte usos inapropiados del tipo.

Constructores de Encapsulamiento
Los programas grandes tienen dos necesidades especiales:
- Necesidad de *organización*, más que simple división en unidades
- Necesidad de *compilación parcial* (unidades de compilación más pequeñas que el programa completo)

La solución obvia es agrupar las unidades que están lógicamente relacionadas en una unidad que pueda ser compilada separadamente (*unidad de compilación*). Esta solución conduce al encapsulamiento.

Un *encapsulamiento* es una colección de código y datos lógicamente relacionados, que puede ser compilada sin la recompilación del resto del programa. Generalmente se utilizan como librerías, y están disponibles para reuso en otros programas, más allá del programa para el cual fueron implementadas.

{ Ejemplos – Encapsulamiento en LP }

Fortran95 → Los programas se pueden organizar anidando la definición de unidades
C → Cada archivo que contenga una o más unidades puede ser compilado independientemente. La interfaz se coloca en un archivo *header*.
Ada → Los paquetes pueden incluir la declaración de cualquier número de datos y unidades. Estos paquetes se pueden compilar en forma separada. La declaración y el cuerpo del paquete también pueden compilarse separadamente.

Encapsulamiento de Nombres

Los programas grandes definen muchos nombres globales, por lo que debe haber una forma de subdividir estos nombres en espacios lógicos. El *encapsulamiento de nombres* es usado para crear un nuevo alcance para los nombres (define un ambiente o espacio de nombres).

{ Ejemplos – Encapsulamiento de Nombres en LP }

C++ - *Namespace* → Se puede colocar cada librería en su propio espacio de nombres y cualificar las llamadas cuando se está afuera.

Java – *Packages* → Pueden contener más de una definición de clase. Los clientes del paquete pueden usar llamadas cualificadas o utilizar la declaración *import*.

Ada – *Package* → Los paquetes se definen en jerarquías que corresponden a la jerarquía de archivos. La visibilidad es otorgada mediante la cláusula *with*.

EVOLUCIÓN DEL CONCEPTO DE TIPO

Los lenguajes como FORTRAN y COBOL limitan la creación de nuevos tipos de datos, y la definición de unidades. Con la evolución del concepto de tipo de dato, se diseñaron nuevos lenguajes capaces de especificar y diseñar tipos de datos abstractos completos (Ej: *package* de Ada, *clases* de C++ o Java).

Concepto de Tipo en los LP:

Fortran →Un *tipo* determina la interpretación de un valor almacenado y permite chequeos.

Pascal → Un *tipo* es el conjunto de valores que puede tomar un dato.

SIMULA → Un *tipo* es un conjunto de valores y un conjunto de operaciones.
Un *TDA* es una entidad que encapsula valores y operaciones (sólo es accesible a través de las operaciones).
Un *TDA genérico* es una entidad que encapsula un conjunto de valores (debe incluir las operaciones) y un conjunto de valores de manera independiente.

ABSTRACCIÓN

El oscurecimiento es un objetivo de diseño de suma importancia para lograr un buen encapsulamiento. A partir del encapsulamiento se logra la programación modular, logrando:

- Mejorar la programación y la compilación
- La idea central es descomponer el problema en módulos de propósito simple, interfaz clara, e implementación eficiente

→ Para lograr estos objetivos, es necesario dividir el módulo en dos partes: *interfaz* e *implementación*.
Al usuario no le interesa cómo se implementa el módulo, ni tampoco puede accederlo.

Concepto de Abstracción

Una *abstracción* es una vista o representación de una entidad, la cual incluye sólo los atributos más significativos. Este concepto es fundamental en programación y en ciencias de la computación.

La *abstracción* permite pensar a un LP como una máquina virtual más poderosa y simple que la máquina real. El lenguaje provee abstracciones primitivas y mecanismos para definir otras nuevas: *funcionales* o *basadas en datos.*

El aumento del nivel de abstracción favorece:

- Legibilidad
- Oscurecimiento
- Reusabilidad
- Verificación
- Confiabilidad
- Extensibilidad

El concepto general detrás del diseño de abstracciones definidas por el programador es el *ocultamiento de información*. Cuando la información es encapsulada en una abstracción, significa que el usuario de dicha abstracción:

- No necesita conocer la información oculta para poder hacer uso de la abstracción
- No está permitida la manipulación directa de la información oculta aunque se desee hacerlo.

El *ocultamiento de información* es una cuestión de *diseño de programas*. Es posible en cualquier programa que sea diseñado apropiadamente independientemente del lenguaje de programación utilizado.

El *encapsulamiento* es una cuestión de *diseño del lenguaje*. Una abstracción resulta efectivamente encapsulada sólo cuando el lenguaje prohíbe el acceso a la información oculta en esa abstracción.

Abstracción de Datos - TDA

Casi todos los lenguajes de programación soportan *abstracción de procesos* a través de las unidades, ya que especifican cómo se efectúa un proceso sin proveer los detalles acerca de cómo se hace.

Los LP diseñados desde 1980 soportan en general *abstracción de datos*. La definición completa de un TDA debería estar encapsulada de manera que el usuario del tipo sólo necesite conocer el nombre del tipo, la semántica, y las operaciones disponibles.

[Sebesta]
Un *tipo de dato abstracto* es un tipo de dato definido por el usuario que satisface las siguientes condiciones:

- Los objetos del tipo son representados como unidades sintácticas simples (Modificabilidad – Organización).
- La representación del tipo es escondida del programa que utiliza estos objetos, por lo tanto las únicas operaciones posibles son aquellas provistas por la definición del tipo (Confiabilidad).

→Es un encapsulamiento que incluye la representación de datos para un tipo de dato específico, y los subprogramas que proveen las operaciones para el tipo.

[Pratt]
Un *tipo de dato abstracto* es definido como:

- Un conjunto de objetos de datos, comúnmente utilizando una o más definiciones de tipo.
- Un conjunto de operaciones abstractas sobre los objetos de datos
- Encapsulamiento de todo de manera tal que el usuario de un tipo no pueda manipular los objetos de datos del tipo, excepto mediante el uso de las operaciones definidas.

El Diseño de un TDA debe incluir: (Mínimamente)

- Unidad sintáctica para encapsular la definición del TDA
- Operaciones *built-in*: asignación, comparaciones (igualdad y desigualdad)
- Operaciones *comunes*: iteradores, constructores, destructores
- TDA parametrizados

{ Ejemplos - TDA en LP }

Ada → El constructor de encapsulamiento se llama *package*, y consta de dos partes, cada una de las cuales también es llamada *package*:
- *Especificación*: provee la interfaz del encapsulamiento
- *Cuerpo*: provee la implementación de las entidades que son nombradas en la especificación

La representación del tipo aparece en una parte de la especificación, llamada *private* part.

C++ → La *clase* es el mecanismo de encapsulamiento. Está basado en el *struct* de C y la *clase* de Simula67. Todas las instancias de la clase comparten una sola copia de las funciones miembro, pero tienen su propia copia de los datos.

La información se oculta a través de los *modificadores*:
- *Private*: para entidades ocultas
- *Public*: para entidades de interfaz
- *Protected*: para herencia

Los *constructores* son funciones para inicializar los datos de las instancias (no crean el objeto). Pueden alocar almacenamiento, si parte de los objetos es dinámica en heap. Pueden incluir parámetros para proveer parametrización de los objetos. Son llamados implícitamente cuando se crea un objeto, y también pueden ser llamados en forma explícita. El nombre del constructor debe ser el mismo que el de la clase.

Los *destructores* son funciones de limpieza luego de que una instancia es destruida, usualmente únicamente son usados para reclamar el almacenamiento del heap. Son llamados implícitamente cuando el tiempo de vida de un objeto termina. El nombre del destructor debe ser el mismo que el de la clase, precedido por el símbolo ~

Tipos de Datos Abstractos Parametrizados

Los *tipos de datos parametrizados* permiten el diseño de un TDA que puede almacenar elementos de cualquier tipo. Estos tipos de datos son también conocidos como *clases genéricas.*

[*Pratt*] → Una *definición genérica de tipo abstracto* permite especificar por separado un atributo del tipo de esta clase, de modo que se pueda dar una definición de tipo base, con los atributos como parámetros, y luego se puedan crear varios tipos especializados derivados del mismo tipo base. La estructura es similar a la de una definición de tipo con parámetros, excepto que en este caso los parámetros pueden afectar la definición de las operaciones en la definición del TDA, así como las definiciones mismas del tipo, y los parámetros pueden ser nombres de tipo y también valores.

{ Ejemplos – TDA parametrizados en LP }

Java 5.0 → Provee una forma restringida de este TDA parametrizado

C# → No provee soporte para este TDA

Ada → Provee soporte para los TDA parametrizados a través de los *Generic Packages*.
Ej: TDA Pila - Hace que el tipo stack sea más flexible haciendo que el tipo elemento y el tamaño de la pila sean genéricos.

C++ → Las clases pueden ser de alguna manera genéricas escribiendo funciones constructoras parametrizadas.

HERENCIA

Suele ocurrir que información que se conoce en una parte del programa, es necesitada y utilizada en otra parte. Esta utilización de información puede ocurrir:

- *Explícitamente*: Pasaje de parámetros, pasaje de información de los parámetros actuales a los formales. La ligadura se hace en la llamada explícita.
- *Implícitamente*: Generalmente llamada *herencia*. Sucede cuando una componente del programa recibe propiedades o características de otra componente, siguiendo una relación especial existente entre ambas componentes

Obs: La primera forma de herencia aparece en las reglas de alcance del os datos en los programas estructurados por bloques. A pesar de esto, el término *herencia* se utiliza más frecuentemente para indicar pasaje de datos y unidades entre módulos independientes de un programa (Ej: clases de C++).

Si tenemos una relación entre A y B (A => B) se dice que A es la clase *padre* o *superclase* de B, y B es la clase *hija* o *subclase* de A. Si una clase tiene un solo padre diremos que la relación es de *herencia simple*. Si una clase tiene más de un padre diremos que tiene *herencia múltiple*.

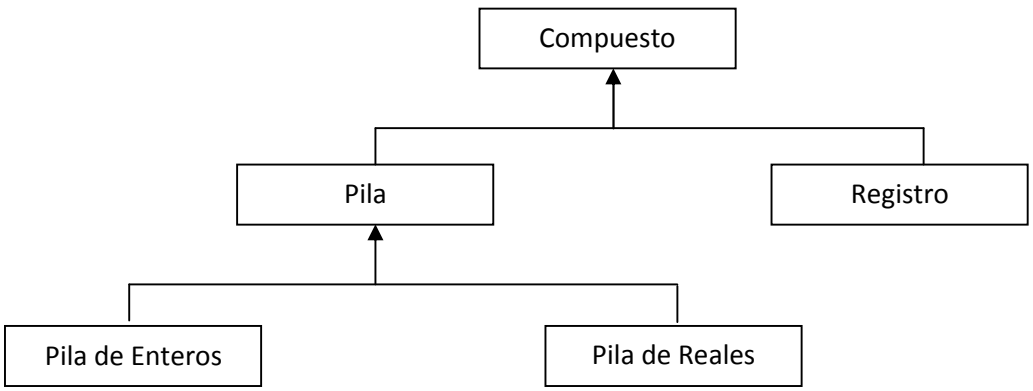
El concepto de *encapsulamiento* suele verse como un mecanismo para *dividir y conquistar* el control del programa. El programador sólo tiene acceso a aquellos datos que son parte de la especificación del segmento de programa a ser desarrollado. La especificación de cualquier otra parte del programa está más allá del dominio de conocimiento del programador.

La *abstracción* relacionada con la herencia, puede verse como más que una pared que evita que el programador vea los contenidos de objetos de datos impropios.

La *herencia* provee el mecanismo para pasar información entre objetos de clases relacionadas. Hay cuatro relaciones que resumen el uso de la herencia en los lenguajes:

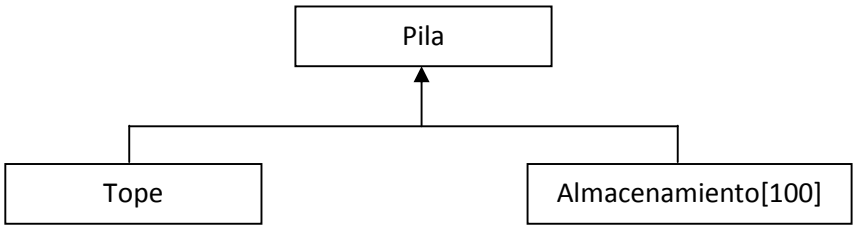
- *ESPECIALIZACIÓN* (*is-a*): Es la forma más común de herencia, la cual permite a la subclase obtener propiedades más precisas. El concepto opuesto es el de *generalización*.

  Ej: Una Pila es más precisa que un dato Compuesto



- *DESCOMPOSICIÓN*: Sigue el principio de separar una abstracción en sus componentes. El concepto opuesto es el de *agregación*

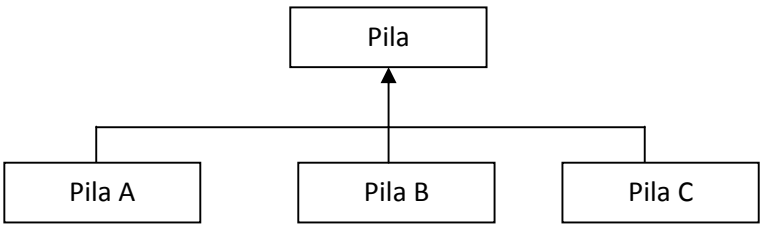
  Ej: Una Pila consiste internamente de una variable tope, y un arreglo almacenamiento[100]



  Ada → Este es el mecanismo de encapsulamiento de lenguajes como Ada sin los métodos de herencia.

- *INSTANCIACIÓN*: Es el proceso de crear instancias de una clase. Es esencialmente una operación de copia. El concepto inverso es la *clasificación*.

  Ej: Pila A, B, C

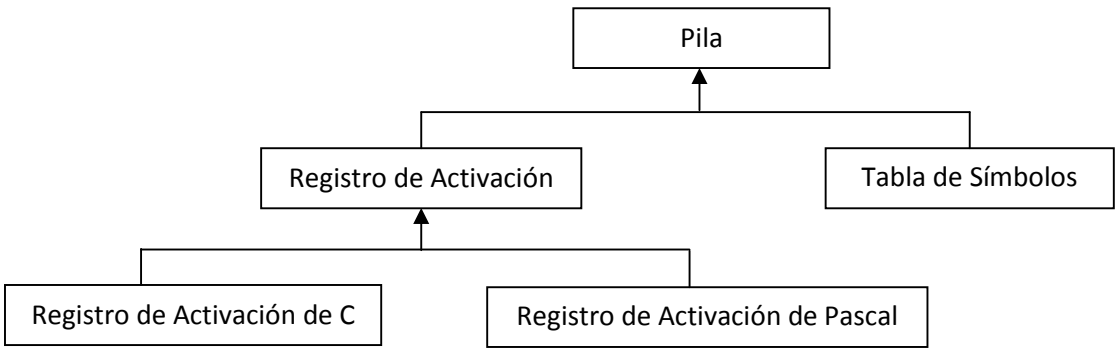


  Esto causa que el ambiente de ejecución haga tres copias del objeto de tipo Pila y los nombre A, B y C (Inversamente, podemos clasificar A,B y C como instancias de la clase Pila)

  C++ → La declaración de instancias de un objeto clase en un programa escrito en C++ es un ejemplo de instanciación. Es el proceso de declarar un objeto X de tipo Y en un programa.

- *INDIVIDUALIZACIÓN*: Agrupa objetos para un propósito común. El concepto inverso es *agrupamiento*.

  Ej: Entre las Pilas de un traductor puede haber registros de activación y tablas de símbolos. Los registros de activación en C y Pascal se implementan como pilas, pero son pilas con características diferentes.



CAPÍTULO III – Sintaxis y Semántica

3.1 INTRODUCCIÓN

Un lenguaje de programación es una notación formal que describe un modelo de la resolución de un problema. La *especificación precisa de un LP* es esencial para describir el comportamiento computacional del mismo. Sin una notación clara del efecto de los *constructores* del lenguaje es imposible determinar dicho comportamiento.

En general, la especificación formal de los lenguajes es útil a varios propósitos:

- Ayuda a la comprensión del lenguaje
- Brinda soporte para la estandarización de lenguajes
- Guía en el diseño de lenguajes
- Ayuda al sistema de escritura del compilador y lenguaje
- Soporta la verificación de correctitud de los programas
- Modela la especificación de software

La descripción de cualquier lenguaje de programación incluye tres aspectos:

- ***SINTAXIS*** *(forma)*: Determina la forma de las expresiones, sentencias y unidades del programa.

- ***SEMÁNTICA*** *(significado)*: Define el significado gramatical correcto de las expresiones, sentencias y unidades. Es más compleja y difícil de expresar, porque involucra definir qué ocurre durante la ejecución de un elemento del lenguaje o programa.

- *PRAGMÁTICA (uso)*: Define el uso correcto de los elementos del lenguaje.
  El traductor necesita proveer opciones al usuario para debugear, para interactuar con el SO, y tal vez para interactuar con un entorno de desarrollo de SW. Estas opciones conforman el *pragmatismo* del traductor (en general, estas facilidades forman parte de la definición del lenguaje).

En general, los lenguajes pueden ser formalmente definidos mediante:

- *Reconocedores de Lenguajes:* Un reconocedor lee una cadena de entrada del lenguaje y decide si la cadena pertenece o no al lenguaje (la acepta o la rechaza).

- *Generadores de Lenguajes:* El generador cuenta con un dispositivo que genera sentencias del lenguaje. Puede ser visto como un descriptor del lenguaje.

Con un generador es posible determinar si la sintaxis de una sentencia particular es correcta, comparándola con la estructura del generador. Con el reconocedor es necesario adivinar la sintaxis mediante prueba y error.

Tres mecanismos describen el diseño e implementación de los LP:

- *Expresiones Regulares*: Permiten definir los *lexemas* o *tokens* (trabajo del analizador léxico).
- *Gramáticas Formales*: Definen la sintaxis
- *Gramáticas de Atributos*: Definen la *semántica estática*

3.3 MÉTODOS FORMALES PARA DESCRIBIR LA SINTAXIS

La *sintaxis* de un LP es el conjunto de reglas que describen la estructura de los programas válidos en el lenguaje. Afecta a la legibilidad, pero no a la expresabilidad del lenguaje.

Los métodos para describir la sintaxis son:
- *Gramáticas Libres de Contexto* – BNF – Diagramas sintácticos
- *BNF Extendida* (mejora la facilidad de lectura y escritura de las BNF tradicionales)
- *Gramáticas y Reconocedores*

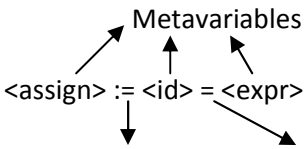
3.3.1 Gramáticas Libres de Contexto y BNF

Entre mediados y fines de 1950, Noam Chomsky y John Backus desarrollaron el mismo formalismo de descripción de sintaxis, que subsecuentemente se convirtió el método más utilizado para la sintaxis de lenguajes de programación.

Gramáticas Libres de Contexto: Los *tokens* de los lenguajes de programación pueden describirse mediante gramáticas regulares. Las gramáticas libres de contexto sirven para describir LP en forma completa.

Backus-Naur Form: BNF es una notación natural para describir la sintaxis de un LP, y es muy similar a la de las gramáticas libres de contexto. Un *metalenguaje* es un lenguaje utilizado para describir otros lenguajes. BNF es un metalenguaje para lenguajes de programación, que usa abstracciones para las estructuras sintácticas.
Las abstracciones en una descripción BNF son llamadas *símbolos no terminales*, y los lexemas y tokens de las reglas son llamados *símbolos terminales.* Una descripción BNF o gramática es simplementa una colección de reglas.

Ej – La regla de producción que describe una sentencia de asignación en un LP reducido:



Metasímbolo

Lexema: Está dado por una especificación léxica separada de la descripción sintáctica del LP (palabras reservadas, literales, etc)

<id> := A | B | C
<expr> := <id> + <expr> | <id> * <expr> | (<expr>) | <id>

3.3.2 BNF Extendida

Las extensiones a la BNF (llamadas Extended BNF o EBNF) no incrementan el poder descriptivo de la BNF, solamente mejoran la facilidad de lectura y escritura.
Tres extensiones son comúnmente incluidas en las distintas versiones de EBNF:
- [] (corchetes): representa un elemento opcional en la RHS.
- {} (llaves): son utilizadas para indicar cero o más instancias de los elementos en la RHS. Esta extensión permite construir listas con una sola regla, en lugar de utilizar recursión y 2 reglas.
- | y () (operador *or* y paréntesis): es utilizada para indicar múltiples opciones. Cuando un elemento debe ser elegido entre un grupo, las opciones son ubicadas entre paréntesis, y separadas por el operador *or* "|".

RHS: Right-Hand Side (parte derecha de la regla).

Derivación

- Las cadenas de lenguajes se generan a través de una *derivación*, partiendo del símbolo inicial.
- Cada cadena de símbolos en la derivación es una *forma sentencial*.
- Una *sentencia* es una forma sentencial que sólo tienen símbolos terminales.
- Una *derivación* es una secuencia de aplicaciones de reglas. En cada aplicación se obtiene una *forma sentencial*, y la derivación termina cuando se obtiene una *forma sentencial* sin metavariables.
- Una *derivación izquierda* es aquella en la cual se expande el no terminal más a la izquierda de cada forma sentencial.

Árbol de Derivación ≡ *Árbol Sintáctico*: Los nodos internos son metavariables o símbolos no terminales. Las hojas son lexemas o símbolos terminales.

Hay que tener presentes los aspectos que puedan surgir con respecto a la *ambigüedad* (para una misma expresión existe más de un árbol de derivación). Si la ambigüedad no se resuelve, se tienen varias interpretaciones para la misma sentencia.

3.3.3 Gramáticas y Reconocedores

Existe una relación cercana entre mecanismos generadores y reconocedores para un determinado lenguaje. Dada una gramática libre de contexto, puede construirse en forma algorítmica un reconocedor para el lenguaje generado por la gramática. Una gran cantidad de sistemas de SW ha sido desarrollada para realizar esta construcción. Tales sistemas permiten la creación rápida del analizador sintáctico del compilador para un lenguaje.

Un *compilador* es un reconocedor que recibe una cadena, la analiza sintácticamente, y si es válida genera código equivalente. Un *analizador sintáctico* asocia un subprograma a cada símbolo no terminal o metavariable. Cada subprograma recibe una cadena y construye el árbol de derivación cuya raíz sea la metavariable.

3.4 GRAMÁTICAS DE ATRIBUTOS

Una *gramática de atributos* es un mecanismo utilizado para describir más de la estructura de un LP de lo que se puede describir con una gramática libre de contexto. Es una extensión de las gramáticas libre de contexto, y permite la descripción de ciertas reglas del lenguaje, tal como la compatibilidad de tipos.

Semántica Estática: Considera los aspectos de la estructura de los LP que no pueden ser capturados por la sintaxis (BNF), debido a dificultad o imposibilidad de la gramática, pero que pueden ser evaluados en compilación.

Ej: La regla que establece que todas las variables deben estar declaradas antes de ser referenciadas no puede ser especificada en la BNF.

3.4.2 Conceptos Básicos

Una *gramática de atributos* es una gramática libre de contexto a la cual se le han agregado atributos, funciones para la evaluación de los atributos, y funciones predicado:

- *Atributos:* Están asociados a los símbolos de la gramática, y son similares a las variables (en el sentido de que pueden tener valores asociados). Describen propiedades semánticas de cada no terminal asociado a un constructor del LP.

  Atributos Sintetizados: Son usados para pasar información semántica hacia arriba en el árbol sintáctico. Su valor es computado a partir de los valores de los atributos ligados a los descendientes del nodo.

  Atributos Heredados: Pasan información semántica hacia abajo en el árbol sintáctico. Su valor se computa a partir de los valores de los atributos ligados a los ancestros.

  Atributos Intrínsecos: Son atributos sintetizados de nodos hoja cuyos valores son determinados fuera del árbol de derivación.

- *Funciones para la Evaluación de Atributos (Funciones Semánticas):* Están asociadas a las reglas de la gramática. Son utilizadas para especificar cómo los valores de los atributos son computados. Permiten capturar los valores de los atributos.

- *Funciones Predicado:* Determinan parte de sintaxis y de las reglas semánticas estáticas del lenguaje, y están asociadas a las reglas de la gramática.

3.4.3 Definición de Gramáticas de Atributos

Una *gramática de atributos* es una gramática libre de contexto G=(S,N,T,P) con los siguientes agregados:

- Asociado a cada símbolo de la gramática X, hay un conjunto de *atributos* A(X). El conjunto A(X) consiste de dos conjuntos disjuntos S(X) y H(X), llamados *atributos sintetizados y heredados* respectivamente. Además, hay *atributos intrínsecos* en las hojas del árbol de derivación.

- Se asocia a cada regla de la gramática un conjunto de *funciones semánticas* que definen ciertos atributos de los no terminales en la regla.

- Se asocia a cada regla de la gramática un conjunto (posiblemente vacío) de *funciones predicado* para chequear la consistencia de los atributos.

Dados los valores de los atributos intrínsecos en el árbol de derivación, las funciones semánticas pueden utilizarse para computar los valores de los atributos restantes.

Una función predicado tiene la forma de una expresión booleana sobre un conjunto de atributos $\{A(X_0),\ldots, A(X_n)\}$. Las únicas derivaciones permitidas con una gramática de atributos son aquellas en las que cada predicado asociado con cada símbolo no terminal es verdadero. Una función predicado con valor falso indica una violación de las reglas sintácticas o semánticas del lenguaje.

3.5 SEMÁNTICA DINÁMICA

La *semántica dinámica* trabaja con el significado de los constructores del lenguaje. No existe una noción universalmente aceptada de cómo expresar este tipo de semántica, aunque se han desarrollado varios métodos.

Los programadores necesitan conocer precisamente lo que hacen las sentencias del lenguaje. Pero generalmente lo infieren de las explicaciones que aparecen en los manuales de los lenguajes, las cuales son frecuentemente imprecisas e incompletas. Los creadores de compiladores determinan la semántica del lenguaje para la que escriben el compilador a partir de estas especificaciones en lenguaje natural, dada la complejidad de las descripciones semánticas formales.

Se han desarrollado varios sistemas notacionales para la definición formal de la semántica, los cuales están incrementalmente en uso:

- *A través del Manual de Referencia del Lenguaje:* Este es el método más común, pero adolece de falta de precisión asociada a los defectos del uso de lenguaje natural. Terminan resultando inadecuadas por estar basadas en técnicas de implementación e intuición.

- *A través del Traductor:* Es el método más común para cuestionar el comportamiento de un lenguaje en forma interactiva. En esta aproximación, el programa es ejecutado para determinar su comportamiento. Su principal desventaja radica en que el traductor depende de la máquina, y no puede ser portable a todas.

- *A través de una Definición Formal:* Es el método más común para cuestionar el comportamiento del programa a través de métodos matemáticos, que son precisos pero también complejos y abstractos. Su ventaja es que son tan precisos que pueden probar su validez matemática, y la traducción puede ser verificada de manera de asegurar el comportamiento exacto de la definición.

Los *métodos formales* resultan importantes por varias razones:
- Proveen una manera no ambigua de definir el lenguaje.
- Proveen estándares de manera que el lenguaje no sufra variaciones de implementación en implementación.
- Proveen las bases para pruebas de correctitud tanto de los compiladores como de los programas.

Los siguientes métodos son apropiados para la definición de la semántica de lenguajes imperativos:

Métodos de Prueba

- *Semántica Operacional*: Describe el significado de un programa ejecutando sus sentencias en una máquina (simulada o real). De esta manera el cambio de la máquina (en la memoria, en los registros, etc.) define el significado de la sentencia.

- *Semántica Axiomática*: Está basada en la lógica formal. Los axiomas o reglas de inferencia están definidos para cada tipo de sentencia en el lenguaje, transformando expresiones en otras expresiones (llamadas aserciones). Originalmente fue pensada para la verificación formal de programas.

- *Semántica Denotacional*: Está basada en la teoría de funciones recursivas. Este método es el más abstracto de los métodos para describir semántica → Modelo Teórico

3.5.1 Semántica Operacional

La *semántica operacional* define el comportamiento de un programa ejecutando sus acciones sobre una máquina abstracta o hipotética. Requiere que las operaciones de la máquina utilizada para la definición semántica estén definidas en forma precisa.

Establece la forma de expresar semántica a más bajo nivel, ya que la misma está definida por la arquitectura de la máquina. Sin embargo esta aproximación tiene aspectos que no la hacen totalmente apropiada para la especificación semántica:

- Las acciones pueden ser muy difíciles de entender por complejidades del HW o del SO.
- La especificación semántica dada, sólo sirve para sistemas de cómputo idénticos.
- Para lenguajes complejos, es difícil escribir intérpretes correctos.

Algunos problemas pueden evitarse sustituyendo la máquina abstracta por simulaciones sobre una computadora real (simulación sobre una máquina virtual de más bajo nivel). Se simula con una cantidad discreta de estados y secuencias explícitas de operaciones de cómputo.
El proceso consiste en construir un *traductor* (traduce código fuente al código máquina de la computadora ideal), y un *simulador* para la computadora ideal.

Una descripción operacional concreta usa un autómata intérprete que define la semántica de cada constructor. El intérprete actúa como una computadora virtual: cada constructor del lenguaje se traduce a una secuencia de instrucciones que van a ser ejecutadas por una computadora virtual.
La *especificación semántica operacional de un LP* L a partir de un autómata concreto requiere la definición de:

- Un LP de bajo nivel M que brinde un repertorio simple y reducido
- Un traductor que convierta una instrucción en el lenguaje L al lenguaje de bajo nivel M
- Una máquina virtual que ejecute el LP M

El cambio de estado en la máquina virtual causado por la ejecución del código traducido de una sentencia en el lenguaje L, define el significado de la sentencia → Este método describe el significado de las sentencias de un lenguaje de alto nivel en términos de sentencias en un lenguaje de bajo nivel más simple.

[Otro Libro]
→ El *significado de un constructor es especificado por* la computación que induce cuando es ejecutado en una máquina (*los cambios que provoca*). En particular, es de interés *cómo* se produce el efecto de la computación.

En una explicación operacional del significado de un constructor, se indica *cómo* se ejecuta el constructor:

- Para ejecutar una secuencia de sentencias separadas por ";", se ejecuta cada sentencia individualmente una después de la otra, y de izquierda a derecha (*Secuencia*).
- Para ejecutar una sentencia que consiste de una variable seguida de "=" y otra variable, se determina el valor de la segunda variable y luego se lo asigna a la primera (*Asignación*).

De esta forma, se tiene una *abstracción* acerca de *cómo* el programa es ejecutado en una máquina. Es importante observar que de hecho es una abstracción, ya que se ignoran detalles como el uso de registros y las direcciones de las variables. Por esto, la semántica operacional es independiente de la arquitectura de la máquina y de las estrategias de implementación.

Evaluación:

- Buena si es usada en forma informal (descripciones simples e informales).
- Dado que se expresan las sentencias de un LP de alto nivel en términos de otro de bajo nivel, pueden darse circularidades (conceptos definidos en términos de sí mismos).
- Depende de LP de más bajo nivel, no de la matemática.

3.5.2 Semántica Axiomática

En *semántica axiomática* se utilizan elementos de la lógica matemática para especificar la semántica de un lenguaje de programación y sus componentes.

Con esta semántica se describe el significado de cada programa sintácticamente correcto asociando a cada constructor las propiedades que tienen las variables antes de que la ejecución comience, y luego de que el programa o el constructor terminan. Básicamente, se utilizan *fórmulas de correctitud*, es decir, *predicados o aserciones* (precondiciones y post-condiciones de las sentencias).
El cálculo de predicados es utilizado como metalenguaje de la semántica axiomática, a fin de expresar las suposiciones iniciales y los resultados esperados de la ejecución→ Fórmula pre-post condición: {P} sentencia {Q}

La idea es determinar la post-condición del programa, es decir, expresar cuál es el resultado esperado. Luego, ir retrocediendo el programa hasta alcanzar la primera sentencia. Si la precondición de la primera sentencia es la misma que la de la especificación del programa, entonces el programa es correcto.

La *precondición más débil* (*weakest precondition*) es la precondición menos restrictiva que garantiza la validez de la post-condición asociada. Si la weakest precondition puede ser calculada a partir de la post-condición, para cada sentencia del lenguaje, entonces pueden construirse pruebas de correctitud para los programas escritos en ese lenguaje.

{ Axiomas }

<u>*Asignación*</u> → Sea x=E una sentencia de asignación y Q su post-condición, su precondición P se define por el axioma:

$P = Q_{x \rightarrow E}$ P se computa como Q con todas las instancias de x reemplazadas por E.

El axioma de asignación se garantiza verdadero sólo si la asignación no tiene efectos colaterales.

<u>*Regla de Consecuencia*</u> → La forma general es $\frac{S_1, S_2, \ldots, S_n}{S}$

Significa que si S_1, S_2,.., S_n son verdaderas, entonces se infiere la veracidad de S. La forma de la regla de consecuencia es:

$$\frac{\{P\}\ S\ \{Q\},\ P \rightarrow P',\ Q \rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

<u>*Secuencia*</u> → Si {P1} S1 {P2} y {P2} S2 {P3}, la regla de inferencia para tales dos sentencias es:

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

<u>*Selección*</u> →

$$\frac{\{B \text{ and } P\}\ S1\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{Q\}}$$

Ciclos tipo While: Este axioma es inherentemente más complejo debido a que el número de iteraciones no puede ser determinado en todos los casos. Cuando la cantidad de iteraciones es conocida, el ciclo puede tratarse como una secuencia.

El problema de computar la precondición más débil de un ciclo implica inducción. Para ello debemos plantear una hipótesis, la cual corresponde al *invariante de ciclo*. Su determinación es crucial para encontrar la precondición más débil.

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\} \text{ while } B \text{ do } S \text{ end } \{I \text{ and (not } B)\}}$$

Para encontrar el *invariante* de ciclo se puede utilizar un método similar al usado para determinar la hipótesis inductiva en una inducción matemática. El invariante debe verificar:

1. La precondición del ciclo debe garantizar la veracidad del invariante
2. El invariante debe garantizar la veracidad de la post-condición cuando termina el ciclo
3. Durante la ejecución del ciclo, el valor de verdad del invariante no debe verse afectado. Es decir, la evaluación del control del ciclo no debe cambiar su valor de verdad.

Otro aspecto complicado es la terminación, si se garantiza Q como post-condición del ciclo, entonces la precondición P del mismo debe garantizar Q al terminar el ciclo, como así también su terminación.

Si I es el invariante de ciclo, se requieren los siguientes *axiomas* para el ciclo:

1. $P \rightarrow I$
2. $\{I\}\ B\ \{I\}$
3. $\{I \text{ and } B\}\ S\ \{I\}$
4. $\{I \text{ and (not } B)\} \rightarrow Q$
5. El ciclo termina

[Resumen]
La *semántica axiomática* asocia a cada constructor del lenguaje un axioma o regla de inferencia. Cada programa puede pensarse como una máquina de estados. Cada axioma o RI especifica el cambio de estado que provoca la ejecución de una instancia específica del constructor.

Cada instrucción específica puede asociarse a 2 aserciones que describen el cambio de estado que provoca la ejecución de la instrucción: precondición y post-condición. Una aserción precediendo a una instrucción (precondición) describe las restricciones de las variables antes de la ejecución de la instrucción. Una aserción a continuación de una instrucción (post-condición) describe las restricciones de las variables después de la ejecución de la instrucción.

Cada instrucción sintácticamente válida, junto con los predicados que describen el estado anterior y posterior conforman una fórmula {P} S {Q}.

[Otro Libro]
→ La semántica axiomática fue *definida para probar la correctitud de programas*. Las propiedades específicas del efecto de ejecutar constructores son expresadas como *aserciones*.

Muchas veces son de interés las *propiedades de correctitud parcial* de programas. Un programa es parcialmente correcto con respecto a una precondición y una post-condición, si cuando el estado inicial garantizar la precondición y el programa termina, entonces está garantizado que el estado final satisface la post-condición.

La semántica axiomática provee un *sistema lógico* para probar las propiedades de correctitud parcial de programas individuales.

Evaluación:
- Definir axiomas y reglas de inferencia para todas las sentencias del lenguaje no es tarea sencilla.
- Es una excelente herramienta para determinar la correctitud de programas, y provee un excelente framework para razonar sobre los programas durante su construcción y posteriormente.
- No es muy útil para los programadores y para los diseñadores de compiladores.
- Su utilidad para describir el significado de los programas queda limitada por los programadores o quienes escriben los compiladores.
- Los sistemas lógicos proveen una forma sencilla de probar propiedades de los programas.

3.5.3 Semántica Denotacional

La *semántica denotacional* es el método más riguroso conocido para describir el significado de los programas. Está basado en la teoría de funciones recursivas. En esta aproximación, los programas pueden ser traducidos a funciones, y sus propiedades probadas utilizando la teoría de funciones matemática estándar, conocida como *cálculo funcional.*

Define el comportamiento de un lenguaje de programación aplicando funciones matemáticas a programas, para representar su significado. Se define para cada entidad del lenguaje un objeto y una función matemática que mapea instancias de la entidad en instancias del objeto matemático.

El estado de un programa está dado por el valor de las variables corrientes: S= [<i1,v1>, <i2,v2>, … , <in,vn>]. Sea VARMAP la función que recibe el nombre de una variable y un estado y retorna el valor corriente de esa variable: VARMAP(ij,S) = vj

Números Decimales: <num_decimal> → 0|1|2|3|4|5|6|7|8|9|
<num_decimal> (0|1|2|3|4|5|6|7|8|9)

El dominio de los valores de los objetos semánticos es el conjunto de enteros. La función semántica, que llamaremos M_{dec}, mapea los objetos sintácticos a objetos enteros:

M_{dec}('0') = 0 M_{dec}('1') = 1 …. M_{dec}('9') = 9

M_{dec}(<num_decimal>'0') = 10* M_{dec}(<num_decimal>)
M_{dec}(<num_decimal>'1') = 10* M_{dec}(<num_decimal>) +1
…
M_{dec}(<num_decimal>'9') = 10* M_{dec}(<num_decimal>)+9

Expresiones: Es necesario asumir que las expresiones no generan efectos colaterales. El único error que se considera en las expresiones, es que la variable tenga un valor indefinido. El conjunto de todos los valores que una expresión puede evaluar es $Z \cup \{error\}$

Asignación: Mapea conjuntos de estados a conjuntos de estados

Ciclos While: Mapea conjuntos de estados a conjuntos de estados. El significado del ciclo es el valor de las variables del programa luego de que las sentencias del ciclo han sido ejecutadas el número previsto de veces, asumiendo que no hubo errores.

En esencia, el ciclo ha sido convertido de iterativo a recursivo, donde el control recursivo es matemático, definido por otras funciones de mapeo recursivo de estados. Esto es bueno porque la recursión puede ser descripta más fácilmente con rigor matemático que la iteración.

[Otro Libro]
→ El *significado* es modelado por funciones matemáticas que representan el *efecto de ejecutar los constructores*. El *efecto* de los constructores es lo único que interesa, no cómo fue obtenido, y esto es modelado por funciones matemáticas:

- El efecto de una secuencia de sentencias separadas por ";" es la composición funcional de los efectos de las sentencias individuales (Secuencia).
- El efecto de una sentencia que consiste de una variable seguida de "=" y otra variable, es la función que dado un estado retorna un nuevo estado que es como el estado original, salvo que el valor de la primera variable de la sentencia es igual al de la segunda variable (Asignación).

Evaluación:

- Puede ser utilizada para probar la correctitud de programas.
- Se puede predecir el comportamiento de un programa sin necesidad de correrlo en una computadora.
- Provee una forma rigurosa de pensar sobre programas.
- Puede ser una ayuda en el diseño de lenguajes (aunque para la implementación es preferible la semántica operacional).
- Ha sido utilizada en sistemas de generación de compiladores.
- Debido a su complejidad, son de poco uso por parte de los usuarios de los lenguajes de programación.

Implementación de LP Imperativos

INTRODUCCIÓN

Al momento de desarrollar un lenguaje de programación, la arquitectura influencia el diseño del lenguaje de dos maneras:

- La computadora subyacente sobre la cual los programas escritos se ejecutarán
- El modelo de ejecución, o computadora virtual, que soporta a dicho lenguaje sobre el HW actual.

La *computadora* consiste básicamente de seis *componentes*, los cuales tienen una correspondencia con los aspectos de los lenguajes de programación:

- Datos
- Operaciones Primitivas
- Control de Secuencia
- Acceso a los Datos
- Manipulación de Almacenamiento
- ~~Operaciones de Entorno~~

Un lenguaje es una notación formal que permite escribir especificaciones ejecutables. Según la característica del lenguaje, hay distintos conceptos que son de especial importancia:

Lenguajes de Alto Nivel: Expresividad, Confiabilidad, Portabilidad
Lenguajes de Bajo Nivel: Eficiencia, Costo de realización, Flexibilidad

Los *traductores* que transforman un lenguaje de alto nivel en un lenguaje comprensible por la computadora son:

- *Compiladores*
- *Intérpretes*
- *Híbridos*

De acuerdo a la etapa de desarrollo del lenguaje (diseño o implementación) son las características del lenguaje que se definen:

Diseñador:
- Estructura del programa
- Reglas de alcance

Implementador:
- Tiempos de Ligadura
- Intérprete o compilador

Las decisiones de implementación tomadas no deberían afectar la semántica.

La *ligadura* entre una variable y su tipo, y entre los procedimientos y su código *influye en la decisión* acerca de utilizar *compiladores o intérpretes*:

- Cuando la mayor parte de las *ligaduras* son *estáticas*, es más apropiado utilizar un *compilador.*
- Cuando la mayor parte de las *ligaduras* son *dinámicas,* es más apropiado utilizar un *intérprete*.

[MÉTODOS DE IMPLEMENTACIÓN – 1.7 Sebesta]

1.7.1 Compilación

El *compilador* traduce los programas a un lenguaje máquina que puede ser ejecutado directamente en la computadora. Genera un programa objeto a partir de un programa fuente.
+ Ejecución rápida de programas

Proceso de Compilación:

1. Análisis léxico
2. Análisis sintáctico y generación de la tabla de símbolos
3. Análisis semántico y generación de código intermedio
4. Optimización
5. Generación de código máquina
6. Vinculación

1.7.2 Interpretación Pura

El *intérprete* chequea y ejecuta cada instrucción en el momento que es alcanzada. En general, para cada constructor el lenguaje brinda un subprograma que se invoca cuando el constructor es alcanzado. Es muy parecido a la máquina convencional que efectúa: fetch – decode – execute.
+ Implementación fácil de operaciones de debug a nivel fuente (los errores en tiempo de ejecución pueden referirse al código fuente de las unidades – línea de código donde está el error).
- Ejecución mucho más lenta que con compilación, dado que se decodifican instrucciones de alto nivel, en lugar de instrucciones en lenguaje máquina (ejecución de 10 a 100 veces más lenta).
- Requiere más espacio, ya que además del código fuente del programa debe almacenarse la tabla de símbolos durante la interpretación.

Proceso de Interpretación:

1. Obtener la próxima instrucción
2. Analizar la sintaxis y la semántica
3. Determinar las acciones a ser ejecutadas (seleccionar el subprograma correspondiente al tipo de instrucción)
4. Ejecutar las acciones (ejecutar el subprograma o acciones correspondientes al subprograma)

1.7.3 Sistemas de Implementación Híbridos

El traductor *híbrido* traduce lenguajes de programación de alto nivel a un lenguaje intermedio, el cual puede ser interpretado fácilmente. En lugar de traducir el código en lenguaje intermedio a lenguaje máquina, simplemente interpreta el código intermedio.

Pasos del Traductor Híbrido:

1. Analizador léxico
2. Análisis sintáctico y generación de la tabla de símbolos
3. Análisis semántico y generación de código intermedio

→ Obtener la próxima instrucción en código intermedio. Traducirla y ejecutarla

Características de Diseño del LP que afectan a la Compilación:

- Unidades sintácticas.
- El diseño del lenguaje especifica restricciones que llevan a que la compilación sea separada o independiente
- Las características del lenguaje también afectan los tiempos en que se producen los chequeos entre los módulos.
- Posibilidad de invocar a subprogramas de otros lenguajes

VINCULACIÓN Y CARGA

El *vinculador* (*linker*) es un programa asociado al compilador que toma unidades objeto (en ~~lenguaje ensamblador~~ o lenguaje máquina) y genera código reubicable → Recolecta programas del sistema y los vincula a los programas de usuario. Además de los programas de sistema, los programas de usuario frecuentemente deben ser linkeados con otros programas de usuario compilados previamente (librerías).
La ejecución va precedida de una etapa de *carga* (se cambian las direcciones relativas por direcciones absolutas), y a veces se realizan las dos cosas juntas (vinculación y carga).

Puede darse el caso de que se haga compilación y vinculación, y la traducción no sea al lenguaje máquina, sino a otro lenguaje intermedio (*compilación cruzada*).

TABLA DE SÍMBOLOS

Durante la compilación se genera la *tabla de símbolos* que tiene operaciones de actualización:

- Inserción
- Eliminación
- Búsqueda → Cuando el compilador alcanza la sección ejecutable y declaración

Cuando el compilador alcanza la sección de declaraciones

El compilador busca una variable con el mismo nombre:

- Si se trata de una *declaración* y la encuentra en el *mismo nivel*, entonces sucede un error.
- Si se trata de una *declaración* y la encuentra en *otro nivel léxico*, entonces el análisis continúa sin problemas.

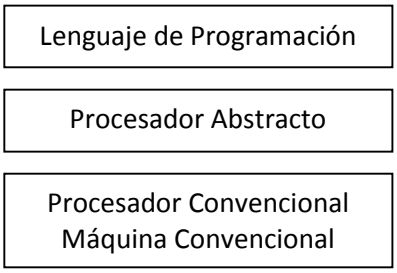
→ Para las declaraciones generalmente las búsquedas se realizan en forma local.

La *traducción* se hace cuando el compilador alcanza el código ejecutable. No se actualiza la tabla pero si hay que consultarla con referencia a una variable, ésta se reemplaza por su desplazamiento (en el código que el compilador está traduciendo).

Obs: Cuando el compilador sale del código ejecutable de P, se eliminan las variables locales de la tabla. Si en P hay una referencia a una variable k (del programa principal) no se puede reemplazar k por su desplazamiento a partir del área de P, sino de otra área.

SEMÁNTICA OPERACIONAL

Los LP de alto nivel definen una máquina virtual, es decir, no existe una máquina que ejecute las instrucciones de ese lenguaje.



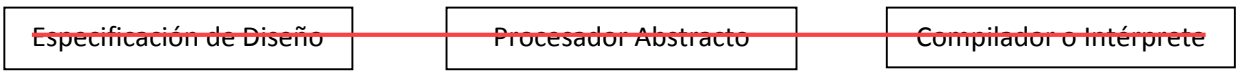
La especificación del LP está dada por la semántica operacional para cada constructor del lenguaje. El diseñador da la traducción (de cómo estará en el procesador abstracto). Si la traducción es muy detallada puede ser demasiado parecida a la de la máquina convencional. Por otra parte, si el lenguaje es muy diferente a la arquitectura específica, no conviene arquitectura específica.

Problemas de semántica por:

- Ambigüedad en la definición del diseñador
- Diferencias entre el lenguaje de programación y su traducción en términos del procesador abstracto
- Diferencias al traducir instrucciones del procesador abstracto al procesador convencional

PROCESADOR ABSTRACTO

La semántica de cada constructor del lenguaje se describe en término de instrucciones provistas por una máquina virtual.



- Cuestiones implícitas en la definición del lenguaje de programación pueden ser interpretadas por el implementador de diferentes maneras.
- Diferencias en las decisiones tomadas por el implementador para definir la máquina virtual especificada implícitamente por el lenguaje.
- Diferencias en las decisiones tomadas por el implementador para construir una implementación concreta para la máquina virtual a partir de las facilidades reales.

Componentes:

- *Memoria*: Celdas del mismo tamaño, donde cada celda tiene una dirección y un contenido.
    - *Memoria para Datos:* DATOS D PILA
    - *Memoria para Código:* CODIGO C
- *Registros Rápidos*:
    - *IP*: Registro especial que contiene la dirección de la próxima instrucción a ejecutar.
    - *R1, R2,…*
- *Repertorio de Instrucciones*: Conjunto de operaciones provistas por el procesador abstracto.

Repertorio de instrucciones: (Argumentos: origen, destino)

- LEER A	Ej: LEER D[11]	
- ESCR A	Ej: ESCR D[11]	
- ASIGN A,R	Ej: ASIGN D[11], D[8]	{ D[8] ← D[8] }
- SUMA A,B,R	Ej: SUMA D[11], D[15], D[8]	{ D[8] ← D[11] + D[15] }
- DIFE A,B,R	Ej: DIFE D[11], D[15], D[8]	{ D[8] ← D[11] - D[15] }
- MULT A,B,R	Ej: MULT D[11], D[15], D[8]	{ D[8] ← D[11] * D[15] }
- DIVI A,B,R	Ej: DIVI D[11], D[15], D[8]	{ D[8] ← D[11] / D[15] }
- IGUA A,B,R	Ej: IGUA D[11], D[15], D[8]	{ D[8] ← D[11] = D[15] }
- NOT A, R	Ej: NOT [D11], D[8]	{ D[8] ← not(D[11]) }
- AND A,B,R	Ej: AND D[11], D[15], D[8]	{ D[8] ← D[11] and D[15] }
- OR A,B,R	Ej: OR D[11], D[15], D[8]	{ D[8] ← D[11] or D[15] }
- NADA		
- PARA		

Se agregan operaciones que permiten mejorar la lógica del programa:

- SET dD,E — donde dD es una dirección de la memoria de datos y E es una expresión

- DVSF E,dC
- JUMP E,dC
- JUMP dC

} donde dC es una dirección de la memoria de código y E es una expresión

→Limitaciones:

1. Toda transferencia de control (modificación del IP) se realiza a través de la operación ASIG, sea condicional o incondicional.
2. Toda instrucción algebraica se traduce generando una secuencia de instrucciones de bajo nivel (oscurece la lógica del programa).

REPRESENTACIÓN DE DATOS EN MEMORIA

Visión convencional de la memoria: La estructura de memoria es simple, es una secuencia de bits agrupados en bytes o palabras. Cada celda tiene una dirección y un contenido.

Los LP de alto nivel brindan una visión abstracta de la memoria:

- La memoria se organiza según estructuras más complejas que las provistas por el HW
- Las celdas no son referenciadas por su dirección sino por un nombre simbólico, que muchas veces permite referenciar bloques de celdas
- Cada celda o bloque de celdas tiene un tipo asociado, que permite interpretar el contenido.

→ Para eliminar la diferencia entre las dos visiones, se utiliza el concepto de *objeto de dato*.

Un *objeto de dato* es un agrupamiento de una o más celdas que contienen un valor de dato (es un contenedor), está asociado a una dirección, y puede ser *simple o estructurado*. Todo objeto de datos tiene atributos: celdas, valor, alcance.

Puede estar asociado a una variable accesible por el programador, o estar restringido al manejo del sistema. En ejecución se mantienen objetos de datos con algunas asociaciones a las variables del programador, y otras a la representación interna.

[Pratt]
IMPLEMENTACIÓN DE TIPOS DE DATOS
La *implementación de un tipo de datos* establece:

- *La estructura de los objetos de datos*: Representación de almacenamiento que es usada para representar los objetos de datos del tipo de dato en el almacenamiento durante la ejecución del programa. La *representación interna* de los datos puede ser *simple o estructurada* (tipos de datos elementales o estructurados).

- *Las operaciones*: La forma en que las operaciones definidas para el tipo de datos son representadas en términos de procedimientos o algoritmos que manipulan la representación elegida para almacenar los objetos de datos.
  Existen tres alternativas para su *implementación*:

  - *Directamente como una operación de HW* (Ej: suma y resta de enteros, si se usa la representación provista por el HW).
  - *Como un procedimiento o función*: implementar la operación por SW.
  - *Como una secuencia de código in-line*: Una secuencia de código in-line es también una implementación por SW para la operación. Sin embargo, en lugar de utilizar un subprograma, las operaciones que hubieran estado en el subprograma son copiadas en el programa en el lugar donde el subprograma hubiera sido invocado.

Siempre que haya una operación de HW asociada a una operación provista por un tipo de dato del lenguaje, la traducción es directa. Cuando no la hay, se puede reemplazar la operación por una secuencia de instrucciones en lenguaje máquina (o llamar a un subprograma que las contenga).

REPRESENTACIÓN DE DATOS ELEMENTALES
La representación para almacenar los tipos de datos elementales está fuertemente influenciada por la computadora subyacente que ejecutará los programas. Si se utiliza la representación provista por el HW, entonces las operaciones básicas sobre el tipo de datos pueden ser implementadas utilizando las operaciones provistas por HW.
Los *atributos* de los objetos de datos elementales son *tratados de dos posibles formas*:

- Por *eficiencia,* muchos lenguajes son diseñados para que los atributos de datos sean determinados por el compilador. De esta manera los atributos no son almacenados en el almacenamiento en ejecución (Ej: C).
- Los atributos pueden ser almacenados en un *descriptor* como parte del objeto de dato en tiempo de ejecución. Este modelo se utiliza en lenguajes que persiguen más la *flexibilidad* que la eficiencia. Como la mayoría del HW no provee facilidades para representar los descriptores directamente, estos deben ser simulados por SW.

La representación de los objetos de datos en general, es independiente de su localización en memoria. La representación en el almacenamiento es generalmente descripta en términos del tamaño de bloque de memoria requerido. Usualmente la dirección de la primera palabra o byte de tal bloque de memoria es usada para representar la ubicación del objeto de dato.

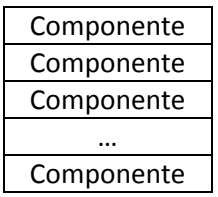
{ Las alternativas de implementación de operaciones son las descriptas para tipos de datos en general }

REPRESENTACIÓN DE DATOS ESTRUCTURADOS

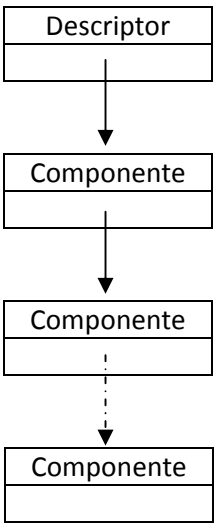
Los objetos de datos estructurados pueden tener representación de tamaño *fijo* o *variable*.
La representación del tipo de datos incluye el *dato* (almacenamiento para las componentes de la estructura) y un *descriptor* (almacena algunos o todos los atributos de la estructura), el cual tiene al menos el tamaño del objeto → Esta distinción es fundamental en datos de estructura variable.

La representación para objetos de datos estructurados puede ser:

- *Secuencial*: La estructura de datos es almacenada en un solo bloque contiguo que incluye el descriptor y los componentes → Usada para estructuras de tamaño fijo, y a veces para estructuras homogéneas de tamaño variable (Ej: strings).



- *Enlazada*: La estructura se particiona en bloques de memoria contiguos, que se encuentran enlazados de alguna manera (a través de punteros). Los bloques pueden ser del mismo tamaño o de tamaño variable. En las primeras celdas se almacena el descriptor y en la última celda del bloque la dirección del próximo bloque (enlace) → Comúnmente usada para estructuras de tamaño variable (Ej: listas).



Implementación de Operaciones (Acceso a las Componentes)
El *acceso a las componentes* es de suma importancia en la implementación de la mayoría de las ED. Debe proveerse eficientemente tanto el acceso random, como el acceso secuencial.

{Representación Secuencial}

- *Acceso Random*: La selección random se realiza a partir de una *dirección base* y un desplazamiento dentro del bloque (*offset*), utilizando una fórmula de acceso

- *Acceso Secuencial*: Para estructuras homogéneas (Ej: arreglos) la selección secuencial de componentes de la estructura es posible a través de los siguientes pasos:
    1. Seleccionar el primer componente utilizando la fórmula de acceso para selección random.
    2. Para avanzar al siguiente componente en la secuencia, añadir el tamaño del componente actual a la ubicación del componente actual. Como se trata de una estructura homogénea, todos los componentes tienen el mismo tamaño, y por lo tanto cada elemento de la secuencia puede hallarse sumando un valor constante a la ubicación del componente anterior.

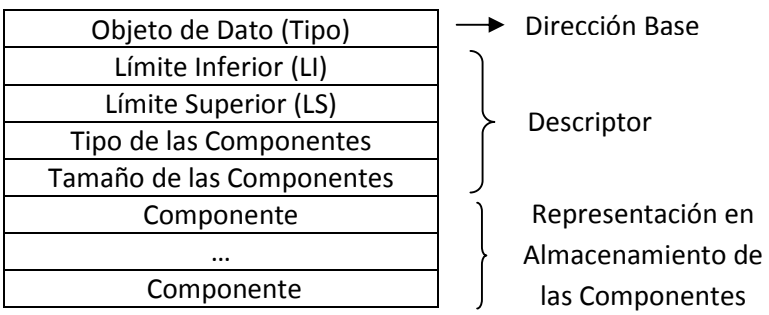
{Representación Enlazada}

- *Acceso Random*: La selección random implica recorrer los bloques desde el primer bloque de la estructura que se encuentra almacenado, hasta alcanzar la componente deseada.

- *Acceso Secuencial*: Seleccionar una secuencia de componentes implica seleccionar el primer componente mediante el método de acceso random, y luego seguir el enlace desde el componente actual hasta el siguiente componente subsecuentemente para cada selección.

ARREGLOS Y VECTORES
La homogeneidad de las componentes y el tamaño fijo del arreglo o vector hacen que la manipulación del almacenamiento y el acceso a las componentes sean directos. La homogeneidad implica que el tamaño de cada componente es el mismo, lo cual implica que el número y posición de cada componente no se modifican durante su tiempo de vida. Teniendo en cuenta esto, la *representación secuencial* es adecuada, quedando las componentes almacenadas en forma secuencial.

Es necesario almacenar un descriptor para mantener la información sobre atributos que no tienen valor hasta la ejecución (no son conocidos con anterioridad), o que son requeridos para realizar chequeos en ejecución. La representación es entonces:



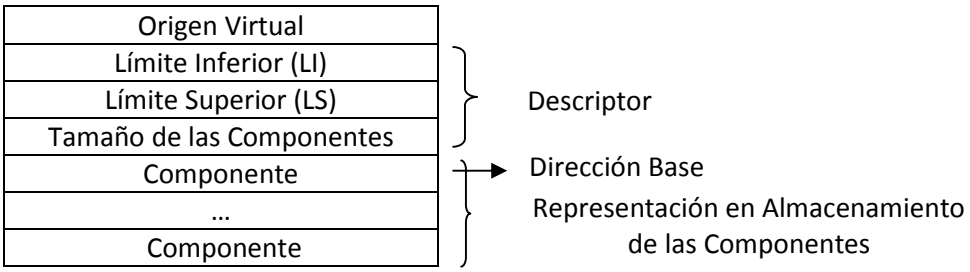
La *fórmula de acceso a los componentes* está dada por:

A[i] = D[α + tamaño del descriptor del arreglo + (i - LI) * S]

Dirección del 1° componente del arreglo

α = Dirección Base del Objeto de Datos
S = Tamaño de la Componente
LI = Límite Inferior

→ En lenguajes como C y Pascal, los tipos se omiten en el descriptor por no ser necesarios en ejecución:



La fórmula de acceso a componentes entonces queda:

A[i] = D[Origen Virtual + i * S] Origen Virtual = α + tamaño del descriptor - LI * S

→ Cuando los *arreglos de caracteres o booleanos* se empaquetan se hace más eficiente el manejo de espacio, pero la fórmula de acceso es más complicada, ya que en cada celda de memoria puede guardarse más de un elemento.

ARREGLOS MULTIDIMENSIONALES

Una matriz es convenientemente implementada como un arreglo de arreglos; un arreglo tridimensional (cubo) es un arreglo cuyos elementos son arreglos de arreglos, y así sucesivamente.

La representación en almacenamiento para un arreglo multidimensional sigue directamente de la de un arreglo. El descriptor para el arreglo multidimensional es el mismo que para el unidimensional, excepto que se necesita un límite superior e inferior para el rango permitido de cada dimensión.

El acceso a una componente de un arreglo multidimensional es similar al efectuado para arreglos de una dimensión:

$$A[i,j] = D[\alpha + \text{tamaño del descriptor del arreglo} + (i - LI_1) * S + (j - LI_2) * E]$$

ó

$$A[i,j] = D[\text{Origen Virtual} + i * S + j * E]$$

$$\text{Origen Virtual} = \alpha + \text{tamaño del descriptor del arreglo} - LI_1 * S - LI_2 * E$$

α = Dirección Base del Objeto de Datos
LI_1 y S = Límite inferior y Tamaño de la componente en la dimensión 1
LI_2 y E = Límite inferior y Tamaño de la componente en la dimensión 2

REGISTROS (Producto Cartesiano)
La representación en almacenamiento está dada por un solo bloque de memoria secuencial, en el cual las componentes se almacenan en secuencia. En general, no se requiere de un descriptor para los registros, aunque las componentes pueden llegar a requerirlo (para realizar chequeos de tipo u otros atributos).

El *acceso a componentes* puede ser implementado fácilmente porque el nombre de las componentes (nombre de los campos), su tamaño y posición dentro del bloque son conocidos durante la traducción.
La fórmula de acceso es entonces:

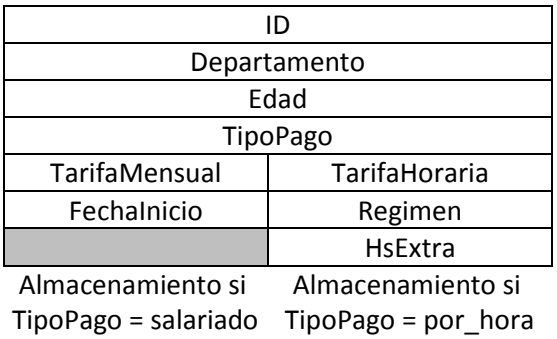
$$R.i = D[\alpha + \sum_{j=1}^{i-1} (\text{tamaño de } R.j)] \qquad \alpha = \text{Dirección Base del Objeto de Datos}$$

La sumatoria es necesaria porque los componentes pueden ser de distintos tamaños. Sin embargo, como el tamaño de los componentes es conocido al momento de la traducción, la sumatoria puede ser computada durante la traducción, y ser reemplazada por una constante que determina el *offset* del componente del registro.

La operación de *asignación* de un registro completo a otro con idéntica estructura, puede ser implementada como una copia simple del contenido del bloque de almacenamiento que representa el primer registro en el bloque que representa al segundo.

REGISTROS VARIANTES (Uniones)
La implementación de registros variantes es más simple que su uso correcto. Durante la traducción se determina la cantidad de espacio requerida para las componentes de cada variante, y se reserva (aloca) espacio en el registro para el variante más grande posible.

Ej:
```
type Pago = (salarial, por_hora);
type Empleado = record
                    ID: integer;
                    Departamentot: array[1..3] of char;
                    Edad: integer;
                    case TipoPago:Pago of
                    salarial: (TarifaMensual:real; FechaInicio:integer);
                    por_hora: (TarifaHoraria:real; Regimen:integer; HsExtra:integer)
                end;
```



Durante la ejecución no se requiere de un descriptor ya que el *tag* (tipo del variante) se considera como un campo más del registro, y los desplazamientos se calculan en la traducción.

La *selección de los campos variantes*, cuando no se realizan chequeos, se efectúa igual que para los campos ordinarios. La *asignación* a un componente que no existe, si no se realizan chequeos, cambia el contenido de la locación de memoria accedida.

Si se realiza chequeo dinámico para la *selección de campos variantes*, el cálculo de acceso (dirección base + offset) se realiza de la misma forma, pero primero debe chequearse el valor del campo correspondiente al *tag*, para asegurarse que el tag indica que el variante actual realmente existe (evitar accesos a variantes inexistentes).

REPRESENTACIÓN DE ESTRUCTURAS DE LONGITUD VARIABLE (Listas, Grafos, Pilas, etc.)
Dada la naturaleza dinámica de estas estructuras y el hecho de que sus componentes son raramente homogéneas, los métodos de almacenamiento utilizados para arreglos y vectores no resultan apropiados. En estos casos, un manejo de almacenamiento para listas enlazadas es típicamente utilizado.
Resulta indispensable decidir si se utilizará *direccionamiento absoluto* o *relativo*.

Representación de estructuras en memoria secundaria:

- *Interfaz para las operaciones*: Las operaciones en sí se implementan como llamadas a rutinas del SO.
- *Creación y mantenimiento de buffers y descriptores*.

REPRESENTACIÓN EN EJECUCIÓN DE EXPRESIONES
Debido a la dificultad de decodificar las expresiones en su forma infija tradicional, es necesario traducirlas a una forma ejecutable que pueda ser fácilmente decodificada durante la ejecución. Las alternativas más importantes en uso son:

- *Secuencia de Código Máquina (instrucciones)*: La expresión se traduce a código máquina actual, realizando los dos pasos de la traducción en un paso. El orden de las instrucciones refleja la estructura de control de la expresión original. Se utilizan almacenamientos temporales para los resultados intermedios. La representación de código máquina permite el uso de un intérprete de HW permitiendo una ejecución más rápida.

- *Estructura de Árboles*: Las expresiones pueden ser ejecutadas directamente a partir de su estructura natural de árbol utilizando un intérprete de SW (la ejecución se alcanza recorriendo el árbol).

- *Forma Prefija o Postfija*: Las expresiones en esta notación pueden ser ejecutadas por un algoritmo de interpretación simple. El código máquina es esencialmente representado en forma postfija. La representación prefija es ejecutable en muchas implementaciones.

IMPLEMENTACIÓN DE INSTRUCCIONES

Asignación
La evaluación se realiza de izquierda a derecha y se utilizan registros rápidos para los datos auxiliares. El tipo de la expresión lo determina el operando de mayor jerarquía (Ej: entre integer y real → real). El tipo de la asignación es determinado por la parte izquierda de la misma.

Una alternativa es utilizar la instrucción SET (indicar la expresión completa en el *destino*), en cuyo caso no queda especificado el orden de evaluación, ni la conversión de tipos.

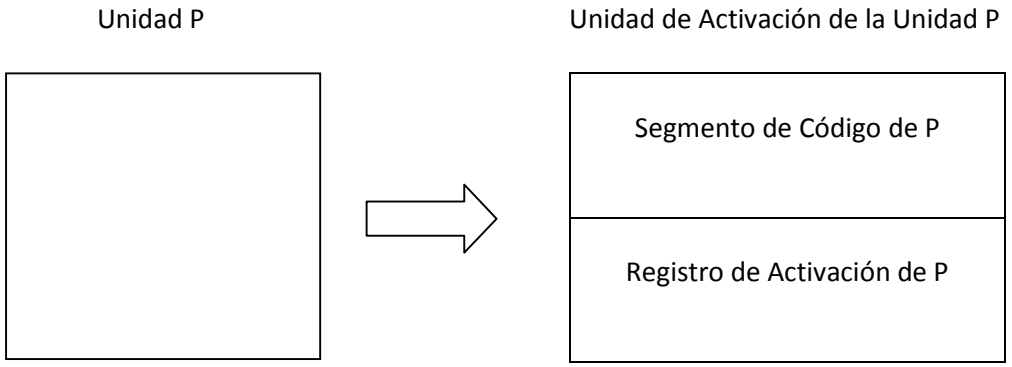
Condicional
Las sentencias *if* generalmente son implementadas usando las instrucciones de branch y jump brindadas por el HW (*goto* condicional e incondicional provisto por HW).

Las sentencias *case* son comúnmente implementadas usando una *tabla jump* para evitar el testeo repetido del valor de una variable. Una *tabla jump* es un arreglo almacenado secuencialmente en memoria, donde cada uno de sus componentes es una instrucción de jump. La expresión que forma la condición del *case* es evaluada, y el resultado es transformado en un entero que representa el offset dentro de la *tabla jump*. Cuando se ejecuta la instrucción jump en ese offset, transmite el control al bloque de código representando la alternativa elegida.

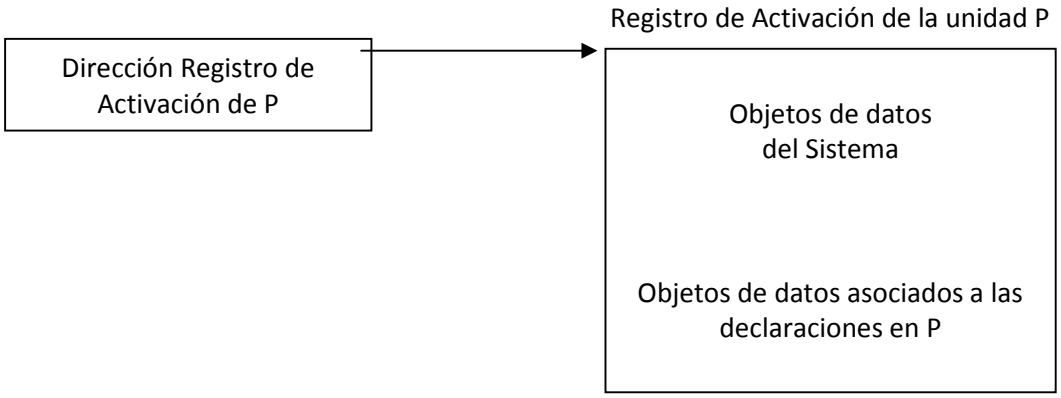
Iteración
La implementación de las sentencias de control iterativas es directa a través del uso de las instrucciones branch y jump provistas por HW.

Para implementar un ciclo *for*, las expresiones en el encabezado del ciclo que definen el valor final y el incremento deben ser evaluadas en el comienzo del ciclo (entrada inicial) y guardadas en un área especial de almacenamiento temporal. En cada iteración serán recuperadas para el testeo de la condición e incremento de la variable de control.

[Clases]
IMPLEMENTACIÓN DE UNIDADES
Si el lenguaje considera unidades, el procesador abstracto debe considerar la *unidad de activación de P* (unidad de activación de una unidad). La unidad de activación contiene toda la información relevante en ejecución de la unidad.



El *registro de activación de la unidad* (P) contiene la información sobre los objetos de datos del sistema, y los declarados en la unidad. Su nombre se debe a que los datos que describe son relevantes sólo durante la activación del subprograma, y su forma es estática.



Obs: Los objetos de datos del sistema (puntero de retorno, enlace estático, enlace dinámico) relacionan los distintos registros de activación y los segmentos de código de las distintas unidades.

El *acceso a cualquier objeto de dato declarado en P* se calcula de la siguiente forma:

Dirección Registro de Activación P + offset de la variable (determinado en la declaración)

CLASIFICACIÓN DE LP

Los lenguajes de programación pueden clasificarse según las decisiones que se toman en su diseño. Se pueden clasificar de acuerdo al *manejo de memoria* y a las *reglas de alcance*.

Clasificación de acuerdo al Esquema de Manejo de Memoria:

- Estático (se reserva memoria en compilación, no permite recursividad)
- Basado en Pila
- Dinámico

Clasificación de acuerdo a las Reglas de Alcance y al Esquema de Manejo de Memoria:

- *Alcance Estático*
  - Alocación Estática
  - Alocación Semiestática
  - Alocación Semidinámica
  - Alocación Dinámica

  } Basada en Pila
- *Alcance Dinámico*

→ Finalmente, teniendo en cuenta la resolución de invocaciones a objetos de datos: *acceso local, acceso global, esquema de alocación*.

→ En el área de datos D se almacenarán los RA de las unidades (y programa principal), y en el área de código C los segmentos de código.

LP CON ALCANCE ESTÁTICO – ALOCACIÓN ESTÁTICA

- En *compilación* se determina:

  - El alcance de cada variable y el ambiente de referenciamiento de cada unidad.
  - El espacio requerido para mantener cada variable local a la unidad, y su desplazamiento dentro del registro de activación.
  - El espacio requerido para ejecutar la unidad.
  - La asociación entre cada variable local del código y cada objeto de datos del registro de activación (RA), a través de: *dirección base del RA + offset*.
  - La asociación entre cada variable global del código y cada objeto de datos del área compartida (common): *dirección base del área compartida + offset.*

- En *vinculación* se conoce el desplazamiento de:

  - La primera instrucción de cada unidad en el área de código C, respecto a una dirección base dC.
  - Cada RA en el área de datos D, respecto a una dirección base dD.

- En la *carga* del programa:

  - Se reserva espacio de almacenamiento para todo el código y todos los registros de activación.
  - Cada segmento de código queda asociado a un único registro de activación.

Pasos a realizar en la *activación de una unidad*:
- Guardar el puntero de retorno en el primero objeto de dato del registro de activación de la unidad llamada.
- Actualizar el IP con la dirección de la primera instrucción de la unidad llamada

La traducción involucra estas dos instrucciones del procesador abstracto:

SET dU, IP+2 (se guarda en el área de datos una dirección del área de código)
JUMP dC

dU = dirección base del área de datos de la unidad llamada
cU = dirección base del segmento de código de la unidad llamada

Obs: Si se trata de unidades simples, sin parámetros, el único objeto del sistema que es imprescindible guardar es el puntero de retorno (ptr).

Pasos a realizar en el *retorno de una unidad*:
- Transferir el control recuperando el puntero de retorno almacenado en el primer objeto de dato del registro de activación de la unidad llamada

La traducción involucra la siguiente instrucción del procesador abstracto: JUMP D[dU]

LP CON ALCANCE ESTÁTICO – ALOCACIÓN BASADA EN PILA

{En general para las tres alternativas}

- En *compilación*:

  - Se determina la distancia entre la unidad que referencia a una variable y la unidad que contiene la declaración.
  - No es posible anticipar el espacio de memoria requerido para ejecutar el programa.

- En *ejecución*:

  - En el momento de activación de la unidad, se reserva espacio de almacenamiento para el RA.
  - Se conoce la dirección base de cada RA
  - Los RA ocupan y liberan memoria siguiendo un comportamiento de pila.
  - Cada segmento de código puede quedar asociado a varios RA.

Resulta necesario mantener la *cadena dinámica de llamadas* (*enlace dinámico*), para poder actualizar la dirección base del RA de la unidad activa. Es por ello que el registro de activación necesita mantener más información que el modelo anterior:

RA de la Unidad

Puntero de Retorno
Enlace Dinámico

Alocación de Memoria: La alocación de memoria sigue un comportamiento de pila. Para trabajar con la pila de datos y tener el control de los registros de activación vamos a contar con dos registros rápidos:

- *Actual*: mantiene una referencia al comienzo del registro de activación correspondiente a la unidad que se encuentra en ejecución.
- *Libre*: Mantiene una referencia al área de datos a partir de donde la misma está disponible para su uso.

Pasos a realizar en la *activación de una unidad*:
- Crear el RA
- Guardar el puntero de retorno (ptr)
- Guardar el enlace dinámico
- Actualizar Actual
- Actualizar IP

Pasos a realizar en el *retorno de una unidad*:
- Actualizar Actual utilizando el enlace dinámico
- Actualizar IP utilizando el puntero de retorno
- Destruir el RA

Este modelo admite recursividad. En caso de utilizarla, en la cadena dinámica aparece más de un registro de activación asociado a la misma unidad.

Si se resuelven los *accesos no locales* en lenguajes con estructuras de bloques anidados, se debe mantener información sobre el *enlace estático* de la unidad (dirección base del RA del bloque donde se encuentra definida). Una forma de hacerlo es manteniéndolo en el RA de la unidad:

RA de la Unidad

Puntero de Retorno
Enlace Dinámico
Enlace Estático

Por lo tanto, el acceso al entorno del bloque en que se encuentra definida la unidad es mediante D[Actual+2].

Si la distancia entre el ambiente de referenciamiento de la unidad que referencia a una variable y la unidad en la que está declarada es 1, la situación se resuelve con una indirección. Sin embargo, la cantidad de indirecciones puede ser mayor. La cantidad de indirecciones involucradas es igual a la distancia entre el uso y la declaración. A mayor cantidad de indirecciones, mayor es el costo.

→ Una alternativa para mantener la estructura lexical del programa y evitar los niveles de indirección es utilizar un *Display*, el cual es un arreglo de registros rápidos que mantiene la estructura lexical del programa. El uso del display reemplaza al enlace estático.

Todo acceso al área de datos se hace a través del display:

- Si la variable es *local*, se accede a través de DISPLAY[0]
- Si la variable es *no local*, se accede con DISPLAY[n], siendo n la distancia entre la unidad que hace referencia a la variable y la unidad donde se encuentra declarada.

Ventajas y Desventajas del uso del Display:

+ No importa la distancia, siempre se accede con un único nivel de indirección.
- La actualización del display es costosa, cada vez que se invoca o se termina la ejecución de una unidad hay que actualizarlo.
- No tiene sentido acceder a las variables locales a través de DISPLAY[0]. Puede utilizarse Actual y se elimina un nivel de indirección.
- La cantidad de registros rápidos a utilizar es el máximo nivel de anidamiento del programa. La cantidad de registros se calcula estáticamente por el compilador.

Alcance Estático – Alocación Semiestática {Basada en Pila}

- En *compilación* se determina:

  - El espacio requerido para mantener cada variable local a una unidad.
  - El tamaño del RA.
  - El desplazamiento de cada variable local dentro del RA.
  - La distancia entre la unidad que referencia a una variable y la unidad que contiene la declaración.
  - ~~La asociación entre cada variable local del código y cada objeto de dato del registro de activación a través de: *dirección base del RA + offset*.~~
  - La asociación entre cada variable global del código y cada objeto de dato en la pila. La dirección base considera la distancia entre la declaración y la referencia, utilizando la cadena estática o el display.

- En *ejecución*: (idem alocación basada en pila en general)

  - En el momento de activación de la unidad, se reserva espacio de almacenamiento para el RA.
  - Se conoce la dirección base de cada RA.
  - Los RA ocupan y liberan memoria siguiendo un comportamiento de pila.
  - Cada segmento de código puede estar asociado a más de un RA.

Pasos a realizar en la *activación de una unidad*:

a) Guardar el puntero de retorno: Pila[Libre] ← IP+T

   Siendo T la cantidad de instrucciones que se van a saltear dentro de la unidad llamadora (como mínimo hay que saltear la instrucción JUMP que realiza la transferencia de control, y si hay pasaje de parámetros hay que saltear las instrucciones vinculadas y otras)

b) Guardar el Enlace Dinámico: Pila[Libre+1] ← Actual

c) Guardar el Enlace Estático (LP con bloques anidados): Pila[Libre+2] ← *Expresión de cálculo de EE* (*)

d) Actualizar el valor del registro Actual: Actual ← Libre

e) Actualizar el valor del registro Libre: Libre ← Libre+S

   Siendo S el tamaño del registro de activación (objetos de datos del sistema, parámetros y variables locales), el cual es calculado por el compilador.

f) Actualizar el valor del IP: IP ← Dirección del segmento de código de la unidad invocada (Cc)

{Obs! Los RA se guardan en la pila, que se accede como Pila[i] }

Pasos a realizar en el *retorno de una unidad*:

a) Actualizar el valor del registro Libre: Libre ← Actual

b) Actualizar el valor del registro Actual: Actual ← Pila[Libre+1] {ED de la unidad que se ejecutó}

c) Actualizar el valor del IP: IP ← Pila[Libre] {Puntero de Retorno}

(*) *Para el cálculo del EE hay tres situaciones posibles:*

```
Program Principal
        Procedure Q
                Procedure R
                End R
        End Q
        Procedure P
        End P
End Principal
```

1. Dos unidades P y Q ambas en el mismo nivel léxico y P invoca a Q (P → Q).
2. Una unidad Q invoca a otra unidad R que está declarada dentro de ella (Q → R).
3. Una unidad R invoca a una unidad Q que la contiene (R → Q).

1. P → Q: La distancia entre la unidad llamadora y la llamada es 0 (ambas tienen el mismo nivel léxico), por lo tanto el enlace estático de Q es el de P.

   EE de Q ← Pila[Actual+2]

2. Q → R: La distancia entre la unidad llamadora y la llamada es 1, por lo tanto el enlace estático de R es la dirección del registro de activación de la unidad llamadora, es decir Q.

   EE de R ← Actual

3. R → Q: La distancia entre la unidad llamadora y la llamada es negativa (-1). En este caso debe existir una instancia de Q suspendida anteriormente, ya que no habría instancia de R sin alguna instancia de Q. Existe por lo tanto una forma de recursión cruzada involucrada.

   EE de Q ← Pila[Pila[Actual+2]+2]

   EE de R = dirección base RA de Q

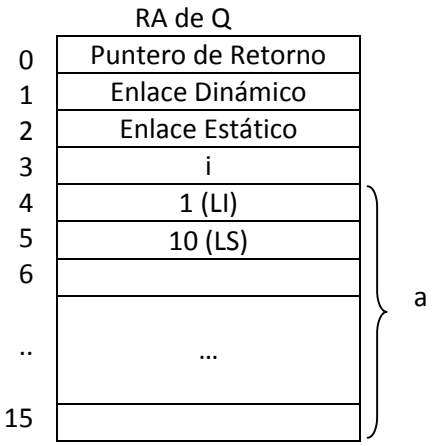
   Se debe recorrer la cadena estática tantos niveles como intervienen en la distancia

Implementación de Arreglos – Alcance Estático – Alocación Semiestática {Basada en Pila}

Al trabajar con arreglos se mantiene un descriptor para los objetos de datos vinculados a este tipo, que básicamente mantiene Límite Inferior (LI) y Límite Superior (LS). Por lo tanto, estos datos deberán también incluirse en el registro de activación (son conocidos en compilación).

Ej:

```
Program Principal
        Procedure Q
        var i:integer;
            a: array [1..10] of real;
        begin
            …
        EndQ
Begin
        …
        Call Q
End Principal
```



Acceso a componentes del arreglo:

a[i] = D[α + tamaño del descriptor del arreglo + (i - LI) * S] = D[(Actual+4) + 2 + (D[Actual+3] – D[Actual+4]) *S]

Obs: Debería chequearse antes que el índice esté dentro de los límites definidos (LI y LS)

Implementación de Arreglos – Alcance Estático – Alocación Semidinámica {Basada en Pila}

- Estructuras de datos de tamaño conocido al momento de activarse la unidad.
- El tamaño del RA no se conoce estáticamente, pero sí en el momento de su creación.
- Se mantienen una estructura de pila para la memoria.
- El acceso a las variables semidinámicas requi9ere un nivel de indirección.

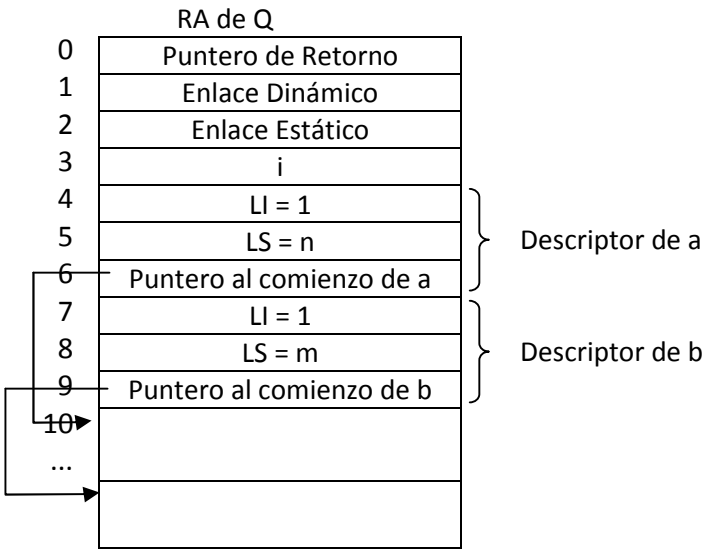
Al trabajar con arreglos semidinámicos, el tamaño del arreglo se conoce recién en ejecución, por lo que el registro de activación se divide en tres partes:

Registro de Activación

Puntero de Retorno
Enlace Dinámico
Enlace Estático
Objetos de datos locales estáticos
Descriptores de objetos de datos semidinámicos
Objetos de datos semidinámicos

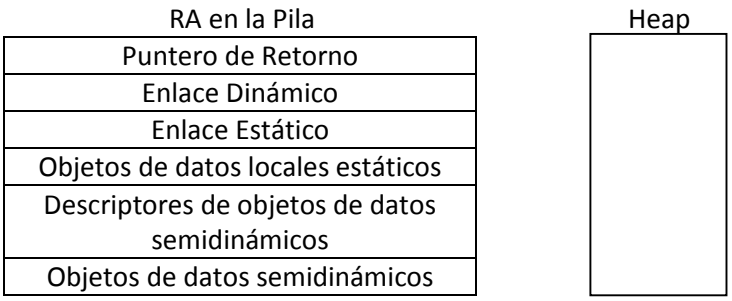
Ej:

```
Program Principal
        Procedure Q
        var i:integer;
            a: array [1..n] of real;
            b: array [1..m] of real;
        begin
            …
        EndQ
Begin
        …
        Call Q
End Principal
```



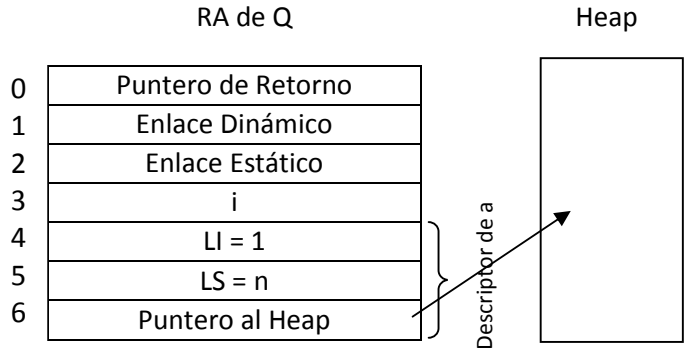
Implementación de Arreglos – Alcance Estático – Alocación Dinámica {Basada en Pila}

- Estructuras de tamaño variable.
- Se mantiene la estructura de pila y se agrega un heap.
- Al trabajar con arreglos dinámicos, el tamaño del arreglo no queda determinado en forma fija, ni siquiera en ejecución.
- El acceso a los objetos de dato dinámicos se realiza con un nivel de indirección.
- Con la direccipon del RA y el desplazamiento del arreglo, se obtiene la dirección efectiva del arreglo, es decir, la dirección del descriptor.
- Desplazándose en el descriptor se obtiene la dirección en el heap.
- Con la dirección en el heap y el subíndice se calcula el desplazamiento de la componente referenciada.



Ej:

```
Program Principal
        Procedure Q
        var i:integer;
            a: array [1..n] of real;
b: array [1..m] of real;
        begin
            …
        EndQ
Begin
        …
        Call Q
End Principal
```



IMPLEMENTACIÓN – PASAJE DE PARÁMETROS

Hay diferentes convenciones para el pasaje de parámetros. La elección está o bien predefinida por el lenguaje, o puede ser escogida por el programador entre un grupo de opciones. Es importante conocer cuál es la opción elegida, ya que puede afectar el significado del programa.

Requiere reservar espacio en el registro de activación para parámetros, y afecta a las operaciones a realizar en la *invocación*, *prólogo* y *epílogo* de la unidad:

- *Prólogo*: En la llamada, dependiendo del pasaje de parámetros, copia los valores de los parámetros o las referencias.
- *Epílogo*: En el retorno, si es necesario, copia los valores del resultado.

Consideraremos las siguientes formas de pasaje de parámetros:

- Pasaje por Referencia
- Pasaje por Copia
  - Por Valor
  - Por Resultado
  - Por Valor-Resultado
- Pasaje por Nombre

Pasaje de Parámetros por Valor {Copia}

Con este tipo de pasaje de parámetros, el parámetro formal se piensa como una variable local que se inicializa con el valor del parámetro actual en la invocación de la unidad.

→ En el RA de la unidad invocada se almacena: ptr, ED, EE, Parámetros, objetos de datos locales,…

Pasaje de Parámetros por Resultado {Copia}

La unidad llamadora es responsable de indicarle a la unidad llamada dónde quiere que se guarden los valores de los parámetros antes de retornar el control. La unidad llamada lo desconoce, debido a que el lugar donde debe guardar los valores se modifica de llamada en llamada.

→ El espacio para el resultado de los parámetros se reserva en la pila entre el RA de la unidad llamadora y el de la unidad llamada (En Pila[Actual-1], para que los almacene la unidad llamada en su epílogo).

Pasaje de Parámetros por Valor-Resultado {Copia}

La llamada es igual que en el caso anterior, al igual que el epílogo. Adicionalmente, requiere que en el prólogo se realicen las asignaciones de los valores de los parámetros actuales en los formales.

Pasaje de Parámetros por Referencia

También llamado pasaje "*by sharing*". Una referencia al correspondiente parámetro formal en la unidad llamada es tratada como una referencia a una locación cuya dirección ha sido pasada. Para ello, la unidad llamadora guarda la referencia que va a pasar en la pila antes de efectuar la invocación (Al igual que para pasaje por resultado, se almacena la dirección del parámetro destino en la pila, entre los RA de las unidades llamadora y llamada).

Si se tiene pasaje pro referencia y se invoca con una expresión, hay dos posibilidades:

- El compilador informa un error
- Se asigna al parámetro formal la dirección que tiene asociada la celda que contiene el resultado de la evaluación de la expresión

Pasaje de Parámetros por Nombre
La técnica básica de implementación consiste en reemplazar cada referencia a un parámetro formal con una llamada a una rutina (*thunk*) que evalúa una referencia al parámetro actual en el entorno apropiado.

Se crea una rutina para cada parámetro actual. En cada referencia se llama a la rutina que calcula el valor del parámetro actual, con lo que la sobrecarga en ejecución es importante, y hay una pérdida de legibilidad interesante en la traducción.

Se guarda en la pila la dirección de la primera instrucción de la rutina y el ambiente de referenciamiento (actual generalmente).

Procedimientos y Funciones como Parámetros
Los lenguajes que soportan variables de tipo rutina se dice que tratan a las unidades como objetos de primera clase. Si permiten que las mismas sean pasadas como parámetros, esos parámetros reciben el nombre de *parámetros procedurales*.

Esta práctica resulta útil en algunas situaciones prácticas, pero involucra la resolución de aspectos que hasta ahora eran inexistentes. Hasta ahora el llamado a una unidad involucraba reservar espacio en la pila para el registro de activación y setear el enlace estático → Ahora es imposible hacerlo, ya que no se conoce en la traducción cuál es la rutina que se recibe como parámetro. Esta información es recién conocida en ejecución.

(Si U pasa la unidad A por parámetro a otra unidad B, U debe determinar el EE y el tamaño de RA de A. Ya que A está en el contexto de U, no de B)

Para resolver estos problemas, la unidad llamada debe recibir:
- El tamaño del RA de la unidad que recibe como parámetro
- El correspondiente enlace estático

Se puede determinar el enlace a pasar por las reglas de alcance, y las reglas que haya decidido utilizar el lenguaje para resolver los accesos no locales en estos casos.

Para que una unidad U pueda pasar una unidad A como parámetro a otra unidad B, existen dos posibilidades:

a) La unidad U debe tener la unidad A dentro de su alcance, es decir, A debe ser visible.
   → El enlace estático a ser pasado es un puntero al RA que está a D pasos a lo largo de la cadena estática originada en la unidad llamadora. Donde D es la distancia entre el punto de llamada en la unidad donde el parámetro es recibido y su declaración.

b) A puede ser un parámetro formal de U.
   → El enlace estático a ser pasado es el mismo que fue oportunamente pasado al llamador (U lo recibió cuando recibió la unidad A como parámetro).

Con respecto a la invocación en sí, la única diferencia entre una invocación a una unidad tradicional y a una realizada a una unidad que se recibió como parámetro, es que la información con respecto al tamaño del RA y el EE se copia desde el área de los parámetros.

El impacto de este agregado a un LP no es menor. Es necesario extender los procedimientos de invocación para poder lidiar con la semántica adicional y la complejidad. Las invocaciones deben tratar con diferentes tipos de objetos, por lo que todas las partes involucradas sufren un incremento considerable en la complejidad.

Como ventaja, se tiene un lenguaje más uniforme, ya que las unidades son tratadas como cualquier otro objeto del lenguaje.

{ *Ejemplos – Traducción de programas con pasaje de unidades como parámetro* }

Ej1.

```
Proc T
Var v
        Proc Q (proc X)
        Var u
            Call X
        End Q

        Proc P
        Var u
                Proc R
                    Call Q(R)
                End R
        End P
End T
```

Ej2.

```
Proc T
Var v
        Proc Q (proc X)
        Var u
                Proc R
                Var u,v
                End R
        End Q

        Proc P
        Var u
            Call Q(R)
        End P
End T
```

Ej3.

```
Proc Q (proc X)
Var u,v,y
        Proc T (proc X)
            y=1
        End T

   Call X
   Call Q(T)

End Q
```

Ej1 → En este contexto la traducción es correcta, ya que cuando se realiza la llamada *call Q(R)*, P está activo y por lo tanto R está en la tabla de símbolos.
En el momento en que el compilador traduce, mira en la tabla de símbolos, encuentra R, y permite que se realice la llamada. Liga el EE del proc X a P, lo cual es lógico, porque R está en el ambiente de referenciamiento de P.

Ej2→ La llamada *call Q(R)* no es correcta, ya que cuando el compilador quiere traducir R, ya no está en la tabla de símbolos porque Q ya terminó de compilarse, y la información sobre R ya no está en la tabla de símbolos.

Ej3 → Cuando se resuelve el EE de T, el único RA activo es el primer registro de activación de Q. Por ello, cuando se hace referencia a *y* en T, se hace referencia a la variable que está en el primer RA de Q.

LP CON REGLAS DE ALCANCE DINÁMICO

Este tipo de lenguajes cuenta con las siguientes características:

- Generalmente, las ligaduras de tipo son también dinámicas
- La estructura de los programas es de bloques chatos
- Se cuenta con declaraciones implícitas o explícitas

Bajo esta modalidad, el uso de un objeto de dato provoca la búsqueda del nombre de ese objeto en el RA de la unidad en curso. Si no lo encuentra allí, lo busca en la unidad que la invocó, y así sucesivamente (se sigue la cadena dinámica).

El descriptor de los objetos de datos se mantiene en el *heap*, el cual es un bloque de memoria compuesto de elementos de tamaño fijo o variable, dentro del cual se puede reservar y liberar espacio de manera no estructurada.

De este modo, en el RA de la unidad llamada en la sección de variables locales, se guarda el nombre del identificador, y adicionalmente se almacena:

- ~~Un puntero al heap en el cual se encuentra toda la información que en el caso de alcance estático era guardada en la tabla de símbolos, además del valor del objeto de dato.~~

- Dos referencias al heap junto con el nombre del identificador. Una al lugar donde está la información sobre el objeto de dato, y otra al lugar donde se almacena el valor.

[En lugar de almacenar el descriptor en la pila, se almacena en el heap, y se guarda en la pila un puntero al descriptor]

Implementación de LP Orientados a Objetos – POO

Los aspectos más importantes de la POO son una técnica de diseño dirigida a la determinación y delegación de responsabilidades. En general, los lenguajes orientados a objetos proveen:

- *Abstracción de Datos* (encapsulamiento de un estado con operaciones).
- *Encapsulamiento* (ocultamiento de información)
- *Polimorfismo* (pasaje de mensajes)

Adicionalmente implementan *Herencia* y *Ligadura Dinámica* (de mensajes a métodos).

Abstracción de Datos

La representación de las estructura de datos y operaciones es fija, indicando que el tipo de dato abstracto es implementado. Desde el punto de vista teórico, un TDA es:

- Un conjunto de objetos de datos. Comúnmente se utilizan una o más definiciones de tipo.
- Un conjunto de operaciones para manipular esos objetos de datos.

En las soluciones no OO, las estructuras de datos y las operaciones son piezas diferentes. En contraste, las implementaciones OO requieren encapsulamiento, dada la naturaleza dual de los objetos →El constructor para definir TDA es la *clase*.

Encapsulamiento

Al contar con encapsulamiento, el usuario de la abstracción no necesita conocer la información oculta en la abstracción. No es posible utilizar o manipular la información oculta, aunque se desee hacerlo.

Polimorfismo (Por Inclusión)

En el contexto de la programación orientada a objetos, implica que en una misma jerarquía de herencia, se puede responder al mismo mensaje en forma diferente.

Todas las clases derivadas de la misma clase pueden ser vistas informalmente como versiones especializadas de su clase base → Los LOO proveen variables polimórficas que pueden referenciar a objetos de diferentes clases (Ej: un objeto declarado como de clase *Pila* puede referenciar a un objeto de clase *PilaEnteros*).

Representación de Objetos

La representación del objeto debería mantener la información que cambia de instancia en instancia. La información que se mantiene igual para todas las instancias se mantiene en la *representación de la clase*, e incluye los métodos de clase e instancia.

La *representación del objeto* es básicamente un *registro* que contiene los valores de las variables de instancia, y una referencia a la representación de la cual es instancia (solución más simple).

Representación de Clases

Es necesario mantener la siguiente información:

- Nombre de la clase
- La superclase (en caso de no ser especificada, es *object*)
- Los nombres de las variables de instancia
- Un diccionario con los métodos de clase
- Un diccionario con los métodos de instancia

Representación del Registro de Activación

Se mantiene un RA de manera tal que es posible mantener la información relevante a la activación de un método. El registro sigue manteniendo tres partes, como en su versión para lenguajes imperativos:

- *Entorno*: Contexto utilizado para la ejecución del método (entorno local y no local).
- *Instrucciones*: La instrucción a ejecutarse cuando el método se reanude.
- *Emisor*: El registro de activación del método que envió el mensaje invocando al método actual (enlace dinámico).

Enviando y Retornando Mensajes

Al momento de *enviar un mensaje* se debe:

- Crear el RA para el método llamado.
- Identificar el método invocado extrayendo el patrón del objeto receptor o superclase.
- Transmitir los parámetros al RA del receptor.
- Suspender al llamador, salvando su estado en el RA.
- Establecer el camino de regreso al emisor, y colocar el RA como el registro activo.

Al *retornar* se debe:

- Transmitir el objeto resultante (si lo hay) al emisor.
- Reanudar la ejecución del emisor restaurando su estado con la información en su registro de activación.

Implementación de Constructores Orientados a Objetos

Hay al menos dos partes de la POO que provocan preguntas interesantes a los implementadores de LP:

- *Las estructuras de almacenamiento para las variables de instancia*
- *La ligadura dinámica de los mensajes a los métodos*

Almacenamiento de los Datos de una Instancia

La estructura de almacenamiento tiene claras similitudes al registro de activación. Se llama CIR (*Class Instance Record*). La estructura del CIR es estática, se construye en tiempo de compilación y es utilizada como template para la creación de instancias de clase. Los accesos a los atributos de una instancia se resuelven de igual manera que los accesos a las componentes del registro de activación, y con la misma eficiencia.

Herencia
El contar con herencia no agrega demasiado trabajo al traductor, ya que en una clase derivada sólo los nombres de la clase padre son agregados al espacio de nombres local, y sólo los públicos tienen visibilidad para los usuarios de la clase. La determinación del CIR siegue siendo estática, debido a las declaraciones en la definición de la clase. Hay dos aproximaciones a la implementación de herencia:

- *Aproximación basada en Copia*: Es la más simple y directa de implementar. Los objetos instancia de la clase derivada tienen todos los detalles de su implementación.

- *Aproximación basada en Delegación*: En este modelo cada objeto instancia de una clase derivada utiliza el almacenamiento de la clase padre. Las propiedades heredadas no son duplicadas en el objeto derivado. Para la implementación de este modelo es necesario contar con alguna forma para compartir datos, de manera que los cambios en el objeto base generen cambios en el objeto derivado.

Métodos Virtuales (métodos abstractos de Java)
Los métodos virtuales pueden ser implementados de manera similar a los registros de activación. Cada método virtual en una clase derivada reserva un slot de registro que define la clase. El procedimiento constructor simplemente rellena la locación con la información que necesita.

Ligadura Dinámica de los Mensajes a los Métodos
[Una clase derivada puede no sólo agregar nueva funcionalidad a su clase base, sino que además puede añadir datos privados y redefinir algunas de las operaciones provistas por la clase base. Esta ligadura dinámica puede conducir a violaciones de tipo → (~~Ej. a una variable de clase Pila que referencia a un objeto de clase Pila, pero puede almacenar un objeto de clase PilaEnteros, se le pide el método desapilarEntero(), el cual no tiene definido)~~].

Los métodos en una clase están estáticamente ligados y no es necesario incluirlos en el CIR de la clase. Sin embargo, los métodos que pueden ligarse dinámicamente deberían mantenerse en esta estructura. La entrada podría consistir simplemente de un puntero al código del método, puntero que puede ser seteado al momento de creación del objeto.

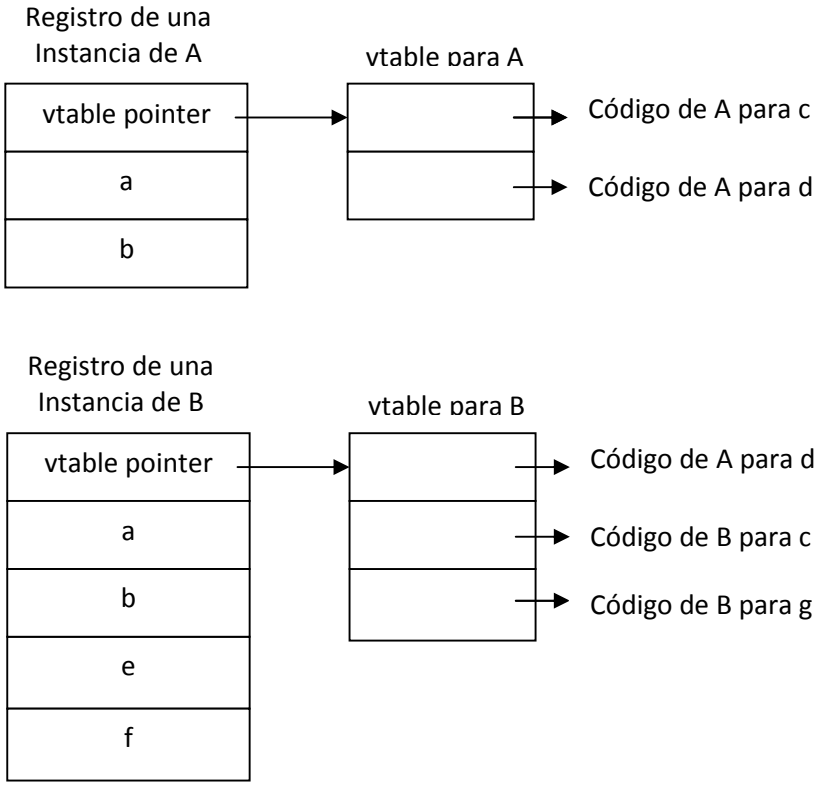
El aspecto negativo de esta solución es que todos los objetos instancias de una clase deben mantener los punteros de todos los métodos que se liguen dinámicamente. La desventaja radica en que el conjunto de estos métodos es el mismo para todas las instancias de la clase (depende de la clase, no de la instancia) y por lo tanto, pueden almacenarse una sola vez. Este almacenamiento puede ser referenciado por todas las instancias de la clase.

→ De esta manera, el CIR sólo necesita mantener un solo puntero a una lista para poder encontrar el método llamado. La estructura para almacenar esta lista es llamada *tabla virtual de métodos* (*vtable*). Las llamadas a los métodos pueden ser representadas por el *offset* desde el comienzo de *vtable*.

{ Ejemplo – Implementación con vtable }

```
Public class A {

public int a, b;

public void c() { … }
public void d() { … }

}
```

```
Public class B extends A {

public int e, f;

public void c() { … }
public void g() { … }

}
```



Genericidad

Un tipo de dato abstracto genérico, en principio, tiene una implementación directa. Los parámetros genéricos deben darse cuando se instancia el paquete. El compilador usa la clase genérica, como un molde al cual una vez que se le insertan los parámetros adecuados funciona como una clase tradicional.

Polimorfismo en Lenguajes Dinámicos

Para los lenguajes tipados estáticamente, el polimorfismo no agrega complejidad. En cambio, para los que permiten *polimorfismo dinámico* aparecen dificultades, ya que la cantidad de argumentos de una función polimórfica debe determinarse durante la ejecución (Ej: para lenguajes altamente dinámicos, donde un método se considera polimórfico si tiene más de una definición con el mismo nombre → puede tener distinta cantidad de parámetros, de distinto tipo, etc. Entonces en compilación no se puede determinar cuáles son los parámetros correctos).

Hay dos maneras de pasar los argumentos a una función polimórfica:

- A través de un descriptor inmediato (se utiliza un bit para indicar cuál es el tipo del parámetro).

- A través de un descriptor. Se otorga la información completa del tipo (es como dar la estructura completa de los objetos de datos compuestos).

[VER HERENCIA Y TIPADO FUERTE EN LENGUAJES OO! – Pág 47 – Capítulo VI Sebesta – Capítulo III Ghezzi]

IMPLEMENTACIÓN DE EXCEPCIONES [Capítulo XI – Pratt]

Durante la ejecución de un programa pueden ocurrir eventos o condiciones que son considerados excepcionales (excepciones), y que tienen dos posibles orígenes:

- *Condiciones detectadas por la máquina virtual*: El SO puede lanzar excepciones directamente a través de interrupciones de HW ~~o traps~~ (Ej: overflow en operaciones aritméticas) o en SW de soporte (Ej: EOF).

- *Condiciones generadas por la semántica del LP*: El LP puede proveer excepciones adicionales, si el traductor del lenguaje añade instrucciones adicionales en el código ejecutable (Ej: chequeo de límites de un arreglo cuando se produce un acceso → chequeo costoso tanto en almacenamiento de código como en tiempo de ejecución).

Una vez lanzada la excepción, la transferencia de control al manejador (handler) en el mismo programa generalmente se implementa como un jump directo al comienzo del código del manejador.

La *propagación de excepciones* a través de la cadena dinámica de llamadas puede hacer uso de la cadena dinámica formada por los punteros de retorno que se encuentran en los registros de activación de los subprogramas en la pila. A medida que se avanza por la cadena dinámica, cada subprograma debe ser terminado con una forma especial de instrucción de retorno, para retornar el control al llamador y para lanzar la excepción nuevamente en el llamador (propagarla).

La secuencia de retornos continúa a través de la cadena dinámica hasta que se encuentra un subprograma que posee un manejador para la excepción lanzada. Una vez encontrado en manejador, es invocado mediante una llamada común a un subprograma. Cuando el manejador termina se termina en cascada la ejecución de todos los métodos que propagaron la excepción. Desde que se ejecutó el manejador hacia adelante (lo que no propagó la excepción) se continúa la ejecución normal.

Obs: En el caso que se finalice la búsqueda del handler (se recorrió toda la cadena dinámica y no se encontró) se produce la terminación del programa.

Paradigma Lógico – Prolog

INTRODUCCIÓN

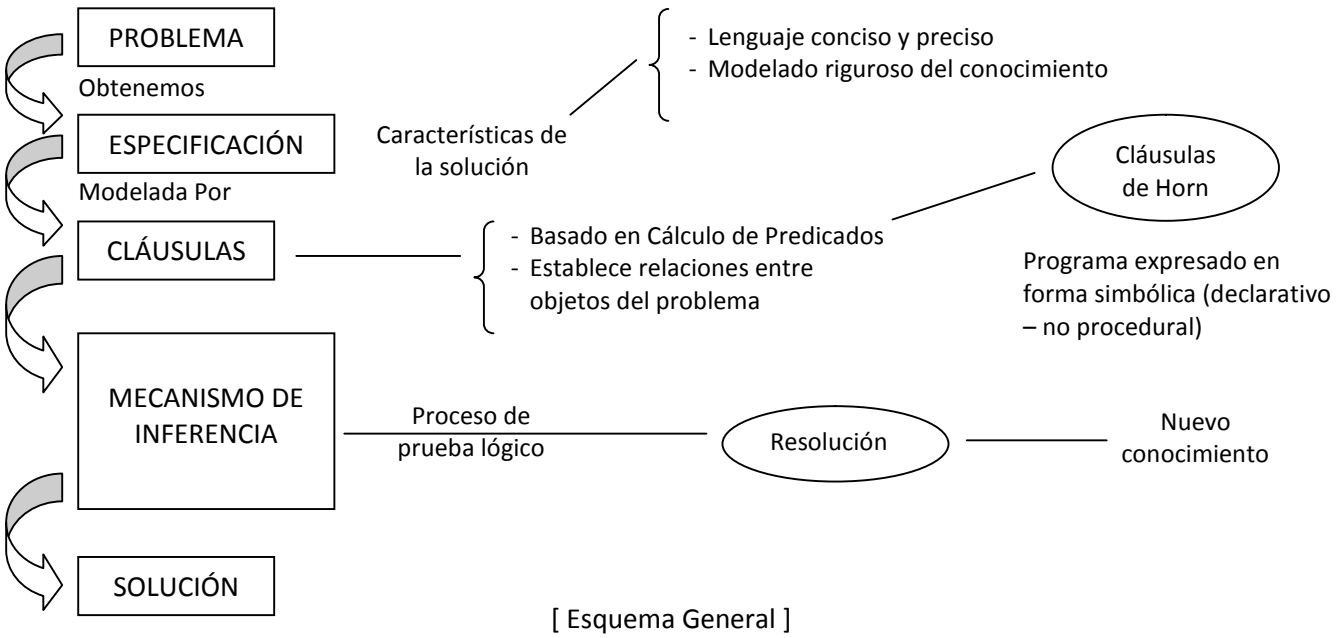
Una forma de razonar para resolver problemas en matemática se fundamenta en la *lógica de primer orden*. El conocimiento básico de la matemática se puede representar en la lógica en forma de axiomas, a los cuales se le agregan reglas formales para deducir cosas verdaderas (teoremas). Los lenguajes que utilizan esta lógica se llaman *lenguajes declarativos*, porque todo lo que tiene que hacer el programador para solucionar un problema es describirlo vía axiomas y reglas de deducción.

[Paradigma Declarativo → Un programa establece un conjunto de propiedades que describen cómo alcanzar la solución. En ningún caso se indica cómo se computa. Un lenguaje declarativo brinda las facilidades para especificar estas propiedades y una forma de computar que sea transparente. - Poca eficiencia + Programas elegantes y concisos]

Este concepto de *programación lógica* está ligado históricamente a *Prolog* (Programmation en Logique). Prolog es utilizado para el desarrollo de aplicaciones de IA debido a su forma de representar el conocimiento, la escritura de compiladores, la construcción de sistemas expertos, el procesamiento de lenguaje natural, búsqueda de patrones, y programación automática.

Obs: La lógica proposicional sólo puede representar hechos acerca del mundo. La *lógica de primer orden* describe un mundo que consta de objetos y propiedades (o predicados) de esos objetos. Entre los objetos, se verifican varias relaciones. Una función es una relación en la cual sólo hay un valor para un input dado.

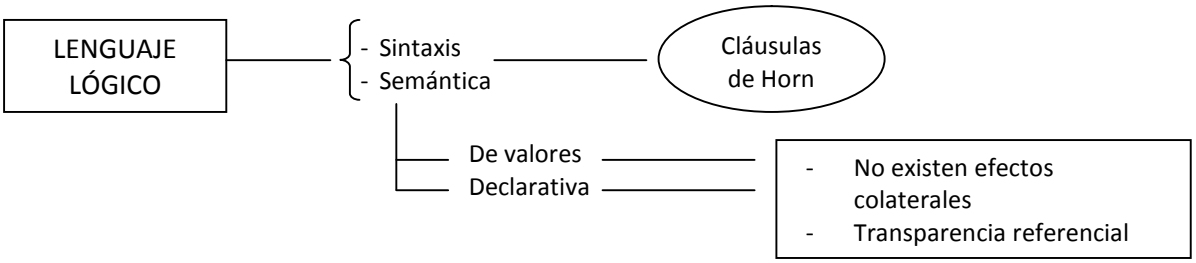
PARADIGMA LÓGICO



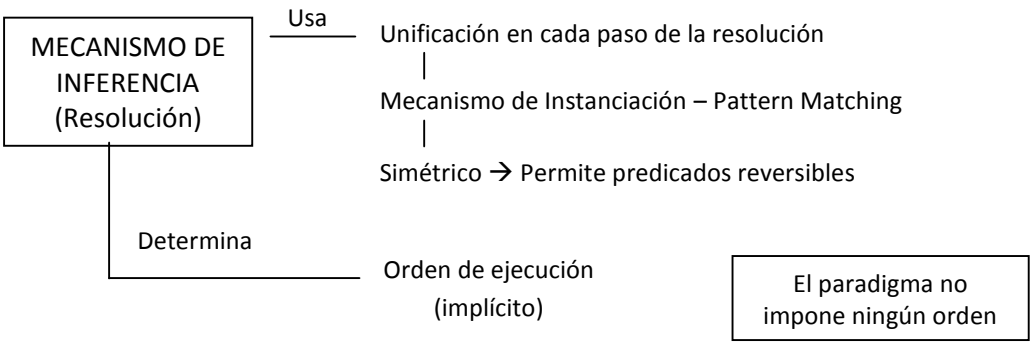
[Esquema General]

Elementos del Paradigma

El *lenguaje lógico* define la sintaxis y semántica de las especificaciones



El *mecanismo de inferencia* permite derivar nuevo conocimiento y hallar soluciones

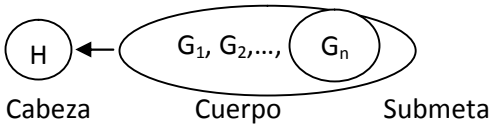


<u>*Lenguaje Lógico – Sintaxis*</u>:

Los lenguajes lógicos constan de *términos*:
- Una variable es un término
- Si f es un símbolo funcional n-ario y t_1, t_2, … , t_n son términos, entonces f(t_1, t_2, … , t_n) es un término. Si n=0 → f es una constante.
- Los términos son los argumentos de los predicados.

Predicado: Si p es un símbolo relacional n-ario y t_1, … , t_n son términos, entonces p(t_1, … , t_n) es un átomo.
→ El conjunto de predicados conforman el programa lógico o base de conocimiento.

Forma de una <u>*Cláusula de Horn*</u>:



El significado declarativo de la cláusula es: si G_1,…, G_n son verdaderas, entonces H es verdadera.

Son cláusulas que tienen a lo sumo un literal positivo: $(p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q) \equiv (\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_n \vee q)$

$H \leftarrow G_1,\ldots, G_n$

Se interpreta el cuerpo de la regla como una nueva consulta. La nueva consulta es satisfecha $\Leftrightarrow$ todas las sub-metas son satisfechas. Si n=0 tenemos un *hecho*.

Semántica: coincide con la provista por la lógica. Cada regla es una implicación lógica.

+ Simplifican el mecanismo de resolución
+ Expresan conocimiento sin redundancia

Mecanismo de Inferencia:

El mecanismo de inferencia que se utiliza se denomina *resolución* y permite derivar consecuencias lógicas a partir de programas lógicos (conjunto de cláusulas). La cláusula obtenida se denomina la *resolvente* del conjunto de cláusulas inicial.

El mecanismo de *unificación* es el que permite realizar el matching entre las metas que se quieren probar y la cabeza de alguna regla. Se realiza una instanciación en cada paso de la resolución.

Dado un programa lógico y una consulta compuesta por una o varias metas, el mecanismo de *resolución* intentará unificar alguna de dichas metas con la cabeza de alguna cláusula del programa, en algún orden. Como resultado se obtendrá una nueva cláusula que puede verse como una nueva consulta. El proceso se repite hasta que:

- Alguna meta no pueda unificarse con ninguna cabeza, en cuyo caso la consulta fallará
- Para todas las metas existen hechos con los cuales unificar, por lo tanto la consulta es exitosa.

Mecanismo de Inferencia – Unificación

No todas las variables están obligadas a quedar ligadas.

Ej: h(X), h(Y) unifican pero las variables X e Y no quedan ligadas. No obstante, ambas variables permanecen unificadas entre sí. Si posteriormente ligamos X al valor 3 (por ejemplo), entonces automáticamente la variable Y tomará ese mismo valor. Lo que ocurre es que al unificar los términos dados, se impone la restricción de que X e Y deben tomar el mismo valor aunque en ese preciso instante no se conozca dicho valor.

El = es el operador de unificación.

Para saber si dos términos unifican:

- Una variable siempre unifica con un término, quedando ligada a dicho término.
- Dos variables siempre unifican entre sí, además, cuando una de ellas se liga a un término, todas las que unifican con ella se ligan a dicho término.
- Para que dos términos unifiquen deben tener el mismo functor y la misma aridad. Después se comprueba que los argumentos unifican uno a uno, manteniendo las ligaduras que se produzcan en cada uno.
- Si dos términos no unifican, ninguna variable queda ligada.

[Pratt] → Prolog usa la *unificación* (o sustitución de variables) en relaciones, para "matchear" patrones con el fin de determinar si una consulta tiene una sustitución válida, consistente con las reglas y hechos en la base de conocimiento. Mediante el *pattern matching* descubre si una consulta se resuelve con un hecho de la base de creencias, o si se puede deducir usando reglas aplicadas a otras reglas y hechos.

Consideraciones sobre el Paradigma

- El *paradigma lógico* es un paradigma declarativo → se especifica la solución, no cómo llegar a ella.
- Existen distintas *formas de explorar el espacio de soluciones*:
  - *Forward Chaining*: Se procede desde los hechos hacia la consulta.
  - Backward Chaining: Se procede desde la consulta hacia los hechos.
- El proceso (*recorrido del espacio de soluciones*) puede realizarse:
  - *A lo ancho (BFS)*: garantiza encontrar una prueba si es que existe. Sensatez y completitud. Mayor tiempo de ejecución y menor eficiencia.
  - *En profundidad (DFS)*: sensatez, no completitud. Sensible a la forma de escribir la especificación.
- La lógica subyacente provee al paradigma un lenguaje riguroso, preciso, elegante, uniforme y ortogonal, dando el resultado de programas más legibles y concisos (cada predicado se define en términos de sí mismo).
- Semántica de Valores: una variable queda ligada a un único valor (no se tiene la noción de locación de memoria que almacena un valor y puede ser modificado).
- Autodescriptible: entorno de ejecución especificado en el mismo lenguaje.
- No hay noción de Tipos de Datos → menor seguridad. Cualquier restricción es especificada por el programador mediante reglas.
- El paradigma lógico no impone restricciones de orden en cuanto al flujo de control de la ejecución. El flujo de control se rige por el mecanismo de unificación implícitamente de manera no determinística.
- La unificación determina cuál o cuáles cláusulas matchean con la consulta realizada.
- No es eficiente en algunos casos → mayor tiempo en exploración en espacio de soluciones.
- El matching es simétrico, entonces permite predicados reversibles

Predicados Reversibles

La propiedad de *reversibilidad* no existe en otros lenguajes, y consiste en la habilidad de los argumentos de los predicados para actuar indistintamente como argumentos de entrada y/o salida.

Ej: El predicado para sumar dos números enteros en notación s^n también sirve para restar.

```
suma(X,0,X).
suma(X,s(Y),s(Z)):- suma(X,Y,Z).
```

→ Para sumar 1+1 se puede consultar
```
?- suma(s(0), s(0),R)
R = s(s(0))
```

→ Para restar 1-1 se puede consultar
```
?- suma(s(0), R, s(0))
R = 0
```

A las diferentes formas de invocar un predicado se las denomina *modos de uso*. Los modos de uso indican qué combinación de argumentos deben estar instanciados para que una consulta tenga sentido. Se dice que un argumento está instanciado cuando no es una variable libre.
No todos los predicados son reversibles (Ej: predicados de comparación aritmética – mayor(X,1) → no es posible conocer todos los números mayores que 1, ya que son infinitos).

Variables Lógicas

El paradigma lógico soporta el concepto matemático de variable. Las *variables* son nombres que representan valores desconocidos. No existe una asociación con una locación de memoria en la cual se almacenan distintos valores en distintos instantes de tiempo. Las variables se asocian a valores (*semántica de valores*).
El valor que puede tomar una variable consiste en cualquier término, por eso se dice que los datos que maneja Prolog son términos.
Cuando una variable no tiene valor se dice que está *libre*. Una vez que se le asigna valor, este ya no cambia, por eso se dice que la variable está *ligada*.

Noción de Tipos

No se provee la noción de tipo de los lenguajes imperativos. Sólo se tienen términos y símbolos predicativos.
La ausencia de un sistema de tipos permite hacer un mal uso de ciertos predicados, dando menor seguridad y confiabilidad. El programador debe encargarse de especificar axiomáticamente cualquier restricción que desea reflejar en el programa.
Ej: Es posible utilizar el predicado suma en notación s^n de la siguiente forma:

```
?- suma(algo,s(0),X)
X = s(algo)
```

Interpretaciones del Paradigma

Declarativa: La semántica de cada predicado se deriva de su definición. No se requiere examinar su contexto.
Procedural: Es la más alejada del paradigma, y es necesaria para analizar ciertas características de los lenguajes lógicos no puros.

→ La ejecución de cierta cabeza implica la ejecución de su cuerpo, así se hace explícito el flujo de ejecución pensando en una cláusula como la llamada a una unidad.

LENGUAJE LÓGICO PURO – CARACTERÍSTICAS

- Permitir la especificación de predicados basados en la lógica, que definan relaciones entre objetos.
- Poseer un mecanismo de inferencia que permita derivar nuevo conocimiento a partir del existente.
- Proveer el mecanismo de unificación para lograr la instanciación de variables en términos que permitan al mecanismo de inferencia obtener nuevo conocimiento.
- Proveer la noción matemática de variable (semántica de valores).
- No imponer restricciones para la selección de reglas ni metas en el cómputo de las soluciones.
- Ser autodescriptible (autocontenido). Debiera ser posible obtener un entorno de ejecución especificado en el mismo lenguaje (Ej: metaintérprete de Prolog).
- Asegurar la ausencia de efectos colaterales de manera de obtener la declaratividad en las especificaciones (relacionado con el concepto de transparencia referencial)

PROLOG

- Respeta la sintaxis y semántica definidas por el paradigma
- Concepto de ligadura: unificación
- Concepto de variable: semántica de valores
- Tipos de datos: no hay noción de tipos
- Evaluación de expresiones

Evaluación de Expresiones

Cuando aparece una expresión en un predicado, no siempre es evaluada. Las expresiones son evaluadas cuando se utiliza el predicado *is* y cuando se utilizan operadores relacionales.

Cuando se tiene una expresión que involucra el predicado *is*, la parte derecha debe estar evaluada, y si la parte izquierda está evaluada, se comparan por igualdad:

A is X/Y → X, Y deben estar instanciadas con valores aritméticos.
Si A está instanciada, se compara por igualdad

Cuando se utilizan operadores relacionales, las apariciones de variables a izquierda y a derecha deben estar instanciadas con valores enteros:

A>=B → A y B deben estar instanciadas con valores enteros

Ventajas y Desventajas del predicado *is*:

+ Facilidad para el programador No implementa aritmética simbólica.
- Menos uniformidad para el lenguaje por restricciones en la ligadura entre variables y expresiones.
- Falta de reversibilidad

Características que alejan a Prolog del Paradigma Lógico

Existen características que alejan a Prolog de lo propuesto por el paradigma lógico. Dichas cuestiones le quitan legibilidad, uniformidad y ortogonalidad al lenguaje, pero son incluidas principalmente para obtener una mejor performance en la implementación, y para dotar al lenguaje de características adecuadas en ciertos dominios de aplicación.

1. Estrategia de búsqueda en el espacio de soluciones – Orden de evaluación

Si bien el paradigma lógico no impone ninguna restricción, Prolog adopta *backward chaining* con DFS y backtracking. Este esquema si bien utiliza menor cantidad de recursos, compromete la terminación de programas lógicos en caso de que se exploren ramas infinitas, aún cuando existen ramas exitosas de longitud finita.

Prolog intenta unificar por la primera regla que aparece en el programa, y evalúa las submetas de izquierda a derecha →El programador debe tener en cuenta este orden para mejorar la performance de los programas y especialmente para no comprometer la terminación de los mismos.

2. Occur Check

En el algoritmo de unificación existe un chequeo que controla si es posible unificar una variable con un término que contiene la misma variable. En caso de que así suceda, claramente ambos términos no unifican, y el algoritmo concluye sin una sustitución válida. Es un chequeo dinámico que depende de la implementación de Prolog.

Por cuestiones de eficiencia, Prolog no lo realiza. Esto compromete la terminación de los programas y afecta la semántica, pues una consulta que debería ser rechazada (según la semántica del paradigma) nunca termina.

Ej: X = f(X)
f(f(f(f(f(f(f(f(f(f(f(…

3. Control de Backtracking - cut

Prolog cuenta con un predicado predefinido que permite realizar control sobre el backtracking por parte del programador. Dicho predicado se denomina *cut* y se denota con "!". Es un aspecto que debe ser analizado bajo la interpretación procedural.

Es un predicado que siempre se satisface, pero tiene el efecto colateral de acotar el espacio de búsqueda una vez ejecutado.

Ventajas y desventajas del uso del *cut*:
+ Implementaciones más eficientes en algunos casos
+ Elimina soluciones repetidas si se hace un uso cuidadoso del mismo
- Oscurece la semántica del programa lógico

4. Negación por Falla

En Prolog, debido a la restricción de que únicamente posee cláusulas de Horn, los programas lógicos sólo pueden mantener información positiva respecto al problema. Prolog sólo puede probar las verdades especificadas en el programa lógico: puede probar que algo es verdadero pero sólo asumir que algo es falso cuando no pudo probar su veracidad (Closed World Assumption – CWA).

Se incluye el predicado predefinido not/1 que corresponde a la negación por falla:

```
not(X):- call(X), !, fail.
not(X)
```

La semántica de la negación por falla difiere de la negación de la lógica de predicados, y no admite una interpretación declarativa. El programador para utilizarlo correctamente debe estar consciente de los efectos que se generarán en ejecución (semántica procedural).

La semántica de la doble negación no es la provista según la lógica. Además para comprender la respuesta a la segunda consulta (Ej: not(not(member(X,[a,b,c])))) hay que analizar el flujo de control bajo una óptica procedural. Este es un ejemplo de cómo el *not* puede quitarle declaratividad al lenguaje.

5. Evaluador aritmético is – Predicados Relacionales

El evaluador aritmético *is* y los operadores relacionales imponen restricciones en sus argumentos y no son reversibles → Le quitan uniformidad al lenguaje.

El operador *is* principalmente liga una variable al resultado de evaluar una expresión aritmética. La evaluación se realiza utilizando la aritmética de la arquitectura subyacente. Se exige que las variables de la expresión aparezcan instanciadas con valores aritméticos al momento de la ejecución del operador.

Algunas implementaciones de Prolog exigen que la variable a la que se ligará el resultado de la expresión no se halle ligada al momento de la ejecución del *is*. Otras implementaciones permiten que ya tenga ligado un valor, en cuyo caso proceden a realizar una comparación entre el valor que posee la variable y el resultado de evaluar la expresión, y sólo si coinciden se satisface el operador.

Para los operadores relacionales se exige que las variables involucradas en la comparación se encuentren instanciadas con valores enteros [Ojo! Versiones nuevas permiten reales].

Las restricciones impuestas en cuanto a la ligadura de las variables involucradas y a la falta de reversibilidad del *is* y los operadores relacionales, son características que le quitan uniformidad al lenguaje. Sin embargo, si bien es posible implementar aritmética simbólica mediante estructuras recursivas como la notación s^n, el operador *is* no deja de ser una facilidad para el programador.

6. Predicados assert y retract

Estos predicados predefinidos permiten la incorporación y eliminación de cláusulas dinámicamente → Su utilidad radica en el *efecto colateral* de añadir/eliminar cláusulas.

Debe destacarse que cuando el mecanismo de backtracking de Prolog concluye la búsqueda en una rama del espacio de soluciones, los efectos colaterales de estos predicados no son deshechos.

Ventajas y desventajas:
+ Facilita el desarrollo de aplicaciones que necesitan modificación dinámica de la base de conocimiento.
- Uso cuidadoso, requiere análisis procedural.

Obs: Los predicados eliminados con *retract* deben estar declarados como dinámicos. Muchas implementaciones de Prolog no permiten la eliminación de predicados que no fueron añadidos dinámicamente mediante *assert*.

Paradigma Funcional – ML

LENGUAJES FUNCIONALES

El *paradigma funcional* (o aplicativo) se fundamenta en la evaluación de expresiones construidas a partir de la combinación de funciones. Una *función*, al igual que en matemática, mapea valores tomados de un conjunto denominado *dominio*, en valores de otro conjunto conocido como *rango* o codominio.

→La computación se logra mediante la evaluación de expresiones (NO mediante evaluación de expresiones y ejecución de sentencias). Las expresiones pueden o no tener un tipo (fuerte o débilmente tipadas), retornar un valor o no (excepción o no "terminación"), y pueden ocasionar ~~un efecto o no (efecto colateral – elevar una excepción, modificar la memoria,~~ realizar entrada o salida, etc.)

Componentes de los Lenguajes Funcionales

Los lenguajes funcionales se caracterizan por cuatro componentes:

- *Un conjunto de Objetos*: El conjunto provisto depende de las características del lenguaje, ya que son los elementos de los dominios y rangos de las funciones (Ej: Standard ML es fuertemente tipado y sus objetos deben ser de algún tipo predefinido o definido por el programador).

- *Un conjunto de Funciones Primitivas*: Las funciones predefinidas por el lenguaje dependen del dominio de aplicación del mismo.

- *Un conjunto de Formas Funcionales*: Una *función de primer orden* es aquella en la cual ninguno de sus argumentos y/o resultado es una función. Las *formas funcionales (funciones de alto orden)* son aquellas cuyos argumentos y resultado pueden ser funciones.

  El conjunto de formas funcionales provistas por un lenguaje puede tener un fuerte impacto en el poder expresivo del mismo. El exceso en el número de funciones primitivas puede ir en detrimento de la sencillez del lenguaje sin aumentar su expresividad.

- *Una Operación de Aplicación*: La operación de aplicación de funciones constituye el principal mecanismo de control de los lenguajes funcionales. Una función puede llamar a otra, o ser recursiva de manera ~~in~~directa o indirecta. La aplicación puede ser definida de distintas formas (Ej: Standard ML soporta la aplicación parcial o currying).

  En general, una aplicación tiene la forma: *e e'*, donde *e* y *e'* son expresiones. Esto establece que la expresión *e* será aplicada a la expresión *e'* (Ej: doble 4*5 → El resultado de esta aplicación es 40). En esta forma de establecer la aplicación se asume implícitamente que todas las funciones poseen un único argumento.

  Para pasar más de un parámetro a una función hay dos alternativas:

  - Empelar estructuras de datos para agrupar argumentos (se sigue teniendo un único argumento, sólo que se trata de un valor compuesto).
  - Currying

Propiedades de los Lenguajes Funcionales Puros
Los *lenguajes funcionales puros* poseen varias propiedades que los distinguen:

- *Semántica de Valores*: Desaparece la noción de estado (von Neumann), dado que no existen los conceptos de locación de memoria (cuya abstracción son las variables) y de modificación de las mismas (sentencia de asignación) → El ambiente de ejecución asocia variables a valores únicamente, y una variable dentro de un ambiente no puede cambiar su valor.

- *Transparencia Referencial*: En un determinado contexto, la semántica del sistema puede ser determinada a partir de la semántica de sus partes sin importar computaciones previas ni el orden de evaluación (y la semántica de cada una de sus partes es independiente de la semántica de las demás).

  [**Lenguajes Funcionales**] Una función computa valores basándose únicamente en sus argumentos (parámetros de entrada) → *La ejecución de una función para el mismo valor del argumento siempre dará el mismo resultado* (no produce efectos colaterales).
  Obs: Si un LF fuera extendido con variables globales pareciera que se perdería la transparencia referencial, pero no, ya que posee semántica de valores (las variables se ligan a un único valor)

  → La transparencia referencial es importante porque permite al programador *razonar acerca del comportamiento del programa*, lo cual puede ayudar a probar su correctitud, encontrar bugs que no pueden ser encontrados mediante testing, simplificar el algoritmo, o incluso optimizar el código.

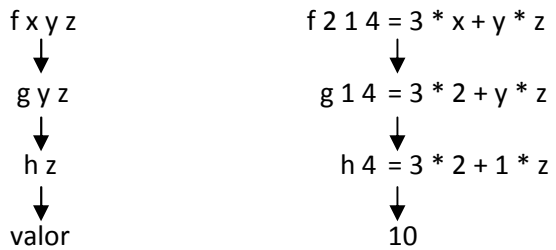
- *Funciones como Valores de Primera Clase*: las funciones pueden ser pasadas como argumentos, retornadas como resultados de otras funciones, y pueden combinarse para componer nuevas funciones.
  Contribuye a la regularidad y flexibilidad de los lenguajes funcionales.

- *Currying*: Permite reducir funciones con múltiples argumentos a funciones de un argumento. Permite evaluar un número variable de argumentos. Cada argumento es manejado en secuencia a través de una aplicación funcional. Cada aplicación reemplaza una de las variables ligadas, resultando en una función "parcialmente evaluada" que puede ser aplicada al siguiente argumento. Esto se conoce como *aplicación parcial* (permite definir nuevas funciones).

  En general, una función *f(x,y,z)* puede ser escrita en forma curried como *g x y z*, lo que corresponde a *(((g x) y) z)*. En este caso, g x retorna una función que será aplicada a *y*. La función resultante de esta última aplicación será luego aplicada a *z*.

  fun curry f x y = f(x,y) → El tipo de la función *curry* es: ('a * 'b -> 'c) -> ('a -> ('b -> 'c))

  Ej: fun f x y z = 3 * x + y * z



[VER EJEMPLOS HOJITA!!]

- *Evaluación perezosa*: Una manera simple de aplicar una función es evaluar primero sus argumentos y luego evaluar la función (*reducción de orden aplicativo* – Evaluación ansiosa). Una alternativa es reducir las expresiones yendo de afuera hacia adentro. De este modo, las subexpresiones recién son evaluadas cuando sus valores son necesarios (*reducción de orden normal*).

  La *evaluación perezosa* usa reducción de orden normal. Se evalúan los argumentos de una función sólo cuando es indispensable. Se recuerdan los valores de los argumentos ya calculados, para evitar su re-evaluación (call-by-need).

  + Permite definir estructuras de datos conceptualmente infinitas.
  - La evaluación de los argumentos es más sencilla cuando se realiza antes de la llamada, y por lo tanto la postergación de estas evaluaciones ocasiona un significativo overhead en la evaluación perezosa.
  - Alto costo computacional, la evaluación de orden normal puede evaluar la misma expresión más de una vez

  → Ej: doble 23*45

  Evaluación ansiosa:
  doble 23*45 → doble 1035 → 1035+1035 → 2070

  Evaluación perezosa:
  doble 23*45 →23*45+23*45 → 1035+23*45 → 1035+1035 → 2070

  [ML emplea *evaluación ansiosa*, dado que tiene asociada la semántica *call-by-value* para los argumentos. Sin embargo, presenta otros aspectos de diseño del lenguaje que son "perezosos", como las expresiones "if-then-else" y "handle". Evaluación con cortocircuito (perezosa) de expresiones booleanas mediante "andthen" y "orelse"].

ML

ML (Meta Language) fue creado por Robin Milner en los años '70 (*Standard ML* nació en 1983). Es un lenguaje fuertemente tipado que provee un fuerte soporte para aspectos tales como modularidad, encapsulamiento, tipos de datos abstractos, manejo de excepciones, etc.

Principales Características de ML

- *Soporta el Paradigma Funcional*: Las funciones son valores de primera clase. El principal mecanismo de control de ML es la aplicación funcional.

- *Fuertemente Tipado*: Esta característica garantiza que un programa no incurra en un error de tipos en tiempo de ejecución.

- *Sistema de Tipos Polimórfico*: Contribuye a la flexibilidad del lenguaje.

- *Soporta Tipos de Datos Abstractos*: Nuevas declaraciones de tipos pueden ser definidas por el programador, junto con una familia de funciones para la manipulación de variables de dichos tipos. Los detalles de la implementación quedan ocultos al usuario del tipo.

- *Alcance Estático*: Las referencias a identificadores se resuelven en tiempo de compilación.

- *Mecanismos para el Manejo de Excepciones*: Aumenta la confiabilidad de los programas escritos en ML.

- *Soporta Programación Modular*: En ML un programa es implementado como una colección de "structures" independientes vinculados mediante "functors". La compilación separada de módulos es soportada a través de la importación y exportación de "functors".

Elementos Básicos de ML [Tipos – Identif, Ligaduras y Declarac – Def de Funciones – Def de Tipos – Def de Módulos – Excepciones – Pattern Matching – Polimorfismo Y Sobrecarga]

- *TIPOS PREDEFINIDOS*:
  - *Simples*:
    - *unit*: consiste de un único valor, notado (). Se asemeja al void de C
    - *bool; int*
    - *string*: provee primitiva *size* y ^ para concatenar
    - *real; char*

    Obs: No *se permiten mezclar tipos en las operaciones aritméticas* (Ej: reales sólo se pueden sumar con reales, análogo para enteros). Se proveen las operaciones *real* y *floor* para conversiones explícitas de entero a real y real a entero respectivamente. El operador "~" indica negación unaria (n° negativos).

  - *Estructurados*:

    TUPLAS: representa el producto cartesiano entre elementos, que se especifican entre paréntesis y separados por comas. Dos tuplas son iguales si cada una de sus componentes coincide en valor y posición. Si se comparan dos tuplas de diferente tipo se produce un error de tipos. El acceso a un elemento de la tupla puede realizarse con *pattern matching,* o con #i(t) para acceder al i-ésimo campo de la tupla t (Ej: si t=(2, 3.5, "hola") → #2(t)=3.5).

    LISTAS: El tipo *T list* consiste de una secuencia finita de valores de tipo *T*. Existen dos notaciones:
    - Notación Recursiva: una *T list* está vacía (nil), o contiene un valor de tipo *T* seguido de una *T list* (cabeza::cola , "::" operador *cons*).
    - Forma Comprimida: se listan los elementos separados por comas y encerrados entre corchetes.

    Se provee la primitiva *length*.
    ML asigna a la lista vacía (nil) el tipo *'a list*, donde 'a es una variable cuyos posibles valores pertenecen a una colección de tipos (nil es tratado como un objeto polimorfo que puede ser considerado de diferentes tipos según el contexto).

    → Un tipo que involucra variables de tipos (como 'a) se denomina *politipo*. Una instancia de un politipo se obtiene sustituyendo sus variables de tipos por tipos, incluyendo politipos (Ej: *int list* y *(int * 'b) list* son instancias del politipo *'a list*).
    Un tipo que no involucra variables de tipos se denomina *monotipo*.

    REGISTROS: Implementa el producto cartesiano y consiste en un conjunto finito de campos etiquetados, cada uno de los cuales contiene un valor de algún tipo. Los campos se encuentran encerrados entre llaves, y se separan por comas. Las componentes de un registro se seleccionan por sus campos y no por sus ubicaciones dentro del mismo.

- *IDENTIFICADORES, LIGADURAS Y DECLARACIONES*: En ML todos los identificadores deben ser declarados antes de ser utilizados. El sistema mantiene un ambiente con todas las ligaduras que el programa crea → Los identificadores quedan declarados cuando se ligan a un valor.

  Una variable es *ligada a un valor* mediante el uso de *val*. Ej: val x = 4*5
  val x:int = 20

  Para establecer una ligadura, ML evalúa el lado derecho de la ecuación y asocia dicho valor al identificador de la izquierda. Múltiples identificadores pueden ser ligados simultáneamente, usando la palabra "and" como separador. Cuando se realizan ligaduras múltiples, su evaluación se produce en paralelo. Otra manera de combinar ligaduras es empleando el ";" como separador, pero en este caso la evaluación se produce de izquierda a derecha, teniendo en cuenta para cada evaluación el entorno generado por la evaluación anterior.

  Declaraciones Locales: ML permite efectuar declaraciones locales mediante el constructor *local*:

  local
  <declaración>
  in
  <ambiente donde se usa>
  end;

  Ejemplo:

```
local
        val x=10
in
        val u = x*x+x*x
        val v = 2*x + (x div 5)
end;

val u: int = 200
val v: int = 22
```

  → La ligadura de x es local a las ligaduras de u y v. En este sentido, x está disponible sólo durante la evaluación de las ligaduras de u y v, pero no después. Como consecuencia, únicamente u y v quedarán ligadas.

  También es posible localizar una declaración a una expresión usando la palabra *let*:

  Ejemplo:

```
let
    val x=10
in
        x*x+2*x+1
end;

val it: int = 121
```

  → La declaración de x es local a la expresión que sigue al *in,* y por lo tanto no es visible fuera de ella.

Inferencia de Tipos: ML es un lenguaje fuertemente tipado donde todos los identificadores deben ser declarados antes de ser utilizados. Sin embargo, los tipos de las variables no son (aunque pueden ser) especificados explícitamente en las declaraciones.

Dado que las variables son declaradas mediante ligaduras a valores, ML realiza inferencia de tipos para determinar el tipo de las variables → El tipo de una variable es el tipo del valor al que está ligada. Si el valor es de un tipo simple la asociación es directa, si es compuesto el tipo se determina a partir de los constructores de tipos.

- *DEFINICIÓN DE FUNCIONES*: ML permite definir nuevas funciones mediante la *ligadura funcional*.
  Las funciones siempre tienen un único argumento. ~~Múltiples argumentos se pasan a través de tuplas~~.

  Las funciones son valores y pueden ser empleados arbitrariamente en expresiones. La aplicación de una función tiene la forma *e e'*. Primero se evalúa *e*, obteniéndose una función *f*. luego se evalúa *e'* obteniéndose un valor *v.* Finalmente se aplica *f* a *v* → El resultado de la función es el resultado de evaluar la expresión que la define con el argumento dado.

  Tipo Función: Las funciones son valores, y todos los valores en ML tienen un tipo asociado. El *tipo función* está integrado por funciones. Un valor de este tipo tiene la forma A → B, pronunciado "de A a B", donde A y B son tipos. Una expresión de este tipo devuelve una función que puede ser aplicada a un valor de tipo A, la cual retorna valores de tipo B. Los tipos de A y B son llamados *tipos del dominio y rango* de la función respectivamente.

  Para garantizar que en una aplicación *e e'* la evaluación de *e* retornará una función y no otro valor, basta con asegurar que el tipo de *e* corresponda a un valor del tipo función. De este modo, una aplicación *e e'* es legal si *e* tiene tipo A→B, y *e'* tiene tipo A. Luego el tipo de la expresión completa *e e'* es B.

  Una función es un objeto complejo, cuya estructura es invisible al programador. No admite igualdad (no es posible determinar si dos funciones son iguales).

  Funciones Definidas por el Usuario: Para declarar nuevas funciones el programador establece ligaduras mediante la palabra *fun*, seguida por el nombre de la función y sus argumentos.

  fun <*nombre*> <*argumento*> = <*expresion*>;

  Ejemplo: fun doble x = 2*x;
  val doble: int ->int = fn

  Cuando una función definida por el usuario es aplicada a un valor, este es ligado al argumento generándose un nuevo ambiente en el cual se evaluará el cuerpo de la función.

  Restricciones de Tipo: Cuando se utiliza la *expresión condicional* para la definición de funciones, la cual tiene la forma *if* e *then* e_1 *else* e_2, el tipo del primer argumento (e) debe ser booleano. Además, las expresiones e_1 y e_2 deben ser del mismo tipo. Es importante destacar que la cláusula *else* no es opcional, ya que se trata de una *expresión condicional* y no una sentencia condicional.

Funciones Definidas mediante Cláusulas: Algunas funciones requieren una definición por partes, de acuerdo al valor de su argumento (funciones definidas por casos). Deben estar definidas para todos los valores del dominio. Se resuelve el flujo de control mediante pattern matching.

Funciones con referencias a Variables no Locales: Las funciones pueden declarar variables y funciones locales empleando el constructor *let* visto anteriormente. Sin embargo, pueden referir a cualquier variable visible dentro de su ambiente de referenciamiento (no están restringidas al uso de parámetros y variables locales).

Funciones Polimórficas: Funciones que presentan un comportamiento uniforme sobre un conjunto de tipos. Un politipo es un tipo que contiene variables de tipo.
El tipo de una función polimórfica es un politipo → Polimorfismo paramétrico.

```
Ej: fun invertir(nil) = nil |
    invertir(x::xs) = invertir xs@ [x]

    val invertir: 'a list -> 'a list = fn
```

Funciones Curried: ML permite que el usuario defina *funciones curried*. Son funciones que, por la manera en que fueron definidas, permiten definir nuevas funciones mediante la aplicación parcial.

Ej: Sea la función producto que multiplica dos números

```
fun producto x y = x*y;
val producto: int -> int -> int = fn        [El tipo es: int -> (int -> int)]
```

Luego se define la función porTres

```
fun porTres = producto 3;      ← Aplicación parcial de producto
val porTres: int -> int = fn
```

Ejemplos:

```
MLWorks> fun curry f x y = f(x,y);
  val curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c = fn

MLWorks> fun noCurry(f, x, y) = f(x,y);
  val noCurry : ((('a * 'b) -> 'c) * 'a * 'b) -> 'c = fn

MLWorks> fun por(a, b)=a*b;
  val por : (int * int) -> int = fn

MLWorks> val pp=curry por;
  val pp : int -> int -> int = fn

MLWorks> val pp2=pp 2;
  val pp2 : int -> int = fn

MLWorks> fun porC a b=a*b;
  val porC : int -> int -> int = fn

MLWorks> val ppC=curry porC;
 Line 1: error: Function applied to argument
of wrong type
```

[VER HOJITA!!!]

Formas Funcionales (fcs de alto orden): Las funciones en ML son valores, por lo que pueden ser pasadas como argumentos y devueltas como resultado de otras funciones. Una función que emplea otras funciones como argumentos y/o resultado se denomina *forma funcional*.
Existen en ML varias formas funcionales:

Composición → Permite construir una nueva función a partir de la composición de otras dos.

Ej: fun comp (f, g) x = f(g(x));
val comp: (('a -> 'b) * ('c ->'a)) -> 'c -> 'b = fn

Para que la composición sea válida f y g deben tener tipos compatibles (rango de g = dominio de f).

Map (Aplicar a todos) → Toma como argumento una función *f* y una lista L. El valor retornado es una lista cuya componente y_i es el resultado de aplicar *f* al i-ésimo elemento de L (Es una función curried).

La función *simpleMap* puede definirse:

```
MLWorks> fun simpleMap(F, nil) = nil
       | simpleMap(F, x::xs) = F(x)::simpleMap(F, xs);
val simpleMap : (('a -> 'b) * 'a list) -> 'b list = fn
```

Ej – función map:
```
MLWorks> map length [[1], [2,4,5], []];
val it : int list = [1, 3, 0]
```

Operador de Inserción → El operador de inserción a derecha permite generalizar una función binaria a n-aria. Esta función puede definirse como:

```
MLWorks> fun foldr(f,s,nil) = s
       | foldr(f, s, (cabeza::cola)) = f(cabeza, foldr(f, s ,cola));
val foldr : ((('a * 'b) -> 'b) * 'b * 'a list) -> 'b = fn
```

← Para aplicar la función f (binaria) a una lista, se la aplica a la cabeza y al resultado de la aplicación generalizada (foldr) sobre la cola; donde s es el elemento neutro de la operación

Ej: Se puede sumar una lista de enteros como sigue:

```
MLWorks> foldr (op +, 0, [1, 2, 3, 4]);
val it : int = 10
```

Análogamente puede definirse el operador de inserción a izquierda *foldl*.

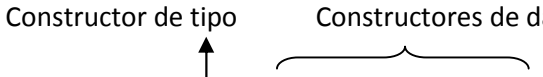
- *DEFINICIÓN DE TIPOS*: El sistema de tipos en ML es extensible. El programador puede definir nuevos tipos usando alguna de las siguientes formas: *abreviación de tipo*, *definición de tipo de dato*, *definición de tipo de dato abstracto*.

  Abreviación de Tipo (type): Es la forma más elemental para definir un tipo. Se liga una expresión de tipo a un identificador de tipo.

  Ej:
  ```
  MLWorks> type parenteros = int*int;
  eqtype parenteros = (int*int)

  MLWorks> type 'a par = 'a * 'a;                ← Es un politipo
  eqtype 'a par = ('a * 'a)

  MLWorks> type parbooleanos = bool par;         ← Es un subtipo del politipo 'a par
  eqtype parbooleanos = (bool * bool)
  ```

  Definición de Tipo (datatype): Se define tanto el nombre (y quizás algunos parámetros de tipo), que es el constructor de tipo que permitirá construir otros tipos (posiblemente polimórficos), y un conjunto de constructores de valores del tipo (constructores de datos, para construir los valores de las variables de ese tipo).

  Ej:


  ```
  MLWorks> datatype = color = Rojo | Azul | Amarillo;
  datatype color =
    Blue |
    Red |
    Yellow
  val Blue : color
  val Red : color
  val Yellow : color
  ```

  → *Constructor de tipos con parámetros*: Los parámetros de un constructor de tipos son variables de tipos, que se usan para los constructores de datos (constructor *of* tipo). El parámetro del constructor de tipos se escribe antes del nombre del constructor de tipos.

  Ej:
  ```
  MLWorks> datatype 'a opcion = NADA | ALGO of 'a;
  datatype 'a opcion =
    ALGO of 'a |
    NADA
  val ALGO : 'a -> 'a opcion
  val NADA : 'a opcion
  ```

  →En este caso el resultado es un nuevo tipo polimórfico. Obtenemos el tipo *'a opcion*, y podemos crear el tipo *int opción*, así como existe el tipo *'a list* y podemos crear el tipo *int list*.

  ```
  MLWorks> val nada = NADA;
  val nada : 'a opcion = NADA

  MLWorks> val alg = ALGO "hola";
  val alg : string opcion = ALGO "hola"
  ```

Ej:

```
MLWorks> datatype 'a grupo = uno of 'a | varios of 'a list;
datatype 'a grupo =
  uno of 'a |
  varios of 'a list
val uno : 'a -> 'a grupo
val varios : 'a list -> 'a grupo

MLWorks> val u = uno 2;
val u : int grupo = uno 2

MLWorks> val h = varios[1,2,43];
val h : int grupo = varios [1, 2, 43]
```

→ Una variable de tipo *'a grupo* contiene: un elemento de tipo 'a, o una lista de elementos de tipo 'a. Es un tipo polimórfico

Ej:

```
MLWorks> datatype dinero = sindinero | centavos of int | pesos of int | cheque of string*int;
datatype dinero =
  centavos of int |
  cheque of (string * int) |
  pesos of int |
  sindinero
val centavos : int -> dinero
val cheque : (string * int) -> dinero
val pesos : int -> dinero
val sindinero : dinero

MLWorks> val plata = centavos 34;
val plata : dinero = centavos 34

MLWorks> val plata = pesos (100);
val plata : dinero = pesos 100

MLWorks> val check = cheque ("banco nacion", 134);
val check : dinero = cheque ("banco nacion", 134)
```

← En este caso el tipo no es polimórfico, ya que se indicaron los tipos int y string para los constructores de datos

El uso de paréntesis para indicar el parámetro cuando es opcional, cuando se tiene un solo parámetro

Se pueden usar los constructores de datos como patrones en la definición de funciones:

```
MLWorks>  fun contarcentavos (sindinero) = 0
        |  contarcentavos (centavos(cant)) = cant
        |  contarcentavos (pesos(cant)) = 100*cant
        |  contarcentavos (cheque(banco,cant)) = 100*cant;
val contarcentavos : dinero -> int = fn
```

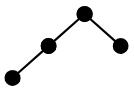
→ *Constructor de tipos Recursivo*: Los tipos de datos también pueden ser recursivos.

Ej: Se puede definir un tipo arbolbinario como sigue:

```
MLWorks> datatype arbolbinario = vacio | hoja | nodo of arbolbinario*arbolbinario;
datatype arbolbinario =
  hoja |
  nodo of (arbolbinario * arbolbinario) |
  vacio
val hoja : arbolbinario
val nodo : (arbolbinario * arbolbinario) -> arbolbinario
val vacio : arbolbinario
```

← Permite representar la estructura de árboles binarios, cuyos elementos son árboles con estructura nodo(HI,HD), hojas, o árboles vacíos.

```
MLWorks> val arbol = nodo(nodo(hoja,vacio), hoja);
val arbol : arbolbinario = nodo (nodo (hoja, vacio), hoja)
```



También se pueden especificar *tipos de datos recursivos paramétricos*. Un ejemplo es el tipo predefinido *T list*, el cual corresponde a la siguiente definición:

```
datatype 'a list = nil | :: of 'a * 'a list;
```

→ Sin embargo, si el programador ingresa está definición, ML Works retornará un error de compilación debido a que el operador "::" está reservado.

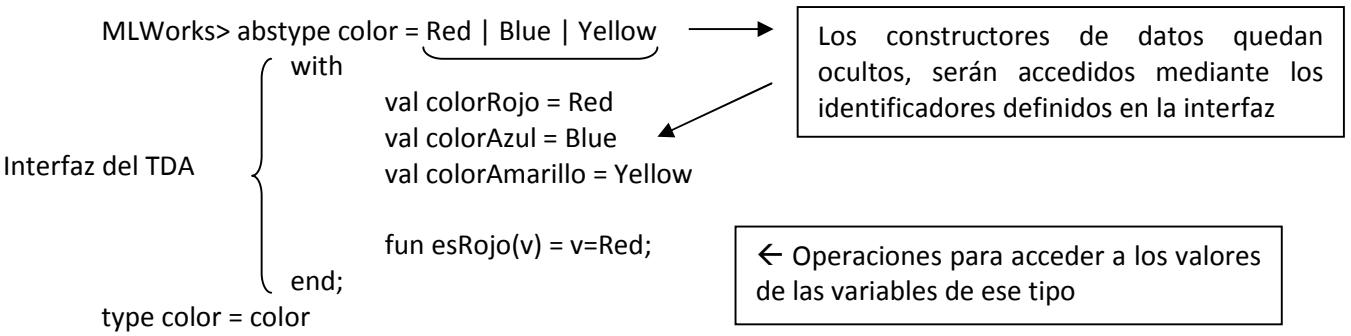
```
MLWorks>datatype 'elem lista = NIL | CONS of 'elem * 'elem lista;
datatype 'a lista =
  CONS of ('a * 'a lista) |
  NIL
val CONS : ('a * 'a lista) -> 'a lista
val NIL : 'a lista
```

→ Constructor de tipos recursivo paramétrico con casi el mismo poder expresivo que el tipo *list* predefinido

```
MLWorks> fun longLista NIL = 0 | longLista CONS(_,Cola) = 1 + longLista(Cola);
val longLista : 'a lista -> int = fn

MLWorks> val ListaEnteros = CONS(1, CONS(2, CONS(3,NIL)));
val ListaEnteros : int lista = CONS (1, CONS (2, CONS (3, NIL)))

MLWorks> val listaStrings = CONS("primero", CONS("segundo",NIL));
val listaStrings : string lista = CONS ("primero", CONS ("segundo", NIL))

MLWorks> val L = longLista NIL;
val L : int = 0

MLWorks> val L2 = longLista (CONS(1, NIL));
val L2 : int = 1
```

Definición de Tipo de Dato Abstracto (abstype): Un TDA está formado por un tipo de dato y un conjunto de funciones que permiten manipular valores de dicho tipo. La principal característica de los TDA es que permiten definir nuevos tipos ocultando su representación interna.

Ej:




```
type color = color
val colorAmarillo : color = _
val colorAzul : color = _
val colorRojo : color = _
val esRojo : color -> bool = fn

MLWorks> val sangre = colorRojo;
val sangre : color = _

MLWorks> esRojo(sangre);
val it : bool = true
```

La representación interna de un valor de tipo *color* no es visible fuera del TDA. Únicamente se puede acceder a la interfaz del TDA (bloque comprendido entre el *with* y el *end*). A diferencia de los tipos de datos (datatype), los constructores de datos *Red*, *Blue*, *Yellow* quedan ocultos. Un programa de usuario del TDA sólo puede definir una nueva variable de tipo color y ligarle algunos de los valores definidos en la interfaz (*colorRojo, colorAzul, colorAmarillo*).

- *DEFINICIÓN DE MÓDULOS*: ML se vale de dos constructores para la definición de módulos: las "*structures*" permiten combinar declaraciones de tipos, valores y otras structures relacionadas; las "*signatures*" especifican una clase de structures mediante el listado de los nombres y tipos de cada componente → Las signatures y structures de ML guardan similitud con las partes de definición e implementación de módulos de Modula2.
  (También se proveen los *"functors*" que son structures parametrizados con otras sutructures).

  Structures: Las distintas declaraciones efectuadas dentro del módulo son encerradas entre las palabras reservadas *struct* y *end*. En toda parte del programa donde una *structure* sea visible, sus componentes pueden ser accedidos mediante una notación similar a la de los registros de Pascal (acceso mediante "."). Dentro del cuerpo de la structure las componentes son conocidas por sus identificadores.
  → Es importante notar que la representación interna de los tipos definidos en una structure, no está oculta dentro del módulo.

Signatures: Una *signature* especifica la información que ML necesita para integrar las distintas unidades que conforman un programa de manera segura → Es una descripción de las componentes de una structure.
Una vez que se efectúa la declaración de una *structure*, ML responde mostrando su correspondiente *signature*, cuyo cuerpo está encerrado entre los identificadores *struct* y *end*. La misma sólo muestra los nombres y tipos de las componentes definidas en la structure, junto con los identificadores correspondientes a los tipos declarados.

→ Dado que todo lo definido dentro de la structure es visible al programador, es posible definir una nueva signature para el módulo, con el motivo de ocultar definiciones de tipo, operaciones, etc.

Ej: Si queremos definir una *structure* para representar números complejos:

```
structure Complejo =
 struct
  type t = real*real;
  val cero = (0.0, 0.0);
  fun suma ((x, y),  (x', y')) = (x+x', y+y') : t;
  fun resta ((x, y), (x', y')) = (x-x', y-y') : t;
  fun producto ((x, y), (x', y')) = (x*x' – y*y', x*y' + y*x') : t;
  fun reciproco (x, y) = let
                              val p = x*x + y*y
                         in
                              (x/p, ~y/p) : t
                         end;
  fun division (z, z') = producto (z, reciproco z');
 end;
```

ML responde mostrando la *signature* de Complejo:

```
structure Complejo =
 struct
  type t = (real * real)
  val cero : (real * real) = (0.0, 0.0)
  val division : ((real * real) * (real * real)) -> (real * real) = fn
  val producto : ((real * real) * (real * real)) -> (real * real) = fn
  val reciproco : (real * real) -> (real * real) = fn
  val resta : ((real * real) * (real * real)) -> (real * real) = fn
  val suma : ((real * real) * (real * real)) -> (real * real) = fn
 end
```

Si se quisiera ocultar la operación *reciproco* podría definirse una nueva signature:

```
signature ARITMETICA =
sig
  type t;
  val cero : t;
  val resta : t * t  -> t;
  val division : t * t  -> t;
  val producto : t * t  -> t;
  val suma : t * t  -> t;
end ;
```

← La signature inferida por ML muestra el tipo de los argumentos como real * real, mientras que ARITMETICA sólo emplea el tipo t

Esta signature puede ser asociada a la structure *Complejo*en el encabezamiento de la structure:

```
structure Complejo: ARITMETICA =
  struct
                …
  end;
```

Alternativamente, se puede asociar una structure con una signatura diferente a la sugerida por ML de la siguiente manera:

```
structure NewComplejo: ARITMETICA = Complejo;
```

← Una misma signature puede estar asociada a más de una structure. *NewComplejo* es la misma structure que *Complejo*, pero con diferente signature

- *EXCEPCIONES*: Muchas funciones son parciales, es decir, no devuelven un valor para alguno de los posibles argumentos del tipo del dominio de la función. Es esenciar poder capturar tales errores. Si ocurre una excepción y no es manejada, causa la terminación del programa con un mensaje de "uncaught exception".

  La excepción de división por cero de enteros (Div) será lanzada si se intenta efectuar: 5 div 0.

  La aritmética real en ML evita la generación de excepciones, usando dos constantes especiales: *inf* para la división por cero, *nan* para los "not a number":

  n/0 → inf
  -n/0 → ~inf
  inf/inf → nan

  Si se aplica el operador predefinido *chr* a un número entero fuera del rango 0-255 se provocará una excepción (Chr).

Excepciones definidas por el Usuario: Es posible querer definir nuevas excepciones y lanzarlas cuando se encuentran condiciones excepcionales en el código.

- En ML existe un tipo predefinido, denominado *exn*, cuyos valores se conocen como *valores de excepción*. Este tipo se asemeja a los tipos de datos (datatype), pero a diferencia de estos, nuevos constructores pueden ser sumados al mismo mediante la declaración:

  exception <nombre>

- Adicionalmente, pueden definirse *excepciones con parámetros*:

  exception <nombre> *of* <tipo>

  El nombre es un constructor de excepción, y al lanzar la excepción debe indicarse un argumento del tipo especificado en la declaración (no es posible lanzarla sin el argumento).

  Esta capacidad de definir excepciones como funciones brinda la posibilidad de que el manejador reciba, a través de los argumentos de la excepción, información que le permita decidir la secuencia de acciones a realizar para recuperarse de la falla.

- Es posible *definir más de una excepción a la vez*, separando la lista de excepciones mediante la palabra reservada *and*: exception <e1> and <e2>

- *Activación de excepciones*: Una excepción puede ser activada mediante la expresión *raise* Ex.

  Ej: Sea la función *cociente* que calcula la división entre dos números enteros (a y b), y la excepción DivisionPorCero que se asume declarada:

  ```
  fun cociente (a,b)= if b=0 then raise DivisionPorCero
                      else a/b;
  ```

  El comportamiento normal de *cociente* es devolver el cociente a/b, pero si B=0 se activa la excepción correspondiente → Esta situación viola el principio de ML de que "*una función invariablemente debe retornar un resultado apropiado (un valor dentro del rango de su tipo)*". Las excepciones son la única violación a este principio.

Manejo de Excepciones: Lanzar una excepción que no se captura, causa la terminación del programa. Es preferible que cuando se lanza una excepción, se intente producir un valor apropiado y continuar la computación (El manejador captura la excepción y retorna un valor dentro del rango del tipo de la función).

La ligadura entre una excepción y su manejador es dinámica. Si *f* invoca a *g*, *g* invoca a *h* y *h* activa una excepción *Ex*, se buscará un manejador para *Ex* siguiendo la cadena *h*, *g*, *f*. El primer manejador encontrado captura la excepción.

Básicamente un manejador (*handler*) puede definirse de la siguiente forma (como una especie de *case*):

$$E \text{ handle } P_1 \Rightarrow E_1 \mid P_2 \Rightarrow E_2 \mid \ldots \mid P_n \Rightarrow E_n$$

Donde la expresión E es una expresión que puede causar una expresión. Los patrones de matching P_i son excepciones, y son asociados con expresiones E_i del mismo tipo que E (pueden incluir mensajes de error).

- Si E produce un valor v y no lanza una excepción, entonces el matching no se aplica a v, y v es el resultado de la expresión *handle*.
- Si E lanza una excepción (puede tener parámetros), entonces se aplica el matching. El primer patrón que matchea con la excepción provoca la evaluación de su expresión asociada, la cual corresponde al valor de la expresión *handle*. Si ninguno de los patrones matchea, entonces la excepción no es capturada en este punto. La excepción puede ser manejada por otro handler, o puede permanecer no capturada, en cuyo caso al propagarse al nivel principal causará la finalización de la computación.

Ej: Dada la definición de la función *cociente* que activa una excepción de división por cero, podemos definir la siguiente función calcularCociente, que posee un manejador:

```
fun calcularCociente(a,b) = cociente(a,b) handle DivisionPorCero => 0;
```

Ej: Definición de una función que calcula la combinatoria de dos números enteros, con un manejador:

```
MLWorks> exception OutOfRange of int*int;
exception OutOfRange of (int * int)

MLWorks> fun comb1(n,m) = if n<=0 then raise OutOfRange(n,m)
                          else if m<0 orelse m>n then raise OutOfRange(n,m)
                          else if m=0 orelse m=n then 1
                          else comb1(n-1,m) + comb1(n-1,m-1);
val comb1 : (int * int) -> int = fn

MLWorks> fun comb(n,m)= comb1(n,m) handle
OutOfRange(0,0) => 1
|OutOfRange(n,m) => (print("out of range: n="); print(Int.toString(n)); print(" m=");
print(Int.toString(m)); print("\n");  0);

MLWorks> comb(4,2);
val it:int =6

MLWorks> comb(0,0);
val it:int =1

MLWorks> comb(3,4);
out of range: n=3 m=4
val it:int =0
```

- *PATTERN MATCHING*: Muchas veces, es de interés obtener un componente de un tipo de datos estructurado. Para ello, existe un mecanismo denominado *pattern matching* que permite descomponer valores compuestos. La técnica consiste en aparear componentes con identificadores no ligados.

  Ej:

```
MLWorks> val tup1 = ("jorge", 20, true);
val tup1 : (string * int * bool) = ("jorge", 20, true)

MLWorks> val (nombre,edad,soltero) = tup1;                    (*)
val edad : int = 20
val nombre : string = "jorge"
val soltero : bool = true
```

  La parte izquierda de la ligadura (*) es un patrón (pattern), que es construido usando variables, constantes y constructores de tipos → En definitiva, un *pattern* es una expresión especial que puede contener variables no ligadas previamente.

  Por otra parte, puede ocurrir que sólo sea de interés recuperar información en forma parcial (algunas cosas). En tal caso, ML brinda un *wilcard pattern* (_)que permite realizar el matching sin crear una ligadura.

  Ej:
```
MLWorks> val (nombre,_,_) = tup1;
val nombre : string = "jorge"
```

  El pattern matching también puede efectuarse sobre *registros* a través de los nombres de sus campos.

  Ej:
```
MLWorks> val reg1 = {nombre="jorge", edad=20, soltero=true};
val reg1 : {edad: int, nombre: string, soltero: bool} = {edad=20, nombre="jorge", soltero=true}

MLWorks> val {nombre=n, edad=e, soltero=s} = reg1;
val e : int = 20
val n : string = "jorge"
val s : bool = true
```

  Aveces puede resultar conveniente utilizar un *record wilcard* (…), para seleccionar sólo algunos campos del registro. Existe una restricción para el uso del record wilcard: debe ser posible determinar en tiempo de compilación el tipo del pattern record completo → Debe ser factible la inferencia de todos los campos y sus tipos, a partir del contexto.

  Ej:
```
MLWorks> val {edad=age,...} = reg1;
val age : int = 20
```

  Una última forma de pattern matching soportada por ML es el denominado *layered pattern* (*as*), el cual permite anidar patrones.

  Ej:
```
MLWorks> val x = (("casa",true),14);
val x : ((string * bool) * int) = (("casa", true), 14)

MLWorks> val (tup as (tizq,tder),der) = x;
val der : int = 14
val tder : bool = true
val tizq : string = "casa"
val tup : (string * bool) = ("casa", true)
```

  [***Obs*:** Cabe destacar que los pattern poseen una significativa limitación: deben ser lineales (una variable no puede aparecer más de una vez en el pattern – Ej: no es posible (x,x) para identificar partes simétricas)]

- <u>*POLIMORFISMO Y SOBRECARGA*</u>: Las *funciones polimórficas* son aquellas que trabajan sobre una clase de tipos de manera uniforme. En ML el tipo de una función polimorfa es siempre un *politipo*. Esta clase de polimorfismo se denomina *polimorfismo paramétrico*, dado que el tipo está parametrizado por las variables de tipo. El concepto de *sobrecarga* es una noción de *polimorfismo ad-hoc*, dado que se mira más el nombre de la función que su estructura de definición.

  → En el caso de una función sobrecargada, se tienen dos funciones con el mismo nombre. En el caso de una función polimórfica, se tiene una sola función que puede ser aplicada sobre diferentes tipos.

  En el caso de las funciones aritméticas (Ej: +), si no se define el tipo de los argumentos, ML infiere que el tipo de los argumentos es entero (tipo numérico default).

  Las funciones aritméticas predefinidas para los números reales son ~,+,- y*. Es importante aclarar que *no se pueden mezclar tipos en estas operaciones* (Ej: se pueden sumar enteros y reales, pero no un entero y un real). Ej: 2 + 3.5 es una expresión débilmente tipada ← Generará un error de tipo (el segundo argumento debería ser del tipo del primero).

  ML no soporta coerción, pero provee operaciones para realizar *conversiones explícitas* de real a entero (floor) y de entero a real (real).

Paradigma Orientado a Objetos – Smalltalk

PARADIGMA ORIENTADO A OBJETOS – Conceptos Básicos

- *Abstracción de Datos* → Reusabilidad y mantenimiento. Extensibilidad
- *Polimorfismo* → Flexibilidad
- *Técnicas de clasificación para capturar y factorizar aspectos comunes de las entidades* (objetos) → Herencia

Objetos: Entidades con identidad. Los *atributos* determinan su estado interno, y las *operaciones* especifican su comportamiento. La implementación de las actividades de los objetos determina las acciones a seguir ante un requerimiento (métodos).

La *comunicación entre objetos* es a través del *envío y recepción de mensajes*. Los objetos procesan los mensajes escondiendo el conocimiento y la forma en que realizan el procesamiento. Los manejan de acuerdo a las habilidades nativas de su clase, y también hacen uso de las habilidades de las clases antecesoras a su clase.

Protocolo: Conjunto de métodos que controlan el acceso al estado interno del objeto

Clase: Patrón para la creación de objetos con un protocolo y una implementación específica (un objeto es una instancia de una clase).

Características de un Sistema OO

- Los objetos están caracterizados por atributos y tienen un comportamiento de acuerdo al cual responden a los mensajes.
- Los objetos están clasificados de alguna manera
- Los objetos pueden enviar y recibir mensajes dinámicamente.
- Los objetos procesan los mensajes escondiendo el conocimiento y la forma en que realizan ese procesamiento.
- Datos y acciones reciben un tratamiento balanceado (A diferencia del paradigma imperativo)

Paradigma OO Puro (Eiffel, Smalltalk)

- Toda entidad es un objeto
- Toda la computación se realiza a través de mensajes
- Polimorfismo por inclusión
- No hay restricciones para ligar variables a objetos ni para redefinir métodos o definir nuevos
- Ligadura dinámica (de mensajes a métodos)
- Variables no tipadas
- Herencia simple por implementación
- La relación entre clases puede no ser de especialización

Paradigma Híbrido (C++, Object Pascal)

- No todas las entidades son objetos
- Soporte simultáneo de otros paradigmas además del OO (imperativo)
- No siempre hay un segundo nivel de clasificación
- Polimorfismo resringido, por inclusión, ad hoc
- Ligadura dinámica y estática
- Chequeo estático
- Herencia múltiple, por implementación y por interfaz

Paradigma Discriminado (Java, C#)

- Separa el tratamiento de tipos predefinidos
- Polimorfismo restringido y por inclusión
- Ligadura dinámica
- Sistema de tipos fuerte (LP fuertemente tipados)
- Herencia simple y por implementación

Lenguajes Orientados a Objetos – Requerimientos Mínimos

- *Oscurecimiento de la Información*: El estado interno de un objeto sólo puede ser accedido y modificado como respuesta a un mensaje. La representación física de cada clase y la implementación de las operaciones no es visible desde el exterior, y por lo tanto puede ser modificada sin alterar el resto de las clases.

- *Abstracción de Datos*: Una clase define un patrón de comportamiento a partir del cual es posible crear varias instancias.

- *Clasificación (Herencia - Relación)*: Las propiedades de las clases pueden ser factorizadas para definir otras más generales (generalización), o a la inversa: a partir de una clase es posible definir otras más específicas (especialización).
  La *clasificación de primer orden* agrupa objetos (clases). La *clasificación de segundo orden* agrupa clases (Herencia: clasifica las clases en una estructura de árbol o grafo).

  [Los objetos pueden clasificarse según sus propósitos, habilidades y actividades. En el diseño de una aplicación los objetos se clasifican de acuerdo, fundamentalmente, a su comportamiento. La clasificación es entonces una parte esencial del paradigma porque permite organizar el conocimiento. La clasificación es la clave para la reusabilidad y extensibilidad.
  Definición de TDA requiere que el lenguaje brinde alguna forma de oscurecimiento de información. La herencia puede pensarse como una forma de polimorfismo que requiere además de algún mecanismo de abstracción de datos]

- *Polimorfismo*: Se refiere a la capacidad de que objetos de diferentes clases respondan al mismo protocolo.

Algunos exigen también: - Ligadura Dinámica (de mensajes a métodos)
- Concurrencia

HERENCIA

- SIMPLE (Smalltalk, Ada, Java): Cada clase tiene una clase padre (salvo Object)
- MÚLTIPLE (Eiffel, C++, Nuevas versiones de Smalltalk): Permite que una clase se derive de más de una clase base
- PARCIAL (Eiffel)

- CONCRETA
- ABSTRACTA

- DE INTERFAZ (Eiffel): Una clase derivada puede acceder a la clase base como el resto de los clientes, a través de la interfaz pública
- DE IMPLEMENTACIÓN (Smalltalk): Permite que las clases derivadas de una clase base accedan a su implementación interna/parte privada (más eficiente pero menos segura)

HERENCIA SIMPLE

- ESPECIALIZACIÓN/GENERALIZACIÓN: "Es un"
- AGREGACIÓN: "Es parte de" (débil)
- COMPOSICIÓN: "Es parte de" (fuerte)

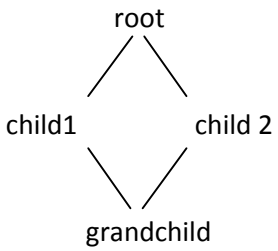
HERENCIA MÚLTIPLE

- COMPOSICIÓN: "Tiene un"

Problemas de contar con Herencia Múltiple:

→ *Colisión de Nombres*: Una clase derivada hereda un atributo/operación con el mismo nombre de dos clases padre. La clase derivada debe ser capaz de referenciar a cada uno de ellos para poder exportar los miembros deseados.

→ *Herencia Diamante*: Ocurre cuando más de una clase padre (de la clase actual) se deriva de la misma clase base. Ej: Los miembros (atributos, métodos) de la clase *root* van a estar repetidos en la clase *grandchild*:



[Ver Herencia!! – Pág 88]

SMALLTALK

Smalltalk surgió con la idea de diseñar un ambiente de programación que integrara SW y HW, que transformara a la computadora en una herramienta fácil de usar. El sistema Smalltalk puede ser pensado como compuesto por cuatro elementos básicos:

- *El kernel del lenguaje de programación*: es el compilador o intérprete del lenguaje, que determina la semántica y la sintaxis.

- *El paradigma de programación*: estilo de uso del kernel, es decir, una vista del significado asignado a las entidades del mismo.

- *El sistema de programación*: es el conjunto de objetos y clases del sistema necesarias para llevar a cabo la programación; este sistema brinda la estructura fundamental y resume en una colección jerárquica de clases la concepción de la programación en Smalltalk.

- *El modelo de interface con el usuario*: la interface gráfica del usuario representa la utilización de las herramientas incorporadas para presentar el sistema al usuario.

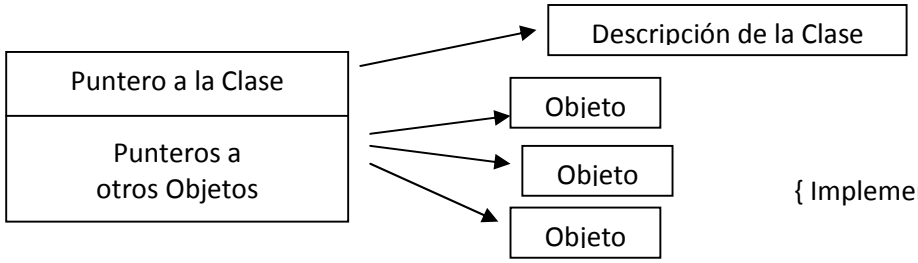
Estos componentes son básicamente jerárquicos: la interface del usuario se implementa sobre el sistema de programación, el cual a su vez está construido siguiendo el paradigma de programación y usando el kernel. La combinación de todas estas piezas conforma el denominado sistema Smalltalk.

Características de Smalltalk

- Toda entidad es un objeto (una clase es un objeto). } Paradigma Puro
- Toda la computación se realiza intercambiando mensajes. }
- Ligadura dinámica del mensaje con el método.
- No hay restricciones para ligar variables a objetos ni para redefinir métodos (si se redefine con la misma signatura esconde al anterior) o definir nuevos métodos.
- Las variables no son tipadas.
- Herencia simple, por implementación.

Elementos de Smalltalk

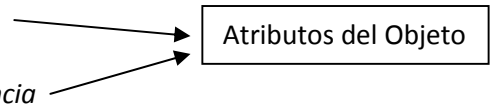
- *OBJETOS Y VARIABLES*: En Smalltalk, cada cosa es un *objeto*. No hay manera de crear una entidad dentro del lenguaje que no sea un objeto. Desde el punto de vista de la *implementación*, los objetos son sólo un conjunto de celdas de memoria que se asignan dinámicamente. Estas celdas mantienen el estado del objeto, y a diferencia de otros lenguajes, las *variables* no almacenan el valor que representan, sino un puntero al objeto (*Semántica de Punteros*):



{ Implementación de un Objeto }

Existen distintas categorías de *identificadores*:

- *Nombre de Clase*
- *Variables de Clase*
- *Variables Globales*
- *Variables de Instancia*
- *Variables Temporales*
- *Pseudo Variables*
- *Mensajes*



Las *variables de clase* guardan atributos comunes a todos los objetos que son instancias de una misma clase. Por lo tanto, son accesibles a la clase y a sus subclases, así como también a todos los objetos que son instancia de dichas clases.

Las *variables de instancia* tienen valores asociados únicamente con cada instancia u objeto creado a partir de una clase. La única manera de acceder y modificar dichos objetos es a través de los métodos pertenecientes al protocolo del objeto.

Las *variables globales* o compartidas son definidas en diccionarios llamados "pools". Diferentes tipos de variables globales son definidas en diferentes pools. El nombre de la variable y su valor son ligados dentro de un objeto que es instancia de la clase *Association*, que se ubica dentro de un pool (Obs: las variables de clase son un caso particular de las globales, y son implícitamente almacenadas dentro de un pool definido para la clase).

Las *variables temporales* están asociadas a un fragmento de código. Se crean cada vez que se ejecuta dicho código, y el espacio que ocupan es desalocado cada vez que finaliza la ejecución del fragmento. Sintácticamente, son declaradas entre barras "|", al principio del código al que están vinculadas (en general, son utilizadas por los métodos).

Además de las variables de instancia y de clase cada objeto tiene asignada una *pseudo variable* denominada *self*, la cual es un designador de objeto que representa al objeto receptor del mensaje (this).

El *nombre* de aquellas variables cuyo alcance sea mayor al de un método u objeto, debe comenzar con *mayúscula* (Nombre de Clase, Variables de Clase, Variables Globales). El nombre de las variables cuyo alcance sea el método (variable local/temporal) o el objeto (variable de instancia) debe comenzar con *minúscula*. [Los nombres de Pseudo Variables y Mensajes también deben comenzar con minúscula]

En cuanto a la *implementación*, las variables en Smalltalk recuerdan a los campos de un registro en lenguajes como Pascal, pero con algunas diferencias:

- En las variables de instancia, el ámbito de referencia está delimitado por el objeto donde se encuentra la definición.

- Las variables son sencillamente designadores de objetos. Son uniformes con respecto al espacio ocupado, más allá del objeto referenciado, y en este sentido, no tienen un tipo asociado.

- *MENSAJES*: La computación tiene lugar como resultado de enviar *mensajes* a los objetos. La sintaxis indica que un mensaje se compone de una referencia al objeto *receptor* seguido por un indicador del mensaje en cuestión, denominado *selector* del mensaje, seguido de cero o más *argumentos*. La semántica asociada a los mensajes puede recordar a los procedimientos, pero estable ce otro orden entre los componentes del lenguaje [El objeto receptor de un mensaje debe contar con el método (selector) definido en su protocolo]. Existen tres tipos de mensajes (indicados en orden de precedencia):

  - *Unarios*: no tienen argumentos, y deben comenzar con una letra minúscula.

    Ej: pilaUno desapilar

    ← *pilaUno* es el objeto receptor, y *desapilar* es el selector del mensaje

  - *Binarios*: Permiten pasar un argumento, y su selector está formado por uno o dos caracteres no alfabéticos adyacentes. En general, se emplean para operaciones aritméticas y relacionales.

    Ej: x < y

    ← *x* es el objeto receptor, < es el selector del mensaje, e *y* es el argumento. En respuesta a este mensaje, *x* devolverá el objeto *true* o el objeto *false*

  - *Palabra Clave*: Los mensajes de palabra clave permiten pasar uno o más argumentos. Cada argumento tiene asociada una palabra clave, la cual debe comenzar con una letra minúscula.

    Ej: pilaUno apilar: 80

    ← Se le pide al objeto *pilaUno* que apile el objeto *80*

    Ej: arregloA at:3 put: 19

    ← En este caso *arregloA* es un objeto instancia de la clase Array, y el mensaje *at: put:* está indicando al objeto *arregloA* que asigne el objeto *19* al objeto que se encuentra en la posición 3 de su aarreglo

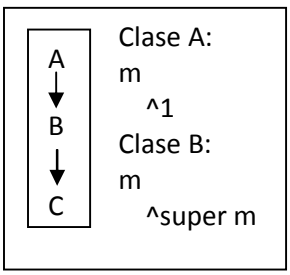
  → Un objeto se puede enviar un objeto a sí mismo mediante el uso de la pseudo variable *self*.

  Ej:
  ```
  apilar: unElem
   (self estaLlena)
    ifTrue:  [self error: 'pila llena']
    ifFalse: [tope := tope + 1.
              elementos at: tope put: unElem]
  ```

  ← En este caso, *self estaLlena* es un mensaje que pilaUno se envía a sí misma para saber si se encuentra llena (esto es, la búsqueda del selector error: comenzará en la clase de la cual pilaUno es intancia)

  → El uso de la pseudo variable *super* obliga al sistema a comenzar la búsqueda del mensaje en la *superclase de la clase que contiene el método en el cual aparece super* (OJO! NO en la superclase de la clase de la cual es instancia el objeto receptor).

  Ej:


  ```
  Clase A:
  m
    ^1
  Clase B:
  m
    ^super m
  ```

  c:= C new. c m

  ← Si bien c es instancia de la clase C, la búsqueda del mensaje *m* comienza en A (superclase de B, en la cual está el *super*)

Mensajes dentro de mensajes: Los mensajes pueden combinarse en forma poderosa. Al ser combinados, los mensajes unarios se evalúan en primer término, luego los binarios, y finalmente los de palabra clave.

Ej: 'casa' size + #(1 2 3 4) size

Arreglo

← Se evalúan primero los mensajes *size* que son unarios, y luego el mensaje +. El resultado de este mensaje es el objeto 8

Expresiones: Los números son objetos. En Smalltalk la evaluación de expresiones se realiza de izquierda a derecha. Además todos los operadores binarios aritméticos y lógicos tienen la misma precedencia, y su argumento se evalúa con "evaluación ansiosa".

Ej: x:= 3+4

← Asigna al objeto x el resultado de enviar el mensaje + al objeto 3 pasando como argumento el objeto 4. En definitiva, x queda asociado al objeto 7

Expresiones en Cascada: Existen situaciones en que se necesita enviar varios mensajes al mismo objeto en forma consecutiva. Para estos casos, Smalltalk provee un tipo especial de expresiones conocidas como *expresiones en cascada*, las cuales permiten enviar múltiples mensajes a un mismo objeto.

Ej: pilaUno:= Pila new; apilar:1; apilar:2

← La *secuencia* de mensajes equivalente sería: pilaUno:= Pila new. pilaUno apilar:1. pilaUno apilar:2.

Expresiones Lógicas: El objeto *true* es instancia de la clase *True* y el objeto *false* es instancia de la clase *False*. Los operadores definidos para las clases *True* y *False* son:

- *Operador Unario* → not
- *Operadores Binarios* → &, |
- *De Palabra Clave* → and:, or:

← Los mensajes de palabra clave *and:* y *or:* se pueden ver como si se implementaran con evaluación perezosa, debido a la ligadura dinámica de los mensajes a los métodos (tienen implementaciones distintas en las clases True y False – Métodos sobrecargados)

La diferencia entre los operadores binarios y los de palabra clave, es que los primeros deben recibir un objeto de clase True o False como argumento mientras que los últimos pueden recibir un bloque por argumento, el cual es o no evaluado dependiendo de la implementación del método (según corresponda a la clase True o False).

Clase: True

and: bloque	*& objeto*	*not*
^bloque value	^objeto	^true
or: bloque	*| objeto*	
^self	^self	

Clase: False

and: bloque	*& objeto*	*not*
^self	^self	^false
or: bloque	*| objeto*	
^bloque value	^self	

- *MÉTODOS*: Los objetos responden a los mensajes mediante la ejecución de *métodos*. Un método se compone de dos partes: un *encabezado* (para identificar al método cuando sea invocado) y un *cuerpo* (implementa la operación correspondiente).

  Ej:

  ```
  apilar: unElem
  (self estaLlena)
  ifTrue: [self error: 'pila llena']
  ifFalse: [tope := tope + 1. elementos at: tope put: unElem]
  ```

  ← El encabezado es *apilar: unElem*, donde *unElem* es el argumento del método. El resto constituye el cuerpo de la operación

  Los métodos existen dentro de los límites demarcados por un objeto, y pueden acceder a todas las variables de instancia del mismo. Esto resume dos propiedades simétricas:

  - El método de un objeto sólo se interesa por el estado interno de dicho objeto
  - Un método no puede afectar directamente las variables de instancia de otro objeto

  La ligadura entre la invocación (envío del mensaje) con la implementación de la operación (método) se lleva a cabo en tiempo de ejecución → Ligadura dinámica de mensajes a métodos

  Los objetos son creados mediante el envío de un *mensaje de creación* (new) a la clase del objeto.

  Ej: pilaUno := Pila new

- *CONTROL*: Las estructuras de control están implementadas mediante el mecanismo de *pasaje de mensajes* más una clase predefinida denominada *BlockContext*. Esta clase brinda la capacidad de encapsular una secuencia de acciones dentro de un objeto: el bloque.

  Un *bloque* es una secuencia de expresiones separadas por puntos, y delimitada por corchetes. Asimismo, un bloque puede tener argumentos, los cuales son especificados entre el corchete izquierdo y una barra. Además, cada argumento debe ir precedido por un ":".

  Ej: [:j :k | (j*k) printString] ← j y k son parámetros

  Dado que un bloque es un objeto, este puede ser asignado a un identificador, o pasado como argumento de un mensaje.

  Para poder ejecutar un *bloque sin parámetros*, la clase BlockContext provee el método *value*. Siempre la ejecución de un bloque retorna el resultado de la última expresión evaluada dentro del mismo.

  Ej: [i:=3. J:=8 printString] value ← el resultado es el objeto '8' (el string 8)

  Es importante notar que en respuesta al mensaje *value*, *el bloque se ejecuta dentro del contexto en que fue definido, independientemente del contexto en que se encuentra al momento de ser ejecutado*.

Ej:

Supongamos que en las clases A y B se definen los métodos metA y metB correspondientemente:

```
metA                          metB: bloque
 i := 10.                      i := 3+4.
 p := B new.                   ^ bloque value.
 ^ p metB: [i := i+1].
```

Luego, si *a* es una instancia de la clase A, y efectuamos la siguiente asignación: z := a metA

Resulta que z queda asignado al objeto 11. Cuando, se evaluó el mensaje ^bloque value, el objeto i del bloque hace referencia al i asignado en metA y no al empleado en metB.

Por otra parte, para ejecutar *bloques con un argumento* se emplea el mensaje *value:*, para el caso de dos argumentos el mensaje *value:value:*, y así siguiendo.

Ej: [:j :k | (j*k) printString] value: 2 value: 3 ← el resultado es el objeto '6'

En este caso, j y k quedan asociados a los objetos 2 y 3 respectivamente.

Selección: Las estructuras de control *condicionales* son implementadas a través de métodos correspondientes a las *subclases True y False* de la clase Boolean (el receptor del mensaje es un objeto booleano: true/false), los cuales tienen la siguiente sintaxis:

- (<expresión de prueba>) ifTrue: [<secuencia-expresiones>]
- (<expresión de prueba>) ifFalse: [< secuencia-expresiones>]

La secuencia de expresiones colocada entre corchetes es un bloque sin argumentos.

La clase *true* implementa el método ifTrue: retornando la evaluación del bloque correspondiente al argumento. La clase *false* lo implementa retornando el objeto *nil*. Análogo para el método ifFalse:.

También es posible utilizar estos mensajes en forma combinada (expresión if-then-else):

(<expresión de prueba>) ifTrue: [<secuencia-expresiones>]
ifFalse: [<secuencia-expresiones>]

Obs: *ifTrue:ifFalse:* es un método definido tanto en la clase True como en la clase False. Si el objeto receptor es *true* entonces se utiliza el método definido en la clase True, donde se envía el mensaje *value* al bloque pasado como argumento en la palabra clave ifTrue:. Si en cambio, el objeto receptor del mensaje es *false*, luego se ejecutará el método ifTrue:ifFalse: de la clase False, donde directamente se evalúa el bloque correspondiente a la palabra clave ifFalse:.

Repetición: En la *clase Block* se definen métodos para expresar repetición. Un bucle *while* tiene dos formas:

- [<secuencia-expresiones>] whileTrue: [<secuencia-expresiones>]
- [<secuencia-expresiones>] whileFalse: [<secuencia-expresiones>]

En el caso de *whileTrue:*, el bloque receptor es repetidamente evaluado. Mientras la evaluación arroje el objeto true, el argumento del bloque es también evaluado, si no retornará el objeto nil. En el caso de *whileFalse:* es similar. En todos los casos, los bloques receptores deben finalizar con una expresión cuyo resultado sea instancia de la clase Boolean (dado que la evaluación de la última expresión del bloque es el resultado arrojado por el bloque).

```
Clase: BlockContext

whileTrue: aBlock
" Evaluate the argument, aBlock, as long as the value of the receiver is true."

self value ifTrue: [ aBlock value. ^self whileTrue: aBlock ]
```

Por otra parte, también se proveen métodos de la *clase Integer* para expresar *repetición*:

- x timesRepeat: [< secuencia de expresiones >] ← itera x veces
- x to: y do: [< secuencia de expresiones >] ← for x to y (incrementa de a 1)
- x to: y by: step do: [< secuencia de expresiones >] ← for x to y (incrementa de a *step*)

```
Clase: Number

timesRepeat: aBlock
"Evaluate the argument, aBlock, the number of times represented by the receiver."

| count |
count := 1.
[count <= self] whileTrue:  [aBlock value. count _ count + 1]


to: stop do: aBlock
" Evaluate aBlock for each element of the interval (self to: stop by: 1)."

| nextValue |
nextValue := self.
[nextValue <= stop] whileTrue:  [aBlock value: nextValue. nextValue _ nextValue + 1]
```

Iteradores: La *clase Collection* provee los denominados *iteradores*. Estos métodos recorren de manera automática distintas estructuras de datos modeladas mediante subclases de la clase Collection, tales como la clase Array y String. Algunso iteradores son *do:*, *select:*, *collect:* y *reject:*.

```
Clase: Collection

do: aBlock
1 to: self size do: [:index | aBlock value: (self at: index)]
```

→ La característica más importante de estas estructuras de control es la combinación de una sintaxis muy sencilla con un mecanismo básico de pasaje de mensajes, que conforman una herramienta de programación muy poderosa y elegante que recuerda a la programación declarativa. Los rasgos de la programación imperativa se ocultan en las profundidades de la jerarquía de clases predefinidas por el sistema.

Recursividad: Smalltalk provee facilidades para definir métodos en forma recursiva.

```
Clase: Integer

factorial
"Answer the factorial of the receiver."

self = 0 ifTrue: [^ 1].
self > 0 ifTrue: [^ self * (self - 1) factorial].
self error: 'Not valid for negative integers'
```

← El último caso podría haber sido:

self <0 ifTrue: [^self error: 'Not valid…']

No es necesario ya que de entrar en alguno de los casos anteriores, corta la ejecución con el retorno (^)

- *HERENCIA*: Smalltalk soporta *herencia simple*, y *de implementación*:

  - *Herencia Simple*: Cada clase tiene exactamente una *superclase* (excepto la clase *Object*, que no tiene ninguna – Es una *metaclase*), y puede tener cualquier cantidad de *subclases*.

Una subclase hereda variables y métodos de su superclase. De este modo, las variables y métodos de instancia de la superclase se convierten automáticamente en variables y métodos de instancia de la subclase.

  - *Herencia de* Implementación: Los métodos de una clase pueden acceder, además de a sus propias variables de instancia, a todas las variables de instancia de todas sus superclases; y pueden invocar a todos los métodos definidos en todas sus superclases.

Observaciones (Herencia en Smalltalk):

- Cuando se le envía un mensaje a un objeto, se ejecutará el método que esté definido en la clase ubicada en el menor nivel de jerarquía, a partir de la clase de la cual el objeto receptor es instancia (ligadura dinámica de mensajes a métodos).

- Cuando se define un método que ya está definido con igual signatura en alguna clase superior en la jerarquía de clases se dice que el método está sobrecargado (redefinición de operaciones) → Los métodos de una subclase invalidan a los métodos heredados con igual nombre (salvo que se haga uso explícito del *super*).

- No es posible utilizar un nombre para definir una variable de instancia en una clase si ese nombre ya fue utilizado para definir alguna variable en alguna de las superclases.

Sintaxis - Smalltalk:

- arreglos se indican con #() → #(elem1 elem2 … elemN)
- chars se indican con $ → $a
- strings se indican con ‘ → ‘cadena de caracteres’

[Smalltalk en Ghezzi]

Clases y Herencia: El mecanismo de encapsulamiento de Smalltalk es la *clase*. Una clase describe un tipo de dato abstracto, cuyas instancias son *objetos*. Una clase contiene la descripción de las *variables de instancia*, que representan la estructura de datos a ser alocada en cada instanciación, y la descripción de operaciones (*métodos de instancia*) usadas para manipular objetos. Las variables de instancia no son visibles para otros objetos, sólo pueden ser manipuladas por los métodos de instancia.

Las clases pueden ser organizadas en una jerarquía, donde cada subclase tiene a lo sumo una superclase (*Herencia Simple*). Una subclase hereda todas las variables de instancia y métodos de su superclase. Además, una clase puede tener sus propias variables de instancia, puede definir métodos que no existían en sus clases ancestro, y puede redefinir métodos heredados.

Cuando un mensaje es enviado a un objeto, se busca en la clase del objeto el método correspondiente. Si no se encuentra, la búsqueda continúa en la superclase y así sucesivamente en la cadena de herencia hasta que se encuentre una clase que provea el método, o hasta que se encuentra la clase de sistema *Object* que no tiene ancestros (en cuyo caso se reporta un error).

Variables y Tipado Dinámico: Smalltalk trata las variables uniformemente como referencias a objetos, las cuales no tienen tipo: una referencia no está ligada estáticamente a una clase, sino que puede referenciar a una instancia de cualquier clase. Un mensaje enviado a una variable es correcto si la clase del objeto actualmente ligado a la variable provee un método para el mensaje, ya sea directamente o indirectamente a través de herencia.

Los objetos son alocados dinámicamente por el programador, a través de la sentencia *new* (constructor de instancia). Las ligaduras dinámicas de los tipos a las variables hacen que las instrucciones de Smalltalk sean polimórficas. El significado de una operación invocada a través del envío de un mensaje a un objeto depende de la clase del objeto que está actualmente ligado a la variable (ocurre un error si no se encuentra un método que matchee atravesando la jerarquía de herencia hacia arriba).

El rol de los Objetos: El concepto primario de Smalltalk es el de objeto. Todo en Smalltalk son objetos, incluyendo clases y estructuras de control. Una clase es un objeto de la clase predefinida *Object* y, en particular, provee un método para responder a mensajes requiriendo una instanciación (*new*).
Los programas pueden ser vistos como datos. Los bloques pueden ser asignados a variables, y las estructuras de control condicional y de iteración son definidas en base a la definición de bloque. En lugar de proveer un número de estructuras de control predefinidas, Smalltalk permite definir nuevas estructuras de control.

Dado que nuevos tipos de datos pueden ser definidos usando clases, Smalltalk puede considerarse un *lenguaje extensible*: todos los mecanismos computacionales del lenguaje pueden ser extendidos por el programador.