Conceptos de Paradigmas y Lenguajes de Programación

Resumen teorías

Clase 1

Idea principal

Introducir, analizar y evaluar los conceptos más importantes de los lenguajes de programación

Criterios para evaluar los lenguajes de programación

Simplicidad y legibilidad

Los lenguajes de programación deberían:

- Poder producir programas fáciles de escribir y de leer
- Resultar fáciles a la hora de aprenderlo o enseñarlo

Ejemplo de cuestiones que atentan contra esto:

- Muchos componentes elementales
- Conocer subconjuntos de componentes
- El mismo concepto semántico pero distinta sintaxis
- Distintos conceptos semánticos pero la misma notación sintáctica
- Abuso de operadores sobrecargados

Claridad en los bindings

Los elementos de los lenguajes de programación pueden ligarse a sus atributos o propiedades en diferentes momentos:

- Definición del lenguaje
- Implementación del lenguaje
- En escritura del programa
- Compilación
- Cargado del programa
- En ejecución

La ligadura en cualquier caso debe ser clara

Confiabilidad

La confiabilidad está relacionada con la seguridad:

- Chequeo de tipos: cuanto antes se encuentren errores menos costoso resulta realizar los arreglos que se requieran
- Manejo de excepciones: la habilidad para interceptar errores en tiempo de ejecución, tomar medidas correctivas y continuar

Soporte

Debería ser accesible para cualquiera que quiera usarlo o instalarlo

Lo ideal sería que su compilador o intérprete sea de dominio público

Debería poder ser implementado en diferentes plataformas

Deberían existir diferentes medios para poder familiarizarse con el lenguaje: tutoriales, cursos textos, etc.

Abstracción

Capacidad de definir y usar estructuras u operaciones complicadas de manera que sea posible ignorar muchos de los detalles

Abstracción de procesos y de datos

Ortogonalidad

Significa que un conjunto pequeño de constructores primitivos puede ser combinado en número relativamente pequeño a la hora de construir estructuras de control y datos

Cada combinación es legal y con sentido

El usuario comprende mejor si tiene un pequeño número de primitivas y un conjunto consistente de reglas de combinación

Eficiencia

En tiempo y espacio, de esfuerzo humano. Es optimizable

Sintaxis

Conjunto componer de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas

Proporciona mecanismos para decir si un programa es válido

Características de la sintaxis

La sintaxis debe ayudar al programador a escribir programas correctos sintácticamente

La sintaxis establece reglas que sirven para que el programador se comunique con el procesador

La sintaxis debe contemplar soluciones a características tales como:

- Legibilidad
- Verificabilidad
- Traducción
- Falta de ambigüedad

La sintaxis establece reglas que definen cómo deben combinarse las componentes básicas, llamadas “word”, para formar sentencias y programas

Elementos de la sintaxis

- Alfabeto o conjunto de caracteres
- Identificadores:
    - Elección más ampliamente utilizada: cadena de letras y dígitos, que deben comenzar con una letra
    - Si se restringe la longitud se pierde legibilidad
- Operadores
- Comentarios y uso de blancos
- Palabras clave y palabras reservadas:
    - Palabras clave o keywords son aquellas que tienen un significado dentro de un contexto
    - Palabras reservadas son palabras clave que además no pueden ser usadas por el programador como identificador de otra entidad
    - Ventajas:

- Permiten al compilador y al programador expresarse claramente
- Hacen los programas más legibles y permiten una rápida traducción

Estructura sintáctica

Vocabulario o words

Conjunto de caracteres y palabras necesarias para construir expresiones, sentencias y programas. Ej: identificadores, operadores, palabras claves, etc.

Las words no son elementales se construyen a partir del alfabeto

Expresiones

Son funciones que a partir de un conjunto de datos devuelven un resultado

Son bloques sintácticos básicos a partir de los cuales se construyen las sentencias y programas

Sentencias

Componente sintáctico más importante

Tiene un fuerte impacto en la facilidad de escritura y legibilidad

Hay sentencias simples, estructuradas y anidadas

Reglas léxicas y sintácticas

- Reglas léxicas: Conjunto de reglas para formar las “word”, a partir de los caracteres del alfabeto
- Reglas sintácticas: Conjunto de reglas que definen cómo formar las “expresiones” y “sentencias”

Tipos de sintaxis

- ABSTRACTA: se refiere básicamente a la estructura
- CONCRETA: se refiere básicamente a la parte léxica
- PRAGMÁTICA: se refiere básicamente al uso práctico

Ejemplo de sintaxis concreta y abstracta

C	Pascal

while (x!= y){
<código>
}
	while x<>y do begin
	<código>
	end;

Son diferentes respecto a la sintaxis concreta porque existen diferencias léxicas entre ellas (!= vs <>, {} vs begin end;)

Son iguales respecto a la sintaxis abstracta, ya que ambas tienen la misma estructura: <while> seguido de <condición> seguido de <bloque>

Ejemplo de sintaxis pragmática:

Ej.1: <> es más legible que !=

Ej.2: En C y Pascal {} o begin-end; pueden omitirse si el bloque está compuesto por una sola sentencia

Pragmáticamente puede conducir a error ya que si se necesitara agregar una sentencia debe agregarse el begin-end; o los {} (es decir que no define una única forma de escribirse)

Cómo definir la sintaxis

Se necesita una descripción finita para definir un conjunto infinito (conjunto de todos los programas bien escritos)

Formas para definir la sintaxis:

- Lenguaje natural: Ej.: Fortran
- Utilizando la gramática libre de contexto, definida por Backus y Naun (BNF): Ej: Algol
- Diagramas sintácticos que son equivalentes a BNF pero mucho más intuitivos

BNF

- Es una notación formal para describir la sintaxis
- Es un metalenguaje
- Utiliza metasímbolos: “<” “>” “::=” “|”
- Define las reglas por medio de “producciones”

Gramática

Conjunto de reglas finita que define un conjunto infinito de posibles sentencias válidas en el lenguaje

Una gramática está formada por una 4-tupla: G = (N, T, S, P)

N

Conjunto de símbolos no terminales

T

Conjunto de símbolos terminales

S

Símbolo distinguido de la gramática que pertenece a N (lo que vamos a definir)

P

Conjunto de producciones

Diagramas sintácticos (Conway)

- Es un grafo sintáctico o carta sintáctica
- Cada diagrama tiene una entrada y una salida, y el camino determina el análisis
- Cada diagrama representa una regla o producción
- Para que una sentencia sea válida, debe haber un camino desde la entrada hasta la salida que la describa

- Se visualiza y entiende mejor que BNF o EBNF

Clase 2

Semántica

Describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático que es sintácticamente válido

Conjunto de reglas para dar significado a los programas sintácticamente válidos

Proporciona mecanismos para que, dado un programa válido, la máquina pueda decir que significa

Hay características de la estructura de los lenguajes de programación que son difíciles o imposibles de describir con las gramáticas BNF/EBNF

Semántica estática

Se la llama así porque el análisis para el chequeo se hace en compilación (antes de la ejecución)

No está relacionada con el significado de la ejecución del programa, está más relacionado con las formas válidas (con la sintaxis)

El análisis está ubicado entre el análisis sintáctico y el análisis de semántica dinámica, pero más cercano a la sintaxis

Gramática de atributos

Para describir la sintaxis y la semántica estática formalmente sirven las denominadas gramáticas de atributos, inventadas por Knuth en 1968

Son gramáticas sensibles al contexto (GSC): si se relacionan con el significado

La usan los compiladores, antes de la ejecución

Generalmente resuelven los aspectos de la semántica estática

A las construcciones del lenguaje se les asocia información a través de “atributos” asociados a los símbolos de la gramática (terminales o no terminales), que sirven para detectar errores

Un atributo puede ser: el valor de una variable, el tipo de una variable o expresión, lugar que ocupa una variable en la memoria, dígitos significativos de un número, etc.

Los valores de los atributos se obtienen mediante las llamadas “ecuaciones o reglas semánticas” asociadas a las producciones gramaticales

De la ejecución de las ecuaciones:

- Se ingresan símbolos a la tabla de símbolos
- Detectar y dar mensajes de error
- Detecta dos variables iguales
- Controla tipo y variables de igual tipo
- Ciertas combinaciones no permitidas (reglas específicas del lenguaje)
- Generar un código para el siguiente paso

Semántica dinámica

Es la que describe el significado de ejecutar las diferentes construcciones del lenguaje de programación

Su efecto se ve durante la ejecución del programa

Influirá la interacción con el usuario y errores de la programación

Formales y complejas

- Semántica axiomática
- Semántica denotacional: la usan los diseñadores de compiladores

No formal

Semántica operacional: la usan para manuales de lenguajes

Sirven para comprobar la ejecución, la exactitud de un lenguaje, comparar funcionalidades de distintos programas

Se pueden usar combinados, no todos sirven para todos los tipos de lenguajes de programación

Semántica axiomática

Considera al programa como “una máquina de estados” donde cada instrucción provoca un cambio de estado

Se parte de un axioma (verdad) que sirve para verificar "estados y condiciones" a probar

Los constructores de un lenguaje de programación se formalizan describiendo como su ejecución provoca un cambio de estado (cada vez que se ejecuta)

Se desarrolló para probar la corrección de los programas

La notación empleada es el “cálculo de predicados”

Un estado se describe con un predicado

El predicado describe los valores de las variables en ese estado

Existe un estado anterior y un estado posterior a la ejecución del constructor

Cada sentencia se precede y se continúa con una expresión lógica que describe las restricciones y relaciones entre los datos

Precondiciones (condiciones de estado previo)

Poscondiciones (condiciones de estado posterior)

Semántica denotacional

Se basa en la teoría de funciones recursivas y modelos matemáticos, es más exacto para obtener y verificar resultados, pero es más difícil de leer

Define una correspondencia entre los constructores sintácticos y sus significados

Describe la dependencia funcional entre el resultado de la ejecución y sus datos iniciales

Lo que hace es buscar funciones que se aproximen a las producciones sintácticas

Semántica operacional

El significado de un programa se describe mediante otro lenguaje de bajo nivel implementado sobre una máquina abstracta

Cuando se ejecuta una sentencia del lenguaje de programación los cambios de estado de la máquina abstracta definen su significado

Es un método informal porque se basa en otro lenguaje de bajo nivel y puede llevar a errores

Es el más utilizado en los libros de texto para explicar el significado de los lenguajes

Ejemplo:

Pascal	Máquina abstracta

for i := pri to ul do
begin
  <código>
end;
	i := pri (inicializo i)
	lazo if i > ul goto sal
	  <código>
	  i := i +1
	  goto lazo
	sal

Procesamiento de un programa

Primero: Unos y ceros (alto embole y te cagas confundiendo cada 2x3)
Segundo: Código mnemotécnico (ADD, SUB, NE, JMP)
Problemas:

- Cada máquina o familia de procesadores tiene su propio SET de instrucciones, entonces cada uno tiene que tener su propio:
    - Lenguaje ensamblador
    - Programa ensamblador
    - Código nemotécnico
- Imposible intercambiar programas entre distintas máquinas o de distintas familias de procesadores
- Diferentes versiones para una misma CPU pueden tener sets de instrucciones incompatibles
- Modelos evolucionados de una familia de CPU pueden incorporar instrucciones nuevas

Tercero: Lenguajes de alto nivel (así sí 😎)

¿Cómo pasamos de lenguaje de alto nivel a unos y ceros?: con programas traductores (intérpretes, compiladores)

Interpretación

Hay un Programa que está escrito en lenguaje de programación interpretado

Hay un Programa llamado Intérprete que realiza la traducción de ese lenguaje interpretado en el momento de ejecución

El proceso que realiza cuando se ejecuta sobre cada una de las sentencias del programa es:

1. Leer
2. Analizar
3. Decodificar
4. Ejecutar
5. Vuelta a 1

Por cada una de las instrucciones del programa

Ventaja: solo pasa por algunas instrucciones, no por todo el programa (varía según la ejecución)

Desventaja: cada vez que vuelvo a ejecutar el programa se repite toda la secuencia

Compilación

Tenemos nuestro programa escrito en un lenguaje de alto nivel de este tipo

Hay un programa llamado compilador que realiza la traducción a lenguaje de máquina

Se traduce/compila antes de ejecución

Pasa por todas las instrucciones antes de la ejecución

Ventaja: El código que se genera se guarda y se puede reusar ya compilado

El compilador toma todo el programa escrito en un lenguaje de alto nivel que llamamos lenguaje fuente antes de su ejecución

Luego de la compilación va a generar:

- O un lenguaje objeto que es generalmente el ejecutable (en lenguaje de máquina)
- O un lenguaje de nivel intermedio (lenguaje ensamblador)

Comparación entre compilador e intérprete

Por cómo se ejecuta

Intérprete:

- Se utiliza el intérprete en la ejecución
- Ejecuta el programa línea por línea
- Por donde pase dependerá de la acción del usuario, de la entrada de datos y/o de alguna decisión del programa
- Siempre se debe tener el programa interprete
- El programa fuente será público (necesito ambos)

Compilador:

- Se utiliza el compilador antes de la ejecución
- Produce un programa ejecutable equivalente en lenguaje objeto
- El programa fuente no será público
- Una vez compilado no necesito tener el programa compilador

Por el orden de ejecución

Intérprete: sigue el orden lógico de ejecución (no necesariamente recorre todo el código)

Compilador: sigue el orden físico de las sentencias (recorre todo)

Por el tiempo consumido de ejecución

Intérprete:

- Por cada sentencia que pasa realiza el proceso de decodificación (lee, analiza y ejecuta) para determinar las operaciones y sus operandos
- Es repetitivo
- Si la sentencia está en un proceso iterativo (ej.: for/while), se realizará la tarea de decodificación tantas veces como sea requerido
- La velocidad de proceso se puede ver afectada por esto

Compilador:

- Pasa por todas las sentencias

- No repite lazos
- Traduce todo de una sola vez
- Genera código objeto ya compilado
- La velocidad de compilar dependerá del tamaño del código

Por la eficiencia posterior

Intérprete:

- Más lento en ejecución
- Se repite el proceso cada vez que se ejecuta el mismo programa o pasa por las mismas instrucciones
- Para ser ejecutado en otra máquina se necesita tener si o si el intérprete instalado
- El programa fuente será público

Compilador:

- Más rápido ejecutar desde el punto de vista del hardware porque ya está en un lenguaje de más bajo nivel
- Detectó más errores al pasar por todas las sentencias
- Está listo para ser ejecutado
- Ya compilado es más eficiente
- Por ahí tardó más en compilar porque se verifica todo previamente
- El programa fuente no será público

Por el espacio ocupado

Intérprete:

- No pasa por todas las sentencias entonces ocupa menos espacio de memoria
- Cada sentencia se deja en la forma original y las instrucciones interpretadas necesarias para ejecutarlas se almacenan en los subprogramas del intérprete en memoria
- Tablas de símbolos, variables y otros se generan cuando se usan en forma dinámica (Ej. Python, Ruby)

Compilador:

- Pasa por todas las sentencias

- Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto
- Cosas cómo tablas de símbolos, variables, etc. se generan siempre se usen o no
- El compilador en general ocupa más espacio

Por la detección de errores

Intérprete:

- Las sentencias del código fuente pueden ser relacionadas directamente con la sentencia en ejecución entonces se puede ubicar donde se produjo el error
- Es más fácil detectarlos por donde pasa la ejecución
- Es más fácil corregirlos

Compilador:

- Se pierde la referencia entre el código fuente y el código objeto
- Es casi imposible ubicar el error, pobres en significado para el programador
- Se deben usar otras técnicas (ej. Semántica Dinámica)

Combinación de Técnicas

Interpretación pura y Compilación pura son dos extremos

En la práctica muchos lenguajes combinan ambas técnicas para sacar provecho a cada una

Los compiladores y los intérpretes se diferencian en como reportan los errores de ejecución

Ciertos entornos de programación contienen las dos versiones

Primero interpretar y después compilar

Se utiliza el intérprete en la etapa de desarrollo para facilitar el diagnóstico de errores

Con el programa validado se compila para generar un código objeto más eficiente

Primero compilar y luego interpretar

Se hace traducción a un código intermedio a bajo nivel que luego se interpretará

Sirve para generar código portable, es decir, código fácil de transferir a diferentes máquinas y con diferentes arquitecturas

Ejemplos:

- Compilador Java genera un código intermedio llamado “bytecode” que luego es interpretado por máquina virtual Java (JVM) en la máquina cliente
- C# (C Sharp) de Microsoft .NET
- VB.NET (Visual Basic .NET de Microsoft)
- Python

Funcionamiento de los compiladores

Pueden generar un "código ejecutable" (.exe) o un "código intermedio" (.obj)
La compilación puede ejecutarse en 1 o 2 etapas
En ambos casos se cumplen varias sub-etapas, las principales son:

1. Etapa de Análisis:
    - Análisis léxico (Programa Scanner)
    - Análisis sintáctico (Programa Parser)
    - Análisis semántico (Programa de Semántica estática)

    Podría generarse un código intermedio entre el paso 1 y el 2

2. Etapa de Síntesis
    - Optimización del código
    - Generación del código final

1. Está más vinculado al código fuente
2. Está más vinculado a características del código objeto y del hardware y arquitectura

Etapa de análisis del programa fuente

Análisis léxico (Scanner)

Es un proceso que lleva tiempo

Hace el análisis a nivel de palabra (LEXEMA)

Divide el programa en sus elementos/categorías: identificadores, delimitadores, símbolos especiales, operadores, números, palabras clave, palabras reservadas, comentarios, etc.

Analiza el tipo de cada uno para ver si son TOKENS válidos Filtra comentarios y separadores (como: espacios en blanco, tabulaciones, etc.)

Lleva una tabla para la especificación del analizador léxico. Incluye cada categoría, el conjunto de atributos y acciones asociadas

Pone los identificadores en la tabla de símbolos

Reemplaza cada símbolo por su entrada en la tabla de símbolos

Genera errores si la entrada no coincide con ninguna categoría léxica

El resultado de este paso será el descubrimiento de los ítems léxicos o tokens y detección de errores

Análisis sintáctico (Parser)

El análisis se realiza a nivel de sentencia/estructuras

Usa los tokens del analizador léxico

Tiene como objetivo encontrar las estructuras presentes en su entrada

Estas estructuras se pueden representar mediante el árbol de análisis sintáctico, que explica cómo se puede derivar la cadena de entrada en la gramática que especifica el lenguaje

Se identifican las estructuras de las sentencias, declaraciones, expresiones, etc. ayudándose con los tokens

El analizador sintáctico (Parser) se alterna/interactúa con el análisis léxico y análisis semántico

Usualmente usan técnicas de gramática formal

Se usa una gramática para construir el "árbol sintáctico"/"árbol derivación" del programa

El objetivo principal de un árbol de derivación es representar una sentencia del lenguaje y validar de esta forma que pertenece o no a la gramática para ver que lo que entra es correcto

Análisis semántica (semántica estática)

Debe pasar antes bien Scanner y Parser

Es la fase medular, una de las más importantes

Procesa las estructuras sintácticas (reconocidas por el analizador sintáctico)

Agrega otro tipo de información implícita y la estructura del código ejecutable continúa tomando forma

Realiza la comprobación de tipos (aplica gramática de atributos)

Agrega a la tabla de símbolos los descriptores de tipos

Realiza comprobaciones de duplicados, problema de tipos, etc

Realiza comprobaciones de nombres (toda variable debe estar declarada en su entorno)

Es el nexo entre etapas inicial y final del compilador(Análisis y Síntesis)

Generación de código intermedio

Es realizar la transformación del "código fuente" en una representación de "código intermedio" para una máquina abstracta

Queda una representación independiente de la máquina en la que se va a ejecutar el programa

Idea de la generación de código intermedio:

- Debe ser fácil de producir
- Debe ser fácil de traducir al programa objeto

El código intermedio más habitual es el código de 3-direcciones

Pasa todo el código del programa a secuencia de proposiciones de la forma:

x:= y op z

Etapa de síntesis

Construye el programa ejecutable y genera el código necesario

Si hay traducción separada de otros módulos (módulos, unidades, librerías, procedimientos, funciones, subrutinas, macros, etc.)

Interviene el Linkeditor (Programa) y se enlazan los distintos módulos objeto del programa

Se genera el módulo de carga: programa objeto completo

Se realiza el proceso de optimización (Optativo)

El Loader (Programa) lo carga en memoria

Optimización

Es Optativo. No se hace siempre y no lo hacen todos los compiladores

Los optimizadores de código (programas) pueden ser herramientas independientes, o estar incluidas en los compiladores e invocarse por medio de opciones de compilación

Hay diversas formas y cosas a optimizar:

- Elegir entre velocidad de ejecución y tamaño del código ejecutable
- Generar código para un microprocesador específico dentro de una familia de microprocesadores
- Eliminar la comprobación de rangos o desbordamientos de pila
- Evaluación para expresiones booleanas
- Eliminación de código muerto o no utilizado
- Eliminación de funciones no utilizadas

Clase 3

Semántica operacional

La semántica operacional es fundamental para diversos aspectos del proceso de desarrollo de software, como el diseño de lenguajes de programación, la verificación de programas y la comprensión de cómo se ejecutan los programas en un nivel más bajo

Entidades

- Variables: Nombre, tipo, área de memoria, etc.
- Rutinas: Nombre, parámetros formales y reales, convención de pasaje de parámetros, etc.
- Sentencias: Acción asociada

Ligaduras

Es el momento en el que un atributo de una entidad se asocia con un valor
Los programas trabajan con entidades
Las entidades tienen atributos (valor de la variable, nombre, etc.)
Estos atributos tienen que establecerse antes de poder usar la entidad

Los lenguajes difieren en:

- El número de entidades
- El número de atributos que se les pueden ligar
- El momento de la ligadura (binding time) (estática y dinámica)
- La estabilidad de la ligadura: ¿una vez establecida se puede modificar o es fija? (¿y si se puede modificar y es una constante?)

Ligadura estática

1. Se establece antes de la ejecución
2. No se puede modificar

El término estática hace referencia al binding time (1) y a su estabilidad (2).

Ligadura dinámica

1. Se establece durante la ejecución
2. Se puede modificar durante la ejecución de acuerdo a alguna regla específica del lenguaje

Excepción: constantes (el binding es en runtime pero no puede ser modifica luego de establecida)

Algunos atributos pueden ligarse en el momento de la definición del lenguaje, otros en el momento de implementación, en tiempo de traducción (compilación), y otros en el tiempo de ejecución

Atributos de las variables

Nombre

String de caracteres que se usa para referenciar a la variable (identificador)

Es introducido por una sentencia de declaración

La longitud máxima varía según el lenguaje (se define en la etapa de definición del lenguaje)

Alcance

Es el rango de instrucciones en el que se conoce el nombre, es visible, y puede ser referenciada (visibilidad)

Las instrucciones del programa pueden manipular las variables a través de su nombre dentro de su alcance. Afuera de ese alcance son invisibles

Ligadura por alcance estático

Llamado alcance léxico

Se define el alcance en términos de la estructura léxica del programa

Puede ligarse estáticamente a una declaración de variables (explícita o implícita) examinando el texto del programa, sin necesidad de ejecutarlo

La mayoría de los lenguajes adoptan reglas de ligadura de alcance estático

Ligadura por alcance dinámico

Define el alcance del nombre de la variable en términos de la ejecución del programa

Cada declaración de variable extiende su efecto sobre todas las instrucciones ejecutadas posteriormente, hasta que una nueva declaración para una variable con el mismo nombre es encontrada durante la ejecución

Global

Son todas las referencias a variables creadas en el programa principal

Local

Son todas las referencias a variables que se han creado dentro de una unidad (programa o subprograma)

No local

Son todas las referencias que se utilizan dentro del subprograma pero que no han sido creadas en el subprograma. (son externas a él)

Espacio de nombre

Es una zona separada abstracta del código donde se pueden agrupar, declarar y definir objetos (variables, funciones, identificador de tipo, clase, estructura, etc.)

Ayudan a evitar problemas con identificadores con el mismo nombre en grandes programas, o cuando se usan bibliotecas externas para evitar colisión de nombres

Al espacio de nombre se le asigna un nombre o identificador propio

Son utilizados por los lenguajes de tipo dinámico

Es un recurso de ciertos lenguajes de programación

Ayudan a resolver el alcance dentro de ese espacio de nombres

Tipo

Es el tipo de variables definidas, tiene asociadas rango de valores y operaciones permitidas

Se define el tipo de una variable cómo la especificación de:

- El conjunto de valores que se pueden asociar a la variable
- Un conjunto de operaciones permitidas (crear, acceder, modificar)
- Una variable, de un tipo dado, es una instancia

Cuando se define el lenguaje, ciertos nombres de tipos están vinculados a ciertas clases de valores y conjuntos de operaciones. Por ejemplo, el tipo entero y sus operadores asociados (+-*/) están vinculados a su representación matemática

Cuando se implementa el lenguaje los valores y operaciones están vinculados a una determinada representación de máquina

El binding también puede restringir el conjunto de valores que se pueden representar según la capacidad de almacenamiento de la máquina de destino

Antes que una variable pueda ser referenciada debe ligarse a un tipo

El tipo de una variable ayuda a:

- Proteger a las variables de operaciones no permitidas
- Chequear tipos
- Verificar el uso correcto de las variables (cada lenguaje tiene sus reglas de combinaciones de tipos)
- Detectar errores en forma temprana y a la confiabilidad del código

Tipos predefinidos

Son los tipos base que están descritos en la definición del lenguaje (enteros, reales, flotantes, booleanos, etc....)

Cada uno tiene valores y operaciones

Tipo boolean valores: true, false operaciones: and, or , not

Los valores se ligan en la implementación a representación de máquina según la arquitectura

Tipos definidos por el usuario

Permiten al programador con la declaración de tipos definir nuevos tipos a partir de los tipos predefinidos y de los constructores

Son esenciales para la organización y la abstracción

Permite al programador crear abstracciones, encapsular lógica y datos, reutilizar código y mejorar la claridad y legibilidad del código

Son fundamentales para el desarrollo de programas complejos y para mantener un código organizado y mantenible

Tipos de datos abstractos (TDA)

Son estructuras de datos que representa a un nuevo tipo abstracto con un nombre que identifica

Está compuesto por una colección de operaciones definidas (rutinas). Las rutinas son usadas para manipular los objetos de este nuevo tipo

TAD comunes: Listas, colas, pilas, árboles, grafos, etc.

Cada TAD define un conjunto de operaciones permitidas, pero oculta los detalles de implementación interna

No hay ligadura por defecto, el programador debe especificar la representación y las operaciones

Momento de ligadura estático

El tipo se liga en compilación y no puede ser cambiado en ejecución

La ligadura entre variable y tipo se hace con la declaración

El chequeo de tipo también será estático

Explícito

La ligadura se establece mediante una sentencia de declaración

La ventaja reside en la claridad de los programas y en una mayor fiabilidad, porque cosas como errores ortográficos en nombres de variables pueden detectarse en tiempo de traducción

Implícito

Si no fue declarada la ligadura se deduce por "reglas propias del lenguaje"

Esto ocurre sin que el programador tenga que especificar explícitamente el tipo de datos de la variable

Ej. Fortran 77: variables que empiezan con I a N son Enteras variables que empiezan con el resto de las letras son Reales

Inferido

El tipo se deduce automáticamente de los tipos de sus componentes.

Se basa en el contexto del código y en el valor asignado a la variable

Se realiza en la traducción

Aplica en general a Lenguajes Funcionales

Momento de ligadura dinámico

El tipo se liga a la variable en ejecución y puede modificarse

Cambia cuando se le asigna un valor mediante una sentencia de asignación (no declaración)

No se detectan incorrecciones de tipo en las asignaciones

El tipo de la parte izquierda simplemente se cambia al tipo de la derecha

El costo de implementación de la ligadura dinámica es mayor (sobre todo el tiempo de ejecución por: comprobación de tipos, mantenimiento del descriptor asociado a cada variable en el que se almacena el tipo actual, cambio en el tamaño de la memoria asociada a la variable, etc.)

Chequeo dinámico

Menor legibilidad

Más errores

Los lenguajes interpretados en general adoptan ligadura dinámica de tipos

L-valor

Es el lugar de memoria asociado con la variable, está asociado al tiempo de vida (variables se alocan y desalocan)

Las variables se alocan en un área de memoria

Esa área de memoria debe ser ligada a la variable en algún momento

El L-valor de una variable es el área de memoria ligada a la variable durante la ejecución

Las instrucciones de un programa acceden a la variable por su L-Valor

Tiempo de vida (lifetime) o extensión

Periodo de tiempo que existe la ligadura

El tiempo de vida es el tiempo en que está alocada la variable en memoria y el binding existe

Es desde que se solicita hasta que se libera

Alocación

Momento en que se reserva la memoria para una variable

Estática

Se hace en compilación (antes de la ejecución) cuando se carga el programa en memoria en zona de datos y perdura hasta fin de la ejecución (sensible a la historia)

Variables estáticas (constantes(?)

Dinámica

Se hace en tiempo de ejecución

Puede ser:

1. Automática: cuando aparece una declaración en la ejecución
2. Explícita: requerida por el programador con la creación de una sentencia, a través de algún constructor (por ej. algún puntero (new en pascal))

Persistente

Los objetos persistentes que existen en el entorno en el cual un programa es ejecutado, su tiempo de vida no tiene relación con el tiempo de ejecución del programa

Persisten más allá de la memoria

Ejemplo: archivos una vez creados/abiertos permanecen y pueden ser usados en diversas activaciones hasta que son borrados con un comando del sistema operativo (lo mismo sucede con las bases de datos)

R-valor

Es el valor codificado almacenado en la ubicación de la variable

R-Valor de una variable es el valor codificado almacenado en la locación asociada a la variable (l-valor)

La codificación se interpreta de acuerdo con el tipo de la variable

Ejemplo: 01110011 almacenado en una ubicación de memoria:

- Interpretado como nro. entero si la variable es tipo int (115)
- Interpretado como cadena si la variable es tipo char (s)

Objeto: (l-valor, r-valor) (dirección memoria, valor)

Se accede a la variable por el l-valor (ubicación)

Se puede modificar el r-value (valor) (salvo un caso especial)

Binding Dinámico

Binding dinámico de una variable a su valor:

- El valor (r-valor) puede cambiar durante la ejecución con una asignación
- El valor (r-valor) no puede cambiar si se define como constante simbólica definida por el usuario

b := a (copia el r-valor de a en el l-valor de b y cambia el r-valor de b)

a :=17 (asigna un valor directamente)

Constante

Se congela el valor

Ligadura estática

El valor que proporciona una expresión debe evaluarse en tiempo de compilación

El binding es el tiempo de compilación

El compilador puede sustituir legalmente el valor de la constante por su nombre simbólico en el programa

- const pi = 3.14 (está bien)
- int cant_items = 23
  const nro_paginas = 3 * cant_items (está mal, porque se usa el valor de una variable, el cual se va a resolver en ejecución, para definir el valor de la constante)

Ligadura dinámica

El valor se puede dar como una expresión que involucra otras variables y constantes, en consecuencia, el enlace se puede establecer en tiempo de ejecución, pero sólo cuando la variable es creada

Variables anónimas y referencias

Algunos lenguajes permiten que variables sin nombre sean accedidas por el r-valor de otra variable

Ese r-valor se denomina referencia o puntero a la variable

La referencia puede ser a el r-valor de una variable nombrada o el de una variable referenciada llamada access path de longitud arbitraria

Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable

Puntero

Variable que sirve para señalar la posición de la memoria en que se encuentra otro dato almacenado como valor, con la dirección de ese dato

Alias

Se da si hay variables comparten un objeto en el mismo entorno de referencia, y sus caminos de acceso conducen al mismo objeto

El objeto compartido modificado vía un camino se modifica para todos los caminos

Ventajas

Compartir objetos se utiliza para mejorar la eficiencia

Desventajas

Generar programas que sean difíciles de leer

Generar errores porque el valor de una variable se puede modificar incluso cuando no se utiliza su nombre

Sobrecarga

Un nombre está sobrecargado si en un momento referencia más de una entidad

Debe estar permitido por el lenguaje

No aplica a todos los lenguajes

Hay que tener suficiente información para permitir establecer la ligadura unívocamente (por ejemplo del tipo (los tipos en una suma permiten saber cuál de las funcionalidades del operador + se va a utilizar))