White paper: Lenguajes de programación

Autor: Ramiro A. Gómez
Sitio web: www.peiper.com.ar

Introducción

Las personas para comunicarse entre sí utilizan un lenguaje que puede ser oral o escrito. En general, para comunicar algo siempre se usa un lenguaje.

La informática no queda excluida del uso de lenguajes, ya que estos son la manera de especiaficar las acciones que se desea sean realizadas en la computadora.

En otras palabras, son la interfase entre el programador y la computadora. A través de ellos podemos desarrollar programas o aplicaciones, que se componen por un conjunto de instrucciones que luego se ejecutarán en la computadora haciendo uso de sus recursos (CPU, memoria, disco, etc.).

Los lenguajes de programación están destinados a distintos ambitos, dependiendo de sus características que simplifican algunas tareas y complejizan otras.

Pueden estar destinados a *aplicaciones científicas, aplicaciones de negocios, inteligencia artificial, programación de sistemas, scripting,* y también disponemos de *lenguajes de propósitos especiales.*

Los lenguajes de programación tienen una estructura compleja que se compone de varias partes: sintáxis, semántica, elementos del lenguaje, nivel de abstracción, paradigma, estructuras de control para ordenar la ejecución de los programas, tipos de datos (números, letras, etc.), y funciones o procedimientos (unidades) que contienen un conjunto de instrucciones, entre otras.

No hay un único tipo de lenguajes, sino que se clasifican según las características que posean y según el paradigma y conceptos que soporten.

- *Por su nivel*:

    - *Bajo nivel*. No hay abstracciones de datos ni de procesos. Ejemplos: Assembler, editores hexadecimales.
    - *Alto nivel.* Permite abstraer varios aspectos que simplifican la programación. En general son algo menos eficientes que los de bajo nivel. Ejemplos, Pascal, Ada, Java, C++, Prolog, etc.
- *Por su jerarquización*:
    - *Primera generación*. Fue el primer lenguaje. Se utilizan unos y ceros para representar los "cableados", que anteriormente se hacían a mano. Ej: Lenguaje máquina.
    - *Segunda generación*. Se asignan nombres simbólicos para las distintas instrucciones, que internamente son combinaciones de unos y ceros. Ej: Assembler.
    - *Tercera generación*. Lenguajes de alto nivel. Son los más populares y más fáciles de usar.
    - *Cuarta generación*. Lenguajes 4GL, o de gestión de bases de datos; como SQL, QueryByExample, etc.
- *Por el manejo de las instrucciones*:
    - *Imperativos*: Un programa se especifica por medio de una secuencia de instrucciones que se ejecutan de esa manera, en secuencia. Ej: FORTRAN.
    - *Orientados a objetos*: Soportan abstracciones de datos y procesos conocidas como objetos. Ej: C++, Ada, Java, Smalltalk.
    - *Funcionales*: Especifican una solución como un conjunto y una composición de funciones. Ej: Miranda, Haskell, ML, Camel.
    - *Lógicos*: Permiten obtener resultados lógicos o relaciones entre elementos. Ej: Prolog.
    - *Concurrentes, paralelos y distribuidos*: Soportan procesamiento paralelo, es decir, al mismo tiempo. Pascal Concurrente, Java (hilos).
- *Por la programación:*
    - *Imperativos:* Incluye los paradigmas imperativo y orientado a objetos.
    - *Declarativos:* Incluye los paradigmas funcional y lógico.

Características de los lenguajes de programación

Tenemos muchas características, pero en general las más deseables son que el lenguaje sea expresivo, legible y eficiente.
Otras las enumeramos a continuación.

- ✓ ***simplicidad*** : Aumenta la legibilidad y la facilidad de escritura, aunque demasiada simplicidad tiene el efecto contrario sobre la legibilidad. Aumenta la confiabilidad del software ya que al ser más sencillo, la verificación y detección de errores es más sencilla.
- ✓ ***estructuras de control***: Controlan el flujo de ejecución de los programas. Influyen en la legibilidad y en la facilidad de escritura. Aumentan el control que el programador tiene sobre un programa, y por lo tanto aumenta la confiabilidad. Ejemplos son las estructuras IF-THEN-ELSE, WHILE, FOR, etc.
- ✓ ***tipos y estructuras de datos***: son muy útiles ya que organizan la información de acuerdo a su tipo y en estructuras de datos convenientes. Los tipos y estructuras de datos aumentan la confiabilidad ya que es posible el chequeo de tipos.
- ✓ ***Diseño de sintaxis***: Determina la manera en que se combinan los símbolos y elementos de un lenguaje. Influye en la legibilidad y facilidad de escritura, en la confiabilidad y en los costos.
- ✓ ***Soporte para abstracción***: minimiza la complejidad de los problemas a resolver agrupandolos de acuerdo a ciertas características. Está comprobado que esta característica aumenta la legibilidad y facilidad de escritura así como la confiabilidad.
- ✓ ***Expresividad***: Se refiere a la naturalidad con la que un lenguaje expresa sus sentencias. Aumenta la legibilidad y la confiabilidad, y en general disminuye la facilidad de escritura y el costo de aprendizaje.
- ✓ ***Chequeo de tipos***: Impacta mucho en la confiabilidad ya que muchos programadores tienen tendencia a cometer errores de tipos (por ejemplo, cuando se necesita una matriz, usan un vector). El chequeo puede ser estático (en tiempo de compilación) o dinámico (durante la ejecución). El chequeo dinámico es más flexible pero produce sobrecarga durante la ejecución.
- ✓ ***Manejo de excepciones***: Aumenta la confiabilidad porque permite al programador definir el comportamiento que tendrá el programa ante tal o cual excepción. Es una

característica muy deseable, aunque sólo se encuentra disponible en los lenguajes más modernos. Ciertos lenguajes antiguos han incorporado el manejo de excepciones en sus versiones más nuevas.

Sintáxis de los lenguajes de programación

La sintaxis de un lenguaje de programación es la estructura en que se organizan los distintos elementos sintácticos, como espacios, identificadores, operadores, etc. Es decir el orden que tienen unos con respecto a otros.

Una sintaxis se evalúa según varios criterios: que sea fácil de leer, de escribir, de verificar (chequear en busca de errores), fácil de traducir y que carezca de ambigüedad. Esta última significa que un mismo código puede tener 2 o más interpretaciones o traducciones posibles, en cuyo caso no se puede decidir que alternativa elegir.

Los elementos de la sintáxis son: alfabeto, identificadores (nombres de variables, constantes, etc.), símbolos de operadores (+,-,etc.), espacios en blanco, delimitadores y corchetes, palabras pregonadas (palabras que se pueden omitir sin alterar el significado), palabras clave y reservadas (propias del lenguaje), comentarios y expresiones.

El principal problema de una sintáxis es cómo se la define. Para esto existen metalenguajes que permiten definir la manera en que se combinan los símbolos y elementos. Estos metalenguajes o notaciones formales son un conjunto de reglas que especifican el modelo de construcción de las cadenas aceptadas por el lenguaje. Es decir que un metalenguaje es un lenguaje que define a un lenguaje de programación. Sus elementos son símbolo distinguido, metasímbolos y metavariables.

- ***Símbolo distinguido***: Punto de partida para la generación de todas las cadenas.
- ***Metasímbolos***: (| (o), ::= (es), <metavariable>).
- ***Metavariables***: Pueden ser terminales o no terminales.
    - ***Terminales***: Son palabras que forman los símbolos no terminales.
    - ***No terminales***: Son identificadores que tienen un definición hecha con más metavariables, de manera que es posible reemplazarlas por sus definiciones.

Otra manera de llamarlo a este tipo de gramáticas es “gramáticas BNF” (Backus Naur Form), que son un metalenguaje que define

las reglas de formación de las cadenas aceptadas por un lenguaje de programación. Fueron inventadas por Backus Naur para el desarrollo del lenguaje de programación Algol 60.

Ejemplo: **<expresión> ::= {<suma>}+ <multiplicación> | {<suma>}+ <división>**

Existe una representación visual de una gramática BNF, que son los grafos sintácticos. Esots usan flechas, círculos y rectángulos. Sus ventajas son que son más fáciles de entender para el ojo humano.

Una vez que tenemos definada la sintáxis de nuestro lenguaje, el compilador deberá determinar si las cadenas de texto ingresadas por los usuarios son válidas de acuerdo a esta sintáxis. Para esto se utilizan árboles de análisis sintáctico y algoritmos de parsing.

Semántica de los lenguajes de programación

La semántica de un lenguaje de programación se refiere al significado que adoptan las distintas sentencias, expresiones y enunciados de un programa. La semántica engloba aspectos sensibles al contexto, a diferencia de la sintaxis que considera aspectos libres de contexto.

Los principales elementos de la semántica son:

- ***variables***: se refieren a locaciones de memoria ligadas a un nombre y a un tipo.
- ***valores y referencias***: los valores son el estado de determinada celda o grupo de celdas de la memoria, mientras que las referencias indican la posición de esa celda en memoria. Estos 2 conceptos están muy involucrados con los punteros. La mayoría de los lenguajes los soportan, aunque son una conocida fuente de errores de programación.
- ***expresiones***: Son construcciones sintácticas que permiten combinar valores con operadores y producir nuevos valores. Son los elementos de los que se componen los enunciados. Las expresiones pueden ser aritméticas (a +b*c), relacionales (a <= b && c > a), lógicas (a && b || c) o condicionales (if (a*b > 2)...). Cada una de estas tiene una semántica especifica que la define. Por ejemplo en una expresión aritmética el tipo esperado es numérico (esto es int, long, etc.), los operadores deben ser +,-,*, /; y las funciones utilizadas dentro de ésta deben retornar valores numéricos.

Semántica estática. Gramática de atributos: Las gramáticas de atributos son más poderosas que las BNF porque permiten formalizar aspectos sensibles al contexto.
Por ejemplo, el chequeo de tipos depende del contexto porque debemos saber el tipo esperado y el actual y determinar si son compatibles. El tipo esperado lo obtenemos del contexto analizando la definición de la variable. Si la variable num1 está definida de tipo String, y más adelante le queremos asignar el valor 1000, se producirá un error de tipo.
La gramática de atributos de compone de:

- ***atributos***:
  - ***heredados***: son los que se obtienen de un elemento más general. Esquema de arriba hacia abajo. Por ejemplo, number y float. En este caso float heredará los atributos de number e incorporará nuevos atributos propios de su tipo.
  - ***sintetizados***: se conocen a partir de los sub-elementos, como ser un subrango. Esquema de abajo hacia arriba, opuesto a los atributos heredados.
  - ***intrínsecos***: Estos atributos se obtienen de tablas externas, fuera del árbol de derivación.
- ***condiciones***: Son hechos que se evalúan si suceden o no. Por ejemplo, if(num>=20) ...
- ***reglas***: A partir de un conjunto de condiciones se forman las reglas. Por ejemplo:

```
if (tipo(actual) == tipo(esperado)) {
        analisisBien = true; }
else {
        analisisBien = false;
}
```

Elementos de los lenguajes de programación

Presentaremos a continuación varios de los elementos que conforman un lenguaje de programación.
Entidades: Son los elementos sobre los que trabaja un programa. Pueden ser subprogramas, variables, rótulos y sentencias. Tienen atributos, a los que se asocian a través de un mecanismo llamado ligadura.

Atributos: nombre, valor, tipo, locación de memoria, tiempo de vida, alcance y alias. Estas son las características de cada una de las entidades, es decir que un subprograma tendrá un nombre, una locación de memoria, tendrá un valor (el valor que retorna), tendrá un alcance, etc.

Ligaduras: Se refiere a asociar un atributo con una entidad. Las entidades ligables son subprogramas, variables y constantes, tipos, parámetros formales, excepciones y etiquetas. Las ligaduras se pueden producir en distintos momentos, como en la compilación o en la ejecución.

Tiempos de ligadura: Según el momento en que se realicen se las llama estáticas o dinámicas.

- ***Ligadura estática***: Son menos flexibles, pero más seguras, detección temprana de errores y permiten una mayor eficiencia. Se produce en lenguajes orientados a la compilación.
- ***Ligadura dinámica (en tiempo de ejecución)***: son más flexibles pero menos seguras. Mayor costo de ejecución. Se produce en lenguajes orientados a la ejecución. Corresponden a las que no se pueden realizar en un paso anterior y necesitan del programa en ejecución para hacerlas. Un ejemplo es la asignación de un valor a un nodo de un árbol. Como antes de la ejecución no se conoce la locación de memoria en la que se almacena el nodo, es imposible hacer la asignación. En este caso el atributo de locación se conoce sólo en tiempo de ejecución.

Declaraciones: Son sentencias que crean una ligadura entre un tipo, un nombre y un valor. A veces el valor se puede omitir, por ej: String suNombre.

Tenemos 2 tipos de declaraciones.

- ***declaraciones implícitas***: ocurren cuando se usa por primera vez la variable. Son muy inseguras y pueden provocar errores difíciles de detectar. Un ejemplo es FORTRAN, en el que las variables que empiezan con i son de tipo entero.
- ***declaraciones explícitas***: se usa el tipo de manera explícita.
  - *secuenciales*: usan en ellas declaraciones previas. Por ejemplo, en Pascal podemos definir un tipo sub-rango que va de 0 a 365, y a este lo usamos como tipo de los elementos de un tipo arregloAño que definamos.

- ***recursivas***: se las utiliza para definir estructuras dinámicas como árboles, grafos y listas. Se llaman recursivas porque la definición se incluye a sí misma.

Ambiente de referenciamiento: Es el conjunto de entidades que son visibles o pueden ser referenciadas en una parte determinada del código. El uso de una entidad fuera del ambiente de referenciamiento al que pertenece produce un error. El lenguaje C dentro de UNIX provee un mecanismo novedoso para pasar ambientes de referenciamiento de un lugar a otro, y se llama "conductos".

Alcance y visibilidad: Muy relacionadas con ambientes de referenciamiento. Una entidad es visible dentro de su alcance. Y el alcance es hasta donde una entidad puede ser referenciada.

Estructura general de un programa: Tenemos 3 tipos de estructuras que puede tener un programa, y cada una se corresponde con determinados lenguajes.

- ***Estructura monolítica***: Se compone de un programa principal y variables globales. Las declaraciones, ambiente y alcance son globales. Es la estructura más antigua y menos flexible. La incorporan lenguajes como COBOL. Aumentan los costos de mantenimiento y hacen tediosa esta labor.
- ***Estructura en bloques chatos***: Las declaraciones son globales y en varios subprogramas. El ambiente es global y propio de cada subprograma y el alcance dentro de un subprograma es a este y al ambiente global. Es decir que no está permitido acceder desde un subprograma al ambiente de referenciamiento de otro. La incorporan lenguajes como C++ y Java.
- ***Estructura en bloques anidados***: Similar a la de bloques chatos pero es posible definir subprogramas dentro de subprogramas. Más nivel de profundidad. Las declaraciones son globales y locales a cada subprograma. El alcance es dentro de cada subprograma e incluye los subprogramas de nivel más alto. El ambiente incluye cada subprograma, los subprogramas de nivel más alto y el ambiente global.

Datos: desde el punto de vista informático, es un hecho que se representa en la computadora. Los programas manipulan datos con el fin de producir información. Por ejemplo, una base de datos almacena gran cantidad de datos y debe asegurar su

consistencia. Un programa contable registra y procesa datos de ventas, compras y movimientos de dinero. Es decir, los datos son la entrada de los procesos o subprogramas.

Variables y constantes: Los datos se almacenan en variables y constantes. Las variables son identificadores que poseen una locación de memoria, un nombre, un tipo, etc. Pueden cambiar su valor durante la ejecución del programa. Las constantes no pueden cambiar su valor y se las define en tiempo de implementación del programa. Pueden ser literales o definidas (por el programador).

Atributos de las variables y constantes:

- ***nombre***: el nombre es la cadena de caracteres que está asociada con una posición de memoria. De esta manera se logra mejorar la legibilidad (al descartar el uso de la memoria a través de números).
- ***valor***: es el estado que contiene cada variable o constante. Este estado se representa con secuencias de bits. El dominio de valores que puede tomar una variable se determina por su tipo. Por ejemplo, si una variable es de tipo booleano los valores que podrá tomar serán true y false, y ningún otro. La correspondencia entre los bits y los valores que representan se definen en tiempo de definición del lenguaje (para el caso de los tipos primitivos).
- ***tipo***: el tipo es muy importante porque permite hacer verificación de tipos. El tipo indica el dominio de valores que podrá tomar la variable o constante. Dentro de los tipos tenemos los numéricos enteros y flotantes (int, long, double, etc.), los caracteres y los booleanos. Hay casos en los que también se incluyen dentro del lenguaje el tipo "fracción" y el tipo "número imaginario".
- ***locación de memo***ria: es la posición de memoria que representa la variable o constante. En la mayoría de los lenguajes la asignación de ésta es automático. Menos en lenguaje máquina, assembler y similares. Puede ser estática, basada en pila o dinámica (heap).
- ***alcance***: una variable es visible dentro de su alcance. Puede definirse estáticamente (en base a la posición en que se encuentra dentro del código) o dinámicamente (según la secuencia de ejecución de los subprogramas). Por ejemplo, decimos que el alcance de una variable declarada en un subprograma tiene un alcance local y global, o sea que puede ser referenciada desde el

ambiente global y desde el subprograma, pero no se puede desde otro subprograma.

- ***tiempo de vida***: se refiere al intervalo de tiempo durante el que la variable o constante existen. Generalmente una variable local tendrá un tiempo de vida igual al que se ejecuta el subprograma en el que está definida.
- ***alias***: muchas veces puede ser útil disponer de más de un nombre para una variable. Con el uso de alias pueden surgir problemas con respecto a las referencias.

Expresiones: Son construcciones sintácticas que a partir de valores y operadores calculan nuevos valores. Sus componentes son operadores y operandos. Se las usa para modificar el estado de las variables. En general se las asigna a variables. Mediante el uso de expresiones se pueden realizar cálculos matemáticos, lógicos o relacionales.

- *Clasificación*
  - ***aritméticas***: calculan el resultado de hacer operaciones de suma, resta, multiplicación y división entre varios valores. Pueden ser unarias, binarias o ternarias. Generalmente estas expresiones son las que se evalúan primero. El resultado de su evaluación es numérico, y a su vez se pueden usar en una nueva expresión.
  - ***relacionales***: se usan para determinar si un valor es mayor, menor, mayor o igual o menor o igual que otro valor. El resultado de la evaluación de estas expresiones es un valor lógico (verdadero o falso). En general, estas expresiones se evalúan después de las aritméticas.
  - ***lógicas***: calculan un valor lógico a partir de otros valores del mismo tipo. Los operadores de estas expresiones son "y" (and, &&), "o" (or, ||) y "no" (not, !). Se evalúan después de las relacionales.
  - ***condicionales***: se usan para ejecutar un código u otro de acuerdo a la evaluación lógica de estas, que pueden ser una combinación de distintos tipos de expresiones. El resultado final de esta combinación tiene que devolver un valor booleano, verdadero o falso.

Estructuras de control a nivel expresión: Tenemos estructuras de control en varios niveles. Estos son a nivel expresión, a nivel sentencia y a nivel unidad. Gracias a las estructuras de control podemos manejar y controlar la ejecución de un programa y la evaluación de los operandos.

- ***Implícitas***: son reglas que tiene el lenguaje que no son visibles. Están documentadas en el manual de referencia de cada lenguaje.
  - ***precedencia***: los operadores se encuentran clasificados en niveles, y según en el que estén se evaluarán primero o no. Por ejemplo los operadores aritméticos están en un nivel más bajo que los lógicos.
  - ***asociatividad***: dentro de cada nivel también hay un orden, según el cual se evalúan los operandos.
- ***Explícitas***: se determina por medio de los símbolos de paréntesis el orden que debe seguir la evaluación de una expresión.
  - ***uso de paréntesis***: si deseamos que la expresión se evalúe en un orden distinto al que indican las reglas de precedencia y asociatividad, es posible usar paréntesis. El uso indiscriminado de paréntesis compromete la legibilidad del código. Ejemplo: sin paréntesis a+b*c. Acá se evalúa b*c y luego se le suma a. En cambio con paréntesis (a +b)*c se evalúa a+b y a este resultado se lo multiplica por c.

Tipos de datos

Un tipo es un atributo que poseen las variables (y objetos) y que permite definir el dominio de valores que puede tomar cada una. También permiten el chequeo de tipos para detectar errores de programación.

Tenemos 3 grandes grupos de tipos: los tipos predefinidos y definidos por el usuario, los tipos estructurados y los dinámicos (que usan punteros).
Los tipos predefinidos y definidos por el usuario representan los numéricos flotantes y enteros, los caracteres y los booleanos (valores verdadero o falso). Entre otros tipos extendidos tenemos los fraccionales y los numéricos imaginarios.

Los tipos estructurados almacenan información con distintas características según la estructura.

Estructuras estáticas

Tipos primitivos:

- *Tipos numéricos*:
  - ***enteros***: se representan en memoria como una secuencia de bits que de acuerdo a su posición y valor (0 o 1) simbolizan el valor.
  - ***flotantes***: existen 2 implementaciones básicas, flotantes de 32 y 64 bits. Se valor se representa por 3 partes que son el signo, el exponente y la mantisa. En los de 32 bits, el exponente ocupa 8 bits y el resto la mantisa y el signo. En los de 64 bits el exponente ocupa 11 bits.
- ***carácteres***: Cada secuencia de bits tiene un símbolo asociado. Originalmente ocupan 8 bits como representación ASCII. Luego la representación se extendió a 16 bits como Unicode para almacenar más cantidad de símbolos (de otros idiomas). Esta representación facilita la internacionalización.
- ***booleanos***: son un tipo relativamente nuevo. Se desperdicia espacio de almacenamiento porque los valores verdadero y falso se pueden representar con un solo bit, pero el mínimo permitido en los sistemas actuales son 8 bits. Posibilidad de empaquetamiento para vectores de booleanos. Permiten operaciones lógicas. Aportan mayor legibilidad al código y más confiabilidad (menos posibilidad de confusiones).

Tipos definidos por el usuario:

- ***tipos enumerados***: se pueden representar internamente con valores numéricos secuenciales, aunque las enumeraciones sean palabras u otros datos.
- ***sub-intervalos***: al ser menos cantidad de valores, necesitan menos espacio de almacenamiento. Posibilidad de empaquetarlos.

Tipos estructurados:

- ***arreglos***: almacenan tipos iguales. El acceso a sus elementos es, en general, por posición. Los arreglos pueden ser estáticos o dinámicos, y redimensionables o no. Los arreglos se almacenan de manera continua en memoria, salvo que sean redimensionables.
- ***registros***: Pueden almacenar distintos campos dentro de sí, incluidos más registros y arreglos. Son muy útiles para

mejorar la legibilidad y facilidad de escritura. En Pascal y en Ada también existen los registros variantes. En C se usa la palabra reservada "struct" para definirlos.

- ***unión***: es una locación de memoria asociada a muchos tipos de datos, o sea que puede ser interpretada de distintas maneras. La provee el lenguaje C.
- ***conjunto***: es una estructura de datos con muchos elementos distintos, es decir que no los hay repetidos. Los conjuntos permiten hacer operaciones de unión, intersección y diferencia entre ellos.

Estructuras dinámicas: son ideales para modelar situaciones de la vida real, con capacidades excepcionales de flexibilidad.

- ***listas***: almacenan elementos de distinto tipo. Son redimensionables e ideales para insertar y eliminar elementos. Pueden ser simple o doblemente enlazadas. Se implementan con una referencia al siguiente elemento, que puede estar en cualquier locación de memoria. Con 2 referencias, una direcciona al elemento anterior y otra al siguiente. Óptimas para recorrerlas de arriba hacia abajo y viceversa.
- ***árboles***: cada nodo almacena 2 o más apuntadores a sus nodos hijos. Tenemos muchos tipos de árboles, como árboles binarios, B+, B*, etc. Además pueden tener una referencia al nodo padre (uno solo) y esto permite recorrer el árbol de abajo hacia arriba.
- ***Grafos***: cada nodo almacena una o varias referencias a los nodos con los que está conectado. Si el grafo es dirigido (con dirección de lectura) hay que proyectar almacenar esta información. Los grafos son útiles en aplicaciones que usan mapas de ciudades, sistemas operativos, videojuegos, etc. Requieren algoritmos especiales para recorrerlos ya que es una de las estructuras dinámicas más complejas.

Chequeo estático y dinámico de tipos: Los chequeos estáticos comprueban la consistencia de tipos en tiempo de compilación, es decir, estáticamente. Es más seguro porque no genera el ejecutable hasta que no haya ningún error de tipo. Es menos flexible que el dinámico. Los lenguajes con chequeo estático son fuertemente tipados, esto es que el chequeo se realiza de una manera exhaustiva.

El chequeo dinámico, por su parte, es más flexible pero mucho menos seguro y confiable que el estático. Aumenta la sobrecarga

durante la ejecución. El chequeo se realiza en tiempo de ejecución.

Abstracción de datos: La abstracción de datos permite dividir un gran problema en problemas menores para simplificar, definir interfaces de acceso para la manipulación del dato, y oscurecer la implementación de forma de hacerla transparente para el usuario.
Una manera de lograr cierto nivel de abstracción son los "**tipos genéricos".** Son abstracciones de datos que permiten ser usados en distintos contextos y usar una misma implementación para distintos tipos y estructuras de datos. Ej: En Java están implementados con el uso de la clase Object, Number, Container, etc. Ahorra trabajo y esfuerzo al programador, a la vez que simplifican la implementación.
Los **"tipos parametrizados"** son otra manera de generar abstracción**..** Se construyen en base a parámetros.

Polimorfismo: El polimorfismo está basado en el concepto de la manipulación de entidades de distinto tipo. Un lenguaje polimórfico es aquel que sus entidades pueden estar ligadas a más de un tipo. Una función es polimórfica si acepta operandos de distintos tipos. Y un tipo es polimórfico si sus operaciones son polimórficas.

Coerción, conversión y compatibilidad: La coerción es la conversión automática de un tipo a otro. Ocurre cuando el tipo en cuestión es un subtipo de otro (o una subclase en el paradigma orientado a objetos). El principio que gobierna las coerciones es no perder información. Las conversiones son transformaciones del tipo de una entidad a otro. Por ejemplo, en muchos lenguajes las asignaciones de enteros de 32 bits a enteros de 16 bits requieren la acción del programador porque se puede perder información (no hay manera de representar el rango de 32 bits con sólo 16).

Inferencia de tipos: en algunos lenguajes como ML es posible la inferencia de tipos. Al no especificarse el tipo de un dato, es inferido a partir del uso que se hace de él. Es decir, de las funciones que lo manipulan. Ejemplo en ML: fun resto(n) = n mod 2.
En esta función el tipo del parámetro no está indicado, pero como la función mod se aplica únicamente a tipos enteros se

infiere este tipo para “n”. En algunos casos es muy difícil hacer la inferencia debido al alto polimorfismo de la función usada (a la cantidad de tipos que acepta). Por ejemplo en fun (a,b) = a+b; es difícil determinarlo porque “+” puede sumar números o concatenar cadenas de texto.

Instrucciones

Generalmente disponemos de 2 tipos de instrucciones, asignación y de control.

La asignación puede ser simple, múltiple o condicional. Se puede realizar en las declaraciones. Algunos lenguajes permiten asignación múltiple, mientras que otros sólo admiten la simple.

Por su parte, así como disponemos de estructuras para controlar la evaluación de las expresiones y la ejecución de las unidades, también las tenemos para controlar instrucciones.

Estructuras de control a nivel instrucción:

- ***Secuencia***:: Es la ejecución en orden, una sentencia después de la anterior. Se puede modificar con saltos incondicionales como el GOTO, aunque el uso excesivo de este altera demasiado la secuencialidad del código y lo hace ilegible.
- ***Selección***:: Probablemente es una de las primeras estructuras usadas. Permiten bifurcar o seleccionar el código a ejecutar de acuerdo al valor lógico que resulta de evaluar una condición. Generalmente se implementa con la palabra reservada “if...else...”.
- ***Iteración***:: La iteración evita repeticiones de código innecesarias. Disponemos de iteraciones condicionales e incondicionales. Las primeras consumen más tiempo que las segundas debido a que se debe evaluar la condición en cada ciclo. Por su parte, las incondicionales se usan para recorrer vectores, matrices, etc.

Unidades

Los subprogramas se dividen en dos categorías, procedimientos y funciones. La diferencia entre estos es que los procedimientos no retornan valores, sino que realizan una acción. Por ejemplo, procesarArreglo(int[ ] ...). Las funciones devuelven un valor que puede ser un tipo primitivo, y en algunos en lenguajes modernos pueden ser también objetos.

Un subprograma tiene 3 partes básicas, a saber: inicio, suspensión y control. El inicio puede producirse por una llamada

desde el programa principal o desde otro subprograma. La suspensión se produce con la invocación de un subprograma. Cuando un programa o subprograma la hace, se suspende temporalmente mientras se ejecuta el subprograma invocado. Esto pasa porque generalmente es necesario procesar un conjunto de datos antes de hacer otro proceso, y no se puede comenzar el segundo sin la totalidad de los primeros datos procesados. Por otra parte, hay distintas maneras de controlar un subprograma, que pueden ser implícitas (excepciones) o explícitas (relación jerárquica, simétrica o paralela).

Pasaje de parámetros

Las unidades, ya sean procedimientos o funciones, aceptan datos de entrada. Una de las maneras para hacrlo es a través del pasaje de parámetros. En la mayoría de los lenguajes tiene un forma:

funcion(parámetro1, ..., parámetroN)

procedimiento(parámetro1, ..., parámetroN)

Hay varias formas de utilizar los parámetros pasados a una unidad. Ellas son:

- ***in mode***: la semántica es de ida pero no de vuelta. El parámetro se usa como datos de entrada pero no como de salida. En general lo especifica el programador según el uso que haga del mismo.
- ***out mode***: la semántica es de vuelta, pero no de ida. El parámetro se usa como datos de salida pero no de entrada. O sea que se escribe en el parámetro pero no se obtienen datos de él.
- ***in out mode***: Es una combinación de los dos modelos anteriores. Se obtienen datos del parámetro y también se escriben en él. En el caso específico de Ada, existen restricciones para el pasaje de parámetros in out mode en las funciones. Un ejemplo: ordenarYdarSuma(int[] ...). Esta función procesa los datos del arreglo para obtener su suma y lo ordena.

- ***por copia***: se copian los valores de los parámetros. Sus ventajas son que permite el paso de expresiones y la protección del parámetro real. Su desventaja es que es costoso en el caso de datos de gran tamaño.
- ***por referencia***: se copian las referencias a memoria en vez de copiar todo el dato de una locación a otra. Esto permite ahorrar mucho tiempo porque sólo se copia una referencia. Sus ventajas: son flexibles y eficientes tanto

en tiempo como almacenamiento (uso eficiente de memoria). Su desventaja son los efectos colaterales que se pueden producir y el acceso a memoria (que es más lento que el procesador). Una variante del pasaje de parámetros por referencia es por nombre. En su caso se evalúa el parámetro cada vez que se usa.

Hay que recalcar que algunos lenguajes tienen la capacidad de pasar procedimientos como parámetros, y no sólo datos.

Estructuras de control a nivel unidad: Existen mecanismos implícitos o explícitos. Los primeros se activan por eventos que surgen de la ejecución del programa. Los segundos estructuran la ejecución de acuerdo a qué categoría pertenezcan.

- ***Mecanismos implícitos***
  - ***Excepciones***: permiten al programador especificar el comportamiento del programa ante eventos anómalos como división por cero, desbordamiento de pila, error de lectura, etc.
- ***Mecanismos explícitos***
  - ***Relación jerárquica***: se basa en suspender momentáneamente la ejecución cuando se hace una invocación a fin de permitir procesar el subprograma invocado. Son las estructuras de control a nivel unidad más comúnmente utilizadas. Útil en esquemas no concurrentes.
  - ***Relación simétrica***: provisto en general por las corrutinas. Se comienza a ejecutar una, luego de algunas instrucciones se pasa a ejecutar la segunda, después se vuelve a la primera, etc. Se conmuta la ejecución alternadamente entre varias corrutinas. Útil para implementar concurrencia como hilos.
  - ***Relación paralela***: se provee a través de las tareas. La ejecución es concurrente, se ejecutan al mismo tiempo. Los procesos pueden ser disjuntos (no intercambian datos), cooperativos (intercambian datos y colaboran en resolver un problema) o competitivos (compiten por el uso de los recursos y datos). Problemas del "abrazo mortal" y de la "inanición".

Intérpretes y compiladores

Los lenguajes tienen 2 maneras de "entender" un código escrito por los programadores. La primera es interpretando, las segunda es compilando. También existe una tercera forma que se denomina híbrido, que consiste en compilar y luego interpretar.

- ***Intérpretes***: procesan el código fuente en el momento de ejecución o carga. Son más lentos que los compiladores pero permiten portabilidad. Su estructura básica se compone de un código fuente y datos de entrada.
- ***Compiladores***: hacen la traducción a código máquina o similar antes de la ejecución. Por eso son muy eficientes y rápidos. Por ejemplo, Pascal, Ada, C++, C, etc. son compilados. Hay varias fases o etapas de compilación. Si se privilegia la velocidad de ejecución se aumentan las fases, y si se privilegia la velocidad de compilación se puede implementar en un paso o dos. Las fases son:
  - ***análisis léxico***: se procesan los símbolos del código fuente uno por uno, y se los agrupa en distintos elementos sintácticos como palabras reservadas, espacios en blanco, comentarios, etc. Es uno de los pasos que más tiempo consume.
  - ***análisis sintáctico***: se comprueba la correctitud del código a nivel sintáctico; es decir que satisfaga las reglas de formación sintáctica del lenguaje. Como entrada tiene los elementos sintácticos obtenidos de la etapa de análisis léxico. Se utiliza el árbol de análisis sintáctico. Hay distintos algoritmos de análisis sintáctico.
  - ***análisis semántico***: una vez terminada la etapa de análisis sintáctico y el código es "sintácticamente correcto" se evalúa el aspecto semántico en base a las reglas semánticas del lenguaje. Si el chequeo de tipos es estático, se hace en esta fase. En esta etapa se genera parte del código ejecutable, aunque puede ser ineficiente. La salida es el código objeto intermedio.
  - ***optimización***: usa el código objeto intermedio de la etapa anterior y como puede ser muy ineficiente, lo optimiza. Por ejemplo, si una misma variable es leída dos veces de memoria en instrucciones cercanas, se la almacena en

memoria caché. La ejecución se puede acelerar un 50% en algunos casos.

- ***vinculación y carga***: se vinculan las distintas unidades y subprogramas compilados en un solo código. También se pueden manejar referencias a unidades independientes, a fin de mejorar la eficiencia y evitar la copia innecesaria de código ejecutable. Como salida de este etapa tenemos el programa listo para ser ejecutado.

- ***Híbridos***: hacen uso de la compilación y de la interpretación. El código se compila a un código intermedio para obtener eficiencia, y éste se interpreta luego. Lenguajes como Java son híbridos para asegurar la portabilidad, y tienen la filosofía "se compila una vez, se ejecuta en todos lados".

Implementación de instrucciones y unidades

Registros de activación: Son bloques de datos y código que contienen información de la ejecución de cada unidad. Es decir que cada unidad está representada por un registro de activación y un segmento de código. Cada vez que se invoca un subprograma o unidad se debe alocar su correspondiente registro de activación, estructura cuya instancia almacena información perteneciente al programa o subprograma y a su ejecución. Los componentes de un registro de activación en un esquema estático son: parámetros, variables locales, valor de retorno y dirección de retorno. Un esquema basado en pila además de estos componentes agrega enlace estático (dentro de que unidad se encuentra definida en el código) y enlace dinámico (que unidad la invocó).

Esquemas de alocación de memoria:

- ✓ ***esquema estático***: no permiten recursividad, se reserva espacio para todos los registros de activación, aunque algunas unidades no se usen. Esquema ineficiente en memoria pero eficiente en cuanto a ejecución. Cada unidad está ligada a una única instancia de registro de activación. Muy sencillo de implementar. Lo usan lenguajes antiguos.
- ✓ ***esquema basado en pila***: basado en una estructura de datos tipo pila. Permiten recursividad, el espacio se

reserva a medida que se ejecutan las unidades. Esquema muy flexible y eficiente en cuanto a uso de memoria y ejecución. Presente en la mayoría de los lenguajes modernos. Cuando finaliza la ejecución de la unidad, el espacio de su instancia de registro de activación se libera. Un ejemplo es el cálculo recursivo del factorial.

- ✓ ***esquema dinámico***: se usa un heap, que es una porción de memoria usada dinámicamente cuyo sentido de crecimiento es opuesto al de la pila. Permite el uso de estructuras de datos dinámicas como árboles, listas, etc. Esquema muy flexible y eficiente, aunque hay cierto costo asociado a la alocación dinámica.

Autor: Ramix (Ramiro A. Gómez)