Paradigmas de Programación

Elementos de la programación

La programación imperativa, al igual que los demás tipos de programación, posee elementos y características que son importante destacarlos, como por ejemplo valores y tipos, expresiones, variables, comandos, enlaces etc.

Valores y tipos

Llamaremos valor a algo que pueda se evaluado, almacenado, incorporado a una estructura de datos, pasado como argumento a un procedimiento o función, retornado como resultado de una función etc. En otras palabras, un valor es cualquier entidad que existe durante una computación.

Por ejemplo en Pascal encontramos los siguientes valores:

Valores primitivos (valores de verdad, caracteres, enumerados, enteros y números reales)
Valores compuestos (registros, arreglos, conjuntos y archivos)
Punteros
Referencias a variables
Abstracción de procedimiento y función.

Los tres primeros grupos son evidentes que son valores, sin embargo los otros dos también son considerados como valores porque pueden ser pasados como argumentos. Englobaremos con el término abstracción a las funciones y procedimientos y a cualquier otra entidad similar en otro lenguaje.

Los valores pueden ser agrupados en **tipos**, como sugiere la lista anterior. Como ejemplo podemos decir que siempre se hace una distinción entre valores de verdad y los números enteros, debido a que en los enteros se pueden hacer operaciones de suma y multiplicación que no se pueden hacer con los valores booleanos.

¿Qué es exactamente un tipo? La respuesta más obvia, es quizás, que un tipo es un conjunto de valores. Cuando decimos que *v* es un valor del tipo T, significa simplemente que $v \in T$. Cuando decimos que una expresión E es del tipo T, significa que el resultado de evaluar la expresión E dará un valor de tipo T.

Sin embargo, no todos los conjuntos de valores pueden decirse que pertenecen a un mismo tipo. Debido a que todos los valores de un tipo deben exhibir un comportamiento uniforme bajo operaciones asociadas a ese tipo. Por ejemplo el conjunto {23, verdadero, Lunes} no es un tipo; pero {verdadero, falso} si lo es porque exhibe un comportamiento uniforme bajo las operaciones lógicas de negación, conjunción y disyunción; y {..., -2, -1, 0, 1, 2, ...} también lo es, porque todos sus valores exhiben un comportamiento uniforme bajo las operaciones de suma, multiplicación, etc.

Vemos entonces que un tipo se caracteriza no solo por el conjunto de valores sino también por las operaciones sobre esos valores.

Todos los lenguajes de programación poseen tanto *tipos primitivos*, los cuales son valores atómicos, como *tipos compuestos* cuyos valores están compuestos por valores simples. Algunos lenguajes tienen también *tipos recursivos*, los cuales están compuestos por otros valores del mismo tipo.

Tipos primitivos

Los tipos primitivos son aquellos que están compuestos por valores atómicos y por lo tanto no pueden ser descompuestos en valores más simples.

Estos tipos pueden variar con cada lenguajes dependiendo de la especialización del mismo, ya sea comercial (COBOL) o científico (Fortran) etc.

Seguiremos la siguiente notación para los tipos primitivos:

Valor_de_verdad `{verdadero, falso}`

Entero	`{..., -2, -1, 0, 1, 2, ...}`
Real	`{..., -1.0, ..., 0.0, ..., 1.0, ...}`
Carácter	`{..., 'a', 'b', ..., 'z', ...}`

En Pascal y Ada se puede definir un tipo primitivo completamente nuevo enumerando sus valores (más precisamente, enumerando identificadores que denotarán sus valores). Este tipo de datos se llama *tipo enumerado*.

```
Meses    {ene, feb, mar, abr, may, jun, jul, ago, sep, oct, nov, dic}
```

En algunos lenguajes se pueden definir subconjuntos de un tipo existente, obteniendo un *tipo subrango.* Por ejemplo el tipo subrango 28..31 tiene los valores {28, 29, 30, 31} y es un subconjunto de enteros, el cual debe estar compuesto por un conjunto de valores consecutivos.

Un punto interesante es la cardinalidad de un conjunto (o de un tipo). Escribiremos #S para significar el número de valores distintos en S. Por ejemplo:

#valor_de_verdad	= 2
#meses	= 12
#enteros	= 2 x maxint + 1 (en Pascal)

Tipos compuestos

Un tipo compuesto (o tipo de datos estructurado) es un tipo cuyos valores están compuestos o estructurados a partir de valores más simples. Los lenguajes de programación soportan una amplia variedad de estructuras de datos: tuplas, registros, arreglos, conjuntos, strings, listas, arboles, archivos secuenciales, archivos directos, etc. Todas estos tipos pueden ser estudiados a partir de un pequeño número de conceptos:

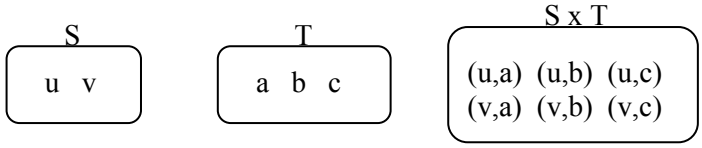
- Producto cartesiano. (tuplas y registros)
- Uniones disjuntas. (variants y uniones)
- Mapeo (arreglos y funciones)
- Conjunto Potencia (conjuntos)
- Tipos recursivos (estructuras de datos dinámicas)

Cada lenguaje de programación tiene su propia notación para definir tipos compuestos. Introduciremos una notación matemática simple y estándar.

Producto Cartesiano

Un tipo simple de composición de valores es el Producto Cartesiano, donde los valores de dos tipos (posiblemente diferente) son apareados. S x T es el conjunto de todos los pares ordenados de valores, donde el primer valor de cada par se toma del conjunto S y el segundo del T. Formalmente:

$S \times T = \{ (x,y) / x \in S; y \in T\}$



La cardinalidad del producto cartesiano es #(S x T) = #S x #T
Entendiéndose por cardinalidad de un tipo a la cantidad de valores del mismo.

Se puede extender la idea de pares ordenados a tripletes, cuadrupes etc.

Podemos tomar como ejemplo de producto cartesiano la definición de un registro:

```
Type   Fecha :    record
```

```
        m : meses;
        d : 1..31;
        end;
```

El tipo `Fecha` tiene el siguiente conjunto de valores {ene, feb, ..., dic} x {1, 2, 3, ..., 31}. O sea 372 pares de la forma (ene, 1) (ene, 2) ... (feb, 1) ... (dic, 31).

Un caso especial de producto cartesiano es donde todos los componentes de las tuplas son tomados del mismo conjunto S. Las tuplas se dicen homogéneas. Escribiremos:

$S^n = S \times S \times ... \times S$ (n veces)

En este caso la cardinalidad estará dada por $\#(S^n) = \#(S)^n$

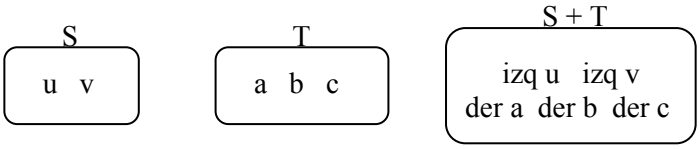
Finalmente consideremos un caso especial donde n = 0. La ecuación anterior nos dice que S^0 tendrá un único valor. Este valor es la 0-tupla(). Una tupla sin ningún componente. Este tipo será util para definir un tipo con un único valor:

Unit = { () }
Se corresponde con el *unit* de ML y *void* de algol-68 y C. Unit no es un conjunto vacío, contiene una única tupla que no contiene ningún componente.

Uniones Disjuntas

Otro tipo de composición de valores son las uniones disjuntas donde los valores se toman de uno de dos tipos (normalmente diferentes). S + T es el conjunto de valores donde cada valor es tomado ya sea de S o de T y es rotulado (*izq* o *der*) para indicar de qué conjunto fue tomado:

$S + T = \{ izq\ x / x \in S\} \cup \{der\ y / y \in T\}$



La cardinalidad está dada por #(S + T) = #S + #T

Podemos extender esta operación a *n* tipos: S1, S2... Sn en donde cada uno de estos puede ser rotulado por *i*.

Los registros variables en Pascal y Ada son ejemplo de esta operación. Consideremos la siguiente definición de tipos en Pascal:

```
Type  precision =(Exact, Aprox);
      numero = record
                  Case prec: precision of
                        exact: (valEnt: Integer);
                        aprox: (valReal: Real)
               end
```

El mismo ejemplo en ML

```
datatype
      numero =  exact of int
              | aprox of real
```

El conjunto de valores de este tipo es Integer x Real. Sus valores son:
{..., exact (-2), exact (-1), exact (0), exact (1), exact (2), ...}
$\cup$ {..., aprox (-1.0), ... aprox (0.0), ... aprox (1.0), ... }

Aquí los rótulos son valores *exact* y *aprox*. Como se ve en el ejemplo, en Pascal los rótulos son valores de cualquier tipo primitivo discreto.

Por ejemplo si tenemos una variable `num` del tipo `numero` que posee el valor exacto 7, `num.prec` tendrá el valor *exact*, y `num.valEnt` tendrá el valor 7. Si el programa intenta acceder a `num.valReal`, el cual no existe actualmente, obtendremos un error en tiempo de ejecución. La asignación de *aprox* a `num.prec` tiene el efecto de destruir `num.valEnt` y crear `num.valReal` con un valor indefinido, por lo tanto el valor de `num` ha cambiado en un paso de *exact 7* a *aprox idefinido*.

Cabe aclarar que la unión disjunta no es lo mismo que la unión común. El rótulo nos permite identificar de dónde un valor fue tomado. Por ejemplo si T = {a, b, c}

$T \cup T = \{a, b, c\} = T$
$T + T = \{izq\ a, izq\ b, izq\ c, der\ a, der\ b, der\ c\}$

Mapeo

La noción de mapeo o función de un conjunto a otro es muy importante en los lenguajes de programación. Consideremos una función m que relaciona cada valor x en S con un valor de T. Los valores de T son llamados imágenes de x bajo m y se escribe m(x).

$m: S \rightarrow T$ es una función de S en T.

Si consideramos a S = {u, v} y T = {a, b, c}. Por ejemplo $m(x) = \{u \rightarrow a, v \rightarrow c\}$ puede ser una función definida de S en T.

Con la notación $S \rightarrow T$ definimos el conjunto de todas las funciones posibles de S en T.

$S \rightarrow T = \{m / x \in S \Rightarrow m(x) \in T\}$

La cardinalidad está dada por $\#S \rightarrow T = (\#T)^{\#S}$

Por ejemplo un array puede ser entendido como un mapeo:

```
Array [S] of T

Type  color = (rojo, verde, azul);
      Pixel =  array [color] of 0..1
```

El conjunto de valores del tipo arreglo es pixel = color $\rightarrow$ {0, 1} donde color = {rojo, verde, azul} Los valores para el tipo pixel está dado por ocho combinaciones.

{rojo $\rightarrow$ 0, verde $\rightarrow$ 0, azul $\rightarrow$ 0}	{rojo $\rightarrow$ 1, verde $\rightarrow$ 0, azul $\rightarrow$ 0}
{rojo $\rightarrow$ 0, verde $\rightarrow$ 0, azul $\rightarrow$ 1}	{rojo $\rightarrow$ 1, verde $\rightarrow$ 0, azul $\rightarrow$ 1}
{rojo $\rightarrow$ 0, verde $\rightarrow$ 1, azul $\rightarrow$ 0}	{rojo $\rightarrow$ 1, verde $\rightarrow$ 1, azul $\rightarrow$ 0}
{rojo $\rightarrow$ 0, verde $\rightarrow$ 1, azul $\rightarrow$ 1}	{rojo $\rightarrow$ 1, verde $\rightarrow$ 1, azul $\rightarrow$ 1}

La mayoría de los lenguajes de programación permiten definir arreglos multidimensionales. Podemos pensar en un arreglo multidimensional como tener un único índice que sea una n-tupla.

Por ejemplo:

```
Type ventana = array [0..799, 0..599] of 0..1
```

El conjunto de valores de este tipo es ventana = {0, 1, …, 799} x {0, 1, …, 599} $\rightarrow$ {0, 1} y está indexado por un par ordenado de enteros.

Además de la forma de arreglos, los mapeos ocurren en los lenguajes de programación en la forma de abstracción de función. Una función implementa un mapeo de $S \rightarrow T$ por el significado del algoritmo que toma cualquier valor de S (el argumento) y calcula su imagen en T (el resultado). El conjunto S no necesita ser finito.

Consideremos otro la siguiente función definida en Pascal:

```
Function par (n: Integer) : Boolean;
      Begin
        Par := (n mod 2 = 0);
      End;
```

Esta función implementa un mapeo particular definido de Integer → boolean y sus valores son:

{0 → true, ±1 → false, ±2 → true, ±3 → false, ...}

Una función `impar`, implementará otro mapeo de Integer → boolean.

Si una función recibe n valores como parámetros podemos interpretarlo como un simple argumento. Una n-tupla.

Conjunto Potencia

Consideremos un conjunto de valores denominado S. Estamos interesados en valores que son entre si, subconjuntos de S. El conjunto de todos los subconjuntos se llama *Conjunto Potencia* de S y se escribe formalmente:

$\wp S = \{s / s \subseteq S\}$

Las operaciones básicas a un conjunto son las operaciones usuales de la teoría de conjuntos: pertenencia, inclusión, unión e intersección.

Cada valor de S puede ser miembro o no de un conjunto particular en $\wp S$. Y además la cardinalidad del conjunto potencia de S está dada por:

$\#(\wp S) = 2^{\#S}$

Por ejemplo, considere la siguiente definición Pascal...

```
Type  color = (rojo, verde, azul);
      ConjPot = set of color
```

El conjunto de valores de este tipo es PoSet = $\wp$color es el conjunto de todos los subconjuntos de Color = {rojo, verde, azul}. Los siguientes 8:

{ }	{rojo}	{verde}	{azul}
{rojo, verde}	{rojo, azul}	{verde, azul}	{rojo, verde, azul}

Pascal permite solo conjuntos de valores primitivos (boolean, char, enumeados y enteros).

Tipos Recursivos

Se considera un tipo recursivo a aquel que se compone de valores del mismo tipo y se define en términos de si mismo.

Uno de los tipos recursivos son las *listas* las cuales pueden tener cualquier número de componentes inclusive ninguno. El número de componentes es llamado largo de la lista. Una lista es homogénea si todos los elementos de la misma son del mismo tipo.

Supóngase que queremos definir un tipo cuyos valores son listas de enteros. Podemos definir dicha lista de enteros como un valor que puede ser vacío o un par que consiste de un entero (cabeza) y la una lista de enteros (cola). Esta definición es recursiva y puede ser escrita como:

Lista_enteros = unit + (Entero x Lista_enteros)

En otras palabras:

Lista_enteros = { nil () } ∪ {cons (i,l) / i ∈ Entero, l ∈ Lista_enteros}

Donde nil y cons indicarían los rótulos considerados en el producto cartesiano.

En definitivas esta definición puede ser descripta por los siguientes conjuntos:

{ nil }
$\cup$ {cons (i, nil) / i $\in$ Entero }
$\cup$ {cons (i, cons(j, nil)) / i, j $\in$ Entero }
$\cup$ {cons (i, cons(j, cons(k, nil))) / i, j, k $\in$ Entero }
$\cup$ ...

Generalizando la ecuación de un conjunto recursivo:

L = Unit + (S x L) que es un conjunto todas las listas finitas de valores tomados de S.

Un tipo recursivo es aquel cuyos valores están compuestos por valores del mismo tipo. Un tipo recursivo se define en términos de si mismo. En general el conjunto de valores de un tipo T, se definirán por una ecuación recursiva de la forma:

T = ... T ...

La solución de la ecuación puede determinarse de la siguiente manera: reemplazar el conjunto vacío para T en la parte derecha de la ecuación, esto da una primera aproximación de T. Luego sustituir la primera aproximación de T en la parte derecha de la ecuación nuevamente para obtener la segunda aproximación de T y así sucesivamente.

La cardinalidad de un tipo recursivo es infinito. Por lo tanto nunca podremos enumerar todos los valores para un tipo recursivo.

Otro tipo recursivo son los árboles; la definición de un árbol en ML puede ser:

```
datatype
      arbol = hoja of int
            | rama of arbol * arbol
```

Según la definición de producto cartesiano e unión disjunta será: arbol = int + (arbol x arbol)

Control de tipos

Agrupar valores en tipos permite al programador describir los datos. Previene además que los programas realicen operaciones sin sentido como por ejemplo multiplicar un carácter por un valor de verdad. En este sentido los lenguajes de alto nivel se distinguen de los de bajo nivel donde los *tipos* son byte y word.

Para prevenir la realización de operaciones sin sentido la implementación de los lenguajes deben realizar una *comprobación de tipos* sobre los operandos. Cuando se realiza una multiplicación debe asegurarse de que los dos operandos sean números. Similarmente antes de realizar una operación lógica, el o los operandos deben chequearse para asegurarse de que sean valores de verdad.

Sin embargo hay bastante libertad en cuanto al momento de hacer el control de tipos; puede realizarse en tiempo de compilación o de ejecución; este hecho marca una importante clasificación de los lenguajes de programación en lenguajes estáticamente tipeados y dinámicamente tipeados.

En un lenguaje **tipeado estáticamente**, cada variable y parámetro tiene un tipo fijo y es elegido por el programador. Por lo tanto el tipo de cada expresión puede ser deducido y la comprobación de tipo se realiza en tiempo de compilación. La mayoría de los lenguajes de alto nivel son estáticamente tipeados.

En un lenguaje **tipeado dinámicamente**, solo los valores tienen un tipo fijo. Una variable o parámetros no tienen un tipo designado, pero pueden tomar valores de diferente tipo en diferentes momentos. Esto implica que el tipo de los operandos deben ser chequeados inmediatamente antes de realizar una operación en tiempo de ejecución. Lisp y Smalltalk son lenguajes dinámicamente tipeados.

Por ejemplo, considere la siguiente función Pascal:

```
Function par (n: integer): boolean;
      Begin
            Par := (n mod 2 = 0)
      End;
```

No se puede conocer de antemano el valor que tomará la variable `n`, de lo que sí podemos estar seguros es que será un valor entero debido a que fue declarado como tal. Asimismo podemos deducir que los operandos de **mod** serán ambos enteros, y que su resultado también lo será. Finalmente podemos deducir que el valor devuelto por la función será un valor de verdad como se especifica en la definición.

En cada llamada a la función `par (i+1)` podemos chequear que el argumento deberá ser un entero.

En tiempo de compilación ya se conoce el tipo de cada variable y de cada expresión.

Considere ahora la siguiente función de un lenguaje hipotético tipeado dinámicamente:

```
Function par (n);
      Begin
            Par := (n mod 2 = 0)
      End;
```

El tipo de `n` es desconocido de antemano, por lo tanto es necesario un control de tipo en tiempo de ejecución debido a que la función **mod** necesita que ambos operandos sean enteros. Esta función puede ser llamada con argumentos de diferentes tipos, por ejemplo `par (i+1)` o `par (true)` o `par (x)` donde el tipo de `x` tampoco es conocido de antemano.

El tipeo dinámico implica que la ejecución de un programa será un poco más lenta debido al control de tipos necesario en tiempo de ejecución y también implica que cada valor debe ser rotulado con su tipo de manera de hacer posible el control de tipos.

Esta sobrecarga de tiempo y espacio son evitados en un lenguaje estáticamente tipeado porque todo el control de tipos se hace en tiempo de compilación. Una ventaja importante del tipeo estático es la seguridad de que los errores de tipos están garantizados que serán detectados por el compilador.

La ventaja de un tipeo dinámico es su flexibilidad.

Equivalencia de tipos

Considere una operación que espera un operando del tipo T. Suponga que en realidad el operando en cuestión es de tipo T'. El lenguaje debe chequear si el tipo T es equivalente a T' ($T \equiv T'$). Deberíamos encontrar una forma de determinar si un tipo es equivalente a otro. Una de las posibles maneras de definir la *Equivalencia de tipos* podría ser:

Equivalencia Estructural:

$T \equiv T'$ si y solo si T y T' tienen el mismo conjunto de valores.

Mediante la equivalencia estructural se hará el chequeo de tipos comparando las *estructuras* de los tipos T y T'. (No es necesario, y en general imposible, enumerar todos sus valores).

Las siguientes reglas ilustran como podemos decidir si $T \equiv T'$, definido en términos de productos cartesiano, uniones disjuntas y mapeos.

- Si T y T' son dos tipos primitivos. Luego $T \equiv T'$ si y solo sí T y T' son idénticos.
  Ej: Integer ≡ Integer.
- Si $T = A \times B$ y $T' = A' \times B'$. Luego $T \equiv T'$ si y solo sí $A \equiv A'$ y $B \equiv B'$.
  Ej: Integer x Character ≡ Integer x Character
- Si $T = A + B$ y $T' = A' + B'$. Luego $T \equiv T'$ si y solo sí $A \equiv A'$ y $B \equiv B'$ o $A \equiv B'$ y $B \equiv A'$.
  Ej: Integer + Character ≡ Character + Integer
- Si $T = A \rightarrow B$ y $T' = A' \rightarrow B'$. Luego $T \equiv T'$ si y solo sí $A \equiv A'$ y $B \equiv B'$.
  Ej: Integer → Character ≡ Integer → Character
- En otro caso, T no es equivalente a T'.

A pesar de que estas reglas son simples, no es fácil ver si dos tipos recursivos son estructuralmente equivalentes. Considere lo siguiente:

T = Unit (A x T)
T' = Unit (A x T')

Intuitivamente, T y T' son estructuralmente equivalente. Sin embargo, el razonamiento necesario para decidir si dos tipos recursivos arbitrarios son estructuralmente equivalentes, hace que el chequeo de tipos sea muy difícil.

Otra posible definición de la equivalencia de tipos podría ser:

Equivalencia de Nombres
T ≡ T' si y solo si T y T' fueron definidos en el mismo lugar.

Por ejemplo, considere la siguiente definición Pascal.

```
type  T1 = file of Integer;
      T2 = file of Integer;

var   F1: T1;
      F2: T2;

procedure p (var F: T1);
      begin  ... end;
```

La llamada a procedimiento "`p (F1)`" pasaría la verificación de tipos debido a que los tipos de `F` y `F1` son equivalentes. Sin embargo la llamada "`p (F2)`" fallaría la verificación de tipos debido a que los tipos `F` y `F2` no son equivalentes; `T1` y `T2` están definidos en diferentes declaraciones.

Principio de completitud de tipos

Las funciones y procedimientos se consideran como valores porque pueden ser pasadas como argumentos; pero no pueden ser evaluados, no pueden ser asignados, ni usados como parte de valores compuestos. Por lo tanto se dice que estos valores son *valores de segunda clase*; por otro lado, los valores lógicos, enteros, registros, arreglos, etc. pueden usarse para cualquiera de estas operaciones, por lo tanto se dicen *valores de primera clase*.

Este tipo de distinción es común en la mayoría de los lenguajes de programación (como Fortran, Algol-60, Pascal y Ada). Sin embargo otros lenguajes (como ML, Miranda y algunas extensiones de Algol-68) intentan evitar esta distinción de clases y permiten que todos los valores, incluidas las abstracciones, se manejen de manera similar.

Por ejemplo el resultado de una función en Pascal debe ser un valor primitivo o un puntero, no puede ser un string, conjunto o registro. Esta distinción no tiene lugar en lenguajes más modernos, como el Ada que permite que el resultado de una función sea de cualquier tipo.

Del otro extremo, algunos lenguajes (mencionados arriba) intentan eliminar toda distinción de clases y de alguna manera tratan de cumplir el principio de la completitud de tipos:

> *"Las operaciones que se puedan realizar en valores pertenecientes a los tipos no deberían ser arbitrarias a los mismos".*

Este principio debe contribuir a que los lenguajes no posean restricciones sobre las operaciones que pueden aplicarse a determinados tipos y de esa manera reducir el poder de un lenguaje de programación.

Sin embargo, una restricción podría justificarse por otra, conflicto, consideraciones de diseño etc.

Sistema de tipos

Los lenguajes de programación clásicos, tales como Pascal, tienen un sistema de tipos muy simple. Cada constante, variable, resultado de función y parámetro formal, deben declararse con un tipo específico. Este sistema de tipos se denomina monomórfico, y hace que el control de tipos sea sencillo.

Desafortunadamente, la experiencia muestra que un sistema de tipos monomórfico puro, es insatisfactorio, especialmente para escribir software reusable. Muchos algoritmos estándares (tales como los algoritmos de ordenación) son inherentemente estándares, en el sentido de que solo dependen del tipo de valores que se están manejando. Un sistema de tipos monomórfico obliga a declarar los valores que se manejan de un tipo particular.

Este y otros problemas, obligan a desarrollar sistemas de tipos más poderosos, los cuales están implementados en lenguajes modernos como el Ada y ML. Los conceptos relevantes son *sobrecarga*, que es la habilidad de un identificador u operador a denotar varias abstracciones simultáneamente; *polimorfismo*, que tiene que ver abstracciones que pueden operar uniformemente sobre valores de diferentes tipos y *herencia*, que se refiere al hecho de que subtipos hereden características de supertipos.

Monomorfismo

Un sistema de tipos monomórfico es aquel en dónde las constantes, variables, parámetros y resultado de función, tienen un único tipo. El sistema de tipos de Pascal es básicamente monomórfico.

Por ejemplo:

```
type conjNum = set of 1..10;

function disjunto (s1, s2: conjNum) : boolean;
begin
     disjunto := (s1 * s2 = []);
end;

var
   numeros : conjNum;

begin
   numeros := [9,2,5];
   if disjunto (numeros, [1,2,3])
     then writeln ('Son disjuntos')
     else writeln ('No son disjuntos');
end.
```

El tipo de la función `disjunto` es: $\wp$1..10 x $\wp$1..10 → Valor_verdad; y evalúa si el conjunto `numeros` es disjunto a [1, 2, 3]. El cuerpo de la función usa operaciones de conjuntos. Sin embargo, la función es monomórfica, no puede aplicarse con argumentos del tipo $\wp$caracteres ni $\wp$color, ni con ningún otro tipo que no sea $\wp$1..10.

Pascal fuerza a que se especifique el tipo exacto de todos los parámetros formales y del resultado de toda función. Como consecuencia, todas las funciones y procedimientos definidos en Pascal son monomórficas.

Sin embargo, ni Pascal ni otros lenguajes son estrictamente monomórficos; por ejemplo, muchas de las funciones o procedimientos predefinidos del Pascal, tienen un tratamiento especial por el compilador. Considere el procedimiento predefinido `write`, cuya llamada tiene la forma `write (`*E*`)`. El efecto de esta llamada a procedimiento depende del tipo de *E*. Existen varias posibilidades. Por ejemplo, si *E* es del tipo `Char`, un simple carácter será escrito. O si *E* es del tipo `string`, una secuencia de caracteres deben escribirse, un valor entero deberá convertirse a una cadena de caracteres (un valor decimal rellenado con espacios en blanco) y esa secuencia deberá escribirse. En realidad, el identificador `write` denota simultáneamente distintos procedimientos. Cada uno con su propio tipo. Este es un ejemplo de *sobrecarga*.

Considere ahora la función predefinida de Pascal eof. Esta función toma un argumento de cualquier tipo de archivo y controla por el fin de archivo. El tipo de la función es File (τ) $\rightarrow$ valor_verdad, donde τ, es de cualquier tipo. Esta función se dice que es polimórfica. Acepta argumentos de distintos tipos, pero opera uniformemente en todos los casos.

No debemos confundir los conceptos de sobrecarga y polimorfismo.

Sobrecarga significa que un número (pequeño) de abstracciones distintas están asociadas al mismo identificador; estas abstracciones no necesariamente deben tener tipos relacionados, ni realizar necesariamente operaciones similares en sus parámetros.

Polimorfismo es una propiedad de una única abstracción que tiene una (amplia) gama de tipos relacionados; la abstracción opera uniformemente sobre sus argumentos cualquiera sea su tipo.

Pareciese que la sobrecarga no aumenta el poder de un lenguaje, debido a que puede ser fácilmente eliminada renombrando las abstracciones sobrecargadas. Así el procedimiento `write` del Pascal podría renombrarse por `writechar`, `writestring`, etc.

Pascal es inconsistente: todas las abstracciones definidas por el programador son monomórficas, pero muchas de las abstracciones predefinidas por el lenguaje están sobrecargadas o son polimórficas. Otros lenguajes tratan de evitar tal inconsistencia y hacen que la sobrecarga y el polimorfismo esté disponible tanto para las abstracciones predefinidas como para las definidas por el programador.

Otro concepto que se puede encontrar en Pascal, aunque sólo en su mínima expresión, es el concepto de herencia. Considere el tipo subrango '`type rango = 28..31`'. El conjunto de valores de este tipo es {28, 29, 30, 31} $\subset$ Integer.

Cualquier operación que espere un entero puede aceptar un valor del tipo `rango`. El tipo `rango` se dice que hereda todas las operaciones del tipo `Integer`. Un tipo subrango en Pascal hereda todas las operaciones de su tipo padre. Sin embargo, no todos los tipos en Pascal heredan las operaciones de cualquier otro tipo distinto.

Sobrecarga

Un identificador u operador se dice que está **sobrecargado** si denota simultáneamente dos o más funciones distintas. En general, la sobrecarga es aceptable solo cuando cada llamada a función no es ambigua; donde la función a ser llamada puede identificarse unívocamente usando la información de tipos disponible.

En Pascal y ML, solo identificadores y operadores que denotan abstracciones predefinidas pueden sobrecargarse.

Ejemplo 1: el operador '`-`' denota simultáneamente cinco funciones distintas.

- Negación de un entero (una función de Integer $\rightarrow$ Integer)
- Negación de un real (una función de Real $\rightarrow$ Real)
- Diferencia de enteros (una función de Integer x Integer $\rightarrow$ Integer)
- Diferencia de reales (una función de Real x Real $\rightarrow$ Real)
- Diferencia de conjuntos (una función de Set x Set $\rightarrow$ Set)

No existe ambigüedad. En las llamada a función tales como '`-y`' y '`x-y`', el número de parámetros actuales y sus tipos determinan unívocamente la función que se llamará.

Ejemplo 2: el operador '/' en Ada denota dos funciones distintas:

- División de enteros (una función de Integer x Integer $\rightarrow$ Integer)
- División de reales (una función de Real x Real $\rightarrow$ Real)

La llamada '`7/2`' retornará 3 y '`7.0/2.0`' retornará 3.5.

La siguiente función Ada puede sobrecargar el operador '/':

```
fuction "/" (m, n : integer) return float is
begin
   return float (m) / float (n);
end;
```

el cuerpo de la función llama a la función división de reales. Esta sobrecarga hace que '/' también denote:

- División real de enteros (una función de Integer x Integer → Real)

  La llamada a la función '/' dependerá del contexto tanto como del número y tipo de parámetros actuales. Considere los siguientes casos: (/ para división de reales y % para división de enteros).

```
n: integer; x: float;
...
x := 7.0/2.0;          calcula 7.0/2.0  = 3.5
x := 7/2;              calcula 7%2 = 3.5
n := 7/2;              calcula 7%2 = 3
n := (7/2)/(5/2);      calcula (7%2)%(5%2)  =  1
x := (7/2)/(5/2);      calcula (7%2)/(5%2) = 1.5   ó   (7/2)/(5/2) = 1.4
                       (el lenguaje debe prohibir estas expresiones)
```

Un ejemplo en C++

```
/* PROGRAMA: PRACTICO
   ACCION  : OPERACIONES CON COMPLEJOS: Sobrecarga de operadores   */

#include <iostream.h>
#include <conio.h>
#include <math.h>

class complejo
  {
    float real;
    float imag;
  public:
    complejo (float a = 0., float b = 0.)
      { real = a; imag = b; }
    ~complejo (void);
    complejo operator+ (complejo c);
    complejo operator- (complejo t);
    complejo operator= (complejo r);
    complejo operator* (complejo m);
    complejo operator/ (complejo n);
    friend complejo operator++ (complejo &i);
    friend complejo operator-- (complejo &i);
    void mostrar (void)
      {
      cout << "(" << real << "," << imag << ")\n";
      }
  };

complejo::~complejo (void)
  {
  //  "destruccion"
  }

complejo complejo::operator+ (complejo c)
  {
  complejo temp;

  temp.real = this -> real + c.real;
  temp.imag = this -> imag + c.imag;
  return temp;
  }

complejo complejo::operator- (complejo t)
  {
  complejo temp;

  temp.real = real - t.real;
  temp.imag = imag - t.imag;
```

```
  return temp;
  }

complejo complejo::operator= (complejo r)
  {
  real = r.real;
  imag = r.imag;
  return *this;
  }

complejo complejo::operator* (complejo m)
  {
  complejo temp;

  temp.real = real * m.real - imag * m.imag;
  temp.imag = imag * m.real + real * m.imag;
  return temp;
  }

complejo complejo::operator/ (complejo n)
  {
  complejo temp;

  temp.real = (real * n.real + imag * n.imag) / (pow(n.real,2) + pow(n.imag, 2));
  temp.imag = (imag * n.real - real * n.imag) / (pow(n.real,2) + pow(n.imag, 2));
  return temp;
  }

complejo operator++ (complejo &i)
  {
  i.real++;
  i.imag++;
  return i;
  }

complejo operator-- (complejo &i)
  {
  i.real--;
  i.imag--;

  return i;
  }


void main (void)
  {
  complejo a(1,2), b(3,1), c;

  clrscr();
  cout << "Numero a: ";
  a.mostrar ();

  cout << "Numero b: ";
  b.mostrar ();

  cout << "Numero c: ";
  c.mostrar ();

  c = a + b;
  cout << "c = a + b: ";
  c.mostrar ();

  c = a - b;
  cout << "c = a - b: ";
  c.mostrar ();

  c = a * b;
  cout << "c = a * b: ";
  c.mostrar();

  c = a / b;
  cout << "c = a / b: ";
  c.mostrar();

  a++;
  cout << "a++: ";
  a.mostrar ();

  b--;
  cout << "b--: ";
  b.mostrar ();
```

```
getch();
}
```

Podemos caracterizar la sobrecarga en términos de los tipos de las funciones sobrecargadas. En pascal y ML, el tipo del parámetro de la función sobrecargada es siempre distinto. En Ada, el tipo del parámetro o del resultado de la función sobrecargada es distinto.

Mas generalmente, considere un identificador un operador I que denota a una función f1 del tipo S1 $\rightarrow$ T1 y otra función f2 del tipo S2 $\rightarrow$ T2. (Note que se cubren los casos en que las funciones tengan varios argumentos: S1 o S2 podrían ser productos cartesianos). Existen dos tipos de sobrecarga:

- Sobrecarga independiente del contexto (como en Pascal y ML): requiere que S1 y S2 sean distintos. Considere la llamada a la función I (E). Si el parámetro actual E es del tipo S1, entonces I denota a f1 y el resultado es del tipo T1; si E es del tipo S2, entonces I denota f2 y el resultado será del tipo T2. Con la sobrecarga dependiente del contexto la función que se llama es identificada unívocamente por el parámetro actual.

- Sobrecarga dependiente del contexto (como en Ada): requiere que S1 y S2 sean distintos o que T1 y T2 sean distintos. Si S1 y S2 son distintos, la función a llamar puede identificarse como anteriormente. Si S1 y S2 no son distintos, pero T1 y T2 son distintos, el contexto debe tenerse en cuenta para identificar la función que se llamará. Considere la llamada a la función I(E), donde E es del tipo S1 ($\equiv$ S2). Si la llamada a la función ocurre en un contexto donde se espera una expresión del tipo T1, entonces I debe denotar a f1; si la llamada a la función ocurre en un contexto donde se espera una expresión del tipo T2, entonces I debe denotar f2. Con la sobrecarga dependiente del contexto, es posible formular expresiones en las cuales la función a llamar no pueda ser identificada unívocamente pero el lenguaje debe prohibir esas expresiones ambiguas.

Polimorfismo

En un sistema de tipos polimórfico, podemos escribir abstracciones que operan uniformemente sobre argumentos de una amplia gama de tipos relacionados. ML fue el primer lenguajes de programación que tuvo un sistema de tipos polimórfico.

Abstracciones polimórficas

En ML es simple definir una función polimórfica. La clave es definir el tipo de tales funciones usando tipos variables, en vez de tipos específicos.

Ejemplo 1:

La siguiente función acepta un par de enteros y retorna su suma:

```
fun sum (x: int, y: int) = x + y
```

Esta función es del tipo Integer x Integer $\rightarrow$ Integer. La llamada a la función `sum (13, 21)` retornará 34.

Ahora considere la función siguiente, la cual acepta un par de enteros y simplemente retorna el segundo de ellos.

```
fun segundo (x: int, y: int) = y
```

Esta función también es del tipo Integer x Integer $\rightarrow$ Integer. La llamada a la función `segundo (13, 21)` retornará 21. La llamada `segundo (13, true)` debería ser ilegal porque el par de argumentos no consiste de dos enteros. La llamada `segundo (13)` o `segundo (1859, 2, 21)` también debería ser ilegal, porque el argumento no es un par.

Pero por qué restringir a segundo a aceptar un par de enteros? No hay nada en el cuerpo de la función que es específicamente una operación sobre enteros, por lo tanto el argumento de la función podría, en principio, ser un par de valores cualesquiera. Eso es de hecho posible al definir segundo de la siguiente manera:

```
fun segundo (x: σ, y: τ) = y
```

esta función se dice que es del tipo σ x $\tau \rightarrow \tau$. Aquí σ y τ se entienden como algún tipo cualquiera.

Ahora la llamada a la función `segundo (13, true)` es legal. Su tipo se determina como sigue: podemos asociar el tipo de argumentos Integer x valor_verdad al tipo de la función σ x $\tau \rightarrow \tau$. Por lo tanto al sustituir sistemáticamente se entiende que el tipo del resultado de la función será un valor_verdad (true en este caso).

La función `segundo` es polimórfica. Esto no implica que aceptará cualquier argumento. Una llamada como `segundo (13)` o `segundo (1859, 2, 21)` todavía son inválidas, porque el argumento no puede asociarse con el tipo σ x $\tau \rightarrow \tau$. Los parámetros permitidos son solo aquellos valores que tienen tipos de la forma σ x τ (pares).

σ y τ son tipos variables, se entiende como un tipo desconocido. (se escriben en letras griegas por convención).

Ejemplo 2:

Si Pascal permitiese funciones polimórficas, `disjunto` podría ser escrita de la siguiente manera:

```
function disjunto (s1, s2: set of τ) : boolean;
begin
     disjunto := (s1 * s2 = []);
end;
```

El tipo de la función `disjunto` es: $\wp\boldsymbol{\tau}$ x $\wp\boldsymbol{\tau} \rightarrow$ Valor_verdad.

Tipos parametrizados

Un tipo parametrizado es un tipo que tiene otro(s) tipo(s) como parámetros. Por ejemplo en Pascal el tipo `file`, `set` y `array`. Por ejemplo podemos definir `file of char` o `file of integer`. Podemos pensar que `file` es un tipo parametrizado de la forma **`file of τ`**.

En un lenguaje monomórfico, solo algunos tipos predefinidos son parametrizados. El programador no pude defnir nuevos tipos parametrizados.

Ejemplo:

En ML podemos hacer la siguiente definición:

```
type τ par = τ * τ
```

En esta definición $\boldsymbol{\tau}$ actúa como un parámetro que denota un tipo desconocido. Un ejemplo del uso de `par` podría ser `int par`. Al sustituir `int` por τ, vemos que es equivalente a `int * int`, y denota un tipo en el cual sus valores es un par de enteros. Otro ejemplo podría ser `real par`, que denota un tipo en el cual cada valor es un par de números reales.

Coersiones

Una coersión es un mapeo implícito entre valores de un tipo a valores de un tipo diferente. Ejemplos típicos de coersiones son mapeos de enteros a números reales y mapeos de caracteres a strings de un único carácter. Una coersión se realiza automáticamente cuando el contexto lo demanda.

Considere un contexto en el cual se espera un opeando del tipo T pero se sustituye por un operando del tipo T' (no equivalente a T). El lenguaje de programación puede permitir una coersión en este contexto proveyendo un mapeo del tipo T al T'.

Esto es ilustrado por la expresión Pascal `sqrt (n)`, donde la función `sqrt` espera un argumento del tipo real, pero `n` es de tipo entero. Existe un mapeo obvio de Integer a Real:

{..., -2 $\rightarrow$ -2.0, -1 $\rightarrow$ -1.0, 0 $\rightarrow$ 0.0, 1 $\rightarrow$ 1.0, 2 $\rightarrow$ 2.0, ...}

por lo tanto la expresión `sqrt (n)` es legal.

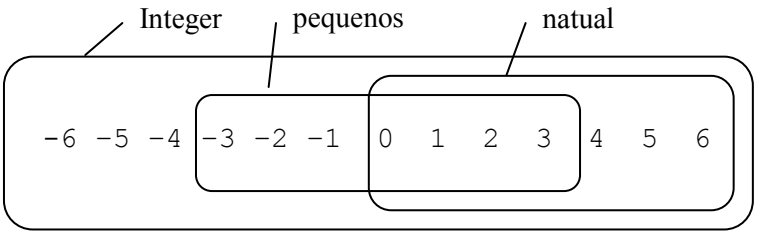
Subtipos y herencia

Si consideramos un tipo T como un conjunto de valores, podríamos pensar en construir subconjuntos de T. Llamaremos a cada subconjunto de T ***subtipo*** de T.

Pascal reconoce una única clase de subtipo. Podemos definir subrangos de un tipo primitivo T discreto.

Ejemplo:

```
type natural = 0..maxint;
     pequenos = -3..+3;
var  i: integer;
     n: natural;
     s: pequenos;
```



Una condición necesaria para que S sea un subtipo de T es que $S \subseteq T$. Así un valor de S pude usarse cuando se espera un valor de T.

Asociado con cada tipo T, existen un número de operaciones aplicables a los valores de T. Cada una de estas operaciones también serán aplicables a los valores de cualquier subtipo S de T. Podemos pensar que S hereda todas las operaciones asociadas con T. Por ejemplo una función del tipo Integer $\rightarrow$ valor_verdad será heredada por todos los subtipos de Integer, tales como Natural y Pequenos.

El término herencia viene de la programación orientada a objetos. Un objeto a tiene uno o más componentes (ocultos), los cuales pueden accederse por operaciones asociadas con a. Un segundo objeto b podría declararse para tener todos los componentes de a (y quizás componentes adicionales). Luego b hereda todas las operaciones asociadas de a (b puede tener operaciones adicionales que acceden solo a los componentes de b).

Expresiones

Habiendo considerados valores y tipos, examinemos las expresiones. Una expresión es una frase de programa que será evaluada para retornar un valor.

Las expresiones pueden formarse de diversas maneras. Las fundamentales son:

- Literales
- Agregados
- Llamadas a funciones
- Expresiones condicionales
- Accesos a constantes y variables

Literales

Es la clase más simple de expresión, denota un valor fijo y manifiesto de algún tipo. Por ejemplo:

```
6356  3.1416      '%'    ' CONSULTA'
```

Los cuales denotan un entero, un real, un carácter, y un string respectivamente.

Agregados

Es una expresión que construye un valor compuesto a partir de sus valores componentes. Los valores componente se determinan al evaluar subexpresiones.

Ejemplos en ML:
El siguiente agregado de tupla:

```
(a * 2.0, b/2.0)
```

construye un valor de la tupla del tipo real * real

El siguiente agregado de registro:

```
{ y = anio + 1, m= "Ene", d = 1}
```

construye un valor del tipo de registro {y:int, m: string, d: int}

El siguiente agregado de lista:

```
[31, if bisiesto (anio) then 29 else 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

construye un valor del tipo `int list`

Para construir un valor del tipo registro en Pascal hay que asignar cada componente uno por uno.
En Ada podemos escribir:

```
NuevoDia := (y => anio + 1, m => 'Ene', d => 1);
```

Usa un agregado de registro parecido a ML.

Llamadas a funciones

Una llamada a una función calcula un resultado al aplicar una abstracción de función a un argumento. Una llamada a función normalmente tiene la forma `F (pa)` donde `F` determina la función a aplicarse, y los **parámetros actuales** '`pa`' determina los argumentos que le serán pasados.

Expresiones condicionales

Una expresión condicional tienen varias subexpresiones de las cuales solo una se toma para ser evaluada. No todos los lenguajes proveen expresiones condicionales que no es lo mismo que comandos condicionales.

Por ejemplo, el lenguaje C provee una expresión condicional:

```
strNum  = (a>0? 'positivo' : 'negativo')
```

En cambio el Pascal y el Ada no proveen este tipo de expresiones y deben recurrir a comandos condicionales:

```
if a>0 then strNum := 'positivo'
       else strNum := 'negativo'
```

Acceso a constantes y variables

Un tipo de expresiones tiene que ver con el acceso a constantes y variables mediante sus nombres.
Por ejemplo:

```
const pi = 3.1416;
var   r : real;
```

En la expresión `2*pi*r` se produce un acceso al valor que posee la constante `pi` la cual retorna el valor 3.1416 y a la variable `r` la cual retorna el valor que pueda contener la misma.

Variables y Actualizaciones

En computación, una variable es un objeto que contiene un valor, este valor puede ser inspeccionado y actualizado tanto como se desee. Las variables se usan para modelar objetos del mundo real que poseen estados, tales como la población del mondo, la fecha de hoy, el estado del tiempo actual, o la economía de un país.

Nuestra definición de variables es bastante general. Lo más familiar para un programador son las variables que se crean y se usan en un programa particular, tales variables son típicamente actualizadas por la asignación; y son de vida corta. Sin embargo, archivos y bases de datos son también variables. Estas son generalmente de larga vida, existen independientemente de un programa particular.

Introduciremos la idea de almacenamiento para entender como las variables se actualizan. Los medios reales de almacenamiento (tales como memorias y discos) tienen propiedades que son irrelevantes para nosotros (tales como tamaño de palabra, capacidad, convenciones de direccionamiento). Por lo tanto propondremos un modelo abstracto de almacenamiento.

Un almacén es una colección de celdas.
Cada celda tiene un estado actual: reservado o no reservado.
Cada celda reservada tiene un contenido actual, que es un valor *almacenable* o indefinido.

Ejemplo: considere la variable n en un programa Pascal.

1. La declaración de la variable (`var n: Integer`) causa que alguna celda no reservada cambie su estado a reservada, y su contenido cambie a indefinido.
2. La asignación `n:= 0` cambia a cero el contenido de la celda denotada por `n`.
3. La expresión `n+1` retorna uno más que el contenido de la celda denotada por `n`. La `asignación n:= n+1` suma uno al contenido de esa celda.
4. Al final del bloque que contiene la declaración de `n`, se revierte el estado de la celda denotada por `n` a no reservado.

Variables Compuestas

El valor de un tipo compuesto consiste de componentes que pueden ser inspeccionado separadamente. Asimismo, una variable de un tipo compuesto consiste de componentes que son variables también y el contenido de esos componentes pueden ser inspeccionados y actualizados separadamente.

Un valor de un tipo primitivo ocupa una celda simple, pero una variable de un tipo compuesto puede ocupar varias celdas.

Actualización total y selectiva

Variables tipo Arreglo

Tiempo de Vida de una Variable

Una propiedad de toda variable es que la misma es creada en algún momento definido y puede ser eliminada en otro momento posterior cuando ya no se la necesita. El intervalo entre la creación y la eliminación es llamado ***tiempo de vida*** de una variable. Sólo durante este tiempo la variable ocupa espacio de almacenamiento. Luego de que es eliminada la/s celda/s de almacenamiento puede ser utilizada para otro propósito.

Variables globales y locales

Una variable local es aquella que se declara en un bloque y que se usará solo en dicho bloque (por ejemplo un bloque es el cuerpo de una función o procedimiento en Pascal).

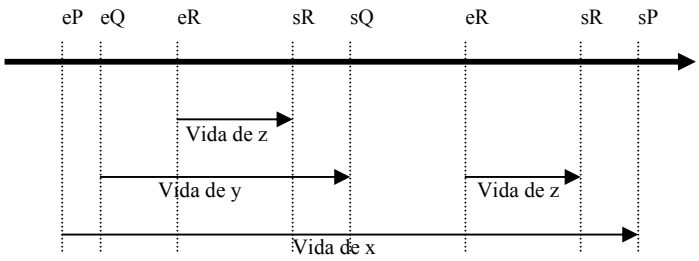
Una variable global es aquella que se declara en el bloque más externo de un programa.

La activación de un bloque es el intervalo de tiempo durante el cual ese bloque se ejecuta. En los lenguajes del estilo Algol, el tiempo de vida de una variable local corresponde exactamente a una activación de un bloque que contiene la declaración de la misma.

Considere el siguiente esqueleto de un programa Pascal:

```
program P;
  var x: ...;
  procedure Q;
    var y: ...;
    Begin
    ... R ...
    end
  procedure R;
    var z: ...;
    begin
      ...
    end;

  begin
   ...  Q ... R ...
  end.
```



Los bloques en el programa son el programa principal P (el bloque más externo), y los procedimientos Q y R. Suponga que P llama a Q y este a R y que a su vez P llama más tarde a R nuevamente. La figura muestra los diferentes tiempos de vida de las variables `x`, `y`, `z`.

Una variable local puede tener diferentes tiempos de vida si el bloque en el que está es activado varias veces. En el ejemplo R es activado en dos momentos distintos por lo tanto los tiempos de vida de z son disjuntos. Si el bloque R es activado recursivamente los tiempos de vida de z serán anidados. Esto tiene sentido si se entiende que cada tiempo de vida de z es realmente el tiempo de vida de una variable distinta, que es creada al inicio de la activación del bloque y eliminada al final de la misma.

Algunos lenguajes como PL/I y C permiten definir variables estáticas (`static`) cuyo tiempo de vida es toda la ejecución del programa aunque se definan dentro de un bloque interno, aunque solo son *alcanzables* desde el bloque en que fueron definidas.

Variables de pila (montículo)

Los tiempos de vidas anidados que son característicos de las variables locales y globales es adecuado para muchos propósitos, pero no para todos. A veces es necesario que las variables se creen y se eliminen en función a voluntad.

Este tipo de variables pueden ser creadas y borradas en cualquier momento. Son anónimas y se crean con un comando. Son accedidas a través de un puntero.

Un programa puede construir complejas estructuras de datos, inclusive incorporando a los punteros, que representan las conexiones entre los nodos. Tales estructuras pueden ser actualizadas selectivamente, se pueden agregar y eliminar nodos, cambiar la conexión que existe entre ellos, etc. manipulando los punteros.

Los tiempos de vida de las variables de este tipo no siguen un patrón particular. La creación de una variable de pila se realiza por una operación que retorna un puntero a la variable de pila creada recientemente, la cual estará disponible durante el tiempo en que el puntero lo apunte, hasta que deja de apuntarlo o se la elimine. En Pascal `new` reserva memoria para una variable de pila y `dispose` la libera.

Variables Persistentes

Una variable persistente es aquella cuyo tiempo de vida va más allá de la activación de un determinado programa. En contraste una variable transitoria es aquella cuyo tiempo de vida está comprendido en la activación del programa en el cual es creada (var. locales, de pila.)

Los parámetros a un programa son variables persistentes porque existen cuando la activación de un programa comienza y continua hasta después que la activación termina.

Comandos

Habiendo considerado la variables y el almacenamiento, examinemos ahora los comandos. Un comando es una frase de programa que será ejecutada para actualizar variables.

Los comandos son una característica de los lenguajes de programación imperativos (como el Fortran, Cobol, Algol, etc.)

Hay varios tipos de comandos. Unos son comandos primitivos y otros son compuestos de comandos más simples.

Saltos

Es un comando que no tiene efecto alguno (skip). Ej `if E then C else skip`

Asignaciones

Tienen la típica forma V := E. Actualiza la variable V con el valor retornado por la evaluación de la expresión E.

Ej:
```
(if ... then m else n) := 7
m := n := 0          (asignación multiple)
m, n := n, m         (asignación simultánea)
n += 1               (asignación combinada)
```

Llamadas a procedimientos

Aplica una abstracción de procedimiento a una lista de parámetros.

Comandos secuenciales

El orden en que se ejecutan los comandos es importantes según se actualizan las varibles.

Comandos colaterales

El orden en que se ejecutan los comandos no es irrelevante. Ej: `m:= 7, n:= n+1`.

Comandos condicionales

Tiene un número de subcomandos de los cuales exactamente uno será elegido para ejecutarse. El típico comando es el `if`. (`case`)

Comandos iterativos

También conocido como loop. Tiene un grupo de subcomandos que se ejecutarán repetidamente con un tipo de frase que determina cuando terminará la iteración. (`for, repeat, while`)

Enlace

Un concepto común a todos los lenguajes de programación es la habilidad del programador para enlazar o relacionar identificadores con entidades tales como constantes, variables, procedimientos, funciones y tipos. La buena elección de identificadores ayuda a hacer que los programas sean más comprensibles y fáciles de interpretar. Más aún, relacionar un identificador con una entidad en un lugar y luego utilizar ese identificador para hacer referencia a esa entidad en otros lugares, hace que los programas sean más flexibles: si la implementación de la entidad cambiara, los cambios se deberían hacer en un único lugar y no en todos los lugares donde se usa.

Enlace y ámbitos

Considere la expresión `n+1` y `f(x)`, y comandos tales como `x := 0`. No pueden ser interpretados aisladamente. La interpretación depende de lo que denota cada identificador y de cómo fue declarado.

La mayoría de los lenguajes de programación permiten definir un identificador en diferentes partes de un programa, inclusive representando entidades distintas.

Por ejemplo, considere la siguiente declaración Pascal:

```
Const n = 7
```

`n` representa el número entero 7, por lo tanto la expresión `n+1` significa sumar 7 más 1.

Por otro lado, si consideramos la siguiente declaración:

```
Var n: integer;
```

`n` representa una variable y por lo tanto la expresión `n+1` significa sumar uno al contenido actual de la variable `n`.

Podemos entender los efectos de las declaraciones teniendo en cuenta los conceptos de *enlace* y *ámbitos*. Podemos decir que una declaración produce una asociación o un **enlace** entre el identificador declarado y la entidad que denotará. La definición de la constante del ejemplo anterior relaciona o enlaza `'n'` al número 7. La declaración de variable enlaza `'n'` a una nueva variable creada.

Un *entorno* o *ámbito* es un conjunto de enlaces. Cada expresión y comando se interpreta en un ámbito particular y todos los identificadores que se encuentran en la expresión o comando, deben haberse enlazado en ese entorno. Expresiones y comandos en diferentes partes del programa deben interpretarse como diferentes ámbitos.

Por ejemplo, considere en siguiente programa Pascal:

```
Program p;
      Const z = 0;
      Var   c: char;

      Procedure q;
            Const c = 3.0e6;
            Var   b: boolean;

            Begin
        (2) ...
            end;

      begin
  (1) ...
      end.
```

El ambito en el punto (1) es:

{ `c` → una variable de tipo carácter,
`q` → un procedimiento,
`z` → el entero 0}

El ámbito en el punto (2) es:

{ `b` → una variable que contendrá un valor de verdad,
`c` → el número real 3000000.0,
`q` → un procedimiento,
`z` → un entero 0 }

Una característica de los lenguajes de programación tiene que ver con qué entidades pueden ser enlazados a identificadores. Estas entidades se denominan *enlazables* del lenguaje. (También pueden llamarse denotables).

Por ejemplo en Pascal son:

- Valores primitivos y strings — en la definición de constantes
- Referencias a variables — en la declaración de variables
- Procedimientos y funciones — en la definición de procedimientos y funciones
- Identificadores de tipos — en las definiciones de tipos.

Note que solo ciertos tipos de valores pueden enlazarse en la definición de constantes en Pascal. Por ejemplo, no se puede enlazar un identificador a un valor de registro, ni a un arreglo (que no sea un string), ni a un conjunto.

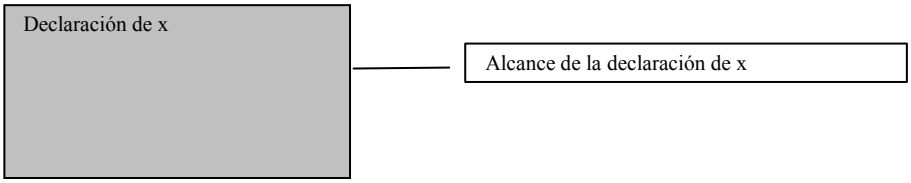
Alcance

Solo en un lenguaje muy simple cada declaración afecta el ámbito del programa completo. En general cada declaración tiene un cierto *alcance*, que tiene que ver con la porción del programa sobre el cual al declaración es efectiva. Para entender el concepto introduciremos la idea de *estructura de bloque*.

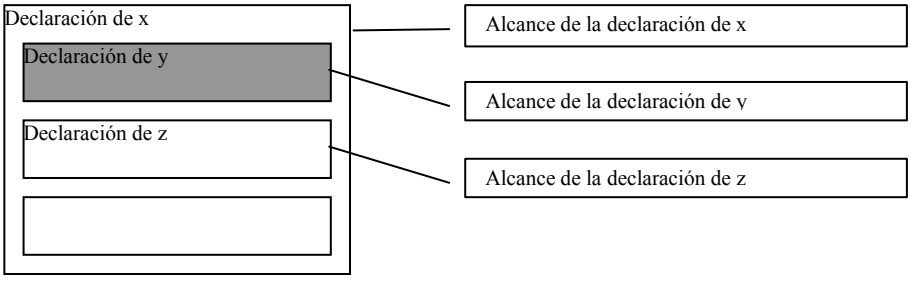
Estructura de bloque

Un bloque es una frase del programa que delimita el alcance de cualquier declaración que puede contener. En particular centraremos nuestra atención en como bloques pueden ser anidados en otros bloques.

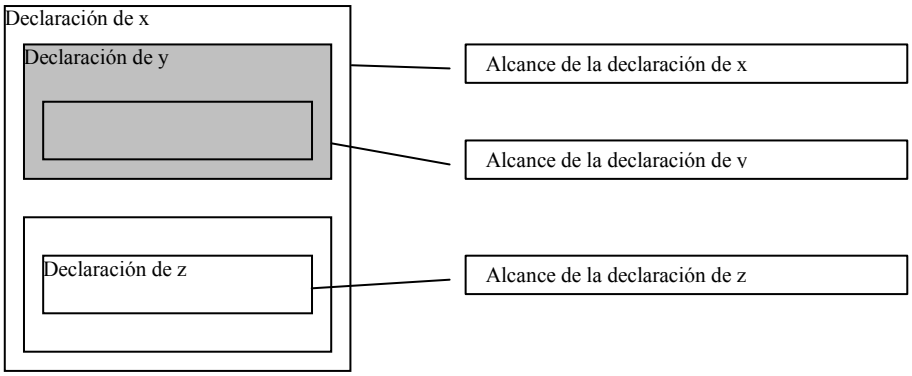
Las figuras siguientes muestran los tres tipos de estructura de bloques. Cada bloque está representado por un recuadro. El alcance de algunas declaraciones están sombreadas.



Estructura de bloque único



Estructura de bloque plano



Estructura de bloque anidado

La estructura de bloque más simple posible es la de bloque único en la cual el programa entero es un bloque simple. Por ejemplo viejas versiones de Cobol. El alcance de cualquier declaración es el programa completo. Todas las declaraciones deben agruparse en el mismo lugar, esto permite asegurarse que todas las entidades declaradas tengan distintos identificadores.

El segundo caso, en donde se tiene una estructura de bloques plano, consiste en que el programa es particionado en distintos bloques. El Fortran es un lenguaje que implementa este tipo de estructura, en el cual todos los subprogramas están separados y cada uno actúa como un bloque. Una variable pude declararse en un subprograma particular y es entonces local a ese subprograma.

El alcance de los identificadores de subprogramas es el programa completo, así como el alcance de las variables globales es el programa completo. Una desventaja de esta estructura de bloques es que todos los subprogramas y variables globales deben tener distintos identificadores. Otra desventaja es que cualquier variable que no puede se local a un subprograma en particular es forzado a ser global aún cuando solo es utilizado por un par de subprogramas.

La estructura de bloques más popular es la anidada donde cada bloque puede ser anidado en otro bloque. Los lenguajes tipo Algol utilizan este concepto. Un bloque puede ser colocado conde sea conveniente y los identificadores se declaran dentro de los mismos.

- Los bloques pueden ser abstracciones de procedimiento y función pero también pueden ser bloques de comandos.

Supongamos este ejemplo en donde `m` y `n` son variables enteras y sus valores son ordenados de tal manera que `m` contenga el menor valor. En Pascal escribiríamos esto:

```
If m > n then
      Begin t:= m; m:= n; n:= t end
```

Pero la variable auxiliar `t` debe declararse en algún lugar, quizás en el encabezado del programa o del cuerpo de la función o procedimiento aunque sea utilizado solo en este lugar.

Si Pascal contara con el concepto de bloques de comandos podríamos declarar `t` localmente:

```
If m > n then
      var t : integer;
      begin t:= m; m:= n; n:= t end
```

esta localización tiende a hacer a los programas más fáciles de entender y de modificar.

- Un concepto aún más potente es el de expresiones de bloque. Es una expresión que es evaluada para producir un enlace con un identificador que se usará solo en el bloque donde se define.

ML maneja este concepto de la forma "**`let`** *D* **`in`** *E* **`end`**". Supongamos que a, b y c son los lados de un triángulo. La siguiente expresión de bloque retorna el área ese triángulo:

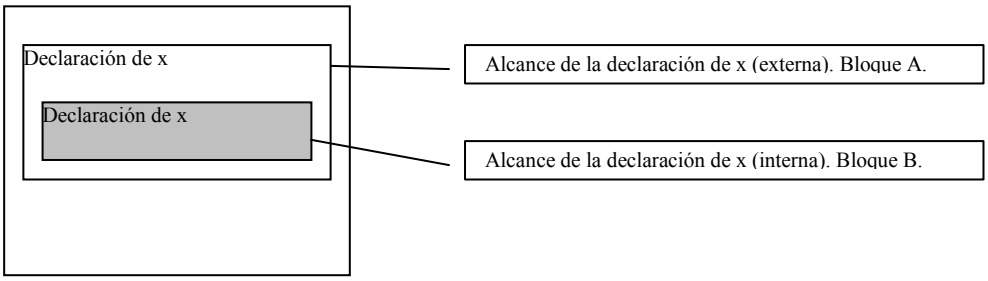
```
let val s = (a + b + c) * 0.5
in sqrt (s * (s-a) * (s-b) * (s-c))
end
```

Alcance y visibilidad

Los identificadores pueden aparecer en dos contextos diferentes. Llamaremos ***ocurrencia de enlace*** al punto en donde un identificador se declara, y llamaremos ocurrencia de aplicación a cada aparición del identificador cuando denota a la entidad a la cual fue enlazado.
Por ejemplo:
La aparición de `n` en `const n = 7` es una ocurrencia de enlace, en donde `n` se enlaza al 7.
Las apariciones subsecuentes de `n` en la expresión `n * (n+1)` son ocurrencias de aplicación, en donde `n` denota al 7.

Cuando un programa posee más de un bloque, es posible que un identificador se declarado en distintos bloques. En general el identificador puede denotar entidades distintas en cada bloque (aunque no es aconsejable que así lo sea).

Esto da la libertad a los programadores a declarar identificadores sin tener en cuenta si ya fueron declarados en otros bloques. Pero que pasa si un identificador es declarado en dos bloques anidados (ver figura). La mayoría de los lenguajes con estructura de bloques anidados permiten esto.



Supongamos que dentro del alcance del bloque A está la definición del identificador I, este tiene un alcance en todos los bloques anidados de A y cualquier ocurrencia de aplicación de I se referirá al declarado en A. Si el bloque B posee una declaración del identificador I, cualquier ocurrencia de aplicación dentro del bloque B se referirá al I declarado en B. La declaración de I en el bloque A se dice que es invisible dentro de B o que está oculto por la declaración de I en B. Todas las demás declaraciones en A o dentro del programa principal son visibles dentro de B.

Enlace estático y dinámico

La terminología de enlace estático y dinámico tiene que ver con el momento en que determinamos que ocurrencia de enlace del identificador I se corresponde con la ocurrencia de aplicación de I.

Con enlace estático podemos hacer esto en tiempo de compilación.

Con enlace dinámico debemos retardarlo hasta el tiempo de ejecución.

Con el enlace estático podemos determinar que ocurrencia de enlace de I se corresponde con una ocurrencia de aplicación de I dada, solo con examinar el código del programa, encontrando el bloque más pequeño que contiene una ocurrencia de aplicación de I que también contiene una ocurrencia de enlace de I. La asociación entre la ocurrencia de aplicación y enlace es fija.

Con enlace dinámico la ocurrencia de enlace de I que se corresponde con una ocurrencia de aplicación de I depende del flujo de control dinámico del programa. La entidad denotada por I es la declaración elaborada más recientemente de I dentro del bloque activo actual.

Considere en siguiente código:

```
const s = 2;

function escalar (d: integer): integer;
      Begin
        escalar := d * s;
      end;

procedure ... ;
      Const s = 3;

      Begin
(2)   ... escalar (h) ...
      end;

begin
(1) ... escalar (h) ...
end.
```

El resultado de `escalar (h)` depende de cómo interpretemos la ocurrencia de `s` en el cuerpo de la función `escalar`.

La interpretación más común es llamada ***Enlace Estático***. El cuerpo de la función es evaluado en el ámbito de la *definición* de la función. En esta punto `s` denota 2 por lo tanto la `escalar (h)` siempre retornará el doble de `h`, sin tener en cuenta de dónde se llama a la función.

Otra alternativa de interpretación es llamada ***Enlace Dinámico***. El cuerpo de la función es evaluado en el ámbito de la *llamada* a la función. Por lo tanto la llamada a la función `escalar (h)` en el punto (1) retornará el doble de `h` debido a que `s` denota 2 en este punto. Por otro lado la llamada a la función `escalar (h)` en el punto (2) retornará el triple de `h` debido a que `s` denota 3 en este punto.

Es evidente que el enlace dinámico no encaja con el control d tipos estático. Si consideramos nuevamente el ejemplo anterior y reemplazamos la definición global de `s` por **`const`** `s = 0.5`, con el enlace estático se detectará un error de tipos en `escalar := d * s`. Con el enlace dinámico la llamada a la función en el punto (2) no será afectada pero en el punto (1), producirá un error de tipos en tiempo de ejecución. Por esta razón los lenguajes con enlace dinámico (como el Lisp y Smalltalk) también utilizan el tipeo dinámico. El problema ilustrado en el enlace dinámico tiende a hacer que las funciones y procedimientos sean difíciles de entender. Si un procedimiento P accede a una constante o variable no local, o llama a una función o procedimiento no local, el efecto dependerá de dónde se llama a P. Debido a esto lenguajes de programación del estilo Algol, optan por el enlace estático.

Abstracciones

Abstracción es un modo de pensamiento en el cual nos concentramos en las ideas generales más que en las manifestaciones específicas de estas ideas. En el análisis de sistemas, la abstracción es la disciplina por la cual nos concentramos en los aspectos esenciales del problema y se ignoran aspectos irrelevantes.

En programación, la abstracción alude a la distinción entre (a) qué hace una parte de un programa y (b) cómo está implementado. Un lenguaje de programación contiene construcciones que son en última instancia abstracciones del lenguaje de máquina. Pero esto no es el final de la historia, se explotan las abstracciones cuando se introduce el concepto de función o procedimiento. Cuando se llama a un procedimiento nos concentramos sólo en *qué* hace dicho procedimiento; sólo cuando escribimos un procedimiento nos interesará el *cómo* implementarlo. Al implementar procedimientos de más alto nivel a partir de procedimientos de más bajo nivel, los programadores pueden introducir tantos niveles de abstracción nuevos como deseen.

Estudiaremos varios tipos de abstracción, tales como procedimientos y funciones; parametrización de abstracciones con respectos a los valores de los que depende y las relaciones entre parámetros y enlace.

Tipos de abstracción

Definiremos una abstracción como una entidad que engloba una computación. Por ejemplo una abstracción de función engloba una expresión a evaluarse, una abstracción de procedimiento engloba un comando a ejecutarse. Dicha computación es llevada a cabo cuando la abstracción es llamada. La eficiencia de la abstracción es aumentada con la parametrización.

Abstracción de función

Una abstracción de función engloba una expresión a evaluarse, y cuando es llamada retorna un valor como resultado. El usuario de la abstracción observa solo su resultado, no los pasos por los cuales se evalúa la función.

Una abstracción de función es construida a partir de una definición de función:

```
Function I (fp1;  ... ; fpn) : T;
      B
```

Donde fp_i son los parámetros formales, y donde B es el bloque que contiene al menos un comando de la forma `I := E`. La abstracción de función se llamará con una expresión del tipo:

`I (ap1, ..., apn)` donde a_i son los parámetros actuales.

Por ejemplo; considere la siguiente abstracción de función en ML:

```
fun potencia (x: real, n: int) =
      if n = 1 then x
               else x * potencia (x, n - 1)
```

esta definición enlaza `potencia` a una abstracción de función. La vista del usuario de esta abstracción de función es que mapea cada par real-entero (x, n) a x^n. La vista del implementador es la manera en cómo se calcula el resultado, como está codificado el algoritmo de cálculo.

Abstracción de procedimiento

Una abstracción de procedimiento engloba un comando a ejecutarse, y cuando es llamado actualizará variables. El usuario de la abstracción de procedimiento observa solo esas actualizaciones y no lo pasos por los cuales fueron efectuados

Una abstracción de procedimiento es construido en Pascal, a partir de una definición de procedimiento:

```
Procedure I (fp1;  ... ; fpn);
      B
```

Donde fp_i son los parámetros formales, y donde B es un bloque. La abstracción de procedimiento se llamará con un comando del tipo:

`I (ap1, ..., apn)` donde a_i son los parámetros actuales.

Por ejemplo:

```
type ArregloPalabras = array [...] of word;
procedure sort (var words: ArregloPalabras);
      ...
```

esto enlaza el identificador `sort` a una abstracción de procedimiento. El cuerpo del procedimiento podría, por ejemplo, implementar el algoritmo de inserción. Esta es la visión del implementador. La visión del usuario es que un comando del tipo `sort (diccionario)` tendría el efecto de ordenar los valores que contiene el arreglo `diccionario`. Una implementación del algoritmo QuickSort en la abstracción produciría una ejecución más eficiente, pero el usuario observaría exactamente el mismo efecto neto.

El principio de abstracción

Como un resumen podemos decir:

- Una *abstracción de función* es una abstracción sobre una *expresión*. Quiere decir que una *abstracción de función* tiene un cuerpo que es una *expresión*, y una *llamada a la función* es una *expresión* que *retornará un valor* al evaluar el cuerpo de la abstracción de función.
- Una *abstracción de procedimiento* es una abstracción sobre un *comando*. Quiere decir que una *abstracción de procedimiento* tiene un cuerpo que es un *comando*, y una *llamada a procedimiento* es un *comando* que *actualizará variables* al ejecutar el cuerpo de la abstracción de procedimiento.

Es posible construir abstracciones sobre cualquier clase sintáctica, con tal que las frases de esa clase especifiquen algún tipo de computación.

Por ejemplo, Pascal posee una clase sintáctica para acceso a variables. Un acceso a variable retorna una referencia a una variable. Imagine expandir el lenguaje Pascal diseñando una abstracción sobre acceso a variables. Un tipo de abstracción que cuando es llamada retorna una referencia a una variable. La llamaremos abstracción de selección.

- Una *abstracción de selección* es una abstracción sobre el *acceso a variable*. Quiere decir que una *abstracción de selección* tiene un cuerpo que es un *acceso a variable*, y una *llamada al selector* es un *acceso a variable* que *retornará una referencia a una variable* al evaluar el cuerpo de la abstracción de selección.

Por ejemplo considere las siguientes definiciones Pascal:

```
type cola = ...    // cola de enteros
function primero (q: cola) : integer;
      ...          // retorna el primer entero en q
```

con una llamada a la función del tipo `i := primero (UnaCola)`, permite recuperar el primer entero de `UnaCola`.
Pero no hay manera de actualizarlo de este modo `primero (unaCola) := 0`. (Por qué no?)

Suponga ahora que Pascal permite la posibilidad de utilizar abstracciones de selección. Podemos definir `Primero` para que sea un `selector` y no una `function`.

```
selector primero (var q: cola) : integer is
      ...          // retorna una referencia del primer elemento de q
```

esto permitiría escribir las siguiente llamadas, debido a que el selector retorna una referencia a una variable y no el valor actual.

```
i:= primero (UnaCola);
primero (UnaCola) := primero (UnaCola) + 1
```

Parámetros

Si simplemente hay una expresión dentro de una función o un comando dentro de un procedimiento, construiremos una abstracción que realiza más o menos lo mismo cada vez que es llamada. Para aprovechar la potencia del concepto de abstracción, necesitamos abstracciones parametrizadas respecto a los valores con que opera.

Por ejemplo la siguiente abstracción de función en ML:

```
val pi = 3.1416;
val r = 1.0;
fun circunf () = 2 * pi * r.
```

La llamada a la función `circunf()` retorna la circunferencia de un circulo de radio 1. En otras palabras, una llamada a esta función siempre realiza la misma computación mientras `r` esté enlazado al mismo valor.

Podemos hacer que la abstracción de función sea más util mediante la parametrización respecto de `r`.

```
fun circunf (r: real) = 2 * pi * r
```

Ahora podemos llamar a la función con diferentes parámetros y calcular circunferencias de distintos círculos.

Un identificador que se usa en una abstracción para denotar un argumento se ***denomina parámetro formal***. Una expresión o frase que se pasa como argumento es denominado ***parámetro actual*** por ejemplo `1.0` o `a+b` en las llamadas `circunf (1.0)` y `cicunf (a+b)` respectivamente.

Un ***argumento*** es el valor que puede pasarse a una abstracción

Cuando se llama a abstracción, cada parámetro formal estará asociado, en algún sentido, con su correspondiente argumento. Estudiaremos ahora en detalle las posibles asociaciones entre los parámetros formales y actuales, lo que se conoce con el nombre de mecanismos de parámetros.

Diferentes lenguajes proveen una variedad de estos mecanismos, por ejemplo parámetros por valor, parámetros resultado, parámetros constantes, parámetros variables etc. Todos estos mecanismos pueden ser entendidos en términos de un pequeño número de conceptos.

Mecanismo de copia

Un mecanismo de copia permite que valores se copien a y/o desde una abstracción cuando se la llama. El parámetro formal X denota una variable local para la abstracción. Un valor se copia en X a la entrada de la abstracción, y/o se copia de X (a una variable no local) a la salida de la abstracción. Debido a que X es una variable local, se crea a la entrada de la abstracción y se elimina a la salida.

Un parámetro por valor es una variable local X que se crea a la entrada de la abstracción y se le asigna el valor del argumento. Debido a que X se comporta como una variable local, su valor puede ser inspeccionado y actualizado. Sin embargo cualquier actualización de X no tiene efecto en ninguna variable no local.

Un parámetro resultado es todo lo contrario del anterior. En este caso, el argumento debe ser una referencia a una variable. De nuevo una variable local X se crea, pero su valor inicial es indefinido. A la salida de la abstracción el valor final de X es asignado a la variable argumento.

Estos dos mecanismos pueden ser combinados para dar resultado a los parámetros valor-resultado. El argumento, de nuevo, debe ser una referencia a variable. En la entrada de la abstracción, la variable local X se crea y se le asigna el valor actual de la variable argumento. A la salida, el valor final de X se asigna nuevamente a la variable argumento.

Por ejemplo, considere el siguiente pseudo código pascal:

```
Type vector = array [1..n] of real;
Procedure sumar (value v, w: vector; result sum: vector);
  (1) Var i: 1..n;
      Begin
            For i := 1 to n do  sum[i] := v[i] + w[i];
  (2) End;
```

```
Procedure normalizar (value result u: vector);
  (3) Var i : 1..n; s: real;
      Begin
            s:= 0.0;
            for i := 1 to n do  s := s + sqr (u[i])
            s := sqrt(s)
            for i := 1 to n do u[i] := u[i]/s
  (4) end;
```

Supóngase que `a`, `b` y `c` son variables del tipo vector. La llamada al procedimiento `sumar (a, b, c)` tiene el siguiente efecto: en el punto (1), se crean las variables locales `v` y `w` y se les asigna los valores de `a` y `b` respectivamente; se crea la variable local `sum` pero no se inicializa. El cuerpo de la función `sumar` luego asigna valores a los componentes de `sum`. En el punto (2), el valor final de `sum` se asigna a `c`.

La llamada al procedimiento `normalizar(c)` tiene el siguiente efecto: en el punto (3), se crea la variable local `u` y se le asigna el valor de `c`; el cuerpo de `normalizar` luego actualiza los componentes de `u`. En el punto (4), el valor final de `u` se asigna nuevamente a `c`.

Debido a que este mecanismo se basa en el concepto de asignación, este mecanismo no es aplicable a tipos donde no se puede aplicar la asignación (por ejemplo archivos en Pascal). Otra desventaja es que la copia de valores compuestos grandes se hace muy cara.

En la siguiente tabla se resume este mecanismo.

Mecanismo	*Argumento*	*Efecto de entrada*	*Efecto de salida*
Parámetro valor	Valor de 1ª clase	X:= argumento	-
Parámetro resultado	Una variable	-	argumento := X
Parámetro valor-resultado	Una variable	X := argumento	argumento := X

Mecanismo por definición

Este mecanismo permite que el parámetro formal X se enlace directamente al argumento. Esto da una semántica simple y uniforme para el paso de parámetros de cualquier valor en el lenguaje de programación (no solo los valores de primera clase).

- En el caso de un *parámetro constante*, el argumento es un valor (de primera clase). X se enlaza al valor del argumento durante la activación de la llamada a la abstracción.
- En el caso de un *parámetro variable (o por referencia)*, el argumento es una referencia a una variable. X se enlaza a la variable argumento durante la activación de la llamada a la abstracción. Por lo tanto cualquier actualización o inspección de X es realmente una actualización o inspección del argumento.
- En el caso de un *parámetro procedural*, el argumento es una abstracción de procedimiento. X se enlaza al procedimiento (argumento) durante la activación de la llamada a la abstracción. Por lo tanto cualquier llamada a X es realmente una llamada al procedimiento (argumento).
- En el caso se un *parámetro funcional*, el argumento es una abstracción de función. X se enlaza a la función (argumento) durante la activación de la llamada a la abstracción. Por lo tanto cualquier llamada a X es realmente una llamada a la función (argumento).

Es importante observar que estos no son mecanismos distintos. En cada caso, el efecto es como si el cuerpo de la abstracción esté rodeado por un bloque en el cual hay una definición que enlaza X al argumento — de ahí, nuestra terminología de mecanismo por definición.

Por ejemplo, considere el siguiente pseudo código Pascal similar al ejemplo anterior:

```
Type vector = array [1..n] of real;
Procedure sumar (const v, w: vector; var sum: vector);
  (1) var i: 1..n;
      Begin
            For i := 1 to n do  sum[i] := v[i] + w[i];
  (2) End;
```

```
Procedure normalizar (var u: vector);
  (3) Var i : 1..n; s: real;
      Begin
            s:= 0.0;
            for i := 1 to n do  s := s + sqr (u[i])
            s := sqrt(s)
            for i := 1 to n do u[i] := u[i]/s
  (4) end;
```

La llamada al procedimiento `sumar (a, b, c)` tiene el siguiente efecto: en el punto (1), `v` y `w` son enlazados a los valores de `a` y `b` respectivamente; y `sum` es enlazado a la variable `c`. El cuerpo de `suma` inspecciona indirectamente los componentes de `a` y `b` e indirectamente actualiza los componentes de `c`.

La llamada al procedimiento `normalizar (c)` tiene el siguiente efecto: en el punto (3), `u` se enlaza a la variable `c`. el cuerpo de `normalizar` inspecciona y actualiza indirectamente los componentes de `c`.

Note que no sucede nada en los puntos (2) y (4). Note también que tampoco se necesita una copia para implementar los parámetros constantes. Debido a que los parámetros formales `v` y `w` son constantes, no pueden realizar asignaciones a los mismos; de tal manera que los argumentos correspondientes `a` y `b` no pueden ser indirectamente actualizados, aún cuando estos parámetros se implementan mediante pasos de referencias de `a` y `b`.

Los ejemplos anteriores ilustran el hecho que parámetros constantes y variables proveen un poder similar al mecanismo de copia. La elección de uno de estos mecanismos es una importante decisión de los diseñadores de lenguajes.

El mecanismo por definición tiene una semántica más simple, y es aplicable a todos los tipos de valores (no solo para los tipos para los cuales la asignación está disponible). Además se apoya en el acceso indirecto de los argumentos, lo cual es a menudo más eficiente que la copia de los datos.

Una desventaja de los parámetros variable es la posibilidad de *crear alias*. Esto ocurre cuando dos o más identificadores son simultáneamente enlazados a la misma variable. Esto tiende a hacer que los programas sean más difíciles de entender.

Por ejemplo

```
Procedure confuso (var m, n : integer);
      begin
      n:= 1; n:= m + n
      end;
```

Si la variable `i` tiene actualmente el valor 4, es de esperar que la llamada a la función `confuso (i, i)` actualice `i` a 5 — y sería así si `m` es un parámetro por valor —. Pero el efecto que se produce es que `i` se actualiza a 2!. Esto es porque `m` y `n` son alias de `i`.

Orden de Evaluación

El orden de evaluación se refiere exactamente al momento en que cada parámetro actual se evalúa cuando se llama a una abstracción. Existen básicamente dos posibilidades:

- Evaluar el parámetro actual en el punto de la llamada.
- Retardar su evaluación hasta que el argumento realmente se usa.

Considere la función definida de la siguiente manera:

```
fun sqr (n: int) = n * n
```

y la llamada a dicha función `sqr (p + q)`. Suponga que el valor de `p` es 2 y que el valor de `q` es 5. Hay dos maneras diferentes de evaluar la llamada a la función:

- Primero evaluar `p+q` que retornaría 7, y luego enlazar el parámetro formal `n` a 7. Finalmente evaluar `n*n` lo cual retornaría 7 x 7 = 49.

- Primero enlazar el parámetro formal `n` a la expresión `p+q` en sí, luego cada vez que el valor de `n` es requerido durante la evaluación de `n*n`, reevaluaremos la expresión a la cual `n` fue enlazada. Por lo tanto haríamos el siguiente cálculo (2+5)x(2+5) = 49.

El primer orden de evaluación se denomina "**Evaluación Impaciente**" (o evaluación en orden aplicativo). Se evalúa el parámetro actual una sola vez, y en efecto se sustituye el resultado en cada ocurrencia del parámetro formal.

El segundo orden de evaluación se denomina "**Evaluación en orden normal**". No se evalúa inmediatamente el parámetro actual, sino que se sustituye el parámetro actual por cada ocurrencia del parámetro formal.

En el caso de la función `sqr`, ambos tipos de evaluación retornan el mismo resultado (a pesar de que evaluar en orden impaciente es más eficiente). Sin embargo algunas funciones pueden producir comportamiento distinto dependiendo del orden de evaluación. Considere el siguiente ejemplo:

```
fun cand (b1, b2 : bool) =
      if b1 then b2 else false
```

y la llamada a la función `cand (n>0, t/n>0.5)`. Primero suponga que el valor de `n` es 2 y el de `t` es 0.8:

- Con evaluación impaciente, `n>0` es verdadero, `t/n>0.5` es falso, por lo tanto la función retorna falso.
- Con evaluación en orden normal, se evaluará `if n>0 then t/n>0.5 else false`, lo cual retorna falso.

Pero ahora suponga que el valor de `n` es 0:

- Con evaluación impaciente, `n>0` es falso pero `t/n>0.5` falla (por división por 0), por lo tanto la llamada a la función falla.
- Con evaluación en orden normal, se evaluará `if n>0 then t/n>0.5 else false`, lo cual retorna falso.

La diferencia fundamenteal entre las funciones `sqr` y `cand` tiene que ver con que si sus argumentos realmente se necesitan. La función `sqr` se dice que es ***estricta***, lo cual significa que una llamada a esta función puede evaluarse solo si sus argumentos pueden ser evaluados. La función `cand` se dice que es ***no estricta*** en su segundo argumento, lo cual significa que una llamada a esta función puede a veces evaluarse aún cuando su segundo argumento pueda no ser evaluado. (La función `cand` es estricta en su primer argumento)

La evaluación en orden normal puede tener algún efecto colateral, por ejemplo:

Supongamos que `getint (f)` lee un entero de un archivo `f` y retorna ese entero. Consideremos también la función `sqr` definida arriba y la llamada a la función `sqr (getint (f))`.

- Con evaluación impaciente se leerá un entero y retornaría el cuadrado del mismo.
- Con evaluación en orden normal se leerían dos enteros y se retornaría su producto.

La evaluación en orden normal es claramente ineficiente cuando causa que el mismo argumento sea evaluado varias veces. Si el valor del argumento siempre es el mismo, puede ahorrarse tiempo si el valor del argumento es almacenado tan pronto como sea evaluado por primera vez, y el valor almacenado pueda ser usado cada vez que se necesite. Este evaluación se denomina perezosa (***lazy evaluation***); el argumento se evalúa solo cuando se la necesita por primera vez (que podría ser nunca si la función es no restricta)

Indice