Lenguajes de Programación
Conceptos Fundamentales y Lenguajes Imperativos

Dr. Mauricio Araya

Primer Semestre 2014

Agenda

Índice

1. Introducción a Lenguajes Formales

Conceptos Básicos

- Lenguaje (Formal) : Conjunto de cadenas de símbolos restringidas por reglas que son específicas a un lenguaje formal particular.
- Sintaxis : Conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje de programación.
- Semántica : Estudio del significado de los signos lingüísticos y de sus combinaciones.
- Alfabeto : Conjunto de símbolos, letras o tokens con el cual se puede formar la cadena de un lenguaje.
- Gramática (formal) : Estructura matemática con un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje formal.

Lenguajes de Programación

- **Sintaxis**: Forma de expresiones, sentencias y unidades de programa
- **Semántica**: Significado de estas expresiones, sentencias y unidades de programa
- **Alfabeto**: Conjunto de caracteres, palabras reservadas, parentesis, etc.
- **Gramática**: Analizador sintáctico del compilador o interprete

Example 1. **`while`** `(<expr>)` **`do`** `<sentencia>`

1.1. Sintaxis y Gramática

Sintaxis

Definición de Sintaxis: La sintaxis de un lenguaje es la descripción precisa de todos los strings gramaticalmente correctas. Métodos formales para definir precisamente la sintaxis de un lenguaje son importantes para el correcto funcionamiento de los traductores y programas (e.g. usando BNF).

Una sintaxis se define

- **Por Reconocimiento:** Reconoce si el string de entrada pertenece al lenguaje (práctico)
- **Por Generación:** Genera todos los strings que pertenecen al lenguaje (formal)

Formalmente...

- **Reconocimiento**: Una maquina sintáctica R acepta o no un string S en el lenguaje $\mathcal{L}$ y alfabeto Σ.
- **Generación**: Una especificación G generara el conjunto de todos los strings $\mathcal{S} = \{S_1, S_2, \ldots\}$ tal que S_i pertenece al lenguaje $\mathcal{L}$ y alfabeto Σ.

Gramática de Libre Contexto

Se define por una 4-tupla $G = (V, \Sigma, P, S_0)$

- V: Conjunto finito de símbolos no terminales o variables. Cada variable define un sub-lenguaje de G.
- Σ: Conjunto finito de símbolos terminales , que define el alfabeto de G.
- P: Relación finita $V \to (V \cup \Sigma)^*$. Cada miembro de P es una regla de producción de G.
- S_0: Símbolo de partida, donde $S_0 \in V$.

Nota: siempre recuerde que una gramática permite expresar formalmente la sintaxis de un lenguaje (formal).

Ejemplo de Gramática

Example 2. $G = (V, \Sigma, P, S_0)$, con: $V = \{S, X, Y, Z\}$, $\Sigma = \{a, b\}$, $S_0 = S$, y donde P tiene 4 reglas de producción:

- $S \to X|Y$
- $X \to ZaX|ZaZ$
- $Y \to ZbY|ZbZ$
- $Z \to aZbZ|bZaZ|\epsilon$

Esta gramática genera todos las cadenas con un número desigual de símbolos a y b.

1.2. Backus-Naur Form

Backus-Naur Form

La notación BNF es un metalenguaje, creado por John Backus (1959), que permite especificar formalmente gramáticas libres de contexto.

Elementos de BNF

- **Símbolos Terminales**: Elementos en el dominio del lenguaje. ”Tokens”
- **Símbolos no Terminales**: Secuencia de símbolos terminales y no terminales. ”Reglas”
- **Símbolo de Partida**: Símbolo no terminal que permite generar strings pertenecientes al lenguaje

Especificación de un BNF

Reglas de producción se especifican como:

```
<simbolo> ::= <expresion con simbolos>
```

Características

- Lado izquierdo corresponde a un símbolo no terminal.
- Lado derecho (expresión) representa posible sustitución para símbolo terminal.
- Expresión usa | para indicar alternación (opciones).
- Símbolos terminales nunca aparecen a la izquierda.
- Normalmente, primera regla corresponde a símbolo no terminal de partida.

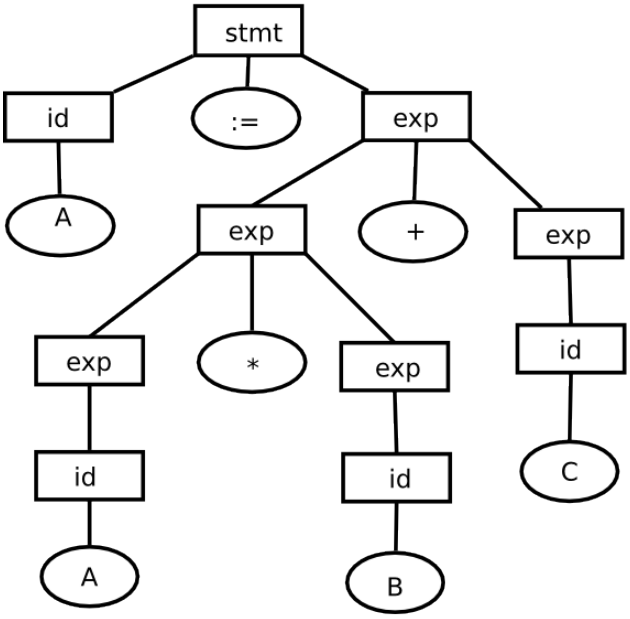
Gramática en BNF

Example 3. `<program> ::= begin <stmtlst> end`

```
<stmtlst> ::= begin <stmt> ; <stmtlst> end | NULL

<stmt>    ::= <id> := <exp>

<id>      ::= A | B | C

<exp>     ::= (<exp>)

<exp>     ::= <id> | <exp> + <exp> | <id> * <exp>
```

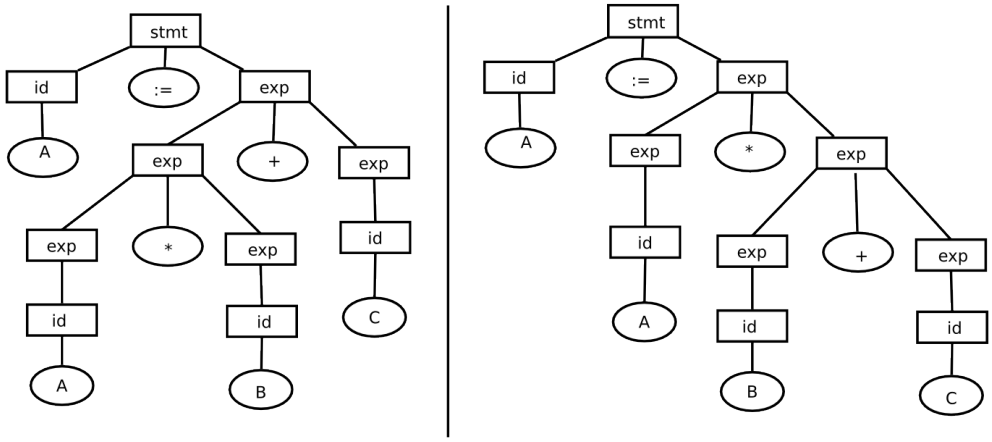
Parse Tree

Expresión: `A := A*B + C`



Ambigüedad

Para `A := A*B + C` existen dos Parse Tree

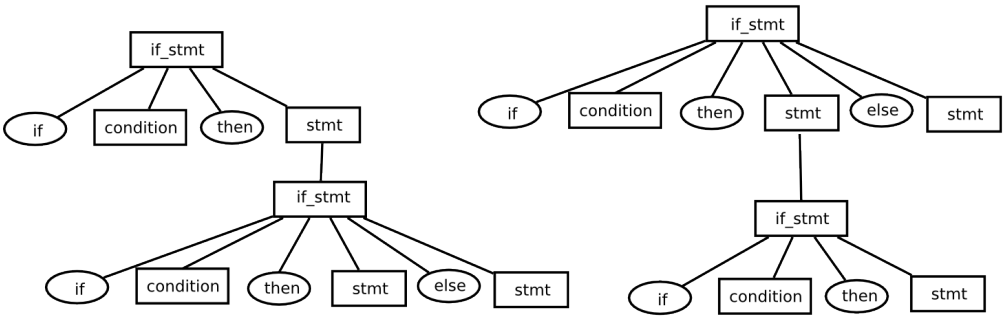


Otro Ejemplo de Ambigüedad

Dada la siguiente gramática:

```
<if_stmt> ::= if <condition> then <stmt>
| if <condition> then <stmt> else <stmt>
<stmt>    ::=  <if_stmt> | ...
```

Reconocer: `if (C1) then if (C2) then S1 else S2`



Resolución de Ambigüedad

Por construcción
Agregar símbolos que agrupen, como ( ), [ ].

Por precedencia
Ciertos símbolos se procesan antes, como * antes de +. Si tienen igual precedencia, por orden sintáctico (i.e. de izquierda a derecha)

Por anidación
Según la profundidad de la anidación. Ejemplo: Asociar el `else` con el último `if`.

Extended BNF (EBNF)

- Elemento *opcional* se indica con `[...]`
- Alternativa puede usar `|` en una regla
- Repetición de elementos se indica con {...}

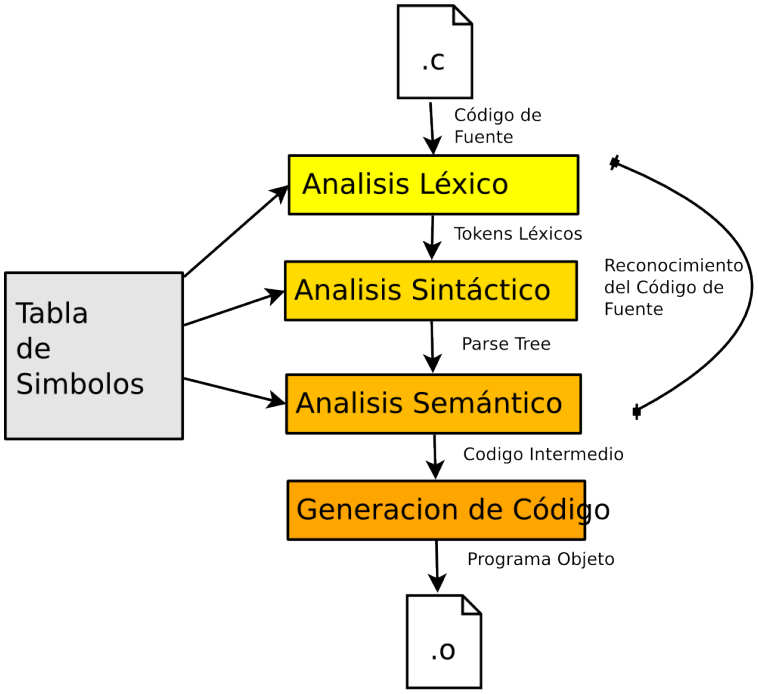
Example 4.
```
<if_stmt> ::= if <condition> then <stmts>
              [ else <stmts>]
<identifier> ::= <letter> {<letter> | <digit> }
```

Elementos Sintácticos en Lenguajes de Programación

- Conjunto de caracteres
- Identificadores
- Símbolos de operadores
- Palabras claves y reservadas
- Comentarios
- Blancos, delimitadores y paréntesis
- Expresiones
- Sentencias

2. Compilación y Enlazado

Proceso de Compilación



Proceso de Compilación

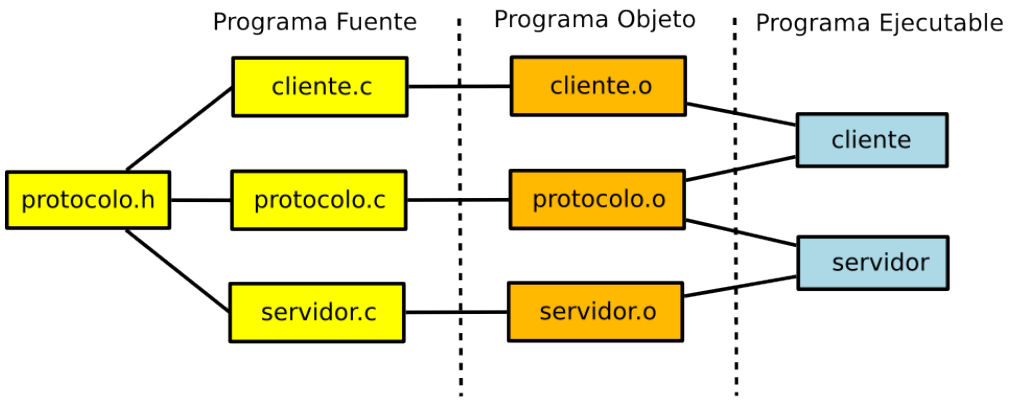
Programa fuente
Un programa en el lenguaje de alto nivel.

Programa objeto
Un programa a nivel de lenguaje de máquina que resulta de la compilación de un programa fuente. Contiene una tabla de símbolos para efectuar el enlazado.

Programa ejecutable
Un programa a nivel de lenguaje de máquina que resulta del enlazado de varios programas objetos.

Símbolos
Nombres de variables, funciones, y otros componentes necesarios para el enlazado. Se agrupan en una *tabla de símbolos*.

Ejemplo en C



Ejemplo en C

- El archivo de *cabecera* `protocolo.h` se incluye textualmente al compilar, contiene declaraciones requeridas para usar definiciones en `protocolo.c` en los demás archivos
- Tanto `cliente.c` como `servidor.c` pueden ocupar funciones y variables de `protocolo.c`, el compilador será capaz de verificar su empleo correcto siempre y cuando estén declaradas en `protocolo.h`.
- Para generar el ejecutable es necesario *enlazar* las funciones y variables exportadas.
- Las cabeceras del sistema (como `stdio.h`) contienen declaraciones para uso de la biblioteca estándar de C.

protocolo.h

```
#ifndef _PROTOCOLO_H_
#define _PROTOCOLO_H_

struct datos {
        int codigo;
        int tipo;
        char mensaje[1000];
};

int enviar_datos(struct datos *mis_datos);
struct datos *recibir_datos(void);
#endif
```

protocolo.c

```
#include "protocolo.h"

int enviar_datos(struct datos *mis_datos)
{
        int a, b;
        /* ... */
        return 0;
}

struct datos *recibir_datos()
{
        struct datos *mis_datos;
```

```
        /* ... */
        return mis_datos;
}
```

cliente.c

```
#include "protocolo.h"

int main()
{
        struct datos mis_datos = {
                .codigo = 1,
                .tipo = 2,
                .mensaje = "Hola Compadre"
        };
        enviar_datos(&mis_datos);
        return (0);
}
```

servidor.c

```
#include "protocolo.h"
#include <stdio.h>

int main()
{
        struct datos *datos;

        datos = recibir_datos();
        if (datos->tipo == 2) {
                printf("codigo: %d\n", datos->codigo);
                printf("mensaje: %s\n", datos->mensaje);
        }
        return 0;
}
```

Como se compila

```
gcc -Wall -c protocolo.c
gcc -Wall -c cliente.c
gcc -Wall -c servidor.c
gcc -Wall -o cliente cliente.o protocolo.o
gcc -Wall -o servidor servidor.o protocolo.o
```

Usando `Makefile`

```
# Makefile para ejemplo imperativo

CC      = gcc
CFLAGS  = -g

all: cliente servidor

cliente: cliente.o protocolo.o
        $(CC) -o $@ $^

servidor: servidor.o protocolo.o
```

```
	$(CC) -o $@ $^

cliente.o: protocolo.h
servidor.o: protocolo.h
protocolo.o: protocolo.h

clean:
	rm -f cliente.o servidor.o protocolo.o \
	      cliente servidor

.PHONY: all clean
```

3. Nombres, Ligado y Ámbito

Conceptos

- Nombres e identificadores
- Variables
- Tipos
- Ámbito
- Constantes

Nombres

- *Identificador* que designa en el lenguaje un elemento u objeto de programa (una variable, un tipo, una constante, una función, etc.)
- Aspectos de diseño
  - Largo (permitido y significativo) del nombre
  - Caracteres aceptados (como conector _)
  - Sensibilidad a mayúsculas y minúsculas
  - Palabras reservadas

Variable

Abstracción de un objeto de memoria, que tiene los siguientes atributos:

- *Nombre* (identificador)
- *Dirección* (l-value: ¿Dónde está localizada?)
- *Valor* (r-value: Contenido)
- *Tipo* (Tipo del dato que almacena, incluye operaciones válidas)
- *Tiempo* de vida (¿Cuándo se crea y cuándo se destruye?)
- *Ámbito* (¿Dónde se puede referenciar?)

Dirección de una Variable

Un *nombre* puede estar **asociado** con diferentes *direcciones* en diferentes partes y tiempo de ejecución de un programa

- El *mismo nombre* puede ser usado para referirse a diferentes objetos (direcciones), según el ámbito.
- *Diferentes nombres* pueden referirse a un mismo objeto (alias), lo que puede provocar efectos laterales.

Example 5.
- **union** y punteros en C y C++
- Variables dinámicas en Perl

Ligado (Binding)

Definición

Proceso de asociación de un atributo a una entidad del lenguaje.

- *Variable*: Dirección, tipo $\longrightarrow$ Nombre
- *Operador*: Código $\longrightarrow$ Símbolo

Tiempo de Ligado

Instante en que sucede la asociación:

- *Estática*: Diseño o implementación del lenguaje; compilación, enlace o carga del programa
- *Dinámica*: Durante la ejecución del programa

(a) Ligado de Tipos a Variables

- Variables deben ser *ligadas a un tipo* antes de usarlas.
- **Ligado Estático**: Con declaración explícita o implícita
  - Lenguajes compilados modernos usan sólo declaración explícita
  - C y C++ hacen la diferencia entre declaración y definición
- **Ligado Dinámico**: En el momento de la asignación
  - *Ventaja*: Permite programar en forma genérica
  - *Desventaja*: Disminuye capacidad de detección de errores y aumento del costo (ejecución)

(b) Ligado a Memoria y Tiempo de Vida de Variables

- El carácter de un lenguaje está en gran medida determinado por cómo administra la memoria.
- A la variables se les puede asignar y liberar memoria de un espacio común (liberación implícita requiere de un recolector de basura).
- Las variables, según tipo de ligado, se clasifican en:
  1. Estáticas
  2. Dinámica de stack
  3. Dinámica de heap

(b.1) Variables Estáticas

- Ligadas antes de la ejecución
- *Ventajas*
  - Útil para variables globales
  - Para variables sensibles a la historia de un subprograma (p.ej. uso de **static** en variables de funciones C y C++)
  - Acceso directo permite mayor eficiencia
- *Desventajas*
  - Falta de flexibilidad
  - No soportan recursión
  - Impide compartir memoria entre diferentes variables

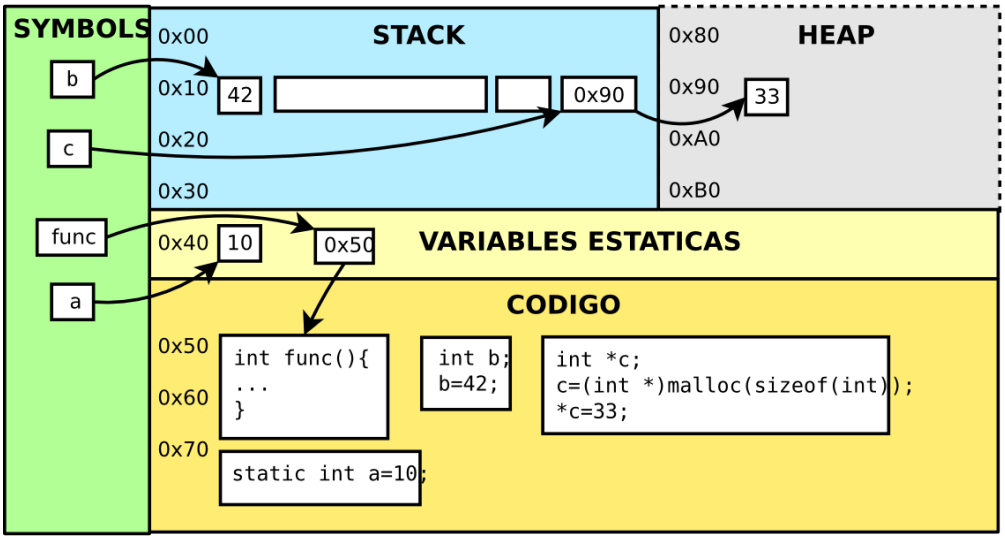
(b.2) Variables Dinámicas de Stack

- Ligadas (a la memoria) en el momento en que la ejecución alcanza el código asociado a la declaración.
- En Pascal, en C y en C++ se asigna memoria en el momento de ingresar al bloque que define el ámbito de la variable.
- En C y C++ las variables por omisión son de este tipo (denominadas *variables automáticas*).

(b.3) Variables Dinámicas de Heap

- La memoria se asigna y libera en forma explícita por el programador, usando un operador del lenguaje o una llamada
  - C++ dispone de operadores **new** y **delete**
  - C usa llamadas malloc() y free ()
  - Java usa el operador **new**, no permite la liberación explícita
- *Ventaja*: Útil para estructuras dinámicas usando punteros.
- *Desventaja*: Dificultad en su uso correcto

Ejemplo: Programa en Memoria



Tipo de Datos

- Para generalizar, supongamos:
  - La *asignación* es un operador binario con una variable y una expresión como operandos
  - Un *subprograma* es un operador cuyos parámetros son sus operandos
- *Verificación de tipo* es asegurar que los operandos de un operador son de tipo compatible.
- Un *tipo compatible* es uno legal o que mediante reglas el compilador puede convertir a legal.
- La conversión automática de tipo se denomina *coerción*

Verificación de Tipo

- Si todos los tipos se ligan *estáticamente*, la verificación de tipos se puede hacer al compilar.
- Ligado *dinámico* requiere realizar una verificación de tipo en tiempo de ejecución.
- La verificación de tipo se complica cuando se permite almacenar valores de diferente tipo en una variable (como registro con variante en Pascal y **union** en C).

Tipificado Fuerte (strong typing)

- Un lenguaje es de *tipificado fuerte* si permite detectar siempre los errores de tipo:
  - Cada nombre debe ser ligado estáticamente a un tipo
  - Ligado dinámico a memoria requiere verificación en tiempo de ejecución
- Ejemplos:
  - Perl ni siquiera se acerca...
  - C y C++ no lo son (parámetros variantes en funciones y **union**)
  - ¡Pascal casi lo es! (registro con variante es excepción)
  - Java sí lo es (permite forzar conversión de tipo)

Compatibilidad de Tipos

- Compatibilidad de nombre
  - Las variables están en la misma declaración o usan el mismo nombre de tipo
  - Fácil implementación, pero muy restrictivo
- Compatibilidad de estructura
  - Si los tipos tienen la misma estructura
  - Más flexible, pero de difícil implementación
  - Tiene efectos no intuitivos
- Algunos lenguajes permiten forma intermedia! (i.e. Equivalencia declarativa en Pascal)

Ejemplo de Compatibilidad

```
struct s1 {int c1; float c2;};
struct s2 {int c1; float c2;};
typedef char *pchar;

int main()
{
        struct s1 x;
        /*struct s2 y = x;
         * error de compatibilidad */
        pchar p1, p2;
        char *p3 = p1;
        return 0;
}
```

Ámbito (Scope)

- Rango de sentencias en el cual un *nombre* es visible.
- Nombres pueden ser sólo referenciadas *dentro* del ámbito.
- Nombres *no locales* son los que son visibles dentro de un bloque, pero han sido declarado fuera de él.

Ejemplo de Ámbito

```
#include <stdio.h>
static int a = 2;
int *c;

void func()
{
        static int b = 0;
        b++;
        int a;
        a = 3;
        printf("%d\n",b);
}

void func2()
{
        a = 1;
}

int main()
{
   func();
   func();
   func();
        printf("a = %d\n", a);
        func2();
        printf("a = %d\n", a);
        //int a = b;
        //printf("a = %d\n", a);
        return 0;
}
```

4. Tipos de Datos

4.1. Tipos de Datos Simples

Introducción

- ¿Cómo calzan los tipos de datos con problemas del mundo real?
- Evolución de los tipos:
  - Números enteros y reales
  - Arreglos y Registros
  - Cadenas de Caracteres
  - Definidos por el usuario
  - Tipo de dato abstracto

Tipos Ordinales

- Un *tipo ordinal* es aquel que puede ser asociado a un número natural (ordenados)
- Tipos ordinales primitivos:
  1. Entero
  2. Caracter
  3. Booleano
- Tipos ordinales definidos por el usuario:
  1. Enumerados
  2. Subrangos

Representación de Números

- Características de Representación
  - Conjunto finito
  - Rango de representación depende del largo del registro
- Tipos de Representación
  - Números enteros
  - Números de punto fijo
  - Números de punto flotante

Tipo Enumerado

- Se enumeran todos los posibles valores a través de constantes literales.
- *Relación de orden* permite definir operadores relacionales y predecesor y sucesor.
- Mejoran facilidad de lectura y fiabilidad
- Normalmente no se usan en E/S

Sintaxis en C y C++

```
<enum-type> ::= enum [<ident>] {<enum-list>}
<enum-list> ::= <enumerador>|<enum-list>,<enumerador>
<enumerador> ::= <ident>|<ident> = <cons-exp>
```

Ejemplo de enum

```
enum color { rojo, amarillo, azul = 7, verde };

int main()
{
        enum color col = rojo;
        enum color *cp = &col;

        *cp += 1;
        if (*cp == amarillo)
                col = 7;
        col += amarillo;
        printf("%d",col);
}
```

Tipo Subrango

- Subsecuencia contigua de un tipo ordinal
- Introducido por Pascal
- Mejora lectura y fiabilidad
- Ejemplo: Pascal

```
type
    mayuscula       = 'A'..'Z';
    indice          = LUNES .. VIERNES;
```

Tipos de Datos Primitivos

- Numérico
  - **Entero** (C ofrece varios tipos de enteros: **signed**, **unsigned**; **char**, **short**, **int**, **long**, **long long**)
  - **Punto flotante** (C ofrece **float**, **double** y **long double**)
  - **Decimal** (típicamente 4 bits por dígito decimal)
- **Booleano** (típicamente ocupa un byte)
- **Caracter** (típicamente un byte y código ASCII; Java usa Unicode con UTF-16)

4.2. Tipos de Datos Estructurados

Arreglos

- Es un tipo estructurado consistente en una secuencia de elementos que se identifican por su posición relativa mediante un índice.
- Hay un tipo asociado a los elementos y al índice.
- Índice se escribe entre:
  - Paréntesis redondo (FORTRAN, BASIC) o
  - cuadrado (Pascal, C, C++ y Java).
- Verificación de rango del índice mejora fiabilidad (Pascal y Java lo hacen).

Arreglos Multidimensionales

- **Pascal** escribe para dos dimensiones **TYPE** matriz=**ARRAY** [subidx, subidx] **OF real**;
- **C** y **C++** usan **double** matriz[DIM1][DIM2] (en rigor, C y C++ no tienen arreglos de más de una dimensión, tienen arreglos de arreglos eso sí)

Inicialización de Arreglos

- ANSI **C** y **C++** **char** $*$mensaje="Hola␣mundo\n"; **char** $*$dias[] = {"lu", "ma", "mi", "ju", "vi"};
- **Java** **int** [] edades = {7, 12, 13, 21};
- **Pascal** no lo permite

Operaciones con Arreglo

- Pascal y C y no tienen soporte especial (sólo selector con subíndice [])
- C++ permite definir un operador subíndice por el usuario, con lo que se pueden crear arreglos; y se pueden definir operadores adicionales como asignación, inicialización, etc.
- Java define los arreglos como objetos especiales. Permite uso de subíndice, cálculo del largo y otros métodos.

Implementación de Arreglos

- La memoria es un arreglo unidimensional de celdas
- Un arreglo es una abstracción del lenguaje
- Un arreglo debe ser mapeado a la memoria
- Arreglos bidimensionales se almacenan como fila de columnas, o viceversa (*por columna* o *por fila*)
- La *dirección* de un elemento de un arreglo bidimensional se puede calcular. En almacenamiento por filas sería $dir(arr[x][y]) ::= dir(arr[0][0]) + (y \cdot MAX_X + x) \cdot sizeof(tipo)$

Arreglos Asociativos

- Colección no ordenada de elementos que son accedidos por una clave
- Estructura de alto nivel que manejan lenguajes de scripting, como AWK o Perl.

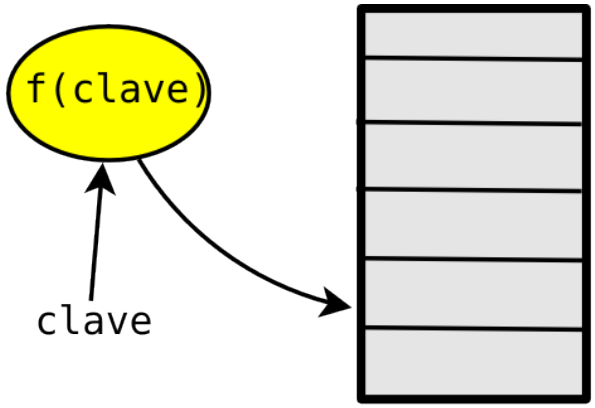
Example 6. En Perl se vería: `%ventas = ('lun'=>10, 'mar'=>21,'mie'=>75); print $ventas{'mie'}; $semana = $ventas{'lun'} +`

Arreglos Asociativos

- C++ provee mapas y multimapas a través de STL (Standard Template Library)
  - Mapas: Una clave, un valor.
  - Multimapas: Una clave, múltiples valores.
- Java provee interfaces para diccionarios e implementaciones de tablas hash.
- Perl provee una estructura con el nombre de hash

Hashing

- Ubicación de un elemento dentro del arreglo por una función matemática sobre la clave.
- Los arreglos son casos especiales, donde la clave es el índice y la función matemática es la identidad.
- Permite que las claves sean estructuras más complejas.



Tipo Registro

- Permite composición heterogénea de elementos de datos
- Cada elemento se identifica por un nombre (campo o miembro)
- Introducido por COBOL
- C y C++ proveen **struct**, Pascal tiene **record**
- Concepto de clase en programación orientada a objetos extiende esta idea (C++ y Java)

Ejemplo: Registros en C

```c
#include <string.h>
struct empleado {
        struct {
                char pila[10], pat[10], mat[10];
        } nombre;
        int sueldo;
};

int main()
{
        struct empleado del_mes, otro;

        del_mes.sueldo = 500000;  otro.sueldo = 300000;
        strcpy(del_mes.nombre.pila, "Juan");
        return 0;
}
```

Ejemplo: Registros en Pascal

```
type
   empleado = record
      nombre: record
         primer:  packed array [1..10] of char;
         paterno: packed array [1..10] of char;
         materno: packed array [1..10] of char;
      end {nombre};
      sueldo: integer
   end;

var
   del_mes, otro: empleado;

begin
   del_mes.sueldo := 50000;
```

Cadena de Caracteres (String)

- Principalmente para la comunicación máquina-usuario y para manipulación de textos
- Mejora la facilidad de escritura
- ¿Es una cadena un tipo primitivo?
- Algunos lenguajes lo proveen como tipo básico (Java, BASIC, y Perl)
- Otros sólo como arreglo de caracteres (C, C++ y Pascal)
- ¿Puede el largo variar dinámicamente?

Strings: Operaciones Básicas

- Asignación
- Comparación

- Concatenación
- Substring
- Largo
- Transformación (por ejemplo de string a entero)

Ejemplo: Strings en C

```c
#include <stdio.h>
#include <string.h>

int main()
{
        char palabra[30];

        strcpy(palabra, "contertulio");
        /* ... */
        if (strcmp(palabra, "tulio") != 0)
                printf("%s\n", palabra);
        return 0;
}
```

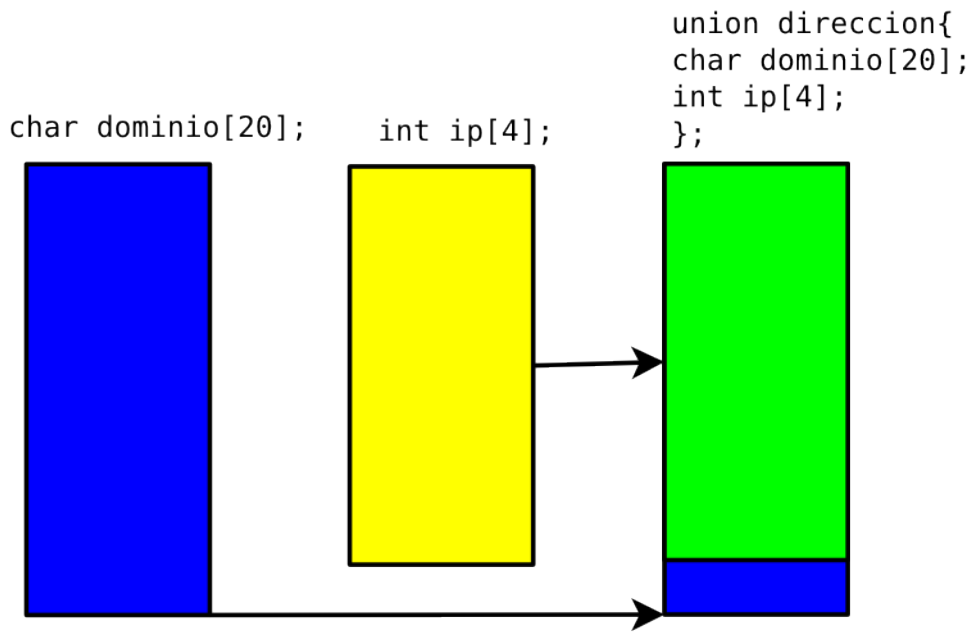
Diseño de String

- Diseño de string considera:
  - Largo estático (Pascal)
  - Largo dinámico limitado (C, C++, Java)
  - Largo dinámico (BASIC, Perl)
- Último es el más flexible, pero es más costoso de implementar y ejecutar.

Tipos Variantes

- Permite almacenar diferentes tipos de datos en diferentes tiempos en una misma variable.
- Reserva espacio de memoria igual al mayor miembro definido.
- Todos los miembros comparten la memoria y comienzan desde la misma dirección.
- Su uso es en general poco seguro.
- Pascal provee **record** variantes que contienen un campo (*discriminador*) que indica cuál variante está activa

Ejemplo Variante en C



4.3. Tipo Puntero

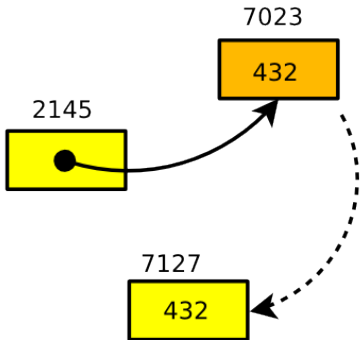
Tipo Puntero

- Su valor corresponde a una dirección de memoria, habiendo un valor especial nulo (llamado NULL en C, nil en Pascal, etc) que no apunta a nada.
- Aplicaciones:
  - Método de gestión dinámica de memoria (acceso a variables dinámicas de heap)
  - Método de direccionamiemto indirecto
- No corresponde a un tipo estructurado, aún cuando se definen en base a un operador de tipo

Operaciones con Punteros

- *Asignación*: Asigna a la variable como valor una dirección a algún objeto de memoria.
- Desreferenciación: Entrega el valor del objeto apuntado.
  - Operador $*$ en C y C++ (vale decir, $*$ptr)
  - Operador ^ en Pascal (por ejemplo, ptr^)
  - C y C++ permiten aritmética con punteros que apunten dentro del mismo arreglo: La diferencia da los elementos entre los punteros, un puntero más un entero apunta tantos elementos más allá, etc

Ejemplo Punteros en C




```c
#include <stdlib.h>
int main()
{
        int j, *ptr;

        ptr = malloc(sizeof(int));
        *ptr = 432;
        j = *ptr;
        return 0;
}
```

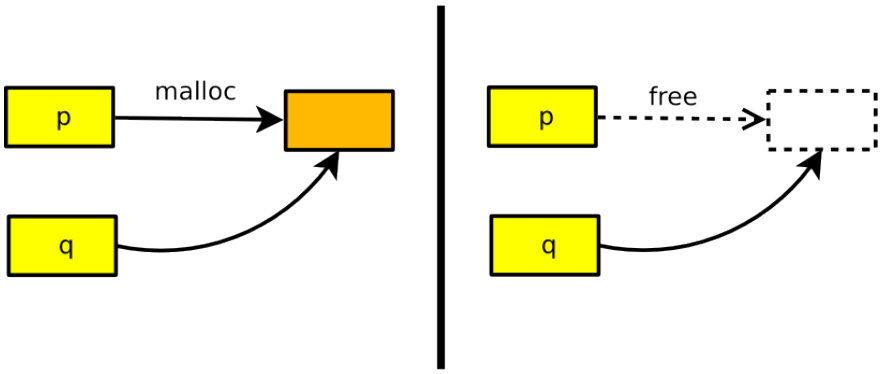
Problemas con Punteros

(1) Dejar Colgado (Dangling)

```c
#include <stdlib.h>

int main()
{
        int *p, *q;

        p = malloc(sizeof(int));
        /* ... */
        q = p;
        /* ... */
        free(p);
        /* ... *//* La variable puede reasignarse! */
        *q = 12;
        return 0;
}
```

Problemas con Punteros

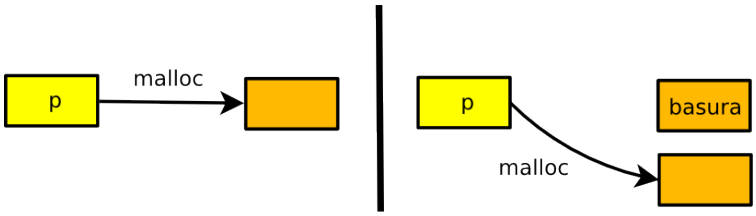
(1) Dejar Colgado (Dangling)



Problemas con Punteros

(2) Pérdida de variables dinámicas de heap

```
int main()
{
        int *p;

        p = malloc(sizeof(int));
        /* ... */
        p = malloc(sizeof(int));
        /* ... */
        return 0;
}
```



Punteros y Estructuras en C

```
#include <stdlib.h>

struct nodo {
        int info; struct nodo *siguiente;
};

int main()
{
        struct nodo *pn;

        pn = malloc(sizeof(struct nodo));
        (*pn).info = 30;
        pn->siguiente = NULL;
        free(pn);
        return 0;
}
```

Punteros y Estructuras en C++

```
struct nodo {
        int info; struct nodo *siguiente;
};

int main()
{
        struct nodo *pn;

        pn = new nodo;
        (*pn).info = 30;
        pn->siguiente = NULL;
        delete pn;
        return 0;
}
```

Punteros y Arreglos en C y C++

En C, un arreglo puede considerarse como un puntero constante a su primer elemento

```
#include <stdio.h>

int main()
{
        int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        int *pa, i;

        pa = &a[0];
        pa = a;                 /* hace lo mismo que la linea anterior */

        for (i = 0; i < 10; i++)
                printf("%d", a[i]);
        for (i = 0; i < 10; i++)
                printf("%d", *(pa + i));
        for (i = 0; i < 10; i++)
                printf("%d", *(a + i));
        /* los tres for hacen lo mismo */
        return 0;
}
```

Aritmética de Punteros en C y C++

```
#define ALLOCSIZE 1000          /* tamanˆo del buffer */

static char buf[ALLOCSIZE];     /* buffer */
static char *ap = buf;          /* posicion libre */

char *alloc(int n)              /* puntero a 'n' char */
{
        if (buf + ALLOCSIZE - ap >= n) { /* cabe? */
                ap += n;
                return ap - n; /* antigua direccion */
        } else
                return 0;
}

void afree(char *p)             /* libera desde p */
{
        if (buf <= p && p < buf + ALLOCSIZE)
                ap = p;
}
```

Tipo Referencia en C++

- Un tipo referencia en C++ es un puntero constante que es implícitamente desreferenciada.
- Dado que es constante, debe ser inicializada en su definición.

```
int main()
{
        int valor = 3;
        int &ref_valor = valor;
```

```
        ref_valor = 100;
        /* ahora valor==100*/
        return 0;
    }
```

- La referencia actúa como alias de una variable.

Referencias en Java

- Java extiende la forma de *variables de referencia* de C++, eliminando el uso explícito de punteros.
  - Java permite asignar un nuevo objeto a una variable de referencia
  - Todo objeto sólo se referencia con estas variables
- En Java liberación de objetos es implícita
- Requiere de un recolector de basura

Métodos de Reclamo de Basura

Contadores de Referencia (impaciente)

- Se mantiene un contador de referencia por cada celda
- Se incrementa con cada nueva referencia y se decrementa cuando se pierde una referencia
- La celda se libera tan pronto la cuenta llega a cero (cuando se convierte en *basura*)

Métodos de Reclamo de Basura

Recolección de Basura (perezoso)

- Se *acumula basura* hasta que se agota la memoria
- Si se agota la memoria se identifican las celdas de basura y se pasan a la lista de celdas libres
- Técnicas modernas van recuperando basura incrementalmente, por ejemplo un poco cada vez que se solicita memoria

Recolección de Basura

```
void* allocate (int n){
  if (!hay_espacio) { /* recolectar basura */
    1) marcar todo los objetos del heap como basura;
    2) for (todo puntero p) {
       if (p alcanza objeto o en el heap)
          marcar o como NO basura;
       } /* for */
    3) liberar todos los objetos marcados como basura
  }
  if (hay_espacio) {
    asignar espacio;
    return puntero al objeto;
  } else return NULL;
}
```

Evaluación de los Métodos

Contadores de Referecias

- Requiere bastante memoria para mantener contadores
- Asignaciones a punteros requiere de más tiempo de ejecución para mantener contadores

Recolección de Basura

- Basta un bit por celda para marcar basura
- Mal desempeño cuando queda poca memoria

5. Expresiones y Asignaciones

Introducción

- Lenguajes imperativos se caracterizan por el uso dominante de expresiones y asignaciones
- El valor de las expresiones depende del orden de evaluación de operadores y operandos
- Ambigüedades en el orden de la evaluación puede conducir a diferentes resultados

Expresiones Aritméticas

- Orden de evaluación está principalmente definida por las reglas de precedencia y asociatividad
- Paréntesis fuerzan determinado orden

Tipos de operadores

Los operadores se clasifican según su *aridad* y su *posición*.

- Operadores *unarios* toman un único operando. Ejemplos son ! y ++ en C.
- Operadores *binarios* toman dos operandos. Ejemplos son / y && en C.
- El único ejemplo común de operadores de mayor aridad es el operador ternario ?: de C y lenguajes afines.
- Operadores *prefijo* se escriben antes de sus operandos, como $-$i, $*$p, !b en C.
- Operadores *infijo* se escriben entre de sus operandos, como a + b, a || b, a $*=$ b en C.
- Operadores *postfijo* se escriben después de sus operandos, como a++ en C.
- Las llamadas a funciones son operadores n-arios, prefijo.

Precedencia y asociatividad

- *Precedencia* define en primera instancia en qué orden se ejecutan las operaciones. En el álgebra común, las multiplicaciones se efectúan antes de las sumas, se dice que la multiplicación tiene *mayor precedencia*; y las sumas y restas se efectúan juntas, con lo que tienen la *misma precedencia*.
- *Asociatividad* determina el orden en que se efectúan operaciones de la misma precedencia. Si las operaciones se ejecutan de derecha a izquierda se dice que son *asociativas derechas*, si se ejecutan de izquierda a derecha se dicen *asociativas izquierdas*. En caso que la combinación no se permita se dice *no asociativa*.

Los lenguajes de programación generalmente siguen las reglas del álgebra, pero los hay que tienen reglas diferentes.

Expresión Condicional

```
/* Version N#1 */
int abs(int n)
{
        if (n >= 0)
                return n;
        else
                return -n;
}

/* Version N#2 */
int abs_pro(int n)
{
        return (n >= 0) ? n : -n;
}
```

Efectos Laterales de Funciones

Orden de evaluación de los operandos puede producir resultados diferentes si existen efectos laterales

```
int a = 2;
int f1()
{
        return a++;
}

int f2(int i)
{
        return --a * i;
}

int main()
{
        printf("%\n", f2(3));
        printf("%\n", f1());
        printf("%\n", f1() * f2(3));
        return 0;
}
```

¿Cómo evitar efectos laterales de funciones?

- Deshabilitar efectos laterales en la evaluación de funciones
  - Quita flexibilidad, por ejemplo habría que negar acceso a variables globales
- Imponer un orden de evaluación a las funciones
  - Evita que el compilador puede realizar optimizaciones
  - Enfoque seguido en Java (evaluación de izquierda a derecha).

Expresiones Relacionales

Operación	Pascal	C	Ada	FORTRAN
Igual	=	==	=	.EQ.
No es igual	<>	!=	/=	.NE.
Mayor que	>	>	>	.GT.
Menor que	<	<	<	.LT.
Mayor o igual que	>=	>=	>=	.GE.
Menor o igual que	<=	<=	<=	.LE.

Operadores Booleanos

- Incluye: **AND**, **OR**, **NOT** y, a veces, **XOR**.
- La precedencia está generalmente definida de mayor a menor: **NOT**, **AND** y **OR**.

Operadores relacionales y lógicos

C:
$b + 1 > b * 2$ /* aritmeticos primero */

C:
$a > 0 \;||\; a < 5$ /* relacional primero */

Pascal:
$a > 0$ or $a < 5$ { es ilegal }

Corto Circuito o Término Anticipado de Evaluación

```
#include <stdio.h>
int cuenta = 0;
int f1(int a)
{
        cuenta++;
        return a %2;
}
int f2(int b)
{
        cuenta++;
        return b > 10 || b < 20;
}

int main()
{
        int a = 0, b = 21;
        if (f1(++a) && f1(++a) && f1(++a) && f1(++a)) {
                printf("Paso el if 1\n");
        }
        if ((f1(b) || f2(b)) && f1(++a)) {
                printf("Paso el if 2\n");
        }
        printf("%d\n", cuenta);
        return 0;
}
```

Corto Circuitos en los Lenguajes Imperativos

- C, C++ y Java definen corto circuito para: && y || (**AND** y **OR**)
- Pascal especifica que *no* debe usarse evaluación en corto circuito (algunas implementaciones igual lo ofrecen como opción)
- Ada tiene dos versiones de los operadores, una sin cortocircuito (**and**, **or**) la otra con (**and then**, **or else**)

Sentencias de Asignación

- Permite cambiar dinámicamente el valor ligado a una variable
- Basic, C, C++ y Java usan =
- Algol, Pascal y Ada usan :=
- C, C++ y Java permiten incrustar una asignación en una expresión (actúa como cualquier operador binario)

Operadores Compuestos y Unarios de Asignación

Sólo presentes en C, C++ y Java...

Example 7.
```
int main()
{
        int sum, A[20], i;
        sum += A[i];
        /* sum = sum + A[i]; */

        sum = ++i;
        /* i = i + 1;
           sum = i; */
        return 0;
}
```

Asignación en Expresiones

- C, C++ y Java permiten incrustar asignaciones en cualquier expresión
- Permite codificar en forma más compacta
- La desventaja de esta facilidad es que puede provocar efectos laterales, siendo fuente de error en la programación
- Es fácil equivocarse confundiendo == y =

Asignación en Expresiones

```
int main()
{
        int x, y;

        x = 0;
        y = 1;
        if (x = y)
                x++;
        return 0;
}
```

C entrega x=2 e y=1. Java no tolera este hecho.

6. Estructuras de Control

Introducción

- Ejecución es típicamente secuencial
- Normalmente se requieren dos tipos de sentencias de control:
  - Selección de alternativas de ejecución
  - Ejecución repetitiva de un grupo de sentencias
- En principio basta tener un simple **goto** selectivo, pero es poco estructurado

(a) Sentencias Compuestas

- Permite agrupar un conjunto de sentencias
- **begin** y **end** en Pascal
- Paréntesis de llave en C, C++ y Java

(b) Sentencias de Selección

- Selección binaria Ejemplo: **if else** o **if** (solo) en C
- Selección múltiple Ejemplos:
  - **case** en Pascal
  - **switch** en C, C++ y Java
  - **if** elseif **else** en Ada

(c) Sentencias Iterativas

- Bucles controlados por contador Ejemplo: **for** en Pascal
- Bucles controlados por condición Ejemplos: **while**, **do while** en C, C++ y Java **repeat until** en Pascal
- Bucles barrocos, como en C: Controlados por condición, pero la actualización de las variables es arbitraria
- Bucles controlados por estructuras de datos Ejemplo: **foreach** en Perl Java 1.5 acaba de introducir una especie de foreach.

(d) Salto Incondicional

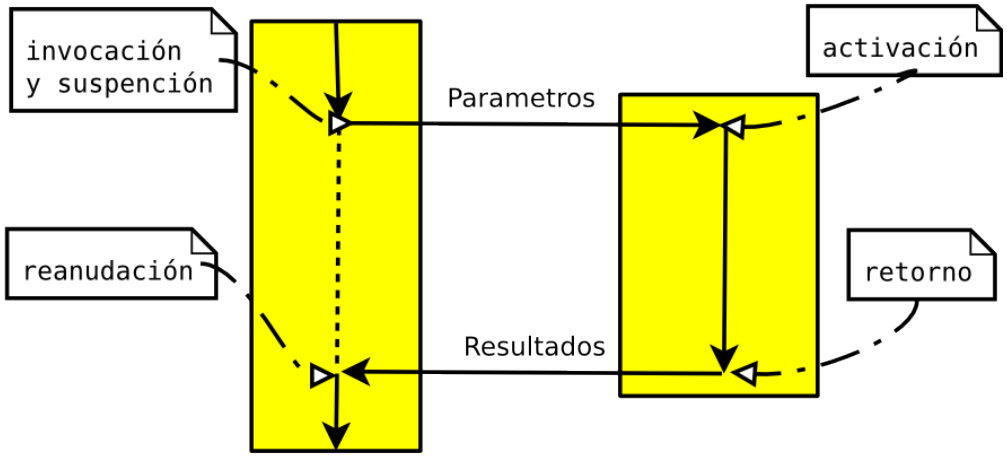
- Uso de rótulos o etiquetas Ejemplo: **goto** FINAL
- Restricciones:
  - Sólo rótulos en el ámbito de la instrucción
  - El salto es a un rótulo fijo
- Saltos incondicionales con blanco implícito (**break** y **continue** en C)

7. Subprogramas

Características de un Subprograma

- Permite crear *abstracción de proceso*:
  - encapsulando código
  - definiendo una interfaz de invocación para paso de parámetros y resultados
- Permite reutilizar código, ahorrando memoria y tiempo de codificación.
- Existe en forma de procedimiento y función.

Mecanismo de Invocación



Firmas y Protocolos de la Interfaz

- La firma (signature), o prototipo, es un contrato entre el invocador y el subprograma que define la semántica de la interfaz.
- El protocolo especifica como debe realizarse la comunicación de parametros y resultados (tipo y orden de los parámetros y, opcionalmente, valor de retorno).

```
procedure random(in real semilla; out real aleat);
```

Parámetros

- *Parámetros formales* son variables mudas que se ligan a los parámetros reales cuando se activa el subprograma.
- Normalmente ligado se hace según *posición* en la lista.
- Parámetros permiten *comunicación explícita* de datos y, a veces, también (otros) subprogramas.
- *Comunicación implícita* se da a través de variables no locales, lo que puede provocar efectos laterales.

Semántica de Paso de Parámetros

- Modo de interacción de parámetro actual a formal puede ser:
  - Entrega de valor (IN)
  - Recibo de valor (OUT)
  - Ambos (INOUT)
- La implementación de la transferencia de datos puede ser:
  - Copiando valores
  - Pasando referencias (o punteros)

(a) Paso por Valor (pass-by-value)

- Modo *IN* e implementado normalmente con copia de valor
- Implementación con paso de referencia requiere protección de escritura, que puede ser difícil
- Permite proteger de modificaciones al parámetro actual, pero es más costoso (más memoria y tiempo de copiado)
- Permite usar expresiones como parámetro actual

(b) Paso por Resultado (pass-by-result)

- Modo *OUT* y normalmente implementado con copia (mismas complicaciones que por valor)
- Parámetro formal actúa como variable local, pero al retornar copia valor a parámetro actual
- Parámetro actual debe ser variable
- Dificultades:
  - Existencia de colisiones en los parámetros actuales, puede conducir a ambigüedad
  - Cuándo se evalúa dirección de parámetro actual?

(c) Paso por Valor-Resultado (pass-by-value-result)

- Modo *INOUT* con copia de parámetros en la entrega y en el retorno
- Por esto, llamada a veces paso por copia
- Mismas dificultades que paso por valor y paso por resultado

(d) Paso por Referencia(pass-by-reference)

- Modo *INOUT* e implementación con referencias
- Parámetro formal y real comparten misma variable
- *Ventajas*: Comunicación es eficiente en:
  - espacio (no requiere duplicar variable)
  - tiempo (no requiere copiar)
- *Desventajas*:
  - Acceso es más lento (indirección)
  - Es fuente de error (modificación de parámetro real)
  - Creación de alias a través de parámetros actuales

Paso de Parámetros

- *C*: Paso por valor, y por referencia usando punteros
  - Puntero puede ser calificado con **const**; se logra semántica de paso por valor (sin permitir asignación)
  - Arreglos se pasan por referencia (decaen en punteros)
- *C++*: Igual que C, más paso por referencia indicado por operador &. Este operador también puede ser calificado con **const**, permitiendo semántica de paso por valor con mayor eficiencia (i.e. paso de grandes arreglos)
- *Java*: Todos los parámetros son pasados por valor, excepto objetos que se pasan por referencia. No existencia de punteros no permite paso por referencia de escalares (si como parte de un objeto)
- *Pascal*: Por omisión paso por valor, y por referencia si se usa calificativo **var**.

Funciones como Parámetro

```c
#include <math.h>
#include <stdio.h>

#define PRE 2000

double integra(double fun(double), double bajo, double alto)
{
        int i;
        double res = 0.0, leng = (alto - bajo) / PRE;

        for (i = 0; i < PRE; i++)
                res = res + (fun(bajo + (double)i * leng) * leng);
        return res;
}

int main()
{
        printf("Res: %f\n", integra(cos, -M_PI / 2, M_PI / 2));
        return 0;
}
```

Sobrecarga de Subprogramas

- En el mismo ámbito existen diferentes subprogramas con el mismo nombre.
- Cada versión debiera tener una firma diferente, de manera que a partir de los parámetros reales se pueda resolver a cual versión se refiere.
- Las versiones pueden diferir en la codificación
- Es una conveniencia notacional, que es evidente cuando se usan nombres convencionales, como en siguiente ejemplo

Sobrecarga de Funciones en C++

```c
float my_abs(float num)
{
        return (num > 0) ? num : -num;
}

int my_abs(int num)
{
        return (num > 0) ? num : -num;
}

int my_abs(int num, int plus)
{
     return my_abs(num) + plus;
}

int main(){
        int a = -1, c = -2;
        float b = 1.5;

        a=my_abs(a);
        b=my_abs(b);
        c=my_abs(c, 2);
        return 0;
}
```

Subprogramas Genéricos

- Permite crear *diferentes subprogramas* que implementan el mismo algoritmo, el cual actúa sobre *diferentes* tipos de datos.
- Mejora la reutilización, aumentando productividad en el proceso de desarrollo de software.
- *Polimorfismo paramétrico*: Parámetros genéricos de tipos usados para especificar los tipos de los parámetros de un subprograma
- Sobrecarga de subprogramas corresponde a un *polimorfismo ad-hoc*

Funciones Genéricas en C++

```
template <class Tipo>
Tipo maximo(Tipo a, Tipo b)
{
   return a > b ? a : b;
}

int x, y, z;
char u, v, w;

z = maximo(x, y);
w = maximo(u, v);
```

Sobrecarga de Operadores definida por el Usuario

```
#include <iostream>
using namespace std;

struct complejo { /* a + bi */
        double a, b;
};

complejo operator +(const complejo& x, const complejo& y)
{
        complejo z;

        z.a = x.a + y.a; z.b = x.b + y.b;
        return z;
}

int main()
{
        complejo x, y, z;

        x.a = 10; x.b = -3;
        y.a = -5; y.b =  6;
        z = x + y;
        cout << z.a << " + " << z.b << 'i' << endl;
        return 0;
}
```