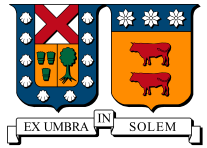
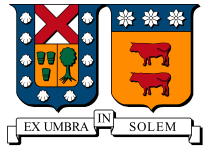


Expresiones de Control

Roberto Díaz
roberto.diazu@usm.cl

Universidad Técnica Federico Santa María
Departamento de Informática – Santiago, Chile

2021-2



Expresiones y Asignación

Expresiones Aritméticas

Consisten en operadores, operandos (argumentos), paréntesis y llamadas a funciones, donde el orden de evaluación determina su valor (definida por reglas de precedencia y asociatividad, más el uso de paréntesis).

Aridad de los operadores:

- **Unario (unitario):** un solo operando
- **Binario:** dos operandos
- **Ternario:** tres operandos

Orden de Evaluación de Operadores

Precedencia: determina el orden de evaluación de operadores adyacentes con diferente nivel de precedencia. Se prioriza el orden de evaluación de mayor a menor nivel de precedencia. Niveles típicos de precedencia:

- Operadores unarios (más alto)
- Exponenciación (si es soportada)
- $*$, $/$ y mod (binario)
- $+$ y $-$ (binarios)

Asociatividad: define el orden de evaluación de operadores adyacentes con el mismo nivel de precedencia.

Paréntesis: altera y fuerza orden de evaluación.

Reglas de Precedencia
Ejemplos

Fortran	Pascal	C	Precedencia
$**$	$*$, /, div, mod	Postfix ++, −−	Alta
$*$, /	+, −	Prefix ++, −−	
+, −		+, − (unario)	
		$*$, /, %	
		+, − (binario)	Baja

Reglas de Asociatividad
Ejemplos

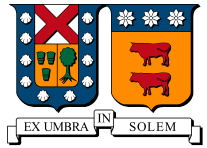
FORTRAN	**Izq:** $*$, $/$, $+$, $-$ **Der:** $**$
Pascal	Todos por la izquierda
C++	**Izq:** ++ y −− postfijo; $*$, $/$, %, + y − binario **Der:** ++ y −− prefijo; + y − unarios
C	**Izq:** $*$, $/$, %, + y − binario **Der:** ++, −−, − y + unario

Orden de Evaluación de Operandos

Evaluación de Operandos: el orden determina el valor y precisión.

- **Variables:** son leídas desde memoria.
- **Constantes:** A veces leídas de la memoria y otras veces incrustada en las instrucciones de máquina.
- **Paréntesis:** Evalúa todos los operandos y operadores contenidos primero, y luego su valor puede ser usado como operando.

Orden de Evaluación de Operandos



Efectos Laterales

- Si la evaluación de un operando altera el valor de otro en una expresión, entonces existe un efecto lateral.
- Si ninguno de los operandos de un operador tiene efectos laterales, el orden de evaluación de éstos es irrelevante; caso contrario si importa.
- Casos de efecto lateral:
  - Funciones con parámetros bidireccionales (e.g. INOUT).
  - Referencias a variables no locales.

Efectos Laterales
Ejemplo

```
int a = 2;

int f1() {
  return a++;
}

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

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

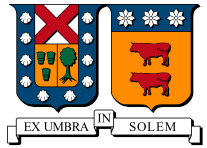
Efectos Laterales
¿Cómo evitarlos?

1. Definir un lenguaje que deshabilite efectos laterales en la evaluación de funciones.
   - Quita flexibilidad al lenguaje. Por ejemplo: habría que negar acceso a variables no locales y no permitir parámetros bidireccionales

Efectos Laterales
¿Cómo evitarlos?

1. Definir un lenguaje que deshabilite efectos laterales en la evaluación de funciones.
   - Quita flexibilidad al lenguaje. Por ejemplo: habría que negar acceso a variables no locales y no permitir parámetros bidireccionales
2. Imponer un orden de evaluación a las funciones
   - Evita que el compilador pueda realizar optimizaciones
   - Enfoque seguido en Java (evaluación de izquierda a derecha).

Transparencia Referencial



Propiedad de un programa, donde si cualquier par de expresiones en el programa producen el mismo valor, entonces pueden ser intercambiadas sin afectar la acción de programa. Se relaciona con efectos laterales de funciones.

- Tiene la ventaja que programas son más fáciles de entender.
- Lenguajes funcionales tienen en general esta propiedad.

Transparencia Referencial

Propiedad de un programa, donde si cualquier par de expresiones en el programa producen el mismo valor, entonces pueden ser intercambiadas sin afectar la acción de programa. Se relaciona con efectos laterales de funciones.

- Tiene la ventaja que programas son más fáciles de entender.
- Lenguajes funcionales tienen en general esta propiedad.

Ejemplo:

```
res1 = (fun(a) + b) / (fun(a) - c);

temp = fun(a);
res2 = (temp + b) / (temp - c);
```

- Si fun() no tiene efectos laterales, entonces: res1 = res2
- ¡En otro caso, la transparencia referencial es violada!

Tipo de Conversión de Escalares

Por Expansión: convierte un objeto a un tipo que incluye (al menos aproximadamente) todos los valores del tipo original.

- Ejemplo: Paso de entero a punto flotante, o subrango de enteros a entero.

Tipo de Conversión de Escalares

Por Expansión: convierte un objeto a un tipo que incluye (al menos aproximadamente) todos los valores del tipo original.

- Ejemplo: Paso de entero a punto flotante, o subrango de enteros a entero.

Por estrechamiento: convierte un objeto a un tipo que no puede incluir todos los valores del tipo original.

- Ejemplos: Paso de real a entero, o paso del tipo base a subrango.

Tipo de Conversión de Escalares

Por Expansión: convierte un objeto a un tipo que incluye (al menos aproximadamente) todos los valores del tipo original.

- Ejemplo: Paso de entero a punto flotante, o subrango de enteros a entero.

Por estrechamiento: convierte un objeto a un tipo que no puede incluir todos los valores del tipo original.

- Ejemplos: Paso de real a entero, o paso del tipo base a subrango.

Observación: expansión es más segura, pero puede tener algunos problemas (e.g. pérdida de precisión en la mantisa en el paso de entero a real).

Conversión en Expresiones aritméticas

Una expresión puede tener una mezcla de operandos de diferentes tipos. Entonces el lenguaje hace una conversión implícita (**coerción**) para usar el operador adecuado.

- Mayoría de los lenguajes usan conversión por expansión.
  - Java convierte todos los enteros más cortos (byte, short y char) a int para evaluar una expresión.
- Da mayor flexibilidad al uso de operadores, pero reduce la capacidad de detectar algunos errores e introduce código adicional.
- Programador puede controlar explícitamente la conversión mediante mecanismo de "*casting*".
  - Ejemplo en C: `(int)angulo`

Sobrecarga de Operadores

Un mismo operador (símbolo) es usado para diferentes propósitos, siendo que tienen distinto comportamiento. Ayuda a una mejor legibilidad.

Ejemplo:

```
int a, b, c;
float u, v, w;

c = a + b;
w = u + v;
u = a + w;
```

Sobrecarga de Operadores

Un mismo operador (símbolo) es usado para diferentes propósitos, siendo que tienen distinto comportamiento. Ayuda a una mejor legibilidad.
Ejemplo:

```
int a, b, c;
float u, v, w;

c = a + b;
w = u + v;
u = a + w;
```

Observaciónes:

- En algunos casos de sobrecarga de operadores la lógica es diametralmente diferente (e.g. & como AND y Dirección)
- Algunos lenguajes permiten dar nuevos significados a los símbolos de operadores (e.g. C++ y C#).

Errores en Expresiones

El lenguaje realiza verificación de tipos (estática o dinámica) para evitar ocurrencia de errores. Sin embargo, en algunos casos igual pueden ocurrir por las siguientes causas:

- Coerción de operandos en las expresiones.
- Rango limitado de representación de números.
- División por cero.

Errores comunes:

- Desbordamiento (overflow), subdesbordamiento (underflow) y división por cero.
  - Algunos lenguajes producen una excepción en estos casos.

Expresiones Relacionales y Lógicas

Expresión Relacional: usa un operador relacional (binario) para comparar los valores de dos operandos. El valor de la expresión es booleano (salvo que el lenguaje no tenga este tipo).

- Normalmente estos operadores están sobrecargados para diferentes tipos.
- Incluye: ==, !=, <, <=, >, >= (con diferentes sintaxis)

Expresiones Relacionales y Lógicas

Expresión Relacional: usa un operador relacional (binario) para comparar los valores de dos operandos. El valor de la expresión es booleano (salvo que el lenguaje no tenga este tipo).

- Normalmente estos operadores están sobrecargados para diferentes tipos.
- Incluye: ==, !=, <, <=, >, >= (con diferentes sintaxis)

Expresión Booleana: consiste de variables, constantes y operadores booleanos, donde se puede combinar con expresiones relacionales.

- Incluye: `AND`, `OR`, `NOT` y, a veces, `XOR`.

Operadores 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.

Precedencia de Operadores
Relacionales y Booleanos

La precedencia de los operadores booleanos está generalmente definida de mayor a menor: `NOT`, `AND` y `OR` (excepto ADA, que todos tienen igual precedencia, sin considerar `NOT`).

En general, los operadores aritméticos tienen mayor precedencia que relacionales, y los relacionales mayor que booleanos (excepto Pascal).

En C, C++ y Java, la asignación es el operador que tiene la menor precedencia.

Corto Circuito

Se denomina **corto circuito** al termino anticipado de una operación cuando su resultado ya está definido. Lenguajes de programación, en general, poseen un sistema de **corto circuito** para operadores lógicos.

- C, C++ y Java definen corto-circuito para operadores lógicos: `&&` y `||` (`AND` y `OR`).
- Perl, Python y Ruby evalúan todos sus operadores lógicos con corto-circuito.
- Modula-2 también lo define para: `AND` y `OR`
- Pascal no lo especifica (algunas implementaciones los tienen, otras no); algo parecido sucede con Fortran.
- ADA permite especificar explícitamente con los operadores: `and then` y `or else`

Precedencia
Ejemplos

```
C: b + 1 > b * 2

C: a > 0 || a < 5

Pascal: a > 0 OR a < 5 {es ilegal}
```

Precedencia y Corto Circuito
Ejemplos

```
C: a >=0 || b < 10

C: while ( (c = getchar()) != EOF && c != '\n')
   {
        //procesar caracter;
   }

C: q && r || s--
```

Sentencia de Asignación

Mecanismo que permite cambiar dinámicamente el valor ligado a una variable.

Ejemplos:

- Fortran, Basic, PL/I, 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).

Sentencia de Asignación
Asignación Condicional

C, C++ y Java (y otros derivados) permiten:

```
(a > 0) ? cuenta1 = 0: cuenta2 = 0;
```

que equivale a:

```
if (a > 0){
    cuenta1 = 0;
}else{
    cuenta2 = 0;
}
```

Sentencia de Asignación
Operadores Compuestos y Unarios

En C, C++ y Java:

equivale

```
sum += A[i];
```
⟶
```
sum = sum + A[i];
```

```
sum = ++contador;
```
⟶
```
contador = contador + 1;
sum = contador;
```

```
sum = contador++;
```
⟶
```
sum = contador;
contador = contador + 1;
```

Sentencia de Asignación
Asignación Múltiple

```
PL/I   : SUM, TOTAL = 0

C      : SUM = TOTAL = 0;

Python: x, y, z = 20, 30, 40

Perl   : ($first, $second, $third) = (20, 30, 40);
```

Sentencia de Asignación
Asignación en Expresiones

Muchos lenguajes consideran asignación como un operador, cuya evaluación define un tipo y un valor del dato, lo que permite incrustar la asignación en cualquier expresión.

Sentencia de Asignación
Asignación en Expresiones

Muchos lenguajes consideran asignación como un operador, cuya evaluación define un tipo y un valor del dato, lo que permite incrustar la asignación en cualquier expresión.

Ejemplo: C, C++ y Java

```
while ((ch = getchar())!= EOF){...}
```

Sentencia de Asignación
Asignación en Expresiones

Muchos lenguajes consideran asignación como un operador, cuya evaluación define un tipo y un valor del dato, lo que permite incrustar la asignación en cualquier expresión.

Ejemplo: C, C++ y Java

```
while ((ch = getchar())!= EOF){...}
```

Ventaja:

- Permite codificar en forma más compacta.

Sentencia de Asignación
Asignación en Expresiones

Muchos lenguajes consideran asignación como un operador, cuya evaluación define un tipo y un valor del dato, lo que permite incrustar la asignación en cualquier expresión.

Ejemplo: C, C++ y Java

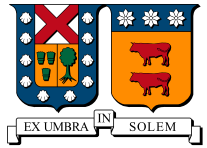
```
while ((ch = getchar())!= EOF){...}
```

Ventaja:

- Permite codificar en forma más compacta.

Desventajas:

- Es fácil equivocarse confundiendo == y =.
- Esta característica puede provocar efectos laterales, siendo fuente de error en la programación.



Estructuras de Control de Ejecución de Sentencias

Estructuras de Control de Flujo

Ejecución es típicamente secuencial, quedando implícitamente definida por el orden de definición de sentencias. Normalmente se requieren dos tipos de sentencias de control para alterar esta secuencia:

- **Selección:** Permite ofrecer múltiples alternativas de ejecución, controladas por una condición.
- **Iteración:** Ejecución repetitiva de un grupo de sentencias, también controladas por una condición.

Se requiere, además, un mecanismo de agrupación de sentencias sobre las cuales se ejerce el control (composición de sentencias).

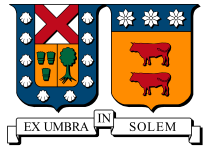
En principio basta tener un simple `goto` selectivo para alterar el orden, pero es poco estructurado.

Estructuras de Control de Flujo
Sentencias Compuestas

Una sentencia compuesta es un mecanismo que permite agrupar un conjunto de sentencias como una unidad o bloque.

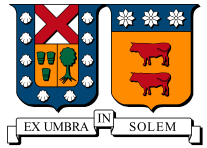
Ejemplo:

- `begin` y `end` en derivados de Algol (p.e. Pascal)
- Paréntesis de llave $\{ \ldots \}$ en derivados de C (p.e. C++ y Java)
- Python lo realiza por indentación.



Sentencia Binaria: Dos alternativas.

- En C `if-else` o `if` (solo)

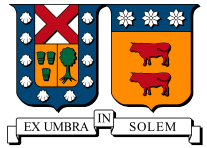


Sentencia Binaria: Dos alternativas.

- En C `if-else` o `if` (solo)

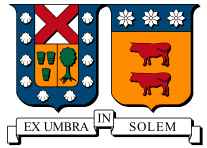
Sentencia Múltiple: Múltiples alternativas.

- `case` en Pascal y ADA.
- `switch` en C, C++ y Java.
- `else if` en C, C++, Java y ADA.
- `elif` en Python.



Bucles controlados por contador: Se especifica valor inicial y final de una variable que controla el número de iteraciones.

- `for` en C y Pascal.

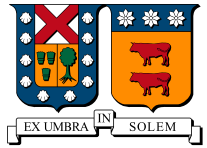


Bucles controlados por contador: Se especifica valor inicial y final de una variable que controla el número de iteraciones.

- `for` en C y Pascal.

Bucles controlados por condición: Existe condición lógica (booleana) de término. Normalmente condición debiera incluir variable que es modificada por el bloque.

- `while` y `do-while` en C, C++ y Java.



Bucles controlados por contador: Se especifica valor inicial y final de una variable que controla el número de iteraciones.

- `for` en C y Pascal.

Bucles controlados por condición: Existe condición lógica (booleana) de término. Normalmente condición debiera incluir variable que es modificada por el bloque.

- `while` y `do-while` en C, C++ y Java.

Bucles controlados por Estructuras de Datos:

- `for-each` en Perl, Python y Ruby.

Bucles controlados por contador: Se especifica valor inicial y final de una variable que controla el número de iteraciones.

- `for` en C y Pascal.

Bucles controlados por condición: Existe condición lógica (booleana) de término. Normalmente condición debiera incluir variable que es modificada por el bloque.

- `while` y `do-while` en C, C++ y Java.

Bucles controlados por Estructuras de Datos:

- `for-each` en Perl, Python y Ruby.

Observación: Lenguajes proveen también mecanismos de escape de la iteración, como `continue` y `break` en derivados de C.

Estructuras de Control de Flujo
Ejemplo Mecanismos de Escape

```c
#include <stdio.h>
#include <conio.h>
void main() {
    clrscr();
    int n;
    do {
        printf("\nIngrese numero:");
        scanf("%d", &n);
        if (n < 0) {
            break;
        }
        if (n > 10) {
            printf("Saltarse el valor\n");
            continue;
        }
        printf("El numero es: %d", n);
    } while (n!= 0);
    printf("Fin del programa\n");
}
```

Estructuras de Control de Flujo
Salto Incondicional

Un salto incondicional usa rótulos o etiquetas para especificar el punto de transferencia del control cuando se ejecuta una determinada sentencia de salto.

Restricciones: ¿Sólo rótulos en el ámbito de la variable y éstos deben ser constantes?

Estructuras de Control de Flujo
Salto Incondicional

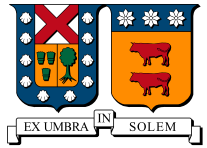
Un salto incondicional usa rótulos o etiquetas para especificar el punto de transferencia del control cuando se ejecuta una determinada sentencia de salto.

Restricciones: ¿Sólo rótulos en el ámbito de la variable y éstos deben ser constantes?

Ejemplo:

```
int n=0;

LOOP:
printf("%d\n",n);
n++;
if( n < 10 ) goto LOOP;
```



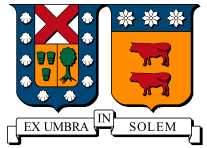
El tratamiento de errores y excepciones que suceden en la ejecución requiere de mecanismos para transferir el control al lugar donde se maneje, posiblemente interrumpiendo el flujo normal del control.

Estructuras de Control de Flujo
Manejo de Errores y Excepciones

El tratamiento de errores y excepciones que suceden en la ejecución requiere de mecanismos para transferir el control al lugar donde se maneje, posiblemente interrumpiendo el flujo normal del control.

Ejemplo: C++

```
try {
    ...  //Bloque controlado
} catch (Overflow) {
    ...  //Manejar Overflow
} catch (MathError) {
    ...  //Manejar MathError que no sea Overflow
}
```



Subprogramas

Subprogramas

Un **subprograma** describe una interfaz, que abstrae del proceso de computación definido por su implementación (cuerpo), que a través de un mecanismo de invocación permite su ejecución, con una posible transferencia de parámetros y resultados.

- Encapsula código, haciéndolo transparente para el invocador.
- Permite reutilizar código, ahorrando memoria y tiempo de codificación.
- Normalmente requiere memoria dinámica de *stack*.

Subprogramas

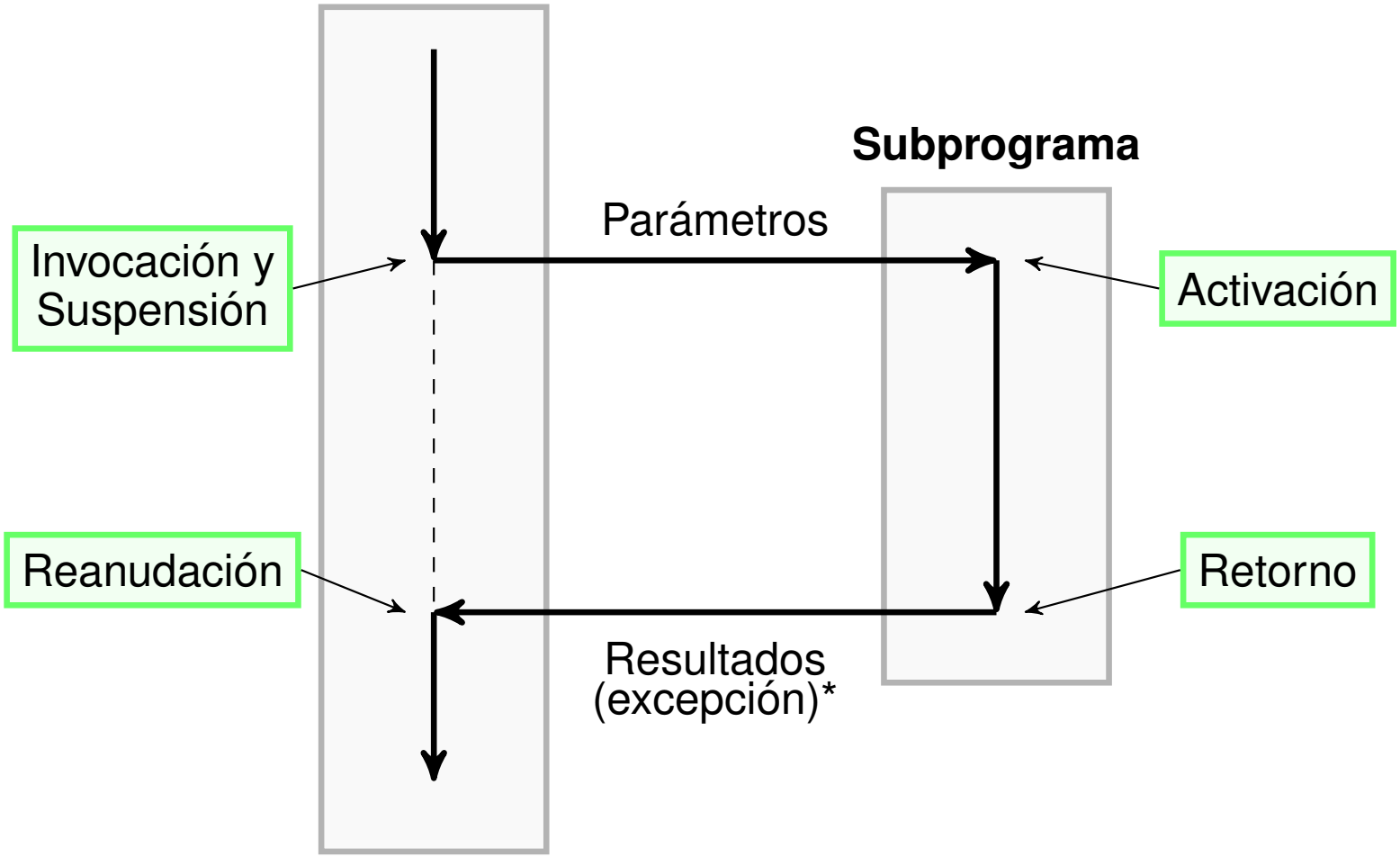
Un **subprograma** describe una interfaz, que abstrae del proceso de computación definido por su implementación (cuerpo), que a través de un mecanismo de invocación permite su ejecución, con una posible transferencia de parámetros y resultados.

- Encapsula código, haciéndolo transparente para el invocador.
- Permite reutilizar código, ahorrando memoria y tiempo de codificación.
- Normalmente requiere memoria dinámica de *stack*.

Alternativas:

- Procedimientos y subrutinas (no definen valor de retorno).
- Funciones (similar, pero con valor de retorno).
- Métodos y constructores en lenguajes orientados a objetos.

Subprogramas
Mecanismo de Invocación



Se incluyen los siguientes elementos:

- **Nombre:** permite hacer referencia al subprograma como unidad e invocarlo.
- **Parámetros (Opcional):** Define la comunicación de datos (nombre, orden y tipo de parámetros formales).
- **Valor de retorno:** Opción para funciones (tipificado).
- **Excepciones (Opcional):** Permite manejo de un evento de excepción al retornar el control anormalmente (e.g. error en ejecución del subprograma).

Subprogramas
Protocolo

El **Perfil de parámetros** de un subprograma contiene el número, orden y tipo de sus parámetros formales.

El **Protocolo** es el perfil de parámetros más el tipo de retorno en caso de ser una función.

En lenguajes como C y C++ una función debe al menos ser declarada indicando su protocolo aunque sin necesidad de definir su cuerpo. Esta declaración es llamada **Prototipo**.

Subprogramas
Ejemplos en C y Python

```
float potencia(float base, float exp){ ... }
calculo = x * potencia(y, 2.5);
```

Subprogramas
Ejemplos en C y Python

```c
float potencia(float base, float exp){ ... }
calculo = x * potencia(y, 2.5);
```

```c
int notas[50];
...
void sort (int lista[], int largo);
...
sort(notas, 50);
...
void sort (int lista[], int largo){ ... }
```

Subprogramas
Ejemplos en C y Python

```c
float potencia(float base, float exp){ ... }
calculo = x * potencia(y, 2.5);
```

```c
int notas[50];
...
void sort (int lista[], int largo);
...
sort(notas, 50);
...
void sort (int lista[], int largo){ ... }
```

```python
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a)
        a, b = b, a+b

>>>fib(40)
0 1 1 2 3 5 8 13 21 34
```

Los parámetros permiten comunicación explícita de datos, pero también a veces de (otros) subprogramas que se pasan como referencia.

- **Parámetros formales** son variables mudas que se ligan a los **parámetros actuales** cuando se activa el subprograma.
- Normalmente el ligado se hace según posición en la lista, definida en el protocolo.
- **Comunicación implícita** e indirecta sucede si el subprograma accede a variables no locales (puede producir efectos laterales).

Subprogramas
Clases de Valores

Según el tratamiento que permiten los lenguajes con el valor de variables que representan subprogramas (referencias o punteros), se definen las siguientes clases:

- **Primera Clase:** puede ser pasado como parámetro o retornado en un subprograma, como también asignarlo a una variable.
- **Segunda Clase:** puede ser pasado como parámetro, pero no retornado o asignado a una variable.
- **Tercera Clase:** no puede ser usado como parámetro, retornado ni asignado a una variable.

Estructura: un subprograma consiste de dos partes separadas:

- **Código:** representa el código real del subprograma, que es normalmente inmutable (ejecutable).
- **Registro de Activación:** variables locales, parámetros y dirección de retorno, entre otros.

Estructura: un subprograma consiste de dos partes separadas:

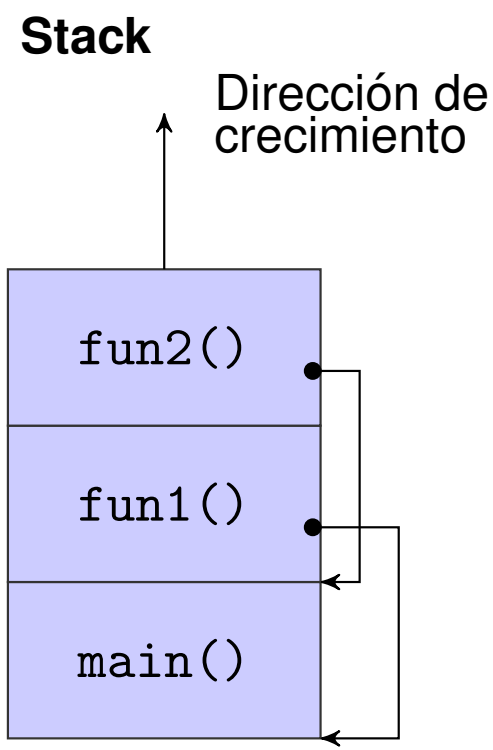
- **Código:** representa el código real del subprograma, que es normalmente inmutable (ejecutable).
- **Registro de Activación:** variables locales, parámetros y dirección de retorno, entre otros.

Tipos de implementación:

- **Simple:** no se permite anidamiento de un subprograma y datos del registro de activación son estáticos (p.e. primeras versiones de FORTRAN).
- **Stack:** permite anidamiento de llamadas, usando variables dinámicas de stack. Soporta bien recursión.

Estructura de Registro de Activación

Valor funcional	Valor de retorno (cuando aplica)
Variables Locales	
Parámetros	Para paso de Parámetros
Enlace Dinámico	Apunta a base de registro anterior
Dirección de Retorno	Dirección de próxima instrucción de llamador



Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```

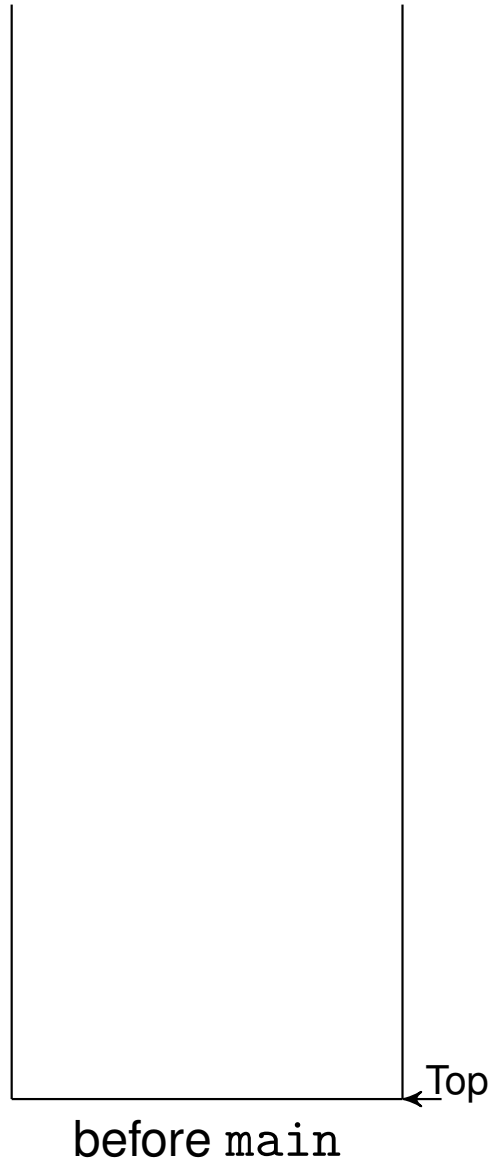
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



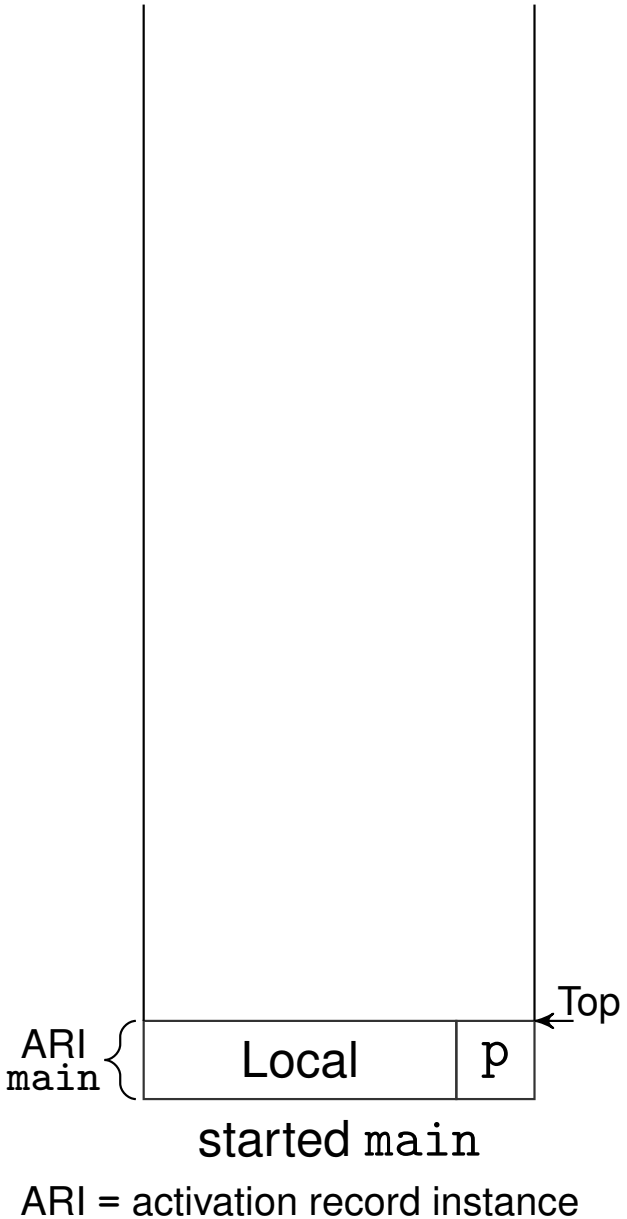
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



ARI = activation record instance

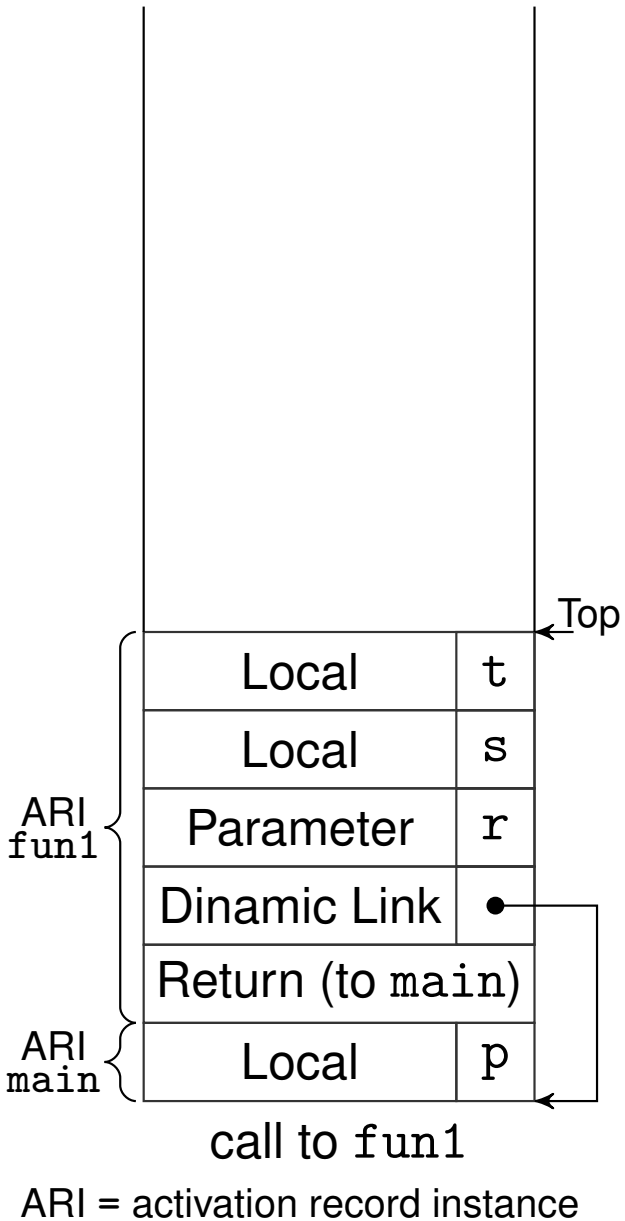
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



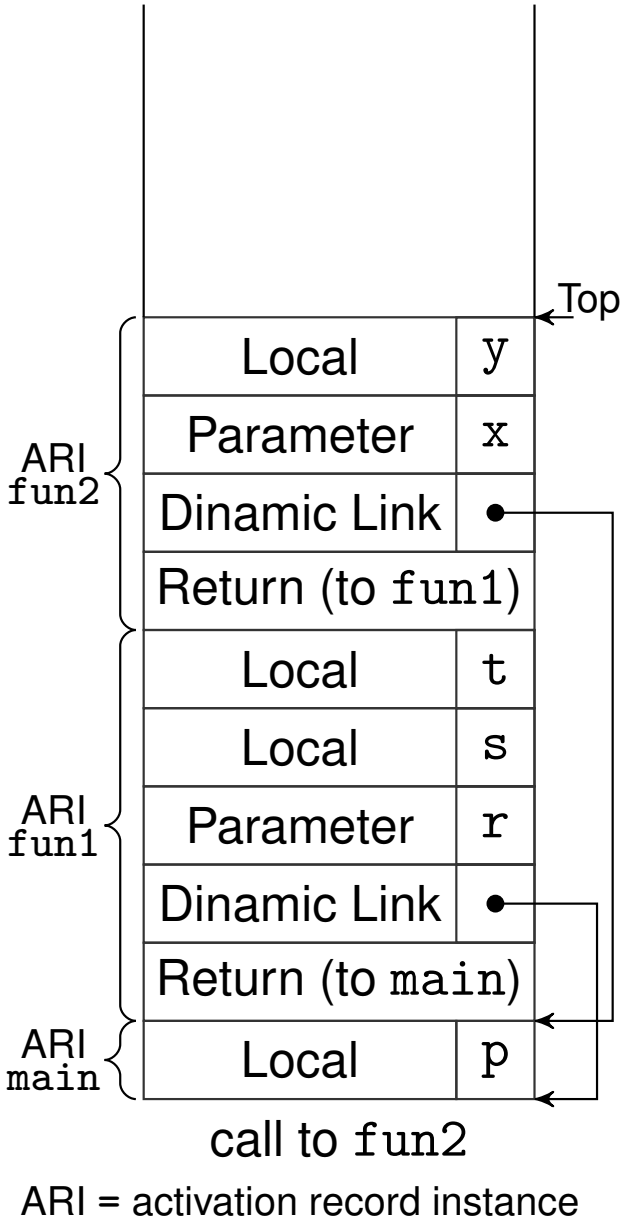
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



ARI = activation record instance

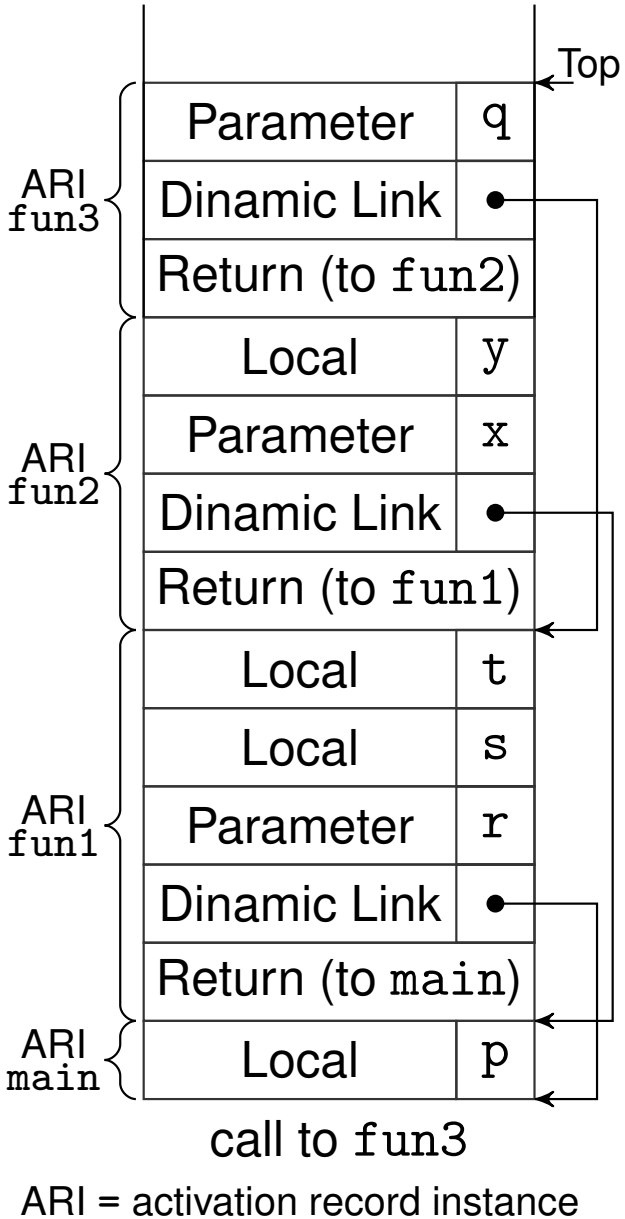
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



ARI = activation record instance

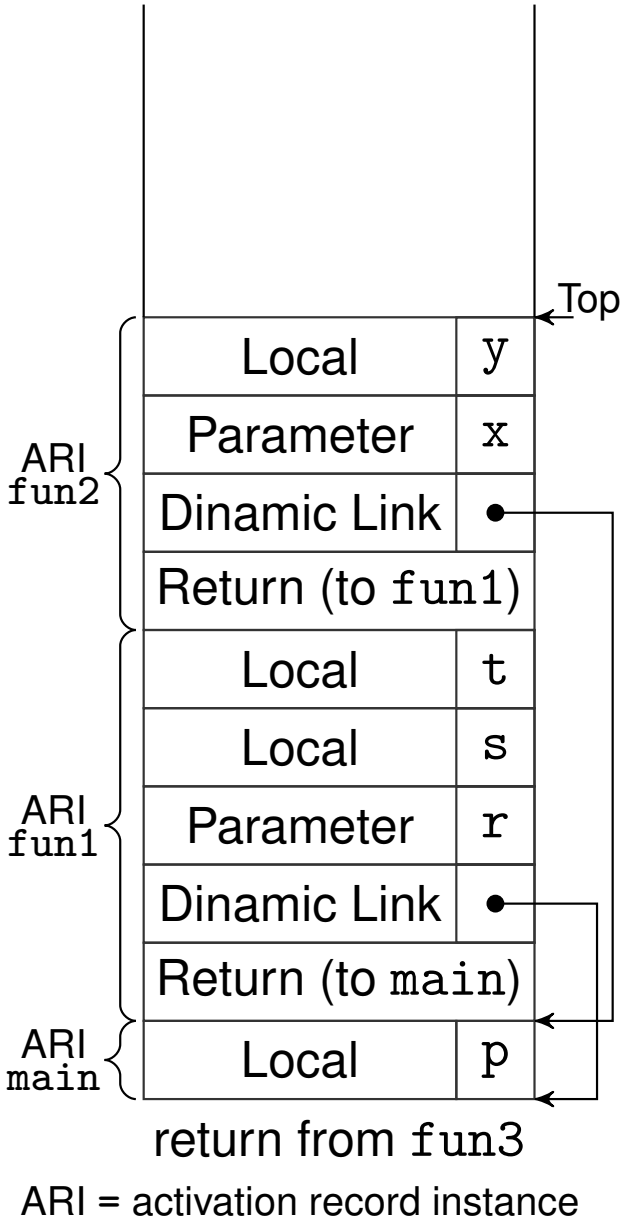
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



ARI = activation record instance

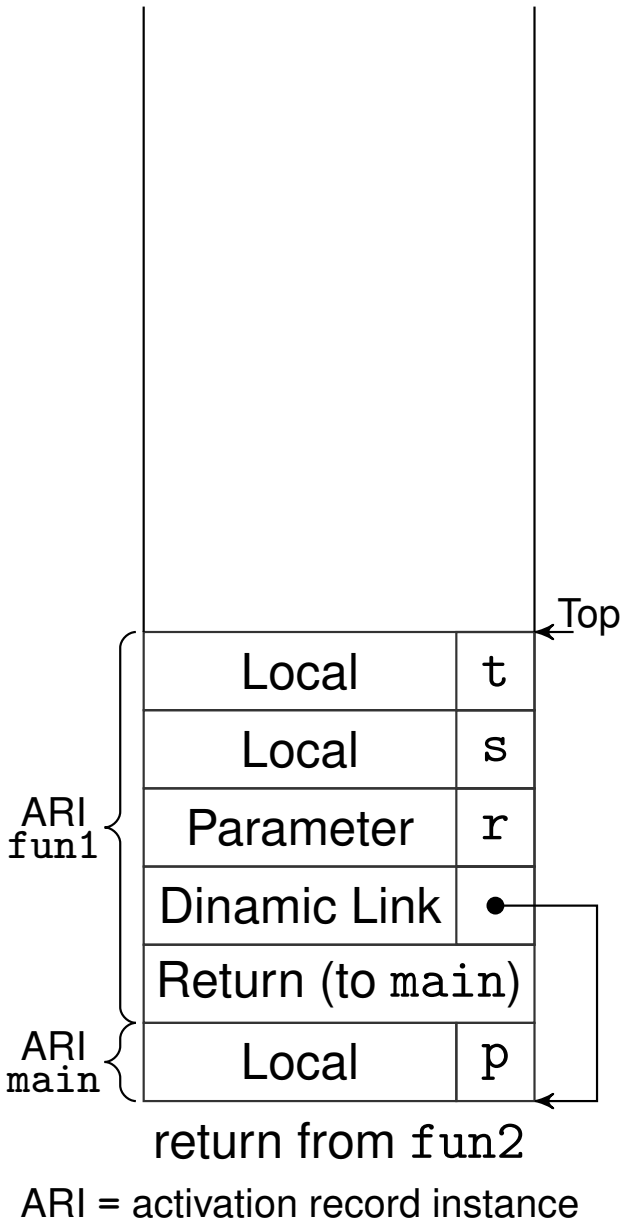
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



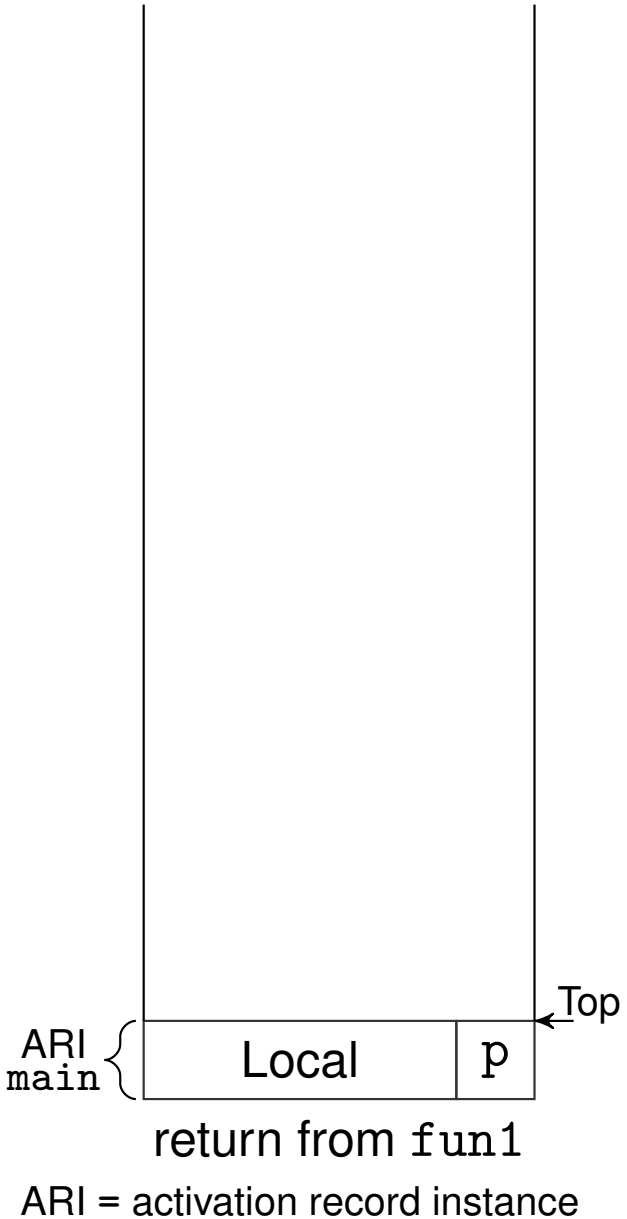
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



ARI = activation record instance

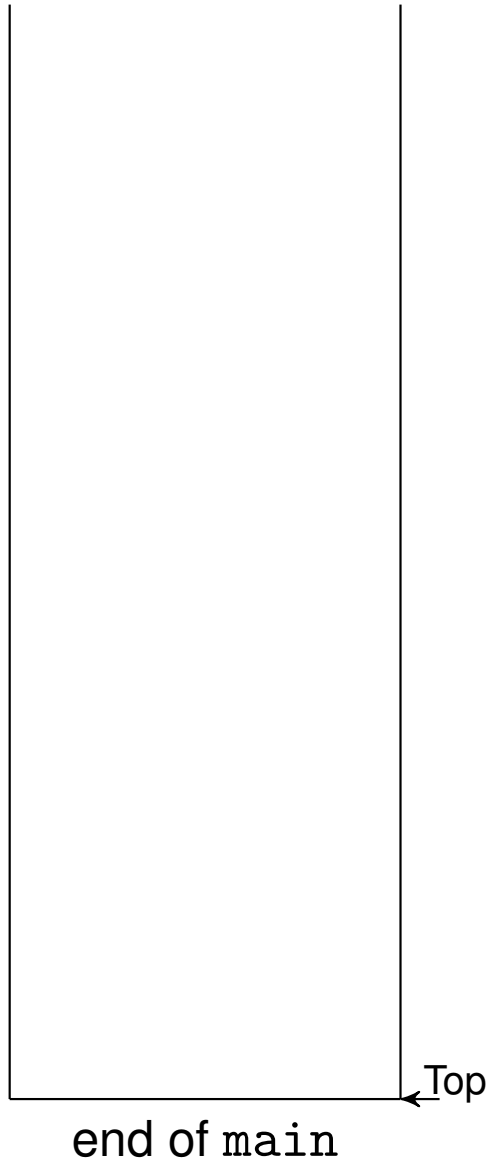
Subprogramas
Registro de Activación

```
void fun1(float r) {
    int s, t;
    fun2(s);
}

void fun2(int x) {
    int y;
    fun3(y);
}

void fun3(int q) {
}

void main() {
    float p;
    fun1(p);
}
```



Paso de Parámetros
Modelo Semántico

Dirección: Modo de interacción de parámetro actual a formal puede ser:

- Entrega de valor (IN)
- Recibo de valor (OUT)
- Ambos (INOUT)

Paso de Parámetros
Modelo Semántico

Dirección: Modo de interacción de parámetro actual a formal puede ser:

- Entrega de valor (IN)
- Recibo de valor (OUT)
- Ambos (INOUT)

Implementación: La implementación de la transferencia de datos puede ser:

- Copiando valores
- Pasando referencias (o punteros)

Paso de Parámetros
Paso por Valor

Modo IN es implementado normalmente con copia de valor

- Implementación con paso de referencia requiere protección de escritura, que puede ser difícil.

Permite protegerse de posibles modificaciones al parámetro actual, pero es más costoso

- Requiere más memoria y tiempo de copiado, pero es más seguro.

Permite usar expresiones como parámetro actual.

Paso de Parámetros
Paso por Resultado

Modo OUT es normalmente implementado con copia

- Mismas complicaciones que en paso por valor

Parámetro formal actúa como variable local, pero al retornar copia valor a parámetro actual.

Parámetro actual debe ser una variable.

Modo OUT es normalmente implementado con copia

- Mismas complicaciones que en paso por valor

Parámetro formal actúa como variable local, pero al retornar copia valor a parámetro actual.

Parámetro actual debe ser una variable.

Dificultades:

- Existencia de colisiones en los parámetros actual, puede conducir a ambigüedad.
- ¿Cuándo se evalúa la dirección de parámetro actual?

Paso de Parámetros
Paso por Valor-Resultado

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.

Paso de Parámetros
Paso por Referencia

Modo INOUT es implementado con referencias.

Parámetro formal y actual comparten misma variable.

Paso de Parámetros
Paso por Referencia

Modo INOUT es implementado con referencias.

Parámetro formal y actual comparten misma variable.

Ventaja: Comunicación es eficiente en:

- Espacio: no requiere duplicar variable.
- Tiempo: no requiere copiar.

Paso de Parámetros
Paso por Referencia

Modo INOUT es implementado con referencias.

Parámetro formal y actual comparten misma variable.

Ventaja: Comunicación es eficiente en:

- Espacio: no requiere duplicar variable.
- Tiempo: no requiere copiar.

Desventaja:

- Acceso es más lento: usa indirección.
- Es fuente de error: modificación de parámetro actual.
- Creación de alias a través de parámetros actuales.

Paso de Parámetros
Paso por Valor-Resultado v/s Paso por Referencia

```
procedure EJEMPLO1;
integer x;

procedure SUB (inout integer PARAM)
begin
    x := 2;
    PARAM := PARAM + 1;
end;

begin {EJEMPLO1}
    x := 1;
    SUB(X);
end
```

Paso de Parámetros
Paso por Valor-Resultado v/s Paso por Referencia

```
procedure EJEMPLO1;
integer x;

procedure SUB (inout integer PARAM)
begin
    x := 2;
    PARAM := PARAM + 1;
end;

begin {EJEMPLO1}
    x := 1;
    SUB(X);
end
```

- Por Valor-Resultado: x=>2
- Por Referencia: x=>3

Paso de Parámetros
Aspectos de Implementación

Comunicación de parámetro se realiza mediante el *stack*:

- **Por valor:** al invocar, valor de la variable se copia al *stack*.
- **Por resultado:** al retornar, valor se copia del *stack* a la variable.
- **Por valor-resultado:** combinando las anteriores.
- **Por referencia:** se escribe la dirección en el *stack*, y luego se usa direccionamiento indirecto (el más simple de implementar).

Paso de Parámetros
Implementación en Diferentes Lenguajes

C: paso por valor y por referencia usando punteros (parámetros deben ser desreferenciados).

- Punteros pueden ser calificado con `const`; se logra semántica de paso por valor (sin permitir asignación).
- Arreglos se pasan por referencia (son punteros).

Paso de Parámetros
Implementación en Diferentes Lenguajes

C: paso por valor y por referencia usando punteros (parámetros deben ser desreferenciados).

- Punteros pueden ser calificado con `const`; se logra semántica de paso por valor (sin permitir asignación).
- Arreglos se pasan por referencia (son punteros).

C++: igual que C, más paso por referencia usando operador `&` (sin necesidad de desreferenciar).

- Este operador también puede ser calificado con `const`, permitiendo semántica de paso por valor con mayor eficiencia (p.e. paso de grandes arreglos)

Paso de Parámetros
Implementación en Diferentes Lenguajes

Java: todos los parámetros son pasados por valor, pero variables a objetos son en realidad referencias a esto por lo que se puede considerar como paso por referencia.

Pascal y Modula-2: por defecto paso por valor, y por referencia si se usa calificativo `var`.

ADA: por defecto paso por valor, pero todos los parámetros se pueden calificar con `in`, `out` y `inout`.

C#: por defecto paso por valor, pero los parámetros se pueden calificar con `in`, `out` y `ref`.

Python: se usa sólo paso por referencia (paso por asignación), porque en realidad todos los datos son referencias a objetos; el parámetro actual es asignado al parámetro formal.

Paso de Parámetros
Comprobación de Tipos de Parámetros

La tendencia es hacer comprobación de tipos (estática y dinámica), lo cual permite detectar errores en el mal uso de parámetros, haciendo más confiables los programas.

- Pascal, Modula-2, Fortran 90, Java y ADA lo requieren.

C en su primera versión no lo requiere, pero a partir de ANSI C, como también en C++, es requerido (método de prototipo).

- Ante tipos distintos, el compilador realiza coerción, si esto es posible.

En Python y Ruby variables no tienen tipo, por lo que no es posible hacer comprobación de tipos de parámetros.

Paso de Parámetros
Comprobación de Tipos de Parámetros en C++

Todas las funciones deben usar la forma de prototipo (ANSI C lo adoptó de C++)

Sin embargo se puede desactivar mediante una elipsis.

Paso de Parámetros
Comprobación de Tipos de Parámetros en C++

Todas las funciones deben usar la forma de prototipo (ANSI C lo adoptó de C++)

Sin embargo se puede desactivar mediante una elipsis.

Ejemplos:

```
double power(double base, float exp);

int printf(const char*, ...);
```

Subprograma como Parámetro

En Pascal:

```
function integrar (function fun(x: real): real;
                   bajo, alto: real): real;
...
x := integrar(coseno, -PI/2, PI/2);
...
```

Subprograma como Parámetro

En Pascal:

```
function integrar (function fun(x: real): real;
                   bajo, alto: real): real;
...
x := integrar(coseno, -PI/2, PI/2);
...
```

En C (solo usando puntero a funciones):

```
float integrar (float (*fun)(float), float bajo,
     float alto) {...}

float coseno (float rad) {...}

float seno (float rad) {...}

x = integrar(&coseno, -PI/2, PI/2);
y = integrar(&seno, -PI/2, PI/2);
```

Sobrecarga de Subprogramas

En el mismo ámbito, existen diferentes subprogramas con el mismo nombre. Cada versión debiera tener un protocolo diferente, de manera que a partir de los parámetros actuales se pueda resolver a cuál versión se refiere.

- Las versiones pueden diferir en la codificación.
- Es una conveniencia notacional, que es evidente cuando se usan nombres convencionales, como en el ejemplo siguiente.

Sobrecarga de Subprogramas

En el mismo ámbito, existen diferentes subprogramas con el mismo nombre. Cada versión debiera tener un protocolo diferente, de manera que a partir de los parámetros actuales se pueda resolver a cuál versión se refiere.

- Las versiones pueden diferir en la codificación.
- Es una conveniencia notacional, que es evidente cuando se usan nombres convencionales, como en el ejemplo siguiente.

Ejemplos:

- C++, Java, C# y Ada incluyen subprogramas predefinidos y permiten escribir múltiples versiones de subprogramas con el mismo nombre, pero diferente protocolo.

Sobrecarga de Funciones
Ejemplo en C++

```
double abs(double);
int abs(int);

abs(1);     // invoca int abs(int);
abs(1.0);   // invoca double abs(double);


//sobrecarga de print
void print(int);
void print(char*);
```

Sobrecarga de Operadores
Ejemplo en C++

```
int operator* (const vector<int> &a,
               const vector<int> &b)
{
    int len = a.size()<b.size()?a.size():b.size();
    int sum = 0;
    for (int i = 0; i < len; i++)
      sum += a[i] * b[i];
    return sum;
}

vector<int> x, y;
...
printf("%i", x * y); // producto punto
```

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

Subprogramas Genéricos
Ejemplo 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);
```

Clausuras

Una clausura es un subprograma y un ambiente referencial donde éste fue definido (que permite invocarlo de diferentes ámbitos).

Clausuras

Una clausura es un subprograma y un ambiente referencial donde éste fue definido (que permite invocarlo de diferentes ámbitos).

Ejemplo: Python

```
>>> def make_adder(x):
...         def adder(y):
...             return x+y
...         return adder
...
>>> add10 = make_adder(10)
>>> add5 = make_adder(5)
>>> add10(20)
30
>>> add5(20)
25
```

Subprogramas
Resumen

La definición de un **subprograma** describe las acciones que representa éste.

Subprogramas pueden ser procedimientos, funciones, y en OO también métodos o constructores.

Variables de subprogramas pueden ser **estáticas** o **dinámicas** de stack. Esta última soporta bien **recursión**.

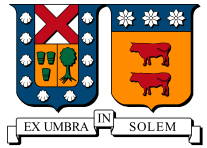
Tres modelos de paso de parámetro: **IN**, **OUT** y **INOUT**.

Algunos lenguajes permiten **sobrecargar** subprogramas, también operadores.

Subprogramas pueden ser **genéricos**, donde el tipo de parámetros se establece al usarlos.

Una **clausura** es un subprograma con su ambiente referencial.

Referencias



- Capítulos VII, VIII y IX de [Sebesta, 2011]
- Capítulos IX y X de [Louden, 2011]
- Capítulo IX y X de [Tucker, 2006]