LENGUAJES DE PROGRAMACIÓN

Equipo de Profesores:
Jorge Díaz Matte - José Luis Martí Lara – Rodrigo Salas Fuentes

Unidad 4

Control

4.1 Expresiones y Asignaciones

4.2 Estructuras de Control de Ejecución de Sentencias

4.3 Subprogramas

4.1 Expresiones y Asignaciones

Introducción

- Expresiones son el medio fundamental para especificar en un lenguaje computacional lo que debe realizar la máquina.
- El valor de las expresiones depende del orden de evaluación de operadores y operandos, determinado por reglas de asociatividad y precedencia.
- Ambigüedades en el orden de la evaluación puede conducir a diferentes resultados, según la implementación del traductor de lenguaje.
- Lenguajes imperativos están dominados por asignaciones a variables que modifican el estado del programa, con potenciales efectos laterales.
- En lenguajes funcionales, en principio, las variables se usan como parámetros de funciones y no producen efectos laterales.

Expresiones Aritméticas

DEFINICIÓN: consisten en operadores, operandos (argumen-tos), 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

Expresiones Aritméticas: Evaluación de Operadores

- Precedencia: determina el orden de evaluación de opera-dores adyacentes con diferente nivel de precedencia. Se prioriza el orden de evaluación de mayor a menor nivel de precedencia.
- Asociatividad: define el orden de evaluación de operadores adyacentes con el mismo nivel de precedencia.
- Paréntesis: altera y fuerza orden de evaluación.

Expresiones Aritméticas: Evaluación de Operadores

Reglas de Precedencia:

Fortran	Pascal	C
**	*, /, div, mod	Postfix ++, --
*, /	+, -	Prefix ++, --
+, -		+, - (unario)
		*, /, %
		+, - (binario)

Alta

Baja

Expresiones Aritméticas: Evaluación de Operadores

Reglas de Asociatividad:

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

Expresiones Aritméticas: 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 incrustadas en las instrucciones de máquina.
- Paréntesis: evalúa todos los operandos y operadores contenidos primero, y luego su valor puede ser usado como operando.

Expresiones Aritméticas: 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 (ej.: INOUT).
  - Referencias a variables no locales.

Expresiones Aritméticas: Evaluación de Operandos

EFECTOS LATERALES:

```
int a = 2;

int f1() {
   return a++;
}

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

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

Expresiones Aritméticas: Evaluación de Operandos

EFECTOS LATERALES: ¿cómo evitarlos?

- 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 o sólo permitir parámetros bidireccionales.

- 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 Aritméticas: Evaluación de Operandos

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!

Sobrecarga de Operadores

DEFINICIÓN: un mismo operador es usado para diferentes propósitos, siendo que tienen distinto comportamiento. Ayuda a una mejor legibilidad.

- Ejemplo:

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

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

Conversión de Tipos

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

- Ejemplo: de entero a punto flotante; de subrango de enteros a entero.

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

- Ejemplos: de real a entero; paso del tipo base a subrango.

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

Conversión de Tipos: explícita

DEFINICIÓN: una expresión puede tener una mezcla de operandos de diferentes tipos; entonces el lenguaje hace una conversión implícita (coerción) para usar el operador adecuado…la mayoría de los lenguajes usan conversión por expansión.

- Ejemplo: Java convierte todos los enteros más cortos (*byte*, *short*) 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.
- El programador puede controlar explícitamente la conversión mediante mecanismo de "*casting*".
  - Ejemplo en C: (int) angulo

Conversión de Tipos: Errores en Expresiones

DEFINICIÓN: 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:

- división por cero, desbordamiento (*overflow*), "sub"des-bordamiento (*underflow*).
- algunos lenguajes producen una excepción en estos casos.

Expresiones Relacionales y Booleanas

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 booleanas, donde se puede combinar con expresiones relacionales.

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

Expresiones Relacionales y Booleanas

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.

Expresiones Relacionales y Booleanas

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

```
C:        b + 1 > b*2        /* aritméticos primero */

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

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

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

Cortocircuito

Término anticipado de evaluación:

C:

```
(13*a) * (b/13 -1)

(a >=0) || (b < 10)

while ( (c = getchar()) != EOF && c != ´\n´) {
     procesar linea;
```

PASCAL:

```
i := 1;
WHILE (i <= listlen) AND (list[i] <> key) DO
         i := i+1;
```

Cortocircuito

- C, C++ y Java definen cortocircuito para operadores lógicos: && y || (AND y OR).
- Perl, Python y Ruby evalúan todos sus operadores lógicos con corto-circuito.
- Pascal no lo especifica (algunas implementaciones los tienen, otras no); algo parecido sucede con Fortran.

Sentencias de Asignación

DEFINICIÓN: 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).

Sentencias de 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;
```

Sentencias de Asignación: Operadores

C, C++ y Java permiten operadores unarios y compuestos:

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

equivale a:

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

```
sum =  ++contador;
```

equivale a:

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

```
cont++;
```

equivale a:

```
cont = cont+1;
```

Sentencias de Asignación: Múltiple

- PL/I permite:
- SUM, TOTAL = 0

- C, C++ y Java permiten:
- SUM = TOTAL = 0

- Python además permite:
- x, y, z = 20, 30, 40

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

Sentencias de Asignación: en Expresiones

DEFINICIÓN: 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.
- Desventaja:
- Esta facilidad puede provocar efectos laterales, siendo fuente de error en la programación (ej: es fácil equivocarse confundiendo *==* y *=*)

4.2 Estructuras de Control de Ejecución de Sentencias

Estructura del Flujo de control

DEFINICIÓN: 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.

Sentencias Compuestas

DEFINICIÓN: es un mecanismo que permite agrupar un conjunto de sentencias como una unidad o bloque.

Ejemplos:

- begin y end en derivados de Algol (ej.: Pascal).
- Paréntesis de llave {…} en derivados de C (ej.: C++ y Java).
- Python lo realiza por indentación.

Sentencias de Selección

Selección binaria: dos alternativas (o una o ninguna)

- Ejemplo: En C if-else o if (solo)

Selección múltiple: múltiples alternativas

- Ejemplos:
  - case en Pascal
  - switch en C, C++ y Java
  - elseif en ADA

```
/* version N °1 */

int abs(int n) {
  if (n>=0)
      return n;
  else return –n;
}
```

```
/* version N °2 */

int abs(int n) {
  return (n>=0)? n : -n;
}
```

Sentencias de Iteración

- Bucles controlados por contador: se especifica valor inicial y final de una variable que controla el número de iteraciones. Ej.: *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. Ejs.: *while, do-while* en C, C++ y Java; *loop-exit* en ADA.

- Bucles controlados por Estructuras de Datos: como *foreach* en Perl, Python y Ruby.

Sentencias de Iteración

Observación: los lenguajes de programación proveen también mecanismos de escape de la iteración, como *continue* y *break* en derivados de 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);
}
```

Salto Incondicional

DEFINICIÓN: 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: PHP

```
<?php
        for($i=0,$j=50; $i<100; $i++) {
                while($j--) {
                if($j==17) goto end;
                }
        }
        echo "i = $i";
        end:
        echo 'j hit 17';
?>
```

4.3 Subprogramas

Subprogramas

DEFINICIÓN: describen 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.

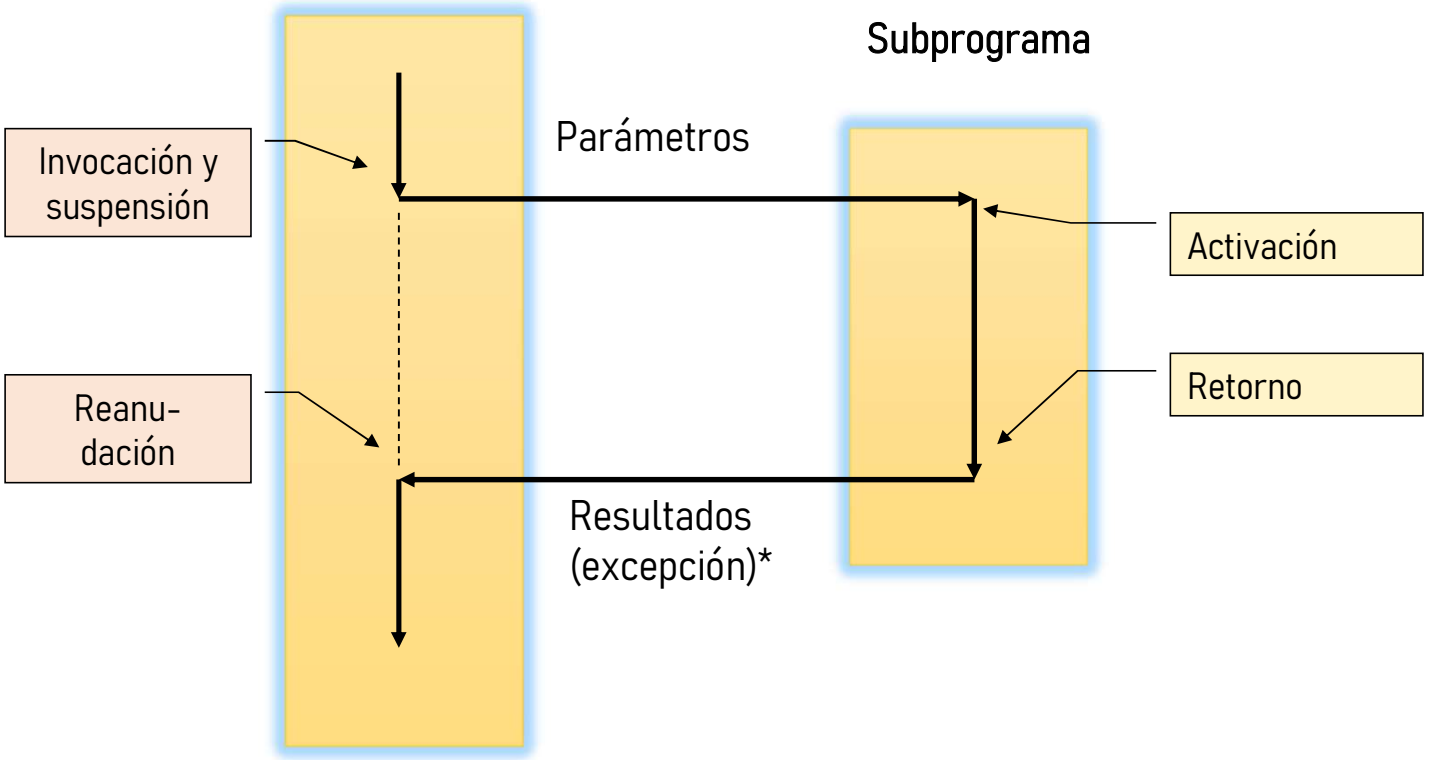
- Encapsula código, haciéndolo transparente para el invocador.
- Permite reutilizar código, ahorrando memoria y tiempo de codificación.
- Normalmente requiere memoria dinámica de *stack*.

Subprogramas

Posibilidades:

- 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



Subprogramas: Interfaces

Se incluyen, al menos, los siguientes elementos:

- Nombre: permite referenciar 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 (ej.: error en ejecución del subprograma).

Subprogramas: Interfaces

- Firma o prototipo: define un contrato entre el invocador y el subprograma, definiendo la semántica de la interfaz para el intercambio de parámetros y resultados durante una invocación.
- Especifica sintácticamente un posible nombre, parámetros formales y retorno, con sus respectivos tipos.
- Corresponde a la cabecera del subprograma.

- Protocolo: especifica cómo debe realizarse la comunicación de parámetros y resultados (tipo y orden de los parámetros y, opcionalmente, valor de retorno). Establece cómo se asocian los parámetros reales con los parámetros formales.

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

Subprogramas: Parámetros

- DEFINICIÓN: 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 reales 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:

- Ejemplos:

```
float potencia(float base, float exp);

calculo = x * potencia(y, 2.5);
```

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

C

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

Python

Subprogramas: Clases de Valores

DEFINICIÓN: 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 pasados como parámetro, pero no retornado o asignado a una variable.
- Tercera Clase: no puede ser usados como parámetros, retornados ni asignados a una variable.

Subprogramas: aspectos de Implementación

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: tipos de implementación:

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

Subprogramas: Registros de Activación

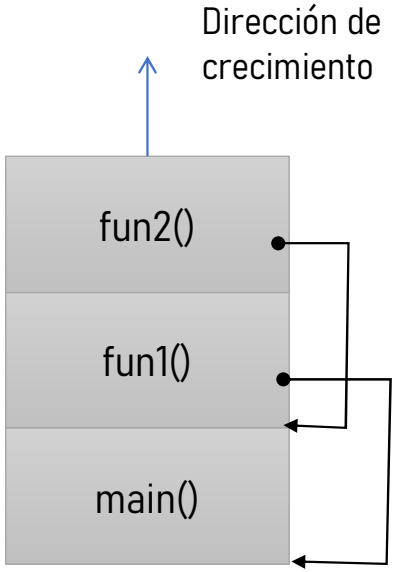
Estructura de Registro de Activación

Paso de parámetros

Apunta a base de registro anterior

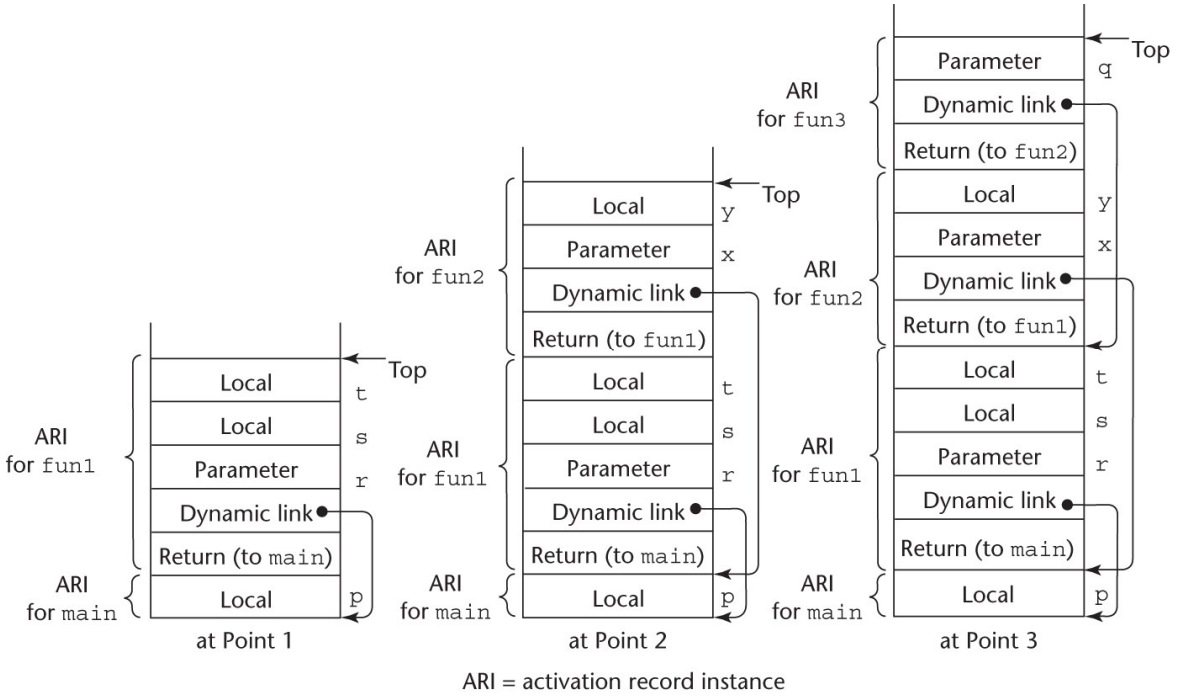
Dirección de próxima instrucción de llamador

Stack



Subprogramas: Registros 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: Modelos Semánticos

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: Modelos Semánticos

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 real, pero es más costoso.
- Requiere más memoria y tiempo de copiado, pero es más seguro.
- Permite usar expresiones como parámetro real.

Paso de Parámetros: Modelos Semánticos

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 real.
- Parámetro real debe ser una variable.

Dificultades:

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

Paso de Parámetros: Modelos Semánticos

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: Modelos Semánticos

Paso por Referencia:

- Modo INOUT es implementación con referencias
- Parámetro formal y real comparten misma variable

Ventaja: Comunicación es eficiente en:

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

Desventajas:

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

Paso de Parámetros: Modelos Semánticos

Ejemplo:

```
procedure EJEMPLO1;
integer x;

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

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

¿por valor?

¿por valor – resultado?

¿por referencia?

Paso de Parámetros: ejemplos

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 el & (sin necesidad de desreferenciar).

- Este operador también puede ser calificado con *const*, permitiendo semántica de paso por valor con mayor eficiencia (ej.: paso de grandes arreglos).

Paso de Parámetros: ejemplos

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

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

Java: todos los objetos son pasados por referencia, y sólo los parámetros que no lo son se pasan por valor – la no existencia de punteros no permite paso por referencia de los tipos escalares (si como parte de un objeto).

Python: se usa sólo paso por referencia (paso por asignación), porque en realidad todos los datos son objetos que tienen una referencia; por lo que al pasar el valor del objeto, lo que se pasa es su referencia.

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: comprobación de Tipos

- 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, ADA

- En su primera versión C no la requiere, pero a partir de ANSI C, como también en C++, es requerido (método de prototipo). Ante incompatibilidad de tipos, el compilador realiza coerción, si esto es posible.

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

Paso de Parámetros: comprobación de Tipos

- 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* ...);
```

Paso de Parámetros: subprogramas

- En Pascal:

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

- En C (sólo usando punteros a funciones):

```
float integrar (float *fun(float),
                float bajo, float alto);
...
x = integrar(&coseno, -PI/2, PI/2);
...
```

Subprogramas: Sobrecarga

DEFINICIÓN: 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 cuál versión se refiere (protocolo diferente).

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

Ejemplos:

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

Subprogramas: Sobrecarga en C++

- de Funciones:

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

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

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

- de Operadores:

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

➔

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

Subprogramas: Genéricos

DEFINICIÓN: 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.

Subprogramas: Genéricos

Observación: tipos de polimorfismos.

- Polimorfismo dinámico (o polimorfismo paramétrico): es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado a todo tipo de datos compatible. Ej.: funciones genéricas.
- Polimorfismo estático (o polimorfismo *ad hoc*): es aquél en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizado.

Subprogramas: Genéricos

- Ejemplo de Función Genérica 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);
```

Clausura

- DEFINICIÓN: una clausura es un subprograma y un ambiente referencial donde éste fue definido (que permite invocarlo de diferentes ámbitos).
- Ejemplo: JavaScript

```
 function makeAdder(x) {
     return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);

document.write("add 10 to 20: " + add10(20) + "<br />");
document.write("add 5 to 20: " + add5(20) + "<br />");
```

Subprogramas: Resumen

- 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 de programación permiten sobrecargar subprogramas, también operadores.
- Subprogramas pueden ser genéricos, donde el tipo de parámetro se establece al usarlos.

Unidad 4

Control

FIN

4.1 Expresiones y Asignaciones

4.2 Estructuras de Control de Ejecución de Sentencias

4.3 Subprogramas