Lenguajes de Programación I

Control de Flujo Estructurado y No Estructurado

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Flujo No Estructurado

"Ah, I might as well jump. Jump!" – Van Halen

- Ensamblable sólo permite saltar para variar el flujo de ejecución.
- Los primeros lenguajes simplemente imitaron este comportamiento.
  - Etiquetas para instrucciones.
  - Instrucción `goto` para saltar a una etiqueta.
  - Traducción trivial de alto nivel a ensamblable.
- *Calculated Goto* – aritmética sobre una variable para producir la etiqueta a la cual hay que saltar.

Goto considered harmful

- 1960-1970: `goto` o no `goto`.
- 1968: Dijkstra pone en evidencia lo perjudicial de usar `goto`
- La consolidación de trabajos anteriores y posteriores a ese, demostraron que cualquier algoritmo imperativo bien diseñado podía expresarse únicamente con secuencias, selecciones e iteraciones **sin** usar `goto`.
  - La Programación Estructurada cobró como víctima al `goto`.
  - Aumentó el nivel de abstracción de los lenguajes de programación – aumentó la complejidad de la generación de código.
- Primitivas `if-then-else`, `for` y `while` son suficientes – aquí coincidieron la teoría y la práctica de lenguajes.

Sólo quedaban algunos casos especiales
(por no decir patológicos).

Solución 1 – Abandonar un ciclo a mitad de camino

- Con `goto` en Pascal

```
    while not eof do begin
      readln(line);
      if all_blanks(line) then goto 42;
      do_something(line);
    end;
42:
```

- `break` en C, `last` en Perl, `exit` en Modula.

```
while ($line = <ARCHIVO>) {
  last if $line =~ /^\w*$/;
  do_something($line);
}
```

Solución 2 – Omitir el resto de la iteración actual

- Con goto en Pascal

```
    while not eof do begin
      readln(line);
      if is_comment(line) then goto 42;
      do_something(line);
42:
    end;
```

- continue en C, next en Perl.

```
while ($line = <FILE>) {
  next if $line =~ /^#.*$/;
  do_something($line);
}
```

Solución 3 – Quiero retornar de una subrutina

- En Pascal era muy frecuente algo como

```
    procedure procesar(var line : string)
    ...
    begin
    ...
      if is_comment(line) then goto 42;
      (* Resto de la subrutina *)
42:
    end;
```

- O con una variable booleana que debía verificarse a cada rato.
- Sólo fue necesario introducir el return explícito

Solución 4 – Retorno Multinivel voluntario

Las rutinas anidadas y sus complicaciones

```
  proc A()
    proc B()
    begin
    ...
      if foo then begin
        goto 42
      end;
    ...
    end B;
  begin
    ...
42:
  end;
```

- La rutina A contiene a la rutina B.
- Estoy ejecutando la rutina B – obviamente A está en pila.
- Retornar pero *desde la rutina A*.
- Saltar a una etiqueta que está *fuera* del alcance inmediato de B.
- No es un simple return
  - Hay registros de activación en medio.
  - Completar "retornos intermedios" – *unwinding the stack*.

Solución 5 – Retorno Multinivel involuntario

- Después de haber ejecutado con varios niveles de anidamiento, es necesario cancelar la ejecución hasta algún punto en el pasado, pues se detectó un error.
  - Sólo con goto – saltar al sitio conveniente, que no tiene por que estar en el alcance estático del origen del salto.
  - Con return simple – valores especiales de retorno para señalar errores.
- Ambos problemas han sido resueltos de forma general con facilidades para *manejo de excepciones* – estudiaremos con detalle más adelante.

¿Cuál es la diferencia entre los Problemas 4 y 5?

Secuenciación

- Separador de instrucciones que evidencia la secuencia.
  - En lenguajes como Java, C, Perl es el punto y coma (;) – prácticamente el estándar de hecho.
  - En lenguajes como Ruby es el *salto de línea*.
  - En lenguajes como Python es la *cantidad de blancos*.
- ¿Separador o terminador?
- Agrupar varias instrucciones para considerarlas una sola
  - *Compound statements*.
  - Enmarcados en delimitadores especiales.
  - Pueden usarse en contextos donde va una instrucción.

Selección

Bifurcación condicional

- Variantes del `if-then-else` de Algol60.

```
if condicion1 then instruccion1
else if condicion2 then instruccion2
...
else instruccionN
```

- Formas prefijas adaptadas a LISP y derivados

```
(cond ((condicion1) (expresiones1))
      ((condicion2) (expresiones2))
      ...
      (t            (expresionesN)))
```

- Extensiones interesantes (Perl)
  - `unless (...)` para no tener que escribir `if (! ...)`.
  - Selectores como modificadores sufijos.

Selección múltiple

...porque una cascada de `if` es inconveniente

```
i := exprI
if i = 1 then
  inst_A
elsif i in 2, 7 then
  inst_B
elsif i in 3..5 then
  inst_C
elsif i = 10 then
  inst_D
else
  inst_E
end
```

```
case exprI
  1:    inst_A
| 2, 7: inst_B
| 3..5: inst_C
| 10:   inst_D
  else  inst_E
end
```

Instrucciones `case/switch`

```
case exprI
  1:    inst_A
| 2, 7: inst_B
| 3..5: inst_C
| 10:   inst_D
  else  inst_E
end
```

- **Brazo** – cada fragmento de código.
- **Etiqueta** – expresión que selecciona un brazo.
  - Disjuntas – selección unívoca.
  - Tipo discreto – comparación exacta.
- Cláusula por omisión – alternativas
  - No hay cláusula por omisión.
  - Palabra clave `else` u `otherwise` – Pascal.
  - Etiquetas deben cubrir `todo` el dominio – Ada.
- Uso de rangos como etiquetas – sintaxis variada entre lenguajes.
- `break` vs. caída a través (*fall through*).

El `switch` de C

```
switch ( ... /* Expresión a probar */ ) {
  case 1:  inst_A
           break;
  case 2:
  case 7:  inst_B
           break;
  case 3:
  case 4:
  case 5:  inst_C
           break;
  case 10: inst_D
           break;
  default: inst_E;
           break;  /* No hace falta */
}
```

¿Qué tiene de malo una cascada de `if`?

El código generado es mediocre

```
i := exprI
if i = 1 then
  inst_A
elsif i in 2, 7 then
  inst_B
elsif i in 3..5 then
  inst_C
elsif i = 10 then
  inst_D
else
  inst_E
end
```

```
    r1 := ...
    if r1 <> 1 goto L1
    inst_A
    goto L6
L1: if r1 = 2 goto L2
    if r1 <> 7 goto L3
L2: inst_B
    goto L6
L3: if r1 < 3 goto L4
    if r1 > 5 goto L4
    inst_C
    goto L6
L4: if r1 <> 10 goto L5
    inst_D
    goto L6
L5: inst_E
L6:
```

Poca localidad – invalidación de caché

¿Qué tiene de bueno un `case`?

El código más simple es mejor que el anterior

```
    goto L6
L1: inst_A    - i = 1
    goto L7
L2: inst_B    - i = 2 o 7
    goto L7
L3: inst_C    - i = 3, 4 o 5
    goto L7
L4: inst_D    - i = 10
    goto L7
L5: inst_E    - else
    goto L7
```

```
T:  &L1
    &L2
    &L3
    &L3
    &L3
    &L5
    &L2
    &L5
    &L5
    &L4
L6: r1 := ...
    if r1 < 1 goto L5
    if r1 > 10 goto L5
    r1 := r1 - 1
    r2 := T[r1]
    goto *r2
L7:
```

¡Acceso indirecto FTW!

Y no es la única alternativa

Implantaciones eficientes según el dominio

- **Tabla de Despacho Indirecta** – el caso simple que vimos.
  - Pocas etiquetas en un dominio denso y con rangos pequeños.
  - Tabla de etiquetas con direcciones destino.
  - Tiempo constante para determinar el brazo.
- **Tablas de Despacho con Búsqueda Binaria**
  - Muchas etiquetas distribuidas en un rango amplio.
  - Tabla ordenada por etiqueta con direcciones destino.
  - Tiempo logarítmico para determinar el brazo.
- **Tablas de Despacho con Hash**
  - Muchas etiquetas distribuidas en un rango amplio.
  - Tabla de hash por etiqueta con direcciones destino.
  - Tiempo constante para determinar el brazo.

Compilador genera código específico según escenario.

Ni `case` ni `switch` en Perl

- Perl tiene instrucciones compuestas etiquetables –
  se interpretan como una iteración de una sola pasada.
- Entonces uno escribe

```
SUICHE: {
  if ($o eq "abc") { $abc = 1; last SUICHE; }
  if ($o eq "def") { $def = 1; last SUICHE; }
  if ($o eq "xyz") { $xyz = 1; last SUICHE; }
  $nothing = 1;
}
```

- Perl 5.8 provee la librería `Switch` que simula `case` y `switch`
  - No se modificó el lenguaje.
  - *Source filter* que convierta a la forma antes descrita.
  - Aprovecha el alcance dinámico para el `switch`
  - Casos pueden ser variables, valores, patrones o funciones anónimas

Iteración

- Típica en los lenguajes imperativos.
- Un lazo (*loop*) o ciclo involucra una instrucción (simple o compuesta) cuya ejecución debe repetirse para que sus efectos de borde sean efectuados
- El control puede ser por Enumeración o Condición.

Iteraciones controladas por Enumeración

Iteración acotada

- Cuerpo se ejecuta tantas veces como valores haya en un rango finito.

```
FOR I := inicial TO final STEP paso
```

- El número de iteraciones se conoce por *adelantado*.
  - Se indican valor inicial y final.
  - Se avanza de uno en uno, a menos que se especifique un *paso*.
- No se ejecutan si el rango definido es vacío.

$$[inicial, inicial + (\lfloor final - inicial / paso \rfloor) \times paso]$$

- El código generado es sumamente eficiente.
- La mayoría de los lenguajes **prohiben**
  - Cambiar la variable índice dentro del cuerpo.
  - Modificar las variables usadas para calcular límites.

Iteraciones controladas por Condición

Iteración no acotada

- Cuerpo se ejecuta hasta que una condición booleana cambia de valor.
- Número de iteraciones no se conoce por adelantado, depende de los efectos de borde del lazo y puede ser infinito.
- ¿Dónde se verifica la condición?
  - Al principio (`while` condición `do` instrucción).
  - Al final (`do` instrucción `while` condición).
  - En medio (`loop` ... `exit if` condicion .. `end`).

Interrupción y Continuación Etiquetada

...por aquello de matar al `goto`

- Posible interrumpir o continuar selectivamente lazos etiquetados.
- **`break`** en C o **`last`** en Perl.
  - Sin etiqueta – interrumpe el lazo más cercano.
  - Con etiqueta – interrumpe el lazo más cercano con esa etiqueta, y todos los anidados en medio.
- **`continue`** en C o **`next`** en Perl.
  - Sin etiqueta – salta el resto de la iteración más cercana.
  - Con etiqueta – salta el resto de la iteración más cercana con esa etiqueta y todos los anidados son terminados.
- PHP no tiene etiquetas – ¡usa un **número** para indicar cuántos niveles de anidamiento se verán afectados!

¿Qué tipo de Iteración es ésta?

- En C o Java

```
for (i = 0; i < 42; i++) { ... }
```

- En Perl

```
for my $letra ("a".."z") { .. }
```

- En Ruby

```
42.times { ... }
```

Continuaciones

El futuro está en tus manos

- La forma más general de control de flujo.
- Invocar una instrucción preservando la "continuación" actual
  - Si la instrucción tuvo éxito, construyó su propio futuro, así que no necesita regresar a su continuación original.
  - Si la instrucción no tuvo éxito, puede continuar hacia el pasado recuperando la continuación que tenía guardada.
- Perfecta en lenguajes puros pues simplemente debe descartar los registros de activación intermendios.
- Disponible explícitamente en Scheme – `call/cc`.
- Disponible explícitamente en Ruby – `callcc`

La madre de todos los `goto`

Bibliografía

- [*Scott*]
  - Secciones 6.2, 6.3, 6.4, 6.5.1 y 6.5.5
  - Ejercicios y Exploraciones
- Dijkstra, 1968
  Goto considered harmful