

Control de Flujo (ch 6.2-)

Kenny Davila
Agosto 18, 2021

Flujo Estructurado y No Estructurado

- En ensamblador el flujo se basa en saltos condicionales
- Muchos lenguajes antiguos lo imitaban usando etiquetas y “goto”

```
    if (A .lt. B) goto 10  
    ...  
10
```

Flujo Estructurado y No Estructurado

- El uso de “goto” en lenguajes de alto nivel ha sido controversial
 - Permite código “spaguetti”
 - Los detractores han ganado
 - Lenguajes antiguos lo soportaban por compatibilidad
 - Algunos lenguajes lo soportan en contextos limitados
 - Lenguajes modernos no lo soportan
- El cambio que hizo que los “goto” desaparecieran fue la **programación estructurada**

La Programación Estructurada

- Surgió en los 70's
- Se basa en múltiples principios:
 - Diseño de arriba hacia abajo (top-down)
 - Modularización del Código
 - Tipos estructurados
 - registros, conjuntos, punteros, arreglos multidimensionales
 - Variables descriptivas
 - Nombres de constantes
 - Convenciones para los comentarios

La Programación Estructurada

- Casi cualquier algoritmo imperativo bien diseñado se puede expresar de manera elegante mediante:
 - Secuenciación
 - Selección
 - Iteración
- En lugar de usar etiquetas, los lenguajes estructurados se basan en los límites de construcciones anidadas como objetivos del control de las ramificaciones del flujo

La Programación Estructurada

- Algol 60 fue un pionero
 - If ... then ... Else
 - For
 - While
- El bloque caso moderno (switch) se introdujo en Algol W

La Programación Estructurada

- Los gotos eventualmente fueron reemplazados por versiones estructuradas según el tipo de salto deseado
- Uno podía usar goto para saltar al final de la subrutina actual
 - Ahora se puede usar “return” en cualquier parte de la subrutina
- Uno podía usar un goto para escapar de una iteración
 - Ahora se permite el uso de “break” o “exit”
 - También se provee “continue” para saltar a la siguiente iteración
- Uno podía usar goto para saltar en caso de encontrar un error
 - Ahora se permite capturar errores en funciones anidadas usando Try... Catch

Retornos Multinivel

- Dar un salto directo desde la rutina actual hasta una rutina que hizo el llamado original en una serie de llamadas anidadas es mas complejo de lo que parece
 - Función A llama a B
 - Función B llama a C
 - Función C llama a D
 - Función D falla y se desea retornar a A inmediatamente
- El salto podría ocurrir en cualquier momento (ocurre un error)

Retornos Multinivel

- El salto podría ocurrir en cualquier momento (ocurre un error)
- Se debe garantizar que el flujo se reestablecerá correctamente
- A esta operación se le conoce como “unwinding”
 - Los registros de activación deben liberarse
 - Los registros del procesador deben restaurarse
 - Se simula que se hizo un retorno de cada rutina en el stack

Manejo de Errores

- Cada función tiene un “contrato” o definición de funciones esperadas
- Si el contrato falla, la función debe “retroceder”
 - Es diferente a retornar
 - Se les conoce como “exceptions”
- Se retrocede a un punto donde el programa tiene los medios para recuperarse

Manejo de Errores

- Se necesita de un “handler” que se pueda utilizar en caso de que ocurra un error
- En caso de error, el “handler” determina y aplica la acción correctiva necesaria
- Si no hay error, el código finaliza de manera regular y el handler no es llamado

Manejo de Errores

- El manejo de errores y el retorno multinivel se parecen pero NO son lo mismo
- El retorno multinivel los medios para retornar de forma “normal”
- El manejo de errores se aplica solo en ciertos casos
 - Cuando la función no puede completar su trabajo
 - En caso de ser ejecutado, se aplica como post-procesamiento

Manejo de Errores

- El manejo de errores es mas común que el retorno multi-nivel
 - Lisp y Ruby son casos raros que soportan ambos
- Common Lisp y Ruby usan los términos catch and throw para referirse al retorno multinivel
 - En otros lenguajes, estos términos se asocian a manejo de errores

Continuaciones

- Se refiere a direcciones de código o entorno de referencia al que se debe retornar junto con la referencia a otra continuación en caso de retornar de manera subsecuente
- La cadena de continuaciones de retorno constituye el “backtrace” de una pila de ejecución
- Una continuación es una abstracción que captura el contexto donde la ejecución podría continuar

Continuaciones

- Algunos lenguajes (Scheme, Ruby) permiten usar las continuaciones como valores de primera clase
 - El programador puede definir nuevas construcciones de flujo de control
- Esto hace que algunos lenguajes de programación sean muy extensibles
- Usados de forma incorrecta pueden llegar a ser muy difíciles de entender

Secuenciación

- Central en la programación imperativa
- Una instrucción sigue a la anterior ...
- Hay construcciones que permiten agrupar instrucciones y usarlas en contextos donde solo se permite una instrucción
 - Begin ... End
 - { ... }
 - Compound Statements
 - CS + Declaraciones = Bloque

Secuenciación

- Algunos lenguajes permiten secuenciación pero limitan los efectos colaterales validos en funciones
 - Ejemplo: Euclid, Turing
- La propiedad matemática de la idempotencia
 - Una operación se puede realizar múltiples veces y siempre retorna el mismo resultado para las mismas entradas
- Los efectos colaterales pueden romper la idempotencia de una función

Secuenciación

- La idempotencia no siempre es deseable:
 - Caso: generación de números aleatorios

```
procedure srand(seed : integer)
    -- Initialize internal tables.
    -- The pseudorandom generator will return a different
    -- sequence of values for each different value of seed.

function rand() : integer
    -- No arguments; returns a new "random" number.
```

Selección

- La notación del if ... Then ... Else fue introducida en Algol 60
- El uso de “Else” puede llegar a ser ambiguo si no hay delimitadores o reglas de asociatividad

Selección

- La anidación de ifs puede ser incomoda, por eso algunos lenguajes introducen el uso de “elsif” o “elif”

- Ruby:

```
if a == b then
  ...
elsif a == c then
  ...
elsif a == d then
  ...
else
  ...
end
```


Selección

- Las expresiones booleanas en los ifs deben evaluarse y su valor no necesita ser almacenado, solo lo usamos para saltar condicionalmente
 - Lo mismo aplica a otros bloques (e.g. while) que usan condiciones
- A este tipo de evaluaciones se les conoce como **jump code**
- En lugar de usar temporales para guardar resultados, se hacen saltos directamente

Selección

- Código sin corto-circuitos:

```
if ((A > B) and (C > D)) or (E ≠ F) then
    then_clause
else
    else_clause
```

```
r1 := A          -- load
r2 := B
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3
r1 := r1 & r2
r2 := E
r3 := F
r2 := r2 ≠ r3
r1 := r1 | r2
if r1 = 0 goto L2
L1: then_clause    -- (label not actually used)
    goto L3
L2: else_clause
L3:
```

Selección

- Código con corto-circuitos:

```
if ((A > B) and (C > D)) or (E ≠ F) then  
    then_clause  
else  
    else_clause
```


```
r1 := A  
r2 := B  
if r1 <= r2 goto L4  
r1 := C  
r2 := D  
if r1 > r2 goto L1  
L4: r1 := E  
    r2 := F  
    if r1 = r2 goto L2  
L1: then_clause  
    goto L3  
L2: else_clause  
L3:
```


Selección

- Resultados booleanos no se guardan en temporales
 - Se usan de forma implícita para saltar
- Nuevamente, el orden afecta la eficiencia
- Pero ... Y si necesitamos usar el valor para una asignación a una variable booleana?

Selección

- Ejemplo: Ada



```
found_it := p /= null and then p.key = val;  
  
if p /= null and then p.key = val then  
    found_it := true;  
else  
    found_it := false;  
end if;
```

```
r1 := p  
if r1 = 0 goto L1  
r2 := r1 → key  
if r2 ≠ val goto L1  
r1 := 1  
goto L2
```

```
L1: r1 := 0
```

```
L2: found_it := r1
```

Selección

- Bloque caso
 - Se introduce para aquellos casos donde una condición se compara con diferentes constantes en tiempo de compilación

Selección

- Ejemplo: Ada

```
i := ... -- potentially complicated expression
if i = 1 then
    clause_A
elsif i = 2 or i = 7 then
    clause_B
elsif i in 3..5 then
    clause_C
elsif i = 10 then
    clause_D
else
    clause_E
end if;
```

```
case ... -- potentially complicated expression
is
    when 1      => clause_A
    when 2 | 7  => clause_B
    when 3..5   => clause_C
    when 10     => clause_D
    when others => clause_E
end case;
```

Selección

- Bloque caso
 - Se introduce para aquellos casos donde una condición se compara con diferentes constantes en tiempo de compilación
- Los bloques de código se conocen como brazos
- La lista de constantes se conoce como “etiquetas del bloque caso”
- Las constantes deben ser disjuntas y deben ser compatibles con la condición de prueba
- Usualmente deben ser valores discretos
 - Enteros, caracteres, enumeraciones sub rangos
 - Cadenas (C#, JAVA)

Selección

- Ejemplo: Ada

```
case ... -- potentially complicated expression
is
    when 1      => clause_A
    when 2 | 7  => clause_B
    when 3..5   => clause_C
    when 10     => clause_D
    when others => clause_E
end case;
```

```
r1 := ... -- calculate controlling expression
if r1 ≠ 1 goto L1
  clause_A
  goto L6
L1: if r1 = 2 goto L2
    if r1 ≠ 7 goto L3
L2: clause_B
    goto L6
L3: if r1 < 3 goto L4
    if r1 > 5 goto L4
    clause_C
    goto L6
L4: if r1 ≠ 10 goto L5
    clause_D
    goto L6
L5: clause_E
L6:
```