Lenguajes de Programación

Control de Flujo

Ma. Laura Cobo

Universidad Nacional del Sur
Departamento de Ciencias e Ingeniería de la Computación
2019

Introducción

El control de flujo es uno de los temas centrales de un lenguaje de programación.

La manera como se incorpore en el lenguaje determina el orden de ejecución de un programa

Los mecanismos del lenguaje utilizados para especificar orden de ejecución, pueden ordenase en categorías

- Secuencia
- Selección
- Iteración
- Abstracción procedural
- Recursión
- Concurrencia
- Manejo de excepciones o especulación
- No determinismo

2

Evaluación de expresiones

Las expresiones son una forma fundamental de especificar computación en un lenguaje de programación.

Además de la forma de las mismas (sintaxis: BNF), su significado (semántica) es crucial.

La semántica, esta gobernada por su forma de evaluación

Es necesario estar conscientes de los ordenes de evaluación de operandos y operadores

Una expresión consiste generalmente de un objeto simple (contante literal, constante nombrada o variable) o un operador o una función aplicada a una colección de operandos o argumentos (cada uno de los cuales define una expresión)

3

Expresiones

Las expresiones constan de:

- Operadores
- Operandos
- Paréntesis
- Llamadas a función

Los operadores pueden ser:

- Unarios: tienen sólo un operando
- Binarios: tienen dos operandos

Algunos lenguajes definen su operadores como "*syntactic sugar*" para conseguir un aspecto mas normal. Así por ejemplo `a+b` es internamente `+(a,b)`

Expresiones Aritméticas

La mayoría de los lenguajes imperativos adoptan una notación infija para los operadores, pero hay otras opciones:

- Prefija
  - `op a b` o `op(a,b)` o `(op a b)`
- Infija
  - `a op b`
- Posfija
  - `a b op`

La mayoría de los lenguajes imperativos utilizan notación infija para los operadores binarios y prefija para los unarios.

5

Expresiones

Aspectos de diseño:

- Reglas de precedencia de operadores
- Reglas de asociatividad de operadores

Control

- Orden de evaluación de los operandos
- Evaluación de efectos colaterales de un operador
- Sobrecarga de un operador
- Modo mixto de expresiones
- Errores en las expresiones

6

Reglas de precedencia y Asociatividad

Define el orden en el cual operadores adyacentes (con diferente precedencia) son evaluados.

Los niveles de precedencia típicos son:

- Paréntesis
- Operadores unarios
- ** (potenciación)
- *, /
- +, -

Las reglas de precedencia y asociatividad pueden modificarse con el uso de paréntesis

7

Asignación

En el paradigma funcional, el computo consiste en evaluar expresiones.

Las expresiones resultan ser los bloques básicos del lenguaje.

En el paradigma imprativo, el computo consiste en una serie ordenada de cambios (de los valores de las variables en memoria).

Las asignaciones resultan el medio principal para realizar esos cambios.

El constructor de una lenguaje de programación produce un **efecto colateral**, si influencia al resto de la computación de otra manera que no sea a través de su retorno

8

Asignación

La sintaxis general tiene la siguiente forma:

<variable destino> <operador de asignación> <expresión>

El operador de asignación varia de lenguaje de programación en lenguaje de programación.

A:= B	Algol, Pascal, Ada
A = B	C, Fortran, Java
MOVE B TO A	Cobol
A ← B	APL

9

Asignación

A simple vista, la asignación parece una operación directa.

A nivel implementación, sin embargo, se producen diferencias sutiles que pueden dar lugar a diferentes tipos de asignación.

Estas diferencias a simple vista pueden resultar invisibles (no afectan el comportamiento en programas simples)

En C:

```
d = a
```

El lado derecho hace referencia al valor de `a`

```
a = b + c
```

El lado izquierdo hace referencia a la locación de `a`

10

Asignación

Se puede definir la operación de asignación como:

1. Computar el valor del lado izquierdo (**l-value**)
2. Computar el valor de la expresión del lado derecho (**r-value**)
3. Asignar el valor computado del lado derecho al computado como objeto de dato del lado derecho
4. Retornar el valor computado como calor del lado derecho como resultado de la asignación

l-value: las expresiones que denotan locaciones

r-value: las expresiones que denotan valores

La semántica de la operación de asignación puede pensarse como:

- Copia de la referencia
- Dereferenciamiento implícito

11

Operadores de asignación combinada

Para evitar situaciones de asignación triviales, como

```
a = a+1
```

algunos lenguajes proveen operadores abreviados , asi la sentencia anterior puede escribirse como:

```
a++
```

12

Asignación

La operación de asignación puede pensarse como:

- una instrucción (no retorna ningún valor)
- una expresión

Instrucción	Pascal, Ada, Algol, Fortran, PL/1
Expresión	C, Java, APL, ML, Snobol 4

Así por ejemplo, en C uno puede escribir la siguiente asignación:

C = (B = 2)

La cual se interpreta como se le asigna el valor de 2 a B. El resultado de la expresión B=2, es decir 2, se la asigna a C. Por lo cual se le asigna 2 a C

13

Asignación

La asignación puede en algunas circunstancias darse en forma **implícita**. Generalmente esto sucede en:

- La inicialización en las declaraciones
- Pasaje de parámetros

```
Var
   integer a;
   float b;
   int *c
```

14

Ordenamiento

Las reglas precedencia y asociatividad establecen el orden en el los operadores infijos de una expresión se aplican pero no determinan el orden de evaluación.

```
a- f(b) - c*d
```

Define como se resuelve el orden de evaluación frente a operadores del mismo nivel de precedencia.

Regla típica: evaluación de izquierda a derecha

Hay dos razones por las que el oden es fundamental:

1. Efectos colaterales
2. Optimización de código

15

Evaluación de operandos

1. Variables: cargar el valor desde memoria
2. Constantes: algunas veces son cargas desde la memoria, otras veces la constante puede ser parte de las instrucciones del lenguaje máquina
3. Expresiones parentizadas: se evalúan las expresiones de acuerdo al modo de evaluación elegido

Si ninguno de los operandos u operadores tiene efectos colaterales, entonces el orden de evaluación es irrelevante

16

Evaluación de expresiones

Nos resta ver las posibles forma de evaluar las expresiones

Los tipos fundamentales son:

1. Evaluación Estricta
2. Evaluación No Estricta

Evaluación Estricta: los argumentos de una función o los operandos de la expresión son siempre completamente evaluados antes de la aplicación de la función u operador.

Evaluación No Estricta: los argumentos de una función o los operandos de la expresión **no** son evaluados a menos que sean utilizados en el cuerpo de la función o sean necesarios para determinar el valor de la expresión.

17

Evaluación estricta

La evaluación estricta es la tradicionalmente utilizada por los lenguajes de programación.

Es también conocida como:

- Evaluación de orden aplicativo (evalúa de izq. a der.)
- Evaluación ansiosa

En este tipo de evaluación una expresión se evalúa al momento de establecer la ligadura con una variable.

Los lenguajes imperativos la utilizan casi siempre (debido a que el orden de ejecución esta definido implícitamente por la organización del código)

18

Evaluación estricta

Ventajas:

- Elimina la necesidad de planificar
- El programador puede determinar el orden de ejecución

Desventajas:

- Fuerza la evaluación de expresiones que pueden no ser necesarias en tiempo de ejecución.
- Puede demorar la evaluación de expresiones cuyo valor se requiere en forma mas inmediata.
- Fuerza al programador a organizar el código fuente para garantizar una ejecución óptima.

19

Evaluación no estricta

Cuando se implementa se denomina evaluación de orden normal.

Características:

- La evaluación se reduce por la izquierda
- Las funciones se aplican antes de evaluar sus argumentos.

Se puede especializar en distintas versiones:

- Evaluación perezosa: agrega la memorización de los argumentos evaluados.
- Evaluación corto-circuito (expresiones booleanas): se retorna el valor de la expresión tan pronto como se pueda determinar el valor de verdad sin ambigüedad

20

Evaluación de orden normal: Ejemplo

Ejemplo: Considere la siguiente llamada sumar(2+3)

```
sumar(x):
    y = 2+3 * 2
    z = 2+3 + 10
```

El resultado, se obtiene sustituyendo x con la expresión 2+3. La expresión 2+3 y no el valor 5, es pasado como el de valor de x en la función.

21

Evaluación perezosa: Ejemplo

Ejemplo: Considere la siguiente llamada sumar(2+3)

```
sumar(x):
    y = 2+3 * 2
    z = 5 + 10
```

El resultado, se obtiene sustituyendo x con la expresión 2+3, en la primera aparición de x dentro del cuerpo de sumar y por el valor de la expresión, 5, en las futuras apariciones de x en el cuerpo de la función.

22

Evaluación perezosa: Ejemplo

Ejemplo en ML fun g(x,y,z) = if x<2 then y+3 else z+6

Ejemplo en C a=0: 0 ? b/a

Es interesante notar que solo el valor de y o el valor de z son requeridos para el cálculo del resultado, pero no ambos.

Ejemplo en Haskell

```
fibs = 0 : 1 : zipWith (+) fibs(tail fibs)
```

":" agrega un elemento a la lista

Tail: retorna una lista sin su primer elemento

zipWith crea una nueva lista tomando el elemento a la cabeza de cada lista, aplicando una función, **+** en este caso

Esta función calcula una lista con todos los números Fibonacci.

23

Evaluación corto-circuito: Ejemplo

Ejemplo en C

```
int a = 0; int b = 4
if (a != b && myfun(b) )
{
    do_something();
}
```

En este ejemplo, debido a la evaluación en corto-circuito, se garantiza que la llamada a myfun(b) nunca suceda. Esto es porque a se evalúa como falso

Otro ejemplo en C

```
int TieneTresCaracteresDeLongitud(const char *p)
{
    return (p != NULL) && (strken(p) ==3);
}
```

24

Control de flujo no estructurado

Se provee a través saltos **condicionales** e **incondicionales**.

Los primeros lenguajes cercanos al hardware de la máquina trabajaban con tipos básicos y con etiquetas o rótulo y saltos como instrucciones.

Aspectos de diseño:

Los rótulos son de suma importancia, es necesario establecer las restricciones sobre ellos

- Constantes
- Expresiones sin cómputo en ejecución
- Expresiones

25

Transferencia de control

```
if E1
    then C1: goto L1
    else C2: goto L2
endif
```

```
L1: C3
    while E2 do C4
L2: C5
    goto L1
    C6
enddo
```

El uso de la instrucciones como el **goto** conducen a programas con diseño no estructurado. Haciendo muy difícil utilizar modelos de correctitud para los programas. En general sus ventajas, no amortizan sus grandes desventajas

Se puede demostrar que el uso de este tipo de instrucciones es superfluo ya que puede fácilmente simularse con secuencias de control estructuradas

26

Control de flujo estructurado

El abandono de esta forma de control fue parte de determinar la programación estructurada.

En este entorno el control no estructurado se mantuvo, pero asociado a casos especiales y con condicionantes (no se puede transferir el control a cualquier lugar).

Opciones de alto nivel:

Break: salida incondicional

Continue: se utiliza en las iteraciones. El control salta al final del bucle, con que el contador se incrementa y comienza otra comprobación

Transferencia de control restringida. Ejemplo

La idea es tener un solo punto de entrada y salida. En general es implementada en los lenguajes a través de la instrucción **break**.

Esta instrucción provoca que el programa avance hacia adelante a un punto explícito al final de la de una dada estructura de control

```
……
   While not eof() do
         readln(line);
         if blanck(line) goto 100;
         consumir(line,out);
10:      escribir(out);
   Endwhile
100: ……
```

```
Procedure consumir(
                  line: string; out:string)
Begin
   …..
   if esComentario(line,i) goto 10;
   …..
End
```

28

Programación estructurada: diseño

Resumiendo, para obtener programas más fáciles de entender, verificar, corregir, modificar y re-verificar se debe enfatizar:

1. Organización jerárquica de la estructura del programa (uso de composición, alternación e iteración)
2. Uso de estructuras de control estructuradas (una entrada – una salida por instrucción)
3. Correspondencia entre el orden del texto del programa y el orden de la ejecución
4. Usar grupos de sentencias con próposito simple, aún cuando se requiera copia.

29

Estructuras de control estructuradas

Una **estructura de control** es una sentencia de control y una colección de sentencias cuya ejecución controla

Las estructuras de control estructuradas son

- Secuencia
- Condicional
- Iteración

30

Secuencia de control estructurado

Un *sentencia compuesta* es un secuencia de instrucciones que pueden ser tratadas como una sola sentencia en la construcción de sentencia más grandes.

Representa básicamente la composición

El orden en el que aparecen en el texto del programa es el orden en el cual son ejecutadas

Aspectos de diseño:

- Delimitadores
- Separadores y terminadores
- Bloques

31

Sentencia Condicional

Una *sentencia condicional* es aquella que expresa una alternativa entre dos o mas sentencias.

El control de las alternativas es controlado por una condición, usualmente expresada a través de una expresión booleana.

La forma mas común de condicional es la sentencia *if-then-else*

Aspectos de diseño:

- Condición simple, doble o múltiple
- Instrucción simple, compuesta o secuencia
- Anidamiento
- Forma y tipo de las expresiones de control

32

Sentencias condicional doble

La forma general que adopta el condicional doble en los leguajes de programación es:

If <condicion> then <instruccion> else <instruccion>

La instrucción puede ser simple o compuesta

Veamos un ejemplo en Java, donde se utiliza este tipo de condicional:

```
If i == 0 then
        If j== 0 then
                i= 0
        else i=2
```

33

Anidamiento de sentencias condicionales

La mayoría de los lenguajes permite anidar sentencias condicionales, lo que puede provocar algunos problemas semánticos (interpretación del programa resultante)

Así si se tienen dos **if** y un solo **else**, aparece el problema de determinar a que **if** corresponde el **else**.

- Convención: asociar el **else** al **if** más cercano: C, C++, C# y Java
- Uso de sentencias compuestas: Perl
- Uso de terminador de sentencia: **endif** en Ada

34

Sentencias condicional múltiple

La sentencia **condicional múltiple** permite la selección de una de una grupo no fijo de sentencias o grupos de sentencias

Aspectos de diseño:

- Alternativas excluyentes o no
- Forma y tipo de las expresiones selectoras
- Como se especifica el segmento seleccionado
- Valores no representados
- Instrucción simple, compuesta o secuencia
- Flujo de ejecución cuando termina

35

Sentencias condicional múltiple

Si las alternativas no son excluyentes, la ejecución es no determinista. En general en estos casos el lenguaje establece reglas para determinar que alternativa va a ejecutarse y si es posible ejecutar más de una.

La sentencia **switch** de C tiene tomadas varias decisiones importantes de diseño:

1. la expresión que controla la sentencia solo puede ser de tipo entero
2. los segmentos de código seleccionables pueden ser: secuencias de sentencias, bloques o sentencias compuestas
3. cualquier cantidad de segmentos pueden ser ejecutados en una ejecución del constructor
4. no hay un salto implícito al final de cada segmento de código seleccionable

36

Sentencias condicional múltiple en los LP

Fortran:

```
IF (N) 10, 20, 30
10: …….
    GOTO 40
20: …..
    GOTO 40
30: ……
40: ……
```

Ada:

```
case <expresion> is
[when [<contante>]*:<instrucción>]*
[when others: <instruccion>]
end case
```

La versión en Ada es más confiable ya que al finalizar la secuencia elegida el control es pasado a la siguiente instrucción luego de la sentencia **case**

PL/1:

```
Select
[ when <condition> → <instrucción>]+
else <instrucción>
endselect
```

C:

```
switch (<expresion>) {
[<constante>: <instruccion>]*
[default: <instruccion>]
}
```

37

Comandos con guarda: múltiple selección con if

Forma general:

if <expresión> → <sentencia> ["[]"<expresión> → <sentencia>]* **fi**

Semántica: cuando el constructor es alcanzado

1. Evaluar todas las expresiones booleanas
2. Si mas de una es verdadera elegir una en forma no-determinista
3. Si ninguna es verdadera, se produce un error en ejecución

```
if B1 then S1
elsif B2 then S2
elsif B3 then S3
fi
```

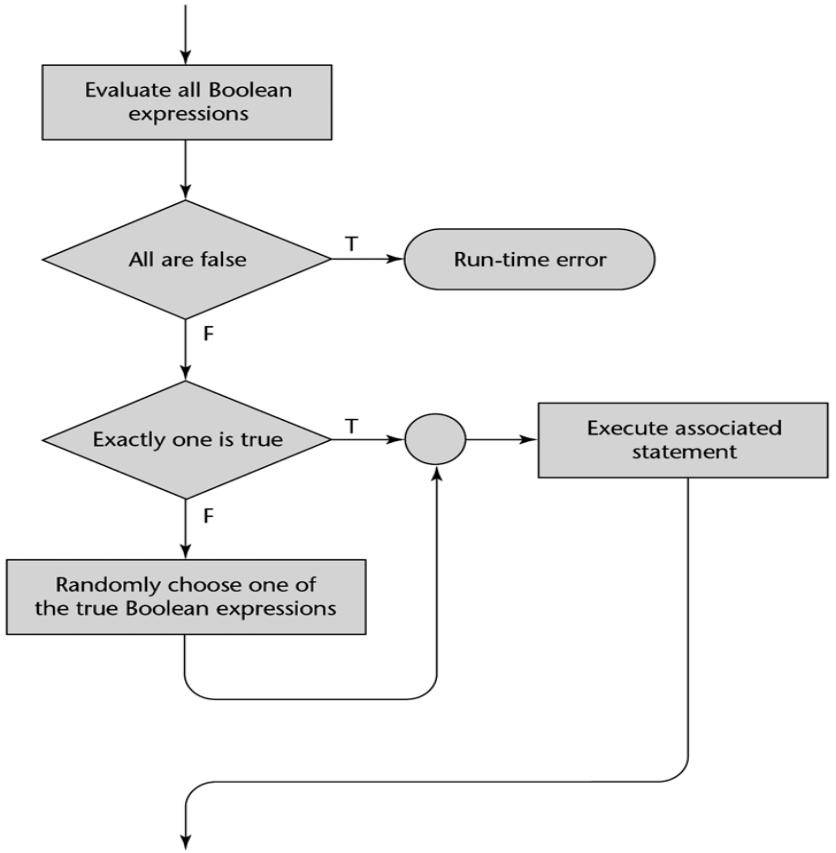
```
if B1 then S1
   else
   if B2 then S2
      else
      if B3 then S3
      fi
   fi
fi
```

38

Comandos con guarda: múltiple selección con if

Figure 8.1

Flowgraph of the approach used with Dijkstra's selector statement



39

Sentencia iterativa

La última de las estructuras de control estructuradas es la **iteración**

Una **sentencia iterativa** es aquella que cause que una sentencia o colección de sentencias se ejecute cero o más veces. Generalmente llamadas **loop** o **ciclos**

Los lenguajes que proveen facilidades para definir iteración conducen a programas menos extensos, flexibles, más simples de escribir y almacenar

Aspectos de diseño:

- ¿cómo se controla la iteración?
- ¿dónde debe aparecen el mecanismo de control en el constructor de ciclo?

40

Sentencia iterativa

Cabeza

Control

Cuerpo

Acción

- Simple
- Con contador
- Con condición
- Infinitos
- Múltiple salida
- En base a datos
- Con guarda

41

Sentencia iterativa: iteración simple

La iteración simple está presente en el lenguaje **COBOL** en el constructor **perform**

PERFORM <cuerpo> <variable> TIMES

<cuerpo>: es una instrucción simple o una secuencia de instrucciones

<variable>: determina de cantidad de iteraciones. Esta variable, puede modificarse en el <cuerpo> afectando la cantidad de iteraciones

42

Sentencia iterativa: iteración con contador

La cabeza establece el valor inicial, final y el incremento del contador o variable de control

Este tipo de iteración está presente en Fortan

DO <rotulo> <variable> = <expresion>, <expresion> [<instruccion>]*
<rotulo> <instrucción>

También en Pascal

For <variable> := <expresion> **to** | **downto** <expresion>
by <expresion> **do** <instrucción>

En **Pascal:** Las variables de iteración deben ser ordinales. Después del ciclo la variable de ciclo está indefinida La variable de ciclo no puede modificarse en el mismo.

En Algol

For <variable> := <expresion> **step** <expresion> **until** <expresion>
do <instrucción>

43

Sentencia iterativa: iteración con contador

Aspectos de diseño:

- ¿cuándo se evalúan las expresiones que controlan el bucle? ¿antes de comenzar el bucle o en cada iteración?
- ¿cuál es el tipo y el alcance de la variable de control?
- ¿cuál es el valor de la variable de control una vez que el bucle termina?
- ¿puede modificarse el valor de la variable de control dentro del bucle? ¿altera la cantidad de iteraciones?

44

Sentencia iterativa: iteración con condición

Aspectos de diseño:

- Mecanismo específico o embebido junto al condicional con contador
- Testeo de la condición (antes, después o ambos)

Iteración con condición en Pascal

while <expresion> **do** <instrucción>

repeat <instrucción> **until** <expresion>

for (i=1;i<=10;i++) {….}

for (i=1;i<=10 && notEndFile;i++) {….}

for (i=1;;i++) {….} (ejecución infinita)

Iteración con condición en C

for ([<expresion>]; [<expresion>]; [<expresion>]) {<instrucción>}

while (<expresion>) {<instrucción>}

45

Sentencia iterativa: iteración con condición

Iteración con condición en Java

for ([<expresion>]; [<expresion>]; [<expresion>]) {<instrucción>}

for (i=1;i<=10;i++) {….}

En C++

do {<instruccion>} **while** <expresion>

while <expresion> **do** {<instruccion>}

En Ada

for <variable> **in** [reverse] <Rango_discreto> [**while** <expresion>]
loop <instruccion> **end loop**

```
for k in index while A(k) /=0 loop
.....
end loop
```

```
loop
 .....
end loop
```

Cómputo infinito

En **Ada**: El rango:discreto es una subrango de los enteros o de un enumerado y el alcance de la variable de ciclo es el ciclo mismo

46

Sentencia iterativa: iteración contador-condición

En Eiffel

from <expresion>

invariant <expresion>

variant <variable>

until <condicion sobre la variable de control>

loop <instruccion>

end

Sentencia iterativa: iteración infinita

Ejemplo

while true **do** {}

i:=1; **while** i<10 **do** {i:= i-1}

47

Sentencia iterativa: iteración con múltiple salida

Aspectos de diseño:

- ¿Es posible salir de un único bucle o de varios anidados?
- ¿La salida se establece con o sin condición?
- ¿La salida es etiquetada?

En los lenguajes de programación:

- C y C++ salidas incondicionadas y sin etiqueta (**break**)
- Java, Pearl y C# salidas incondicionadas con etiqueta

```
outerLoop:
    for (row = 0; row < nF: row++)
        for (col = 0; col < nC: col++)
        { sum += .at[row][col];
          if sum > 1000 break outerLoop; }
```

Ejemplo en C#

48

Sentencia iterativa: iteración en base de datos

Aspectos a tener en cuenta:

- El número de elementos de la estructura de datos controla la cantidad de iteraciones del ciclo
- El mecanismo de control es una llamada a una función iterador que retorna el próximo elemento en algún orden establecido (si es que hay un próximo elemento, sino termina)
- Las intrucciones como el **foreach** de C# iteran sobre los elementos de arreglos y otras colecciones.

```
x.start();
While X.more() { … y = X.next()}
```

```
For (ptr = root; prt == null; traverse(ptr))
    {……}
```

Ejemplo en C++ y Java

```
Foreach $X (@arrayitem) {……}
```

En cada paso del ciclo, la variable escalar $X tendrá el próximo elemento del arreglo @arrayitem

Ejemplo en Pearl

Sentencia iterativa: iteración con guarda

- **Proposito**: soportar una nueva forma de programación que permita la verificación durante el desarrollo
- Resulta básico para dos mecanismos lingüisticos para la programación concurrente (Ada y CSP)
- La idea básica es que si el orden de evaluación no es importante, el programa no debería especificar uno

Forma del ciclo con guarda:

do <expresión> → <sentencia> [<expresión> → <sentencia>]* **od**

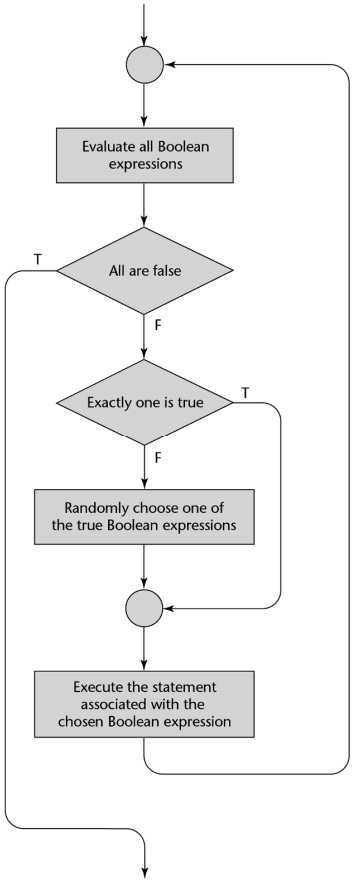
Semántica:

1. Evaluar la expresión (booleana en general)
2. Si mas de una es verdadera, elegir una en forma no determinista y comenzar el ciclo nuevamente
3. Si ninguna es verdadera, salir del bucle

Sentencia iterativa: iteración con guarda

Figure 8.2

Flowgraph of the approach used with Dijkstra's loop statement



Iteradores

En general los ciclos con contador, tipo *for*, iteran sobre una secuencia numérica.

Sin embargo es interesante poder hacerlo sobre los elementos de un conjunto cuyos elementos estén bien definidos (como las clases contenedoras o colecciones de los lenguajes orientados a objetos).

La idea del **iterador** es proveer esta funcionalidad.

Iteradores *"reales"* (true iterators)

En estos casos el iterador es una abstraccion que se parece a un unidad que contiene una sentencia *yield* que produce el cambio en el valor del ciclo

Algunos lenguajes que proveen true iterator son: Clu, Python, Ruby y C#

A diferencia de los iteradores de Java y C++ estos iteradores funcionan en base al contador del programa y las variables locales del entorno de ejecución.

Objetos Iteradores

Estos iteradores no utilizan la setencia *yield*.

Se trata de un objeto ordinario o común de la programación orientada a objetos, que provee métodos para inicializar, generar el próximo valor del índice, testear su terminación. Entre llamadas, el estado del iterador debe mantenerse.

Los iteradores de Java y C++ estos objetos necesitan mantener el estado interno en forma de estructuras de datos explicitas.

En Java el uso se ve como en el siguiente ciclo

```
for (Integer i: myTree) { .....}
```

El iterador y su funcionamiento no se ven en la sintaxis.

Iteradores: Ejemplo

Java

```
Class TreeNode<T> implements Iterable<T> {
        …..
        private class TreeIterator implements
       Iterable<T> {  …..
         }
         ….. }
 ……
 TreeNode<Integer> myTree = ….
 for (Integer i: myTree) { …..}

for (Iterator<Integer> it= myTree.iterator();
        it.hasNext();) { ……
 ….. }
```

Python

```
def counter(low, high):
   current = low
   while current <= high:
      yield current
      current += 1

for c in counter(3,8):
   print c
```

Recursión

Es un mecanismo de control que no requiere sintaxis adicional.

Aquellos lenguajes que proveen abstracciones procedurales y/o funcionales solo deben habilitar que las mismas tengan la posibilidad de llamarse a si mismas.

Esta manera de estructurar el control permite llegar en algunas oportunidades a códigos más elegantes.

No determinismo

Es un mecanismo en el cual la elección entre alternativas, queda deliberadamente sin especificar

Esta característica se ve en algunos constructores para concurrencia, como los comandos con guarda