Lenguajes de Programación I

Control de Flujo – Expresiones

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Nuestras máquinas son Von Neumann

. . . y no parece haber alternativas por ahora

- Sin importar el modelo de cómputo (imperativo o declarativo), es necesario indicar el **orden** en el cual ejecutar.
- Cualquier tarea algorítmica puede dividirse en sub-tareas – el **control de flujo** expresa esa división y el orden a seguir para llegar al resultado final deseado.

Los mecanismos de Flujo de Control pueden clasificarse según su intención.

Secuenciación

Esto primero, esto después. . .

- Indica el orden *específico* en el cual deben evaluarse expresiones o ejecutarse instrucciones.
- Usualmente el orden de aparición en el programa.
- Generalmente el "punto y coma" – en algunos lenguajes basta un salto de línea.

Selección

¿Esto, aquello o lo otro?

- Decidir el flujo de instrucciones entre dos o más alternativas.
- Decisión basada en alguna condición evaluada a tiempo de ejecución.
- Selección simple – `if-then-else`
- Selección múltiple – `case` o `switch`

Iteración

Enjuage y repita hasta el resultado deseado

- Repetir la ejecución de un fragmento de código.
- Iteración acotada – número *calculado* de repeticiones.
- Iteración no acotada – hasta que se cumpla alguna condición evaluada a tiempo de ejecución.
- La acotada es un caso particular de la no acotada – así son los constructores `for`, `while`, `repeat` modernos.

Abstracción Procedimental

Delegar y esconder la complejidad

- Encapsular una colección de instrucciones de control para poder aprovecharla como una unidad.
- Ponerle un nombre y parametrizarla posibilita su reutilización.
- Funciones, procedimientos o subrutinas.

Recursión

Hasta que entienda recursión, estudie recursión

- Definir un cómputo en términos más simples de sí mismo.
- Usualmente se expresa con funciones que hacen referencia a sí mismas directa o indirectamente.
- Tiene exactamente el mismo poder de cómputo que la iteración – algunos problemas son más fáciles de expresar recursivamente.

En algunos lenguajes **sólo** hay Recursión

Concurrencia

Montar bicicleta y mascar chicle al mismo tiempo

- Dos o más fragmentos de programa se ejecutan simultáneamente.
- Con paralelismo real – aprovechando varios procesadores.
- Con paralelismo simulado – compartiendo un sólo procesador.
- Tradicionalmente expresado de manera explícita (y dolorosa) – lenguajes modernos ofrecen construcciones implícitas.

Manejo de Excepciones y Especulación

"Como venga viniendo, vamos viendo"

- Un conjunto de instrucciones se ejecuta de manera "optimista" suponiendo que todo va a salir bien.
- Si *todo* sale bien, el flujo sigue adelante.
- Si *algo* sale mal en cualquier momento,
  - Manejo de Excepciones – se ejecuta un conjunto de instrucciones en lugar del *resto* del conjunto original.
  - Especulación – se ejecuta un conjunto de instrucciones en lugar de *todo* el conjunto original.

No determinismo

. . . porque ser ordenado es *mainstream*

- El orden de ejecución de un conjunto particular de instrucciones no se especifica deliberadamente.
- La escogencia ocurre a tiempo de ejecución y probabilísticamente.
- Aparece en conexión con lenguajes concurrentes o de flujo de datos.

Expresiones

El propósito del cómputo

- Una expresión puede ser:
  - Un objeto simple – nombre de variable o constante literal.
  - La aplicación de una función u **operador** a una colección de argumentos u **operandos**, cada uno de los cuales es a su vez una expresión.
- *Operadores* – funciones incluídas en el lenguaje, pero que tienen sintaxis especial simplificada intencionalmente.

Notación para la aplicación

Función en relación a sus argumentos

- Prefija – LISP, Scheme

```
(+ (* 4 foo) (* 0.5 (sin pi)))
```

- Infija – la mayoría de los lenguajes

```
4 * foo + 0.5 * sin(pi)
```

- Postfija – Forth, Postscript

```
4 foo * pi sin 0.5 * +
```

- Mezclada (*mixfix*) – Smalltalk

```
mail sendTo: "emhn@usb.ve" text: "Hola Mundo"
```

- Meros adornos (*syntactic sugar*) sobre funciones concretas – `a + b` en realidad es `a.operator+(b)` en C++.

Precedencia y Asociatividad

Necesarias para simplificar la implantación

- Notaciones prefija y postfija – *triviales* de implantar, difíciles para la mayoría de los programadores.
- Notación infija es ambigua para las expresiones aritméticas – dificultad intrínseca al lenguaje infijo que debe resolverse.
  - Explícitamente – paréntesis para indicar el orden de evaluación.
  - Implícitamente – reglas de precedencia y asociatividad para ahorrarse los paréntesis (siempre que se haya leído el manual).
- **Precedencia** – en ausencia de paréntesis, ¿cuál operador opera antes que los demás?
- **Asociatividad** – en ausencia de paréntesis, ¿cómo aplicar los operadores de una secuencia de igual precedencia?

Operadores Inusuales

Tomados de lenguajes imperativos estáticos

- Incremento y decremento pre y postfijos.
  - En vez de `a = a + 1`, escribir `a++`.
  - El efecto de borde siempre se efectúa – el valor cambia según la posición del operador.
  - Originalmente para favorecer la optimización a instrucciones especializadas de incremento y decremento en el procesador.
  - Confunde al desprevenido – ¿sabían que Python no tiene ++ por eso?
- Operador ternario `if-then-else`

```
a = (b > c) ? d : e
```

Operadores Inusuales

Tomados de lenguajes imperativos dinámicos

- Producto de enteros por colecciones (cadenas, listas, ...)

```
$a = $n x "a";  # $a tiene $n aes.
@b = (42) x 1;  # Una lista con 42 unos.
@b = (42) x @b; # Ahora son 42 42.
```

- Generadores de listas por enumeración

```
@a = -10..10;
```

- Comparador numérico generalizado – retorna -1, 0 o 1

```
$a <=> $b
```

Más operadores inusuales

... a gusto del programador

- Algunos lenguajes permiten definir nuevos operadores.
  - Asociados a una función particular definida por el programador.
  - Incorporados a la jerarquía de precedencia y asociatividad.
  - Integración limpia para cooperar con el resto del lenguaje.
- "Suma y producto de tuplas numéricas" – Haskell

```
infixl 5 :+:
infixl 7 :*:
(:+:), (:*:) :: (Num a,Num b) =>
                (a,b) -> (a,b) -> (a,b)
(a1,b1) :+: (a2,b2) = (a1+a2,b1+b2)
(a1,b1) :*: (a2,b2) = (a1*a2,b1*b2)
```

y ahora se puede escribir algo como

```
(21,0) :+: (3,2) :*: (7,21)
```

Esto puede llevarse al extremo

- IBM "engendró" APL – lenguaje puramente funcional.
- Lleno de operadores funcionales aplicativos . . .
- Generar la lista de primos de 1 hasta `N`

$$(\sim N \in N \circ . \times N)/N \leftarrow 1 \downarrow \iota N$$

- . . . pero requiriendo un teclado especial

It's dead, Jim!

Asignación

- **Efecto de Borde** (*side effects*) – cuando una instrucción influye en los cómputos que le siguen más allá de producir un valor simple.
- **Transparencia Referencial** – cuando no hay efectos de borde en ninguna parte del lenguaje (como en los lenguajes funcionales).
  - Asignar a un símbolo sólo lo hace disponible para la evaluación actual.
  - Sólo hay valores que producen valores por aplicación funcional.
- En los lenguajes imperativos, los efectos de borde son la norma – modificar el estado es lo que produce resultados.
  - Asignar a un símbolo lo hace disponible durante la evaluación actual y posiblemente más allá de ella.
  - Existen valores y *referencias* a valores.

Valores vs. Referencias

La dualidad existencial de las variables

- En los lenguajes imperativos los nombres de variable denotan "contenedores" para valores – ubicaciones en el estado mutable.
- Una asignación tiene lados derecho e izquierdo con roles distintos:
  - El derecho debe producir un **valor** – qué almacenar.
  - El izquierdo debe referir una **ubicación** – dónde almacenarlo.
- Puede haber expresiones en *ambos* lados de la asignación
  - Si denotan valores se denominan *r-values* – sólo pueden estar en el lado derecho pues interesa el valor.
  - Si denotan ubicaciones se denominan *l-values* – sólo pueden estar en el lado izquierdo pues interesa la ubicación.

l-values vs. *r-values*

- Algunas expresiones nunca pueden ser *l-values*
  - `2 + 3 = a`, no tiene sentido.
  - `a = 2 + 3`, tampoco, cuando `a` es una constante.
- Un *l-value* no tiene que ser siempre un nombre ni tampoco simple
  - Aprovechando apuntadores

```
(f(a)+3)->b[c] = 2
```

  - El lenguaje permite **funciones *l-value***

```
substr($a,5,4) = "hola mundo!"
```

- Un objeto puede actuar como *r-value* o *l-value* en una misma expresión – depende cuál atributo de la asociación interesa.

Modelo de acceso para variables

- **Modelo de Valor** –
  cada variable se considera *r-value* o *l-value* según aplique.
- **Modelo de Referencia** –
  toda variable es considerada un *l-value*.
  - Necesario "seguir la referencia" (*dereferencing*) en ciertos casos.
  - Automático en la mayoría de lenguajes con ese modelo.
  - Produce el inconveniente encajonado (*boxing*) en algunos lenguajes.

Ortogonalidad

- **Lenguaje Ortogonal**
  - Sus características pueden usarse en *cualquier* combinación.
  - *Cualquier* combinación tiene sentido y significado consistente.
- Ortogonalidad total – *todo* el lenguaje (muy raro).
- Ortogonalidad parcial – en algunos elementos (expresiones, tipos, . . .)

Ortogonalidad y Expresiones

Combinar efecto y resultado

- Propuesto originalmente en Algol.
- No hay diferencia entre **expresión** e **instrucción** –
  toda instrucción produce un valor válido como expresión.

```
foo := begin
   a := if b < c then d else e;
   a := begin f(b); g(c) end;
   g(d);
   14 * if true then 3 else 5;
 end;
```

- Díficil comprender el código, especialmente ante efectos de borde –
  ¿y si `f(b)` o `g(c)` mutan estado?
- Sistema de Tipos más complicado – compiladores más complejos.

Ortogonalidad y Expresiones

Combinar efecto y resultado. . . con cuidado

- Separación clara entre expresiones e instrucciones.
- Proveer algunas **instrucciones con valor** (*expression statements*)

```
a = (b < c) ? d : e;
```

Eso incluye la asignación

Muta estado, pero tiene un valor

- La asignación como una expresión – vale lo que asigna.

```
a = b = 1;

$a = ($b = 5) * ($c = f($d) + 9);
```

- Combinar un operador con una asignación.

```
a += 5;
b[8].a->b *= 2;

$str = "foo";
$str .= " bar";
```

- El programador debe ser cuidadoso para comprender el efecto.
- Compilador debe lidiar con el código ensamblable redundante.

. . . y aún así pueden sorprender al desprevenido

- `++foo` incrementa el operando *antes* de producir su valor.
  - Es equivalente a `foo = foo + 1`.
  - Puede aprovecharse `INC` del procesador si lo tiene.
- `foo++` entrega su valor *antes* de incrementar.
  - No es equivalente a `foo = foo + 1` – debe entregar el valor del *r-value* **antes** de modificar el *l-value*.
  - De hecho. . .

    `(t = p; p = p + 1; t)`

    . . . hace falta una variable temporal y *tres* instrucciones.

Aún más complejo cuando se usan para manipular arreglos y apuntadores.

Asignación múltiple

- Asignación múltiple

```
($a,$b) = (42, "hola");
($b,$a) = ($a,$b) # Look ma! No temp
```

- Ortogonalidad en llamadas a funciones

```
($a,$b,$c) = foo(...);
```

  - Tantos argumentos como sea necesario.
  - Tantos resultados como convenga – ¡ortogonalidad en la expresión de funciones y procedimientos!
  - Difícil de implantar en lenguajes compilados.

Inicialización de Variables

- Proveer un valor inicial al momento de la declaración reduce la posibilidad de errores difíciles de identificar.
- Para variables con almacenamiento estático
  - Establecer el valor inicial a tiempo de compilación.
  - Se ahorra tiempo de ejecución.
- Reglas de inicialización automática.
  - Incluir la inicialización con la declaración.
  - Valores por omisión según el tipo primitivo – cero, `null`, etc.
- Asignar manualmente los valores.
- **Constructores** a ser invocados durante la inicialización.

¿Y si olvidamos inicializar?

Claro...

- Las reglas automáticas de inicialización aplican – mínima protección que puede aprovecharse si se conoce.
- El lenguaje podría detectar uso de variables sin inicializar a tiempo de ejecución y emitir un error semántico dinámico.
  - Aprovechar representaciones especiales (*NaN* para punto flotante) y habilidades del procesador para generar excepciones.
  - Enriquecer la representación de datos con *flags* de inicialización.
  - Muy caro para lenguajes compilados.
- **Asignación Definitiva** – una verificación estática conservadora.
  - Analizar todos los caminos de control *hasta* una expresión particular.
  - Verificar que los *r-values* de la expresión han sido asignados efectivamente en *todos* los caminos.
  - Repetir para *todas* las expresiones en el bloque – se aplica en alcances locales únicamente.
  - El problema no es decidible – produce "falsos-positivos"

¿Cuándo se evalúan los operandos?

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

- Puede influir (bugs) si hay efectos colaterales.
- Permite mejorar el código objeto.
  - Asignar registros y agendar instrucciones.
  - La mayoría de los lenguajes no define orden de evaluación para poder generar el mejor código en cada plataforma.
- Algunas excepciones:
  - Java y C# insisten en evaluar de izquierda a derecha.
  - C/C++ evalúan argumentos para funciones de derecha a izquierda.
  - Reordenar expresiones según las propiedades matemáticas.
- Reordenar, pero respetar paréntesis.

Corto circuito – Evaluación de McCarthy

Mínimo trabajo necesario para obtener el resultado

- Evitar la evaluación completa de expresiones booleanas.
  - Secuencia de `or` es cierta tan pronto una sea cierta.
  - Secuencia de `and` es falsa tan pronto una sea falsa.
  - Compilador puede generar *cascading code* para minimizar saltos – se estudia en Lenguajes de Programación III
- Algunos lenguajes no ofrecen evaluación con cortocircuito – Pascal.
- Algunos lenguajes ofrecen ambas posibilidades – Erlang
  - Operadores diferenciados.
  - `and` y `or` – **no** usan cortocircuito.
  - `andalso` y `orelse` – **si** usan cortocircuito.
- Al emplear corto-circuito, dejan de ser funciones estrictas
  - No evalúan *ambos* argumentos, sólo los que van siendo necesarios.
  - Dejan de ser *expresiones* para volverse *instrucciones*.

Bibliografía

- [*Scott*]
  - Sección 6.1
  - Ejercicios y Exploraciones