Lenguajes de Programación I

Alcance *(Scope)*

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

¿Qué quiere decir Alcance?

Dos significados diferentes según el contexto

- Para una asociación particular, su **alcance** es la *región* de un programa en la cual está activa y visible.
- Si consideramos más de una asociación, un **alcance** es una región de un programa de tamaño *máximo* en la cual no hay cambios en las asociaciones – típicamente un *bloque*
  - Al ingresar – se *elaboran* las asociaciones que deben activarse.
  - Al salir – se *destruyen* las asociaciones que deben desactivarse.
- **Ambiente de Referencia** – conjunto de asociaciones activas en un punto dado del programa.
  - Calculado usando reglas de **Alcance Estático** o **Dinámico**.
  - Mantenido usando las técnicas de almacenamiento global o local que estudiamos en la clase pasada.

Alcance Estático

Basado en el texto del programa fuente

- También llamado *Alcance Lexicográfico*.
- Calculado a tiempo de compilación.
- Independiente del flujo de ejecución.
- Variedades simples
  - Sólo variables globales (BASIC).
  - Globales y locales sin anidamiento (Fortran *antes* de 1990).
- Variedades completas – cualquier lenguaje moderno.

Basta mirar el texto del programa,
y hacer **sustitución de variables**

Alcance Estático y Anidamiento

- De subrutinas, de bloques, de módulos, de clases...
- Regla del **Bloque de Alcance más Cercano**
  - Todo nombre declarado en un alcance es visible en ese bloque y en cualquier bloque anidado dentro de él...
  - ...a menos que quede *escondida* por una declaración para el mismo nombre – un *hueco* en el alcance.
- Para determinar la asociación de un símbolo `foo`, buscamos desde el bloque que le contiene hacia los bloques exteriores hasta encontrar la declaración más cercana – nada nuevo...
- ¿Y los elementos *predefinidos* (*built-in*) del lenguaje?

¿Qué puede verse y dónde?

- Globales – dentro de todo el módulo o programa.
- Locales – dentro del procedimiento o bloque que las contiene.
- De "afuera hacia adentro", hay visibilidad.
  - Siempre y cuando haya anidamiento sin redefinición.
  - Si hay redefinición, se oculta parcialmente el alcance externo.
  - En algunos lenguajes es posible tener acceso a la asociación oculta usando construcciones sintácticas a propósito (Ada, C++).
- De "adentro hacia afuera", nunca hay visibilidad.

¿Cómo se implementa el alcance estático?

Con mucho cuidado...

- Compilador construye una **Tabla de Símbolos**
  - Diccionario con información para cada símbolo conocido – nombre, tipo, ocurrencia (línea, columna, etc.)...
  - Símbolo global – dirección absoluta asignada.
  - Símbolo local – *offset* dentro del registro de activación.
- Arbol de tablas, con apuntador al entorno actual, para anidamiento.
  - API sencillo para insertar, buscar y modificar símbolos.
  - Tabla particular para cada procedimiento y cada registro/clase.
    - `enterScope` al entrar – empilar tabla y apuntarle.
    - `exitScope` al salir – desempilar tabla y apuntar al entorno.
- Tabla (y API) incluida en el ejecutable para *debugging* o *profiling*.

Modelo LeBlanc-Cook es el más eficiente,
se estudia (e implanta) en Lenguajes de Programación II

Acceso a Objetos No-Locales

Cuando no hay anidamiento de procedimientos

- Solamente hay procedimientos en el nivel superior – un procedimiento no contiene procedimientos.
- El símbolo `foo` que aparece en el cuerpo de un procedimiento será:
  - Una variable local del procedimiento.
  - Una variable global al programa.
- Basta la información de la Tabla de Símbolos para emitir código
  - Acceder directamente a la dirección absoluta.
  - Acceder directamente a la posición en el registro de activación

Es rápido y práctico – por eso es la costumbre.

Acceso a Objetos No-Locales

Cuando hay anidamiento de procedimientos

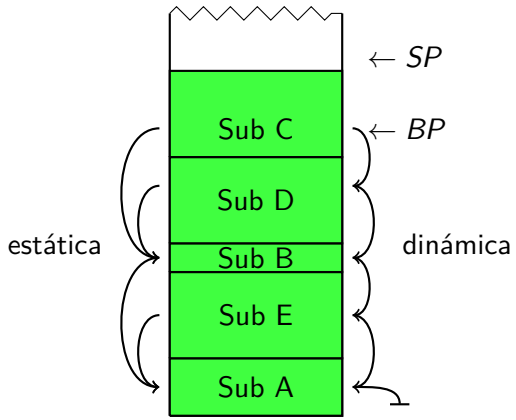
- Si el símbolo `foo` es global o local al procedimiento actual, se reduce al escenario anterior – ¿qué ocurre si se trata de un símbolo en alguno de los bloques que *envuelve* al procedimiento?
- Mantener un enlace estático (*static link*) al registro de activación para la invocación más reciente de la rutina que contiene al registro de activación actual – el *padre* léxico a tiempo de ejecución.
  - k rutinas anidadas – cadena estática de longitud k.
  - Alcanzar una variable/parámetro declarado j alcances hacia afuera – recorrer j pasos por la cadena estática y luego usar el *offset*.
  - El nivel de anidamiento es un atributo adicional de cada símbolo, mantenido por el compilador en la tabla de símbolos.

Estudiaremos todos los detalles más adelante.

Registro de activación *reloaded*

El rastro de la ejecución y el acceso no-local

```
proc A {
  proc B {
    proc C {
    }
    proc D {
      C();
    }
    D()
  }
  proc E {
    B();
  }
  E();
}
```



Módulos

Cuando el alcance rompe la frontera del archivo

- Búsqueda de abstracción *escondiendo información*.
- Reducir la contaminación del espacio de nombres de los programas.
- Los nombres dentro de un módulo:
  - Son visibles entre sí (reglas de alcance por medio).
  - Visibles desde el exterior cuando son **exportados**.
  - Ven símbolos exteriores cuando han sido **importados**.
- C++ los llama *namespaces*, Modula los llama *modules*, Perl los llama *packages*, Clu los llama *clusters*... lo importante es el efecto.

¿Son gestores o son tipos?

Precursores de la Programación Orientada a Objetos

- **Módulo como Gestor o Librería**
  - Cada módulo gestiona un tipo de datos – Clu.
  - Funciones para "preparar" variables del tipo opaco – el nombre del módulo sirve para solicitar un valor.
  - Funciones que reciben variables explícitamente.
- **Módulo como Tipo**
  - Cada módulo representa un tipo de datos – Simula.
  - Tipo opaco construible de manera directa – el nombre del módulo se convierte en un tipo.
  - Funciones "pertenecen" a cada entidad del tipo.

¿Son gestores o son tipos?

Módulo como Gestor

```
import Stack;
var A, B : Stack;
var x, y : Element;
⋮
initStack(A);
initStack(B);
⋮
push(A,x);
⋮
b := pop(B);
```

Módulo como Tipo

```
import Stack;
var A, B : Stack;
var x, y : Element;

⋮
A.push(x);
⋮
b := B.pop;
```

¿Qué hace falta para tener módulos?

- Especificar el inicio del módulo – apertura del *espacio de nombres*.
- Indicar nombres que serán exportados y cómo:
  - Sólo lectura.
  - Opacamente (útil para tipos de datos).
- Separar declaraciones (*module head*) de implantación (*module body*).
- Utilizar otros módulos importando nombres
  - *Alcance Cerrado* (*Closed Scope*) – importar explícitamente.
  - *Alcance Abierto* (*Open Scope*) – importar implícitamente.
  - En general es más útil el *alcance abierto selectivo* – se importan todos, y se indican las *excepciones*.
- Compilador mantendrá tablas de símbolos por módulo – seguramente requerirá compilación separada.

Alcance Dinámico

Basado en el flujo de ejecución del programa

- Calculado a *tiempo de ejecución*.
- Depende del orden de invocación de las subrutinas.
- La asociación *actual* para un nombre particular es la encontrada más recientemente durante la *ejecución* y que aún no haya sido destruida.
- El flujo de ejecución es impredecible – las asociaciones no pueden determinarse a tiempo de compilación.
- Diferir verificaciones semánticas a tiempo de ejecución.

¿Cómo se implementa el alcance dinámico?

Con mucho cuidado...

- Se calcula durante la ejecución del programa.
  - El compilador genera código apropiado para el mecanismo.
  - El interpretador incorpora el mecanismo.
- Mecanismo de **Listas de Asociación** (*A-list*)
  - Lista ordenada de parejas símbolo/valor.
  - Se usan como pilas – `enterScope` y `exitScope` son equivalentes a `push` y `pop` de la pareja.
  - Acceder a un símbolo – buscar su primera ocurrencia para recuperar o modificar la información asociada.
- Mecanismo de **Tablas Centrales de Referencia**
  - Diccionario de símbolos a lista de valores.
  - La lista de valores contiene el más reciente al principio.
- Usar la cadena dinámica si el lenguaje provee alcance estático y dinámico simultáneamente – lento, pero no hay otra forma.

Diferencia muy sutil

¿Qué imprime el programa?

```
a : integer;
procedure primera
  a := 1
end
procedure segunda
  a : integer
  primera()
end
a := 2
if read.integer() > 0
  segunda()
else
  primera()
print(a)
```

Alcance Estático

- Siempre imprime 1.
- La `a` en `primera`, siempre hace referencia a la global.

Alcance Dinámico

- Puede imprimir 1 o 2.
- Cuando `read < 0`, la `a` en `primera`, hace referencia a la global.
- Cuando `read > 0`, la `a` en `primera`, hace referencia a la `a` en `segunda`.

¿Y cómo puede ser útil?

- Simplifica la adaptación de rutinas en algunos casos: en lugar de pasar argumentos, las rutinas pueden usar variables que sean controladas por alcance dinámico.
- Ese problema se puede resolver sin usar alcance dinámico:
  - Usando polimorfismo con parámetros por defecto.
  - Usando varias rutinas con parámetros diferentes.
  - Se puede usar una variable global/local estática, salvarla, poner el valor nuevo, llamar a la rutina y restaurarla después.

Lucía como una buena idea y era fácil de implantar,
pero hoy en día es una curiosidad.

Bibliografía

- [*Scott*]
  - Secciones 3.3 y 3.4.
  - Ejercicios y Exploraciones
- Página de Wikipedia sobre Alcance (*Scope*)