UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

Facultad de Ciencias

Lenguajes de Programación

Alcance estático y dinámico en lenguajes mixtos

Proyecto 2

Presenta:
Lugo Díaz Ordaz Gretel Alexandra
Ramírez Juárez María Fernanda
Rojo Peña Manuel Ianluck

Profesor:
Manuel Soto Romero

Grupo: 7121, 2026-1

Fecha de entrega:
7 de diciembre, 2025

Índice

Pregunta: ¿Por qué algunos lenguajes modernos (como Lisp, Python o JavaScript) adoptan reglas mixtas de alcance y cómo afecta eso al razonamiento sobre el código? Búsqueda: lexical scope vs dynamic scope in Lisp, Python scope chain, closures in JavaScript Programa: Implementar un mini-lenguaje con dos modos de alcance seleccionables, ejecutando un mismo programa bajo ambos y comparando los resultados.

0.1. Introducción

En el diseño de lenguajes de programación existe una distinción conceptual entre **alcance léxico** y **alcance dinámico**,el cual determina cómo se resuelven las referencias a variables y, por ende, cómo se razona sobre el comportamiento de un programa. Aunque el alcance léxico se ha consolidado como el modelo dominante, diversos lenguajes de programación (Lisp en algunas de sus variantes, Python en la organización de sus namespaces y JavaScript en su manejo de closures) incorporan **reglas mixtas** que combinan aspectos de ambos modelos. Esto complica razonar sobre el comportamiento del programa, especialmente en presencia de funciones anónimas, anidamiento o resolución diferida.

La importancia de este tema radica en que las reglas de alcance no son simplemente un detalle técnico, sino un pilar fundamental en la semántica de los lenguajes. Scott señala que las decisiones sobre alcance influyen directamente en la claridad del modelo mental que ofrece un lenguaje y en la facilidad con la que un programador puede predecir la evolución de un programa [**?**]. Asimismo, desde la perspectiva de la expresividad, el tipo de alcance determina qué construcciones pueden representarse de manera directa y cuáles requieren transformaciones adicionales, como argumenta Felleisen en su estudio sobre el poder expresivo de los lenguajes [**?**]. Por lo tanto, comprender por qué ciertos lenguajes adoptan reglas mixtas permite analizar de forma más crítica sus capacidades, limitaciones y la clase de razonamiento que fomentan.

El objetivo de este trabajo es examinar las razones por las que lenguajes modernos emplean esquemas mixtos de alcance y evaluar cómo estas elecciones influyen en la legibilidad y razonamiento del código, particularmente en contexto de *cadenas de ambientes, clousures y resolución diferida*. La metodología combina una revisión teórica de la literatura clásica sobre semántica de lenguajes con la implementación práctica de agregar que vamos hacer de implementación , con el fin de observar de manera concreta el comportamiento derivado de sus reglas de alcance. Esta doble aproximación permite relacionar los fundamentos conceptuales con las consecuencias prácticas para el desarrollo de software y la comprensión de programas.

0.2. Fundamentos teóricos

El concepto de **alcance** es uno de los pilares de la semántica de los lenguajes de programación, pues determina el contexto en el que un identificador puede ser utilizado y cómo se vincula con su valor. Las distintas estrategias de alcance han sido estudiadas y formalizadas extensamente en la literatura, dada su influencia directa en el razonamiento sobre programas, la implementación de ambientes y la expresividad del lenguaje.

Alcance léxico (estático)

El alcance léxico establece que la asociación entre un nombre y su valor se determina según la estructura textual del programa. Es decir, para que un identificador se resuelva se busca en el bloque o función donde se definió, independientemente desde donde se invoque. Scott formaliza este modelo como un sistema donde las regiones textuales del código inducen un conjunto de ambientes anidados que el compilador puede determinar en tiempo de análisis [?].

Sus características principales son:

- **Razonamiento local**: La visibilidad de las variables depende de la estructura del programa, un programador puede identificar cuales variables son visibles sin la necesidad de rastrear toda la ruta de ejecución.
- **Soporte para closures**: Un closure captura el ambiente léxico en el que fue definido. Esto permite que las funciones sigan accediendo a variables externas aunque su bloque original haya terminado. Este es el mecanismo fundamental que vuelve posible la programación funcional basada en funciones de orden superior [?].
- **Optimizaciones**: Como el compilador conoce la estructura de los ambientes de antemano, puede mejorar el rendimiento (usando el stack eficientemente o eliminando variables inútiles) antes de correr el programa.

Alcance dinámico

El **alcance dinámico** busca el valor de un identificador basándose en la cadena de llamadas activa durante la ejecución. Lo que significa que un mismo nombre puede referirse a cosas distintas según quién llamó a la función. Scott señala que este modelo fue común en las primeras versiones de Lisp debido a su simplicidad de implementación y a que facilitaba ciertas formas de extensibilidad [?].

Sus características principales son:

- **Dependencia de la ejecución**: El valor de una variable cambia según quién llame a la función. Esto da mucha flexibilidad, pero hace difícil predecir qué hará el programa sin ejecutarlo.
- **Uso histórico y actual**: El alcance dinámico fue ideal para los primeros lenguajes interpretados porque era muy fácil de implementar donde la simplicidad del intérprete

tenía prioridad sobre la robustez del análisis estático. Hoy en día se mantiene en contextos específicos (como sistemas de macros o configuraciones), donde es útil modificar el comportamiento global sin tener que pasar parámetros explícitamente a cada función.

Modelos mixtos

Ante las limitaciones de ambos enfoques, algunos lenguajes usan un enfoque mixto, combinando estructuras léxicas claras con mecanismos dinámicos que permiten . Abadi señalan que los sistemas mixtos pueden surgir incluso en lenguajes estáticos cuando adoptan características flexibles (como la carga tardía), obligando a unir lo que se sabe al compilar con lo que sucede en tiempo de ejecución [?].

Python basa su alcance en la estructura léxica, pero mantiene reglas dinámicas para lo global y lo predefinido. JavaScript sigue un patrón parecido al mezclar closures con el ámbito global. De igual forma, ciertos Lisp permiten usar variables especiales dinámicas junto a las variables léxicas.

Conceptos clave

- **Closures**: Son funciones que capturan su ambiente léxico en el momento de su definición. Friedman y Wand describen los closures como pares que combinan código y ambiente, permitiendo diferir la ejecución sin perder contexto [?].
- **Ambientes**: Estructuras de mapeo entre identificadores y valores. La implementación eficiente de ambientes, especialmente en presencia de funciones recursivas y llamadas en cola, ha sido estudiada por Clinger, quien analiza la relación entre recursión adecuada (proper tail recursion) y uso constante de espacio [?].
- **Cadenas de alcance**: Secuencias de ambientes que representan los posibles lugares donde se puede resolver una variable. En los lenguajes con alcance léxico, esta cadena se determina estáticamente, en los lenguajes dinámicos, se reconstruye a partir de la pila de llamadas.
- **Contextos de ejecución**: Son marcos creados al invocar funciones o bloques, que almacenan variables locales, parámetros y referencias a ambientes externos.

0.3. Análisis de lenguajes modernos

Cada lenguaje mezcla los alcances según sus propios objetivos. En la historia de Lisp y Scheme, esto se ve claramente en la diferencia entre **let** (que crea un ambiente léxico fijo) y **let-dynamic** (que sigue el flujo de ejecución). Common Lisp estandarizó esto con las llamadas "variables especiales", que actúan de forma dinámica dentro de un sistema que es mayormente léxico [?]. Según Kiczales, estas variables son muy fuertes ya que permiten cambiar el comportamiento global sin tener que pasar parámetros extra, lo cual es la base de sistemas avanzados como los metaobjetos [?].

Python organiza su alcance con la regla **LEGB**: Local, Enclosing, Global y Built-in.Este esquema es fundamentalmente léxico, pero incorpora mecanismos que permiten ajustar la

resolución de nombres desde funciones anidadas: **nonlocal** permite modificar variables en un ambiente externo no global, mientras que **global** habilita cambios sobre nombres del módulo, introduciendo así flexibilidad dinámica en la manipulación de ámbitos [?]. Estas reglas proporcionan un equilibrio entre claridad léxica y flexibilidad, siempre y cuando se preste atención al modificar variables dentro de funciones anidadas.

Por su parte, **JavaScript** combina closures léxicos con un sistema de contextos de ejecución bien definidos. La introducción de **let** y **const** sustituyó gradualmente a var para evitar la confusión que causaba su hoisting (elevación de variables) ya que podía generar ambigüedades semánticas [?]. Además, el uso histórico de la sentencia with que intentaba alterar dinámicamente pero solo lograba hacer el código ilegible. Tal como señala Bracha, estas variaciones reflejan una transición desde modelos más dinámicos hacia diseños con mayor claridad estructural [?].

Pese a sus diferencias sintácticas, estos lenguajes comparten patrones comunes. Todos parten de una base léxica pero incluyen excepciones dinámicas (palabras clave o estructuras del contexto global) que introducen elementos dinámicos. Esto crea modelos híbridos donde la claridad del alcance léxico coexiste con la necesidad de flexibilidad operativa en tiempo de ejecución.

0.4. Ventajas de desventajas de los modelos mixtos

Los modelos mixtos de alcance combinan estructuras léxicas con mecanismos dinámicos, esta mezcla ofrece beneficios notables en lenguajes orientados a la extensibilidad y a la programación funcional.

- **Flexibilidad**: Permiten introducir parámetros implícitos, ajustar comportamientos globales o representar configuraciones contextuales sin modificar explícitamente todas las funciones involucradas. Esto es útil en sistemas que requieren manejar el contexto de ejecución sin romper la estructura modular del programa.

- **Compatibilidad**: Facilitan la evolución del lenguaje, permitiendo que características antiguas (como variables especiales o comportamientos heredados) convivan con mecanismos léxicos modernos.

- **Control**: En el ámbito funcional, la coexistencia de closures con ambientes dinámicos ofrece un control muy preciso del estado. Como señalan Abelson y Sussman, esto es clave para mantener la modularidad en diseños complejos [?].

Sin embargo, estos beneficios vienen acompañados de desventajas:

- **Complejidad**:El programador debe pensar en dos lógicas a la vez (estática y dinámica), lo que dificulta entender el código.

- **Pérdida de seguridad**:Clarke y Drossopoulou advierten que, al permitir que un nombre se resuelva en tantos lugares, se rompe el encapsulamiento y se debilita el sistema [?].

- **Depuración difícil**:Es complicado rastrear errores cuando el valor de una variable depende de una cadena de llamadas compleja.
- **Mantenimiento**: Según Erdweg, en proyectos grandes esto reduce la predictibilidad y hace difícil actualizar el software, especialmente si se usa metaprogramación [**?**].
- **Rendimiento**:El sistema se vuelve más lento porque debe gestionar estructuras para ambos tipos de alcance.

0.5. Implementación práctica

El propósito de esta sección es ilustrar de manera mas concreta el cómo se presentan las reglas de alcance léxico y dinámico, usando como base nuestra implementación de **MINILISP** (realzada como proyecto 1 de la asignatura de lenguajes de programación).

Arquitectura del sistema

Implementamos un intérprete en Haskell para **MINILISP** funcional que soporta evaluación con alcance léxico(estatico) como dinámico.Esta arquitectura tiene un diseño que transforma el programa fuente gradualmente hata llegar al esultado final.

Cada módulo transforma los datos de entrada de la siguiente manera:

- **Entrada**: "(let (x 10) (+ x 5))"
- **Lexer (Alex)**: [TokenPA, TokenLet, TokenPA, TokenVar "x", ...]
- **Parser (Happy)**: Let "x"(Num 10) (Add [Var "x", Num 5])
- **Desugar**: AppV (FunV "x"(AddV (VarV "x") (NumV 5))) (NumV 10)
- **Evaluación (Static/Dynamic)**: NumV 15

Componentes clave del sistema

0.5.1. Sistema Léxico: Alex

En este parte de la implementación su función es la de convertir el texto plano en tokens.

```
tokens :-
$white+                         ;  -- Ignorar espacios
\(                              { \_ -> TokenPA }
\)                              { \_ -> TokenPC }
"lambda"                        { \_ -> TokenLambda }
"-"?$digit+                     { \s -> TokenNum (read s) }
$alpha($alnum)*                 { \s -> TokenVar s }
```

Código 1: Estructura del lexer con Alex

0.5.2. Sistema sintáctod: Parser

El parser se encarga de transformar los tokens en árbol sintáctico usando la herramienta **Happy**. Gramatica:

```
ASA : '(' "let" '(' var ASA ')' ASA ')'  { Let $4 $5 $7 }
    | '(' "lambda" '(' vars ')' ASA ')'  { Lambda (reverse $4) $6 }
    | '(' ASA appArgs ')'                { App $2 (reverse $3) }
```

Código 2: Gramática del parser (Happy)

Estructura de salida (ASA.hs):

```

data ASA = Var String           --Variables
         | Num Int              --Numeros
         | Let String ASA ASA   --Binding local
         | Lambda [String] ASA  --Funciones
         | App ASA [ASA]        --Aplicacion
         | Add [ASA]            --Operadores n-arios
         deriving (Show, Eq)
```

Código 3: Estructura del AST (ASA.hs)

0.5.3. Desugar

La responsabilidad es de convertir el AST .ªzucarado.ª su forma núcleo. Transformaciones (Desugar.hs):

```
--1. Let aplicacion de funciones
desugar (Let i v b) = AppV (FunV i (desugar b)) (desugar v)

--2. Lambda multi-parametro y se currifica
desugar (Lambda ps b) = desugarLmb ps b

--3. Operadores n-arios a binarios
desugar (Add xs) = desugarOps AddV xs
```

Código 4: Transformaciones del desazucarador (Desugar.hs)

Forma nucleo:

```
data ASV = VarV String             -- Variables
         | NumV Int                -- Valores numericos
         | BoolV Bool              -- Valores booleanos
         | FunV String ASV         -- Funciones
         | AppV ASV ASV            -- Aplicaciones
         | Closure String ASV Env  -- Clousures
         | Operadores binarios (AddV, SubV, etc.)
         deriving (Show, Eq)
```

Código 5: Estructura del núcleo (ASV.hs)

0.5.4. Evaluador Léxico (estático)

Esta parte se encarga de evaluar los programas con alcance léxico (estático). Mecanismo (StaticScope.hs)

```haskell
-- 1. Creacion de clausures (CAPTURA el ambiente)
pasitoLex (FunV p c) env = (Closure p c env, env)

-- 2. Aplicacion con ambiente capturado
pasitoLex (AppV (Closure p c e) a) env
  | isValue a = (c, (p, a) : e)

-- 3. Busqueda lexica
pasitoLex (VarV i) env = (lookupEnv i env, env)
```

0.5.5. Evaluador Dinámico

La responsabilidad es la de evaluar programas con alcance dinámico. Mecanismo (DynamicScope.hs)

```haskell
-- 1. Funciones sin captura
pasitoDyn (FunV p c) env = (FunV p c, env)

-- 2. Aplicacion con ambiente actual
pasitoDyn (AppV (FunV p c) a) env
  | isValue a = (c, (p, a) : env)

-- 3. Busqueda dinamica
pasitoDyn (VarV i) env = (lookupEnv i env, env)
```

0.5.6. Interfaz principal

Es el coordinador del sistema. Main.hs

```haskell
evalMode :: Bool -> Bool -> String -> IO ()
evalMode static dynamic expr = do
  let tokens = lexer expr
  let asa    = parse tokens
  let asv    = desugar asa

  if static
    then let res = evalStatic asv []
         in putStrLn $ "[Estatico]: " ++ saca res
    else return ()

  if dynamic
    then let res = evalDynamic asv []
         in putStrLn $ "[Dinamico]: " ++ saca res
    else return ()
```

0.6. Casos de Estudio

0.6.1. Closures en Alcance Léxico: El Caso de JavaScript

Una **closure** es una función que conserva acceso a variables de su ámbito externo, incluso después de que dicho ámbito haya terminado su ejecución. Este mecanismo permite que la función recuerde.[el] ambiente en el cual fue creada, manteniendo acceso a variables que de otra manera estarían fuera de alcance.

```
function outer() {
    let mensaje = "Hola";
    function inner() {
        console.log(mensaje);
        mensaje = "Adios";
    }
    return inner;
}

const closure = outer();
closure();  // Imprime: "Hola"
closure();  // Imprime: "Adios"
```

¿Cómo funciona?

- **Creación del ambiente**: Cuando **outer()** se ejecuta, crea un ambiente local con **mensaje = "Hola"**.
- **Captura léxica**: **inner** se define dentro de este ambiente y, debido al alcance léxico de JavaScript, captura una referencia al ambiente completo.
- **Persistencia**: Aunque **outer()** termina su ejecución, el ambiente persiste porque **inner** mantiene una referencia activa a través de la clausura.
- **Mutación compartida**: La segunda llamada a **closure()** accede al mismo ambiente capturado, donde **mensaje** ya fue modificado a .[A]diós".

Este mecanismo de closures permite implementar variables privadas y estado encapsulado, características fundamentales para la programación modular y orientada a objetos.

0.6.2. Alcance Dinámico: El Caso de Bash

En contraste con el alcance léxico, el alcance dinámico resuelve las variables basándose en el contexto de ejecución actual, no en el ámbito de definición. Bash es uno de los pocos lenguajes modernos que preserva este paradigma.

```
#!/bin/bash

var="global"

function foo() {
    echo "var: $var"
}
```

```

function bar() {
    local var="local"
    foo
}

bar  # Imprime: "var: local"
```

- **Creación del ambiente**: Cuando **outer()** se ejecuta, crea un ambiente local con **mensaje** = **"Hola"**.
- **Ambiente inicial**: Antes de llamar a **bar**, el ambiente contiene **var** = **"global"**.
- **Modificación dinámica**: **bar** crea una variable **local var** = **"local"**.
- **Resolución en tiempo de ejeución**: Cuando **foo** se ejecuta, busca **var** en el ambiente actual de ejecución, encontrando la definición más reciente (**"local"**).
- **Ausencia de closures**: No hay closure de este ambiente; cada función accede al contexto de ejecución vigente.

La principal diferencia conceptual es que en alcance dinámico el entorno de una función es la cadena de activaciones en tiempo de ejecución, similar a una pila, en lugar de un ambiente estático conservado.

0.6.3. Modelado en MiniLisp

Para contrastar ambos comportamientos en nuestro intérprete, modelamos los ejemplos anteriores:

```
(let (x 21)
     (let (foo (lambda (u) x))
          (let (bar (lambda (v) (let (x 73) (foo 0))))
               (bar 0))))
```

Resultados:

```
[MiniLisp]> compareScopes (let (x 21) (let (foo (lambda (u) x))
                          (let (bar (lambda (v) (let (x 73) (foo 0))))
                          (bar 0))))

[Alcance Estatico]: 21
[Alcance Dinamico]: 73
```

Expliación de los resultados:

- **Alcance estático**:
  - **foo** se define cuando **x** = **21**.
  - Se crea un closure que captura este ambiente.

- **(foo 0)** siempre accede al **x** capturado (21), independientemente del contexto de llamada

- **Alcance dinámico**:
  - **foo** no captura un ambiente.
  - **(foo 0)** busca **x** en el ambiente vigente durante la ejecución.
  - Como se llama dentro de **(let (x 73) ...)**, encuentra este binding más reciente.

0.7. Conclusiones

Bibliografía

[1] M. L. Scott, Programming Language Pragmatics, 2nd ed. San Francisco, CA: Morgan Kaufmann, 2006.

[2] M. Felleisen, "On the expressive power of programming languages," Science of Computer Programming, vol. 17,1991.

[3] D. P. Friedman and M. Wand, Essentials of Programming Languages, 3rd ed. Cambridge, MA: MIT Press, 2008.

[4] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," ACM Transactions on Programming Languages and Systems, vol. 13, 1991.

[5] W. D. Clinger, "Proper tail recursion and space efficiency," PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, 1998.

[6] G. L. Steele, Common Lisp the Language, 2nd ed. Mountain View, CA: Digital Press, 1990.

[7] G. Kiczales et al., The Art of the Metaobject Protocol. Cambridge, MA: MIT Press, 1991.

[8] Python Software Foundation, "Python Documentation," 2023.

[9] ECMA International, ECMAScript® 2023 Language Specification, ECMA-262, 2023.

[10] G. Bracha, "Blocks in Java," OOPSLA Workshop on Closures, 2004.

[11] D. Clarke and S. Drossopoulou, "Ownership, encapsulation, and the disjointness of type and effect," OOPSLA, 2002.

[12] S. Erdweg et al., "Sound and predictable software evolution," Onward!, 2015.

[13] H. Abelson and G. J. Sussman, Structure and Interpretation of Computer Programs, 2nd ed., MIT Press, 1996.