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

Capítulo 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.

Capítulo 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.

2.1. Alcance léxico (estático)

El alcance estático, también (también llamado 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 cerraduras**: Una cerradura (o **closures**, palabra con la cuál también nos referiremos a estas a lo largo del reporte) 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.

2.2. 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.

2.3. 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.

2.4. Conceptos clave

2.4.1. Cerraduras

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 [?].

2.4.2. 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 estu-

diada por Clinger, quien analiza la relación entre recursión adecuada (proper tail recursion) y uso constante de espacio [**?**].

2.4.3. 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.

2.4.4. Contextos de ejecución

Son marcos creados al invocar funciones o bloques, que almacenan variables locales, parámetros y referencias a ambientes externos.

Capítulo 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.

3.1. 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.

Capítulo 4

Implementación práctica

El propósito de esta implementación es ilustrar (de manera más concreta) cómo se presentan las reglas de alcance léxico y dinámico.

Para ello, en nuestro programa reutilizaremos la base de nuesto proyecto 1 MiniLisp . Pero agregando las evaluaciones de ambos alcances para ver los diferentes resultados de aplicarlos. A su vez que mostramos paso por paso lo que internamente se va procesando: como las cerraduras que se crean con los ambientes que se caputaran o las asignaciones que se van agregando al ambiente.

Dado que reutilizamos gran parte del programa implementado en MiniLisp , la explicación de gran parte del código es breve y directa.

4.1. Arquitectura del programa

Cada módulo transforma los datos de entrada de la siguiente manera:
Damos el siguiente ejemplo de entrada:

```
"(let (x 10) (+ x 5))"
```

- **Lexer (Alex)**: En este parte de la implementación su función es la de convertir el texto plano en tokens.

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

Código 4.1: Estructura del lexer con Alex

Para nuestro ejemplo:

```
[TokenPA, TokenLet, TokenPA, TokenVar "x", TokenNum 10, TokenPC,
TokenPA, TokenSum...]
```

- **Parser (Happy)**: El parser se encarga de transformar los tokens en árbol sintáctico usando la herramienta **Happy**.

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

Código 4.2: Gramática para lets del parser (Happy)

Estructura de salida (ASA.hs):

```
data ASA = Var String
| Num Int
| Let String ASA ASA
| Lambda [String] ASA
| App ASA [ASA]
| Add [ASA]
deriving (Show, Eq)
```

Código 4.3: Estructura del árbol de sintaxis abstracta (ASA.hs)

Para nuestro ejemplo, al pasarle la lista de tokens:

```
Let "x"(Num 10) (Add [Var "x", Num 5])
```

- **Desugar**: La responsabilidad es de convertir el ASA .ªzucarado.ª su forma núcleo. Para este caso, a diferencia de la implementaición de MiniLisp , desazucaramos los tipos de dato ASA a tipos de dato ASV.

Transformaciones (Desugar.hs):

```
--Let aplicacion de funciones
desugar (Let i v b) = AppV (FunV i (desugar b)) (desugar v)
--Lambda multi-parametro y se 'currifica'
desugar (Lambda ps b) = desugarLmb ps b
--Operadores n-arios a binarios
desugar (Add xs) = desugarOps AddV xs
```

Código 4.4: Transformaciones del desazucarador (Desugar.hs)

Definimos las estructuras de ambientes que usaremos junto con las formas núcleo como sigue:

```
type Env = [(String, ASV)]

data ASV = VarV String
| NumV Int
| BoolV Bool
| FunV String ASV
| AppV ASV ASV
| Closure String ASV Env
deriving (Show, Eq)
```

Código 4.5: Estructura del núcleo (ASV.hs)

En nuestro pequeño ejemplo queda como sigue:

```
AppV (FunV "x"(AddV (VarV "x") (NumV 5))) (NumV 10)
```

Para nuestro intérprete, tenemos los módulos `StaticScope` y `DynamicScope` para realizar la evaluacion con alcance estático y dinámico respectivamente. La principal diferencia entre ellos es el como evalúan las funciones lambda (`FunV`) y las aplicaciones de funciones (`AppV`).

4.1.1. Intérprete Estático

Como sabemos, al utilizar el alcance estatico necesitamos implementar el concepto de cerraduras para caputar el ambiente y preservar el valor de una variable en el momento en el cual fue invocada. Nuestra funcion pasitoLex (pasito porque utilizamos evaluación de paso pequeño y Lex por LexicalScope, el término en inglés para alcance estático) hace lo justo cuando cae en el patrón `FunV`:

```
1  pasitoLex :: ASV -> Env -> (ASV, Env)
2  --Buscamos la variable en el ambiente
3  pasitoLex (VarV i) env = (lookupEnv i env, env)
4  --Creamos las cerraduras a partir de funciones
5  pasitoLex (FunV p c) env =
6  let msg = "[Closure]: lamb" ++ p ++ ". captura el ambiente: { "
7            ++ showEnv env ++ " }"
8  in trace msg (Closure p c env, env)
```

Código 4.6: Evaluación de `FunV` con alcance estático.

Al caer en un `FunV`, simplemente creamos la cerradura capturando el ambiente actual:

```
(Closure p c env, env)
```

MAFER <explicar mejor esta parte de Closure p c env, env>.

A su vez que nos apoyamos de la función `trace` de Haskell para indicarle al usuario que hemos creado la cerradura con el ambiente actual. Además `trace` nos apoyamos de la función `trace`showEnv para mostrar en texto el ambiente que capturamos con la cerradura:

```
1  -- Funcion auxiliar para representar ambientes
2  showEnv :: Env -> String
3  showEnv [] = ""
4  showEnv ((x,v):xs) = x ++ " -> " ++ saca v ++ (if null xs then "" else "
       , " ++ showEnv xs)
```

Código 4.7: Función para mostrar al usuario el ambiente actual.

Hacemos un breve parentesis para explicar que hemos extendido el propósito de la función `saca`, implementada con anterioridad para mostrar al usuario los valores canónicos resultantes de la evaluación al programa dado como una representación comprensible para el mismo. Sin embargo como hemos implementado la función de mostrar nuestro proceso de evaluación con alcances, necesitábamos extender esta función; ya que no podemos asegurar que el ambiente tenga puramente valores finales.

```haskell
saca :: ASV -> String
saca (NumV n)  = show n
saca (BoolV True)  = "#t"
saca (BoolV False) = "#f"
saca (VarV x) = x
saca (AppV f a) = "(" ++ saca f ++ " " ++ saca a ++ ")"
saca (FunV p body) = "lamb" ++ p ++ ". " ++ saca body
saca (Closure p body env) = "<lamb" ++ p ++ ". " ++ saca body ++ ", env
   = {" ++ showEnv env ++ "}>"
```

Código 4.8: Resumen de la función saca extendida para mostrar los ASV al usuario.

Continuando con las reglas de pasitoLex, tenemos ahora la regla para la aplicación de funciones. Recordemos que, tenemos dos casos para evaluar en paso pequeño la aplicación de funciones. En la primer regla no hay mucho que explicar que no hayamos ya comentado en el primer proyecto:

```haskell
pasitoLex (AppV (Closure p c e) a) env
  | isValue a || isClosure a =
      let msg = "[Aplicacion]: Asignando " ++ p ++ " = " ++ saca a
                ++ " en el ambiente capturado"
      in trace msg (c, (p, a) : e)
  | otherwise =
    let (a', env') = pasitoLex a env
    in (AppV (Closure p c e) a', env')
pasitoLex (AppV f a) env = let (f', env') = pasitoLex f env
                           in (AppV f' a, env')
```

Código 4.9: Evaluaciónn de `AppV` con alcance estático.

Simplemente evaluamos continuamos con la aplicación de la función f aplciada al arguemnto a, pero evaluando esta función a su vez que extendemos el ambiente.

Para el siguiente caso, en donde ya tenemos una cerradura la cuál aplicar, comprobamos que sea un valor canónico. De ser este el caso, realizamos la aplicación del cuerpo con el valor a en base al parámetro que se tiene bajo el ambiente de la cerradura, no de la aplicación de función. Aclaramos además, que también mostramos la sustitución que se hace al parámetro al momento de la aplicación. En otro caso, seguimos evaluando el argumento a hasta llegar a un valor.

4.1.2. Intérprete Dinámico

Continuamos ahora con nuestra implementación para la evaluación con alcance dinámico.

Para el alcance dinámico en nuestro programa, implementamos la función `pasitoDyn`, el cuál se diferencia de `pasitoLex` en las evaluaciones a funciones y aplicaciones de funciones. Al evaluar `FunV` no hacemos gran cosa, simplemente regresamos la misma función, pues como hemos mencionado, con el alcance dinámico no generamos cerraduras; resolvemos las variables en base al ambiente en el momento de la ejecución.

Para la aplicación de funciones, evaluamos la función a aplicar hasta llegar al tipo `FunV`. En este caso, una vez el argumento es un valor realizamos la sustitución del parámetro p con ese valor y lo asignamos al ambiente.

```
pasitoDyn :: ASV -> Env -> (ASV, Env)
pasitoDyn (AppV (FunV p c) a) env
  | isValue a || isFunV a =
      trace ("[Aplicacion]: Asignando " ++ p ++ " = " ++ saca a ++ " en
         el ambiente") (c, (p, a) : env)
  | otherwise =
    let (a', env') = pasitoDyn a env
    in (AppV (FunV p c) a', env')
pasitoDyn (AppV f a) env = let (f', env') = pasitoDyn f env
                           in (AppV f' a, env')
```

Código 4.10: Evaluación de `AppV` con alcance dinámico.

Como extra, agregamos la característica de mostrar al usuario cuando `lookupEnv` tiene éxito y encuentra una variable i en el ambiente, mostramos esa variable con el valor asignado. Lo podemos ver a continuación:

```
lookupEnv :: String -> Env -> ASV
lookupEnv i [] = error ("Variable 'Var " ++ i ++ "' no definida")
lookupEnv i ((j, v):e)
  | i == j =
      trace ("[Lookup]: " ++ j ++ " -> " ++ saca v) v
  | otherwise = lookupEnv i e
```

Código 4.11: Evaluación de `AppV` con alcance dinámico.

4.1.3. Interfaz principal

Para probar el programa, agregados una validación de prefijos para que el usuario vea los resultados de evaluar expresiones con alguno de los dos alcances o con ambos:

```
validate :: String -> IO ()
validate input
  | "staticScope" `isPrefixOf` input = let expr = quitPrefix "
     staticScope" input
                                       in evalMode True False expr
  | "dynamicScope" `isPrefixOf` input = let expr = quitPrefix "
     dynamicScope" input
                                        in evalMode False True expr
  | "compareScopes" `isPrefixOf` input = let expr = quitPrefix "
     compareScopes" input
                                         in evalMode True True expr
  | otherwise = do
      putStrLn "Por favor indique el tipo de alcance:"
      putStrLn "staticScope <expr>"
      putStrLn "dynamicScope <expr>"
      putStrLn "compareScopes <expr>"
```

Código 4.12: Función `validate` para validar la entrada del usuario.

Nos apoyamos del operador `` `isPrefixOf` `` para comprobar que el usuario haya solicitado un alcance con el cual evaluar la expresión. En caso de éxito llamamos a la otra función

`evalMode` para comenzar a evaluar la expresión dada por el usuario quitando el prefijo del alcance que solicitado.

```
-- Evaluamos la expresion usando alcance estatico/dinamico Bool1 para
    static Bool2 para dynamic
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 do
    let res = evalStatic asv []
    putStrLn "\n===== Alcance Estatico ====="
    putStrLn (saca res)
    else return ()
  if dynamic
    then do
    let res = evalDynamic asv []
    putStrLn "\n===== Alcance Dinamico ====="
    putStrLn (saca res)
    else return ()
  putStrLn ""

-- Eliminamos el prefijo (staticScope/dynamicScope/bothScope) y espacios
    extra
quitPrefix :: String -> String -> String
quitPrefix pref str = dropWhile (== ' ') (drop (length pref) str)
```

Código 4.13: Funciones auxiliares `evalMode` y `quitPrefix` para procesar la entrada del usuario.

Con `evalMode` evaluamos la expresión dada a través de toda nuestra arquitectura de MiniLisp , desde el analizador léxico, hasta sacar el valor resultante. Antes de ello dependiendo del alcance solicitado se evaluará esa expresión con nuestros intérpretes.

En resumen, para probar el programa, se deben escribir los comandos de la siguiente manera:

```
#Para ver los resultados con alcance estatico unicamente:
[MiniLisp]> staticScope <expr>
...
#Para solo ver los resultados con alcance dinamico:
[MiniLisp]> dynamicScope <expr>
...
#Para ver ambos resultados y comparar:
[MiniLisp]> compareScopes <expr>
```

Capítulo 5

Casos de Estudio

5.1. Closures en Alcance Léxico en JavaScript

Como sabemos, una *cerradura* (o *closure*) es una función que retiene el acceso a variables fuera de la función, incluso después de que dicha función haya terminado su ejecución. Este mecanismo permite que la función "*recuerde*"las asginaciones echas dentro de esta. Lo que denominamos *ambiente*, con el alcance estático hacemos que se guarda el ambiente en el cual fue creada, manteniendo acceso a variables que de otra manera estarían fuera de alcance. Veamos por ejemplo una implementacion en Javascript [**?**]:

```
1   function foo() {
2       let var = "Holaaa!";
3       function hoo() {
4           console.log(var);
5           var = "Adios..."
6       }
7       return hoo;
8   }
9   const closure = foo();
10  closure();
11  closure();
```

Código 5.1: Función ejemplo para mostrar el funcionamiento de cerraduras en JavaScript.

La salida es:

```
1  Holaaa!
2  Adios..
```

En este ejemplo:

En la primer llamada `closure();`, `foo()` se ejecuta creando un nuevo ambiente para este.

En ese ambiente se mete: `[var -> "Holaa!"]`. Como JavaScript implementa alcance estáticos, cuando *hoo* es definida, captura el ambiente donde fue creada, incluyendo `var` pues ya estaba en el ambiente, obviamente. Este ambiente creado es de la forma: `closure = { [var -> "Holaaa!"] }`.

Cuando `foo()` termina y devuelve la función `hoo()`, aunque la pila de ejecución de `foo()` también se borra, su ambiente no lo hace, porque `hoo()` lo sigue guardando por su cerradura.

Por ello decimos que una cerradura "*recuerda*" su ambiente donde fue declarada.

De este modo en la primer llamada, usamos la cerradura y se imprime "*Holaaa!*", y después de imprimir se modifica el ambiente capturado con `[var -> "Adiós..."]`. Así es como se imprime en la segunda llamada "*Adios...*"pues ahora `var` ya fue modificado en el ambiente guardado dentro de la cerradura.

Si lo pensamos bien, las cerraduras estan fuertemente ligadas al tema de varibales locales y globales, pues con estas logramos que una función mantenga variables ocultas y que solo sean accesibles dentro de esa función.

5.2. Alcance Dinámico con Bash

En contraste, tenemos el alcance dinámico resuelve las variables basándose en el contexto de ejecución actual, no en el ámbito de definición. Bash ser caracteriza por ser uno de los pocos lenguajes modernos que aún preserva el uso del alcance dinámico [?].

Veamos el siguiente ejemplo [?]:

```
#!/bin/bash

var="es global"

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

function hoo() {
  local var="es local"
  foo
}

hoo
```

Código 5.2: Ejemplo en Bash para mostrar el alcance dinámico.

La salida es:

```
var: es local
```

En este ejemplo:
`foo()` busca `var` en su donde fue declarada (en `hoo()`), no donde se definió. Antes de que se llame a `hoo()` el ambiente tiene: `[ var ->`"*es global*"`]`, al llamarse a `hoo()`, metemos al ambiente `[ var ->` "*es local*"], por lo que este es nuestra variable `var` más reciente. De este modo, al evaluarse `foo()`, `var` es la cadena .es localza que es la que se encuentra primero. Por ello se imprime "*var: es local*".

El ambiente de la función es simplemente la cadena de activaciones en tiempo de ejecución, como una pila. Por lo que no hay un ambiente estático que conservar, por ello no hay closures. Y es esta la principal diferencia entre alcance dinámico y estático.

5.3. Comparando resultados con nuestro programa

Para contrastar ambos comportamientos en nuestro intérprete, utilizamos los ejemplos:

```
let x = 21;
function foo() {
  console.log(x);
}
function goo() {
  let x = 73;
  foo();
}
goo(); // La salida es 21
```

Código 5.3: Ejemplo evaluación con alcance estático en JavaScript.

```
y = 21
foo() {
  echo $y;
}
goo() {
  local y = 73;
  foo;
}
bar  #Se imprime 73 esta vez
```

Código 5.4: Ejemplo evaluación con alcance dinámico en Bash.

Los ejemplos anteriores los podemos modelar en nuestro programa como sigue:

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

Elegimos parámetros distintos (u y v) en las definiciones de las funciones λ, para evitar colisiones de nombres con la variable libre x. Además hacemos las aplicaciones `(foo 0)` y `(hoo 0)` porque necesitamos forzar una aplicación de función por que: en el alcance estático, se usa el ambiente guardado en la cerradura y en el alcance dinámico, se busca en el ambiente de llamada.

Notemos además, que por el mismo motivo, el valor del argumento en la aplicación no importa, por ello ponemos 0 por omisión.

MAFER <Aqui puedes explicar mejor cada resultado pero si lo ves bien entonces dejalo asi>

Al dar esta expresión por nuestro intérprete podemos ver que la salida corresponde a cada evaluación:

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

===== Alcance Estatico =====
```

```
[Closure]: lx. captura el ambiente: {  }
[Aplicacion]: Asignando x = 21 en el ambiente capturado
[Closure]: lfoo. captura el ambiente: { x -> 21 }
[Closure]: lu. captura el ambiente: { x -> 21 }
[Aplicacion]: Asignando foo = <lu. x, env = {x -> 21}> en el ambiente
   capturado
[Closure]: lhoo. captura el ambiente: { foo -> <lu. x, env = {x -> 21}>, x
    -> 21 }
[Closure]: lv. captura el ambiente: { foo -> <lu. x, env = {x -> 21}>, x
   -> 21 }
[Aplicacion]: Asignando hoo = <lv. (lx. (foo 0) 73), env = {foo -> <lu. x,
    env = {x -> 21}>, x -> 21}> en el ambiente capturado
[Lookup]: hoo -> <lv. (lx. (foo 0) 73), env = {foo -> <lu. x, env = {x ->
   21}>, x -> 21}>
[Aplicacion]: Asignando v = 0 en el ambiente capturado
[Closure]: lx. captura el ambiente: { v -> 0, foo -> <lu. x, env = {x ->
   21}>, x -> 21 }
[Aplicacion]: Asignando x = 73 en el ambiente capturado
[Lookup]: foo -> <lu. x, env = {x -> 21}>
[Aplicacion]: Asignando u = 0 en el ambiente capturado
[Lookup]: x -> 21
21
```

En nuestro resultado con alcance estático:

1. Se define x en un ambiente vacío, luego lo asginamos como $x = 21$ en el ambiente.
2. Se define foo como `(lambda (u) x)`, con `[Closure]: λu.` captura el ambiente: `{ x ->21 }`, `foo` hace lo propio de `foo`, capturar la versión de x que había en ese momento, es decir: $x = 21$.

   Esta es la esencia del alcance estático, la función recuerda el ambiente del momento de su definición.
3. Luego definimos `hoo`, que internamente hace un let `(x 73)` antes de llamar a `foo`.
4. Finalmente hacemos `(hoo 0)`. Dentro de `hoo`, aparece una nueva $x = 73$, pero como la variable libre x de `foo` ya estaba capturada con `x ->21`, la nueva x no afecta a la que usa `foo`.

   Por ello aparece: `[Lookup]: x ->21`, y ese es el resultado final: 21. Coincidiendo correctamente con el ejemplo propuesto en JavaScript.

Al intentarlo con alcance dinámico:

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

===== Alcance Dinamico =====
[Aplicacion]: Asignando x = 21 en el ambiente
[Aplicacion]: Asignando foo = lu. x en el ambiente
[Aplicacion]: Asignando hoo = lv. (lx. (foo 0) 73) en el ambiente
[Lookup]: hoo -> lv. (lx. (foo 0) 73)
[Aplicacion]: Asignando v = 0 en el ambiente
```

```
[Aplicacion]: Asignando x = 73 en el ambiente
[Lookup]: foo -> lu. x
[Aplicacion]: Asignando u = 0 en el ambiente
[Lookup]: x -> 73
73
```

1. Al definir `foo` en modo dinámico, no se captura ambiente, a diferencia del caso estático, sino que podemos ver que se imprime:

   `[Aplicación]: Asignando foo =` λ`u. x en el ambiente`

   Pero nunca aparece la creación de una closure con ambiente capturado.

2. Después, dentro de hoo, redefinimos $x = 73$.

3. Cuando `foo` es llamada, ahora la búsqueda de la variable x ocurre en el ambiente de la llamada, no en el ambiente de definición. Lo podemos ver claramente en: `[Lookup]: x ->73`

   Por lo tanto el resultado es 73.

MAFER <FALTA AGREGAR ESTO DE PYTHON Python:>
Lexical scoping: nonlocal y global permiten modificar bindings en ámbitos externos — esto explica la razón de herramientas especiales para reasignar closures en Python.

Capítulo 6

Conclusiones

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.

[14] Hooooo https://www.geeksforgeeks.org/javascript/closure-in-javascript/

[15] foooo https://www.gnu.org/software/bash/manual/bash.html

[16] goooo https://dev-aditya.medium.com/lexical-vs-dynamic-scoping-1ee3c50f26ea