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:

14 de diciembre, 2025

Índice

Capítulo 1

Introducción

En el diseño de lenguajes de programación existe una distinción conceptual entre **alcance estático** (también llamado **alcance léxico**[1]) 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 estático* 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 cerraduras) incorporan **reglas mixtas** que combinan aspectos de ambos modelos. Esto llega a presenta un desafío al razonar sobre el comportamiento del lenguaje y como ejecuta los programas, 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. *Michael. L. Scott* señala en su obra (*Programming Lanfuahe Pragmatics*) [1] 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 *Matthias Felleisen* en su estudio sobre el poder expresivo de los lenguajes [2]. 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, cerraduras y resolución diferida*. La metodología que llevamos a cabo combina una revisión teórica de la literatura clásica sobre semántica de lenguajes con la implementación práctica con la cual, veremos paso a paso como se desarrolla la evaluación de ambos alcances. Con el fin de observar de manera concreta el comportamiento derivado de usar las reglas dependiendo del alcance empleado.

[1] Utilizaremos ambos nombres indistintamente a lo largo del reporte.

Con esta doble aproximación permitimos 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 una variable (o identificador) pueda ser utilizada y cómo se vincula con su valor con el cuál es referenciado a lo largo del programa. 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*, establece que la asociación entre un nombre y su valor se determina según la estructura textual del programa. Es decir, para que se determine el valor de un identificador, este valor se busca observando en el bloque de código o función donde fue declarado, independientemente desde donde fue invocado. *Michael 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 [1].

En términos más simples, la formalización propuesta por Scott nos dice que: cada bloque de código del programa define un *ambiente* que contiene las variables declaradas en él, y que dichos ambientes se organizan de forma jerárquica siguiendo la estructura del código fuente. Cuando un identificador es utilizado, el compilador (o en nuestro caso el intérprete) resuelve su referencia buscando primero en el ambiente más interno y, si no lo encuentra, continúa la búsqueda en los ambientes externos hasta localizar una definición válida.

Por ejemplo, en el siguiente fragmento de código:

```
x = 10
func f() {
    x = 20
    func g() {
        print(x)
    g()
    }
```

```
}
f()
```

La variable `x` utilizada dentro de la función `g()` se asocia con la variable `x` definida en `f`, ya que esa es la definición más cercana en la estructura textual del programa.
Aunque `g()` es invocada desde `f()`, el valor de `x` que utiliza no depende del punto de llamada, sino del entorno léxico en el que la función fue definida.

Las características principales del *alcance estático* 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 *closure*[1]) captura el ambiente léxico en el que fue definido. Justo como es mencionado en el trabajo de *Daniel P. Friedman* (*Essentials of Programming Languages* [3]), nos referimos a este tipo de estructuras de datos como *cerraduras* porque contienen todo lo que el proceso de evaluación necesita en el orden exacto que debe ser aplicado.

  Esto permite que las funciones sigan accediendo a variables externas aunque su bloque original haya terminado.
- **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* determina el valor de una variable 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, implicando que la asociación entre un nombre y su valor no depende de la estructura textual del código, sino del orden en que las funciones fueron invocadas. Generando como consecuencia, que un mismo identificador puede referirse a valores distintos según el contexto dinámico en el que se ejecuta.
Nuevamente, *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 [1]. Ya que como se mencionó, las variables se resuelven recorriendo los ambientes generados por las llamadas a funciones que están activas en ese momento, comenzando por la más reciente y avanzando (o mejor dicho retrocediendo) hacia las anteriores.

[1]Del mismo modo que con el alcance léxico, nos referiremos a lo largo del reporte a estas como *cerraduras* tanto como *closures*.

Por ejemplo, considérese el siguiente fragmento de código con *alcance dinámico*:

```
x = 10
func f() {
    print(x)
}
func g() {
    x = 20
    f()
}
g()
```

En este caso, cuando la función `f()` accede a la variable `x`, el valor utilizado es 20, ya que la llamada a `f()` ocurre dentro del contexto de ejecución de `g()`, donde `x` fue definida.

Las características principales de este alcance 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 bien definidas en tiempo de compilación, con mecanismos dinámicos que introducen flexibilidad en tiempo de ejecución. En estos sistemas, la estructura textual del programa sigue siendo fundamental para resolver la mayoría de las referencias, pero ciertos identificadores o comportamientos se determinan dinámicamente según el estado de ejecución.
Martín Abadi y *Luca Cardelli* argumentan que la frontera entre lo estático y lo dinámico no es absoluta. En su trabajo sobre tipado dinámico en lenguajes estáticamente tipados () señalan que incluso sistemas diseñados con fuertes garantías estáticas pueden incorporar características dinámicas (como la carga tardía), obligando la información disponible en tiempo de compilación con decisiones que solo pueden tomarse en tiempo de ejecución [4].

Un ejemplo representativo es **Python**, que utiliza *alcance léxico* para las variables locales y cerraduras, pero mantiene reglas dinámicaspara la resolución de los global y lo predefinido. En particular, cuando una variable no se encuentra en el ambiente local ni en los ambientes léxicos externos, el intérprete recurre dinámicamente a ambientes globales y `built-in`. Este

comportamiento ilustra cómo un lenguaje con una base léxica clara incorpora decisiones dinámicas en la resolución final de dichas variables [5].

De manera similar, **JavaScript** sigue un patrón parecido al mezclar *closures* con un ambiente global dinámico y mutable. Si bien las funciones capturan el ambiente en el que fueron definidas, la existencia de un objeto global compartido y mecanismos como `this` (cuyo enlace depende del contexto de invocación) introducen elementos propios del *alcance dinámico*. Así, la referencia a ciertos identificadores no puede determinarse únicamente a partir de la estructura textual del programa [6].

En el caso de **Lisp**, el carácter mixto es aún más explícito. Aunque las versiones modernas utilizan principalmente *alcance estático*, muchos dialectos permiten declarar *variables especiales* con *alcance dinámico*. Estas variables se resuelven siguiendo la cadena de llamadas activa, coexistiendo con variables léxicas ordinarias dentro del mismo programa [7].

2.4. Conceptos clave

2.4.1. Cerraduras

Son funciones que capturan su ambiente léxico en el momento de su definición. *Friedman* y *Wand* como mencionamos anteriormente, describen las *closures* como pares que combinan código y ambiente, permitiendo diferir la ejecución sin perder contexto, justo porque contienen todo lo que la evaluación necesita en el orden exacto en el cual se debe realizar la aplicación [3].

2.4.2. Ambientes

Son estructuras que modelan la asociación entre identificadores y sus valores durante la ejecución de un programa. Formalmente, pueden entenderse como funciones o estructuras de mapeo que, dado un nombre, permiten recuperar el valor o ubicación correspondiente.
La implementación eficiente de ambientes, especialmente en presencia de funciones recursivas y llamadas en cola, ha sido estudiada por *William D. Clinger*, quien los analiza, desde una perspectiva operacional, destacando su papel en la correcta implementación de la *recursión adecuada* (*Proper Tail Recursion*) [8]. *Clinger* demuestra que, bajo un manejo apropiado de los ambientes, las llamadas en posición de cola no requieren la creación de nuevos marcos de ambiente, lo que permite que la ejecución de funciones recursivas en cola se realice utilizando espacio constante.

Podemos decir entonces, que desde este punto de vista, los ambientes no deben interpretarse únicamente como estructuras de acumulación, sino como esctructuras que pueden ser reutilizadas, compartidas o descartadas según la semántica del lenguaje y la posición de las llamadas.

2.4.3. Cadenas de alcance

Son secuencias ordenadas de ambientes que representan los posibles contextos donde se puede resolver una variable. Cada elemento de la cadena corresponde a un ambiente asociado a un bloque o función. En los lenguajes con *alcance estático*, esta cadena se determina estáticamente (valga la redundancia), mientras que en lenguajes con alcance dinámico, la cadena se construye en tiempo de ejecución siguiendo la pila de llamadas activa

2.4.4. Contextos de ejecución

Son *marcos de activación* creados al invocar funciones o bloques de código, que almacenan variables locales, parámetros y referencias a ambientes externos. Además de que estos contienen la información necesaria para reanudar la ejecución, como la dirección de retorno y el estado de evaluación.

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 [9]. Según *Gregor 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 meta-objetos [10].

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 [5]. 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 *cerraduras* léxicas 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) pues, esto podía generar ambigüedades semánticas [11].

Además, el uso histórico de la sentencia `with` que intentaba alterar dinámicamente el ambiente de las variables, pero solo lograba hacer el código ilegible. Tal como señala *Gilda Bracha*, estas variaciones reflejan una transición desde modelos más dinámicos hacia diseños con mayor predicibilidad y claridad estructural [12].

Pese a sus diferencias sintácticas, estos lenguajes comparten patrones comunes. Todos parten de una base léxica pero incluyen excepciones dinámicas (como el manejo de estructuras de contexto global o algunas palabras clave) que introducen elementos dinámicos. El resultado, crea modelos híbridos donde la claridad del *alcance léxico* coexiste con la necesidad de operar de manera flexible en tiempo de ejecución.

3.1. Ventajas de desventajas de los modelos mixtos

Como hemos visto, los modelos de alcance mixtos combinan estructuras léxicas con mecanismos dinámicos, una mezcla que 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 cerraduras 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 [15].

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**: *Dave Clarke* y *Sophia Drossopoulou* advierten que, al permitir que un nombre se resuelva en tantos lugares, se rompe el encapsulamiento y se debilita el sistema [13].
- **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 [14].
- **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)
```

Está parte es una de las mas importantes ya que es el proceso necesario para convertir el alnace dinámico en estático, lo que hace es añadir un tercer campo al constructor que guarda al ambiente actual.

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.

```
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:

```
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 [6]:

```
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:

```
Holaaa!
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[1] [16].

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*".

[1]Como bien se explica en su documentación, citando: "Las variables en la terminal (Shell) implementan el alcance estático para controlar la visibilidad de una variable dentro de las funciones."

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. Closures en Python

En esta sección analizamos el caso particular de Python, un lenguaje que implementa *alcance léxico* de forma similar a JavaScript, pero con procesos más explícitos que reflejan un diseño mixto controlado. Python prioriza la claridad semántica y la prevención de errores. Como se mencionó en el capítulo anterior, esto surge a partir de uno de los axiomas fundamentales del lenguaje [17].

A diferencia de JavaScript, donde una función puede modificar variables de ambientes exteriores sin una declaración previa, Python exige el uso explícito de las palabras clave `nonlocal` o `global` para alterar variables fuera del ambiente local. Esto se hace evidente cuándo una función interactúa con su ambiente externo, reduciendo ambigüedades en el razonamiento del código.

```
def funcion_externa():
    mensaje = "Hola"

    def funcion_interna():
        nonlocal mensaje  # Declaracion explicita
        print(mensaje)
        mensaje = "Adios"

    return funcion_interna

closure = funcion_externa()
closure()  # Imprime: "Hola"
closure()  # Imprime: "Adios"
```

Código 5.3: Ejemplo de closure en Python.

En el ejemplo anterior, `funcion_interna` constituye un *closure*, ya que captura la variable `mensaje` definida en su ambiente envolvente. El uso de `nonlocal` indica explícitamente que la asignación debe afectar a dicha variable y no crear una nueva en el ambiente local de la función interna.

Como se explicó previamente, Python utiliza un proceso de búsqueda de identificadores conocido como la regla **LEGB**, que define el orden en el que se resuelven los nombres:

- **Local**: variables definidas en la función actual.
- **Enclosing**: variables definidas en funciones envolventes.
- **Global**: variables definidas a nivel de módulo.
- **Built-in**: identificadores incorporados por el lenguaje.

```python
x = "global"

def nivel1():
    x = "enclosing"

    def nivel2():
        nonlocal x  # Accede a x de nivel1, no a la global
        x = "modificado"
        return x

    return nivel2

func = nivel1()
print(func())  # "modificado"
print(x)       # "global"
```

Código 5.4: Proceso `nonlocal` y regla LEGB.

Este comportamiento puede representarse en nuestro lenguaje de la siguiente manera:

```
-- Simulacion aproximada del comportamiento de Python
(let (x "global")
     (let (nivel1 (lambda ()
                    (let (x "enclosing")
                         (let (nivel2 (lambda ()
                                        -- En Python seria necesario "
                                           nonlocal"
                                        (let (x "modificado")
                                             x)))
                              (nivel2)))))
          (nivel1)))
```

Código 5.5: Simulación del comportamiento de Python en MiniLisp.

Actualmente, nuestro lenguaje no distingue ambientes local, enclosing ó global; sin embargo, podría extenderse para soportar este modelo.

Python representa un punto de equilibrio en el diseño del manejo de alcance: no es tan permisivo como JavaScript, pero tampoco tan restrictivo como otros lenguajes. Su modelo respeta el alcance léxico, mantiene consistencia conceptual y fomenta buenas prácticas al hacer explícitas las interacciones entre ambientes.

5.4. Comparando resultados con nuestro programa

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

```javascript
  let x = 21;
  function foo() {
    console.log(x);
  }
  function goo() {
    let x = 73;
    foo();
```

```
  }
  goo(); // La salida es 21
```

Código 5.6: Ejemplo evaluación con alcance estático en JavaScript.

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

Código 5.7: 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.

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 = λu. 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.

Capítulo 6

Conclusiones

La implementación de nuestro proyecto es un mini-lenguaje con alcance configurable, muestra que la elección entre alcance léxico y dinámico es una decisión central en el diseño de lenguajes de programación. El alcance léxico facilita el razonamiento sobre el código, favorece el encapsulamiento y contribuye a la creación de programas más claros y mantenibles. En contraste, el alcance dinámico ofrece mayor flexibilidad al permitir que ciertos valores dependan del contexto de ejecución, lo cual resulta útil en escenarios de configuración y parámetros implícitos.

Lenguajes modernos como JavaScript y Python utilizan principalmente alcance léxico, pero lo aplican con enfoques distintos. JavaScript permite modificaciones implícitas del ambiente, lo que incrementa la flexibilidad, mientras que Python exige el uso explícito de `nonlocal` y `global`, priorizando la claridad y la prevención de errores. Estas diferencias evidencian que no existe un modelo único ideal, sino que cada lenguaje equilibra expresividad, seguridad y facilidad de uso según sus objetivos.

La implementación en Haskell permitió observar que, aunque la diferencia entre ambos modelos puede expresarse con cambios mínimos en el evaluador, sus efectos semánticos son significativos. Finalmente, el proyecto confirma que es posible diseñar lenguajes que soporten múltiples reglas de alcance, brindando a los programadores la flexibilidad de elegir el modelo más adecuado, y destaca la importancia de una arquitectura modular como buena práctica en el diseño de lenguajes.

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

[6] MAFER <formato IEEE>https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Closures

[7] MAFER <formato IEEE>https://www.deinprogramm.de/sperber/papers/dynamic-scope-analysis.pdf

[8] 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.

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

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

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

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

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

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

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

[16] MAFER <formato IEEE>foooo https://www.gnu.org/software/bash/manual/bash.html

[17] T. Peters, *The Zen of Python*,PEP 20, Python Software Foundation, 2004.