

INFORME PROYECTO FINAL

FUNDAMENTOS DE INTERPRETACIÓN Y COMPILACIÓN DE LENGUAJES DE PROGRAMACIÓN

Alejandro Marin Hoyos 2259353-3743

Karen Jhulieth Grijalba Ortiz 2259623-3743

Yessica Fernanda Villa Nuñez 2266301-3743

Manuel Antonio Vidales Duran 2155481-3743

**UNIVERSIDAD DEL VALLE SEDE TULUÁ
PROGRAMA ACADÉMICO DE INGENIERÍA DE SISTEMAS
2024**

2.6. Primitivas numéricas

Se implementó una función *apply-primitive* que aplica operaciones primitivas a argumentos, que pueden ser números o cadenas que representan números en diferentes bases. Aquí se explica cómo se manejan estas primitivas numéricas:

1. **Definición de *apply-primitive*:** Esta función toma la operación (*prim*) y una lista de argumentos (*args*), determinando qué operación realizar mediante un *cases*.
2. **Operaciones Primitivas:** Se definen varias operaciones primitivas como suma, resta, multiplicación, módulo, exponenciación y comparaciones. Cada operación llama a una función operaciones con los argumentos y la operación específica.
3. **Función operaciones:**
 - **Con números:** Si ambos argumentos son números, se aplica directamente la operación.
 - **Con cadenas:** Si ambos argumentos son cadenas, se determina su base utilizando *tipoBase*. Si las bases son iguales, las cadenas se convierten a números decimales, se aplica la operación y se convierte el resultado de vuelta a la base original. Si las bases son diferentes, se lanza un error.
4. **Conversión de Bases:** Varias funciones auxiliares se encargan de manejar las conversiones de bases:
 - *tipoBase* determina la base de una cadena que representa un número.
 - *ConvertirBase-Decimal* convierte una cadena de una base específica a su valor decimal.
 - *convertirDecimal-Base* convierte un número decimal a una cadena en una base específica.

Permite realizar operaciones aritméticas y comparaciones entre números, manejando tanto valores numéricos directos como representaciones en las bases pedidas en el enunciado del proyecto(binaria, octal, hexadecimal). La función operaciones verifica los tipos de los argumentos, los convierte a una base común si es necesario, realiza la operación y devuelve el resultado en la base adecuada.

2.7. Primitivas booleanas

La función *apply-primitive-bool* aplica operaciones booleanas a una lista de argumentos *args* dependiendo de la primitiva *prim*. Utiliza un *cases* para manejar diferentes operaciones booleanas como *and*, *or*, *xor*, y *not*. Para operaciones como *and* y *or*, utiliza la función *operation* con una función lambda que implementa la operación booleana respectiva, comenzando con un valor inicial (*#t* o *#f*). Para *not*, simplemente niega el primer argumento de *args*.

XOR: La función **xor** implementa la operación exclusiva *OR* (*xor*) entre dos expresiones *exp1* y *exp2*. Devuelve *#t* si exactamente una de las expresiones es verdadera (*#t*), y *#f* en cualquier otro caso

2.8. Listas

2.9. Primitivas de listas

La función **apply-primitive-list** se utiliza para aplicar una operación primitiva específica a una lista de argumentos. La operación a aplicar se determina por el valor de *`prim`*. Dependiendo de este valor, la función ejecuta una de las siguientes operaciones primitivas:

- **first-primList:** Devuelve el primer elemento de la lista de argumentos.
- **rest-primList:** Devuelve todos los elementos de la lista de argumentos excepto el primero.
- **empty-primList:** Verifica si la lista de argumentos está vacía y devuelve un valor booleano (*`#t`* si está vacía, *`#f`* si no).

La función **apply-primitive-list** es útil para realizar operaciones básicas en listas de forma genérica, seleccionando la operación específica a través del argumento *`prim`*. Esto permite una mayor flexibilidad y reutilización del código al manejar diferentes operaciones de listas con una única función.

2.10. Arrays

2.11. Primitivas de arrays

Se implementó una función **primitiva-array** que aplica operaciones primitivas a arreglos (vectores). A continuación, se explica cómo se manejan estas primitivas de arreglos:

Definición de primitiva-array

La función **primitiva-array** toma dos argumentos: *prim*, que representa la operación de arreglo a realizar, y *arg*, que es una lista de argumentos sobre los cuales se aplicará la operación. Utiliza *cases* para determinar la operación específica según el valor de *prim*.

Operaciones Primitivas de Arreglos

Las operaciones primitivas de arreglos se definen dentro del *cases*, por ejemplo:

- **length-primArr:** Devuelve la longitud del *vector*.

- **index-primArr:** Devuelve el elemento en un índice específico del *vector*.
- **slice-primArr:** Devuelve una sublista (convertida a vector) de un rango específico del *vector*.
- **setlist-primArr:** Establece un valor en un índice específico del vector y devuelve el vector modificado.

Implementación de cada Operación

- **length-primArr:**
 - Devuelve la longitud del vector usando *vector-length*.
- **index-primArr:**
 - Devuelve el elemento en el índice especificado usando *vector-ref*.
- **slice-primArr:**
 - Utiliza una función recursiva lista para crear una lista de elementos del *vector* desde un *índice de inicio* hasta un *índice final*.
 - Convierte esta lista en un vector usando *list->vector*.
- **setlist-primArr:**
 - Establece un valor en el índice especificado del *vector* usando *vector-set!*.
 - Devuelve el *vector* modificado.

El código implementado permite realizar operaciones en arreglos utilizando primitivas específicas para obtener la longitud, acceder a elementos, obtener sublistas y modificar elementos. La función ***primitiva-array*** determina la operación a realizar y maneja las manipulaciones de arreglos de manera eficiente utilizando las funciones adecuadas de Racket para vectores.

2.12. Primitivas de cadenas

apply-primitive-string es una función que aplica operaciones primitivas a cadenas. Toma una primitiva de cadena (*primitivaCadena*) y una lista de argumentos. Utiliza cases para seleccionar la operación correspondiente basada en la primitiva proporcionada. Las operaciones disponibles son:

- **length (*length-primCad*):** Esta operación devuelve la longitud de la primera cadena en la lista de argumentos (*args*). Utiliza *string-length*, una función estándar en Scheme que devuelve el número de caracteres en una cadena dada.
- **elementAt (*index-primCad*):** Esta operación devuelve el carácter en la posición especificada por el segundo elemento de *args* (obtenido mediante *cadr args*) de la primera cadena (*car args*). Utiliza *string-ref*, otra función estándar en Scheme que devuelve el carácter en una posición específica de una cadena. Es importante recordar que las posiciones en Scheme están

indexadas desde 0, por lo que *cadr args* debe ser un número válido que esté dentro del rango válido de índices para la cadena (*car args*). Si el índice especificado es mayor o igual que la longitud de la cadena, o si la cadena está vacía, podría provocar errores de índice fuera de rango.

- **concat (*concat-primCad*)**: Esta operación concatena todas las cadenas de texto proporcionadas en la lista de argumentos (*args*). Utiliza la función unir-cadenas para unir todas las cadenas de texto en una sola cadena, utilizando un espacio vacío "" como separador.

La función **unir-cadenas** es una función auxiliar utilizada por *eval-cadena-expresion* y también por *apply-primitive-string*. Esta función toma una lista de cadenas y un símbolo de separación. Utiliza una función interna unir recursiva para concatenar todas las cadenas de la lista en una sola cadena, separando cada una de ellas por el símbolo especificado. Si la lista está vacía, devuelve una cadena vacía. En caso contrario, concatena el primer elemento de la lista con el resultado de llamar recursivamente a unir con el resto de la lista.

2.13. Entornos de ligaduras y variables

La función **eval-var-decl** toma dos argumentos: *exp*, que representa la expresión de declaración de variable, y *env*, que es el ambiente en el cual se evalúa la expresión. Utiliza *cases* para determinar el tipo de declaración según el valor de *exp*.

Operaciones de Declaración de Variables

Las declaraciones de variables se definen dentro del *cases*, por ejemplo:

- **lvar-exp**: Representa una declaración de variable local.
- **let-exp**: Representa una declaración de variable usando la estructura *let*.

Implementación de cada Operación

1. **lvar-exp**:

- Toma los identificadores (*ids*), las expresiones que los valores (*rands*) y el cuerpo (*body*).
- Evalúa las expresiones *rands* en el ambiente *env* para obtener sus valores.
- Extiende el ambiente *env* con los nuevos identificadores y sus valores evaluados.
- Evalúa el cuerpo de la expresión en el nuevo ambiente extendido.

2. **let-exp**:

- Similar a *lvar-exp*, toma los identificadores, las expresiones de valores y el cuerpo.
- Evalúa las expresiones de valores en el ambiente actual para obtener los valores.

- Extiende el ambiente con los nuevos identificadores y sus valores evaluados.
- Evalúa el cuerpo de la expresión en el entorno extendido.

3. Manejo de Errores:

- Si la declaración de variable no corresponde a ninguna de las formas esperadas, lanza un error indicando que la declaración de variable es inválida.

El código implementado permite evaluar declaraciones de variables utilizando dos formas específicas (*lvar-exp* y *let-exp*). La función *eval-var-decl* evalúa las expresiones de valores en el entorno actual, extiende el ambiente con los nuevos identificadores y sus valores evaluados, y luego evalúa el cuerpo de la expresión en este nuevo ambiente extendido. Además, se maneja la detección de declaraciones de variables inválidas lanzando un error apropiado.

2.14. Condicionales

La función *eval-if-expresion* evalúa una expresión condicional “*exp1*” en un ambiente “*env*”. Si “*exp1*” evalúa a verdadero (*#t*), se evalúa *exp2* en el mismo entorno; de lo contrario, se evalúa “*exp3*”. Esto permite ejecutar diferentes ramas condicionales dentro del programa según el resultado de la evaluación de “*exp1*”.

2.15. Estructuras de control

- **For:** *eval-for-expresion* implementa un bucle for que inicializa una variable identificador con un valor inicial “inicio” obtenido evaluando “*i*” en “*env*”. Luego, itera desde “inicio” hasta “fin” (obtenido evaluando “from” en “*env*”) incrementando en “iterar” unidades en cada iteración. Dentro de cada ciclo, se evalúa “body” en un entorno extendido con “identificador” y el valor actual de “*i*”.
- **While:** *eval-while-expresion* evalúa una expresión “exp” en “env” repetidamente mientras “exp” evalúe a verdadero (*#t*). En cada iteración, se ejecuta “body” en el mismo entorno. Esta función es fundamental para implementar bucles “while” en Scheme, proporcionando una estructura de repetición controlada por la evaluación de una condición.
- **Switch:** *eval-switch-expresion* evalúa una expresión de tipo “switch” similar a las estructuras condicionales en lenguajes como C o JavaScript. La función toma una expresión “exp”, una lista de casos “cases”, una lista de expresiones “listaExp”, una expresión por defecto “default”, y un entorno “env”. Primero, evalúa la expresión “exp” y luego itera sobre los casos. Si encuentra un caso que coincide con el valor de “exp”, evalúa y devuelve la

expresión correspondiente. Si no encuentra ninguna coincidencia, evalúa y devuelve la expresión por defecto.

2.16. Begin y set

Define cómo evaluar dos tipos de expresiones: una expresión de asignación **set-exp** y una expresión de bloque **begin-exp**.

Esta expresión evalúa una asignación, donde se asigna el resultado de una expresión **rhs-exp** a un identificador **id**.

Set

Parámetros:

id: El identificador al que se va a asignar el valor.

rhs-exp: La expresión que se evaluará para obtener el valor a asignar.

- **apply-env-ref env id**: Busca una referencia al identificador `id` en el entorno `env`.
- **eval-expression rhs-exp env**: Evalúa la expresión `rhs-exp` en el entorno `env` para obtener su valor.
- **referencia**: Asigna el valor evaluado al identificador referenciado.

La función `referencia` no está definida en el fragmento proporcionado, pero se asume que esta función asigna el valor calculado al identificador en el entorno.

Begin-exp

Esta expresión evalúa una secuencia de expresiones en orden y devuelve el valor de la última expresión.

Parámetros:

- **exp**: La primera expresión de la secuencia.

- **exps**: Las expresiones restantes de la secuencia.

- **eval-expression exp env**: Evalúa la primera expresión `exp` en el entorno `env` y guarda el resultado en `acc`.
- **(cond ((null? exps) acc))**: Si `exps` es `null` (no hay más expresiones), devuelve el valor de `acc` (el resultado de evaluar `exp`).
- **(else (eval-expression (car exps) env))**: Si hay más expresiones, evalúa la primera expresión de `exps` en el entorno `env`.

set-exp: Implementa la evaluación de una expresión de asignación. Busca el identificador en el entorno y asigna el valor evaluado de `rhs-exp` a ese identificador.

begin-exp: Implementa la evaluación de una expresión de bloque. Evalúa las expresiones en secuencia, pero el fragmento proporcionado solo evalúa la primera expresión de `exps` y devuelve su resultado. Para evaluar correctamente todas las expresiones en `exps`, necesitaríamos una recursión adecuada.

Pruebas:

Datos:

```
689 ✓ (define exp1
690   (scan&parse
691     "
692     let
693       x = b1010
694       y = b1100
695     in
696       (x + y)
697     "
698   )
699 )
700 (display (eval-program exp1))
701
702 (provide (all-defined-out))
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS

```
/Desktop/InterpretadorFLP/archivo.rkt"))'
Welcome to Racket v8.12 [cs].
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b10110
"archivo.rkt"> |
```

```
688 ;(interpretador)
689 (define exp2
690   (scan&parse
691     "
692     let
693       x = b1000
694       y = b10
695     in
696       (x - y)
697     "
698   )
699 )
700 (display (eval-program exp2))
701
702 (provide (all-defined-out))
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS

```
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b10110
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b110
"archivo.rkt"> |
```



```

archivo.rkt M X test3_primitivasnumericasbinarios.rkt
archivo.rkt
687
688 ;(interpretador)
689 (define exp3
690   (scan&parse
691     "
692     let
693     x = b10
694     y = b10
695     in
696     (x * y)
697     "
698   )
699 )
700 (display (eval-program exp3))
701
702 (provide (all-defined-out))

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS

"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b110
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b100
"archivo.rkt">

```

```

archivo.rkt M X test3_primitivasnumericasbinarios.rkt
archivo.rkt
687
688 ;(interpretador)
689 (define exp4
690   (scan&parse
691     "
692     let
693     x = b10
694     y = b011
695     in
696     (x pow y)
697     "
698   )
699 )
700 (display (eval-program exp4))
701
702 (provide (all-defined-out))

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS REPL (archivo.rkt)

PS C:\Users\manue\Desktop\InterpretadorFLP> racket --repl --eval '(enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))'
Welcome to Racket v8.12 [cs].
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b1000

```

```

archivo.rkt M X test3_primitivasnumericasbinarios.rkt
archivo.rkt
687
688 ;(interpretador)
689 (define exp5
690   (scan&parse
691     "
692     let
693     x = b101
694     y = b011
695     in
696     (x mod y)
697     "
698   )
699 )
700 (display (eval-program exp5))
701
702 (provide (all-defined-out))

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS REPL (arc

"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b1000
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
b10

```

Primitivas numéricas:

```

archivo.rkt M X test5_primitivasnumericashex.rkt
archivo.rkt
687
688 ;(interpretador)
689 (define exp2
690   (scan&parse
691     "
692     let
693     x = 0x144
694     y = 0x24
695     in
696     (x - y)
697     "
698   )
699 )
700 (display (eval-program exp2))
701
702 (provide (all-defined-out))

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS REPL (archivo.rkt) +

op/InterpretadorFLP/archivo.rkt"))'
Welcome to Racket v8.12 [cs].
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
0x120

```

Primitivas booleanas:

```
archivo.rkt M X test2_primitivasbooleanas.rkt
archivo.rkt
687
688 ;(interpretador)
689 (define exp1
690   (scan&parse
691     "
692     and((1 > 2), false)
693     "
694   )
695 )
696 (display (eval-program exp1))
697
698 (provide (all-defined-out))

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS REPL (archivo.rkt) +

PS C:\Users\manue\Desktop\InterpretadorFLP> racket --repl --eval '(enter! (file "c:/Users/
op/InterpretadorFLP/archivo.rkt"))'
Welcome to Racket v8.12 [cs].
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
#f
```

Listas:

```
archivo.rkt M X test7_listas.rkt
archivo.rkt
687
688 ;(interpretador)
689 (define exp1
690   (scan&parse
691     "
692     list(1, 2, 3, 4)
693     "
694   )
695 )
696 (display (eval-program exp1))
697
698 (provide (all-defined-out))

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS REPL (archivo.rkt) +

PS C:\Users\manue\Desktop\InterpretadorFLP> racket --repl --eval '(enter! (file "c:/Use
op/InterpretadorFLP/archivo.rkt"))'
Welcome to Racket v8.12 [cs].
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
(1 2 3 4)
```

```

❏ archivo.rkt
687
688 ;(interpretador)
689 (define exp2
690   (scan&parse
691     "
692     cons(1 cons(2 empty))
693     "
694   )
695 )
696 (display (eval-program exp2))
697
698 (provide (all-defined-out))

```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS [x] REPL (archivo.rkt) +

```

"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
(1 2 3 4)
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
(1 2)
"archivo.rkt">

```

Arrays:

```

❏ archivo.rkt
687
688 ;(interpretador)
689 (define exp1
690   (scan&parse
691     "
692     let
693     s = array(1,2,3,4,5)
694     in
695     length(s)
696     "
697   )
698 )
699 (display (eval-program exp1))
700
701 (provide (all-defined-out))

```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS [x] REPL (archivo.rkt) +


```

PS C:\Users\manue\Desktop\InterpretadorFLP> racket --repl --eval '(enter! (file "c:/User
op/InterpretadorFLP/archivo.rkt"))'
Welcome to Racket v8.12 [cs].
"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))
[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]
5

```

Cadenas primitivas:

```
687
688 ;(interpretador)
689 (define exp1
690   (scan&parse
691     "
692     let
693     s = \"hola mundo cruel\"
694     in
695     string-length(s)
696     "
697   )
698 )
699 (display (eval-program exp1))
700
701 (provide (all-defined-out))
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS  REPL (archivo.rkt)

PS C:\Users\manue\Desktop\InterpretadorFLP> racket --repl --eval '(enter! (file \"c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt\"))'

Welcome to Racket v8.12 [cs].

"archivo.rkt"> (enter! (file "c:/Users/manue/Desktop/InterpretadorFLP/archivo.rkt"))

[re-loading c:\users\manue\desktop\interpretadorflp\archivo.rkt]

16

"archivo.rkt">