UNIVERSIDAD DEL VALLE

Informe - FLP

INGENIERÍA DE SISTEMAS

AUTORES:

Nilson Andrés Cuero Ocoro (2067570)

Sebastian Pérez Bastida (2067808)

Nicolás Arturo Valencia (2067802)

DOCENTE:

Carlos Andrés Saavedra Delgado

CURSO:

Fundamentos de Lenguajes Programación

Valle del Cauca - Tuluá

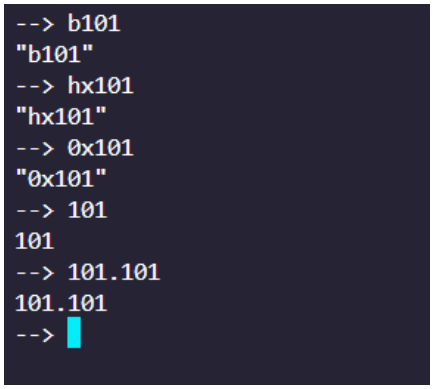
06-2024

Solución

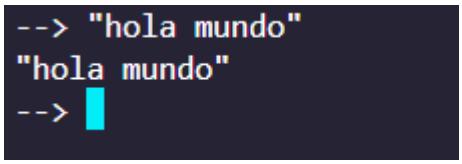
Hemos desarrollado un nuevo lenguaje de programación para la Universidad del Valle, sede Tuluá, basado en Racket. Este lenguaje integra programación imperativa, orientada a objetos, orientada a eventos, funcional y paralela, permitiendo a los estudiantes alcanzar más de 20 resultados de aprendizaje en un semestre y solucionando el problema de la disponibilidad de salas.

Especificación del lenguaje

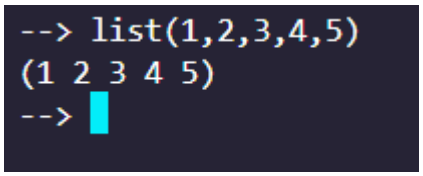
1. **Números:**



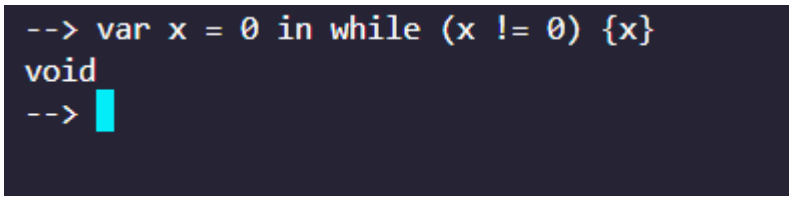
2. **Texto:**



3. **Listas:**



4. **Void:**



1.0 Primitivas numéricas

El código proporcionado está escrito en Scheme y define varias funciones relacionadas con la manipulación de cadenas y operaciones aritméticas en diferentes bases numéricas (binario, octal y hexadecimal). Aquí se proporciona un análisis detallado de cada parte del código:

Función eliminar-caracter

Esta función elimina todas las ocurrencias de un carácter específico de una cadena.

- **Entrada**: stri (cadena), ch (carácter a eliminar).
- **Proceso**: Convierte la cadena en una lista de caracteres y utiliza la función quitar_caracter para eliminar el carácter deseado. Luego, convierte la lista de caracteres de nuevo a una cadena.
- **Salida**: La cadena sin las ocurrencias del carácter ch.

Función sustituir-caracter

Esta función reemplaza todas las ocurrencias de un carácter en una cadena con otro carácter.

- **Entrada**: stri (cadena), ch (carácter a reemplazar), new-ch (nuevo carácter).
- **Proceso**: Convierte la cadena en una lista de caracteres y utiliza la función reemplazar_caracter para reemplazar el carácter deseado. Luego, convierte la lista de caracteres de nuevo a una cadena.
- **Salida**: La cadena con las ocurrencias del carácter ch reemplazadas por new-ch.

Función manejador_de_bases

Esta función maneja operaciones aritméticas en diferentes bases numéricas (binario, octal, hexadecimal).

- **Entrada**: operacion (función aritmética), arg1 y arg2 (números o cadenas), transformar (función para transformar el resultado).
- **Proceso**:
  - Si los argumentos son cadenas, determina la base numérica (binario, hexadecimal, octal) y realiza la operación aritmética adecuada.
  - Si los argumentos son números, simplemente aplica la operación.
- **Salida**: El resultado de la operación transformado por la función transformar.

Función convertir-numero

Convierte un número a una cadena en una base específica (binario, octal, hexadecimal).

- **Entrada**: num (número), base (base numérica).
- **Proceso**: Convierte el número a la cadena correspondiente en la base especificada, manejando números negativos.
- **Salida**: La cadena que representa el número en la base especificada.

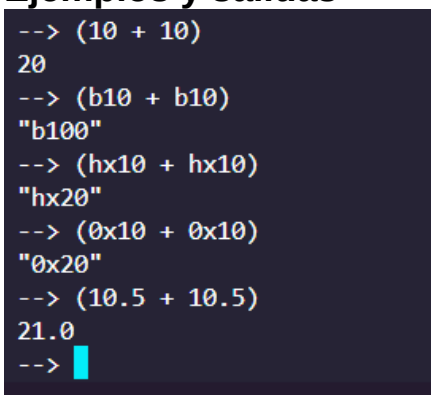
Función apply-primitive

Aplica una operación primitiva a dos argumentos.

Entrada: prim (operación primitiva), arg1 y arg2 (argumentos).

Proceso: Determina la operación primitiva a aplicar y utiliza mandejador_de_bases para manejar los argumentos y obtener el resultado.
Salida: El resultado de la operación primitiva aplicada a los argumentos.

Ejemplos y salidas



2.0 Primitivas booleanas

Funciones Lógicas

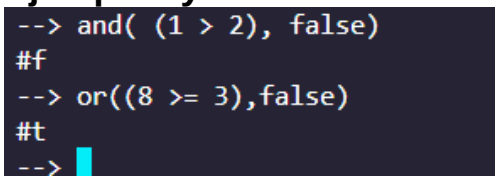
1. **y-logico**: Realiza la operación lógica AND entre dos valores booleanos.
2. **o-logico**: Realiza la operación lógica OR entre dos valores booleanos.
3. **xor-logico**: Realiza la operación lógica XOR (exclusivo OR) entre dos valores booleanos.

Función para Evaluar Expresiones Booleanas

evaluar-booleano:

- **Entrada**: Un operador lógico (op) y una lista de argumentos booleanos (args).
- **Proceso**:
  - Define una función auxiliar evaluar-lista que aplica el operador lógico de manera recursiva a una lista de argumentos.
  - Usa cases para determinar qué operación lógica (AND, OR, XOR, NOT) aplicar basado en op.
- **Salida**: El resultado de aplicar la operación lógica especificada a los argumentos.

Ejemplos y salidas



3.0 Expresión while

- **Estructura:** (while-exp (boolean_exp body_exp) ...)
- **Proceso:**
    - Evalúa boolean_exp en el entorno env.
    - Si boolean_exp es verdadero, evalúa body_exp en el entorno env.
    - Luego, vuelve a evaluar la expresión while completa.
    - Si boolean_exp es falso, retorna 'void'.

Expresión for

- **Estructura:** (for-exp (var start-exp end-exp sum-exp body-exp) ...)
- **Proceso:**
    - Evalúa start-exp, end-exp y sum-exp en el entorno env.
    - Llama a for-eval con el cuerpo del bucle (body-exp), la variable de iteración (var), los valores evaluados de inicio (start), fin (end) y suma (sum), y el entorno env.

Expresion switch

- **Estructura:** (switch-exp test-exp (case1-exp result1-exp) ... (default-exp result-default-exp))
- **Proceso:**
    - Evalúa test-exp en el entorno env.
    - Compara el valor evaluado de test-exp con cada case-exp en la lista de casos.
    - Si encuentra una coincidencia, evalúa y devuelve result-exp correspondiente al case-exp coincidente.
    - Si no encuentra coincidencias, evalúa y devuelve result-default-exp.

Ejemplos y salidas

```
--> let x = 0 in while (x == 0) {x}
user break
  context...:
```

Entra en un bucle infinito

```
--> let x = 0 in for i from 0 until 10 by 10 do i
void
-->
```

4.0 Entorno de ligaduras y variables

Funcionamiento de let y los Entornos

Expresión let-exp

La expresión let-exp en Scheme permite la creación de un entorno local donde se pueden definir variables que solo son visibles dentro de dicho entorno. Aquí está el flujo general del proceso:

1. **Evaluación de las Expresiones de las Variables:**
    - rands: Estas son las expresiones que producen los valores a ser asignados a las variables.
    - eval-rands: Función que evalúa estas expresiones en el entorno actual (env), produciendo una lista de valores.
2. **Extensión del Entorno:**
    - extend-env: Crea un nuevo entorno extendido que incluye los identificadores (ids) y sus valores evaluados. (no permite
    - Este nuevo entorno incluye las asociaciones ids -> valores y referencia al entorno original (env).
3. **Evaluación del Cuerpo (body):**
    - El cuerpo de la expresión let-exp (body) se evalúa en el contexto del entorno extendido.
    - Esto asegura que las variables definidas en el let solo sean visibles dentro de este body.

Función decl-exp y Manejo de Entornos

La función decl-exp maneja las expresiones de declaración de variables (let-exp y lvar-exp).

- **let-exp:**
    - **IDs:** Identificadores de las variables locales.
    - **Rands:** Expresiones cuyos valores se asignan a los identificadores.
    - **Body:** El cuerpo de la expresión donde las variables locales son visibles.
    - **Flujo:**
        - Evalúa rands para obtener los valores de las variables locales.
        - Extiende el entorno actual con los nuevos bindings.
        - Evalúa body en el entorno extendido.
- **lvar-exp (Similar a let-exp):**
    - Implementa la misma lógica que let-exp, evaluando expresiones y extendiendo el entorno. (este maneja un extend-mod-env el cual permite modificaciones)

Entornos (environment)

Los entornos son estructuras de datos que mantienen las asociaciones de variables y sus valores.

- **Definiciones:**
  - empty-env: Representa un entorno vacío.
  - extend-env: Extiende un entorno con nuevas asociaciones de símbolos y valores, manteniendo una referencia al entorno original.

  - extend-mod-env: Extiende un entorno permitiendo modificaciones, donde las asociaciones de símbolos y valores se almacenan en vectores para poder actualizarse dinámicamente sin perder la referencia al entorno original.

Búsqueda de Variables

- apply-env:
  - Busca el valor de una variable en el entorno.
  - Utiliza buscador para recorrer las listas de identificadores y valores en extend-env.
  - Si no se encuentra la variable, se llama recursivamente en el entorno extendido.

Estado de la Implementación

- var: La implementación de var no está completamente definida en el código proporcionado. Parece que se espera un manejo similar a let, pero no se incluye toda la lógica necesaria.

Funcionamiento de match-exp y detector_patron en Scheme

Expresión match-exp

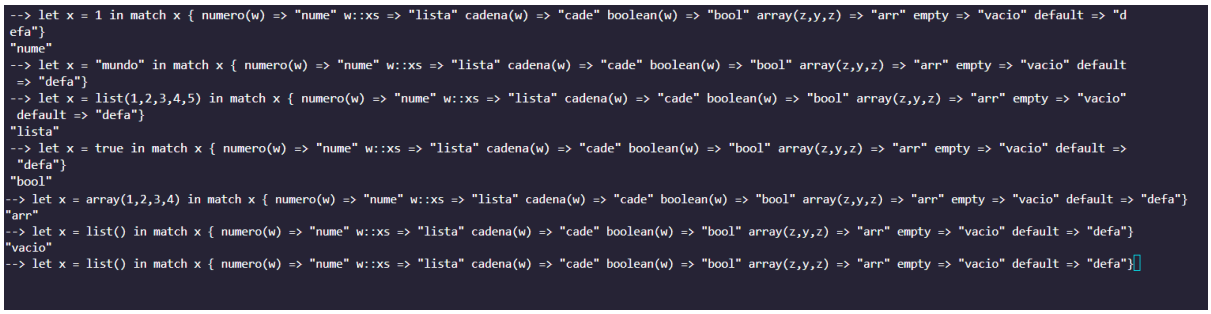
La expresión match-exp evalúa una expresión contra varios patrones y ejecuta el código correspondiente al primer patrón que coincide.

- **Estructura**: (match-exp (exp_var list_casos lista_exp) ...)
- **Proceso**:
  - Evalúa exp_var en el entorno env y almacena el resultado en valor.
  - Llama a detector_patron con valor, list_casos, lista_exp, y env para encontrar la coincidencia adecuada y ejecutar la expresión correspondiente.

5.0 Match (reconocimiento de patrones)

Esta función realiza la coincidencia de patrones y evalúa las expresiones asociadas.

1. **Coincidencia para Expresión Vacía (empty-match-exp)**:
   - Si valor es una lista vacía, evalúa la primera expresión en lista_exp.
   - Si no, sigue buscando coincidencia en los patrones restantes.
2. **Coincidencia para Lista (list-match-exp)**:
   - Si valor es una lista, extiende el entorno con la cabeza y la cola de la lista, y evalúa la primera expresión en lista_exp.
   - Si no, sigue buscando coincidencia en los patrones restantes.
3. **Coincidencia para Número (num-match-exp)**:
   - Si valor es un número, extiende el entorno con el identificador y el valor, y evalúa la primera expresión en lista_exp.
   - También maneja números en formato binario y hexadecimal.
   - Si no, sigue buscando coincidencia en los patrones restantes.
4. **Coincidencia para Cadena (cad-match-exp)**:
   - Si valor es una cadena y no es un número en formato especial, extiende el entorno con el identificador y el valor, y evalúa la primera expresión en lista_exp.
   - Si no, sigue buscando coincidencia en los patrones restantes.
5. **Coincidencia para Booleano (bool-match-exp)**:
   - Si valor es un booleano, extiende el entorno con el identificador y el valor, y evalúa la primera expresión en lista_exp.
   - Si no, sigue buscando coincidencia en los patrones restantes.
6. **Coincidencia para Arreglo (array-match-exp)**:
   - Si valor es un vector, lo convierte a una lista, extiende el entorno con los identificadores y los valores del vector, y evalúa la primera expresión en lista_exp.
   - Si no, sigue buscando coincidencia en los patrones restantes.
7. **Coincidencia por Defecto (default-match-exp)**:
   - Si no hay coincidencias con los patrones anteriores, evalúa la primera expresión en lista_exp.



6.0 Estructuras (struct)

Estructura:

- (struct-decl nombre lista-campos)
- (new-struct-exp identi lista_atributos)
- (get-struct-exp struc atributo)
- (set-struct-exp strucVar atributo nuevo_valor)

Proceso:
Declaración de Estructura:

- (struct-decl nombre lista-campos)
- Define una nueva estructura con nombre y lista-campos.
- Actualiza el entorno struct-env para incluir la nueva estructura.

Instanciación de Nueva Estructura:

- (new-struct-exp identi lista_atributos)
- Busca la estructura identi en struct-env.
- Verifica la longitud de lista_atributos con lista-campos.
- Evalúa los atributos y crea una instancia con los valores correspondientes.

Obtención de un Campo:

- (get-struct-exp struc atributo)
- Evalúa struc para obtener la instancia.
- Encuentra el índice del atributo en lista-campos.
- Devuelve el valor del campo correspondiente.

Modificación de un Campo:

- (set-struct-exp strucVar atributo nuevo_valor)
- Evalúa strucVar para obtener la instancia.
- Evalúa nuevo_valor.
- Encuentra el índice del atributo en lista-campos.
- Modifica el valor del campo en la estructura.

Estas definiciones permiten declarar una estructura, instanciarla, obtener y modificar el valor de un campo. En eval-expresion:

- new-struct-exp busca la estructura, evalúa los argumentos y crea una instancia.
- get-exp evalúa la estructura y obtiene el valor del campo.
- set-struct-exp evalúa la estructura y el nuevo valor, modificando el campo correspondiente.

Ejemplos y salidas

1.

```
-->  struct perro { nombre edad color } let t = new perro ("lucas", 10, "verde") in get t.nombre
"lucas"
-->
```

2.

```
--> struct perro {nombre edad color} let t = new perro ("lucas", 10, "verde") in begin set-struct t. nombre = "pepe";  get t.nombre end
"pepe"
-->
```

3.

```
--> struct perro {nombre edad color} let t = new perro ("lucas", 10, "verde") in set-struct t. nombre = "pepe"
#<void>
-->
```