Universidad del Valle

Fundamentos de Interpretación y Compilación de Lenguajes de Programación

Carlos Andrés Delgado Saavedra

Proyecto Final: Meta-lenguaje

Autores:
Juan Camilo Valencia - 2259459
Juan Pablo Escobar - 2259519
Edgar Andrés Vargas - 2259690
Juan Manuel Arango - 2259571

20 de junio de 2024

1. Valores numéricos

El lenguaje implementado soporta los siguientes tipos de valores numéricos:

- **Números binarios**: Representados con el prefijo `b` seguido de una secuencia de 0s y 1s. Por ejemplo, `b1010`.
- **Números decimales**: Representados como una secuencia de dígitos. Por ejemplo, `123`.
- **Números octales**: Representados con el prefijo `0x` seguido de dígitos del 0 al 7. Por ejemplo, `0x17`.
- **Números hexadecimales**: Representados con el prefijo `hx` seguido de dígitos del 0 al 9 y letras de la A a la F. Por ejemplo, `hx1A3F`.
- **Números flotantes**: Representados como una secuencia de dígitos con un punto decimal. Por ejemplo, `3.14`.

Para la implementación de valores numéricos no decimales, se verifica si el primer elemento del número corresponde al prefijo. En caso afirmativo, el número es positivo, de lo contrario, es negativo. En el caso de los número octales y hexadecimales, debe tenerse en cuenta que el prefijo contiene dos caracteres, 0x y hx respectivamente. Para la aplicación de primitivas, se convierten los argumentos a una base decimal común, se aplica la primitiva y, para finalizar, se revierte el resultado a la base original.

2. Asignaciones y Ligaduras

2.1. let

Para las declaraciones usando `let`, se transforman los valores en una lista y se evalúa el cuerpo de la expresión en un ambiente let (al que hemos llamado `extend-env-let`) que no permite la mutación de variables al guardar los valores en una lista.

2.2. var

Para las declaraciones usando `var`, se transforman los valores en una lista y se evalúa el cuerpo de la expresión en un ambiente extendido, que permite la mutación de variables guardando los valores en forma de vector.

3. Estructuras de Control

Se implementan las siguientes estructuras de control:

3.1. if

Para las expresiones condicionales, llamadas `if-exp`, puede utilizarse la implementación nativa de Racket. `if-exp` incluye una condición y dos expresiones que representan la condición a evaluar, la expresión a devolver en caso de que la condición sea `true` y la expresión a devolver en caso de que la condición sea `false`. Posteriormente, usamos el `if` nativo de Racket, que recibe la condición evaluada con `eval-exp` en el ambiente actual y devuelve la expresión adecuada evaluada también con `eval-exp`. Todas las expresiones son evaluadas en el ambiente actual.

3.2. while

Para la ejecución de estructuras `while`, se definió la función `eval-while`. Esta función construye una lista con los valores devueltos por la evaluación del cuerpo del `while` usando `eval-exp`. En caso de que la condición del `while` deje de cumplirse, se devuelve una lista vacía. Esto se realiza con el fin de tener registro del estado de la variable de control en cada iteración del ciclo. Finalmente, se utiliza la función `aux-for`, que devuelve el último elemento de la lista construida, es decir, el resultado final de las iteraciones.

3.3. for

Para el ciclo `for`, se crea un ambiente let nombrado environment (con el fin de aprovechar la modificación de variables), el cual luego es utilizado al llamar la función `eval-for`. Esta función evalúa los argumentos y guarda los valores evaluados en variables para obtener el valor desempaquetado. Posteriormente, se construye una lista de manera recursiva con los resultados de cada evaluación del cuerpo del ciclo con el fin de mantener registro del estado de las variables en cada iteración. Para finalizar, se llama la función `aux-for` para devolver el último elemento de la lista construida, que corresponde al estado en la última iteración.

3.4. switch

Se guardan en variables los elementos de la expresión utilizando `eval-exp`. Con estos valores, se llama a la función `eval-switch`, en la cual se recorren las listas de casos y valores paralelamente de forma recursiva, por lo tanto, si se encuentra un caso satisfactorio en la posición k de la lista de casos, se devuelve el valor en la posición k de la lista de valores. En caso de que la lista de casos sea nula, significa que se han consumido todos los casos sin hallar uno satisfactorio, por lo que se devuelve el caso por defecto, `default`.

4. Estructuras de Datos

4.1. Listas

Para la construcción de listas, se toman los elementos ingresados por el usuario de acuerdo a la gramática y se convierten en una lista de valores usando `map` y `eval-exp`. En cuanto a las primitivas de listas, se implementaron de la siguiente manera:

- **first:** Se verifica que la lista tenga elementos. En caso afirmativo, se devuelve la cabeza de la misma; de lo contrario, se lanza un error.
- **first:** Se verifica que la lista tenga elementos. En caso afirmativo, se devuelve la cola de la misma; de lo contrario, se lanza un error.
- **empty:** Se devuelve al valor booleano devuelto al aplicar la función `null?`.

4.2. Arrays

Para la construcción de arrays, se toman los elementos ingresados por el usuario de acuerdo a la gramática y se convierten en un vector de valores usando `map` y `eval-exp`. En cuanto a las primitivas de arrays, se utilizaron los métodos ofrecidos por `eopl` para la manipulación de vectores, pues esa es la representación interna de los arrays en nuestro lenguaje.

5. Pruebas de Ejecución

1. Aplicación de primitivas con diferentes bases.

```
-->(b1000 + b10)
"b1010"
-->(0x77 + 0x3)
"0x102"
-->(hx100 - hx2)
"hxfe"
-->(123.2 - 1.2)
122.0
-->(0x4 pow 0x2)
"0x20"
-->(100 mod 10)
0
```

2. Asignación con `var` y `let`

```
-->let a = 8 in (a + 20)
28
-->var a = 15 in begin set x = 32; x end
32
```

3. Estructuras `switch`, `for` y `while`

```
-->var x = 0 in begin while (x < 10) { set x = (x + 1)}; x end
10
-->var x = 0 in begin for i from 0 until 10 by 1 do set x = (x + i); x end
45
-->var x = 0 in begin switch (x) { case 1: 1 case 2: 2 default: 3} end
3
```

4. Operaciones sobre Arrays

```
-->let k = array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) in slice(k, 2, 5)
#(3 4 5 6)
-->let t = array(1, 2, 3, 4, 5) in length(t)
5
-->let t = array(1, 2, 3, 4, 5) in index(t, 2)
3
-->let t = array(1, 2, 3, 4, 5) in setlist(t, 2, 10)
#<void>
```

5. Operaciones sobre Listas

```
-->let l= cons(1 cons (2 cons (3 empty))) in list(first(l),rest(l),empty?(rest(l)))
(1 (2 3) #f)
```

6. Operaciones sobre strings

```
-->let a = "hola" b = "mundo" c = "cruel" in concat(a, b, c)
"holamundocruel"
-->let s = "hola mundo cruel" in string-length(s)
14
```

7. Definición y uso de funciones

```
-->var x = 10 f = func(a) (a + x) in let x = 30 in call f(10)
11
-->var f = func(x) x in call f(10)
10
```

8. Uso de condicionales

```
-->var x = 10 in if (x > 5) {1 else 2}
1
-->var x = 10 in if (x < 5) {1 else 2}
2
```