

Proyecto Final

Alejandro Sierra Betancourt

Juan Pablo Castaño Arango

Universidad del Valle

Ingeniería en Sistemas

Fundamentos De Interpretación Y Compilación De Lenguajes De Programación

Introducción del Proyecto

En este proyecto, se abordó el desafío de desarrollar un nuevo lenguaje de programación que unificara múltiples paradigmas de programación, tales como programación imperativa, orientada a objetos, y programación funcional. Este esfuerzo surgió como respuesta a la necesidad de la Universidad del Valle de integrar y optimizar el aprendizaje de diversos fundamentos de programación en un solo curso intensivo, denominado Meta Programación en modalidad bootcamp.

El equipo de trabajo se dedicó a diseñar y especificar un lenguaje de programación desde sus cimientos, abarcando desde la sintaxis y semántica hasta la implementación de primitivas y estructuras de control. Se definieron las especificaciones léxicas y gramaticales del lenguaje, incluyendo la forma de los números en distintas bases, textos, listas, y estructuras de datos más complejas como arrays y estructuras personalizadas.

Para la implementación, se eligió la librería SLLGEN, facilitando la creación de árboles de sintaxis abstracta y el manejo de valores expresados y denotados. Las primitivas numéricas, booleanas, y de manipulación de listas y arrays se diseñaron cuidadosamente para asegurar una correcta funcionalidad y rendimiento.

Implementación de let y var

let: Se utiliza para crear variables cuyas ligaduras no pueden ser modificadas una vez asignadas. La expresión `let` crea un entorno donde se evalúan las expresiones dentro de ese ámbito sin permitir modificaciones posteriores.

var: Se emplea para crear variables que pueden ser modificadas. A diferencia de `let`, `var` permite el uso de la operación `set` para cambiar el valor de las variables. Esta funcionalidad está soportada en el código mediante estructuras que extienden el entorno con nuevas ligaduras y permiten la asignación de nuevos valores.

Representación de binarios, decimales y octales

Para la representación de los números binarios, octales y hexadecimales no se implementó su representación.

Cómo se implementaron while, for y switch

while: Ejecuta un bloque de código mientras una condición sea verdadera. Se evalúa la condición antes de cada iteración y se ejecuta el cuerpo del bucle si la condición es verdadera.

for: Itera desde un valor inicial hasta un valor final, incrementando una variable de control en cada iteración y ejecutando un bloque de código en cada paso.

Evidencias Implementación

2.5. Datos

Los binarios se pueden expresar así:

```
b01010100  
-b01010101
```

Los decimales

```
23213  
-12312
```

Los octales

```
0x213345  
-0x23123
```

Los hexadecimales

```
hxFAB123  
-hx99EA
```

Los flotantes

```
412312.2312  
-23123.2312
```

Los booleanos

```
true  
false
```

Las cadenas de texto:

```
"hola mundo"  
"hola que tal"
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS  
  
>>> 242124  
242124  
>>> -242124  
-242124  
  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS ...  
  
>>> 3.1416  
3.1416  
>>> -3.1416  
-3.1416  
  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS ...  
  
>>> true  
#t  
>>> false  
#f  
>>> |
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS  
  
>>> "hola mundo"  
"hola mundo"  
>>> "hola que tal"  
"hola que tal"  
>>> |
```

2.7. Primitivas booleanas

Las primitivas booleanas esperan expresiones de valor expresado booleano, ejemplo:

```
and((1 > 2), false)
```

Observese que la primitiva numérica `>` retorna un booleano

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  ...

>>> and((1 > 2), false)
#f
>>> 
```

2.8. Listas

Las listas son agrupaciones de elementos, los cuales son inmutables (no pueden cambiar), se declaran de la siguiente forma:

Ejemplo:

```
list(1,2,3,4)
cons(1 cons(2 empty))
empty
```

Para la listas vamos a usar la representación de cabeza que es cualquier elemento y cola que es una lista.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  ...

>>> list(1,2,3,4)
(1 2 3 4)
>>>

>>> cons (1 cons(2 empty))
(1 2)
>>> empty
()
>>> 
```

2.9. Primitivas de listas

Para las listas tenemos las siguiente primitivas

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

Va a retornar (1,list(2,3), false)

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  ...  Racket Output  + v  []

>>> let
      l = cons(1 cons(2 cons(3 empty)))
      in
        list(first(),rest(l))

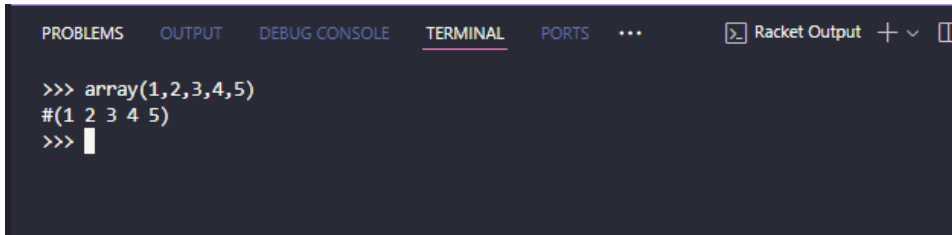
      list(first(l),rest(l), empty?(rest(l)))
(1 (2 3) #f)
>>> 
```

2.10. Arrays

Las arrays o arreglos son colecciones de listas ya que pueden cambiar.

Ejemplo:

```
array (1,2,3,4,5)
```



A screenshot of a Racket REPL terminal window. The window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (selected), PORTS, and a menu icon. The terminal shows the command `>>> array(1,2,3,4,5)` being entered, followed by the output `#(1 2 3 4 5)`. The prompt `>>>` is followed by a cursor.

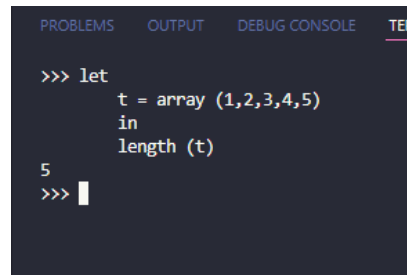
2.11. Primitivas de arrays

Las primitivas que tenemos son de acceso, modificación, longitud y slice.

Para conocer el tamaño tenemos `length`:

```
let
  t = array (1,2,3,4,5)
in
  length (t)
```

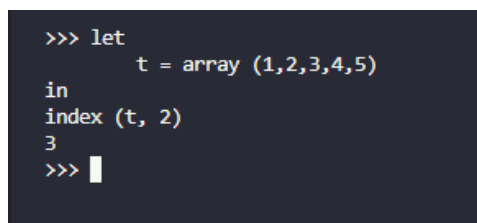
Debe retornar 5



A screenshot of a Racket REPL terminal window. The window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL (selected). The terminal shows the command `>>> let t = array (1,2,3,4,5) in length (t)` being entered, followed by the output `5`. The prompt `>>>` is followed by a cursor.

Para acceder a un elemento, el cual indexamos desde 0, tenemos `index`:

```
let
  t = array (1,2,3,4,5)
in
  index (t, 2)
```



A screenshot of a Racket REPL terminal window. The terminal shows the command `>>> let t = array (1,2,3,4,5) in index (t, 2)` being entered, followed by the output `3`. The prompt `>>>` is followed by a cursor.