Instituto Tecnológico y de
Estudios Superiores de Monterrey

Materia y Grupo

Implementación de métodos computacionales (Gpo 642)

Nombre de la Actividad:

Evidencia 2. Resaltador de sintaxis paralelo

Nombre y matrícula:

Rosa Vanessa Palacios Beltran | A01652612
Raúl Armando Vélez Robles | A01782488
Jesús Alexander Meister Careaga | A01656699

Profe. Jose Daniel Azofeifa Ugalde

29 de Mayo de 2023

Introducción

En esta entrega de evidencia se tiene como objetivo implementar algoritmos que representen información a través de estructuras abstractas de datos, organizar la información con algoritmos básicos, para entender el proceso de representación abstracta de datos a través de programación en los lenguajes de C++, Python y Racket.

Seleccionamos 3 lenguajes de programación de diferentes familias:

- **C++** que pertenece a la familia de lenguajes de programación C y es considerado un lenguaje de programación de alto nivel y bajo nivel.
- **Python** pertenece a la familia de lenguajes de programación de alto nivel. Es un lenguaje de programación interpretado, lo que significa que el código fuente se ejecuta directamente en la máquina anfitriona sin necesidad de compilar previamente.
- **Racket** pertenece a la familia de lenguajes de programación Lisp. Es un lenguaje de programación funcional, lo que significa que el énfasis está en las funciones y en la evaluación de expresiones, en lugar de las instrucciones y los procedimientos.

Categorías léxicas de C++:

- **Procedimientos:**
  - Funciones
  - Métodos
  - Constructores
  - Destructores
- **Condicionales:**
  - if/else
  - switch/case
- **Ciclos:**
  - for
  - while
  - do/while
  - range-based for
- **Tipos de datos:**
  - int
  - float
  - double
  - char
  - bool
  - void
  - enum
  - struct
  - class
  - union
  - typedef

    - auto
    - decltype
- **Caracteres Especiales:**
    - Punto (.)
    - Coma (,)
    - Punto y coma (;)
    - Dos puntos (:)
    - Paréntesis ( )
    - Corchetes [ ]
    - Llaves { }
- **Palabras reservadas (clave):**
    - if
    - else
    - switch
    - case
    - default
    - while
    - do
    - for
    - break
    - continue
    - return
    - const
    - static
    - volatile
    - virtual
    - public
    - private
    - protected
    - friend
    - class
    - struct
    - enum
    - namespace
    - using
    - typedef
    - auto
    - sizeof
    - new
    - delete
    - true
    - false
- **Operadores:**

    - Aritméticos: +, -, *, /, %
    - Asignación: =
    - Comparación: ==, !=, >, <, >=, <=
    - Lógicos: &&, ||, !
    - Bitwise: &, |, ^, ~, <<, >>
    - Ternario: ?:
    - Incremento/decremento: ++, --
- **Literales:**
    - Enteros: 1, 2, 3, ...
    - Reales: 3.14, 1.0, 2.5, ...
    - Caracteres: 'a', 'b', 'c', ...
    - Cadenas de caracteres: "hello", "world", ...
- **Comentarios:**
    - // Comentario de línea
    - /* Comentario de bloque */

Categorías léxicas de Python:

- **Procedimientos:**
    - Funciones
    - Métodos
    - Generadores
    - Decoradores
    - Clases
- **Condicionales:**
    - if/else
    - elif
    - try/except/finally
- **Ciclos:**
    - for
    - while
- **Tipos de datos:**
    - int
    - float
    - complex
    - bool
    - str
    - bytes
    - bytearray
    - tuple
    - list
    - set
    - frozenset
    - dict

    - None
    - Ellipsis
- **Caracteres Especiales:**
    - Paréntesis ( )
    - Corchetes [ ]
    - Llaves { }
    - Coma (,)
    - Dos puntos (:)
    - Punto (.)
    - Barra invertida (\)
    - Comillas simples (')
    - Comillas dobles (")
    - Signo de porcentaje (%)
    - Arroba (@)
- **Palabras reservadas (clave):**
    - if
    - else
    - elif
    - while
    - for
    - in
    - try
    - except
    - finally
    - raise
    - assert
    - def
    - return
    - lambda
    - class
    - yield
    - import
    - from
    - as
    - global
    - nonlocal
    - True
    - False
    - None
    - with
    - pass
    - break
    - continue

    - del
    - not
    - and
    - or
- **Operadores:**
    - Aritméticos: +, -, *, /, %, **, //
    - Asignación: =
    - Comparación: ==, !=, >, <, >=, <=
    - Lógicos: and, or, not
    - Bitwise: &, |, ^, ~, <<, >>
    - Ternario: expr1 if condition else expr2
    - Incremento/decremento: no es aplicable en Python
- **Literales:**
    - Enteros: 1, 2, 3, ...
    - Reales: 3.14, 1.0, 2.5, ...
    - Complejos: 3+4j, 1j, ...
    - Caracteres: 'a', 'b', 'c', ...
    - Cadenas de caracteres: "hello", "world", ...
    - Bytes: b'hello'
    - Listas: [1, 2, 3]
    - Tuplas: (1, 2, 3)
    - Conjuntos: {1, 2, 3}
    - Diccionarios: {'key1': 'value1', 'key2': 'value2', ...}
    - None: None
- **Comentarios:**
    - # Comentario de línea
    - ''' Comentario de bloque '''

Categorías léxicas de Racket:

- **Procedimientos:**
    - Function
    - Macros
- **Condicionales:**
    - if/else
    - cond
    - case
- **Ciclos:**
    - for
    - while
    - do
    - repeat
- **Tipos de datos:**
    - Number

    - String
    - Boolean
    - Character
    - Symbol
    - Pair
    - List
    - Vector
    - Bytevector
    - Hash table
    - Procedure
    - Input/output port
    - Thread
    - Promise
    - Continuation
    - Syntax object
- **Caracteres Especiales:**
    - Paréntesis ( )
    - Corchetes [ ]
    - Llaves { }
    - Coma (,)
    - Dos puntos (:)
    - Punto (.)
    - Comillas simples (')
    - Comillas dobles (")
    - Signo de porcentaje (%)
    - Barra invertida (\)
    - Arroba (@)
    - Símbolo de interrogación (?)
    - Símbolo de exclamación (!)
    - Símbolo de igual (=)
- **Palabras reservadas (clave):**
    - define
    - lambda
    - let
    - let*
    - letrec
    - cond
    - case
    - if
    - and
    - or
    - not
    - quote

    - quasiquote
    - unquote
    - unquote-splicing
    - define-syntax
    - let-syntax
    - letrec-syntax
    - syntax-rules
    - else
    - begin
    - set!
    - require
    - provide
    - namespace
- **Operadores:**
    - Aritméticos: +, -, *, /, %
    - Comparación: =, <, >, <=, >=, !=
    - Lógicos: and, or, not
    - Bitwise: bitwise-and, bitwise-or, bitwise-xor, bitwise-not, arithmetic-shift, bitwise-shift
    - Asignación: =
    - Ternario: no es aplicable en Racket
- **Literales:**
    - Enteros: 1, 2, 3, ...
    - Reales: 3.14, 1.0, 2.5, ...
    - Caracteres: #\a, #\b, #\c, ...
    - Cadenas de caracteres: "hello", "world", ...
    - Listas: (1 2 3), (a b c), ...
    - Vectores: #(1 2 3), #("hello" "world"), ...
    - Booleans: #t, #f
    - Símbolos: 'a, 'b, 'c, ...
    - Vacío: '()
- **Comentarios:**
    - ; Comentario de línea
    - #| Comentario de bloque |#

Categorías léxicas que tienen en **común** los lenguajes de programación que seleccionamos son:

- **Condicionales:**
    - if/else
- **Ciclos:**
    - for
    - while
- **Tipos de datos:**

    - Números enteros
    - Números reales
    - Cadenas de caracteres
- **Caracteres Especiales:**
    - Paréntesis ( )
    - Corchetes [ ]
    - Llaves { }
    - Coma (,)
    - Comillas simples (')
    - Comillas dobles (")
- **Palabras reservadas (clave):**
    - if
    - else
    - for
    - while
- **Operadores:**
    - Aritméticos: +, -, *, /
    - Comparación: =, <, >, <=, >=
    - Asignación: =
- **Literales:**
    - Enteros: 1, 2, 3, ...
    - Reales: 3.14, 1.0, 2.5, ...
    - Caracteres: 'a', 'b', 'c', ...
    - Cadenas de caracteres: "hello", "world", ...
- **Comentarios:**
    - No hay en común

Notación basada en **expresiones-s** para representar **expresiones regulares**

C (**C++**), P (**Python**), R (**Racket**)

C={(“)|(‘)|(.)|(,)|()|(0)|(1)|(2)|(3)|(4)|(5)|(6)|(7)|(8)|(9)|(+)|(++)|(-)|(--)|(&&)|(%)|(||)|(!)|(*)|(/)|(**)|(//)|(=)|(<)|(<<)|(>>)|(>)|(<=)|(>=)|(!=)|(//)|(/)|(/)|(if)|(else)|(switch)|(case)|(default)|(while)|(do)|(for)|(break)|(continue)|(return)|(const)|(static)|(volatile)|(virtual)|(public)|(private)|(protected)|(friend)|(class)|(struct)|(enum)|(namespace)|(using)|(typedef)|(auto)|(sizeof)|(new)|(delete)|(true)|(false)}*

P={(while)|(for)|(if)|(else)|(")|(')|(.)|(,)|(elif)|(try)|(except)|(finally)|(int)|(float)|(complex)|(bool)|(str)|(bytes)|(bytearray)|(tuple)|(list)|(set)|(frozenset)|(dict)|(Ellipsis)|{(}|{)}|({)|(})|([)|(])|()|(,)|(:)|(;)|(.)|(\)|(%)|(@)|(raise)|(assert)|(def)|(return)|(lambda)|(class)|(yield)|(import)|(from)|(as)|(global)|(nonlocal)|(True)|(False)|(None)|(with)|(pass)|(break)|(continue)|(del)|(not)|(and)|(or)|(+)|(-)|(*)|(/)|(**)|(//)|(=)|(<)|(>)|(<=)|(>=)|(!=)|(&)|(|)|(^)|(<<)|(>>)|(0)|(1)|(2)|(3)|(4)|(5)|(6)|(7)|(8)|(9)}*

R={(while)|(for)|(if)|(else)|(")|(')|(.)|(,)|(cond)|(case)|(for)|(while)|(do)|(repeat)|(define)|(lambda)|(let)|(let*)|(letrec)|(cond)|(case)|(if)|(and)|(or)|(not)|(quote)|(quasiquote)|(unquote)|(unquote-splicing)|(define-syntax)|(let-syntax)|(letrec-syntax)|(syntax-rules)|(else)|(begin)|(set!)|(require)|(provide)|(namespace)|(+)|(-)|(*)|(/)|(%)|(=)|(<)|(>)|(<=)|(>=)|(!=)|(#|)|(|#)|(and)|(or)|(not)|(0)|(1)|(2)|(3)|(4)|(5)|(6)|(7)|(8)|(9)}*

C∩P∩R={(while)|(for)|(if)|(else)|(")|(')|(.)|(,)|()|(0)|(1)|(2)|(3)|(4)|(5)|(6)|(7)|(8)|(9)|(+)|(-)|(*)|(/)|(>)|(<)|(=)|( )}*

Algoritmo implementado

- Se lee un archivo en forma de string
- Se separa el string por palabras, símbolos, espacios y cambios de linea, y se almacena en una lista
- Se analiza cada elemento en la lista para ser resultado en dado caso
- Se une la lista
- Se manda a un archivo html

Implementación de la concurrencia: La concurrencia se implementa en dos partes del programa:

- Se analiza una lista de archivos, la lista es dividida en 4 futures, que se unen al terminar su parte.
- Cada archivo analizado es dividido en 4 futures, que se unen al terminar su parte.

Cálculo del speedup obtenido

Para la versión final se realizaron 2 cálculos de speedup (Tiempos de ejecución entre las dos versiones). En ambos casos se probó con 10 archivos de 1 millón de palabras cada uno, un total de 10 millones de palabras.

6 núcleos:

```
cpu time: 25484 real time: 93884 gc time: 12843
'("Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto")
cpu time: 29593 real time: 85015 gc time: 14000
'("Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto")
```

$S_6 \approx 1.1$

8 núcleos:

```
cpu time: 103043 real time: 110081 gc time: 14376
'("Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto")
cpu time: 111172 real time: 91201 gc time: 16693
'("Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto" "Correcto")
```

$S_8 \approx 1.2$

Pruebas positivas

1.-

```
and  break continue elseif else for if in or return select while
 if else for while do int float double boolean char void public
bool var const let function return class

ifelse for while222 define lambda cond let and or not int float double void string struct
    if var else const let return function define lambda def and or condnot letrec
elif while repeat until end begin procedure var function float doubleint

do while repeat if else break reeturn until fn let const for continue int string

+ - * / // & && | || ! == = < > <> <= >= =! " hola+/*=else while ifelse +" "" 'a' # $ % ( ) () ' { } {} [] [ ] , ; :
1 2 3 4 5 6 7 8 9 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
and  break continue elseif else for if in or return select while
 if else for while do int float double boolean char void public
bool var const let function return class

ifelse for while222 define lambda cond let and or not int float double void string struct
    if var else const let return function define lambda def and or condnot letrec
elif while repeat until end begin procedure var function float doubleint

do while repeat if else break reeturn until fn let const for continue int string

+ - * / // & && | || ! == = < > <> <= >= =! " hola+/*=else while ifelse +" " # $ % ( ) () ' { } {} [] [ ] , ; :
1 2 3 4 5 6 7 8 9 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a_2
```

2.-

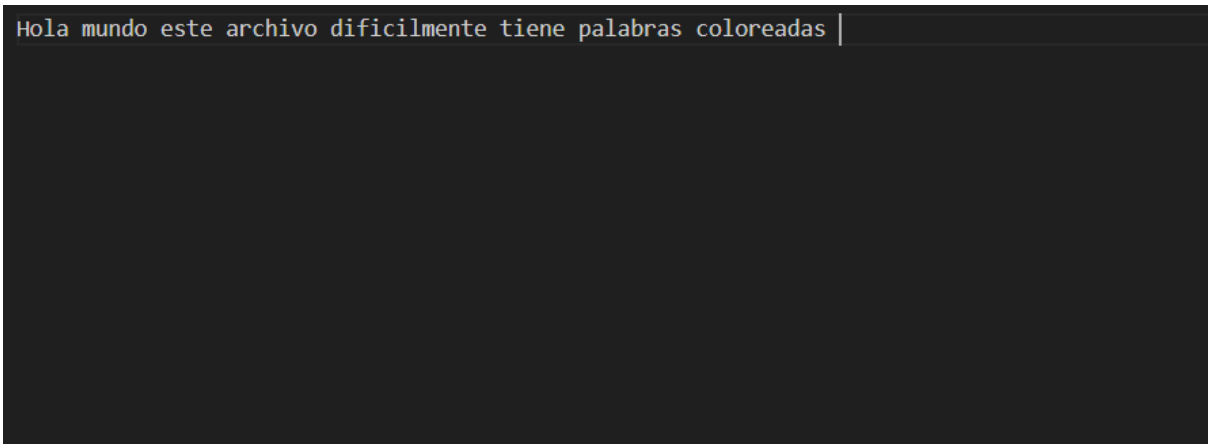
```python
import random

def escribir_palabras_azar(lista, n, archivo):
    # Abre el archivo en modo escritura
    with open(archivo, "w") as f:
        # Repite n veces
        for i in range(n):
            # Elige una palabra al azar de la lista
            palabra = random.choice(lista)
            # Escribe la palabra en el archivo con un salto de línea
            f.write(palabra + "\n")

# Ejemplo de uso
lista = ["while", "for", "+", "-", "hola"]
n = 1000000
archivo = "largo.txt"
escribir_palabras_azar(lista, n, archivo)
```

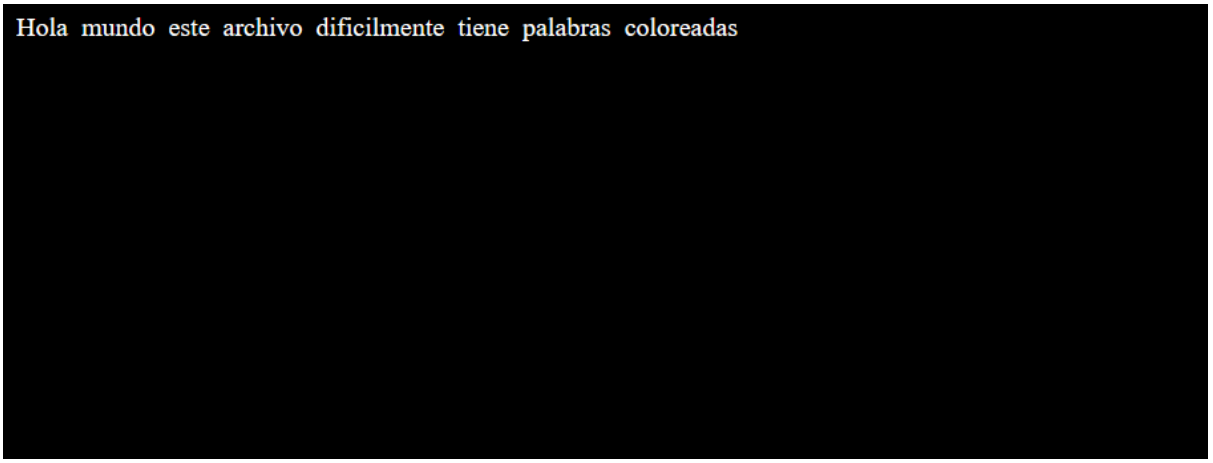
```python
import random

def escribir_palabras_azar(lista, n, archivo):
    # Abre el archivo en modo escritura
    with open(archivo, "w") as f:
        # Repite n veces
        for i in range(n):
            # Elige una palabra al azar de la lista
            palabra = random.choice(lista)
            # Escribe la palabra en el archivo con un salto de línea
            f.write(palabra + "\n")

# Ejemplo de uso
lista = ["while", "for", "+", "-", "hola"]
n = 1000000
archivo = "largo.txt"
escribir_palabras_azar(lista, n, archivo)
```

Pruebas negativas

1.- Archivo vacío

2.- Nada que resaltar





Complejidad del algoritmo

El algoritmo realizado tiene una complejidad de O(n) donde n es el número de caracteres que contiene el archivo a analizar. Esto se debe a que la función principal detrás de nuestra

implementación es la función map y la función regexp-split, que por sí solas tiene una complejidad de O(n), y que aplica una función de complejidad O(1) a cada elemento de una lista, por lo que la complejidad es O(n), donde n es el número de caracteres que tiene el archivo a analizar, pues la función más pesada es regexp-split, que separa por caracteres de un string.

En esta implementación se utilizó una versión paralela del algoritmo, y si bien la implementación del paralelismo reduce el tiempo de ejecución, este no cambia la complejidad temporal del algoritmo, pues este es el mismo algoritmo, dividido en secciones que se ejecutan de manera paralela, en lugar de manera secuencial, esto quiere decir que al ser el mismo algoritmo, su complejidad es la misma, en este caso O(n).

Reflexión

Reflexión del código: En este código, a diferencia de la primera entrega, implementamos el uso del paralelismo en racket para poder optimizar el tiempo de ejecución del algoritmo. La implementación principal del paralelismo en esta entrega se realizó usando separando en 4 futures la lista de archivos a analizar, y a su vez cada archivo analizado también sería dividido en 4 futures. Es importante resaltar que el número de futures utilizado es un número arbitrario, elegido considerando la complejidad del código. En un código de alta complejidad y gran cantidad de datos, es más recomendable usar un mayor número de futures, mientras que en codigos poco complejos, usar grandes números de futures puede resultar contraproducente para el tiempo de ejecución, pues si las tareas a ejecutar son demasiado sencillas, el tiempo añadido por los futures puede ser superior a la ejecución de las mismas. Es por esto que el speedup no es tan notorio, pues el programa realiza operaciones sencillas, por lo que el uso del paralelismo es apenas beneficioso para el tiempo de ejecución, sin embargo es importante resaltar que un mayor de número de elementos también mejorarían el speedup obtenido.

Conclusión

El trabajo que se realizó en este proyecto es un resaltador de sintaxis en el cuál se puede identificar por medio de colores las diferentes categorías de léxicas, en este caso para los lenguajes de C++, Python y Racket. El algoritmo se implementó en Racket y para esta última entrega se usó el paralelismo en una función que divide los datos en hilos separados y combinados los resultados al final.

En cuanto a las implicaciones éticas, al diseñar un resaltador de texto, asumimos la responsabilidad de garantizar su correcto funcionamiento, ya que los usuarios confían en esta herramienta sin ponerla a prueba con frecuencia. Cualquier fallo podría llevar a errores por parte de los usuarios y dificultades la búsqueda y detección de los mismos. Además, muchos programadores dependen tanto de esta aplicación tan sencilla que les resultaría complicado programar sin ella.

El desarrollo de un resaltador de sintaxis es un proyecto que busca facilitar la lectura y comprensión de código fuente en diferentes lenguajes de programación. Un resaltador de sintaxis utiliza colores, estilos y formatos para distinguir las palabras clave, los identificadores, los comentarios y otros elementos del código. Esta tecnología puede tener implicaciones éticas en la sociedad, ya que puede influir en la forma en que los programadores escriben, aprenden y comparten su código. Por ejemplo, un resaltador de sintaxis puede favorecer ciertos lenguajes o estilos de programación sobre otros, o puede ocultar errores o vulnerabilidades en el código. Por lo tanto, es importante que los desarrolladores de un resaltador de sintaxis sean conscientes de estos posibles efectos y que busquen crear una herramienta que sea útil, accesible y respetuosa con la diversidad y la integridad del código.

Así mismo, dado que existen numerosos lenguajes de programación en la actualidad, es crucial poder identificar y diferenciar ciertas palabras clave de otras. Por esta razón, un resaltador de sintaxis es una herramienta clave para facilitar la lectura y comprensión de código fuente en diferentes lenguajes de programación. Al utilizar colores, estilos y formatos para distinguir los diferentes elementos del código, los programadores pueden trabajar de manera más eficiente y efectiva.

Referencias:

Memorias CIMTED. (2021). Programación de datos con Racket. Recuperado de http://memoriascimted.com/wp-content/uploads/2021/04/Programaci%C3%B3n-de-datos-con-Racket.pdf

Pereira, R. N., Couto, M., Ribeiro, F. W., Rua, R., Cunha, J., Fernandes, J. B., & Saraiva, J. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? *Software Language Engineering*. https://doi.org/10.1145/3136014.3136031

Racket Documentation. (s. f.). https://docs.racket-lang.org/