

***Universidad Tecnológica Nacional - Facultad Regional Buenos Aires***

**Sintaxis y Semántica de los Lenguajes**

**2023**

**Trabajo Práctico Grupal N°1**

**Curso:** K2006

**Profesor:** Roxana Leituz

**Ayudantes:** -

**Repositorio:** https://github.com/gianpetrii/TP-Investigacion-Grupal---Sintaxis-Grupo-3.git

**Fecha Estipulada de Entrega:** 26-8-2023

**Grupo N°3**

| **Alumno** | **Legajo** |
| --- | --- |
| Petri Gianluca | 1776990 |
| Munilla Lucila Macarena | 1718411 |
| Ponce de León Alejo Gabriel | 2118609 |
| Mateo Fernandez Cruz | 2099860 |

**Consigna**

Investigar dos lenguajes de programación excluidos: C y C++, buscar las BNF del lenguaje elegido y desarrollar una presentación sobre los siguientes temas:

• Breve historia del lenguaje (“BREVE”)

• Comparación de rendimiento (llamado usualmente benchmark) de dos algoritmos a elección entre los dos lenguajes, por ejemplo método de búsqueda en colecciones, método de ordenamiento, etc y una reseña del modo en que trabajan los mismos para justificar el mayor o menor rendimiento.

El rendimiento debe medirse en ambos lenguajes, es decir deben crear algún programa mínimo para mostrar las mediciones de cada uno.

La presentación(ppt o similar) debe ser un resumen con las ideas principales, este trabajo se aprueba con la presentación del mismo de manera presencial, todos los integrantes deben exponer.

Pueden agregar código y ejemplos prácticos. La presentación debe finalizar con un breve set de preguntas para sus compañeros.

La entrega debe constar de: la BNF, el entregable completo, la presentación (ppt) resumen y los ejemplos en caso de haberlos.

IMPORTANTE! La presentación debe ser de 15 a 20 minutos como máximo, no se debe superar ese plazo, se evaluará este objetivo.

Los pares de lenguajes no deben repetirse en el curso, queda a elección de ustedes cuales usar pero no deben haber dos iguales, la coordinación está a cargo de los alumnos.

LOS TRABAJOS SE DEBEN SUBIR A GIT.

**Resolucion**

**Breve Historia de Prolog:**

Prolog (Programming Logic) es un lenguaje de programación declarativo utilizado para la programación lógica y la inteligencia artificial. Su historia comienza a finales de los 60. Debajo se da un resumen de su evolución a lo largo de los años:

* Década de 1960:
  + Prolog nace de la lógica de predicados, un campo de la lógica matemática que se centra en la representación y el razonamiento sobre relaciones entre objetos. Algunos investigadores, como Alain Colmerauer y Robert Kowalski, comenzaron a explorar cómo usar la lógica de predicados para la programación.
* Década de 1970:
  + En la década de 1970, Alain Colmerauer y Philippe Roussel desarrollaron el lenguaje de programación Prolog en la Universidad de Aix-Marseille, en Francia. Prolog fue diseñado para expresar problemas en términos de relaciones y reglas lógicas, en lugar de los pasos computacionales comunes de la época.
* Década de 1980:
  + Durante los años 80, Prolog ganó popularidad en la comunidad de inteligencia artificial (IA). La capacidad de Prolog para realizar inferencias lógicas y búsqueda en espacios de posibilidades lo hizo útil para problemas como el procesamiento de lenguaje natural, la resolución de problemas y la representación de conocimiento.
* Década de 1990:
  + A medida que Prolog ganaba terreno, se desarrollaron diversas implementaciones y extensiones del lenguaje. También se hicieron esfuerzos para mejorar el rendimiento de Prolog, lo que resultó en optimizaciones y nuevas técnicas de compilación.
* Década Actual (2020 - actualidad):
  + Aunque Prolog no es tan popular como algunos lenguajes más convencionales, sigue siendo utilizado en áreas específicas de la inteligencia artificial y la programación lógica. Su enfoque en la representación de relaciones y razonamiento lógico lo hace valioso para resolver problemas que se adaptan a su paradigma. Por esta última característica se lo utiliza también en ámbitos educativos para mostrar la diferencia entre distintos paradigmas de programación.[[1]](#footnote-0)

**Breve Historia de Javascript:**

JavaScript es uno de los lenguajes de programación más utilizados en la actualidad, se usa principalmente para agregar interactividad y dinamismo a las páginas web. Su historia comienza alrededor del 1990 y debajo se detalla una breve evolución del mismo:

* Década de 1990:
  + A principios de los años 90, la web estaba compuesta principalmente por contenido estático y enlaces hipertexto. Para hacer que las páginas fueran más interactivas, Netscape Communications Corporation (una de las primeras empresas de navegadores web) lanzó Netscape Navigator. Para ello, crearon un lenguaje de scripting llamado "LiveScript".
  + Posteriormente, Netscape se asoció con Sun Microsystems (ahora parte de Oracle) y cambió el nombre del lenguaje a "JavaScript", aprovechando el renombre de Java en ese momento. JavaScript fue lanzado en Netscape Navigator 2.0 en 1995.
* Década de 2000:
  + A medida que la web creció, se hizo evidente que era necesario estandarizar JavaScript. La European Computer Manufacturers Association (ECMA) estableció un comité para definir un estándar para el lenguaje. De acá surge ECMAScript, con su primera edición publicada en 1997. Desde entonces, JavaScript se convirtió en una implementación específica del estándar ECMAScript.
* Década de 2010:
  + Con el aumento de la velocidad de las conexiones a Internet y la mejora de las tecnologías de navegadores, las aplicaciones web comenzaron a ganar popularidad. Frameworks como Angular, React y Vue aparecieron para facilitar la creación de aplicaciones web más complejas, completas y dinámicas.
* Década Actual (2020 - actualidad):
  + JavaScript ya trascendió su papel original en los navegadores y se convirtio en un lenguaje de programación versátil que se utiliza en diversos entornos, como desarrollo de aplicaciones móviles, desarrollo de servidores (Node.js), dispositivos electronicos para dar “inteligencia” a los aparatos (IoT) y más. Aunque su fuerte y dominancia sigue siendo en desarrollo de paginas web[[2]](#footnote-1)

**BNFs genérica para el lenguaje de programación Prolog:**

*<programa\_prolog> ::= <sentencias\_prolog>.*

*<sentencias\_prolog> ::= <sentencia\_prolog> | <sentencia\_prolog>, <sentencias\_prolog>.*

*<sentencia\_prolog> ::= <hecho\_prolog> | <regla\_prolog>.*

*<hecho\_prolog> ::= <predicado\_prolog>.*

*<regla\_prolog> ::= <predicado\_prolog> :- <cuerpo\_prolog>.*

*<cuerpo\_prolog> ::= <predicado\_prolog> | <predicado\_prolog>, <cuerpo\_prolog>.*

*<predicado\_prolog> ::= <nombre> (<argumentos\_prolog>).*

*<argumentos\_prolog> ::= <termino\_prolog> | <termino\_prolog>, <argumentos\_prolog>.*

*<termino\_prolog> ::= <nombre> | <variable\_prolog>.*

*<variable\_prolog> ::= <mayuscula>, <resto\_de\_caracteres>.*

*<resto\_de\_caracteres> ::= 1234567890qwertyuiopasdfghjklzxcvbnm\_*

**BNFs genérica para el lenguaje de programación Javascript:**

*<programa\_js> ::= <sentencias\_js>*

*<sentencias\_js> ::= <sentencia\_js> | <sentencia\_js>, <sentencias\_js>*

*<sentencia\_js> ::= <declaracion\_js> | <asignacion\_js> | <llamada\_funcion\_js>*

*<declaracion\_js> ::= var <variable\_js> = <expresion\_js>;*

*<asignacion\_js> ::= <variable\_js> = <expresion\_js>;*

*<llamada\_funcion\_js> ::= <nombre\_funcion\_js> ( <argumentos\_js> );*

*<argumentos\_js> ::= <expresion\_js> | <expresion\_js>, <argumentos\_js>*

*<expresion\_js> ::= <numero> | <cadena> | <booleano> | <variable\_js>*

*<nombre\_funcion\_js> ::= <nombre> | <nombre\_funcion\_js>.<nombre>*

*<variable\_js> ::= <letra>, <resto\_de\_caracteres>*

*<resto\_de\_caracteres> ::= 1234567890qwertyuiopasdfghjklzxcvbnm\_*

**Comparación de Rendimiento de Algoritmos:**

Método de búsqueda en colecciones - Búsqueda Binaria:

La búsqueda binaria es un método eficiente para encontrar un elemento en una colección ordenada. Divide repetidamente la colección en mitades y compara el valor buscado con el elemento en el medio.

Implementación en Prolog:

*buscar\_binaria(\_, [], \_, \_) :- fail.*

*buscar\_binaria(Elemento, [Elemento|\_], \_, \_) :- !.*

*buscar\_binaria(Elemento, Lista, Inicio, Fin) :-*

*Mitad is (Inicio + Fin) // 2,*

*nth0(Mitad, Lista, ValorMedio),*

*(Elemento = ValorMedio -> true;*

*Elemento < ValorMedio -> buscar\_binaria(Elemento, Lista, Inicio, Mitad - 1);*

*buscar\_binaria(Elemento, Lista, Mitad + 1, Fin)).*

*% Crear un array grande ordenado para prueba*

*tamano\_coleccion(10000000). % Uso como ejemplo coleccion con 10.000.000 de elementos*

*crear\_coleccion(Coleccion) :-*

*tamano\_coleccion(Tamano),*

*numlist(0, Tamano, Coleccion).*

*% El Elemento a buscar*

*elemento\_a\_buscar(Elemento) :-*

*tamano\_coleccion(Tamano),*

*%Elemento is 1.*

*%Elemento is 9999999.*

*Elemento is 5000000.*

*% Medir tiempo de ejecución*

*medir\_tiempo(Ejecucion) :-*

*crear\_coleccion(Coleccion),*

*elemento\_a\_buscar(Elemento),*

*get\_time(Inicio),*

*tamano\_coleccion(Tamano),*

*buscar\_binaria(Elemento, Coleccion, 0, Tamano),*

*get\_time(Fin),*

*Ejecucion is Fin - Inicio,*

*write('Resultado para Prolog'), nl,*

*write('Elemento encontrado: true'), nl,*

*write('Tiempo de ejecución: '), write(Ejecucion), write(' segundos'), nl.*

Implementación en Javascript:

*function buscarBinaria(elemento, array, inicio = 0, fin = array.length - 1) {*

*if (inicio > fin) {*

*return false;*

*}*

*const medio = Math.floor((inicio + fin) / 2);*

*if (array[medio] === elemento) {*

*return true;*

*} else if (array[medio] > elemento) {*

*return buscarBinaria(elemento, array, inicio, medio - 1);*

*} else {*

*return buscarBinaria(elemento, array, medio + 1, fin);*

*}*

*}*

*// Crear un array grande ordenado para prueba*

*const tamañoColeccion = 10000000; // Uso como ejemplo coleccion con 10.000.000 de elementos*

*const coleccion = Array.from({ length: tamañoColeccion }, (\_, i) => i);*

*// El elemento a buscar*

*const elementoABuscar = 1;*

*//const elementoABuscar = 9999999;*

*//const elementoABuscar = 5000000;*

*// Con esto mido el tiempo de ejecucion NO EL DE COMPILACION*

*const inicio = performance.now();*

*const encontrado = buscarBinaria(elementoABuscar, coleccion);*

*const fin = performance.now();*

*console.log("Resultado para Javascript");*

*console.log(`Elemento encontrado: ${encontrado}`);*

*console.log(`Tiempo de ejecución: ${fin - inicio} ms`);*

Para ambos códigos se realizaron pruebas con una colección ordenada de 10 millones de elementos, debajo se muestran los rendimientos para la búsqueda de valores en distintas posiciones en la lista:

Para el valor <1>:

Tiempo de ejecución Prolog: 3.0671834945678712 ms

Tiempo de ejecución Javascript: 0.07370001077651978 ms

Para el valor <9.999.999>:

Tiempo de ejecución Prolog: 71.20349407196045 ms

Tiempo de ejecución Javascript: 0.07719999551773071 ms

Para el valor <5.000.000>:

Tiempo de ejecución Prolog: 1.557612419128418 ms

Tiempo de ejecución Javascript: 0.07559999823570251 ms

Primero que nada es importante recalcar que ambos tiempos son de ejecución y no se están contemplando los de compilación ya que entendemos que eso es lo que se quiere medir,de todas formas se menciona que prolog se compila por un lado y se puede ejecutar las veces que queramos luego de ser compilado ya que son instancias independientes, mientras que javascript normalmente se lo debe compilar cada vez que se lo quiera ejecutar. Esto hace que si aumentamos el tamaño de nuestras colecciones, javascript terminaría pasando a tardar mucho más ya que debería generar nuestra colección cada vez que se quiera ejecutar y por más que la búsqueda en sí fuera rápida el tiempo total sería mucho mayor.

Sacando esta observación de lado podemos ver que los tiempos de ejecución en Javascript son varios dígitos menores, al punto que apenas se percibe una diferencia entre los distintos valores de consultas, mientras que para Prolog los tiempos son claramente mayores pero también con una diferencia definida entre los valores de las distintas consultas, acordes a lo que uno esperaría.

Método de ordenamiento - Mergesort

El algoritmo MergeSort divide la lista en partes más pequeñas, las ordena y luego las combina para obtener una lista ordenada completa. A diferencia del QuickSort, MergeSort divide la lista por la mitad en cada paso y luego combina las mitades ordenadas.

Implementación en Prolog:

*merge\_sort([], []).*

*merge\_sort([X], [X]).*

*merge\_sort(Lista, Ordenada) :-*

*length(Lista, Len),*

*Len1 is Len // 2,*

*length(Izquierda, Len1),*

*append(Izquierda, Derecha, Lista),*

*merge\_sort(Izquierda, OrdenadaIzq),*

*merge\_sort(Derecha, OrdenadaDer),*

*merge(OrdenadaIzq, OrdenadaDer, Ordenada).*

*merge([], Derecha, Derecha).*

*merge(Izquierda, [], Izquierda).*

*merge([X|IzqResto], [Y|DerResto], [X|MergedResto]) :-*

*X =< Y,*

*merge(IzqResto, [Y|DerResto], MergedResto).*

*merge(Izquierda, [Y|DerResto], [Y|MergedResto]) :-*

*merge(Izquierda, DerResto, MergedResto).*

*coleccion([4, 2, 8, 5, 1, 6, 7, 3]).*

*coleccion([4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3]).*

*%coleccion([4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3]).*

*ordenar\_y\_medir :-*

*coleccion(Coleccion),*

*get\_time(Inicio),*

*merge\_sort(Coleccion, Ordenada),*

*get\_time(Fin),*

*Ejecucion is Fin - Inicio,*

*write('Colección ordenada: '), write(Ordenada), nl,*

*write('Tiempo de ejecución: '), write(Ejecucion), write(' segundos'), nl.*

Implementación en Javascript:

*// caso de que el array le queda un solo elemento*

*function mergeSort(array) {*

*if (array.length <= 1) {*

*return array;*

*}*

*// corto el array por la mitad y divido*

*const medio = Math.floor(array.length / 2);*

*const izquierda = array.slice(0, medio);*

*const derecha = array.slice(medio);*

*// vuelvo a llamar a la funcion para volver a cortar los arrays cortados*

*return merge(mergeSort(izquierda), mergeSort(derecha));*

*}*

*function merge(izquierda, derecha) {*

*let result = [];*

*let i = 0;*

*let j = 0;*

*// ordena 2 elementos de menor a mayor*

*while (i < izquierda.length && j < derecha.length) {*

*if (izquierda[i] < derecha[j]) {*

*result.push(izquierda[i]);*

*i++;*

*} else {*

*result.push(derecha[j]);*

*j++;*

*}*

*}*

*// va concatenando y resolviendo ordenanza*

*return result.concat(izquierda.slice(i)).concat(derecha.slice(j));*

*}*

*const coleccion = [4, 2, 8, 5, 1, 6, 7, 3];*

*//const coleccion = [4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3];*

*//const coleccion = [4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3];*

*const inicio = performance.now();*

*const coleccionOrdenada = mergeSort(coleccion);*

*const fin = performance.now();*

*console.log('Colección ordenada:', coleccionOrdenada);*

*console.log('Tiempo de ejecución:', fin - inicio, 'ms');*

Para ambos códigos se realizaron pruebas con una colección ordenada de 10 millones de elementos, debajo se muestran los rendimientos para la búsqueda de valores en distintas posiciones en la lista:

Para la colección *[4, 2, 8, 5, 1, 6, 7, 3]*:

Tiempo de ejecución Prolog: 0.0000209808349609375 ms

Tiempo de ejecución Javascript: 0.19429993629455566 ms

Para la colección *[4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3]*:

Tiempo de ejecución Prolog: 0.11801719665527344 ms

Tiempo de ejecución Javascript: 0.19559991359710693 ms

Para la colección *[4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3, 4, 2, 8, 5, 1, 6, 7, 3]*:

Tiempo de ejecución Prolog: 0.2219676971435547 ms

Tiempo de ejecución Javascript: 0.21860003471374512 ms

Viendo los resultados de las pruebas de ambos lenguajes para este algoritmo nos sorprendieron los resultados de Prolog para los 2 primeros casos, siendo que si bien este es un lenguaje orientado a establecer relaciones y con un paradigma lógico, cuando la lista a ordenar tiene un cardinal de 8 este lleva mucho menos tiempo, lo cual fue bastante inesperado, y podria ser debido a su enfoque declarativo y a que los costos de evaluación perezosa, que esto es no evalúar una expresión hasta que es necesaria, son menos notorios. Si podemos ver que a medida que la cardinalidad de la lista aumenta, el tiempo de ejecución de Prolog también lo hace en medida bastante acelerada mientras que Javascript mantiene sus tiempos de ejecución casi constantes, con un leve incremento como es de esperar pero incomparable al aumento en Prolog. Esto nos da a entender que si bien Prolog está planteado en base a un paradigma lógico y es muy similar a otros lenguajes que se encargan de manejar grandes cantidades de datos, Prolog mismo no está pensado para realizar este tipo de operaciones y su administración de la memoria y la recursión pueden influir en el tiempo de ejecución.

1. https://www.semanticscholar.org/paper/Introduction-to-the-special-issue-on-Prolog-systems-Demoen-Banda/e809f1037154f1ee218bd2cf9df1c497ff3ef539 [↑](#footnote-ref-0)
2. https://www.opensourceforu.com/2021/12/the-evolution-of-javascript/ [↑](#footnote-ref-1)