UNIVERSIDAD AUTÓNOMA DE CHIAPAS

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN CAMPUS 1

ACTIVIDAD.Act. 1.1 Investigar analizador Léxico y Lenguajes Regulares

ING. EN DESARROLLO Y TECNOLOGÍAS DE SOFTWARE

ALUMNO: Gabriel Omar Fuentes Chacon.

DOCENTE: Mtro. Luis Gutiérrez Alfaro

Fecha de entrega: 19/08/2023

Introducción

La importancia de un analizador léxico radica en su papel fundamental en la creación de programas de computadora. Su función principal es tomar el código fuente en un lenguaje de programación y dividirlo en unidades más pequeñas llamadas "tokens". Estos tokens representan las piezas mínimas con significado que el compilador comprende. El proceso que realiza el analizador léxico simplifica el código complejo en fragmentos más manejables, lo que facilita su posterior procesamiento, algunas razones clave para su relevancia son mencionados como la separación del código fuente, este código de un programa puede ser complicado, con palabras clave, símbolos y diversos operadores. El analizador léxico divide este código en partes más manejables llamadas tokens, lo que facilita su manejo en etapas posteriores.

Además, podremos obtener la explicación de detección temprana de errores, un analizador léxico identifica errores en el código, como palabras incorrectas o caracteres desconocidos, esto ayuda a los programadores a corregir errores desde el principio, ahorrando tiempo y evitando problemas en la compilación.

En cuestión de la preparación para el análisis sintáctico, los tokens generados por el analizador léxico son procesados por el analizador sintáctico, verifica si la secuencia de tokens sigue las reglas del lenguaje, sin el analizador léxico, el proceso sería más complejo, tratando con caracteres individuales, hablamos de la optimización de rendimiento, dividir el código en tokens permite al compilador crear una versión más eficiente para análisis y mejoras posteriores, esta versión es más compacta y organizada, facilitando la implementación de mejoras, facilita las características del lenguajes, los lenguajes tienen particularidades que requieren un tratamiento especial durante la compilación, además el analizador léxico identifica y clasifica estas particularidades, ayudando al compilador a manejarlas adecuadamente.

Podremos encontrar mejora del compilado, un analizador léxico bien diseñado permite la expansión y mejora del compilador con nuevas características o soporte para lenguajes adicionales, esto contribuye a que el compilador sea más modular y fácil de mantener, siendo que las funciones del analizador léxico simplifica el código al dividirlo en fragmentos, detecta errores tempranos y prepara el terreno para las etapas posteriores en el proceso de creación de un programa. También mejora el rendimiento y permite al "chef" principal agregar nuevas ideas sin problemas.

En esta investigación podremos informarnos de explicaciones tales como el analizador léxico que es una pieza esencial en la creación de programas de computadora, ya que descompone el código en unidades manejables, detecta errores tempranamente, facilita el análisis sintáctico y mejora el rendimiento del compilador, contribuyendo significativamente a la eficiencia y calidad del proceso de desarrollo de software.

Desarrollo

La importancia que es un analizador léxico:

Un analizador léxico es una parte clave en la creación de programas de computadora. Su trabajo principal es tomar el código fuente escrito en un lenguaje de programación y dividirlo en unidades más pequeñas llamadas "tokens". Estos tokens son las piezas mínimas con significado que el compilador entiende. En esencia, el analizador léxico toma el código complicado y lo descompone en piezas más simples para que pueda ser procesado más fácilmente.

Mencionaremos algunas de las razones más importantes para su relevancia:

Separación del código fuente: El código de un programa puede ser complicado, con palabras clave, símbolos y operadores diversos. El analizador léxico divide este código en partes más manejables llamadas tokens, que son más fáciles de manejar en las siguientes etapas.

Detecta errores temprano: El analizador léxico identifica errores en el código, como palabras incorrectas o caracteres desconocidos. Esto ayuda a los programadores a corregir errores desde el inicio, ahorrando tiempo y evitando problemas en la compilación.

Preparación para el análisis sintáctico: Los tokens generados por el analizador léxico son procesados por el analizador sintáctico. Este verifica si la secuencia de tokens sigue las reglas del lenguaje. Sin el analizador léxico, el proceso sería más complejo, tratando con caracteres individuales.

Optimización de rendimiento: Dividir el código en tokens permite al compilador crear una versión más eficiente para análisis y mejoras posteriores. Esta versión es más compacta y organizada, facilitando la aplicación de mejoras.

Facilita características del lenguaje: Muchos lenguajes tienen particularidades que necesitan tratamiento especial en la compilación. El analizador léxico identifica y clasifica estas particularidades, ayudando al compilador a manejarlas de manera adecuada.

Mejora del compilador: Un analizador léxico bien diseñado permite la expansión y mejora del compilador con nuevas características o soporte para lenguajes adicionales. Esto contribuye a que el compilador sea más modular y fácil de mantener.

Funciones del analizador léxico

El analizador léxico hace que el código sea más manejable al dividirlo en pedacitos, detecta errores tempranos y ayuda a preparar todo para las etapas siguientes del proceso de hacer un programa. También ayuda a mejorar el rendimiento y permite que el "chef" principal agregue nuevas ideas sin problemas.

Divide el código en pedacitos:

- Imagina que tienes una gran pizza de código.
- El analizador léxico toma esa pizza y la corta en trocitos más pequeños llamados "tokens".
- Cada token es como un pedacito de pizza con su propio sabor y significado.

- Busca errores tempranos:

- Piensa en el analizador léxico como un inspector de calidad.
- Revisa cada pedacito de código para asegurarse de que todo esté correcto.
- Si encuentra algo raro o mal escrito, lo señala para que puedas corregirlo antes de seguir cocinando.

Prepara el terreno para el siguiente paso:

- Después de dividir la pizza, el analizador léxico coloca los pedacitos en bandejas organizadas.
- Esto facilita el trabajo del siguiente chef, el "analizador sintáctico".
- Ese chef necesita los ingredientes listos y ordenados para preparar la siguiente parte de la receta.

Ayuda a optimizar el rendimiento:

- Imagina que estás organizando una fiesta y quieres hacerla más eficiente.
- El analizador léxico agrupa los ingredientes similares en bandejas, de manera que se puedan manejar más fácilmente.
- Esto ahorra tiempo y energía durante la preparación.

Entiende el lenguaje especial:

- A veces, los programadores usan palabras o reglas especiales en su receta de código.
- El analizador léxico reconoce estas reglas y las etiqueta para que el chef de más adelante sepa cómo tratarlas correctamente.

Ayuda al crecimiento del chef principal:

- Imagina que el chef principal quiere añadir nuevos ingredientes o hacer que la receta sea más interesante.
- El analizador léxico es como un asistente atento que ayuda al chef a adaptar la receta sin causar caos en la cocina.

Componentes léxicos, patrones y lexema.

Componentes Léxicos:

Los componentes léxicos, también conocidos como tokens, son las unidades más pequeñas de un programa de computadora que el analizador léxico reconoce. Representan elementos individuales en el código fuente, como palabras clave, identificadores, números, operadores y símbolos especiales. Los componentes léxicos son esenciales para el proceso de análisis y comprensión del código por parte del compilador o intérprete.

Patrones:

Los patrones son reglas o descripciones que definen cómo se deben reconocer los diferentes componentes léxicos en el código fuente. Cada tipo de token tiene su propio patrón asociado. Estos patrones se expresan generalmente en forma de expresiones regulares, que son secuencias de caracteres que describen un conjunto de cadenas de texto válidas. Los patrones permiten al analizador léxico identificar y extraer los tokens del código fuente de manera eficiente.

Lexema:

El lexema es la secuencia concreta de caracteres que corresponde a un token específico en el código fuente. En otras palabras, es la instancia real de un componente léxico en el programa. Por ejemplo, en el código int num = 42;, el lexema para el token de número es "42", para el token de identificador es "num", y para el token de operador de asignación es "=".

En resumen, los componentes léxicos son los bloques fundamentales del código fuente, los patrones son las reglas que definen cómo se deben reconocer estos componentes, y el lexema es la instancia concreta de un componente léxico en el código. Juntos, estos conceptos son esenciales para el análisis léxico en la interpretación y compilación de programas de computadora.

Lenguajes Regulares

Los lenguajes regulares forman una categoría de lenguajes formales reconocibles y generables a través de expresiones regulares o autómatas finitos. Son fundamentales en teoría de la computación y lingüística formal, y se aplican directamente en análisis léxico de lenguajes de programación, procesamiento de texto y verificación de patrones. Estos lenguajes se describen mediante expresiones regulares, que son patrones de texto que incluyen caracteres literales y meta caracteres para buscar, extraer o reemplazar patrones en cadenas, como reconocer números de seguro social en el formato (XXX-XX-XXXX) con la expresión \d{3}-\d{2}-\d{4}.

Los autómatas finitos, como los autómatas finitos deterministas (AFD), son modelos abstractos que pueden estar en estados finitos y transaccionar en respuesta a la entrada, un AFD son útiles para

verificar si una cadena pertenece a un lenguaje regular. Los lenguajes regulares son cerrados bajo operaciones como unión, concatenación y clausura de Kleene, lo que permite combinar o modificar dos lenguajes regulares y obtener un lenguaje regular resultante.

Las aplicaciones prácticas de los lenguajes regulares son diversas e incluyen la manipulación de texto en procesamiento de lenguaje natural, verificación de patrones en búsqueda y reemplazo, análisis léxico en lenguajes de programación (identificando tokens como palabras clave, identificadores y números) y establecimiento de reglas para validar entradas de usuario en aplicaciones y formularios web. En conjunto, los lenguajes regulares y sus conceptos asociados son fundamentales en la informática y la tecnología moderna

Explicar el Lema de Bombeo para lenguajes regulares con un ejemplo.

El Lema de Bombeo para lenguajes regulares es una herramienta teórica que se utiliza para demostrar que ciertos lenguajes no son regulares. Este lema establece que, en cualquier lenguaje regular, existe un número llamado "constante de bombeo". Si tomamos una cadena lo suficientemente larga en ese lenguaje, podemos dividirla en partes. Al "bombardear" o repetir una de esas partes, la cadena resultante debe seguir siendo parte del mismo lenguaje.

El lema se desglosa en tres aspectos:

- Longitud de cadena: Para cualquier lenguaje regular L, hay un valor p (la constante de bombeo) tal que cualquier cadena s en L con longitud mayor o igual a p se puede dividir en tres partes: x y z.
- Propiedades de bombeo: Para cualquier i ≥ 0, la cadena “xy^iz” también debe pertenecer a L.
- Restricciones de longitud: La suma de las longitudes de x e y no debe superar p, y la longitud de y debe ser mayor que 0.

Ejemplos donde se ilustran cómo se aplica el lema de bombeo:

Ejemplo 1: Lenguaje de (a^nb^n)

Consideremos el lenguaje L = {a^n b^n | n ≥ 0}, que contiene cadenas de "a" seguidas de la misma cantidad de "b". Queremos demostrar que L no es regular.

Supongamos que L es regular y tiene una constante de bombeo p. Tomemos la cadena s = a^pb^p. Según el lema de bombeo, s puede ser dividida en x y z cumpliendo con las propiedades mencionadas. Si bombeamos la cadena y (repetimos y), obtenemos la cadena xy^2z = a^(p+|y|)b^p.

Sin embargo, ahora el número de "a" excede el número de "b", y por lo tanto, xy^2z ya no está en L. Esto contradice la definición de L y demuestra que L no es regular.

Ejemplo 2: Lenguaje de a^p

Consideremos el lenguaje L = {a^p | p es primo}, que contiene cadenas de "a" repetidas p veces, donde p es un número primo. Queremos demostrar que L no es regular.

Supongamos que L es regular y tiene una constante de bombeo p. Tomemos la cadena s = a^p. Según el lema de bombeo, s puede ser dividida en x y z cumpliendo con las propiedades mencionadas. Si bombeamos la cadena y (repetimos y), obtenemos la cadena xy^2z = a^(p+|y|). Sin embargo, no importa cuántas veces repitamos "y", el número de "a" en la cadena resultante nunca será un número primo, lo cual contradice la definición de L. Por lo tanto, L no es regular.

Explicar las propiedades de cerradura de lenguajes regulares con un ejemplo.

Las propiedades de cerradura en lenguajes regulares se refieren a cómo ciertas acciones que realizamos con estos lenguajes todavía mantienen su "estatus" de ser regulares. Estas acciones son como "operaciones mágicas" que aplicamos a los lenguajes y, aun así, obtenemos lenguajes regulares como resultado. Las operaciones involucradas son la combinación (unión), la secuencia (concatenación) y la creación de múltiples versiones (clausura de Kleene). Permíteme explicar cada una con ejemplos:

Combinación de Unión:

Imagina que tienes dos conjuntos de palabras, digamos uno con todas las formas de "a" y otro con todas las formas de "b". Al combinar estos conjuntos, obtendrás un conjunto que contiene tanto las palabras "a" como las palabras "b". A pesar de la mezcla, todavía estamos lidiando con conjuntos de palabras "simples". Lo mismo ocurre en los lenguajes regulares: si tienes dos lenguajes regulares, al combinarlos, obtendrás un lenguaje regular.

Ejemplo: Si tenemos el lenguaje L1 = {a^n | n ≥ 0} y el lenguaje L2 = {b^n | n ≥ 0}, su combinación L1 ∪ L2 contendrá todas las formas de "a" y "b", pero aún se mantendrá como un lenguaje regular.

Secuencia de Concatenación:

Si tienes un conjunto de palabras que comienza con "a" y otro conjunto que termina en "b", al combinarlos obtendrás palabras que empiezan con "a" y terminan en "b". Esta operación sigue siendo "simple" en términos de complejidad de lenguaje, y lo mismo aplica en los lenguajes regulares. Si concatenas cadenas de dos lenguajes regulares, el resultado seguirá siendo un lenguaje regular.

Ejemplo: Si tomamos los lenguajes L1 = {a^n | n ≥ 0} y L2 = {b^n | n ≥ 0}, su concatenación L1L2 contendrá cadenas como "ab", "aabb", "aaabbb", etc., y aún será un lenguaje regular.

Creación de Múltiples Versiones (Clausura de Kleene):

Imagina tener un lenguaje con una sola palabra, digamos "1". Ahora, aplicando una "magia" teórica, puedes crear versiones adicionales como "", "11", "111", etc. Estás duplicando, triplicando, cuadruplicando y así sucesivamente. Esto también es lo que sucede con la clausura de Kleene en los lenguajes regulares. Al aplicar esta operación, puedes obtener múltiples versiones de las cadenas en el lenguaje, y aun así, sigues tratando con un lenguaje regular.

Ejemplo: Si consideramos el lenguaje L = {0, 1}, su clausura de Kleene L* contendrá versiones repetidas de "0" y "1", como "", "0", "1", "00", "11", "001", "111", etc., y seguirá siendo un lenguaje regular.

Explicar las propiedades de decisión de lenguajes regulares con un ejemplo.

Las propiedades de decisión en los lenguajes regulares son afirmaciones que se pueden comprobar mediante procedimientos claros y precisos para los lenguajes de esta categoría. Estas propiedades nos permiten abordar cuestiones particulares acerca de estos lenguajes, como si contienen ciertas cadenas, si están vacíos, si son idénticos entre sí, entre otros aspectos. Aquí te proporciono dos ejemplos que ilustran estas propiedades de decisión en acción:

Propiedad: Lenguaje Vacío

El lenguaje L no contiene ninguna cadena en absoluto este método de decisión se usa para determinar si un lenguaje L está vacío, podemos construir un tipo especial de "máquina" llamada autómata finito determinista (AFD) que representa a L. Si el AFD tiene al menos un "punto de llegada" que se puede alcanzar desde el "punto de partida", esto nos dice que el lenguaje no está vacío, ya que existe al menos una cadena aceptable. Sin embargo, si no hay ningún punto de llegada desde el punto de partida, entonces el lenguaje está vacío.

Ejemplo: Supongamos que tenemos el lenguaje L = {a^n b^n | n ≥ 0}, que incluye cadenas de "a" seguidas de "b". Al construir un AFD para L, notamos que existe un punto de llegada después de consumir todas las "a". Dado que hay al menos una cadena aceptable, podemos concluir que L no está vacío.

Propiedad: Igualdad de Lenguajes

Los lenguajes L1 y L2 idénticos, contienen exactamente las mismas cadenas, método de decisión, para determinar si dos lenguajes L1 y L2 son iguales, podemos emplear un enfoque de comparación exhaustiva. Crearemos AFDs individuales para cada lenguaje y luego verificaremos si todas las cadenas aceptadas por L1 también son aceptadas por L2, y viceversa. Si encontramos que todas las cadenas en L1 están en L2 y viceversa, podemos concluir que los lenguajes son iguales.

Ejemplo: Consideremos los lenguajes L1 = {a^n b^n | n ≥ 0} y L2 = {w | w contiene el mismo número de "a" y "b"}. Si creamos AFDs para ambos lenguajes y examinamos las cadenas aceptadas por cada uno, notaremos que todas las cadenas aceptadas por L1 también son aceptadas por L2, y viceversa. En consecuencia, concluimos que L1 y L2 son idénticos.

Explicar el proceso de determinación de equivalencias entre estados y lenguajes regulares con un ejemplo:

Proceso de Determinación de Equivalencias entre Estados, dentro de un autómata finito determinista (AFD), dos estados se consideran equivalentes si, para cualquier entrada posible, conducen al mismo estado y aceptan las mismas cadenas.

El procedimiento para decidir si dos estados son equivalentes usualmente abarca dividir los estados en grupos iniciales de estados comparables y luego iterar para perfeccionar esas divisiones hasta que ya no haya cambios.

El proceso de determinar equivalencias entre estados en autómatas finitos como en el proceso de establecer equivalencias entre lenguajes regulares, la clave radica en analizar cómo se comportan y qué cadenas aceptan los elementos involucrados para establecer si son comparables o no.

Iniciación: Se inicia dividiendo los estados en dos categorías: los estados finales y los estados no finales.

Ajuste: Posteriormente, se procede a analizar los grupos de estados y segmentar cada grupo en conjuntos más pequeños basados en cómo se comportan al enfrentar distintas entradas. Esto supone comparar las transiciones y los estados alcanzados por cada entrada desde los estados del grupo actual.

Iteración: El proceso de ajuste se repite hasta que las particiones de estados no sufran más cambios. Si en una iteración no hay segmentaciones adicionales, los estados dentro del grupo actual son considerados equivalentes.

Ejemplo 1:

Imagina un AFD que posee dos estados: q0 y q1. El estado q0, además de ser el estado inicial, es también el estado final. Por otro lado, q1 es un estado no final. Ambos estados tienen transiciones en "0" y "1", sin embargo, el estado q0 también posee una transición en "1" que conduce a q1. En este caso, los estados q0 y q1 no son equivalentes debido a sus distintas transiciones y comportamientos.

Ejemplo 2:

Ahora, considera un AFD con tres estados: q0, q1 y q2. Todos estos estados son finales. Los estados q0 y q1 tienen transiciones en "a" que llevan a q2, no obstante, el estado q0 también tiene una transición en "b" que conduce a q1. En este ejemplo, los estados q0 y q1 son equivalentes ya que aceptan las mismas cadenas y poseen transiciones análogas.

Proceso de Determinación de Equivalencias entre Lenguajes Regulares:

Para determinar si dos lenguajes regulares son equivalentes, es factible emplear enfoques como construir AFDs, cotejar expresiones regulares o realizar pruebas de conjuntos. El objetivo es comprobar si ambos lenguajes aceptan exactamente las mismas cadenas.

Ejemplo 1:

Supongamos que hay dos lenguajes, L1 = {a^n b^n | n ≥ 0} y L2 = {w | w contiene el mismo número de "a" y "b"}. Al construir AFDs o expresiones regulares para cada lenguaje, y luego contrastar las cadenas aceptadas por ambos, se puede observar que L1 y L2 son equivalentes, ya que ambos admiten las mismas cadenas: aquellas con una cantidad igual de "a" y "b".

Ejemplo 2:

Ahora, piensa en los lenguajes L3 = {a^n b^n | n ≥ 0} y L4 = {a^n | n ≥ 0}. Al comparar las cadenas aceptadas por estos lenguajes, es evidente que no son equivalentes, dado que L3 contiene cadenas con cantidades iguales de "a" y "b", mientras que L4 únicamente contiene cadenas con "a".

Explicar el proceso de minimización de DFA

La minimización de un autómata finito determinista (DFA) es un procedimiento mediante el cual simplificamos el DFA original para obtener uno más compacto, pero que todavía acepta exactamente las mismas cadenas. El objetivo es eliminar estados que no son necesarios y combinar estados que hacen lo mismo.

Ejemplo: Simplificando un DFA

Imagina un DFA que reconoce cadenas en el alfabeto {0, 1} que contienen un número par de "0"s:

Estado inicial: q0

Estados finales: q0 “los estados no mencionados son no finales”

0 1

→ q0 ----> q1 ----> q2

Identificando pares de estados no equivalentes:

Comenzamos al observar pares de estados que claramente no son equivalentes. En este caso, el par (q0, q1) no es equivalente ya que q0 es final y q1 no lo es.

Iteración para encontrar estados equivalentes:

Ahora, dividimos los estados en dos grupos: los estados finales y los no finales. Luego, iteramos para refinar estos grupos:

En la primera iteración, los estados finales se dividen en {q0}, mientras que los no finales permanecen en {q1, q2}.

En la segunda iteración, examinamos cómo se comportan los estados no finales al recibir las entradas "0" y "1":

Para el estado q1: al recibir "0", cambia a q0 (final); al recibir "1", va a q2 (no final). Dado que estos estados están en grupos diferentes, q1 no es equivalente a ninguno de ellos.

Para el estado q2: al recibir "0", cambia a q1; al recibir "1", permanece en q2. Dado que q1 y q2 son diferentes, q2 solo es equivalente a sí mismo.

Resultado de la minimización:

Tras estas iteraciones, notamos que q1 y q2 son equivalentes, ya que ambos responden de igual manera a las entradas "0" y "1". Por lo tanto, podemos fusionar q1 y q2 en un solo estado. El DFA simplificado queda así:

Estado inicial: q0

Estados finales: q0q1 (estado único que combina q0 y q1)

	0	1
→	q0q1	q0q1

Este DFA simplificado es igual al original, pero más compacto al eliminar un estado innecesario.

Este procedimiento es general para simplificar DFA, identificamos pares de estados no equivalentes y realizamos iteraciones para mejorar los grupos de estados equivalentes.

Conclusión:

Como conclusión tenemos que la investigación resalta la importancia fundamental del analizador léxico en la creación de programas de computadora, este componente es esencial para dividir el código fuente en unidades más pequeñas llamadas "tokens", lo que facilita su procesamiento y comprensión, algunas razones clave para su relevancia incluyen la separación del código fuente en partes manejables, la detección temprana de errores, la preparación para el análisis sintáctico, la optimización del rendimiento y la facilitación de características del lenguaje, además el analizador léxico descompone el código en tokens, actuando como un inspector de calidad que verifica la corrección de cada fragmento y prepara el terreno para las etapas posteriores, además, contribuye a la optimización del rendimiento al agrupar elementos similares y entender las particularidades del lenguaje, permite el crecimiento del compilador al facilitar la adición de nuevas características sin caos.

Siendo también que los componentes léxicos, patrones y lexemas son conceptos esenciales en el análisis léxico, ya que los componentes léxicos son las unidades básicas reconocibles, los patrones describen cómo se reconocen estos componentes y los lexemas son las instancias reales en el código, podemos también encontrar información valiosa de los lenguajes regulares, definidos por expresiones regulares y autómatas finitos, tienen aplicaciones vitales en diversos campos, desde procesamiento de lenguaje natural hasta análisis léxico en lenguajes de programación, en otra información también se explica el lema de bombeo para lenguajes regulares siendo una herramienta teórica que establece que ciertos lenguajes no son regulares al probar que las cadenas pueden ser bombeadas y aún pertenecer al lenguaje.

Para finalizar tenemos los procesos de determinación de equivalencias entre estados en autómatas finitos busca simplificarlos al agrupar estados equivalentes, mientras que el proceso de minimización de DFA busca simplificar el DFA original al eliminar estados no necesarios y combinar estados equivalentes, cada uno de estos procedimientos son esenciales para mejorar la eficiencia y comprensión de los autómatas.

Referencias bibliográficas en apa

Vázquez, J. C., Constable, L., Jornet, W., & Meloni, B. (2015). Enseñanzas de la Implementación de un Analizador Léxico.

Tarazona, A. Y. V. Propiedades de clausura en lenguajes regulares.

Pérez Díaz, A. F. (2022). Clasificación de lenguajes naturales en la Jerarquía de Chomsky: lenguajes regulares e independientes de contexto.

Cardozo Álvarez, N. (2008). Estudio de relaciones entre la teoria de autómatas, la lógica de segundo orden y el mu-cálculo.

García Fernández, L. A., & Martínez Vidal, M. G. (2005). Primera práctica: Introducción al Analizador Léxico FLEX.