REPORTE FINAL

INGENIERÍA EN SISTEMAS COMPUTACIONALES
7AM

Clavel Vergara Kimbberly Nicole.....171080181
González River Emmanuel Alejandro.....171080089
Diez Bonilla Guerrero Dilan Eduardo.....171080167
Lopez Tello Rodriguez Mario Ivan.....171080102

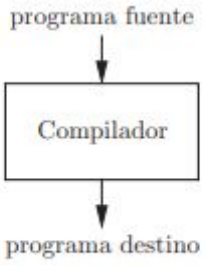
ACTIVIDADES SEMANA 3 (OCT 5-9, 2020)

INTRODUCCIÓN

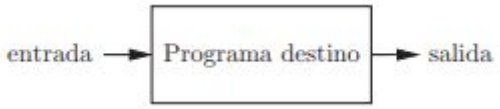
Los lenguajes de programación son notaciones de los cálculos de las personas y las máquinas. Todo el software que se ejecutan las computadoras depende de los lenguajes de programación ya que se escribió en algún lenguaje. Pero antes de poder ejecutar un programa, primero debe traducirse a un formato en el que una computadora pueda ejecutarlo.

1.1 PROCESADORES DE LENGUAJE

Un compilador es un programa que puede leer el lenguaje fuente y traducirlo en un lenguaje destino.



Si el programa destino es un programa ejecutable en lenguaje máquina, entonces el usuario puede ejecutarlo para procesar las entradas y producir salidas.



Un intérprete es otro tipo de procesador de lenguaje. En vez de producir un programa destino como una traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa fuente con las entradas proporcionadas por el usuario.

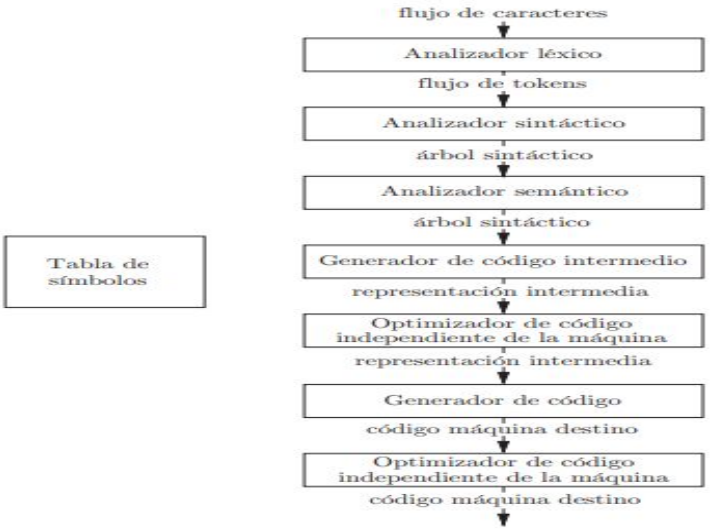
El programa destino en lenguaje máquina que produce un compilador es más rápido que un intérprete al momento de asignar las entradas a las salidas. Pero, por lo regular, el intérprete puede ofrecer mejores diagnósticos de error que un compilador, ya que ejecuta el programa fuente instrucción por instrucción.

1.2 LA ESTRUCTURA DE UN COMPILADOR

Hay dos procesos en esta asignación: análisis y síntesis.

La parte del análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlo. También recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se pasa junto con la representación intermedia a la parte de la síntesis.

La parte de la síntesis construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el front-end del compilador; la parte de la síntesis (propiamente la traducción) es el back-end



1.2.1 ANÁLISIS LÉXICO

A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma:

nombre-token, valor-atributo

Que pasa a la fase siguiente, el análisis de la sintaxis. En el token, el primer componente nombre-token es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente valor-atributo apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se necesita para el análisis semántico y la generación de código.

1.2.2 ANÁLISIS SINTÁCTICO

La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.

Este árbol muestra el orden en el que deben llevarse a cabo las operaciones en la siguiente asignación:

posición = inicial + velocidad * 60

1.2.4 GENERACIÓN DE CÓDIGO INTERMEDIO

Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

1.2.5 OPTIMIZACIÓN DE CÓDIGO

La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código de destino que consuma menos poder.

1.2.6 GENERACIÓN DE CÓDIGO

El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa.

1.2.7 ADMINISTRACIÓN DE LA TABLA DE SÍMBOLOS

La tabla de símbolos es una estructura de datos que contiene un registro para cada nombre de variable, con campos para los atributos del nombre.

1.2.9 HERRAMIENTAS DE CONSTRUCCIÓN DE COMPILADORES

Cualquier desarrollador de software contiene herramientas como editores de lenguaje, depuradores, administradores de versiones, profilers, ambientes seguros de prueba, etcétera. Estas herramientas utilizan lenguajes especializados para especificar e implementar componentes específicos, y muchas utilizan algoritmos bastante sofisticados.

1.4 LA CIENCIA DE CONSTRUIR UN COMPILADOR

Un compilador debe aceptar todos los programas fuente conforme a la especificación del lenguaje; el conjunto de programas fuente es infinito y cualquier programa puede ser muy largo, posiblemente formado por millones de líneas de código. Cualquier transformación que realice el compilador mientras traduce un programa fuente debe preservar el significado del programa que se está compilando.

1.4.1 MODELADO EN EL DISEÑO E IMPLEMENTACIÓN DE COMPILADORES

Estos modelos son útiles para describir las unidades de léxico de los programas (palabras clave, identificadores y demás) y para describir los algoritmos que utiliza el compilador para reconocer esas unidades.

1.4.2 LA CIENCIA DE LA OPTIMIZACIÓN DE CÓDIGO

Optimización en el diseño de compiladores se refiere a los intentos que realiza un compilador por producir código que sea más eficiente que el código obvio.

Procesadores de lenguaje. Un entorno de desarrollo integrado de software incluye muchos tipos distintos de procesadores de lenguaje, como compiladores, intérpretes, ensambladores, enlazadores, cargadores, depuradores, profilers.

Fases del compilador. Un compilador opera como una secuencia de fases, cada una de las cuales transforma el programa fuente de una representación intermedia a otra.

Lenguajes máquina y ensamblador. Los lenguajes máquina fueron los lenguajes de programación de la primera generación, seguidos de los lenguajes ensambladores. La programación en estos lenguajes requería de mucho tiempo y estaba propensa a errores.

Modelado en el diseño de compiladores. El diseño de compiladores es una de las fases en las que la teoría ha tenido el mayor impacto sobre la práctica. Entre los modelos que se han encontrado de utilidad se encuentran: autómatas, gramáticas, expresiones regulares, árboles y muchos otros.

Optimización de código. Aunque el código no puede verdaderamente “optimizarse”, esta ciencia de mejorar la eficiencia del código es tanto compleja como muy importante. Constituye una gran parte del estudio de la compilación.

Lenguajes de alto nivel. A medida que transcurre el tiempo, los lenguajes de programación se encargan cada vez más de las tareas que se dejaban antes al programador, como la administración de memoria, la comprobación de consistencia en los tipos, o la ejecución del código en paralelo.

Compiladores y arquitectura de computadoras. La tecnología de compiladores ejerce una influencia sobre la arquitectura de computadoras, así como también se ve influenciada por los avances en la arquitectura. Muchas innovaciones modernas en la arquitectura dependen de la capacidad de los compiladores para extraer de los programas fuente las oportunidades de usar con efectividad las capacidades del hardware.

Productividad y seguridad del software. La misma tecnología que permite a los compiladores optimizar el código puede usarse para una variedad de tareas de análisis de programas, que van desde la detección de errores comunes en los programas, hasta el descubrimiento de que un programa es vulnerable a uno de los muchos tipos de intrusiones que han descubierto los “hackers”.

Reglas de alcance. El alcance de una declaración de x es el contexto en el cual los usos de x se refieren a esta declaración. Un lenguaje utiliza el alcance estático o alcance léxico si es posible determinar el alcance de una declaración con sólo analizar el programa. En cualquier otro caso, el lenguaje utiliza un alcance dinámico.

Entornos. La asociación de nombres con ubicaciones en memoria y después con los valores puede describirse en términos de entornos, los cuales asignan los nombres a las ubicaciones en memoria, y los estados, que asignan las ubicaciones a sus valores.

Estructura de bloques. Se dice que los lenguajes que permiten anidar bloques tienen estructura de bloques. Un nombre x en un bloque anidado B se encuentra en el alcance de

una declaración D de x en un bloque circundante, si no existe otra declaración de x en un bloque intermedio.

Paso de parámetros. Los parámetros se pasan de un procedimiento que hace la llamada al procedimiento que es llamado, ya sea por valor o por referencia. Cuando se pasan objetos grandes por valor, los valores que se pasan son en realidad referencias a los mismos objetos, lo cual resulta en una llamada por referencia efectiva.

Uso de alias. Cuando los parámetros se pasan (de manera efectiva) por referencia, dos parámetros formales pueden referirse al mismo objeto. Esta posibilidad permite que un cambio en una variable cambie a la otra.

ACTIVIDADES SEMANA 4 (OCT 12-16, 2020)

VIDEO 1

Un compilador es un ejemplo de cómo la teoría se traduce a la práctica.

La complejidad de un compilador surge del hecho de que se requiere mapear un programa como requisitos que están escritos en lenguajes de alto nivel (programa CA).

La teoría de la celosía se utiliza para desarrollar análisis estáticos (se basa en álgebra lineal y paralelización).

El análisis de caché utiliza análisis estático y teoría de la probabilidad.

El uso de pruebas de software requiere un enfoque de análisis de flujo de datos y estimación.

Un compilador toma una limpieza y conoce el programa fuente (este puedes estar en c o c plus).

Las salidas es un programa ensamblador para la máquina en particular.

La salida del ensamblador se llama código de máquina reubicable que se combina con archivos de biblioteca.

Un analizador léxico toma cómo entrar un flujo de caracteres.

El analizador léxico toma como entrada un programa fuente y luego se conoce como una secuencia de tokens.

Los analizadores se pueden generar automáticamente a partir de especificaciones de expresiones regulares.

Un analizador léxico es un autómata determinista de estado finito.

Un analizador corresponde a un autómata de empuje.

El analizador de sintaxis mira los tokens y descubre si la sintaxis está de acuerdo con una especificación gramatical.

Los análisis no pueden manejar características sensibles al contexto de los lenguajes de programación.

Un parámetro no puede ser detectado por el analizador.

El árbol de sintaxis es producido por un analizador de sintaxis.

El analizador también nos muestra información en la tabla de símbolos o en el árbol es sintaxis.

El generador de código de máquina necesita conocer los tipos de variables para generar los tipos apropiados de instrucciones.

La semántica estática de los lenguajes de programación se puede especificar usando lo que son las gramáticas de atributo.

Las gramáticas de atributo son una extensión de gramáticas libres de contexto.

El tipo de código intermedio se basa en la aplicación.

El gráfico de dependencia del programa es útil en paralelización automática, programación instrucciones, canalización de software, etcétera.

El gráfico de dependencia muestra la dependencia entre varios tipos de declaraciones en el programa.

Las mejores pueden ser por tiempo espacio o consumo de energía.

VIDEO 2

La característica común de los lenguajes es que hay cuatro secuencias de caracteres.

El análisis léxico de base en autómatas de estado finito.

El analizador de conferencias hace que tanto el analizador léxico como en el analizador sean más eficientes.

Los tokens son patrones y los lexemas son las cadenas de caracteres.

Un flotador es una palabra reservada.

Una cadena de caracteres que lógicamente pertenece mundos es un token.

Los tokens se tratan como símbolos terminales de la gramática que especifica un idioma.

Un patrón es el conjunto de cuerdas para el que se produce la misma ficha.

Un lexema es la secuencia de caracteres emparejado por un patrón para formar el token correspondiente.

El analizador léxico de tokens no puede detectar algún error significativo a excepción de unos muy simples.

Los errores son llamados por los analizadores sintácticos.

Los diagramas de transición son variantes de estado finito (autómatas) que se usan para implementar definiciones regulares y para reconocer tokens.

Los diagramas de transición modelan los analizadores léxicos.

Una cadena es una secuencia finita de símbolos yuxtapuestos qué son símbolos colocados uno tras otro.

El idioma se define como un conjunto de cadenas de símbolos de algún alfabeto.

Sigma star son todas las cadenas posibles sobre el alfabeto particular cada subconjunto de sigma star es un idioma.

Los lenguajes se representan con expresiones regulares.

La jerarquía de idiomas se conoce como la jerarquía de Chomsky basada en respetar al inventor que propuso esta jerarquía.

Los autómatas lineales delimitados aceptan lenguajes sensibles al contexto y se pueden especificar usando gramáticas sensibles al contexto.

Delta dice como la máquina progresa de un estado a otro al consumir un determinado símbolo de la entrada.

Delta es una función de transición.

El lenguaje aceptado por un estado finito autómata es el conjunto de todas las cadenas dadas por él.

ACTIVIDADES SEMANA 5 (OCT 19-23, 2020)

VIDEO 1

Un autómata finito determinista tiene exactamente una transición en cada símbolo o estado.

Puede haber más de una transición en un símbolo o estado portador.

Es posible hacer una transmisión a un conjunto de Estados de cada estado de NFA.

Cada subconjunto de los estados NFA es una posibilidad de un estado.

Todos los estados del DFA que contienen algún estado final como miembro serían los estados finales de la DFA.

Épsilon en NFA Isla cadena vacía como la conocemos.

Épsilon es una transición silenciosa y no consume ninguna entrada.

Un NFA convertida en particular así te mismo lenguaje que esté en una épsilon.

Las expresiones regulares son específicas.

Las expresiones regulares ayudan especificar analizadores léxicos.

El estado inicial de una DFA puede ser representado por q0

Cada subconjunto de un NFA posiblemente es un estado DFA

El símbolo "*" es llamado como cerradura de Kleene

Para mostrar todas las cadenas que 0 y 1 pueden crear se ocupa la cerradura del Kleene de la siguiente manera (0+1)*

Es un NFA si existe una secuencia correspondiente a una cadena, empezando de un estado inicial a un estado final.

VIDEO 2

Los diagramas de transición son autómatas finitos deterministas generalizados.

Los bordes de los diagramas de transición pueden estar etiquetados con una definición regular.

Algunos estados de aceptación pueden indicarse como estados de retracción.

El diagrama de transmisión deberá traducirse manualmente a un programa analizador léxico.

Los diagramas de transición deben probarse y luego se deben registrar todas las coincidencias y la coincidencia más larga debe ser capaz de ordenar los diagramas de transición de forma adecuada.

Usar Lex para generar analizadores léxicos, lo hace fácil para el escritor del compilador.

Lex tiene un lenguaje para describir expresiones, que están en el corazón del análisis léxico.

El Lex me inscribí conoce todas las expresiones regulares, especificaciones para cada uno de los patrones a detectar en el análisis léxico.

Lex genera un comparador de patrones para la expresión regular.

La herramienta Lex programas que son apropiados para el análisis léxico.

La estructura general de un programa Lex, se divide en reglas y en subrutinas de usuario, mismas que son partes esenciales y son especificaciones de Lex.

La sección del programa Lex, contiene definiciones regulares.

La estructura general de LEX:

{Definiciones}

%%

{Reglas}

%%

{Subrutina de usuario}

Las definiciones y subrutinas son opcionales. El segundo %% es opcional pero el primer %% indica el comienzo de las reglas

Las expresiones regulares corresponden con las gramáticas de 3 tipos de la jerarquía de chomsky. Las expresiones regulares son una forma de especificar patrones, se entiende por patrón la forma de describir cadena de caracteres.

El lenguaje que se reconoce mediante estas expresiones regulares (r) , se denominan lenguaje generado por la expresión regular L(r).

Las operaciones básicas de una expresión regular son representadas como:

Se denomina por | significa que puede ser **a** o **b**. Esta operación equivale a la unión, puesto que tanto a como b valdrían como lexemas de patrón.

- a | b selección entre a y b.

Concatenación:

Se construye poniendo un símbolo al lado del otro y no utiliza ningún carácter para representarlo, por lo tanto el lexema equivalente tiene que ser “ab”.

- ab concatenación de a y b

Cerradura de Kleen:

Se denota por el carácter * identifica una concatenación de símbolos incluyendo la cadena vacía, es decir de 0 a más instancias.

- a* indica la cerradura de Kleen.

VIDEO 3

Las gramáticas libres de contexto son la base para la especificación de lenguajes de programación.

Analizar idiomas en contexto se basa en autómatas pushdown.

El reconocimiento de lenguaje regular se basa en autómatas de estados finitos.

Existen dos tipos de análisis: uno es el análisis de arriba hacia abajo, y el otro es el análisis de abajo hacia arriba.

El análisis de arriba hacia abajo estudia LL y descenso recursivo, técnicas de análisis sintáctico.

El análisis de abajo hacia arriba, estudiaremos las técnicas de análisis LR, así como, una herramienta llamada yacc que se basa en el análisis LR.

Se puede escribir una gramática para describir la sintaxis estructural de programas bien formados o programas correctos.

Las reglas de la gramática establecen cómo se hacen las funciones a partir de la lista de parámetros y declaraciones.

Las reglas de la gramática dirán como los enunciados se componen de expresiones y a su vez, como las expresiones que componen de números, nombres, paréntesis, etcétera.

La herramienta yacc se utiliza para generar analizadores automáticamente.

Yacc es una herramienta que toma una especificación de gramática y escribe un analizador sintáctico en C que reconoce sentencias válidas para dicha gramática.

Sin contexto, las gramáticas son una subclase de lenguajes de programación.

Hay diferentes tipos de lenguajes y gramáticas, hay lenguajes regulares, sin contexto, idiomas, lenguajes, sensibles al contexto y lenguajes de tipo 0.

Gramáticas libres de contexto se utilizan para especificar lenguajes libres de contexto y son estos los más útiles para programar.

Una gramática libre de contexto se denota como G.

G=(N,T,P,S)

Donde:

N: Es un conjunto finito del que se conoce como no terminales o variables.

T: Es un conjunto finito del que se conoce como terminales.

S:Es un no terminal este es un símbolo de inicio.

P: Es un conjunto Finito de producciones.

El analizador verifica que la cadena de tokens para un programa en esa programación en particular donde el lenguaje se puede generar a partir de la gramática que hemos proporcionado como base para el analizador.

En el sistema PDA los símbolos significan:

- Q es un conjunto finito de estados
- Σ es el alfabeto de entrada
- Γ es el alfabeto de la pila
- $q0 \in Q$ es el estado inicial
- $Z0 \in \Gamma$ es el símbolo de inicio de la pila
- $F \subseteq Q$ es el conjunto de estados finales
- δ es la funcion de transicion

Árbol de Derivación

La derivación es representado por ==>

Se representa por x ==> y y es la aplicación de una regla de producción a→b a una cadena x para convertirla en otra cadena o palabra y

Es una representación utilizada para describir el proceso de derivación de una sentencia, donde se cumple lo siguiente:

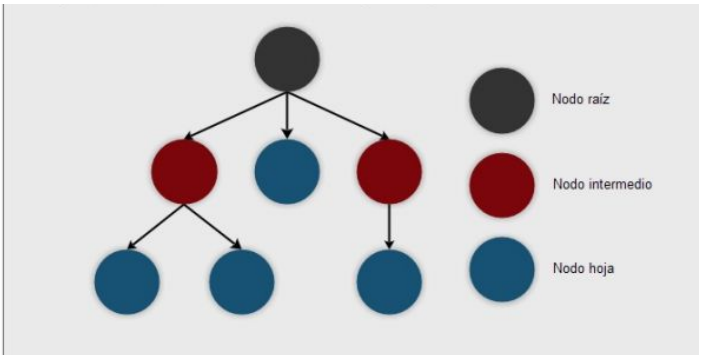
La raíz del árbol es el símbolo inicial de la gramática.

Los nodos intermedios del árbol son los no terminales de las producciones que se utilizan. Estos nodos intermedios tendrán tantos hijos como elementos tenga el lado derecho de la producción.

Los nodos hoja serán los terminales.

Derivaciones por la izquierda y la derecha

Las derivaciones se pueden realizar empezando por el símbolo no terminal que está más a la izquierda de la cadena, y a estas derivaciones se las denomina derivación por la izquierda, o bien se pueden hacer sustituciones en la cadena inicial empezando por el símbolo no terminal que está más a la derecha y a estas derivaciones se las denomina derivaciones por la derecha. Dependiendo de cuál escojamos se obtiene un resultado u otro:



ACTIVIDADES SEMANA 6 (OCT 26-30, 2020)

VIDEO 1

Un PDA M es un sistema (Q . ∑ . Γ . δ . q_0. Z_0. F), donde

- Q es un conjunto finito de estados
- ∑ es el alfabeto de entrada o es el alfabeto de la pila
- $q_o \epsilon$ Q es el estado de inicio
- $Z_0 \epsilon$ Γ el inicio símbolo en la pila (inicialización)
- F ϵ Q es el conjunto de estados finales
- δ es la función de transición, Q x U x { ϵ } x Γ a subconjuntos finitos de Q x Γ *

Una entrada típica de δ está dada por

δ (q.a.z) = {(p_1 γ 1) . ((p_2 . γ 2) ... (pm . ∩m)}

El PDA en el estado q, con el símbolo de entrada a y el símbolo de la parte superior de la pila z, puede ingresar a cualquiera de los estados p_1, reemplazar el símbolo z por la cadena γ i, y avance el cabezal de entrada en un símbolo.

- El símbolo más a la izquierda de γ i será la nueva parte superior de la pila
- a en la función anterior δ podría ser ϵ, en cuyo caso, el símbolo de entrada no se usa y el cabezal de entrada no avanza.
- Para un PDA M, definimos L(M), el lenguaje aceptado por M por estado final, es L (M) = {w | (q_0 . w . Z_0) □* (p . ϵ . γ), para algunos p ϵ F γ E Γ '}
- Definimos N(M), el lenguaje aceptado por M por pila vacía, como N(M) = {w | (q_0 . w . Z_0) □* (p . ϵ . γ), para algunos p ϵ Q.
- Cuando la aceptación es por pila vacía, el conjunto de estados finales es irrelevante y, por lo general, establecemos F = ϕ
- Al igual que en el caso de NFA y DFA, PDA también tiene dos versiones: NPDA y DPDA
- Sin embargo, NPDA son estrictamente más poderosas que la DPDA

- Por ejemplo, el lenguaje, L = {ww^R | w ϵ {a . b} +} solo puede ser reconocido por un NPDA y no por ningún DPDA

- Al mismo tiempo, el lenguaje, L = {WW^R | w ϵ {a. b} +}, puede ser reconocido por un DPDA

- En la práctica necesitamos DPDA, ya que tienen exactamente un movimiento posible en cualquier instante.

- Nuestros analizadores son todos DPDA

- El análisis es el proceso de construir un árbol de análisis sintáctico para una oración generada por una gramática determinada.

- Si no hay restricciones en el idioma y la forma de gramática utilizada, los analizadores para lenguajes libres de contexto requieren O(n^3) tiempo (siendo n el longitud de la cadena analizada)

    - Algoritmo de Cocke-Younger-Kasami

    - Algoritmo de Earley

- Los subconjuntos de lenguajes sin contexto normalmente requieren O(n) tiempo

    - Análisis predictivo usando gramáticas LL(1) (método de análisis de arriba hacia abajo)

    - Shift-Reduce análisis sintáctico utilizando gramáticas LR(1) (método de análisis sintáctico ascendente)

- Análisis de arriba hacia abajo mediante el análisis predictivo, rastrea la derivación más a la izquierda de la cadena mientras se construye el árbol de análisis

- Comienza desde el símbolo de inicio de la gramática y "predice" la siguiente producción utilizada en la derivación

- Dicha "predicción" es ayudado por tablas de análisis (construidas fuera de línea)

- La siguiente producción que se utilizará en la derivación se determina usando el siguiente símbolo de entrada para buscar la tabla de análisis (símbolo de anticipación)

- La tabla de análisis contiene más de una producción

- En el momento de la construcción de la tabla de análisis, si dos producciones se vuelven elegibles para colocarse en el mismo espacio de la tabla de análisis, la gramática se declara no apta para el análisis predictivo

Sea la gramática dada G

- La entrada se amplía con k símbolos, $\$^k$, k es el avance de la gramática
- Introduce una nueva S', no terminal, y una producción, S' à S$\$^K$, donde S es el símbolo de inicio de la gramática dada
- Considere solo las derivaciones a la izquierda y asuma que la gramática no tiene símbolos inútiles
- Una producción A à a en G se llama producción LL(k) fuerte, si en G

  S' è* WA γ è W α γ è* wzy

  S' è* w'A δ è w' β δ è* w'zx

  |z| = k . z ϵ $\sum^*$. w y w' ϵ $\sum^*$, entonces α = β
- Una gramática (no terminal) es fuerte LL(k) si todas sus producciones son fuertes LL(k)
- Las gramáticas LL(k) fuertes no permiten que se usen diferentes producciones del mismo no terminal incluso en dos derivaciones diferentes, si los primeros k símbolos de las cadenas producidas por α γ y β δ son iguales.
- Ejemplo: S à Abc | aAcb, A à ϵ|b|c

  S es un LL(1) no terminal fuerte
- S' è S$ è Abc$ è bc$. Bbc$ y cbc$, en aplicación de las producciones, A à ϵ, A à b, y A à c, respectivamente. z = b, b o c, respectivamente.
- S' è S$ è aAcb$ è acb$. abcb$ y accb$, según la aplicación de las producciones, A à ϵ, A à b, y A à c, respectivamente. z = a, en los tres casos
- En este caso, w = w' = ϵ, α = Abc, β = aAcb, pero z es diferente en las dos derivaciones, en todas las cadenas derivadas
- Por lo tanto, el no terminal S es LL(1).

A no es fuerte LL(1)

- S' è* Abc$ è bc$, w è ϵ, Z = b, α = ϵ (A à ϵ)

S' è* aAbc$ è bbc$, w' = ϵ, Z = b, β = b (A à b)

- Aunque las búsquedas anticipadas son las mismas (z = b), a ≠ β, y por lo tanto, la gramática no es fuerte LL(1)

A no es fuerte LL(2)

- S' è* Abc$ è bc$, w = ϵ, Z = bc, α = ϵ (A → ϵ)

S' è* aAcb$ è abcb$, w' = a, Z = bc, β = b (A → b)

- Aunque los lookaheads son los mismos (z = bc), a ≠ β, y por lo tanto, la gramática no es fuerte LL(2)

A es fuerte LL(3) porque las seis cadenas (bc$, bbc, cbc, cb$. bcb, ccb) se pueden distinguir usando 3-símbolo anticipado (los detalles son para el trabajo a domicilio)

- Llamamos LL(1) fuerte como LL(1) de ahora en adelante y no consideraremos las búsquedas anticipadas más largas que 1
- La condición clásica para la propiedad LL(1) usa conjuntos FIRST y FOLLOW
- Si α es cualquier cadena de símbolos gramaticales (α ϵ (N U T)"), luego FIRST (α) = {a | a ϵ T. y a è* ax. X ϵ T*}

FIRST (ϵ) = {ϵ}

- Si A es un no terminal, entonces

FOLLOW (A) = {a | S è* αAaβ, α. β ϵ (N U T)* . a ϵ T U {$}}

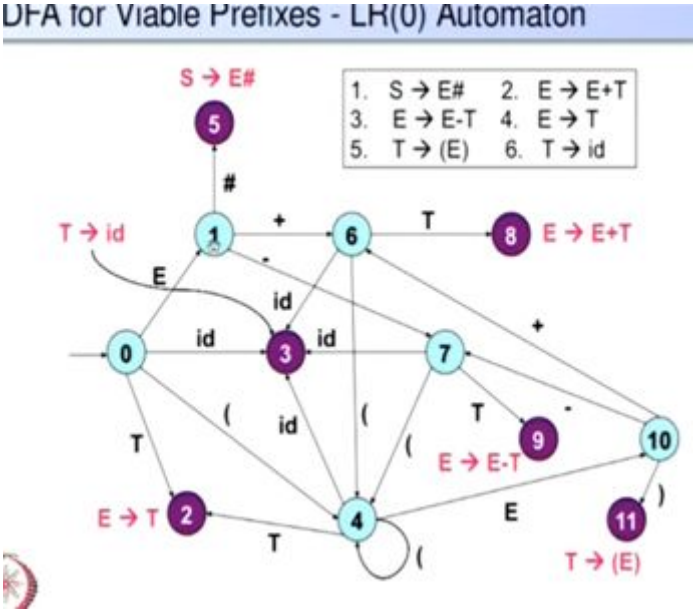
- FIRST (α) está determinado por solo α, pero FOLLOW (A) está determinado por el "contexto" de A, es decir, las derivaciones en las que aparece A.

VIDEO 2

Construir un autómata LR 0

Ejemplo

Como ocurre el análisis LR 0 con respecto a este DFA en particular.



Consideremos la identificación de cadena simple que se deriva de esta gramática. Así que nosotros comenzaremos con S yendo a E hash, aplicar esta producción. Y luego cuando aplicamos la producción E yendo a T y finalmente aplicamos la producción T, yendo a id. Obtenemos la identificación de la cadena todo esto se muestra en la siguiente imagen.

Esta tabla es una simplificación de todo un diagrama de un autómata.

1.	S → E#	2.	E → E+T
3.	E → E-T	4.	E → T
5.	T → (E)	6.	T → id

Ahora comenzamos a explicar cómo es que se realiza un autómata LR 0 del DFA.

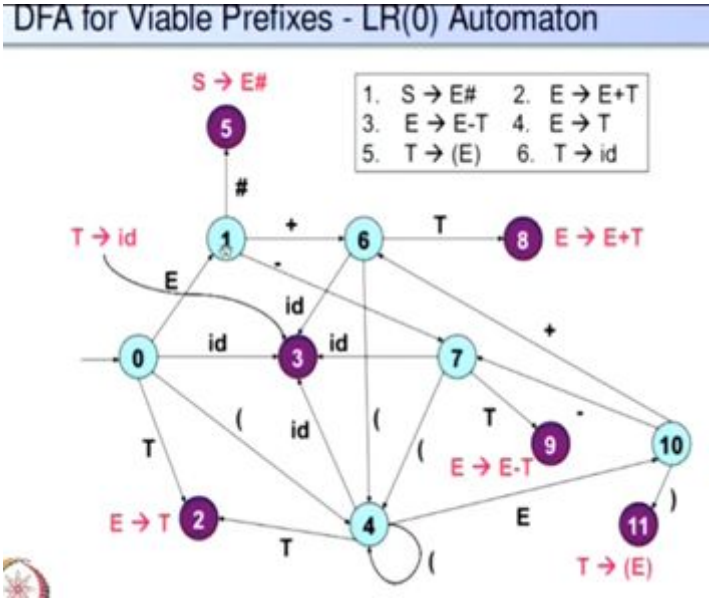
Empezaremos con el estado 0, que es el estado inicial y luego en la identificación de entrada vamos a el número de estado 3, recordando que el estado inicial es 0, ahora en la pila id se empuja a la pila como vemos la pila será la línea que ayude al estado cero a llegar al estado 3, nuevamente es empujado hacia abajo por la pila en lugar de 3 dice es un estado reducido y la reducción es por la producción T yendo a Id, luego vamos al estado numero 2 pero ahora hemos presionado el no terminal T en la pila y un estado número 2 también está en pila ahora dice en ese estado reduzca de E a t, sacamos el estado 2 y el T no terminal de la pila nuevamente obtendremos el estado 0 como la parte superior de la pila.

Y ahora, dado que el no terminal en el lado izquierdo es E aplicamos la función go to en el estado 0 y la E no terminal que nos llevará al estado número 1, lo que

sigue es que el siguiente símbolo de entrada que es un hash porque es un estado de cambio no es un estado de reducción y en el hash va el estado 5 donde se presenta una reducción de S que va a E hash como se indica en el estado 5.

Ahora del estado 1 al 6 nos llevará por el camino del mas y del estado 6 al id y esto significa que hay una reducción, entonces recordando que 3 y 6 están en la pila id, después el estado 6 nos llevará en el no terminal T y esto nos llevará al estado 8 donde tenemos en la pila E à E+T lo que indica una reducción que vamos por el camino E luego por el camino + y por último por el camino T.

Así es como se explica un autómata de este estilo sabiendo cómo van los caminos de que a qué estados están yendo como es que se llama un estado cuando nos encontramos un estado que contenga por ejemplo Eà E + T esto en pocas palabras en la reducción de un camino para llegar al estado donde se encuentra.



Cambiar y reducir la acción

Si un estado contiene un elemento de la forma [Aà alfa] se llama reducir artículo.

Entonces una reducción de a hasta alfa es la acción en ese estado, si no hay “elementos reducidos” en un estado, entonces se debe cambiar la acción apropiada.

Podría haber cambios-reducir conflictos o reducir-reducir conflictos en un estado.

Ambos elementos de cambio y reducción están presentes en el mismo estado como (conflicto S-R) o hay más de un elemento de reducción en un estado como (conflicto R-R).

Es normal tener más de un elemento de turno en un estado si es posible que haya conflictos de turno.

Si no hay conflictos S-R o R-R en ningún estado de un DFA LR0 entonces la gramática es LR(0), de lo contrario, no es LR(0).

Cómo construir la tabla de análisis de SRL

Se dispone de una pila en la que se pueden almacenar estados (enteros no negativos) o símbolos de la gramática (terminales o no terminales). También se dispone de un par de tablas a las que se les da el nombre de acción e ir a. La tabla acción contiene las acciones que puede ejecutar el autómata.

Las acciones se especifican con una letra que especifica la acción y el número de estado al que debe ir el autómata luego de ejecutarla.

Las mismas pueden ser:

1. Desplazamiento: se simboliza con la letra d y un número que indica el estado al que debe pasar el autómata.

2. Reducción: se simboliza con la letra r y un número que indica el número de regla por la cual se reduce.

3. Aceptar: se simboliza con la letra a. No tiene un número de sufijo puesto que no hace falta; el autómata se detiene automáticamente al encontrar esa orden.

4. Nada: si una entrada en esta tabla no contiene alguna de las tres alternativas anteriores entonces en la posición actual hay error de sintaxis. Algoritmos usados por el Generador de Analizadores Sintácticos 29 La tabla ir_a contiene números enteros no negativos que son los estados a los que tiene que ir el autómata luego de ejecutar una reducción.

VIDEO 3

ANÁLISIS SINTÁCTICO DE DESCENSO RECURSIVO

- Estrategia de análisis de arriba hacia abajo
- Un procedimiento de función para cada no terminal
- Las funciones se llaman recursivamente, según la gramática
- Pueden ser automáticamente generadas por la gramática
- La codificación manual también fácil

- La recuperación de errores es superior

GENERACIÓN AUTOMÁTICA DE ANALIZADORES SINTÁCTICOS RD

- El esquema está basado en la estructura de producción
- La función get_token() obtiene el siguiente token del analizador léxico
- La función error() imprime el mensaje de error

ANÁLISIS DE ABAJO HACIA ARRIBA

- Construye pequeños árboles de análisis y después junta estos árboles para crear uno más grande
- A este proceso se le conoce como reducción
- Se implementa el análisis shift-reduce
- Se utiliza el concepto de handle para detectar cuándo realizar reducciones

El algoritmo de análisis tiene 4 funciones principales:

Shift: Esta función sirve para que el siguiente símbolo introducido se posicione en lo alto de la pila

Reduce: Esta función aplica una regla gramatical a algunos de los árboles del análisis reciente, uniéndose como un árbol con un nuevo símbolo

Accept: Este indica que se completó satisfactoriamente el análisis

Error: Este se encarga de llamar a la rutina de recuperación

ANÁLISIS LR

- Son generados automáticamente utilizando los generadores de análisis
- Es método general de shift-reduce de no retroceso

ACTIVIDADES SEMANA 7 (NOV 2-6, 2020)

VÍDEO 1

El análisis sintáctico LR es un método de análisis sintáctico ascendente y es sinónimo de exploración de izquierda a derecha con la derivación más a la derecha en reversa.

LR(k). Es escaneo de izquierda a derecha con derivación más a la derecha en reversa, siendo k el número de tokens de búsqueda anticipada:

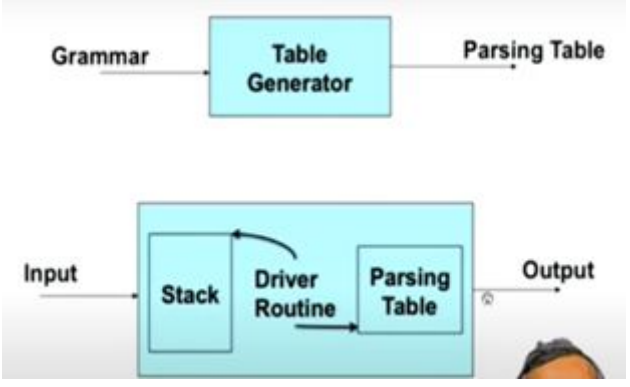
K= 0,1 son de interés práctico.

Los analizadores sintácticos LR también se generan automáticamente utilizando generadores de analizadores sintácticos.

Las gramáticas LR son un subconjunto de CFGS para el que se pueden construir analizadores sintácticos LR y son gramáticas de libre contexto.

La siguiente imagen es un generador de analizador sintáctico, el generador es un dispositivo de gramática como entrada y genera un análisis denominada tabla de análisis LR. La tabla encaja en otra caja que contiene una pila y una rutina de controlador, todo este conjunto es el analizador.

Entonces aquí se apiló la rutina del controlador y la tabla del análisis juntas, toma el programa como entrada y entrega como salida posible una sintaxis libre.



Configuración del analizador LR.

Una configuración de un analizador LR es:

(S0X1S2X2……XmSm aiai+1….an) donde apilar entrada no gastada S0,S1…..Sm, son los estados del analizador, y X1,X2……Xm son símbolos gramaticales (terminales o no terminales).

Una configuración inicial del analizador (S0.a1a2...an) donde , S0 es el estado inicial del analizador , y a1a2.....an es la cadena que se analizará.

La tabla puede obtener dos partes de análisis:

1.- La tabla action puede tener cuatro tipos de entrada: reducción, aceptación o error.

2.- La tabla goto proporciona la información del estado siguiente información de estado que se utilizara después un movimiento reducido.

LR gramática.

Se dice que una gramática es LR(k) si para cualquier cadena de entrada dada, en cada paso de cualquier derivación más a la derecha, el encargado B se puede detectar examinando la cadena y es escaneado como máximo los primeros k símbolos de la cadena de entrada no utilizada t.

Entendamos cómo construir el estado finito determinista del autómata que rastrea los estados del analizador y nos dice cuándo cambiar y cuándo reducir.

Un prefijo viable de una forma enunciativa Bt, donde B denota el identificador es cualquier prefijo de O3. Un prefijo viable no puede contener símbolos a la derecha del identificador.

Ejemplo:

SàE#, EàE+T|E-T| T,TàID|(E)SàE#EàE+TàE+(E)# àE+(T)# àE+(ID)#

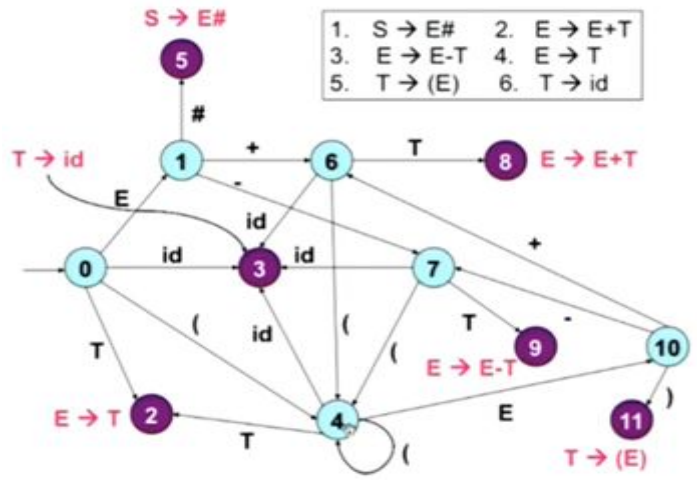
TEOREMA DEL ANÁLISIS SINTÁCTICO LR

El conjunto de todos los prefijos viables de todas las formas oracionales correctas de una gramática es un lenguaje regular.

El DFA de este lenguaje regular puede detectar identificadores durante el análisis LR. Cuando el DFA alcanza un estado de reducción, el prefijo viable correspondiente no puede crecer más y por lo tanto indica una reducción.

El compilador puede construir este DFA utilizando la gramática todos los analizadores sintácticos tienen dicho DFA incorporado.

Aquí mostramos un ejemplo de un DFA:



Entonces exactamente esto hay algunos estados que están en un azul verdoso y hay algunos otros estados que están en violeta, los estados que son de color violeta tienen una producción asociada a ellos y estos son los estados de reducción.

Ya se que los estados 5,8,3,2,9 y 11 son los estados de reducción y los otros estados que son 0,1,6,7,10 y 4 que son de color azul verdoso son los estados de cambio.

Un DFA en particular como se muestra en la anterior imagen no existe un estado final definido de hecho todos los estados son finales, por lo que en este DFA desde el estado inicial no importa que camino tome.

VIDEO 2

CAMBIAR Y REDUCIR ACCIONES

- Si no hay artículos reducidos en un estado, cambiar es la acción apropiada
- Pueden existir conflictos de cambiar-reducir y reducir-reducir en un estado
- Es normal tener más de un articulo cambiado en un estado
- Más de un articulo reducido debe estar presente en un estado
- Si no existe un conflicto SR o RR en algún estado de LR DFA, mientras la gramática es LR, este no es LR

Si la gramática no es LR, se tratara de resolver los conflictos en los estados utilizando un símbolo look-ahead

Los conflictos SR pueden ser resueltos usando el conjunto FOLLOW, se dice que la gramática es SLR

Todas la entrada no definidas en las reglas crearán un error

```
void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  C = {closure({S' → .S, $})};/* C is a set of LR(1) item sets */
  while (more item sets can be added to C) {
    for each item set I ∈ C and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if ((GOTO(I, X) ≠ ∅) && (GOTO(I, X) ∉ C))
        C = C ∪ GOTO(I, X)
```

Cada conjunto en C corresponde a un estado DFA

Este es el DFA que reconoce prefijos viables

VIDEO 3

Analizador LALR (1)

Los analizadores LR(1) tienen una gran cantidad de estados.

Para C, muchos miles de estados.

Un analizador SLR(1) (o LR(0) DFA) para C tendrán algunos cientos de estados con mucho conflicto.

LALR(1) los analizadores tienen exactamente el mismo número de estados que los analizadores SRL(1) para la misma gramática y se derivan de los analizadores SLR(1) para la misma gramática y se derivan de los analizadores LR(1).

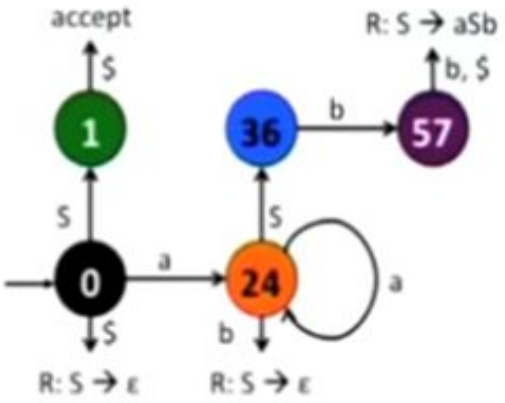
La parte central de los elementos LR(1) la parte que sigue a la omisión del símbolo de búsqueda anticipada es la misma para varios estados de LR(1) los símbolos de búsqueda anticipada serán diferentes.

Fusionar los estados con el mismo núcleo, junto con la búsqueda anticipada de símbolos y cambiarles el nombre.

Las partes ACCIÓN y GOTO de la tabla del analizador se modifican y fusionan las filas de la tabla del analizador correspondiente a los estados fusionados, reemplazar los nombres antiguos de los estados por los nuevos nombres correspondientes para los nuevos nombres correspondientes para los estados fusionados.

Por ejemplo

Si los estados 2 y 4 se fusionan en un nuevo estado 24, y los estados 3 y 6 se fusionan en un nuevo estado 36 todas las referencias a los estados 2,4,3 y 6 serán reemplazados por 24,24 36 y 36.


accept
R: S → aSb
$
b, $
1
36
b
57
S
S
0
a
24
a
S
b
R: S → ε
R: S → ε

ACTIVIDADES SEMANA 8 (NOV 9-13, 2020)

VIDEO 1

• La coherencia semántica que no se puede manejar en la etapa de análisis se maneja aquí

• Los analizadores no pueden manejar características sensibles al contexto de los lenguajes de programación

• Estas son semánticas estáticas de los lenguajes de programación y pueden ser verificadas por el analizador semántico

- Las variables se declaran antes de su uso
- Los tipos coinciden en ambos lados de las asignaciones
- Los tipos de parámetros y los números coinciden en la declaración y el uso

• Los compiladores solo pueden generar código para verificar la semántica dinámica de los lenguajes de programación en tiempo de ejecución

- si se producirá un desbordamiento durante una operación aritmética
- si los límites de la matriz se cruzarán durante la ejecución
- si la recursividad cruzará los límites de la pila
- si la memoria del montón será insuficiente

Muestras de comprobaciones semánticas estáticas en main

• Tipos de py tipo de retorno de dot prod match

• El número y tipo de los parámetros de dot_prod son los mismos tanto en su declaración como en el uso

• p es declarado antes del uso, lo mismo para ayb

Las muestras de comprobaciones semánticas estáticas en dot_prod o d e i se declaran antes de su uso

El tipo de d coincide con el tipo de retorno de dot_prod

El tipo de d coincide con el tipo de resultado de "*"

Los elementos de las matrices xey son compatibles con

• Sea G = (N. T. P. S) un CFG y sea V = NUT.

• Cada símbolo X de V tiene asociado un conjunto de atributos (indicados por XaXb, etc.)

• Dos tipos de atributos: heredados (indicados por Al (X)) y sintetizados (indicados por AS (X))

• Cada atributo toma valores de un dominio específico (finito o infinito), que es su tipo

• Los dominios típicos de atributos son, enteros, reales, caracteres, cadenas, booleanos, estructuras, etc.

• Se pueden construir nuevos dominios a partir de dominios dados mediante operaciones matemáticas como como producto cruzado, mapa, etc.

• matriz. un mapa, N → D, donde, N y D son dominios de números naturales y los objetos dados, respectivamente

• estructura: un producto cruzado, A, x A2 x ... x An, donde n es el número de campos en la estructura, y A, es el dominio del i-ésimo campo

VIDEO 2

Gramática de atributos.

Son gramáticas de libre contexto extendiendo símbolo del conjunto N unión T tiene asociados algunos atributos.

Hay dos tipos de atributos heredados y sintetizados.

1.- Un atributo sintetizado para un no terminal A en un nodo N de un árbol sintáctico se define mediante una regla semántica asociada con la producción en N. Observe que la producción debe tener a A como su encabezado. Un atributo sintetizado en el nodo N se define sólo en términos de los valores de los atributos en el hijo de N, y en el mismo N.

2.- Un atributo heredado para un no terminal B en el nodo N de un árbol de análisis sintáctico se define mediante una regla semántica asociada con la producción en el padre de N. Observe que la producción debe tener a B como un símbolo en su cuerpo. Un atributo heredado en el nodo N se define sólo en términos de los valores de los atributos en el padre de N, en el mismo N y en sus hermanos.

A la segunda clase de definiciones dirigidas por la sintaxis se le conoce como definiciones con atributos heredados. La idea de esta clase es que, entre los atributos asociados con el cuerpo de una producción, las flechas del grafo de dependencias pueden ir de izquierda a derecha, pero no al revés (de aquí que se les denomina “atributos heredados por la izquierda”). Dicho en forma más precisa, cada atributo debe ser:

1. Sintetizado.

2. Heredado, pero con las reglas limitadas de la siguiente manera. Suponga que hay una producción $A \rightarrow X1X2 \cdots Xn$, y que hay un atributo heredado Xi.a, el cual se calcula mediante una regla asociada con esta producción. Entonces, la regla puede usar sólo:

(a) Los atributos heredados asociados con el encabezado A.

(b) Los atributos heredados o sintetizados asociados con las ocurrencias de los símbolos X1, X2, ..., Xi−1 ubicados a la izquierda de Xi.

Un atributo no se puede sintetizar y heredar al mismo tiempo, pero un símbolo puede tener ambos tipos de atributos.

Los atributos de los símbolos se evalúan en un árbol de análisis sintáctico haciendo pases sobre el árbol de análisis sintáctico.

(c) Los atributos heredados o sintetizados asociados con esta misma ocurrencia de Xi, pero sólo en una forma en la que no haya ciclos en un grafo de dependencias formado por los atributos de esta Xi.

Los atributos sintetizados se calculan de abajo hacia arriba a partir de las hojas hacia arriba, siempre sintetizados a partir de los valores del atributo de los hijos del nodo.

Los nodos de la hoja (terminales) tienen atributos sintetizados (solo) inicializados por el analizador léxico y no se pueden modificar los atributos heredados fluyen desde el nodo en cuestión.

Ejemplo

AG para la evaluación de un número real a partir de su representación de cadena de bits.

Ejemplo: Si tiene una cadena de bits 110.101=6.625 este será su valor decimal.

La gramática libre N va a L punto R, L genera varios bits en el lado izquierdo del punto, y si la gramática L va a BL o B de manera similar, R genera los bits en el lado derecho del punto y la gramática es R que va a BR o BB es por supuesto un bit 0 o 1.

- $N \to L.R$. $L \to BL \mid B$. $R \to BR \mid B$. $B \to 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value\uparrow: real\}$.
  $AS(L) = \{length\uparrow: integer.\ value\uparrow: real\}$
  1. $N \to L.R$ $\{N.value\uparrow := L.value\uparrow + R.value\uparrow\}$
  2. $L \to B$ $\{L.value\uparrow := B.value\uparrow;\ L.length\uparrow := 1\}$
  3. $L_1 \to BL_2$ $\{L_1.length\uparrow := L_2.length\uparrow + 1;$
     $L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$
  4. $R \to B$ $\{R.value\uparrow := B.value\uparrow / 2\}$
  5. $R_1 \to BR_2$ $\{R_1.value\uparrow := (B.value\uparrow + R_2.value\uparrow)/2\}$
  6. $B \to 0$ $\{B.value\uparrow := 0\}$
  7. $B \to 1$ $\{B.value\uparrow := 1\}$

VIDEO 3

GRAMÁTICA DE LOS ATRIBUTOS

- Cada símbolo X o V viene asociado con un conjunto de atributos
- Hay dos tipos de atributos: Heredados y sintetizados
- Cada atributo toma valores de un dominio especificado

ALGORITMO DE EVALUACIÓN DE ATRIBUTOS PARA LAGS

```
Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values
void dfvisit(n: node)
{  for each child m of n, from left to right do
    {  evaluate inherited attributes of m;
       dfvisit(m)
    };
   evaluate synthesized attributes of n
}
```

GRAMÁTICA DE LA TRADUCCIÓN DE ATRIBUTOS

- Aparte de las reglas de computacion, algunos programas segmentan ese rendimiento
- Como resultado de estas acciones de segmento de código, el orden de evaluación puede ser constreñido

- Estas acciones pueden ser añadidas a SAG y LAG

ACTIVIDADES SEMANA 11 (30 NOV-4 DIC, 2020)

GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO (CONTEXT-FREE GRAMMARS)

Para describir la sintaxis del lenguaje de programación, necesitamos una notación más poderosa que las expresiones regulares que aún conduce a reconocedores eficientes. La solución tradicional es utilizar una gramática libre de contexto (cfg).

Una gramática libre de contexto, G, es un conjunto de reglas que describen cómo formar oraciones. La colección de oraciones que se pueden derivar de G se denomina lenguaje definido por G, denotado G. El conjunto de lenguajes definidos por gramáticas libres de contexto se denomina conjunto de lenguajes libres de contexto. Un ejemplo puede ayuda. Considere la siguiente gramática, que llamamos SN:

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \texttt{baa}\ \textit{SheepNoise} \\ &\mid \texttt{baa} \end{aligned}$$

La primera regla o producción dice "SheepNoise puede derivar la palabra baa seguida de más SheepNoise".SheepNoise es una variable sintáctica que representa el conjunto de cadenas que se pueden derivar de la gramática. Nosotros llame a dicha variable sintáctica un símbolo no terminal. Cada palabra en el idioma definido por la gramática es un símbolo terminal. La segunda regla dice "SheepNoise también puede derivar la cadena baa"**.**

Para derivar una frase, empezamos con una cadena prototipo que contiene sólo el símbolo de la meta, SheepNoise. Elegimos un símbolo no terminal, α, en la cadena prototipo, elegimos una regla gramatical, $\alpha \rightarrow \beta$, y reescribimos α con β. Repetimos este proceso de reescritura hasta que la cadena prototipo no contenga más no terminales, en cuyo momento se compone enteramente de palabras, o símbolos terminales, y es una frase en el idioma.

DERIVACIÓN: es una secuencia de pasos de reescritura que comienza con el símbolo de inicio de la gramática y termina con una frase en el idioma.

FORMA SENTENCIAL: es una cadena de símbolos que se produce como un paso en una derivación válida.

GRAMÁTICA SIN CONTEXTO: Para un lenguaje L, su CFG define los conjuntos de cadenas de símbolos que son frases válidas en L.

SENTENCIA: Es una cadena de símbolos que pueden derivarse de las reglas de una gramática.

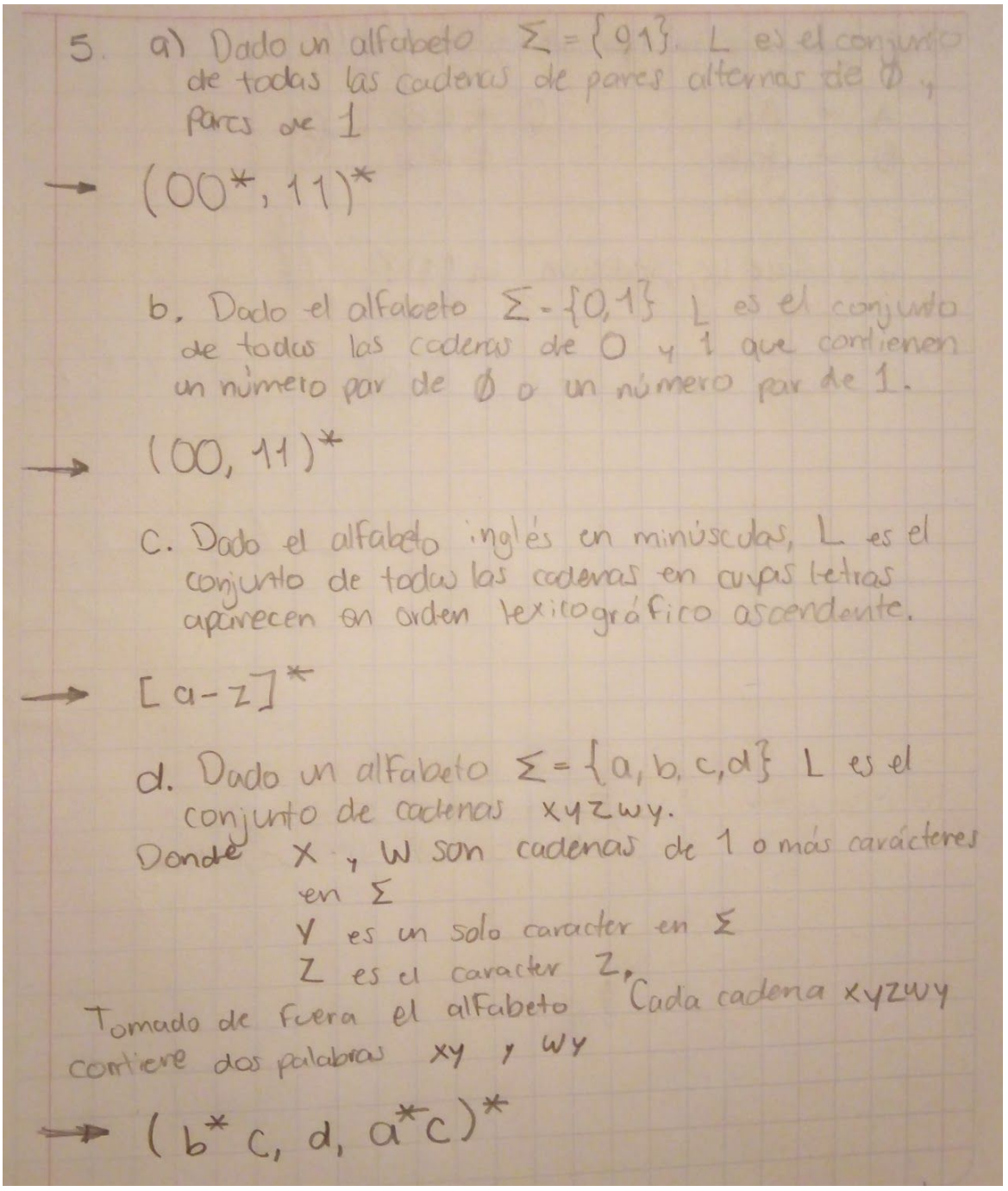
PRODUCCIÓN: Cada regla en CFG es llamada producción.

SÍMBOLO NO TERMINAL: Es una variable sintáctica utilizada en las producciones de una gramática.

SÍMBOLO TERMINAL: Es una palabra que puede aparecer en una frase.

Una palabra consiste en un lexema y su categoría sintáctica. Las palabras están representadas en una gramática por su categoría sintáctica.

EJEMPLOS Y EJERCICIOS.



5. a) Dado un alfabeto $\Sigma = \{0,1\}$. L es el conjunto de todas las cadenas de pares alternas de 0 y pares de 1

→ $(00^*, 11)^*$

b. Dado el alfabeto $\Sigma = \{0,1\}$ L es el conjunto de todas las cadenas de 0 y 1 que contienen un número par de 0 o un número par de 1.

→ $(00, 11)^*$

c. Dado el alfabeto inglés en minúsculas, L es el conjunto de todas las cadenas en cuyas letras aparecen en orden lexicográfico ascendente.

→ $[a-z]^*$

d. Dado un alfabeto $\Sigma = \{a, b, c, d\}$ L es el conjunto de cadenas xyzwy.
Donde X y W son cadenas de 1 o más caracteres en Σ
Y es un solo caracter en Σ
Z es el caracter Z,
Tomado de fuera el alfabeto Cada cadena xyzwy contiene dos palabras xy y wy

→ $(b^* c, d, a^* c)^*$

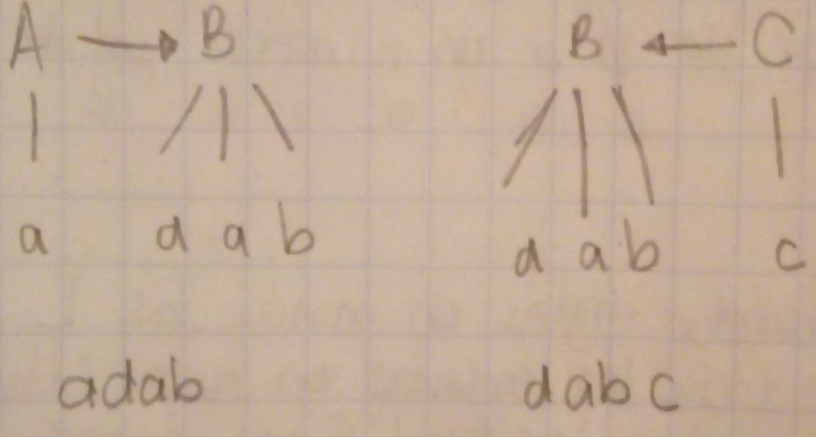
5. Considere la siguiente gramática

$A \rightarrow Ba$

$B \rightarrow dab \mid Cb$

$C \rightarrow cB \mid Ac$

¿Satisface la condición LL(1)?



adab

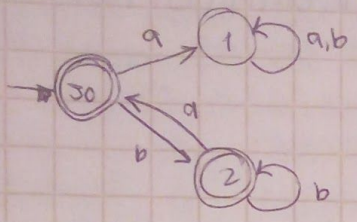
dabc

ACTIVIDADES SEMANA 12 (DIC 7-11, 2020)

CAPÍTULO 2

1. Describa informalmente los lenguajes aceptados por los siguientes Autómatas Finitos

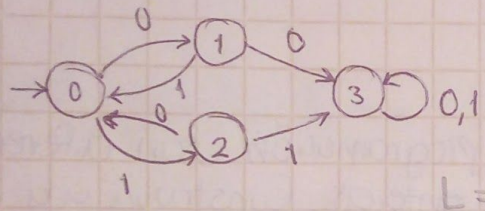
a.



$\Sigma = \{a, b\}$

$L = (a \mid ba)^* ab^* \cup b^*$

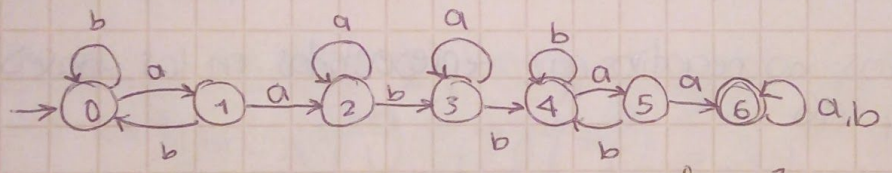
b.



$\Sigma = \{0, 1\}$

$L = (01 \mid 10)^* 0+01^* \cup 1+01^*$

c.



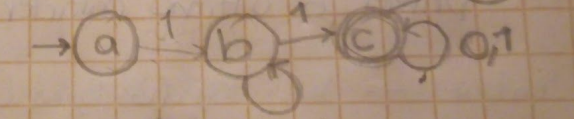
$\Sigma = \{a, b\}$

$L = (ab \mid b)^* a+a^* b+a^* b+b^* ab^* a+ab^*$

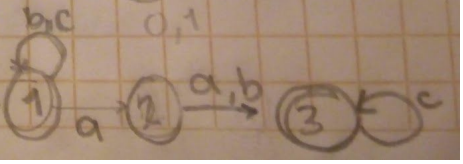
2. Construye un autómata finito aceptando los siguientes idiomas.

a. $\{w \in \{a,b\}^* \mid w\}$

b. $\{w \in \{0,1\}^* \mid w\}$

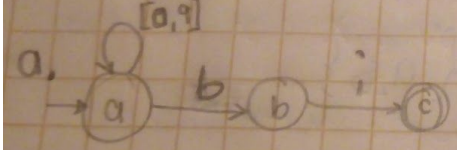


c. $\{w \in \{a,b,c\} \mid \text{in } w \ldots\}$

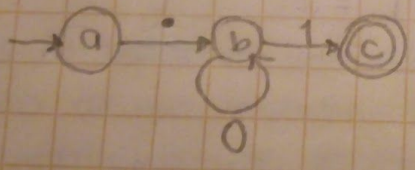


3. Cree FAS para reconocer (a) palabras que representan números complejos y (b) palabras que representan números decimales escritos en lenguaje científico notación.

a.



b.



4. Los diferentes lenguajes de programación usan diferentes notaciones para representar enteros. Construya una expresión regular para cada uno de los siguientes:

a. Enteros no negativos en C representados en las bases 10 y 16

$$ER. = (1^*[0,9]) \cup [A-F]$$

b. Enteros no negativos en Vhdl que pueden incluir guiones bajos (un guión bajo no puede aparecer como primer o último carácter)

```
entero <= to_integer(unsigned('1001'))
```

c. Moneda en dolares, representada como un número decimal positivo redondeado a la centésima mas cercana. Tales números comienzan con el caracter $, tienen comas que se separan cada grupo de 3 digitos a la izquierda del punto decimal y termina con dos dígitos a la derecha del punto decimal, por ejemplo $ 8,937.43 y $ 7,777,777.77.

E.R = $ + 8,1,2 + , + 3,4,5 + . + 6,7

5. Escriba una expresión regular para cada uno de los siguientes idiomas.

a. Dado un alfabeto $\Sigma = \{0,1\}$. L es el conjunto de todas las cadenas de pares alternos de 0 y pares de 1

E.R.=$(00^* 11)^*$

b. Dado un alfabeto $\Sigma = \{0,1\}$, L es el conjunto de todas las cadenas de 0 y 1 que contienen un número par de 0 o un número par de 1

E.R.=$(00 \mid 11)^* 01$

c. Dado el alfabeto inglés en minúsculas. L es el conjunto de todas las cadenas en cuyas letras aparecen en orden léxico ascendente.

E.R.= $[a-z]^*$

D. Dado un alfabeto $\Sigma = \{a, b, c, d\}$, L es el conjunto de cadenas $xyzwy$

Donde x y w son cadenas de uno o más carácteres en Σ

y es un solo caracter en Σ

z es el carácter z, tomado de afuera del alfabeto.

(Cada cadena $xyzwy$ contiene dos palabras xy y wy construidos a partir de las letras en Σ. Las palabras terminan en la misma letra y. Estan separados por z).

$ER. = (b^*c, d, a^*c)^*$

5. Escriba una expresión regular para describir cada una de las siguientes construcciones del lenguaje de programación.

a. Cualquier secuencia de tabulaciones y espacios en blanco.

E.R. = (\f, \n, \t, \v, \r, enter)

b. Comentarios en el lenguaje de programación C

E.R. = (/+*)([a-zA-z])(*+/)

C. Constantes de cadena (sin caracteres de escape)

E.R = (const + T-dato + nombre-const)

d. Números de punto flotante.

E.R. = ([0-9]* + . + [0-9]*)

7. Considere las tres expresiones regulares:

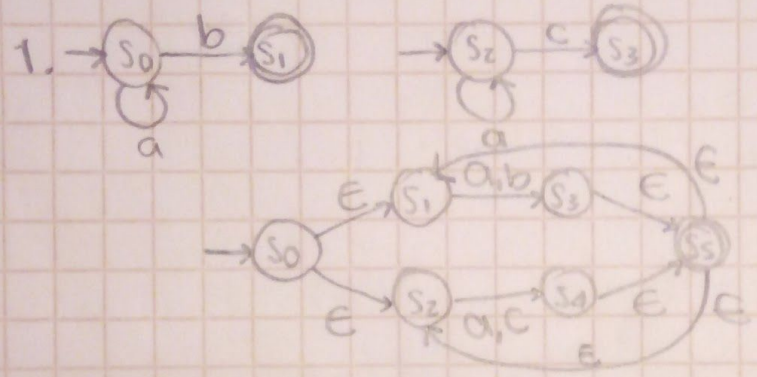
1 (ab | ac)*

2 (0|1)*1100 1*

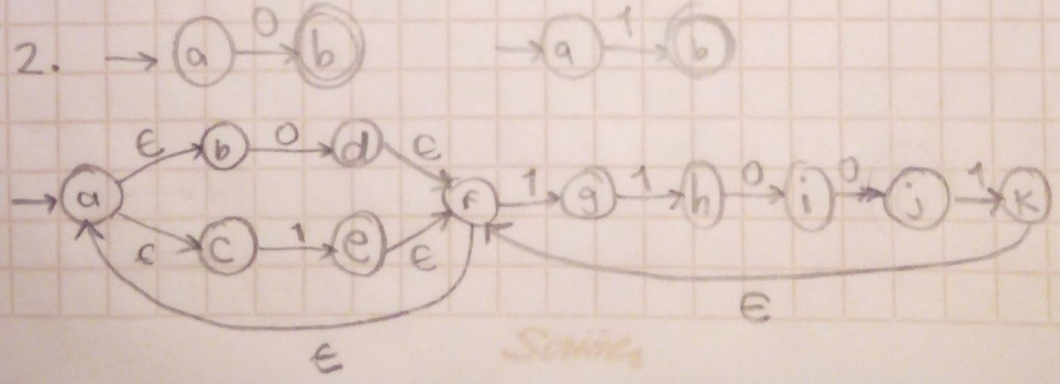
3 (01|10|00)*11

a. Utilice la construcción de Thompson para construir un NFA para cada RE

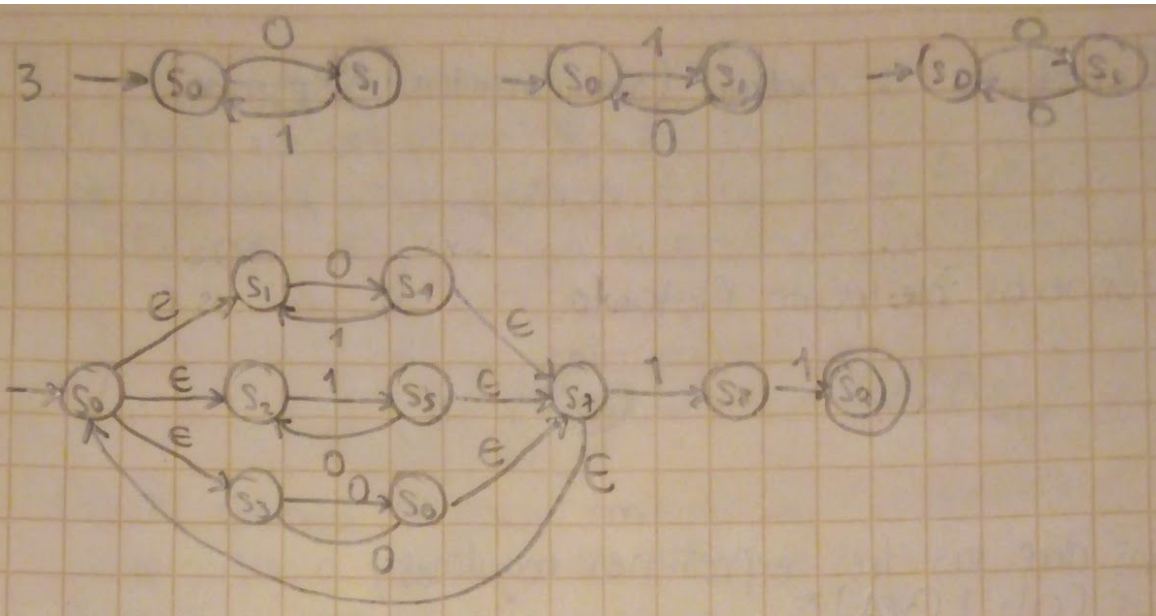
1.



2.

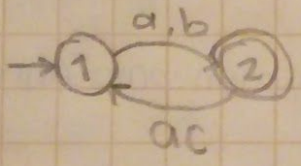


3

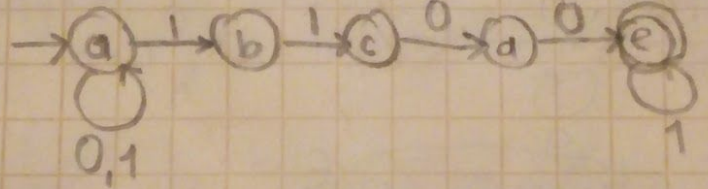


b. Convertir NFA en DFA

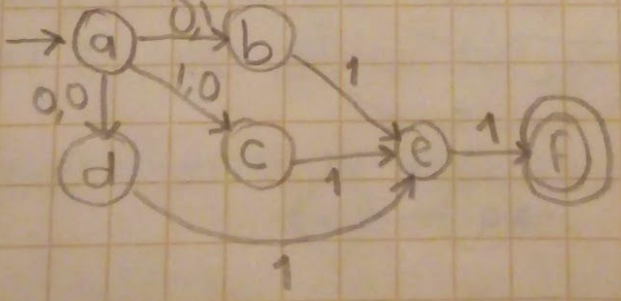
1.



2.

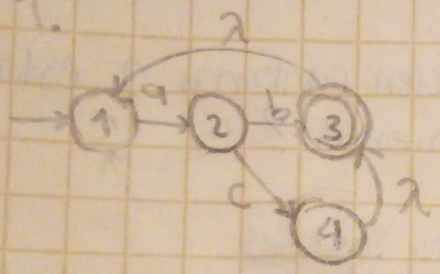


3.

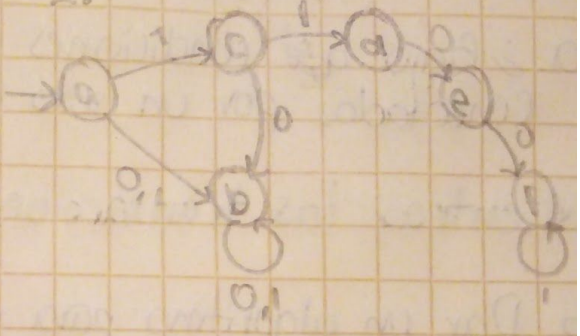


C. Minimizar el DFA.

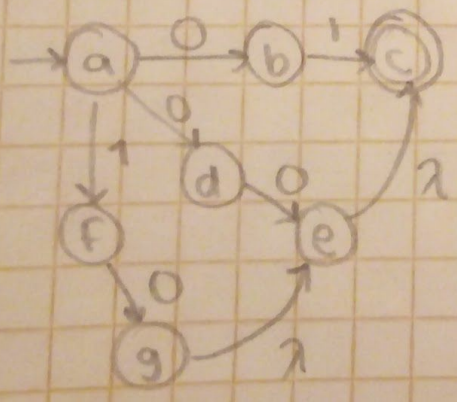
1.



2.

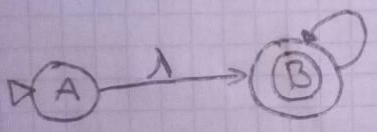


3.



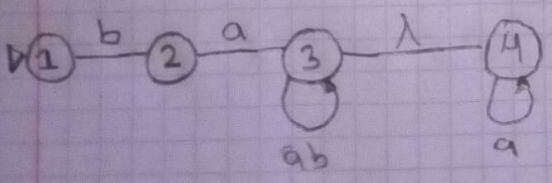
8) Una forma de probar que dos res son equivalentes es construir su minimizado dfas y luego compararlos, si difieren solo por los nombres de los estados, entonces el res son equivalentes. Utilice esta tecnica para comprobar los siguientes pares de res y indique si son equivalentes o no

a) $(0|1)^*$ y $(0^*|10^*)^*$

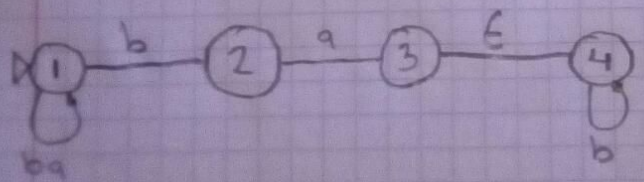


No son equivalentes

b) $(ba)+(a^*b^*|a^*)$ y $(ba)^*ba+(b^*|\epsilon)$



Si son equivalentes

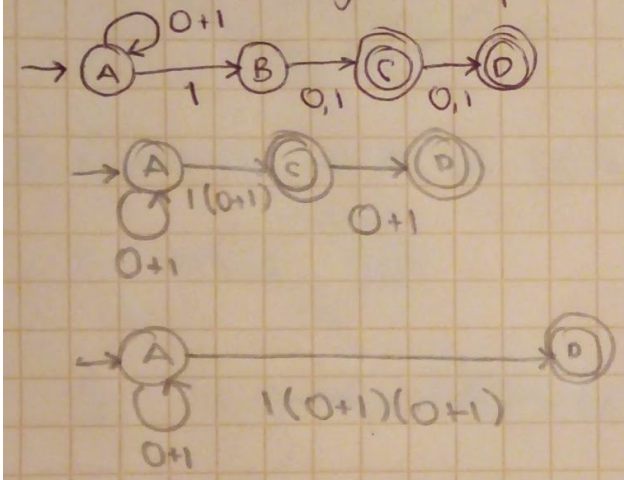


9. En algunos casos se pueden combinar 2 estados, conectados por un movimiento.

a. ¿Bajo qué condiciones se pueden combinar 2 estados conectados por un solo movimiento.?

Mientras las transacciones sean iguales

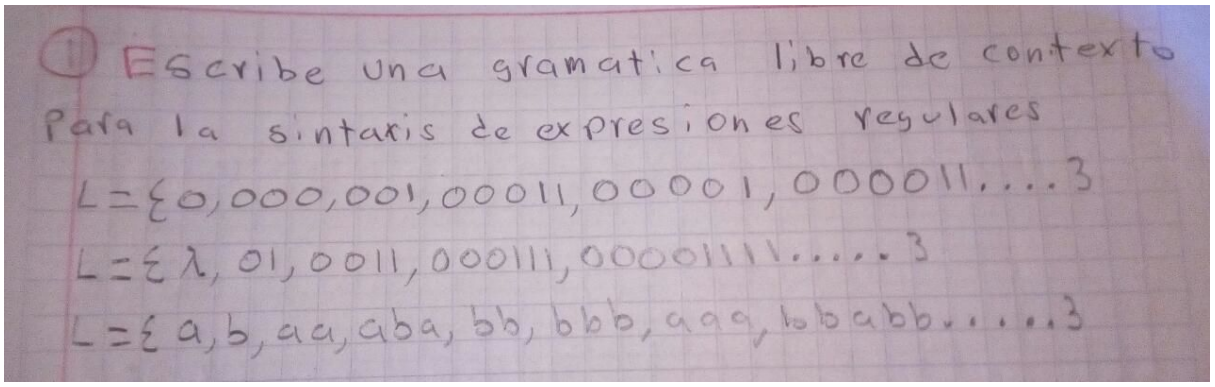
b. Dar un algoritmo para eliminar movimientos.

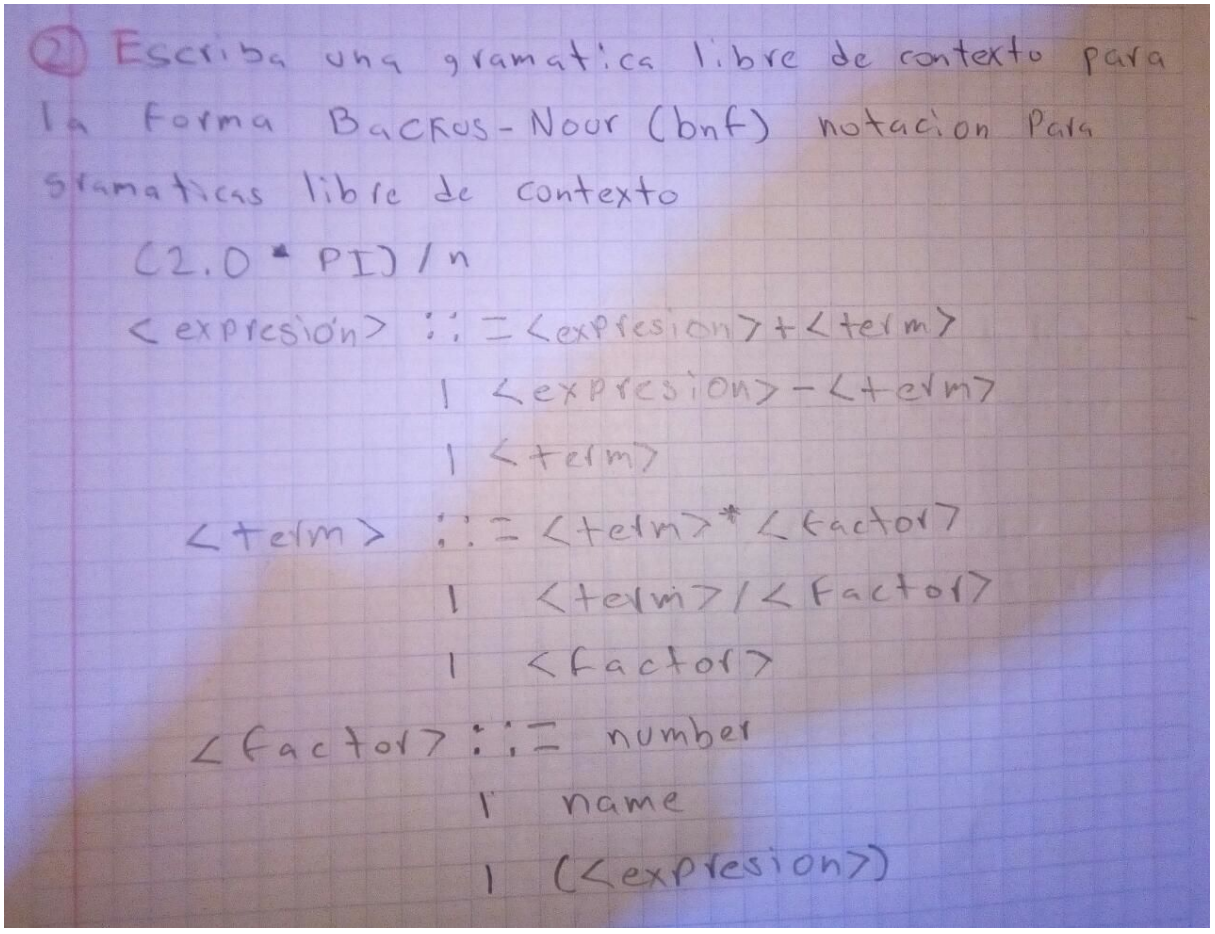


c. ¿Cómo se relaciona su algoritmo con el ε- función de cierre utilizada para implementar la construcción del subconjunto?

Es una redicción clara y fácil en donde sólo se eliminan estados y se unen procesos.

CAPÍTULO 3


1 Escribe una gramatica libre de contexto
para la sintaxis de expresiones regulares
L={0,000,001,00011,00001,000011,...3
L={λ,01,0011,000111,00001111.....3
L={a,b,aa,aba,bb,bbb,aaa,bbabb.....3


2 Escriba una gramatica libre de contexto para
la forma Backus-Nour (bnf) notacion Para
gramaticas libre de contexto
(2.0 * PI) / n
<expresion> ::= <expresion> + <term>
| <expresion> - <term>
| <term>
<term> ::= <term> * <factor>
| <term> / <factor>
| <factor>
<factor> ::= number
| name
| (<expresion>)

③ Cuando se le pregunto acerca de la definición de una gramatica inequivoca sin contexto en un examen, dos estudiantes dieron diferentes respuestas, el primero lo definio como "una gramatica donde cada oracion tiene un arbol de sintaxis unico por derivación mas a la izquierda". El segundo lo definio como una gramatica donde cada oracion tiene un arbol de sintaxis unico por cualquier derivación". ¿Cual es la correcta?

El primer alumno hace referencia al proceso para ir eliminando, así que la respuesta correcta es la del segundo alumno.

④ La siguiente gramatica no es adecuada para un analizador predictivo descendente. Identifique el problema y corrijalo reescribiendo la gramatica. Demuestre que su nueva gramatica satisface las 11 (1) condición

$L \to Ra \mid Qba$

$R \to aba \mid caba \mid Rbc$

$Q \to bbc \mid bc$

$L \to Ra \mid aba$

$R \to a^*b \mid Ca^*b \mid Rbc$

$Q \to b^*c \mid bc$

5: Considere la siguiente gramatica:

$A \rightarrow B\ a \qquad C \rightarrow c\ B$

$B \rightarrow d\ a\ b \qquad | \ A\ c$

$| \ C\ b$

Esta gramatica satisface la condición LL(1) ?

Justifica tu respuesta. Si no lo hace reescribalo como una gramatica LL(1) para el mismo idioma.

La gramatica si satisface la condicion LL(1) porque cumple con dicha estructura del proceso de manera por que no tiene ambigüedad etc.

6) Gramaticas que se pueden analizar de arriba hacia abajo, en un escaneo lineal de izquierda a derecha, con K la palabra anticipada se llama LL(K) gramaticas En el texto, el LL(1) la condición se describe en terminos de primero conjuntos ¿Como definirias el Primero conjuntos necesarios para describir un LL(K) ¿Condicion?

En LL first se encuentra descrita K necesaria Para ambiguar en el automata

⑦ Supongamos que un ascensor esta controlado por dos comandos: ↑ para subir el ascensor un piso y ↓ para bajar el ascensor un piso, Suponga que el edificio es arbitrariamente alto y que el ascensor comienza en el piso X.

Escribe un $1|(1)$ gramatica que genera secuencias de comandos arbitrarias que (1) nunca hacen que el ascensor baje del piso X y (2) siempre regrese el ascensor al piso X al final de la secuencia. Por ejemplo, ↑↑↓↓ y ↑↓↑↓ son secuencias de comandos validas, pero ↑↓↓↑ y ↑↓↓ no son. Por conveniencia, puede considerar valida una secuencia nula. Demuestra que tu gramatica es $1|(1)$.

$K - 1$

$X \to dK_X$ $\quad$ $X \to (K$

$1\ Cx$ $\quad$ $1\ A$

$Kx \to K \overset{1,0}{\to} X1$

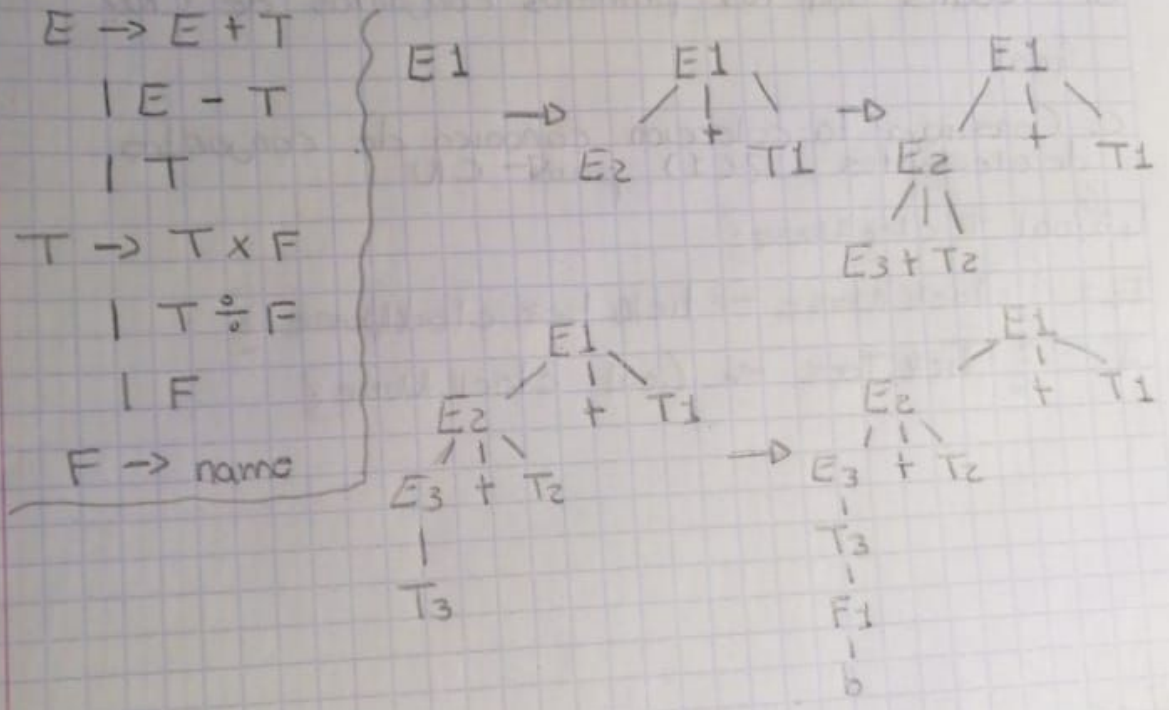
8.- Los analizadores de arriba hacia abajo y de abajo hacia arriba crean arboles de sintaxis en diferentes pedidos. Escriba un par de programas Top Down y Bottom Up, que requieran arbol de sintaxis e imprima los nodos en orden de construcción. De arriba hacia abajo debe ~~mientras~~ mostrar el orden de un analizador de arriba hacia abajo, mientras que de abajo hacia arriba debe mostrar el orden de un analizador de abajo hacia arriba.

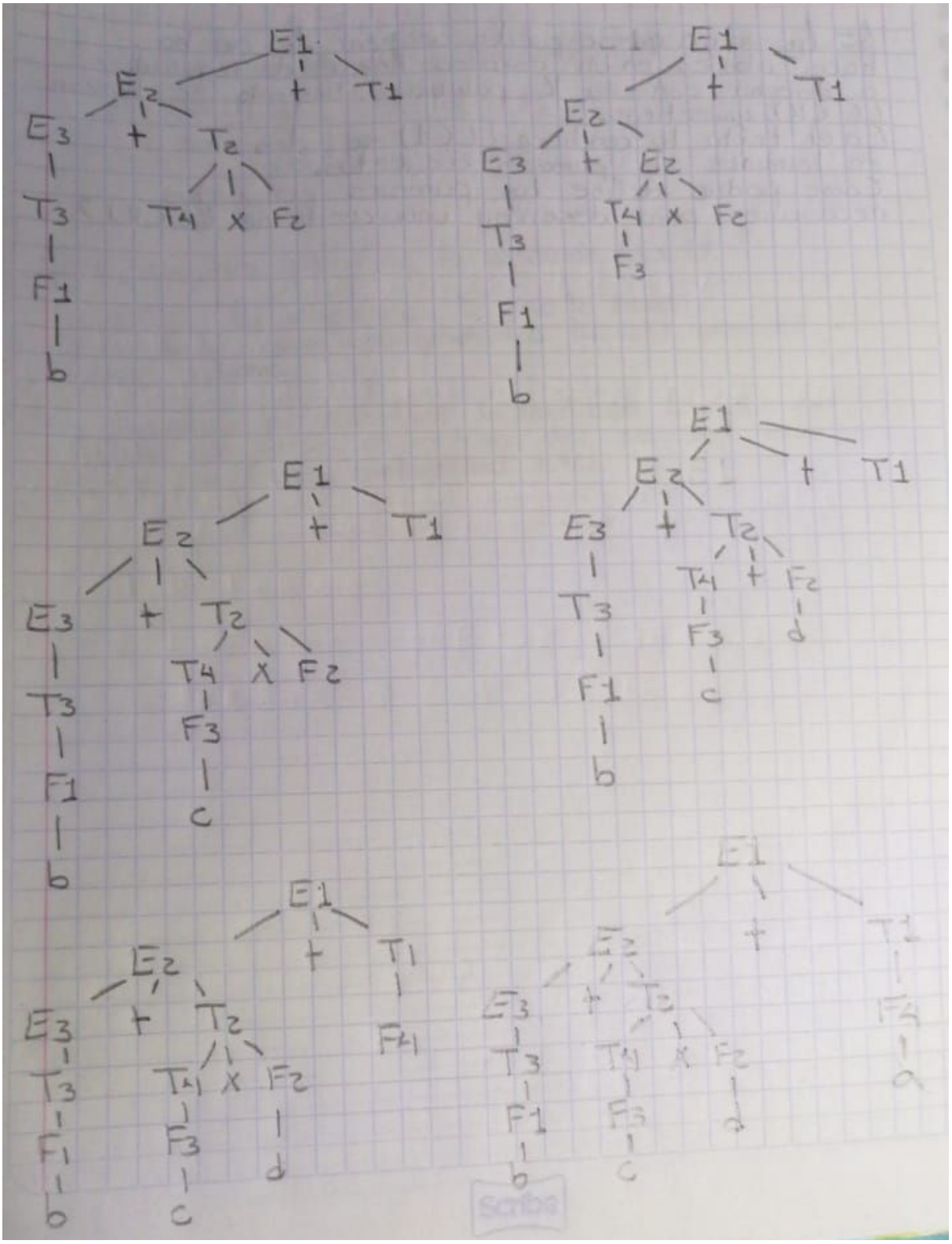
Tomamos la siguiente sintaxis de expresion de suma, resta multiplicación y division y declaración b + c x d + a como

E → E + T
| E - T
| T

T → T x F
| T ÷ F
| F

F → name




E1
E2
+
T1
E3
T2
T3
T4
X
F2
F1
F3
b
c
d

9.- El idioma ClockNoise (CN) esta representado por la siguiente gramatica.

Goal → clockNoise

clockNoise → clockNoise tick tock
| tick tock

a. ¿Cuales son los elementos LR(1)?

Clock Noise

b. ¿Cuales son los primeros conjuntos de CN?

Goal

c. Construya la coleccion canonica de conjuntos de elementos LR(1) para CN

$I_1 = \{Goal \rightarrow clocNoise\}$

$I_2 = \{clockNoise \rightarrow tick\ tock\ clockNoise\}$

$I_3 = \{TickTock \rightarrow Goal\ clockNoise\}$

ACTIVIDADES SEMANA 13 (DIC 14-18, 2020)

UNA INTRODUCCIÓN A LOS SISTEMAS DE TIPO

La mayoría de los lenguajes de programación asocian una colección de propiedades con cada valor de datos. A esta colección de propiedades se llaman el tipo de valor.
Los tipos pueden ser especificados por la pertenencia; por ejemplo, un número entero puede ser cualquier número entero i en el rango -2, 31 ≤ i < 2 31, o el rojo podría ser un valor en un tipo de colores enumerados, definidos como el conjunto {rojo, naranja, amarillo, verde, azul, marrón, negro, blanco}.
Los tipos pueden se especificarán mediante reglas; por ejemplo, la declaración de una estructura en c define un tipo.
El conjunto de tipos en un lenguaje de programación, junto con las reglas que usan tipos para especificar el comportamiento del programa, se llaman colectivamente un sistema de tipo.

ACTIVIDADES SEMANA 14 (ENE 7-8, 2021)

INTRODUCCIÓN A LAS REPRESENTACIONES INTERMEDIAS

Los compiladores suelen organizarse como una serie de pases. Como el compilador
obtiene el conocimiento sobre el código que compila, debe transmitir esa información de un paso a otro. Así, el compilador necesita una representación por todos los hechos que se derivan del programa.

Llamamos a esta representación una representación intermedia, o ir. Un compilador puede tener un solo ir, o puede tener una serie de irs que utiliza al transformar el código de la fuente en su lengua de destino.

El compilador no se refiere de vuelta al texto fuente; en su lugar, mira a la forma ir del código. Las propiedades del ir o irs de un compilador tienen un efecto directo en lo que el compilador puede hacer al código.

un compilador necesita una representación por todo el conocimiento que se deriva sobre el programa que se está compilando la mayoría de las pasadas en el compilador consume ir; el escáner es una excepción. La mayoría de las pasadas el compilador produce ir; los pases en el generador de código pueden ser excepciones.

ACTIVIDADES SEMANA 15 (ENE 11-15, 2021).

INTRODUCCIÓN A LA ABSTRACCIÓN DEL PROCEDIMIENTO.

El procedimiento es una de las abstracciones centrales en la mayoría de los lenguajes de programación modernos. Los procedimientos crean un entorno de ejecución controlado;
cada procedimiento tiene su propio almacenamiento privado.

Los procedimientos ayudan a definir interfaces entre los componentes del sistema; las interacciones entre los componentes son típicamente estructuradas a través de llamadas de procedimiento.

Mapa de ruta conceptual.

Para traducir un programa del lenguaje fuente a código ejecutable, el compilador debe mapear todas las construcciones del lenguaje fuente que el programa usa en operaciones y estructuras de datos en el procesador de destino.

El compilador necesita una estrategia para cada una de las abstracciones apoyadas por el lenguaje de origen. Las estrategias incluyen tanto algoritmos como estructuras de datos que están incrustados en el código ejecutable.