UNIVERSIDAD AUTÓNOMA DE CHIAPAS

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN
CAMPUS 1

Licenciatura en Ingeniería en Desarrollo y Tecnologías de Software

Act. 1.3 Investigar los Conceptos del analizador léxico

6 “M”

Materia: Compiladores

Docente: Luis Gutiérrez Alfaro

ALUMNO:
A 211387
Steven de Dios Montoya Hernández
GIT:

TUXTLA GUTIÉRREZ, CHIAPAS
Sábado, 26 de Agosto de 2023, 23:59

1.1 Expresiones regulares.-

En cómputo teórico y teoría de lenguajes formales, una expresión regular, o expresión racional, también son conocidas como regex o regexp,[3] por su contracción de las palabras inglesas *regular expression*, es una secuencia de caracteres que conforma un patrón de búsqueda. Se utilizan principalmente para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones.

Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto. Las expresiones regulares proporcionan una manera muy flexible de buscar o reconocer cadenas de texto. Por ejemplo, el grupo formado por las cadenas *Handel*, *Händel* y *Haendel* se describe con el patrón "H(a|ä|ae)ndel".

La mayoría de las formalizaciones proporcionan los siguientes constructores: una expresión regular es una forma de representar los lenguajes regulares (finitos o infinitos) y se construye utilizando caracteres del alfabeto sobre el cual se define el lenguaje.

1.2 Autómatas. un ejemplo

La **teoría de autómatas** es una rama de la teoría de la computación que estudia las máquinas abstractas y los problemas que éstas son capaces de resolver. La teoría de los autómatas está estrechamente relacionada con la teoría del lenguaje formal ya que los autómatas son clasificados a menudo por la clase de lenguajes formales que son capaces de reconocer. También son de gran utilidad en la teoría de la complejidad computacional.

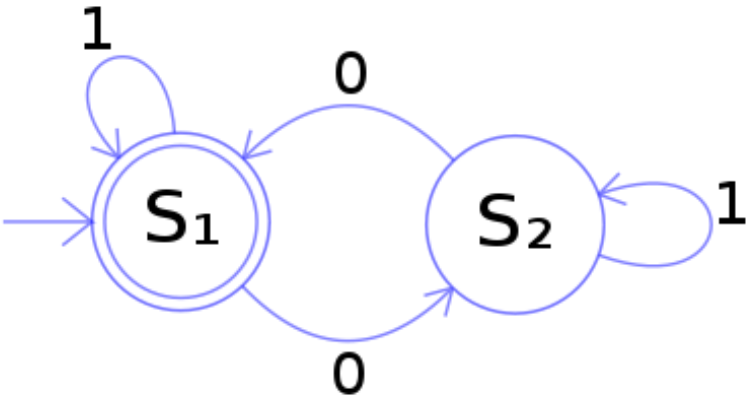
Un autómata es un modelo matemático para una máquina de estado finito (FSM sus siglas en inglés). Una FSM es una máquina que, dada una entrada de símbolos, "salta" a través de una serie de estados de acuerdo a una función de transición (que puede ser expresada como una tabla). En la variedad común "Mealy" de FSMs, esta función de transición dice al autómata a qué estado cambiar dados unos determinados estado y símbolo.

La entrada es leída símbolo por símbolo, hasta que es "consumida" completamente (piense en ésta como una cinta con una palabra escrita en ella, que es leída por una cabeza lectora del autómata; la cabeza se mueve a lo largo de la cinta, leyendo un símbolo a la vez) una vez la entrada se ha agotado, el autómata se detiene.

Dependiendo del estado en el que el autómata finaliza se dice que este ha aceptado o rechazado la entrada. Si este termina en el estado "acepta", el autómata acepta la palabra. Si lo hace en el estado "rechaza", el autómata rechaza la palabra, el conjunto de todas las palabras aceptadas por el autómata constituyen el lenguaje aceptado por sí mismo.

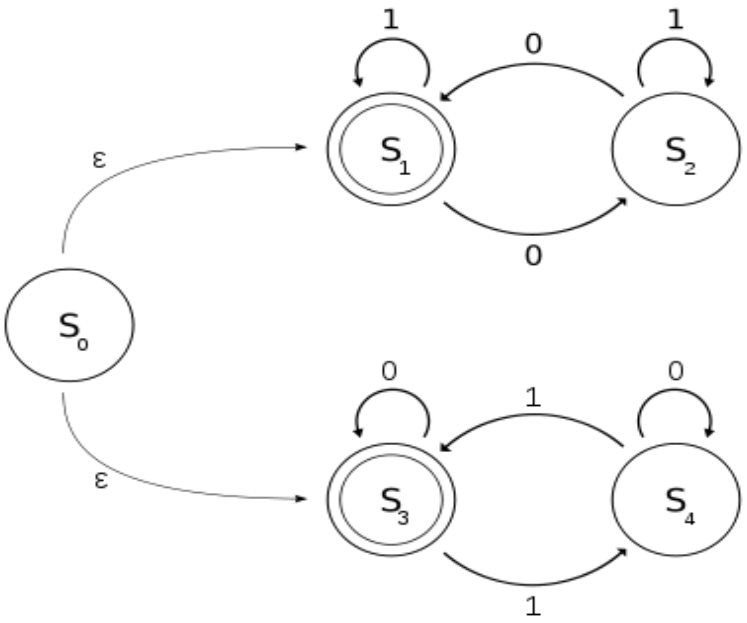
Autómata finito determinista (AFD)

Cada estado de un autómata de este tipo puede o no tener una transición por cada símbolo del alfabeto.



Autómata finito no determinista

Además de ser capaz de alcanzar más estados leyendo un símbolo, permite alcanzarlos sin leer ningún símbolo. Si un estado tiene transiciones etiquetadas con
e, entonces el AFND puede *encontrarse* en cualquier de los estados alcanzables por las transiciones e, directamente o a través de otros estados con transiciones
e. El conjunto de estados que pueden ser alcanzados mediante este método desde un estado que, se denomina la clausura
e de q.



Matrices de transición.- un ejemplo

Las matrices de transición son utilizadas en diversos campos para representar la probabilidad de pasar de un estado a otro en un sistema o proceso. Uno de los ejemplos más comunes es la matriz de transición en teoría de Markov, que describe cómo un sistema evoluciona de un estado a otro en función de probabilidades predefinidas.

Supongamos que tenemos un sistema simple de tres estados: A, B y C. Podemos representar las probabilidades de transición entre estos estados utilizando una matriz de transición. En este caso, estamos asumiendo que el sistema se encuentra en uno de estos tres estados en un momento dado y luego puede cambiar a otro estado en el siguiente paso con ciertas probabilidades.

Aquí está un ejemplo de una matriz de transición para este sistema:

	A	B	C
A	0.7	0.2	0.1
B	0.3	0.5	0.2
C	0.2	0.3	0.5

En esta matriz:

- La fila A representa el estado actual A.
- La fila B representa el estado actual B.
- La fila C representa el estado actual C.
- La columna A representa el estado siguiente A.
- La columna B representa el estado siguiente B.
- La columna C representa el estado siguiente C.

Los valores en la matriz representan las probabilidades de transición entre estados. Por ejemplo, en la celda correspondiente a la fila A y la columna B (0.2), la probabilidad de pasar del estado A al estado B en el siguiente paso es del 20%.
Esta matriz de transición es un ejemplo simplificado y puede ser utilizada en una variedad de contextos, como modelar el comportamiento de sistemas físicos, procesos industriales, sistemas de comunicación y más. La teoría de Markov y las matrices de transición son herramientas poderosas para analizar y predecir cómo evolucionan los sistemas a lo largo del tiempo en función de probabilidades específicas de cambio de estado.

Tabla de símbolos .- un ejemplo

Un compilador utiliza una tabla de símbolos para llevar un registro de la información sobre el ámbito y el enlace de los nombres. Se examina la tabla de símbolos cada vez que se encuentra un nombre en el texto fuente. Si se descubre un nombre nuevo o nueva información sobre un nombre ya existente, se producen cambios en la tabla.

Un mecanismo de tabla de símbolos debe permitir añadir entradas nuevas y encontrar las entradas existentes eficientemente. Los dos mecanismos para tablas de símbolos presentadas en esta sección son listas lioeal.es y tablas de dispersión. Cada esquema se evalúa basándose en el tiempo necesario para añadir n entradas y realizar e consultas. Una lista lineal es lo más fácil de implantar, pero su rendimiento es pobre cuando e y n sé vuelven más grandes. Los esquemas de dispersión proporcionan un mayor rendimiento con un esfuerzo algo mayor de programación y gasto de espacio. Ambos mecanismos pueden adaptarse rápidamente para funcionar con la regla del anidamiento más cercano.

Es útil que un compilador pueda aumentar dinámicamente, la tabla de símbolos durante la compilación. Si la tabla de símbolos tiene tamaño fijo al escribir el compilador, entonces el tamaño debe ser lo suficientemente grande como para albergar cualquier programa fuente. Es muy probable que dicho tamaño sea demasiado grande para la mayoría de los programas e inadecuado para algunos.

Entradas de la tabla de símbolos

Cada entrada de la tabla de símbolos corresponde a la declaración de un nombre. El formato de las entradas no tiene que ser uniforme porque la información de un nombre depende del uso de dicho nombre. Cada entrada se puede implantar como un registro que conste de una secuencia de palabras consecutivas de memoria. Para mantener uniformes los registros de la tabla de símbolos, es conveniente guardar una parte de la información de un nombre fuera de la entrada de la tabla, almacenando en el registro sólo un apuntador a esta información.

No toda la información se introduce en la tabla de símbolos a la vez. Las palabras clave se introducen, si acaso, al inicio.

El analizador léxico busca secuencias de letras y dígitos en la tabla de símbolos para determinar si se ha encontrado una palabra clave reservada o un nombre. Con este enfoque, las palabras clave deben estar en la tabla de símbolos antes de que comience el análisis léxico. En ocasiones, si el analizador léxico reconoce las palabras clave reservadas, entonces no necesitan aparecer en la tabla de símbolos. Si el lenguaje no convierte en reservadas las palabras clave, entonces es

indispensable que las palabras clave se introduzcan en la tabla de símbolos advirtiendo su posible uso como palabras clave.

La entrada misma de la tabla de símbolos puede establecerse cuando se aclara el papel de un nombre y se llenan los valores de los atributos cuando se dispone de la información. En algunos casos, el analizador léxico puede iniciar la entrada en cuanto aparezca un nombre en los datos de entrada. A menudo, un nombre puede indicar varios objetos distintos, quizás incluso en el mismo bloque o procedimiento. Por ejemplo, las declaraciones en C.

```
Int      x;

struct x { float y, z; };
```

utilizan x como entero y como etiqueta de una estructura con dos campos. En dichos casos, el analizador léxico sólo puede devolver al analizador sintáctico el nombre solo (o un apuntador al lexema que forma dicho nombre), en lugar de un apuntador a la entrada en la tabla de símbolos. Se crea el registro en la tabla de símbolos cuando se descubre el papel sintáctico que desempeña este nombre. Para las declaraciones de (1), se crearían dos entradas en la tabla de símbolos para x; una con x como entero y otra como estructura.

Los atributos de un nombre se introducen en respuesta a las declaraciones, que pueden ser implícitas. Las etiquetas son a menudo identificadores seguidos de dos puntos, así que una acción asociada con el reconocimiento de dicho identificador puede ser introducir este hecho en la tabla de símbolos. Asimismo, la sintaxis de las declaraciones de procedimientos especifica que algunos identificadores son parámetros formales.

Diferentes herramientas automáticas para generar analizadores léxicos.- un ejemplo

Herramientas para la construcción de compiladores . El uso y perfeccionamiento de los compiladores ha traído consigo el desarrollo de herramientas que aportan a la realización de los mismos, estas se han ido especializando en las diferentes fases del proceso de compilación, brindándole a los desarrolladores una serie de facilidades a la hora de diseñar e implementar un compilador. En el mundo existen diversas herramientas de apoyo de este tipo, desarrolladas en diferentes lenguajes de programación, las cuales responden a los intereses de los múltiples sistemas operativos. Entre las herramientas más utilizadas se pueden encontrar el Flex, Yacc, Lex, Bison entre otras.

Bison
Es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto en un programa en C que analice esa gramática. Es utilizado en un amplio rango de analizadores de lenguajes, desde aquellos usados en simples calculadoras de escritorio hasta complejos lenguajes de programación.

Lex
Es un generador de analizador léxico, que sirve para generar los token para la siguiente fase . La principal característica de Lex es que va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que se hayan definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que se debe diseñar con cuidado. Internamente Lex va a actuar como un autómata que localiza las expresiones regulares que se le describen, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado a esa regla.

Yacc
Es un programa para generar analizadores sintácticos. Las siglas del nombre significan "Yet Another Compiler Compiler", es decir, "Otro generador de compiladores más". Genera un analizador sintáctico (la parte de un compilador que comprueba que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje) basado en una gramática analítica.Yacc genera el código para el analizador sintáctico en el Lenguaje de programación C.

Flex

Es una herramienta para generar escáneres: programas que reconocen patrones léxicos en un texto. Flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. Estas herramientas de apoyo han sido reescritas para otros lenguajes, incluyendo Ratfor, EFL, ML,Ada, Java, Python, y Limbo. De esta forma se ha logrado una mayor utilización de las mismas en diferentes compiladores desarrollados sobre tecnologías libres. Teniendo en cuenta las características de las aplicaciones antes mencionadas, se ha escogido para la realización del compilador las herramientas Yacc y Lex. En muchos de los compiladores desarrollados en el mundo suelen ser utilizados juntos. Yacc utiliza una gramática formal para analizar un flujo de entradas, algo que Lex no puede hacer con expresiones regulares simples (Lex se limita a los autómatas de estados finitos simples). Sin embargo, Yacc no puede leer en un flujo de entradas simple, requiere una serie de símbolos. Lex se utiliza a menudo para proporcionar a Yacc estos símbolos.

Gramática libre de Contexto

Una gramática libre de contexto (GLC) es una descripción estructural precisa de un lenguaje. Formalmente es una tupla G=<Vn, Vt, P, S>, donde

• Vn es el conjunto finito de símbolos no terminales

• Vt es el conjunto finito de símbolos terminales ($Vn \cap Vt = \varnothing$)

• P es el conjunto finito de producciones que se pueden ver como

relaciones definidas en $Vn \times (Vn \cup Vt^*)$

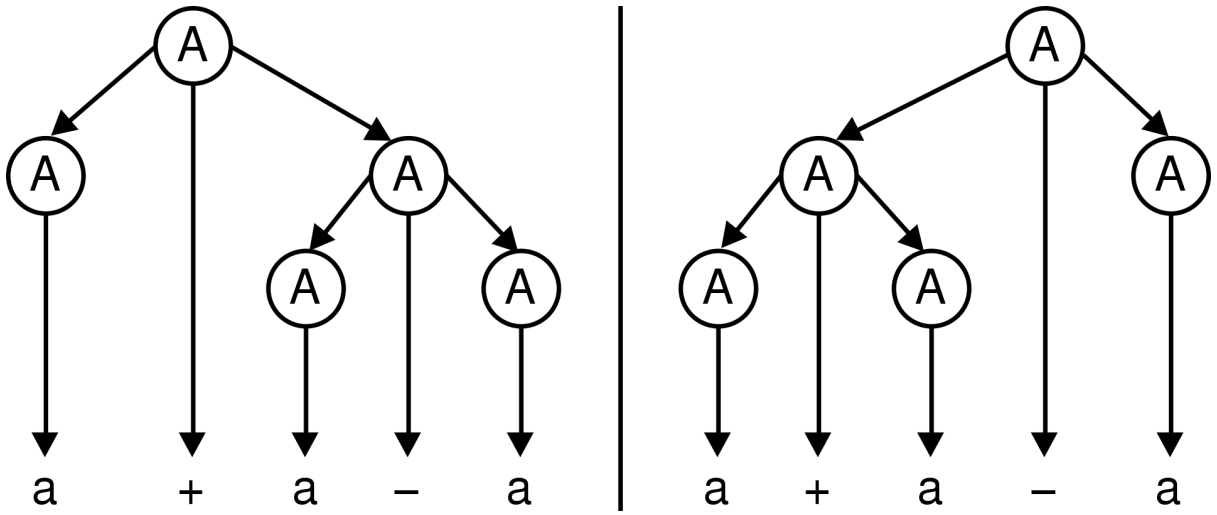
• S es el símbolo inicial de la gramática.

• Las producciones tienen la forma $A \rightarrow \alpha$ donde $A \in Vn$ y α es una expresión ya sea compuesta por símbolos no terminales como terminales ($\alpha \in (Vt \cup Vn)^*$) ó la cadena nula Ɛ

• Una producción puede verse como una regla de reescritura que indica cómo reemplazar símbolos no terminales por la expresión correspondiente en la parte derecha de la producción.

• Así partiendo de algún $A \in Vn$, se pueden aplicar las reglas de P hasta alcanzar a una expresión compuesta únicamente por símbolos terminales. A ésta expresión se le denomina frase o lexema. A las expresiones formadas por símbolos terminales y no terminales se les denomina formas de frase

En Ciencias de la Computación, una gramática ambigua es un Gramática libre del contexto para la que existe una cadena que puede tener más de una derivación a la izquierda, mientras una gramática no ambigua es una Gramática libre del contexto para la que cada cadena válida tiene una única derivación a la izquierda. Muchos lenguajes admiten tanto gramáticas ambiguas como no ambiguas, mientras otros lenguajes admiten solo gramáticas ambiguas. Cualquier lenguaje no vacío admite una gramática ambigua al tomar una gramática no ambigua e introducir una regla duplicada (el único lenguaje sin gramáticas ambiguas es el lenguaje vacío). Un lenguaje que solo admite gramáticas ambiguas se conoce como un Lenguaje Inherentemente Ambiguo, y existen lenguajes libres del contexto inherentemente ambiguos. Las gramáticas libres del contexto deterministas son siempre no-ambiguas, y son una subclase importante de GLCs no-ambiguas; existen GLCs no-deterministas y no-ambiguas simultáneamente.



La gramática libre del contexto

S → S + S | S − S | S * S | id
es ambigua dado que hay dos derivaciones a la izquierda para la cadena a + a + a:

A	→ A + A	A	→ A + A
	→ a + A		→ A + A + A (Primero A es reemplazada por A+A. La sustitución de la segunda A permitiría una derivación similar)
	→ a + A +A		→ a + A + A
	→ a + a +A		→ a + a + A
	→ a + a + a		→ a + a + a

En el siguiente ejemplo, la gramática es ambigua dado que existen dos árboles de derivación para la cadena a + a − a

El lenguaje que genera, aun así, no es inherentemente ambiguo; la siguiente es una gramática no-ambigua que genera el mismo lenguaje:

A → A + a | A − a | a

Las gramáticas son conjuntos de reglas que definen la estructura y sintaxis de un lenguaje. Estas reglas son utilizadas en programación y en el análisis de lenguajes naturales para determinar cómo se pueden construir oraciones o expresiones válidas en un determinado lenguaje.

Existen cuatro componentes clave en una gramática:

Símbolos terminales: Son los elementos básicos del lenguaje, como palabras o caracteres individuales.

Símbolos no terminales: Son variables que representan grupos de símbolos terminales y ayudan a definir la estructura del lenguaje.

Producciones: Son las reglas que indican cómo los símbolos no terminales y terminales pueden ser reemplazados por otros símbolos. Se expresan en la forma "A -> B", donde "A" es un símbolo no terminal y "B" es una secuencia de símbolos (terminales y/o no terminales).

Símbolo inicial: Es el símbolo no terminal con el que comienza cualquier construcción en el lenguaje.

Aquí tienes un ejemplo simple de una gramática para expresiones aritméticas básicas: Símbolos terminales: {+, -, *, /, números} Símbolos no terminales: {E, T} Símbolo inicial: E Producciones:

E -> E + T
E -> E - T
E -> T
T -> T * número
T -> T / número
T -> número

En esta gramática, "E" representa una expresión aritmética completa, y "T" representa un término dentro de la expresión. Las producciones indican cómo construir expresiones a partir de términos y cómo multiplicar/dividir términos por números. Por ejemplo, usando las reglas, podemos derivar la expresión "3 + 2 * 5" de la siguiente manera:

E (inicio)
E + T (aplicando E -> E + T)
T + T (aplicando E -> T)
número + T (aplicando T -> número)
número + T * número (aplicando T -> T * número)
número + número * número (aplicando T -> número)
número + número * número (aplicando T -> número)

Construir una gramática implica definir reglas precisas para generar las expresiones válidas en un lenguaje. Usaré el ejemplo de una gramática para generar oraciones simples en un lenguaje ficticio.

Supongamos que queremos construir una gramática para generar oraciones simples en el lenguaje "Fictiogés". En este lenguaje, una oración consta de un sujeto, un verbo y un objeto. Aquí está el proceso de construcción de la gramática paso a paso:

Identificar los componentes principales:

- Símbolos terminales: {sustantivos, verbos, artículos}
- Símbolos no terminales: {Oración, Sujeto, Verbo, Objeto}

Definir las producciones:

- Primero, definimos las producciones para las partes de la oración:
    - Sujeto -> sustantivos
    - Verbo -> verbos
    - Objeto -> sustantivos
- Luego, definimos la producción para la oración completa:
    - Oración -> Sujeto Verbo Objeto

Agregar reglas adicionales:

- Podemos agregar reglas para los artículos:
    - Artículo -> "el" | "la" | "un" | "una"
    - Sujeto -> Articulo sustantivos

Especificar reglas más detalladas:

- Podemos agregar variaciones a las reglas, como concordancia de género y número:
    - sustantivos -> "gato" | "perro" | "pelota" | ...
    - verbos -> "corre" | "salta" | "juega" | ...
    - Artículo -> "el" | "la" | "un" | "una"
    - Sujeto -> Articulo sustantivos
    - Objeto -> Artículo sustantivos

Ahora, podemos usar esta gramática para generar oraciones en "Fictiogés". Por ejemplo:

- Oración -> Sujeto Verbo Objeto
- Sujeto -> Articulo sustantivos
- Verbo -> "corre"
- Objeto -> Artículo sustantivos

Combinando estas producciones, obtenemos la oración: "El gato corre el perro".
Este proceso de construcción de gramáticas implica la definición cuidadosa de reglas que determinan cómo se pueden combinar los elementos para formar estructuras válidas en el lenguaje deseado.
Recuerda que este es un ejemplo simplificado, pero el proceso general es aplicable a la construcción de gramáticas más complejas para diversos lenguajes, incluidos los lenguajes de programación y los lenguajes naturales.

Las formas enunciativas se refieren a las estructuras gramaticales que se utilizan para hacer afirmaciones o declaraciones. Estas formas son esenciales en la comunicación, ya que nos permiten expresar hechos, opiniones o información de manera directa. Aquí tienes un proceso paso a paso junto con un ejemplo:

Proceso de Construcción de Formas Enunciativas:

Sujeto: El sujeto es la entidad sobre la cual se está haciendo la afirmación. Puede ser una persona, cosa, lugar u objeto.

Verbo: El verbo es la acción que se está realizando en la oración. Indica lo que el sujeto está haciendo.

Complementos: Los complementos pueden proporcionar información adicional sobre la acción o el sujeto. Pueden incluir objetos directos e indirectos, así como otros detalles.

Estructura gramatical: La estructura gramatical de una oración enunciativa suele seguir el orden sujeto-verbo-complementos, aunque puede variar dependiendo del idioma y del contexto.

Ejemplo de Construcción de Formas Enunciativas:
Supongamos que queremos construir una forma enunciativa simple en inglés. Utilizaremos el siguiente enunciado como ejemplo: "The cat is sleeping."

Sujeto: "The cat" (El gato)
Verbo: "is sleeping" (está durmiendo)
Complementos: Ninguno en este caso.
Estructura gramatical: Sujeto (The cat) + Verbo (is sleeping)

Siguiendo este proceso, hemos construido la forma enunciativa "The cat is sleeping", que comunica de manera directa que el gato está durmiendo.

Recuerda que este es un ejemplo simple, y las formas enunciativas pueden volverse más complejas a medida que se agregan más detalles y complementos. Además, las estructuras gramaticales pueden variar en diferentes idiomas.

El proceso de remoción de ambigüedad en gramáticas es fundamental para garantizar que las construcciones tengan una única interpretación clara y evitar confusiones en el análisis sintáctico. A continuación, te explicaré el proceso con un ejemplo.

Proceso de Remoción de Ambigüedad:

Identificación de la ambigüedad: Examina las producciones de la gramática para identificar las reglas que permiten múltiples formas de derivar la misma cadena.

Análisis de las derivaciones ambiguas: Encuentra ejemplos de cadenas que tienen múltiples derivaciones y diferentes interpretaciones bajo las producciones actuales.

Análisis de precedencia y asociatividad: Identifica los operadores en la gramática y decide sobre su precedencia y asociatividad. Esto ayudará a definir la forma en que se agrupan y operan los elementos.

Reescritura de producciones: Modifica las producciones ambiguas para reflejar la precedencia y la asociatividad. Esto a menudo implica dividir las producciones en diferentes niveles de precedencia y definir reglas claras para el orden en que se aplican.

Factorización y reorganización: En algunos casos, es posible factorizar y reorganizar las producciones para eliminar ambigüedades. Esto implica crear reglas más específicas que eviten ambigüedades.

Verificación y pruebas: Asegúrate de que las modificaciones realizadas no afecten la validez de las construcciones válidas. Comprueba que las ambigüedades se hayan eliminado efectivamente.

Ejemplo de Remoción de Ambigüedad:
Supongamos que tenemos la siguiente gramática ambigua para expresiones aritméticas con operadores de suma y multiplicación:

Producciones originales:

Expresion -> Expresion + Expresion
Expresion -> Expresion * Expresión
Expresión -> número

La cadena "2 + 3 * 4" puede derivarse de dos maneras diferentes: como "2 + (3 * 4)" o como "(2 + 3) * 4", lo que genera ambigüedad.

Pasos para eliminar la ambigüedad:
Podemos eliminar la ambigüedad introduciendo reglas de precedencia y factorización:
Producciones modificadas:

Expresion -> Termino + Expresión
Expresion -> Termino
Termino -> Factor * Término
Termino -> Factor
Factor -> número

Ahora, la cadena "2 + 3 * 4" solo tiene una interpretación: "(2 + (3 * 4))".
Mediante la introducción de reglas de precedencia y la reestructuración de las producciones, hemos eliminado la ambigüedad y asegurado una interpretación única para las expresiones aritméticas.

La normalización de una gramática libre de contexto (CFG, por sus siglas en inglés) es un proceso mediante el cual se reorganizan las producciones de la gramática para hacerlas más estructuradas y fáciles de entender.
El objetivo principal de la normalización es simplificar la gramática y eliminar redundancias, lo que puede facilitar su análisis y uso en sistemas de procesamiento de lenguaje natural y compiladores

Proceso de Normalización de CFG:

Eliminar símbolos inútiles: Identifica y elimina cualquier símbolo no terminal o terminal que no pueda alcanzarse desde el símbolo inicial o que no pueda derivar cadenas en el lenguaje.

Eliminar símbolos no generativos: Elimina los símbolos no terminales que no generan cadenas en el lenguaje.

Eliminar producciones epsilon: Si la gramática tiene producciones que generan la cadena vacía (ε), elimina esas producciones y ajusta las reglas afectadas.

Eliminar producciones unitarias: Elimina las producciones que son unitarias (A -> B, donde A y B son símbolos no terminales) y ajusta las reglas afectadas.

Eliminar recursión izquierda directa: Si la gramática tiene producciones de la forma A -> Aα, donde α es una cadena, elimina esta recursión izquierda directa.

Eliminar ambigüedad: Si es posible, reescribe las producciones para eliminar la ambigüedad en la gramática. Pueden ser necesarios cambios en la estructura para garantizar una única interpretación.

Ejemplo de Normalización de CFG:
Supongamos que tenemos la siguiente gramática no normalizada:
Producciones originales:

S -> aA | bB
A -> Ab | ε
B -> Bc | ε

Pasos para la normalización:

Eliminamos símbolos no generativos y no alcanzables:

- Símbolos no alcanzables: ninguna eliminación necesaria.
- Símbolos no generativos: A, B (eliminamos A y B).

Eliminamos producciones epsilon:

- A -> Ab (eliminamos esta producción).
- B -> Bc (eliminamos esta producción).

Eliminamos producciones unitarias:

- Ninguna producción unitaria presente.

Eliminamos recursión izquierda directa:

- Ninguna recursión izquierda directa presente.

Después de los pasos de normalización, la gramática quedaría así:
Producciones normalizadas:

S -> aA | bB
A -> bA | ε
B -> cB | ε

La gramática se ha normalizado para que sea más estructurada y fácil de entender, y se han eliminado símbolos no generativos, producciones epsilon y producciones unitarias.

Bibliografia

Unidad I

En Fundamentos generales de programación, de Luis Joyanes Aguilar, 33-37. México: Mc Graw-Hill, 2013.

Aquilar, Luis joyanes. En Metodologia de la programación Diagramas de flujo, Algoritmos y programación Estructurada, 66-88. Mexico: Mc Graw-Hill, 1987.

En Fundamentos de Programación , de Manuel Santos. Ismael Patiño. Raul Carrasco, 37-40. México : AlfaOmega-Rama, 2006.

Chorda, Gloria de Antonio. Ramon. «Metodologia de la Programación.» 39-41. España: Ra-ma, s.f.

En Lenguaje Ensamblador para Mircrocomputadoras IBM, de J. Terry Godfrey, 73-76. México: Prentice Hall, 1991.

En Introducción a la computación y a la programación Estructurada, de Guillermo Levine, 115-118. Mexico, 1989.

En Sistema Operativos y Compiladores, de Jesus Salas Parilla, 149,150,173,174. México : Mc Graw-Hill, 1992.

Unidad II

En Compiladores, Principios, Técnicas y Herramientas , de Sethi, R, Ullman, J Aho A, 86-89,94-108,115-131,164-187,190-193,200-201,209-212,221-226,264-274,443-444. U.S.A: Adisson-Wesley Iberoamericana , 1990.

En Compiladores e Interpretes. Teoria y Practica, de M Moreno, M de la cruz, A Ortega y E Pulido, 78-85, 191-194, 199-204, 221-223. España: Pearson- Prentice Hall, 2006.

Unidad III

En Compiladores, Principios, Técnicas y Herramientas , de Sethi, R, Ullman, J Aho A, 86-89,94-108,115-131,164-187,190-193,200-201,209-212,221-226,264-274,443-444. U.S.A: Adisson-Wesley Iberoamericana , 1990.

En Compiladores e Interpretes. Teoria y Practica, de M Moreno, M de la cruz, A Ortega y E Pulido, 78-85, 191-194, 199-204, 221-223. España: Pearson- Prentice Hall, 2006.

Unidad IV

En Compiladores, Principios, Técnicas y Herramientas , de Sethi, R, Ullman, J Aho A, 86-89,94-108,115-131,164-187,190-193,200-201,209-212,221-226,264-274,443-444. U.S.A: Adisson-Wesley Iberoamericana , 1990.

En Compiladores e Interpretes. Teoria y Practica, de M Moreno, M de la cruz, A Ortega y E Pulido, 78-85, 191-194, 199-204, 221-223. España: Pearson- Prentice Hall, 2006.

En Sistema Operativos y Compiladores, de Jesus Salas Parilla, 149,150,173,174. México : Mc Graw-Hill, 1992.