UNIVERSIDAD AUTÓNOMA DE CHIAPAS, TUXTLA GUTIÉRREZ A
2024-08-22 FACULTAD DE SISTEMAS.
INGENIERÍA EN DESARROLLO Y TECNOLOGÍAS DE SOFTWARE.

SEXTO SEMESTRE.

ALUMNO GABRIEL HASSAN BRUNO SANCHEZ
MATRICULA: A210483

MATERIA: COMPILADORES
ACTIVIDAD: Act. 1.3
SUBCOMPETENCIA: SUBCOMPETENCIA 1 ANÁLISIS LÉXICO.
PROFESOR: DR. LUIS GUTIÉRREZ ALFARO

Act. 1.3 Investigar los Conceptos del analizador léxico.

Introducción

En la siguiente investigación se aborda el tema de expresiones regulares incluyendo un ejemplo, así como autómatas y sus derivados; autómatas no determinísticos y autómatas determinísticos. Así también se desarrolla el tema de matrices de transición y tablas de símbolos, se explican diferentes herramientas automáticas para generar analizadores léxicos y se desarrolla el tema de gramática de libre contexto.

Desarrollo:

Expresiones regulares

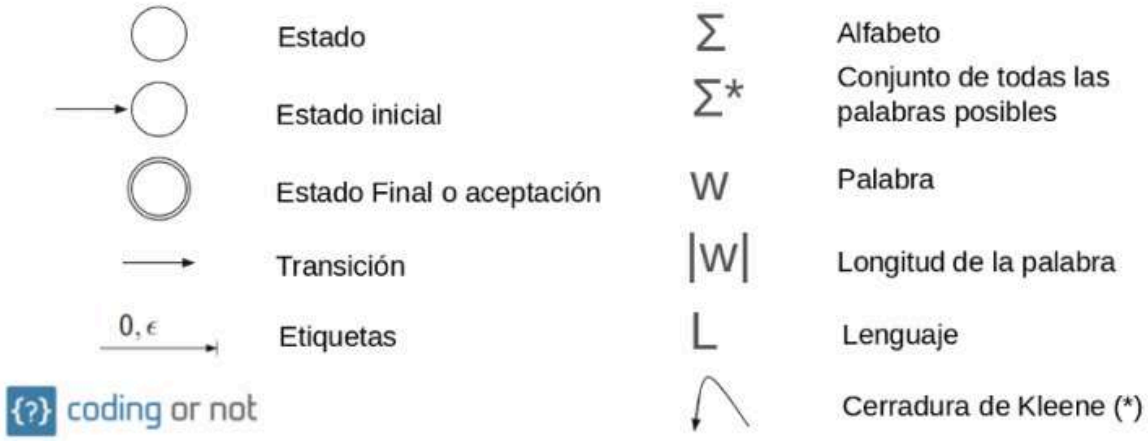
Una expresión regular es un modelo con el que el motor de expresiones regulares intenta buscar una coincidencia en el texto de entrada. Un modelo consta de uno o más literales de carácter, operadores o estructuras.

Ejemplo:

[a-z]	Una letra minúscula

Autómatas

Un autómata es un modelo matemático para una máquina de estado finito, en el que dada una entrada de símbolos, «salta» mediante una serie de estados de acuerdo a una función de transición (que puede ser expresada como una tabla). Esta función de transición indica a qué estado cambiar dados el estado actual y el símbolo leído.



Ejemplo

Dado el alfabeto $\Sigma = \{ a, b \}$, construir un autómata a pila que reconozca el lenguaje:

$$L = \{w \,|w \in \{a,b\}^*, n_a(w) \geq n_b(w)\}$$

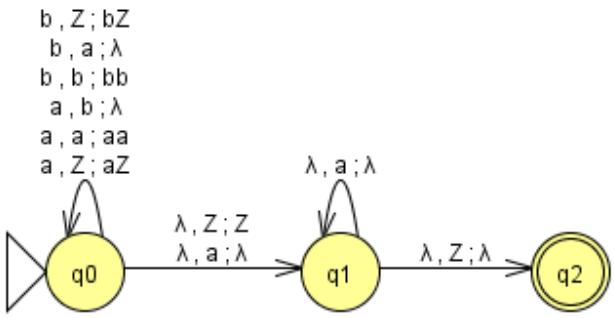
siendo

$$n_a(w)$$

el número de a's en w y $n_b(w)$ el número de b's.

Ver solución

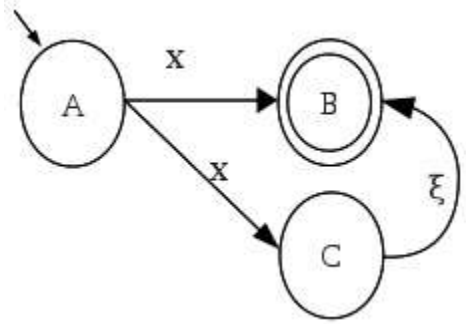
El autómata es el siguiente:



Autómatas no determinista

Es el autómata finito que tiene transiciones vacías o que por cada símbolo desde un estado de origen se llega a más de un estado destino, es decir, es aquel que, a diferencia de los autómatas finitos deterministas, posee al menos un estado , tal que para un símbolo del alfabeto, existe más de una transición posible.

Ejemplo



La definición formal de AFND se basa en la consideración de que a menudo según los algoritmos de transformación de expresiones y gramáticas regulares a AF terminan obteniéndose autómatas con transiciones múltiples para un mismo símbolo o transiciones vacías. Independientemente que sean indeseables, sobre todo para la implementación material, fundamentalmente mecánica, de los autómatas finitos, son imprescindibles durante la modelación de analizadores lexicográficos de los elementos gramaticales de los lenguajes de programación, llamados tókens, como literales numéricos, identificadores, cadenas de texto, operadores, etc.

Autómatas determinísticos

Es aquel que sólo puede estar en un único estado después de leer cualquier secuencia de entradas. El término “determinista” hace referencia al hecho de que para cada entrada sólo existe uno y sólo un estado al que el autómata puede hacer la transición a partir de su estado actual.

El término “autómata finito” hace referencia a la variedad determinista, aunque normalmente utilizaremos el término “determinista” o la abreviatura AFD, con el fin de recordar al lector el tipo de autómata del que estamos hablando
Formalmente, un autómata finito determinista es una quíntupla (Q, Σ, δ, q 0, F), donde:

Q: conjunto finito NO VACIO de estados

Σ: alfabeto de entrada

δ: Q x Σ Q, es la función total de transición

q 0 ∈ Q: estado inicial de autómata

F ⊆ Q: conjunto de estados finales

Ejemplo

Considere un sistema formado por una lámpara y un interruptor. La lámpara puede estar encendida o apagada. El sistema sólo puede recibir un estímulo exterior: pulsar el interruptor. El funcionamiento es habitual: si se pulsa el interruptor y estaba apagada la lámpara, se pasa al estado de encendido, o si esta encendida, pasa a pagada. Se desea que la bombilla este inicialmente apagada.
Considere que 0: encendido, 1: apagado y la única entrada posible (pulsar interruptor) es “p”

$A = (Q, \Sigma, \delta, q_0, F)$, donde:

- $Q = \{0,1\}$
- $\Sigma = \{p\}$
- $\delta(0,p) = 1, \delta(1,p) = 0$
- $q_0 = 0$
- $F = \{\}$

Matrices de transición

Una tabla de transiciones es un arreglo (o matriz) bidimensional cuyos elementos proporcionan el resumen de un diagrama de transiciones correspondiente.

Una tabla de transiciones es una representación tabular convencional de una función, como por ejemplo δ, que toma dos argumentos y devuelve un valor. Las filas de la tabla corresponden a los estados y las columnas a las entradas. La entrada para la fila correspondiente al estado q y la columna correspondiente a la entrada a es el estado δ(q,a).

Ejemplo:

	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	q_2

Otro ejemplo:

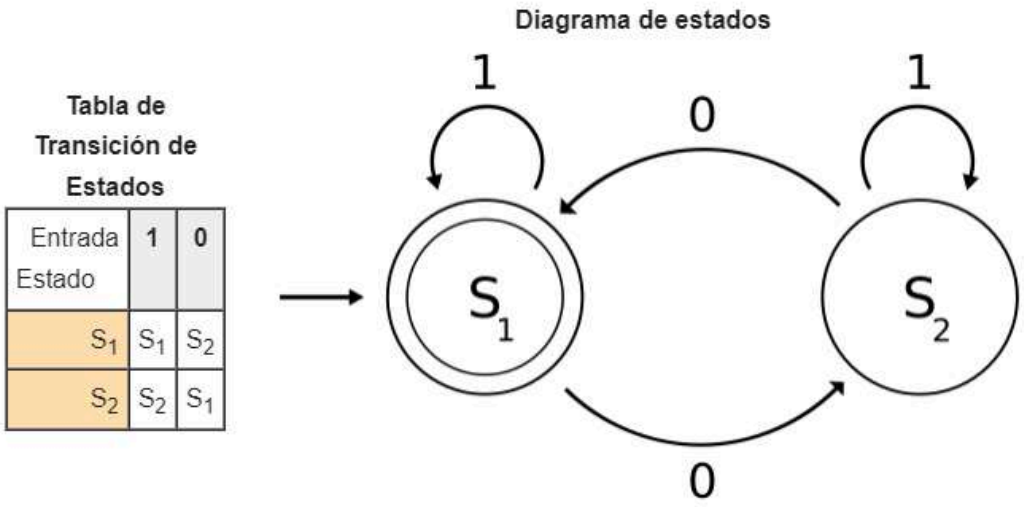


Tabla de símbolos:

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.

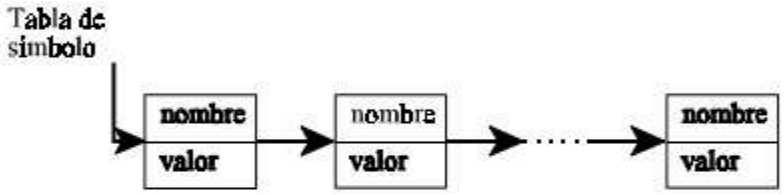
Ejemplo

Vamos a hacer un intérprete. Recordar que en un intérprete la entrada es un programa y la salida es la ejecución de ese programa.

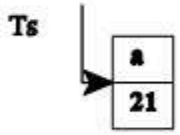
Suponemos que queremos hacer las siguientes operaciones:

a = 7 * 3
b = 3 * a

En la segunda instrucción necesitamos saber cuanto vale 'a'; es decir el valor de 'a' debe estar guardado en algún sitio. Para ello utilizaremos una lista de pares:

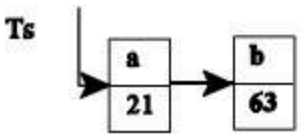


de forma que cuando nos encontremos con la instrucción a = 7 * 3, miremos en la tabla, si no está 'a' en la tabla, creamos un nodo para introducirla.

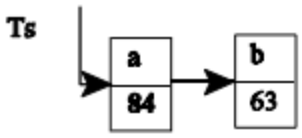


A continuación nos encontramos con b = 3 * a. ¿Qué es 'a'? Busco en la tabla de símbolos y vemos que el valor de 'a' es 21.

b = 3 * a Ahora buscamos b en la tabla de símbolos y como no está lo creamos.



Si ejecutáramos ahora la instrucción: a = a + b
Tendríamos a 'a' y 'b' en la tabla de símbolos con lo cual solo tendríamos que modificar el valor de 'a'.



Como hemos dicho una tabla de símbolos es una estructura que almacena información relativa a los identificadores de usuario. Pero, ¿que información exactamente?, según el propósito que tengamos. En nuestro caso el proposito es hacer un intérprete y nos interesa mantener información, entre otras cosas, del valor de cada variable.

Entradas de la tabla de símbolos

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.

ejemplo

las declaraciones en C.

```
int    x;                    (1)
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.

herramientas automáticas para generar analizadores léxicos

Herramientas tradicionales y ejemplos:

Lex/yacc: es un programa para generar analizadores léxicos, se utiliza comúnmente con el programa yacc que se utiliza para generar análisis sintáctico. Lex, escrito originalmente por Eric Schmidt y Mike Lesk, es el analizador léxico estándar en los sistemas Unix, y se incluye en el estándar de POSIX. Lex toma como entrada una especificación de analizador léxico y devuelve como salida el código fuente implementando el analizador léxico en C.

Yacc: es una herramienta general para describir la entrada a un programa de computadora. El usuario especifica las estructuras de su entrada, junto con el código que puede invocarse como cada estructura como se reconoce y yacc devuelve una subrutina que controla el proceso de entrada, con frecuencia, es conveniente y apropiado que la mayor parte del flujo de control en la aplicación del usuario al cargo de este subprograma. Fue desarrollado por Stephen C. Johnson en para Unix. El nombre es un acrónimo de " Sin embargo, otro compilador de compiladores . " Se genera un programa de análisis (por parte de un compilador que intenta darle sentido sintáctico del código fuente), y genera código para C.

Flex: herramienta que implementa un analizador léxico. 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. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas. Flex genera como salida un fichero fuente en C

Bison: herramienta que implementa un analizador sintáctico. Es un generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos. Bison convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR, en un programa en C, C++, o Java que realiza análisis sintáctico. GNU bison tiene compatibilidad con Yacc.

Herramientas de nueva generación y ejemplos:

ANTLR (ANother Tool for Language Recognition – otra herramienta para reconocimiento de lenguajes): es una herramienta creada principalmente por Terence Parr, que opera sobre lenguajes, proporcionando un marco para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales de los mismos (conteniendo acciones semánticas a realizarse en varios lenguajes de programación). ANTLR genera un programa que determina si una sentencia o palabra pertenece a dicho lenguaje (reconocedor), utilizando algoritmos LL(*) de parsing. Si a dicha gramática, se le añaden acciones escritas en un lenguaje de programación, el reconocedor se transforma en un traductor o interprete. Además, proporciona facilidades para la creación de árboles sintácticos y estructura para recorrerlos. ANTLR es un proyecto bajo licencia BSD, viniendo con todo el

código fuente disponible, y preparado para su instalación bajo plataformas Linux, Windows y Mac OS X.
Actualmente ANTLR genera código Java, C, C++, C#, Python, Perl, Delphi, Ada95, JavaScript y Objective-C.

JavaCC (Java Compiler Compiler): es un generador de analizadores sintácticos de código abierto para el lenguaje de programación Java, es similar a Yacc en que genera un parser para una gramática presentada en notación BNF, con la diferencia de que la salida es en código Java y genera analizadores descendentes. Además agrega un constructor de árboles conocido como: JJTree, construye árboles de abajo hacia arriba.

Gramática libre de Contexto

Estas gramáticas, conocidas también como gramáticas de tipo 2 o gramáticas independientes del contexto, son las que generan los lenguajes libres o independientes del contexto. Los lenguajes libres del contexto son aquellos que pueden ser reconocidos por un autómata de pila determinístico o no determinístico.
Como toda gramática se definen mediante una cuadrupla G = (N, T, P, S), siendo
- N es un conjunto finito de símbolos no terminales
- T es un conjunto finito de símbolos terminales $N \cap T = \varnothing$
- P es un conjunto finito de producciones
- S es el símbolo distinguido o axioma $S \notin (N \cup T)$

En una gramática libre del contexto, cada producción de P tiene la forma

$$A \rightarrow \omega \qquad \begin{cases} A \in N \cup \{S\} \\ \omega \in (N \cup T)^* - \{\varepsilon\} \end{cases}$$

Es decir, que en el lado izquierdo de una producción pueden aparecer el símbolo distinguido o un símbolo no terminal y en el lado derecho de una producción cualquier cadena de símbolos terminales y/o no terminales de longitud mayor o igual que 1.
La gramática puede contener también la producción $S \rightarrow \varepsilon$ si el lenguaje que se quiere generar contiene la cadena vacía.

Ejemplo:

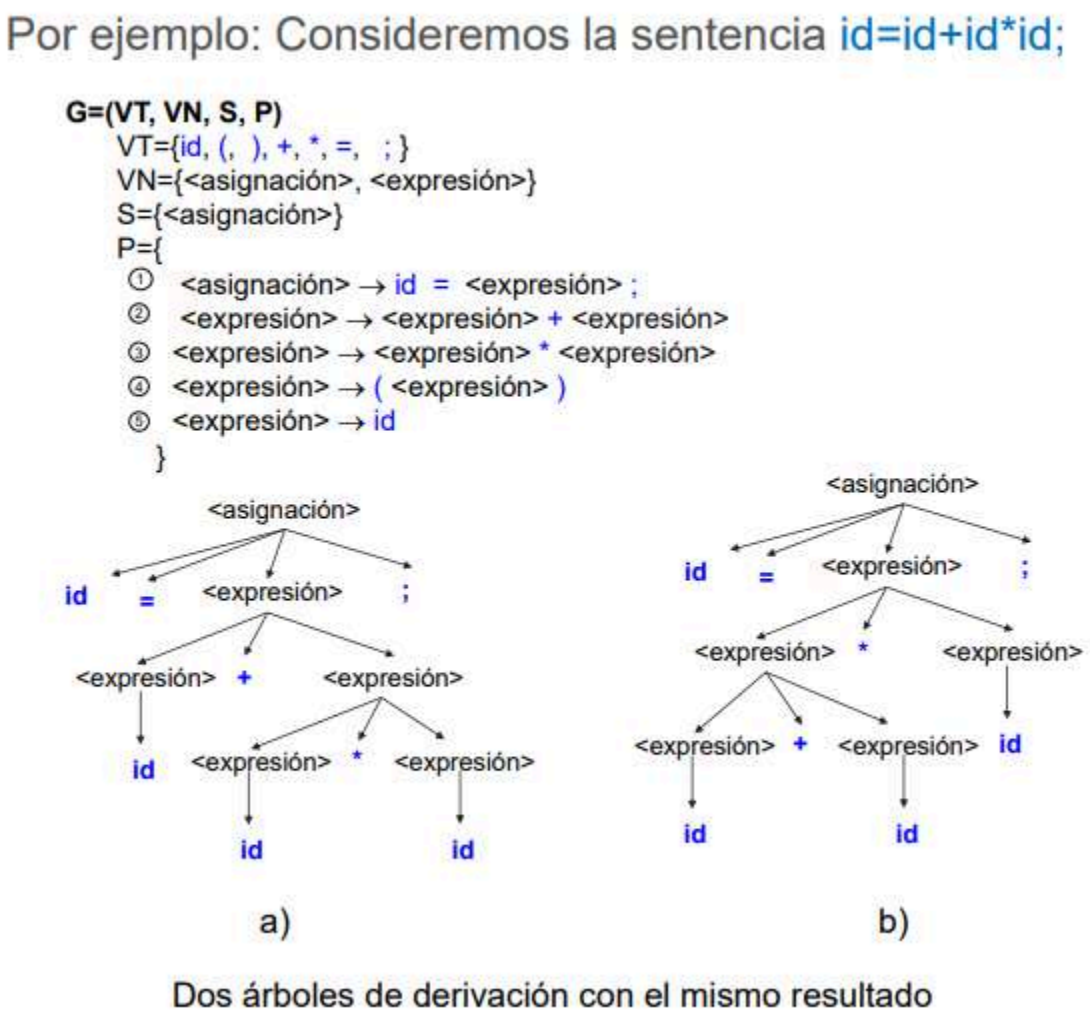
La siguiente gramática genera las cadenas del lenguaje $L_1 = \{wcw^R / w \in \{a, b\}^*\}$
$G_1 = (\{A\}, \{a, b, c\}, P_1, S_1)$, y P_1 contiene las siguientes producciones

$S_1 \rightarrow A$
$A \rightarrow aAa$
$A \rightarrow bAb$
$A \rightarrow c$

Gramática ambigua

Las gramáticas libres de contexto representan el lenguaje formal a partir del cual se construyen los analizadores sintácticos. Muchas de las veces, estas gramáticas deben ser re escritas hasta cumplir con ciertas características, siendo la ambigüedad uno de los problemas más comunes.
Cuando una gramática libre del contexto genera mas de una estructura para alguna o algunas cadenas del lenguajes, se dice que se trata de un gramática ambigua.

Ejemplo:



Dos árboles de derivación con el mismo resultado

En el ejemplo anterior se puede observar que en la derivación a), la primera es sustituida por +, mientras que en la derivación b) la primera es sustituida por * . La diferencia entre ellas dos afecta a la estructura de las expresiones, ya que en a) primero se multiplica y después se suma, mientras que en b) primero se suma y luego se multiplica. Por jerarquía de operaciones sabemos que a) es correcta y b) incorrecta, con lo cual vemos que la gramática no es adecuada debido a que no proporciona una estructura única.

Forma enunciativa y ejemplos:

Sea G = (V, T, P, S) una CFG y α ∈ (V ∪ T) ∗ . Si S ∗⇒ α decimos que α está en forma de sentencia (sentential form). Si S ⇒lm α decimos que α es una forma de sentencia izquierda (left-sentencial form), y si S ⇒rm α decimos que α es una forma de sentencia derecha (right-sentencial form). L(G) son las formas de sentencia que estan en T ∗ .

Si tomamos la gramática del lenguaje sencillo, E ∗ (I + E) es una forma de sentencia ya que: E ⇒ E ∗ E ⇒ E ∗ (E) ⇒ E ∗ (E + E) ⇒ E ∗ (I + E). Esta derivación no es ni más a la izquierda ni más a la derecha. Por otro lado: a ∗ E es una forma de sentencia izquierda, ya que: E ⇒lm E ∗ E ⇒lm I ∗ E ⇒lm a ∗ E y E ∗ (E + E) es una forma de sentencia derecha, ya que: E ⇒rm E ∗ E ⇒rm E ∗ (E) ⇒rm E ∗ (E + E).

proceso de remoción de ambigüedad de gramáticas

Una GLC es ambigua si existe una cadena w Î L(G) que tiene más de una derivación por la izquierda o más de una derivación por la derecha o si tiene dos o más árboles de derivación. En caso de que toda cadena w Î L(G) tenga un único árbol de derivación, la gramática no es ambigua.

Tipos de ambigüedad:

Ambigüedad Inherente:
Las gramáticas que presentan este tipo de ambigüedad no pueden utilizarse para lenguajes de programación, ya que por más transformaciones que se realicen sobre ellas, nunca se podrá eliminar completamente la ambigüedad que presentan.
Un lenguaje L es inherentemente ambiguo si todas sus gramáticas son ambiguas; si existe cuando menos una gramática no ambigua para L, L no es
ambiguo.
– El lenguaje de las expresiones no es Ambiguo
– Las expresiones regulares no son ambiguas

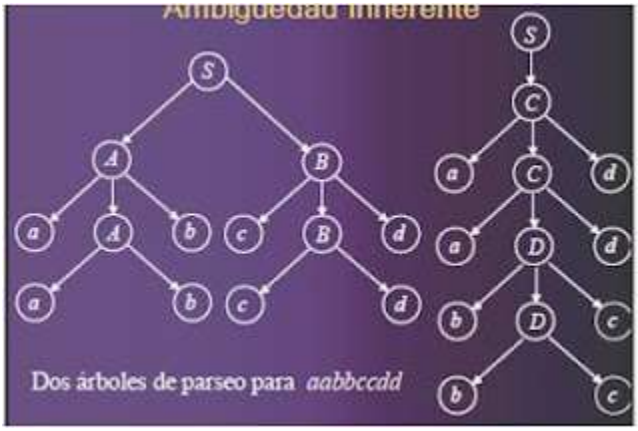
Ejemplo

Ejemplo de un lenguaje inherentemente ambiguo:
L = {anbncmdm | n ³ 1, m ³ 1} È {anbmcmdn | n ³ 1, m ³ 1}

L es un LLC:
S ® AB | C
A ® aAb | ab C ® aCd | aDd
B ® cBd | cd D ® bDc | bc

La gramática es ambigua: hay cadenas con más de una
derivación más izquierda:
Considere: aabbccdd (m = n = 2)
S ⇒ AB ⇒ aAbB ⇒ aabbB ⇒ aabbcBd ⇒ aabbccdd

S ⇒ C ⇒ aCd ⇒ aaDdd ⇒ aabDcdd ⇒ aabbccdd



El lenguaje:
- L = {anbncmdm | n ³ 1, m ³ 1} È {anbmcmdn | n ³ 1, m ³ 1}

- La gramática
- S ® AB | C
A ® aAb | ab C ® aCd | aDd
B ® cCd | cd D ® bDc | bc
_ ¿Por qué todas las gramáticas para este lenguaje son ambiguas?
– Considere cualquier cadena con m = n
– ¡Siempre habrá dos derivaciones para estas cadenas!
Ambigüedad Transitoria:
Este tipo de ambigüedad puede llegar a ser eliminada realizando una serie de transformaciones sobre la gramática original. Una vez que se logra lo anterior, la gramática queda lista para ser reconocida por la mayor parte de los analizadores sintácticos. (Se le considera "ambigüedad" porque existen métodos para realizar análisis sintáctico que no aceptan gramáticas con estas características)
Dónde se presenta la Ambigüedad Transitoria generalmente la ambigüedad se presenta cuando existen producciones con factores comunes (distintas alternativas para un símbolo no-terminal que inician de la misma forma); ó cuando existen producciones que son recursivas izquierdas (producciones para un símbolo no-terminal en las cuales el primer símbolo de su forma sentencial es ese mismo símbolo no-terminal).

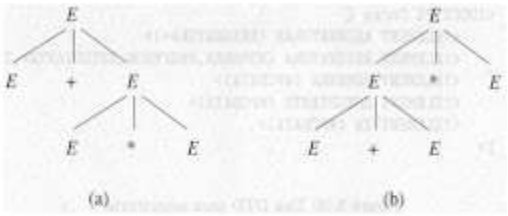
Para eliminar este tipo de ambigüedad, es necesario, primero eliminar:
- Factores comunes izquierdos inmediatos y No-inmediatos.
- Recursividad izquierda inmediata y No-inmediata.

Eliminación de la ambigüedad

– No existe un algoritmo que nos indique si una GIC es ambigua
– Existen LIC que sólo tienen GIC ambiguas: inherentemente ambiguos

– Para las construcciones de los lenguajes de programación comunes existen técnicas para la eliminación de la ambigüedad
– Ejemplo: causas de ambigüedad en la siguiente gramática
• No se respeta la precedencia de operadores
• una secuencia de operadores idénticos puede agruparse desde la izquierda y desde la derecha. Lo convencional es agrupar desde la izquierda.

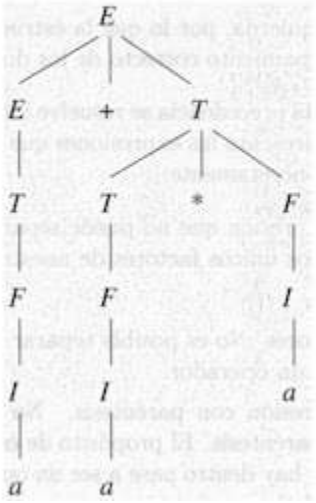
1. $E \to I$
2. $E \to E+E$
3. $E \to E*E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$



Ejemplo: modificamos la gramática para forzar la precedencia

1. $E \to I$
2. $E \to E+E$
3. $E \to E*E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$F \to I \mid (E)$
$T \to F \mid T*F$
$E \to T \mid E+T$



normalización de CFG y Proceso con ejemplos

La forma normal de Chomsky (CNF) es una forma específica de gramáticas libres de contexto (CFG) que impone ciertas restricciones en las reglas de producción. Estas restricciones facilitan el análisis y la manipulación de la gramática, lo que puede ser beneficioso en varias tareas

computacionales, incluidas las relacionadas con la ciberseguridad y la teoría de la complejidad computacional.

En la forma normal de Chomsky, cada regla de producción tiene una de dos formas: un símbolo no terminal que produce exactamente dos símbolos no terminales, o un símbolo no terminal que produce exactamente un símbolo terminal. Más formalmente, una regla de producción en CNF se puede escribir de la siguiente manera:

A -> BC

or

A -> a

donde A, B y C son símbolos no terminales y a es un símbolo terminal. La regla A -> ε, donde ε representa la cadena vacía, también está permitida en CNF.

Las restricciones impuestas por CNF tienen varias ventajas. En primer lugar, simplifican el proceso de análisis. El análisis es la tarea de determinar la estructura de una cadena determinada en función de una gramática. En CNF, el análisis se puede realizar de manera más eficiente utilizando algoritmos como el algoritmo CYK o el algoritmo de Earley.

En segundo lugar, CNF facilita el razonamiento sobre las propiedades de una gramática. Por ejemplo, resulta sencillo determinar si un lenguaje generado por una gramática determinada es regular o independiente del contexto. Esto puede ser útil para analizar la complejidad de los lenguajes y diseñar sistemas seguros que se basen en la teoría del lenguaje formal.

Además, CNF facilita el estudio de la teoría de la complejidad computacional. Al restringir la forma de las reglas de producción, CNF permite un análisis más preciso de la complejidad de los algoritmos de análisis. Este análisis puede ayudar a determinar la complejidad temporal y espacial de los algoritmos utilizados en las aplicaciones de ciberseguridad, como los sistemas de detección de intrusos o las herramientas de análisis de malware.

Ejemplo

Para ilustrar las restricciones de CNF, considere el siguiente ejemplo de CFG:

S -> ASA | aB
A -> B | S
segundo -> segundo | ε

Para convertir este CFG en CNF, necesitamos reescribir las reglas de producción para cumplir con las restricciones. Primero, eliminamos la regla A -> S introduciendo un nuevo símbolo no terminal. El CFG modificado se convierte en:

S -> ASA | aB
A -> B | T
segundo -> segundo | ε
T -> S

A continuación, eliminamos la regla S -> ASA introduciendo otro nuevo símbolo no terminal. El CFG modificado se convierte en:

S -> EN | aB
A -> B | T
segundo -> segundo | ε
T -> S

Finalmente, eliminamos la regla S -> AT introduciendo otro nuevo símbolo no terminal. El CFG final en CNF es:

S -> ES | ab
A -> B | T
segundo -> segundo | ε
T -> COMO
U -> T

La forma normal de Chomsky impone restricciones específicas en las gramáticas independientes del contexto, lo que requiere que cada regla de producción produzca dos símbolos no terminales o un símbolo terminal. Estas restricciones simplifican el análisis, ayudan a razonar sobre las propiedades gramaticales y facilitan el análisis de la complejidad computacional en las aplicaciones de ciberseguridad.

Conclusiones:

En conclusión, el análisis de expresiones regulares, autómatas y gramáticas libres de contexto es fundamental para la creación de analizadores léxicos y sintácticos en la teoría de lenguajes formales. A través de ejemplos y explicaciones detalladas, hemos explorado las diferentes herramientas y técnicas utilizadas para generar analizadores léxicos, incluyendo herramientas tradicionales y de nueva generación. Además, hemos abordado la importancia de la normalización de gramáticas libres de contexto y la eliminación de ambigüedades para garantizar la precisión y eficiencia en el análisis de lenguajes formales. Este conocimiento es esencial para cualquier persona interesada en el desarrollo de compiladores, intérpretes y otros sistemas de procesamiento de lenguajes.

Referencias:

(Se usó la última versión de APA)

Adegeo. (2024, 4 julio). *Lenguaje de expresiones regulares - Referencia rápida - .NET*. Microsoft Learn. https://learn.microsoft.com/es-es/dotnet/standard/base-types/regular-expression-language-quick-reference

GRAMATICAS LIBRES DEL CONTEXTO. Edu.ar. Recuperado el 23 de agosto de 2024, de https://users.exa.unicen.edu.ar/catedras/ccomp1/Apunte5.pdf

Ambiguas, G. (s/f). *Universidad Autónoma del Estado de Hidalgo Escuela Superior de Huejutla Licenciatura en Ciencias Computacionales*. Edu.mx. Recuperado el 23 de agosto de 2024, de https://web2.uaeh.edu.mx/docencia/P_Presentaciones/huejutla/sistemas/2017/Gramaticas_Ambiguas.pdf

De contexto, 1. Gramaticas Libres. (s/f). *Proped ´eutico: Teor ´ıa de Aut ´omatas y Lenguajes Formales Gram ´aticas libres de contexto y lenguajes*. Inaoep.mx. Recuperado el 23 de agosto de 2024, de https://posgrados.inaoep.mx/archivos/PosCsComputacionales/Curso_Propedeutico/Automatas/05_Automatas_GramaticasLibresContextoLenguajes/CAPTUL1.PDF

4.3 La tabla de símbolos. (s. f.). http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/autocontenido/autocon/43__la_tabla_de_smbolos.html

García, M. (2018, 13 julio). *¿Qué es un autómata?* https://codingornot.com/que-es-un-automata

Silvino, A. C. J. J. E. H. E. I. G. R. G. G. N. G. y. R. G. (s. f.). *UNIDAD IV.- Herramientas básicas para generar compiladores*. https://cursocompiladoresuaeh.blogspot.com/2010/11/unidad-iv-herramientas-basicas-para.html

Academy, E. (2023, 2 agosto). *¿Qué es la forma normal de Chomsky y cuáles son las restricciones específicas que impone a las gramáticas independientes del contexto? - Academia EITCA*. EITCA Academy. https://es.eitca.org/cybersecurity/eitc-is-cctf-computational-complexity-theory-fundamentals/context-sensitive-languages/chomsky-normal-form/examination-review-chomsky-normal-form/what-is-chomsky-normal-form-and-what-are-the-specific-constraints-it-imposes-on-context-free-grammars/