Nombre: Rodriguez perez Luis diego

Grupo: 7 AM

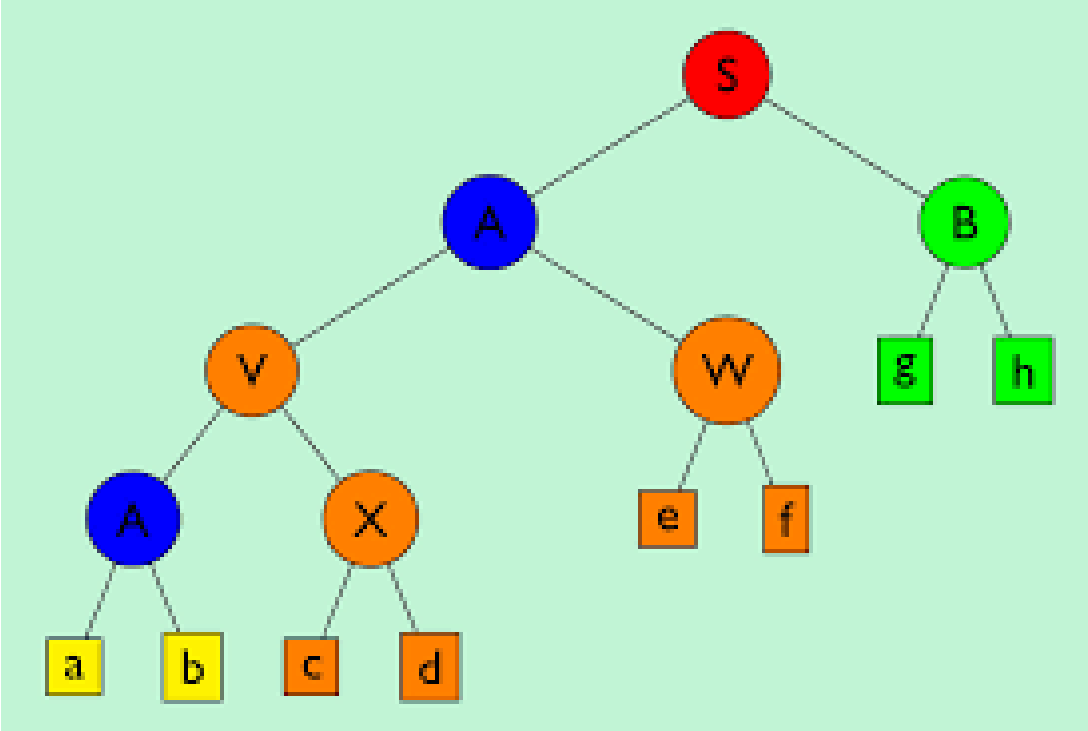
Nombre de la escuela: Instituto Tecnológico de Iztapalapa

Carrera: ING en sistemas computacionales

Materia: lenguajes y autómatas 2

Nombre del profesor: Parra Hernández abiel tomas

no control: 161080170



Actividades semana 11 (30 Nov-4 Dic, 2020)

Leer y estudiar el tema 3.2.2 "Gramáticas independientes del contexto (Context-Free Grammars)" del libro Engineering a Compiler (2nd Ed 2012) de Cooper, Torczon. Después analizar el primer ejemplo complejo del tema 3.2.3

3.2.2 Gramáticas sin contexto

Para describir la sintaxis del lenguaje de programación, necesitamos una notación más poderosa que las expresiones regulares que aún conduce a reconocimientos eficientes. La solución tradicional es utilizar una gramática sin contexto (cfg). Afortunadamente,
grandes subclases de los cfgs tienen la propiedad de que conducen a una eficiente reconocimientos.

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 llama
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

La primera regla o producción dice "SheepNoise puede derivar la palabra baa
seguido de más SheepNoise ". Aquí 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 comprender la relación entre la gramática SN y L (SN), necesitamos
para especificar cómo aplicar reglas en SN para derivar sentencias en L (SN). Empezar,
debemos identificar el símbolo de meta o símbolo de inicio de SN. El símbolo de la meta
representa el conjunto de todas las cadenas de L (SN). Como tal, no puede ser uno de los
palabras en el idioma. En cambio, debe ser uno de los símbolos no terminales
introducido para agregar estructura y abstracción al lenguaje. Dado que SN tiene
solo un no terminal, SheepNoise debe ser el símbolo del objetivo.

Para derivar una oración, comenzamos con una cadena prototipo que contiene solo la Derivación una secuencia de pasos de reescritura que comienza con
el símbolo de inicio de la gramática y termina con un oración en el idioma
símbolo de objetivo, SheepNoise. Elegimos un símbolo no terminal, α, en el prototipo
cadena, elija una regla gramatical, $\alpha \rightarrow \beta$, y reescribe α con β. Repetimos esto
proceso de reescritura hasta que la cadena prototipo no contenga más no terminales,
en cuyo punto se compone enteramente de palabras, o símbolos terminales, y es un
oración en el idioma.

En cada punto de este proceso de derivación, la cadena es una colección de terminales
o símbolos no terminales. Tal cadena se llama forma enunciativa si ocurre
en algún paso de una derivación válida. Cualquier forma oracional puede derivarse de
el símbolo de inicio en cero o más pasos. Del mismo modo, desde cualquier forma oracional
podemos derivar una oración válida en cero o más pasos. Por lo tanto, si comenzamos con
SheepNoise y aplique sucesivas reescrituras utilizando las dos reglas, en cada paso de
el proceso, la cadena es una forma enunciativa. Cuando llegamos al punto
donde la cadena contiene solo símbolos terminales, la cadena es una oración
en L (SN)

Para derivar una oración en SN, comenzamos con la cadena que consta de un símbolo, SheepNoise. Podemos reescribir SheepNoise con la regla 1 o la regla 2. Si reescribimos SheepNoise con la regla 2, la cadena se convierte en baa y no tiene más oportunidades para reescribir. La reescritura muestra que baa es una oración válida en L (SN). La otra opción, reescribir la cadena inicial con la regla 1, conduce a una cadena con dos símbolos: baa SheepNoise. A esta cadena le queda una no terminal; reescribir con la regla 2 conduce a la cadena baa baa, que es una oración en L (SN). Podemos representar estas derivaciones en forma de tabla:

Rule	Sentential Form
	SheepNoise
2	baa

Rewrite with Rule 2

Rule	Sentential Form
	SheepNoise
1	baa *SheepNoise*
2	baa baa

Rewrite with Rules 1 Then 2

Cómo conveniencia de notación, usaremos → + para significar "deriva en uno o más pasos ". Por lo tanto, SheepNoise → + baa y SheepNoise → + baa baa.
La regla 1 alarga la cuerda mientras que la regla 2 elimina el SheepNoise no terminal. (La cadena nunca puede contener más de una instancia de SheepNoise).
Todas las cadenas válidas en SN se derivan de cero o más aplicaciones de la regla 1, seguido de la regla 2. La aplicación de la regla 1 k veces seguida de la regla 2 genera una cadena con k + 1 baas.

BACKUS-NAUR FORM

The traditional notation used by computer scientists to represent a context-free grammar is called *Backus-Naur form*, or BNF. BNF denoted nonterminal symbols by wrapping them in angle brackets, like ⟨SheepNoise⟩. Terminal symbols were underlined. The symbol ::= means "derives," and the symbol | means "also derives." In BNF, the sheep noise grammar becomes:

⟨SheepNoise⟩ ::= baa ⟨SheepNoise⟩
| baa

This is completely equivalent to our grammar *SN*.

BNF has its origins in the late 1950s and early 1960s [273]. The syntactic conventions of angle brackets, underlining, ::=, and | arose from the limited typographic options available to people writing language descriptions. (For example, see David Gries' book *Compiler Construction for Digital Computers*, which was printed entirely on a standard lineprinter [171].) Throughout this book, we use a typographically updated form of BNF. Nonterminals are written in *italics*. Terminals are written in the `typewriter font`. We use the symbol → for "derives."

FORMULARIO BACKUS-NAUR
La notación tradicional utilizada por los científicos informáticos para representar un
La gramática libre de contexto se llama forma Backus-Naur, o BNF. BNF denota símbolos no terminales en volviéndolos entre corchetes angulares, como hSheepNoisei.
Los símbolos terminales estaban subrayados. El símbolo :: = significa "deriva" y el símbolo | significa "también deriva". En BNF, la gramática del ruido de las ovejas se convierte en:

```
(SheepNoise)  ::=  baa (SheepNoise)
               |   baa
```

Esto es completamente equivalente a nuestra gramática SN.
El BNF tiene sus orígenes a finales de la década de 1950 y principios de la de 1960 [273]. Las convenciones sintácticas de los corchetes angulares, subrayado, :: = y | surgió de la opciones tipográficas limitadas disponibles para las personas que escriben descripciones de idiomas. (Por ejemplo, consulte el libro Compiler Construction for Digital Computadoras, que se imprimió completamente en una impresora de línea estándar [171].)
A lo largo de este libro, utilizamos una forma de BNF actualizada tipográficamente. Los no terminales están escritos en cursiva. Los terminales están escritos en tipo de máquina de escribir. Usamos el símbolo → para "deriva".

CONTEXT-FREE GRAMMARS

Formally, a context-free grammar *G* is a quadruple (*T*, *NT*, *S*, *P*) where:

- *T* is the set of terminal symbols, or words, in the language *L*(*G*). Terminal symbols correspond to syntactic categories returned by the scanner.
- *NT* is the set of nonterminal symbols that appear in the productions of *G*. Nonterminals are syntactic variables introduced to provide abstraction and structure in the productions.
- *S* is a nonterminal designated as the *goal symbol* or *start symbol* of the grammar. *S* represents the set of sentences in *L*(*G*).
- *P* is the set of productions or rewrite rules in *G*. Each rule in *P* has the form $NT \rightarrow (T \cup NT)^+$; that is, it replaces a single nonterminal with a string of one or more grammar symbols.

The sets *T* and *NT* can be derived directly from the set of productions, *P*. The start symbol may be unambiguous, as in the *SheepNoise* grammar, or it may not be obvious, as in the following grammar:

```
Paren   →  ( Bracket )        Bracket  →  [ Paren ]
        |  (       )                   |  [     ]
```

In this case, the choice of start symbol determines the shape of the outer brackets. Using *Paren* as *S* ensures that every sentence has an outermost pair of parentheses, while using *Bracket* forces an outermost pair of square brackets. To allow either, we would need to introduce a new symbol *Start* and the productions *Start* → *Paren* | *Bracket*.

Some tools that manipulate grammars require that *S* not appear on the right-hand side of any production, which makes *S* easy to discover.

GRAMÁTICAS SIN CONTEXTO
Formalmente, una gramática libre de contexto G es cuádruple (T, NT, S, P) donde:
T es el conjunto de símbolos terminales, o palabras, en el lenguaje L (G). Los símbolos terminales corresponden a categorías sintácticas devueltas por escáner.
NT es el conjunto de símbolos no terminales que aparecen en las producciones de G. Los no terminales son variables sintácticas introducidas para proporcionar abstracción y estructura en las producciones.
S es un no terminal designado como símbolo de meta o símbolo de inicio de la gramática. S representa el conjunto de oraciones en L (G).
P es el conjunto de producciones o reglas de reescritura en G. Cada regla en P tiene la

forma NT → (T ∪ NT) +; es decir, reemplaza un único no terminal con una cadena de uno o más símbolos gramaticales.
Los conjuntos T y NT pueden derivarse directamente del conjunto de producciones, P.
El símbolo de inicio puede ser inequívoco, como en la gramática SheepNoise, o puede que no sea obvio, como en la siguiente gramática:

```
Paren   → ( Bracket )      Bracket → [ Paren ]
        | ( )                      | [ ]
```

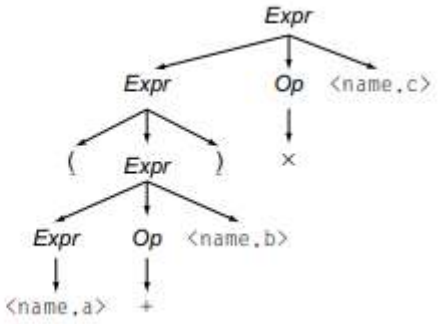
En este caso, la elección del símbolo de inicio determina la forma del exterior soportes. Usar Paren como S asegura que cada oración tenga un exterior par de paréntesis, mientras que el uso de corchetes fuerza un par más externo de cuadrados soportes. Para permitir cualquiera de las dos, tendríamos que introducir un nuevo símbolo Inicio y las producciones Inicio → Paren | Soporte.
Algunas herramientas que manipulan gramáticas requieren que S no aparezca en el lado derecho de cualquier producción, lo que hace que S sea fácil de descubrir.

analizar el primer ejemplo complejo del tema 3.2.3

```
1  Expr → ( Expr )
2       | Expr Op name
3       | name
4  Op   → +
5       | -
6       | ×
7       | ÷
```

estudiar el segundo ejemplo complejo de la sección 3.2.3 que revisaremos en la sesión síncrona de la semana.

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name



aquí en el ejemplo podemos como podemos ver como se expresa como nos muestra el ejemplo se va expresando en código para poder hacer nuestro código que lo podemos hacer en un diagrama de árbol de como va saliendo ahora si que las expresiones regulares para ver cómo se va desarrollando

Actividades semana 13 (Dic 14-18, 2020)

1) Actividades teóricas:

Repasar el tema 4.2 "An Introduction to Type Systems" del libro Engineering a Compiler (2nd Ed 2012) de Cooper, Torczon.

4.2 INTRODUCCIÓN A LOS TIPOS DE SISTEMAS

La mayoría de los lenguajes de programación asocian una colección de propiedades con cada valor de datos. Llamamos a esta colección de propiedades del tipo de valor. El tipo especifica un conjunto de propiedades que tienen en común todos los valores de ese tipo. Los tipos se pueden especificar por membresía; por ejemplo, para integrar ser podría ser cualquier número entero i en el rango -2 31 ≤ yo <2 31, o rojo podría ser un valor en un tipo enumerado de colores, definido como el conjunto {rojo, naranja, amarillo, verde, azul, marrón, negro, blanco}. Los tipos pueden ser especificados por reglas; por ejemplo, la declaración de una estructura en c define un tipo.

En este caso, el tipo incluye cualquier objeto con los campos declarados en la orden declarada; los campos individuales tienen tipos que especifican los rangos de valores permitidos y su interpretación. (Representamos el tipo de estructura como el producto de los tipos de sus campos constituyentes, en orden).

los tipos están predefinidos por un lenguaje de programación; otros están construidos por el programador. El conjunto de tipos en un lenguaje de programación, junto con las reglas que usan tipos para especificar el comportamiento del programa se denominan colectivamente un sistema de tipos.

4.2.2 Componentes de un sistema de tipos

Un sistema de tipos para un lenguaje moderno típico tiene cuatro componentes principales: conjunto de tipos base o tipos integrados; reglas para construir nuevos tipos a partir del tipos existentes; un método para determinar si dos tipos son equivalentes o compatibles; y reglas para inferir el tipo de cada expresión del idioma de origen.

Muchos lenguajes también incluyen reglas para la conversión implícita de valores de un tipo a otro según el contexto. Esta sección describe cada uno de estos en más detalles, con ejemplos de lenguajes de programación populares.

Tipos de base

La mayoría de los lenguajes de programación incluyen tipos base para algunos, sino todos, los siguientes tipos de datos: números, caracteres y valores booleanos. Estos tipos son soportado directamente por la mayoría de los procesadores. Los números suelen venir de varias formas, como enteros y números de coma flotante. Agregar idiomas individuales otros tipos de base. Lisp incluye tanto un tipo de número racional como un recursivo tipo contras. Los números racionales son, esencialmente, pares de enteros interpretados como ratios. Un contra se define como el valor asignado cero o (contras primero resto) donde primero es un objeto, el resto es una desventaja y contras crea una lista a partir de sus argumentos.

Las definiciones precisas para los tipos base y los operadores definidos para ellos, varían según los idiomas. Algunos lenguajes refinan estos tipos base para crear
más; por ejemplo, muchos idiomas distinguen entre varios tipos de números en sus sistemas de tipos. Otros idiomas carecen de uno o más de estos tipos de base. Por ejemplo, c no tiene un tipo de cadena, por lo que los programadores de c usan una matriz
de personajes en su lugar. Casi todos los idiomas incluyen facilidades para construir tipos más complejos de sus tipos base.

Números
Casi todos los lenguajes de programación incluyen uno o más tipos de números como tipos de base. Por lo general, admiten tanto números enteros de rango limitado como números reales aproximados, a menudo llamados números de punto flotante. Mucha programación
Los lenguajes exponen la implementación de hardware subyacente mediante la creación de distintos tipos para diferentes implementaciones de hardware. Por ejemplo, c, c ++ y Java distingue entre enteros firmados y sin firmar

fortran, pl / iyc exponen el tamaño de los números. Tanto cy fortran especificar la longitud de los elementos de datos en términos relativos. Por ejemplo, un doble en fortran tiene el doble de longitud que un real. Ambos idiomas, sin embargo, dan
el compilador controla la longitud de la categoría de número más pequeña. Por el contrario, las declaraciones pl / i especifican una longitud en bits. Los mapas del compilador esta longitud deseada en una de las representaciones de hardware. Por lo tanto, la La implementación de ibm 370 de pl / i asignó un binario fijo (12) y un variable binaria fija (15) a un entero de 16 bits, mientras que una binaria fija (31)
se convirtió en un entero de 32 bits. Algunos lenguajes especifican implementaciones en detalle. Por ejemplo, Java define tipos distintos para enteros con signo con longitudes de 8, 16, 32 y 64
bits. Respectivamente, son byte, short, int y long. Del mismo modo, Java tipo flotante especifica un número de coma flotante ieee de 32 bits, mientras que su doble
type especifica un número de coma flotante ieee de 64 bits. Este enfoque asegura un comportamiento idéntico en diferentes arquitecturas.

Scheme adopta un enfoque diferente. El lenguaje define una jerarquía de tipos de números, pero permite al implementador seleccionar un subconjunto para soportar. sin embargo, el
El estándar establece una cuidadosa distinción entre números exactos e inexactos y Especie un conjunto de operaciones que deben devolver un número exacto cuando todos sus argumentos son exactos. Esto proporciona un grado de flexibilidad al implementador, al tiempo que le permite al programador razonar sobre cuándo y dónde puede producirse una aproximación.

Caracteres
Muchos idiomas incluyen un tipo de carácter. En abstracto, un carácter es una sola letra. Durante años, debido al tamaño limitado de los alfabetos occidentales, esto llevó
a una representación de un solo byte (8 bits) para caracteres, generalmente mapeados en el conjunto de caracteres ascii. Recientemente, más implementaciones, tanto sistemas operativos como lenguajes de programación, han comenzado a admitir caracteres más grandes.

conjuntos expresados en el formato estándar Unicode, que requiere 16 bits. Debe Los idiomas suponen que el juego de caracteres está ordenado, de modo que los operadores de comparación estándar, como <, = y>, funcionan intuitivamente, aplicando el lexicográfico
ordenar. La conversión entre un carácter y un entero aparece en algunos idiomas. Pocas otras operaciones tienen sentido con datos de personajes.

Booleanos
La mayoría de los lenguajes de programación incluyen un tipo booleano que toma dos valores: verdadero y falso. Las operaciones estándar proporcionadas para booleanos incluyen
y, o, xor, y no. Los valores booleanos, o expresiones con valores booleanos, se utilizan a menudo para determinar el flujo de control. c considera los valores booleanos como
subrango de los enteros sin signo, restringido a los valores cero (falso) y una verdad).

Tipos compuestos y construidos
Si bien los tipos básicos de un lenguaje de programación generalmente proporcionan una abstracción adecuada de los tipos reales de datos manejados directamente por el hardware, a menudo son inadecuados para representar el dominio de información necesario.
por programas. Los programas tratan habitualmente con estructuras de datos más complejas, como gráficos, árboles, tablas, matrices, registros, listas y pilas. Estas estructuras constan de uno o más objetos, cada uno con su propio tipo. La habilidad para
Construir nuevos tipos para estos objetos compuestos o agregados es esencial característica de muchos lenguajes de programación. Permite al programador organizar
información de formas novedosas y específicas del programa. Vincular estas organizaciones a el sistema de tipos mejora la capacidad del compilador para detectar programas mal formados. También permite que el lenguaje exprese operaciones de nivel superior, como
asignación de estructura completa.
Tomemos, por ejemplo, Lisp, que proporciona un amplio soporte para la programación. con listas. La lista de Lisp es un tipo construido. Una lista es el valor designadonil o (contras first rest) donde primero es un objeto, el resto es una lista y
cons es un constructor que crea una lista a partir de sus dos argumentos. Una implementación Lisp puede verificar cada llamada a contras para asegurarse de que su segundo argumento sea, de hecho, una lista.

Matrices
Las matrices se encuentran entre los objetos agregados más utilizados. Una matriz agrupa juntos varios objetos del mismo tipo y le da a cada uno un nombre distinto:
aunque sea un nombre implícito calculado en lugar de un nombre explícito diseñado por el programador. La declaración c int a [100] [200]; reserva espacio para 100 × 200 = 20.000 enteros y garantiza que se puedan direccionar utilizando el nombre a. Las referencias a [1] [17] y a [2] [30] acceden de forma distinta y
ubicaciones de memoria independientes. La propiedad esencial de una matriz es que el programa puede calcular nombres para cada uno de sus elementos usando números
(o algún otro tipo discreto ordenado) como subíndices. El soporte para operaciones en arreglos varía ampliamente. fortran 90, pl / i y apl todos admiten la asignación de matrices completas o parciales. Estos lenguajes admiten la aplicación elemento por elemento de operaciones aritméticas a matrices.
por

Matrices 10 × 10 x, y y z, indexadas de 1 a 10, la declaración x = y + z sería sobrescribir cada x [i, j] con y [i, j] + z [i, j] para todo 1 ≤ i, j ≤ 10. apl toma
la noción de operaciones de matriz más allá de la mayoría de los lenguajes; incluye operadores para producto interno, producto externo y varios tipos de reducciones. por
Ejemplo, la reducción de la suma de y, escrito x ← + / y, asigna x la suma escalar de los elementos de y.Una matriz puede verse como un tipo construido porque construimos una matriz
especificando el tipo de sus elementos. Por tanto, la matriz de enteros de 10 × 10 tiene tipo matriz bidimensional de enteros. Algunos idiomas incluyen la matriz
dimensiones en su tipo; por lo tanto, la matriz de enteros de 10 × 10 tiene un tipo diferente que una matriz de números enteros de 12 × 12. Esto permite que el compilador capture operaciones de matriz en el que las dimensiones son incompatibles como un error de tipo. La mayoría de los idiomas permiten matrices de cualquier tipo de base; algunos lenguajes permiten matrices de tipos construidos también.

strings
Los lenguajes de programación tratan las cadenas como un tipo construido. pl / i, para Por ejemplo, tiene cadenas de bits y cadenas de caracteres. Las propiedades, atributos,
y las operaciones definidas en ambos tipos son similares; son propiedades de una cadena. El rango de valores permitido en cualquier posición difiere entre un bit
cadena y una cadena de caracteres. Por lo tanto, verlos como una cadena de bits y una cadena de carácter es apropiado. (La mayoría de los lenguajes que admiten cadenas limitan el soporte integrado a un solo tipo de cadena: la cadena de caracteres). Otros idiomas,
como c, admite cadenas de caracteres manipulandolas como matrices de caracteres. Un verdadero tipo de cadena se diferencia de un tipo de matriz en varios aspectos importantes. Operaciones que tienen sentido en cadenas, como concatenación, traducción y
calculando la longitud, puede que no tenga análogos para arreglos. Conceptualmente, cuerda la comparación debe funcionar desde el orden lexicográfico, de modo que "a" <"boo" y
"tarifa" <"fie". Los operadores de comparación estándar se pueden sobrecargar y utilizar de forma natural. Implementar la comparación para una variedad de caracteres
sugiere una comparación equivalente para una matriz de números o una matriz de estructuras, donde la analogía con las cuerdas puede no ser válida. Del mismo modo, el actual
La longitud de una cadena puede diferir de su tamaño asignado, mientras que la mayoría de los usos de matriz utiliza todos los elementos asignados.

Tipos enumerados
Muchos lenguajes permiten al programador crear un tipo que contiene un conjunto específico de valores constantes. Un tipo enumerado, introducido en Pascal, permite al programador usar nombres auto-documentados para pequeños conjuntos de constantes.
Los ejemplos clásicos incluyen días de la semana y meses. En la sintaxis c, estos pueden ser "07-ch04-161-2 El compilador asigna cada elemento de un tipo enumerado a un valor distinto. Los elementos de un tipo enumerado están ordenados, por lo que las comparaciones entre

Actividades semana 14 (Ene 7-8, 2021)

1) Actividades teóricas:

Estudiar el tema 5.1 "Introduction to Intermediate Representations" del libro Engineering a Compiler (2nd Ed 2012) de Cooper, Torczon.

5.1 INTRODUCCIÓN

Los compiladores se organizan normalmente como una serie de pases. Como compilador
Deriva conocimiento sobre el código que compila, debe transmitir esa información de una pasada a otra. Por tanto, el compilador necesita una representación por todos los hechos que se derivan sobre el 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 usa mientras transforma el código de la fuente idioma a su idioma de destino. Durante la traducción, la forma ir de la entrada
programa es la forma definitiva del programa. El compilador no hace referencia volver al texto fuente; en cambio, busca la forma ir del código. Las propiedades de los ir o irs de un compilador tienen un efecto directo sobre lo que el compilador puede
hacer con el código.

Casi todas las fases del compilador manipulan el programa en su forma ir. Por lo tanto, las propiedades del ir, como los mecanismos para leer y escribir campos específicos, para encontrar hechos o anotaciones específicas y para navegar
alrededor de un programa en su forma, tienen un impacto directo en la facilidad para escribir el pases individuales y sobre el costo de ejecutar esos pases.

Hoja de ruta conceptual

Este capítulo se centra en las cuestiones que rodean el diseño y uso de un ir en compilación. La sección 5.1.1 proporciona una descripción taxonómica de irs y sus propiedades. Muchos escritores de compiladores consideran árboles y gráficos como la representación natural de los programas; por ejemplo, los árboles de análisis capturan fácilmente la
derivaciones construidas por un analizador. La sección 5.2 describe varios IR basados en árboles y gráficos. Por supuesto, la mayoría de los procesadores a los que se dirigen los compiladores tienen lenguajes ensambladores como lengua materna. En consecuencia, algunos compiladores usar ir lineales con el racional de que esos ir exponen las propiedades del objetivo
código de la máquina que el compilador debería ver explícitamente. La sección 5.3 examina irs lineales.

Las secciones finales de este capítulo tratan temas que se relacionan con los irs pero no lo son, estrictamente hablando, sus problemas de diseño. La sección 5.4 explora problemas relacionados con naming: la elección de nombres específicos para valores específicos. Los nombres pueden tener un fuerte impacto en el tipo de código generado por un compilador. Esa discusión
incluye una vista detallada de un ir específico y ampliamente utilizado llamado formulario de asignación única estática, o ssa. La sección 5.5 proporciona una descripción general de alto nivel de cómo

el compilador crea, usa y mantiene tablas de símbolos. La mayoría de los compiladores compilan una o más tablas de símbolos para contener información sobre nombres y valores y
para proporcionar un acceso eficiente a esa información.

Visión general
Para transmitir información entre sus pasadas, un compilador necesita una representación por todo el conocimiento que se deriva del programa que se está compilando.
Por tanto, casi todos los compiladores utilizan alguna forma de representación intermedia para modelar el código que se analiza, traduce y optimiza. La mayoría pasa el compilador consume ir; el escáner es una excepción. La mayoría de pases en el
compilador produce ir; los pases en el generador de código pueden ser excepciones. Muchos Los compiladores modernos utilizan varios IR durante el transcurso de una sola compilación.
En un compilador con estructura de paso, el ir sirve como el principal y definitivo representación del código.
El ir de un compilador debe ser lo suficientemente expresivo para registrar todos los hechos útiles que el compilador podría necesitar transmitir entre pasadas. El código fuente es
insuficiente para este propósito; el compilador deriva muchos hechos que no tienen representación en código fuente, como las direcciones de variables y constantes o el registro en el que se pasa un parámetro dado. Para grabar todos
el detalle que el compilador debe codificar, la mayoría de los escritores de compiladores aumentan el ir con tablas y conjuntos que registran información adicional. Consideramos estos
tablas parte del ir.

Seleccionar un ir apropiado para un proyecto de compilador requiere una comprensión del idioma de origen, la máquina de destino y las propiedades de las aplicaciones que traducirá el compilador. Por ejemplo, una fuente a fuente
El traductor puede usar un ir que se parezca mucho al código fuente, mientras que un El compilador que produce código ensamblador para un microcontrolador podría obtener
mejores resultados con un ir similar a código ensamblador. De manera similar, un compilador para c podría necesitar anotaciones sobre los valores de puntero que son irrelevantes en un compilador de Perl, y un compilador de Java mantiene registros sobre la jerarquía de clases
que no tienen contraparte en el compilador de CA.
La implementación de un ir obliga al escritor del compilador a centrarse en cuestiones prácticas. El compilador necesita formas económicas de realizar las operaciones que realiza
frecuentemente. Necesita formas concisas de expresar la gama completa de constructos que puede surgir durante la compilación. El escritor del compilador también necesita mecanismos
que permiten a los humanos examinar el programa de infrarrojos fácil y directamente. Interés propio debería asegurarse de que los redactores del compilador presten atención a este último punto.
Finalmente, los compiladores que usan un ir casi siempre hacen múltiples pasadas sobre el ir para un programa. La capacidad de recopilar información en una pasada y usarla en otra. mejora la calidad del código que puede generar un compilador.

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 con nombre privado. Los procedimientos ayudan a definir interfaces entre componentes del sistema; las interacciones entre componentes son normalmente estructurado mediante llamadas a procedimientos. Finalmente, los procedimientos son los básicos

unidad de trabajo para la mayoría de los compiladores. Un compilador típico procesa una colección de procedimientos y produce código para ellos que se vinculará y ejecutará correctamente con otras colecciones de procedimientos compilados.

Esta última característica, a menudo llamada compilación separada, nos permite crear grandes sistemas de software. Si el compilador necesitaba el texto completo de un programa para cada compilación, los grandes sistemas de software serían insostenibles. Imagine volver a compilar una aplicación de línea multimillonaria para cada cambio de edición realizado durante

¡desarrollo! Por lo tanto, los procedimientos juegan un papel fundamental en el diseño del sistema y ingeniería como lo hacen en el diseño del lenguaje y la implementación del compilador. Este capítulo se centra en cómo los compiladores implementan la abstracción de procedimientos.

Hoja de ruta conceptual

Para traducir un programa en lenguaje fuente a código ejecutable, el compilador debe mapear todas las construcciones del lenguaje fuente que utiliza el programa en operaciones y estructuras de datos en el procesador de destino. El compilador necesita una estrategia para cada una de las abstracciones soportadas por el idioma fuente. Tesis Las estrategias incluyen tanto algoritmos como estructuras de datos que están integrados en el código ejecutable. Estos algoritmos de tiempo de ejecución y estructuras de datos se combinan para implementar el comportamiento dictado por la abstracción. Estas estrategias en tiempo de ejecución también requieren soporte en tiempo de compilación en forma de algoritmos y datos. estructuras que se ejecutan dentro del compilador.

otro procedimiento (o el sistema operativo). El destinatario puede devolver un valor a su interlocutor, en cuyo caso el procedimiento se denomina función. Esta interfaz entre procedimientos permite a los programadores desarrollar y probar partes de un programa contra problemas en otros procedimientos.

Los procedimientos juegan un papel importante en la forma en que los programadores desarrollan software y los compiladores traducen programas. Tres abstracciones críticas que

Los procedimientos proporcionan permiten la construcción de programas no triviales.

1. Abstracción de llamadas de procedimiento Soporte de lenguajes de procedimiento abstracción para llamadas a procedimientos. Cada idioma tiene un estándar mecanismo para invocar un procedimiento y mapear un conjunto de argumentos, o parámetros, desde el espacio del nombre de la persona que llama hasta el espacio del nombre de la persona que llama.

Esta abstracción típicamente incluye un mecanismo para devolver el control al leer y continuar la ejecución en el punto inmediatamente después de la llamada.

La mayoría de los lenguajes permiten que un procedimiento devuelva uno o más valores al llamador. El uso de convenciones de vinculación estándar, a veces denominadas como secuencias de llamada, permite al programador invocar el código escrito y compilado por otras personas y en otras ocasiones; deja la aplicación invocar rutinas de biblioteca y servicios del sistema.

Espacio de nombres En la mayoría de los idiomas, cada procedimiento crea un nuevo y espacio de nombre protegido. El programador puede declarar nuevos nombres, como variables y etiquetas, sin preocuparse por el contexto circundante. Dentro el procedimiento, esas declaraciones locales tienen prioridad sobre cualesquiera declaraciones para los mismos nombres. El programador puede crear parámetros para el procedimiento que permite a la persona que llama mapear valores y variables en el espacio de nombre de la persona que llama en parámetros formales en el espacio de nombre de la persona que llama.

Dado que el procedimiento tiene un espacio de nombres conocido y separado, puede funcionan correctamente y de forma coherente cuando se llaman desde diferentes contextos. La ejecución de una llamada crea una instancia del espacio de nombre de la persona que llama. La llamada debe crear almacenamiento para los objetos declarados por el destinatario. Esta asignación debe ser automático y eficiente, una consecuencia de llamar al procedimiento.

Los procedimientos de interfaz externa definen las interfaces críticas entre los partes de grandes sistemas de software. La convención de vinculación define reglas que asignan nombres a valores y ubicaciones, que preservan el nombre de la persona que llama entorno de ejecución y crear el entorno del destinatario, y que transferir el control de la persona que llama a la persona que llama y viceversa. Crea un contexto en que el programador puede invocar con seguridad código escrito por otras personas. La existencia de secuencias de llamadas uniformes permite el desarrollo y uso de bibliotecas y llamadas al sistema. Sin una convención de vinculación, tanto el El programador y el compilador necesitarían conocimientos detallados sobre el Implementación del destinatario en cada llamada a procedimiento. Así, el procedimiento es, en muchos sentidos, la abstracción fundamental que subyace a los lenguajes tipo Algol. Es una fachada elaborada creada en colaboración por el compilador y el hardware subyacente, con ayuda del sistema operativo. Los procedimientos crean variables con nombre y mapean ellos a direcciones virtuales; el sistema operativo asigna direcciones virtuales a direcciones físicas. Los procedimientos establecen reglas para la visibilidad de nombres y direccionalidad; el hardware normalmente proporciona varias variantes de carga y operaciones de tienda. Los procedimientos nos permiten descomponer grandes sistemas de software en componentes; Los enlazadores y cargadores los unen en un programa ejecutable que el hardware puede ejecutar avanzando su contador de programa y siguientes ramas.

Una gran parte de la tarea del compilador es implementar el código necesario para darse cuenta de las distintas piezas de la abstracción del procedimiento. El compilador debe Dicte el diseño de la memoria y codifique ese diseño en el programa generado. Dado que puede compilar los diferentes componentes del programa en diferentes momentos, sin conocer sus relaciones entre ellos, este El diseño de la memoria y todas las convenciones que induce deben estar estandarizadas. y aplicado uniformemente. El compilador también debe utilizar las diversas interfaces proporcionado por el sistema operativo, para manejar la entrada y salida, administrar memoria y comunicarse con otros procesos. Este capítulo se centra en el procedimiento como una abstracción y los mecanismos que el compilador utiliza para establecer su abstracción de control, espacio de nombres y interfaz con el mundo exterior.

Practicas

Capitulo 2

1.

a) $\sum = \{a, b\}$

Q= { s_0,s_1,s_2,f}

b) $\sum = \{0,1\}$

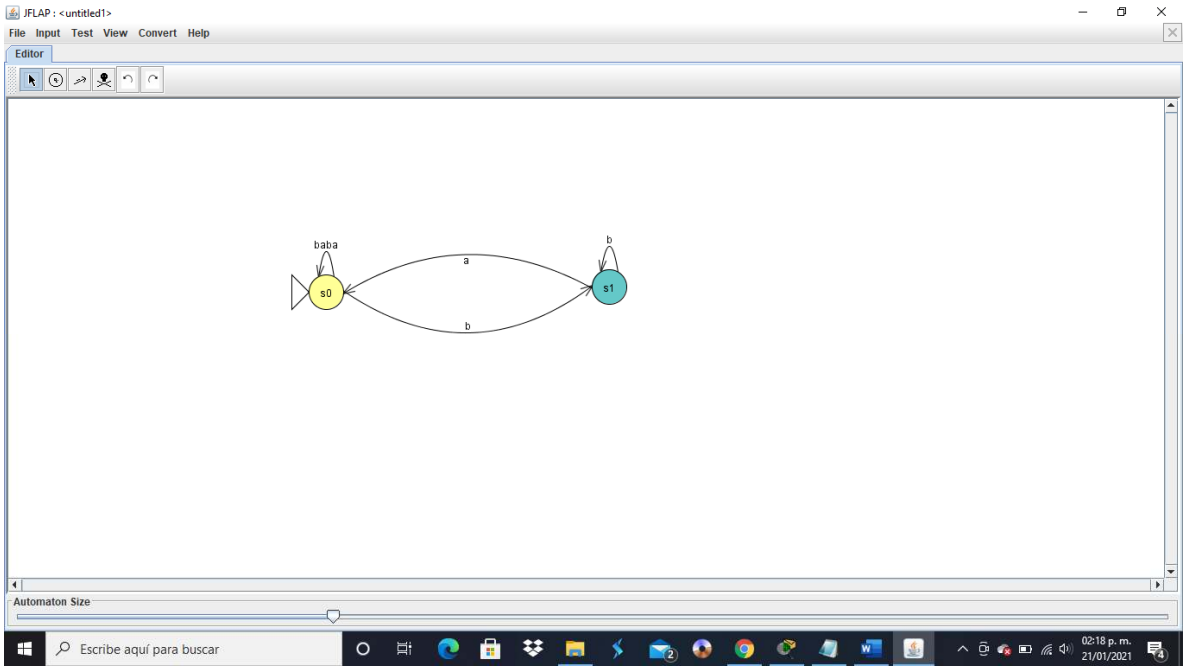
Q= { s_0,s_1,s_2, s_3, s_4,f}

c) a) $\sum\{a, b\}$

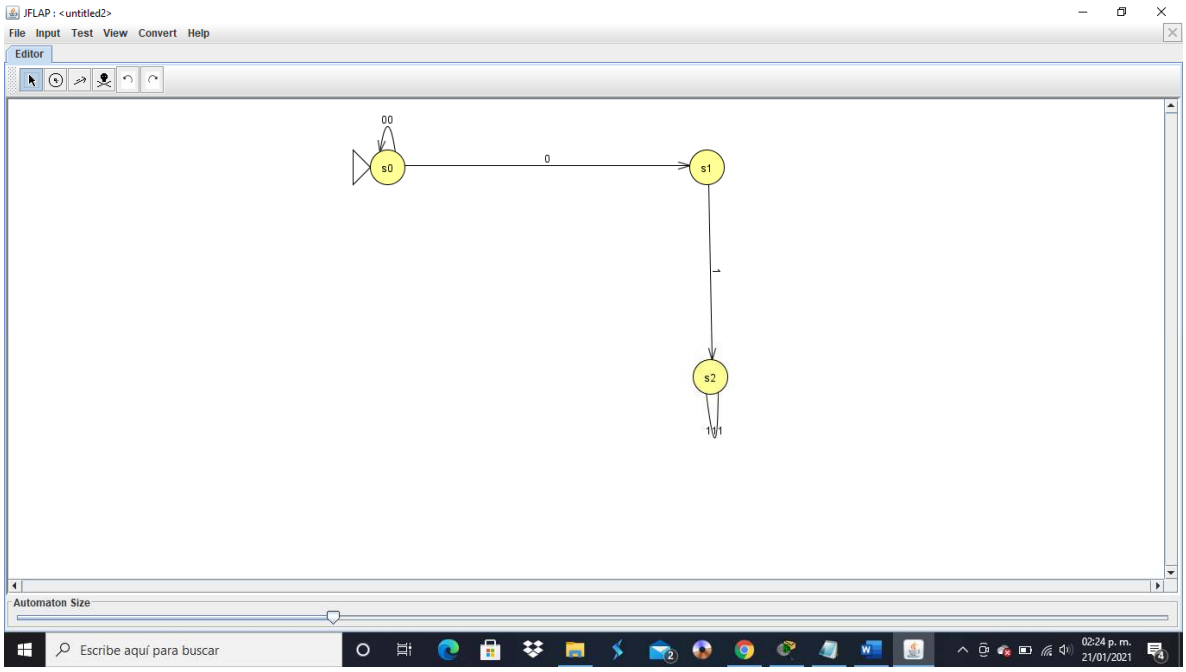
Q= { s_0,s_1,s_2, s_3, s_4, s_5, s_6,f}

2.

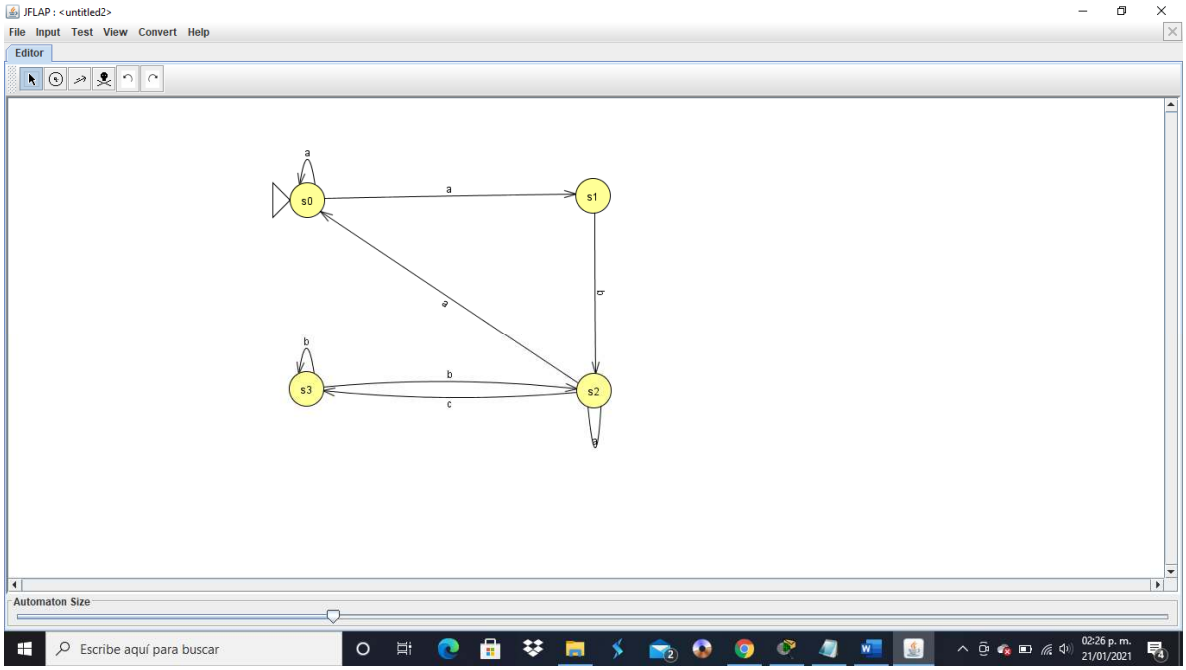
a) baba



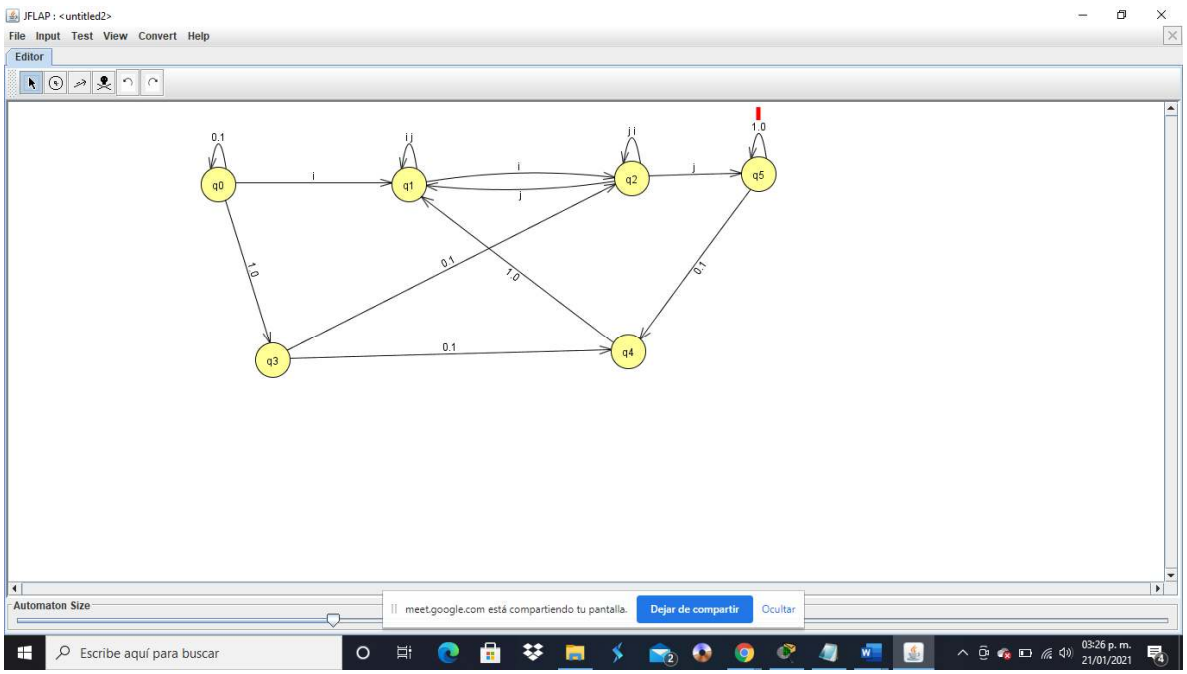
b)


JFLAP : <untitled2>
File Input Test View Convert Help
Editor
00
0
s0
s1
1
s2
1
Automaton Size
Escribe aquí para buscar
02:24 p. m.
21/01/2021

c)


JFLAP : <untitled2>
File Input Test View Convert Help
Editor
a
a
s0
s1
a
b
b
s3
b
c
s2
Automaton Size
Escribe aquí para buscar
02:26 p. m.
21/01/2021

3)



4)

a) $E.R$=$(0-9^*)id^* + (0-9)\backslash H^*$

b)$E.R = (0-9)\backslash W^*$

c)$E.R = (0-9).D^* + (0-9)0.D\backslash * + (0-9)0.0D \backslash^*$

5)

a)$(0+0)^*1(0+0) + (1+1)^*1(1+1)(1+1)$

b)$(0+1)^*1(0+1)^* + (0+1) + 1^*(0+1)(0+1)$

c) $L = (abcdfg) + (hijklmn) + (opqrstu) + (vwxyz)$

d)$(xyzwy) + (x) + (y)^* + (xy) + (z) + (wy)^* + (wxyz^*)$

e) $w = (0-9)^* + (1{,}3)d + (t-^* \backslash) + (\backslash d^*) + ^{11})$

6)

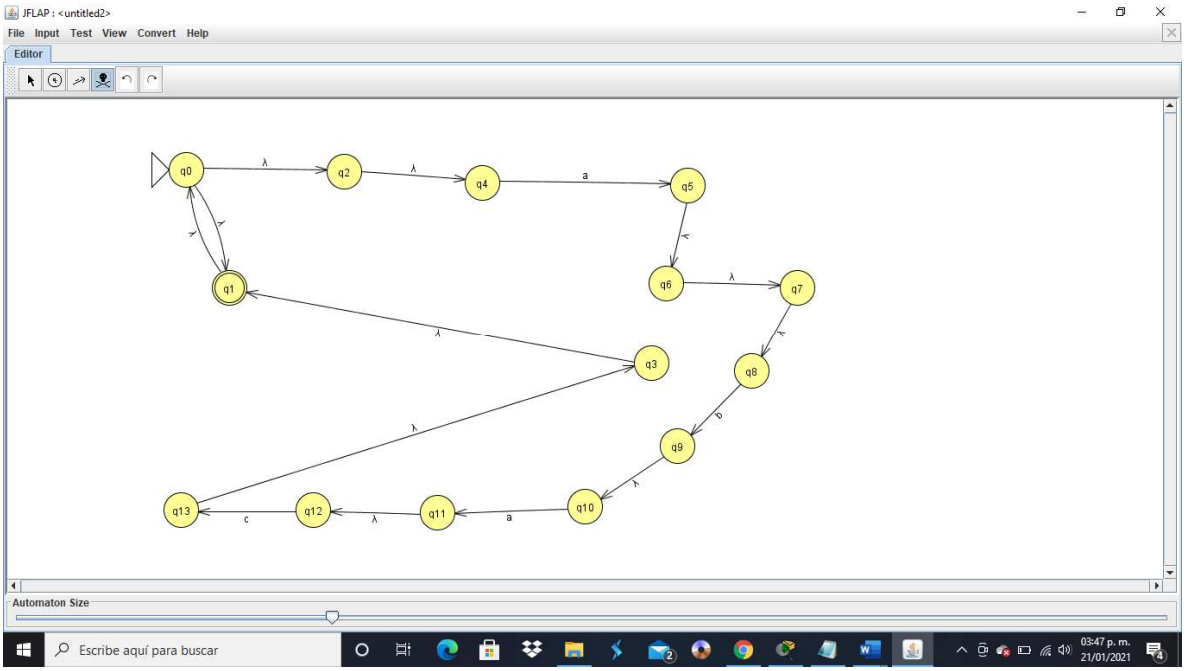
a) (ll^[ ]) $* \lambda$

b) $(b \backslash *) + (z)$

c) $L = \{A, Z\} + \{a, z\}$

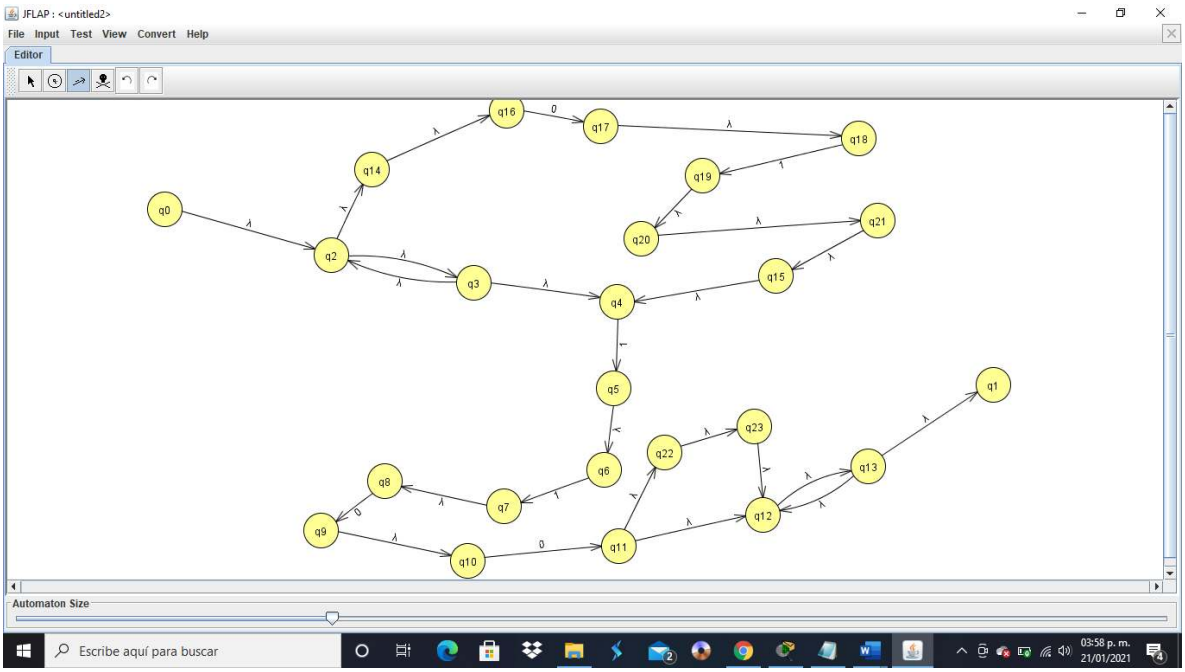
d)$(0 - 9) \backslash d^* \lambda$

7)



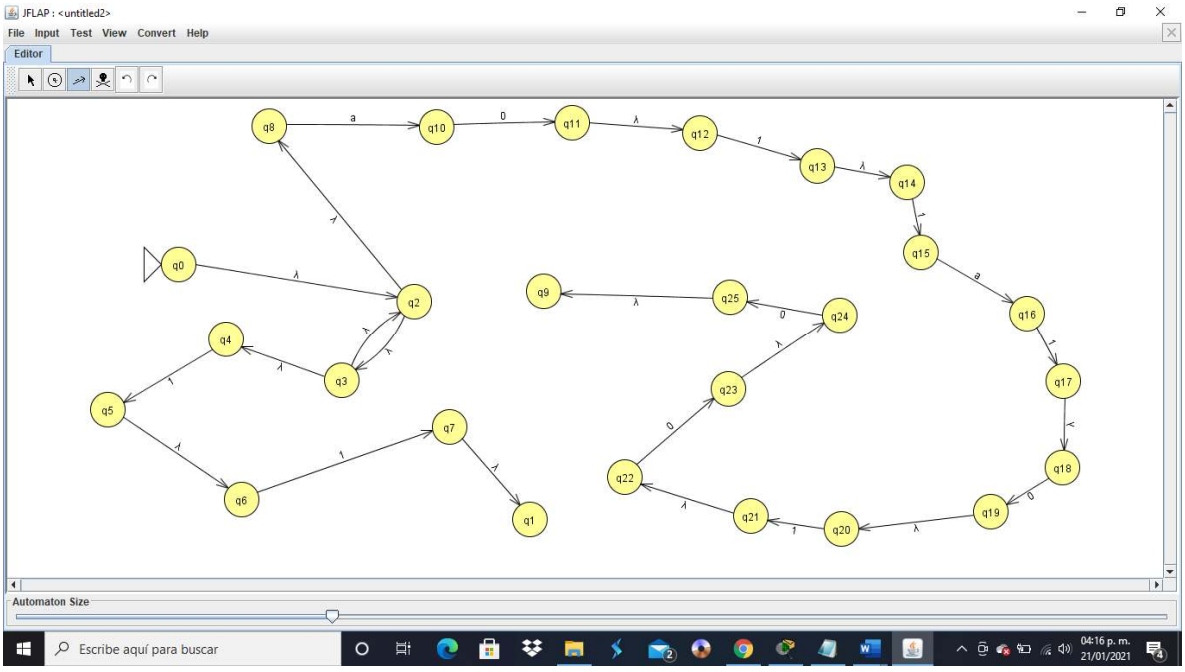
$(a, b, c)^* \lambda$

b)



$(0I1)^* + 01^*$

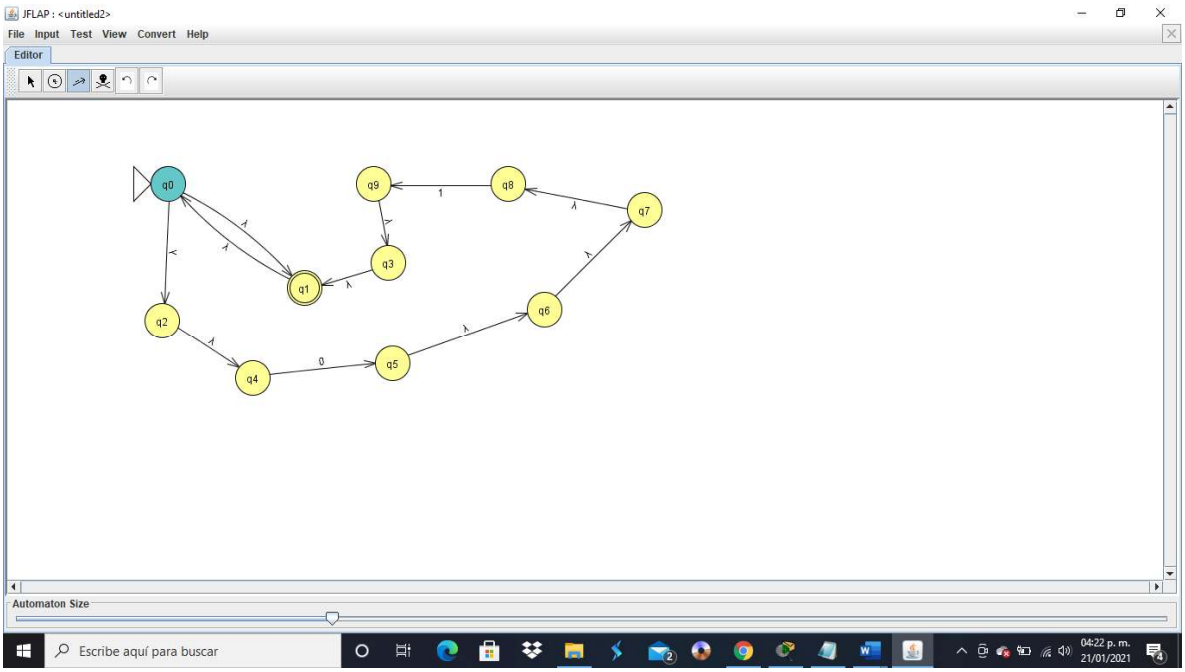
c)



$1 + (01I10) + 0^*$

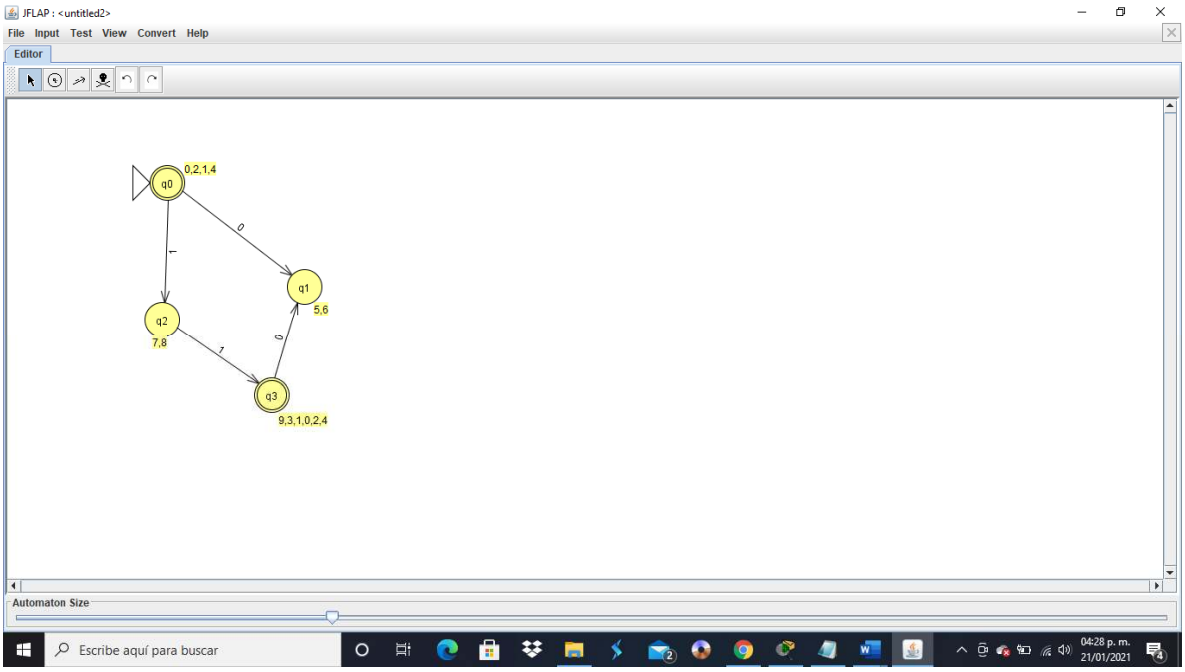
8)

a) original



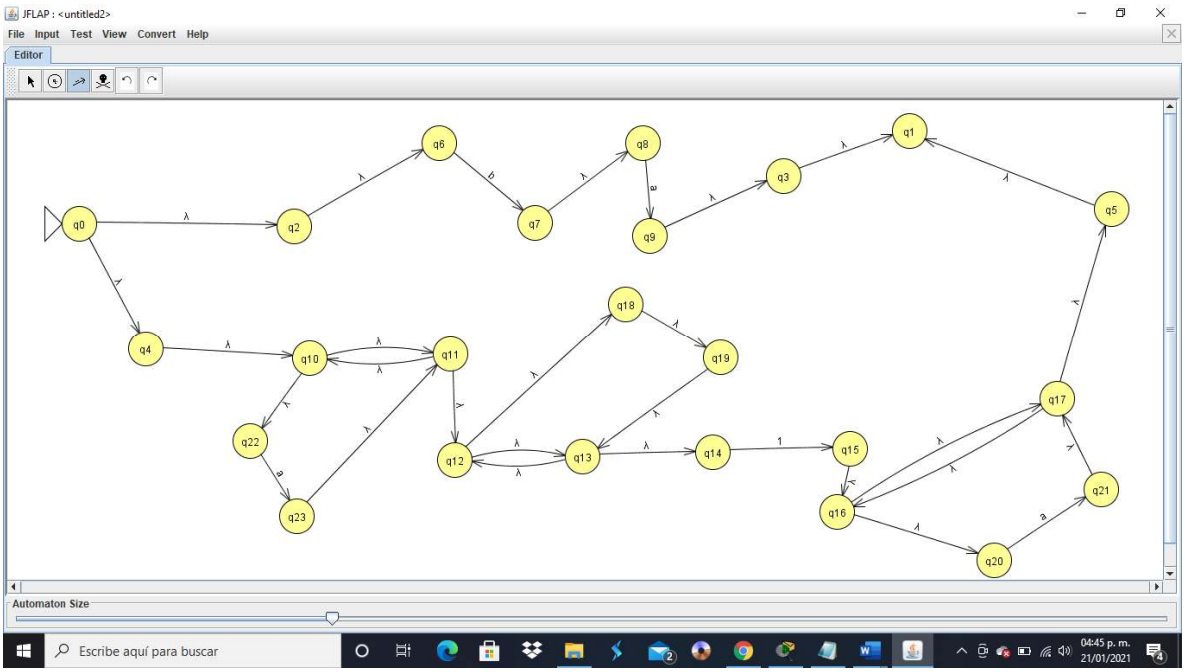
R= es equivalente

Minimizado

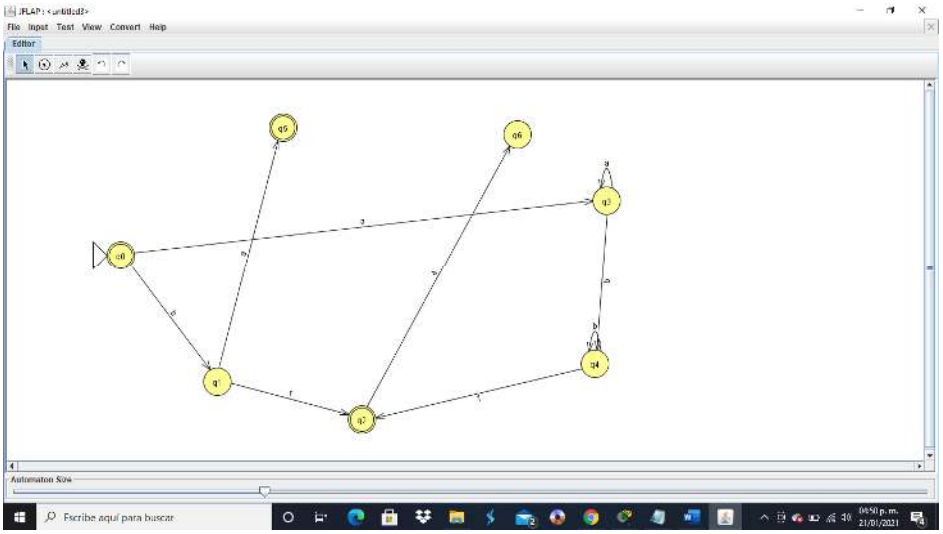


b)

original



Simplificado



R= son equivalentes

9)

a) R= cuando la transacción de dos estados o más sea aceptado por lenguajes