



INSTITUTO TECNOLÓGICO DE IZTAPALAPA

INGENIERIA EN SISTEMAS COMPUTACIONALES

APUNTES Y PRACTICAS

TEMA:

APUNTES Y PRACTICAS DE LA MATERIA: LENGUAJES Y
AUTOMATAS II

PROFESOR:

ABIEL TOMAS PARRA HERNANDEZ

PRESENTA:

RODRIGUEZ GARCIA JESUS

NO. DE CONTROL:

181080027

CIUDAD DE MEXICO

JUNIO / 2021





Índice

Actividades semana marzo 8-12, 2021.....	1
¿Son las computadoras todopoderosas?	1
Golpe cibernético	3
Actividades semana marzo 16-19, 2021.....	7
<i>The Imitation Game (El código enigma)</i>	7
Ensayo “Untangling the tale of Ada Lovelace”.....	8
Actividades semana abril 12-16, 2021.....	12
Resumen del tema: <i>Translator (computing) de Wikipedia</i>	12
Resumen del Artículo <i>The Latin American Supercomputing Ecosystem for Science</i>	14
Actividades semana abril 19-23, 2021.....	17
1.1 Introducción.....	17
1.2 Estructura del compilador	20
LLVM.....	21
Actividades semana abril 26-30, 2021.....	24
1.3 Resumen de la traducción	24
Actividades semana mayo 3-7, 2021	30
2.7 Resumen y perspectiva del capítulo.....	30
3.7 Resumen y perspectiva.....	31
4.6 Resumen y perspectiva.....	32
Prácticas y Actividades en clase.....	34
Actividades en clase	34
Prácticas	36
Bibliografía.....	50



Actividades semana marzo 8-12, 2021

1) Ver con atención la conferencia:

¿Son las computadoras todopoderosas? de Sergio Rajsbaum

2) Leer a detalle el artículo 'Golpe cibernético' global: del 11 de septiembre al 6 de enero de Alfredo Jalife Rahme

Importante: Realizar las actividades 1 y 2 antes de la sesión síncrona para comentarlas en grupo, se recomienda ampliamente que las comenten también con sus respectivos equipos de trabajo antes de la sesión síncrona.

3) Hacer el test de personalidad, después leer a detalle la descripción completa de la personalidad y redactar su opinión del resultado.

4) Hacer un collage digital de sus gustos y pasiones incluyendo un párrafo de semblanza personal de 5 líneas (si trabajan, indicar su horario y actividad laboral).

Importante: Enviar las actividades 3 y 4 juntas por correo al profesor, deben usar su correo institucional, no se tomarán en cuenta correos desde sus cuentas personales. Fecha límite de entrega el domingo 14 de marzo a la media noche.

¿Son las computadoras todopoderosas?

Para empezar, ¿Qué es una computadora?, en el pasado la respuesta a esta pregunta era que era una caja muy mágica, que podía diagnosticar un perrito enfermo, realizar cálculos matemáticos, entre otras cosas. De acuerdo con Turing la computadora es un "Control finito que en todo momento va cambiando de estado". Está compuesta por un conjunto de quintupletas que describen sus posibles operaciones de cómo se mueve la cabeza y como cambia de estados, es por ello que se dice que el conjunto de quintupletas es infinito.

Existen muchos tipos de computadoras como las de madera o reloj calculador, también existen de metal como las de Blaise pascal que creó una máquina de

calcular (operaciones básicas como las sumas). Existen también máquinas electromecánicas como la de James Thompson, la cual tenía como propósito resolver ecuaciones diferenciales. Las invenciones de los seres humanos fueron creciendo respecto a las computadoras por ellos es que usando electroimanes se creó una máquina para jugar ajedrez en 1914. También existieron las máquinas o computadoras electrónicas como colossus en 1942.

Lo más sobresaliente para nosotros respecto a la existencia de los diferentes tipos de computadoras es que también existen las computadoras de carne y hueso como el autista Daniel Tammet, gran capacidad de su cerebro, puede ver la textura y el color de un número, habla 11 idiomas, en una semana aprendió a hablar irlandés y eso fue porque lo retaron al decirle que era uno de los idiomas más difíciles

Nos llamó la atención el hecho de que Norbert Wiener rompió un paradigma del ser humano, y es por el que nacen las ciencias de la Computación.

¿Por qué no hay ciencias de los submarinos?

Es porque las ciencias de la computación se convierten en una disciplina que nos permite llegar a un mejor entendimiento de nosotros mismos y de nuestro universo, es decir de todo lo que nos rodea. Ada Lovelace, es llamada la madre de la programación. Ya que creó los llamados métodos para resolver costos y problemas. Alan Turing fue quien inventó la noción abstracta del cómputo. (Algo muy importante en la historia del ser humano).

Nos llama la atención el gran hecho de que "EL CONJUNTO DE PROBLEMAS NO ES NUMERABLE", esto quiere decir que hay un número infinito de problemas que queremos resolver, pero en la realidad en la que estamos hay más problemas que soluciones.

Abstracción, el computólogo es experto en abstracción, es la que nos permite expresar el entendimiento de un problema, nos permite manejar los problemas más complejos, como las páginas web, transacciones electrónicas, programas, todas estas son capas de abstracción.

Física cuántica – Ni siquiera en el mundo de las matemáticas es posible que una persona lo sepa todo, hay muchos teoremas que no tienen demostración de que sean verdaderas.

En la computabilidad existe un universo infinito de problemas que no sabemos cómo solucionar, hay problemas que son posibles de resolver, pero la complejidad juega un papel importante es decir podríamos tardar hasta millones de años para poder resolverlos y nosotros los humanos no podemos vivir mucho para poder encontrar la solución de estos problemas.

Para cada problema no computable, hay otro problema con menos probabilidad de ser computable, para cada problema no computable hay un problema mucho más difícil, solo nos queda adaptarnos a la práctica e intentar solucionar los problemas computables.

La realidad nadie la ve, cada uno tiene su propia perspectiva del mundo en que vivimos. No hay historia verdadera solo hay perspectivas de cada quien. “El eterno silencio de estos espacios infinitos me llena de temor” Pascal.

Golpe cibernético

El derribo hollywoodense el 11 de septiembre de 2 torres gemelas —luego la de un tercer inmueble por la tarde—, que al parecer terminó “implosión controlada”, desembocó en el principio de la restricción de las libertades primordiales en EEUU

con el republicano Baby Bush, además de sus 2 guerras fallidas en Irak y Afganistán para el control del petróleo de Oriente Medio.

Hace 9 años, Leon Panetta, secretario de Protección con Obama, advirtió que EEUU vivía un "instante cibernético pre-Pearl Harbor 11/9, una vez que 4 actores –Rusia, China, Irán e identidades terroristas sin especificar– se disponen a golpear la crítica infraestructura de EEUU".

Washington intentó obligar su modelo SOPA (Stop En línea Piracy Act) que causó la revolución de la ciudadanía, por lo cual Panetta, el exsecretario de Custodia, impulsó CISP (Cyber Intelligence Sharing and Protection): "iniciativa de ley por orden ejecutiva sin anuencia del Congreso y de los habitantes, lo que posibilita compartir el tráfico de información de internet entre el Régimen de EEUU y privilegiadas transnacionales tecnológicas.

Hoy, con el 6 de enero el propósito es triple:

De esta forma la grotesca toma del Capitolio está siendo explotada por los pletóricos enemigos de Trump del complejo Pentágono - Deep State - Wall Street - Silicon Valley, incluyendo al eje demócrata de los Clinton/los Obama aliado a George Soros y a un nada desdeñable conjunto de poderosos republicanos: Baby Bush, Mitt Romney, Liz Cheney (hija del exvicepresidente Dick Cheney), etcétera. Ahora con el progreso de China en la carrera de la IA (inteligencia artificial), el Pentágono recupera el control del GAFAM/ Twitter por medio del Consejo de Innovación de Protección (DIB: Defense Innovation Board) que jefatura el israelí de Estados Unidos Joshua Marcuse.

29 de marzo 2019, 13:40 GMT

Jatras avizora que "desde el 20 de enero, seguirá una ráfaga de ocupaciones ejecutivas y legislaciones para quitar los últimos vestigios de lo cual ha sido un territorio independiente" una vez que "la Primera Enmienda (libertad de expresión, de creencia y de asociación) son solo formalidades ahora y la Segunda Enmienda (derecho a portar armas, lo que es considerado importante al criterio de Estados Unidos de ciudadanía libre) está en serio riesgo".

Se desprende un trilema que exacerbó el 6 de enero:

¿La tosca toma del Capitolio a lo largo del día —se implica que los golpes verdaderos se propinan en las noches— fue útil de coartada para la imposición de la censura por la cibercracia del GAFAM/ Twitter bajo la égida de la ciberseguridad del Pentágono?

El Pentágono - Deep State - Wall Street - Silicon Valley conocidos como los pletóricos enemigos de Trump, se les atribuye a ellos la acción de censura hacia Donald Trump, por medio del accionar del Big Tech del GAFAM (Google/ Apple/ Facebook/ Amazon/ Microsoft) Twitter que ha emergido como una omnipotente cibercracia que ha propinado un golpe cibernético a nivel mundial y que controlará a la mayoría de los países valetudinarios que fueron hechos prisioneros en sus redes.

Con esto podemos decir que Donald Trump y todos sus seguidores fueron censurados por esta organización o cibercracia llamada GAFAM, la cual se creó con el fin de atacar a el llamado movimiento "pre-Pearl Harbor 11/9" quien contaba con cuatro actores: Rusia, China, Irán e identidades terroristas sin especificar.

Esto afecta a todo el mundo ya que si un país decide ponerse en contra de Estados Unidos puede ser víctima de esta "cárcel invisible" llamada GAFAM, en la cual no



posees la libertad de expresarse libremente, a México esto le afecta ya que Estados Unidos con el nuevo TLC ha obligado a México a no tener relaciones exteriores con Rusia y China, ya que los considera enemigos por los puntos señalados anteriormente, dando por entendido que Estados Unidos no ve a México como un aliado si no como un país más del cual puede sacar provecho.

Para terminar, a nosotros como personas quizás nos afecta directamente o indirectamente la organización GAFAM, ya que la mayoría de nosotros tiene cuentas o hace uso de los servicios de estas Empresas que pertenecen a dicha unión (GAFAM), el hecho de hacer uso de sus servicios o tener una cuenta personal con ellos, implica que tienen acceso a cierta información personal, dándoles el poder de censurarnos cuando nos les parezca correcto algún comentario o acción que nosotros hagamos.

Actividades semana marzo 16-19, 2021

Actividades individuales:

- 1) Ver con atención la conferencia:
- 2) Ver con atención la película The Imitation Game (mal traducida al español con el nombre 'Código Enigma', también la encuentran en la plataforma de video streaming Amazon Prime).

Importante: Se recomienda que generen sus respectivos apuntes personales de todas las actividades semanales y que las comenten con sus equipos de trabajo en las horas de sesión asíncrona.

Actividades en equipo:

- 3) Leer detalladamente el ensayo Untangling the Tale of Ada Lovelace de Stephen Wolfram, después redactar un resumen de dos páginas en equipo en un Google Doc donde cada integrante deberá escribir sus datos (nombre completo, número de control y grupo).

Importante: El representante del equipo debe enviar por correo al profesor la liga de acceso al documento de la actividad 3 empleando su correo institucional. Fecha límite de entrega el domingo 21 de marzo a la media noche.

The Imitation Game (El código enigma)

La película trata sobre el Código Enigma, el cual es creado por una computadora, que usaban los Nazis para comunicarse estrategias de guerra entre ellos. Los mensajes se transmitían por señales de radio, flotaban en el aire al alcance de cualquiera, pero eran intraducibles.

Turing fue el líder del equipo que logró decodificarlo, como un hacker, y lo hizo rompiendo paradigmas: inventando una máquina que, mediante un algoritmo, pudiera leer en minutos cualquier mensaje que hubiera sido creado con el mismo

código de programación que otro y que otro y que otro, y que todos los que se enviaban entre sí los alemanes fueran descifrados.

La película habla principalmente de una guerra entre computadoras, en la que la computadora británica venció a la computadora alemana. Aunque el trasfondo de la historia son campos de batalla, las armas y los muertos, los familiares devastados, las ciudades bombardeadas, en la película no observamos la catástrofe, pero sí una verdad aún más injusta y cruel, mientras que en las guerras hay hombres dando su vida en los campos de batalla, son solo carne de cañón; las grandes decisiones la toman los hombres, seguros, bien comidos y bien vestidos, y con la mente fría y calculadora. Es ahí donde se decide el destino de miles de vidas, donde se opta por el mal menor. El código enigma nos permite ver, el dilema trágico en el que estos hombres de mentes y condiciones privilegiadas vivían, sabiendo que mientras ellos estaban relativamente seguros, de su trabajo dependían las vidas de millones.

El Código Enigma nos da el mensaje de que junto con los soldados que murieron en los campos de batalla, los científicos, ingenieros y matemáticos también ayudaron a ganar la Segunda Guerra Mundial, y que, a pesar de ello, gente como Turing no fue reconocida en su momento debido a la discriminación y el miedo a lo “diferente”.

Ensayo “Untangling the tale of Ada Lovelace”

Ada Lovelace nació hace 200 años hoy. Para algunos, es una gran heroína en la historia de la informática; para otros una figura menor sobreestimada. Durante mucho tiempo he sentido curiosidad por saber cuál es la verdadera historia. Y en

preparación para su bicentenario, decidí intentar resolver lo que para mí siempre ha sido el “misterio de Ada”.

Fue mucho más difícil de lo que esperaba. Hay una sorprendente cantidad de desinformación e interpretación errónea por ahí. Pero después de un poco de investigación, incluida la de ver muchos documentos originales, siento que finalmente he llegado a conocer a Ada Lovelace y he comprendido su historia. De alguna manera es una historia ennoblecedora e inspiradora; de alguna manera es frustrante y trágico.

La vida temprana de Ada

Empecemos por el principio. Ada Byron, como la llamaban entonces, nació en Londres el 10 de diciembre de 1815 de padres de la alta sociedad recién casados. Su padre, Lord Byron (George Gordon Byron) tenía 27 años y acababa de alcanzar el estatus de estrella de rock en Inglaterra por su poesía. Su madre, Annabella Milbanke, era una heredera de 23 años comprometida con causas progresistas, que heredó el título de baronesa Wentworth. Su padre dijo que le dio el nombre de "Ada" porque "es corto, antiguo, vocálico".

Charles Babbage

¿Cuál fue la historia de Charles Babbage? Su padre era un orfebre y banquero emprendedor y exitoso (aunque personalmente distante). Después de varias escuelas y tutores, Babbage fue a Cambridge para estudiar matemáticas, pero pronto se propuso modernizar la forma en que se realizaban allí, y con sus amigos de toda la vida John Herschel (hijo del descubridor de Urano) y George Peacock (más tarde un pionero en álgebra abstracta), fundó la Sociedad Analítica (que más

tarde se convirtió en la Sociedad Filosófica de Cambridge) para impulsar reformas cómo reemplazar la notación basada en puntos de Newton ("británica") para el cálculo con la notación basada en funciones de Leibniz ("continental").

Lo último

Hubo calculadoras mecánicas mucho antes de Babbage. Pascal hizo uno en 1642, y ahora sabemos que incluso hubo uno en la antigüedad. Pero en la época de Babbage, estas máquinas seguían siendo solo curiosidades, no lo suficientemente fiables para el uso práctico diario. Las tablas fueron hechas por computadoras humanas, con el trabajo dividido en un equipo, y los cálculos de nivel más bajo se basaron en la evaluación de polinomios (por ejemplo, de expansiones de series) utilizando el método de diferencias.

De vuelta a Ada

El encuentro de Ada con el motor de diferencias parece ser lo que encendió su interés por las matemáticas. Había llegado a conocer a Mary Somerville, traductora de Laplace y una conocida expositora de la ciencia, y en parte gracias a su estímulo, pronto, por ejemplo, estaba estudiando con entusiasmo a Euclides. Y en 1834, Ada realizó una gira filantrópica de fábricas en el norte de Inglaterra que estaba haciendo su madre, y quedó muy impresionada con el equipo de alta tecnología que tenían.

Papel de Ada

A pesar de la falta de apoyo en Inglaterra, las ideas de Babbage adquirieron cierta popularidad en otros lugares, y en 1840 Babbage fue invitado a dar una



conferencia sobre la Máquina Analítica en Turín, y recibió honores del gobierno italiano.

Babbage nunca había publicado una descripción seria del motor de diferencias y nunca había publicado nada sobre el motor analítico. Pero habló sobre la máquina analítica en Turín, y un tal Luigi Menabrea , que entonces era un ingeniero del ejército de 30 años, tomó notas , pero que, 27 años después, se convirtió en primer ministro de Italia (y también hizo contribuciones a las matemáticas del análisis estructural).

Actividades semana abril 12-16, 2021

Actividades individuales:

- 1) Leer detalladamente el artículo Translator (computing) de Wikipedia.
- 2) Leer detalladamente el artículo The Latin American Supercomputing Ecosystem for Science de la revista Communications de la ACM.

Importante: Cada estudiante debe actualizar sus respectivos apuntes personales de todos los temas y actividades semanales vistos hasta esta semana y compartir sus opiniones con su equipo de trabajo en las horas de sesión asíncrona.

Actividades en equipo:

- 3) Después de leer los artículos de manera individual, en equipo deben hacer un resumen de ambos artículos y grabar un video presentando su resumen (de mínimo tres y máximo 4 minutos dependiendo del número de integrantes), donde cada integrante, además de participar comentando su respectiva parte de la presentación, puede agregar información adicional relevante siempre y cuando mencione las referencias de dicha información.

Importante: El representante del equipo debe enviar por correo al profesor la liga de acceso al video de la actividad 3 junto con los nombres, números de cuenta y grupo de su equipo empleando su correo institucional. La fecha límite de entrega es el domingo 18 de abril a la media noche.

Resumen del tema: Translator (computing) de Wikipedia.

Traductor (informática)

Un traductor o procesador de lenguaje de programación es un término genérico que puede referirse a cualquier cosa que convierta código de un lenguaje de computadora a otro

Un programa escrito en un lenguaje de alto nivel se llama programa fuente. Estos incluyen traducciones entre lenguajes de computadora de alto nivel y legibles por humanos como C ++ y Java , lenguajes de nivel intermedio como el código de bytes de Java , lenguajes de bajo nivel como el lenguaje ensamblador y el código de máquina , y entre niveles similares de lenguaje en diferentes plataformas informáticas , así como de cualquiera de los anteriores a otro.

Compilador

Un compilador es un traductor que se utiliza para convertir un lenguaje de programación de alto nivel en un lenguaje de programación de bajo nivel. Convierte todo el programa en una sesión e informa de los errores detectados después de la conversión.

Necesita tiempo para hacer su trabajo, ya que traduce el código de alto nivel al código de nivel inferior de una vez y luego lo guarda en la memoria.

Depende del procesador y de la plataforma. Se ha abordado con nombres alternativos como los siguientes: compilador especial, compilador cruzado y compilador de fuente a fuente.

Intérprete

El intérprete es similar a un compilador, ya que es un traductor que se utiliza para convertir un lenguaje de programación de alto nivel en un lenguaje de programación de bajo nivel. La diferencia es que convierte el programa una línea de código a la vez y reporta errores cuando los detecta, mientras también realiza la conversión.

Un intérprete es más rápido que un compilador, ya que ejecuta el código inmediatamente después de leerlo. Se utiliza como herramienta de depuración para el desarrollo de software, ya que puede ejecutar una sola línea de código a la vez. Un intérprete es más portátil que un compilador, ya que es independiente del procesador, puede trabajar entre diferentes arquitecturas de hardware.

Ensamblador

Un ensamblador es un traductor utilizado para traducir el lenguaje ensamblador en lenguaje de máquina. Tiene la misma función que un compilador para el lenguaje ensamblador, pero funciona como intérprete. Es difícil de entender ya que es un lenguaje de programación de bajo nivel. Un ensamblador traduce un lenguaje de bajo nivel, como un lenguaje ensamblador, a un lenguaje de nivel aún más bajo, como el código máquina.

Resumen del Artículo The Latin American Supercomputing Ecosystem for Science

Las iniciativas de investigación grandes, costosas e intensivas en informática han promovido la informática de alto rendimiento (HPC) en los países más ricos como EE. UU., Japón y China. El impacto del internet y de la inteligencia artificial ha llevado a la HPC a un nuevo nivel aceptando las economías y sociedades de todo el mundo. En 1993, De América latina sólo México y Brasil aparecieron en el top 500 con supercomputadoras orientadas a la investigación. A junio de 2020 Brasil era el único representante de América latina.

Comparar la situación de HPC en tres países diferentes de América Latina ayuda a comprender las diferencias de la región, no sólo en términos de capacidad general,

sino también en términos de políticas adoptadas para crear y operar este tipo de sistemas de instrumentación científica. Los ejemplos presentados son representativos de otras iniciativas importantes en la región, por ejemplo, NLHPC en Chile, Tupac en Argentina, SC3UIS en Colombia y CeNAT en Costa Rica. En Brasil, el Laboratorio Nacional de Computación Científica (LNCC) es el actor principal de los servicios de HPC para la comunidad científica. LNCC alberga a Santos Dumont, que hasta junio de 2020 era la supercomputadora más grande dedicada a la investigación en América Latina.

Santos Dumont ha fomentado importantes colaboraciones internacionales, como en el diseño racional de candidatos a vacunas contra el Zika. En México, ABACUS, el Laboratorio de Matemática Aplicada y HPC del Centro de Investigación y Estudios Avanzados (CINVESTAV) ejemplifica las diversas iniciativas de HPC agrupadas en la Red Mexicana en HPC ABACUS alberga una de las principales supercomputadoras de investigación de América Latina, ubicada en el puesto 255 en la lista TOP500 de julio de 2015, con un rendimiento total actualizado de ~ 0,5 petaflops y una capacidad de almacenamiento de 1 petabyte. ABACUS ha colaborado con más de 140 proyectos de investigación y más de 250 artículos académicos. Ejemplos de proyectos son: simulación numérica de malformaciones vasculares en el cerebro; estudios de racemización de hélices moleculares; simulación numérica de peligros ambientales, entre otras.

En Uruguay, el Centro Nacional de Supercomputación (Cluster-UY) es una iniciativa para operar una infraestructura científica colaborativa de HPC para impulsar proyectos de investigación e innovación con altas demandas informáticas. Cluster-UY apoyó más de 50 proyectos de investigación que utilizaron más de 11 millones de horas, produjeron más de 250 artículos y



apoyaron 100 tesis de posgrado desde 2018. Se han desarrollado proyectos relevantes utilizando las capacidades informáticas de Cluster-UY, que incluyen: el desarrollo de herramientas de previsión para la gestión de energías renovables en Uruguay (Energía); 10 análisis socioeconómico de precios a corto y largo plazo y el impacto en el bienestar de los ciudadanos de bajos ingresos (Economía / Ciencias Sociales).

Actividades semana abril 19-23, 2021

Actividades individuales: 1) Leer las secciones 1.1 "Introduction" y 1.2 "Compiler Structure" del capítulo 1 del libro "Engineering a Compiler" de Cooper y Torczon (2012).

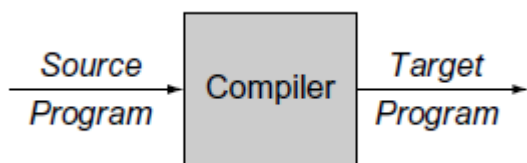
Actividades en equipo: 2) Revisar la sección "Getting Started / Tutorials" de la documentación oficial de LLVM.

1.1 Introducción

El papel de la computadora en la vida diaria crece cada año. Con el surgimiento de la Internet, las computadoras y el software que se ejecuta en ellas proporcionan comunicaciones, noticias, entretenimiento y seguridad. Las computadoras integradas han cambiado las formas en que construimos automóviles, aviones, teléfonos, televisores y radios. La computación ha creado categorías de actividad completamente nuevas, desde videojuegos a redes sociales. Las supercomputadoras predicen el clima diario y el curso de violentas tormentas. Las computadoras integradas sincronizan los semáforos y entregan el correo electrónico a su bolsillo.

Todas estas aplicaciones informáticas se basan en programas informáticos de software que construyen herramientas virtuales sobre las abstracciones de bajo nivel proporcionadas por el hardware subyacente. Casi todo ese software es traducido por una herramienta llamado compilador.

Un compilador es simplemente un programa de computadora que traduce otros programas de computador a otros programas informáticos para prepararlos para su ejecución. Este libro presenta las técnicas fundamentales de traducción automática que se utilizan.



Un compilador es una herramienta que traduce software escrito en un idioma a otro idioma. Para traducir texto

de un idioma a otro, la herramienta debe comprender tanto la forma o la sintaxis como el contenido o el significado del idioma de entrada. Necesita comprender las reglas que gobiernan la sintaxis y el significado en el idioma de salida. Finalmente, necesita un esquema para mapear contenido del idioma de origen al idioma de destino.

La estructura de un compilador típico se deriva de estas simples observaciones. El compilador tiene una interfaz para manejar el lenguaje fuente, tiene una estructura formal para representar el programa en un nivel intermedio, forma cuyo significado es en gran medida independiente de cualquier idioma a mejorar la traducción, un compilador a menudo incluye un optimizador que analiza y reescribe esa forma intermedia.

Descripción general

Los programas de computadora son simplemente secuencias de operaciones abstractas escritas en un lenguaje de programación: un lenguaje formal diseñado para expresar computación. Los lenguajes de programación tienen propiedades y significados rígidos, caso opuesto a los lenguajes naturales, como el chino o el portugués.

Los lenguajes de programación están diseñados para la expresividad, la concisión y la claridad. Los idiomas naturales permiten la ambigüedad. Los lenguajes de programación están diseñados para evitar ambigüedad; un programa ambiguo no

tiene sentido. Los lenguajes de programación están diseñados para especificar cálculos, para registrar la secuencia de acciones, realizar alguna tarea o producir algunos resultados.

Los lenguajes de programación están, en general, diseñados para permitir que los humanos expresen cálculos como secuencias de operaciones, desde los procesadores de computadora, en adelante denominados procesadores, microprocesadores o máquinas, están diseñados para ejecutar esas instrucciones.

Secuencias de operaciones

Las operaciones que implementan un procesador están, en su mayor parte, en un nivel de abstracción mucho más bajo que los especificados en un lenguaje de programación. Por ejemplo, un lenguaje de programación típicamente incluye una forma concisa de imprimir un número en un archivo, pero la declaración del lenguaje de programación debe traducirse literalmente a cientos de operaciones de la máquina antes de que pueda ejecutar.

Los principios fundamentales de la compilación

Los compiladores son objetos grandes, complejos y diseñados. Mientras que muchos de los problemas están en el diseño del compilador son susceptibles de múltiples soluciones e interpretaciones, hay dos principios fundamentales que un escritor de compiladores debe tener en cuenta en todo momento.

El primer principio es inviolable:

“El compilador debe preservar el significado del programa que se está compilando.”

La corrección es un tema fundamental en la programación. El compilador debe preservar la corrección implementando fielmente el "significado" de su entrada programa. Este principio se encuentra en el corazón del contrato social entre los escritores del compilador y usuario del compilador.

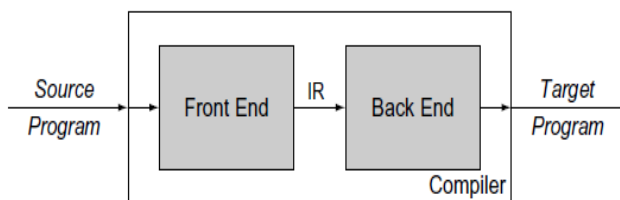
El segundo principio que debe observar un compilador es práctico:

"El compilador debe mejorar el programa de entrada de alguna manera discernible."

1.2 Estructura del compilador

Un compilador es un sistema de software grande y complejo. La comunidad ha creado compiladores de construcción desde 1955, y a lo largo de los años, hemos aprendido muchas lecciones sobre cómo estructurar un compilador. Anteriormente, describimos un compilador como un cuadro simple que traduce un programa fuente en un programa de destino. Pero en realidad, por supuesto, es más complejo que esa imagen simple.

Como sugiere el modelo de caja única, un compilador debe comprender el programa fuente que toma como entrada y mapea su funcionalidad al destino máquina. La naturaleza distinta de estas dos tareas sugiere una división del trabajo y conduce a un diseño que descompone la compilación en dos piezas principales: una front end y back end.

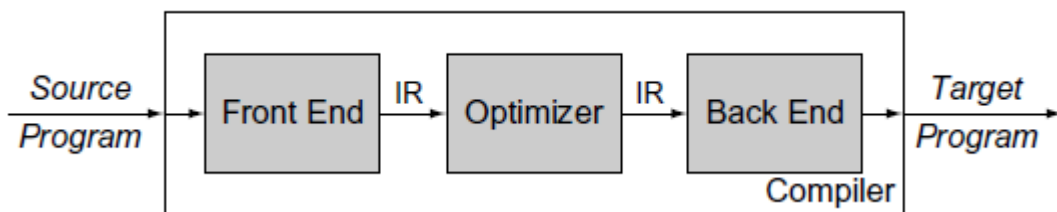


El front end se centra en comprender el programa en el idioma de origen. El back-end se centra en el mapeo de programas a la máquina. Esta separación de

preocupaciones tiene varias implicaciones importantes para el diseño e implementación de compiladores.

La introducción de un IR permite agregar más fases a la compilación. El escritor del compilador puede insertar una tercera fase entre el front-end y el back-end, hablamos del optimizador. Esta sección central, u optimizador, toma un programa como entrada y salida. La sección central de un compilador, también llamada optimizador, analiza y transforma el IR para mejorarlo.

Esta sección central, u optimizador, toma un programa de IR como entrada y produce un programa de IR semánticamente equivalente como salida. Al utilizar el IR como interfaz, el escritor del compilador puede insertar esta tercera fase con



un mínimo de interrupción en la parte del front-end y del back-end. Esto conduce a la siguiente estructura de compilador, denominada compilador de tres fases.

LLVM

Descripción general

El proyecto LLVM tiene múltiples componentes. El núcleo del proyecto en sí mismo se llama "LLVM". Contiene todas las herramientas, bibliotecas y archivos de encabezado necesarios para procesar representaciones intermedias y convertirlas en archivos de objeto. Las herramientas incluyen un ensamblador, un desensamblador, un analizador de código de bits y un optimizador de código de

bits. También contiene pruebas de regresión básicas. Los lenguajes similares a C usan la interfaz de Clang . Este componente compila código C, C ++, Objective C y Objective C ++ en código de bits LLVM y, desde allí, en archivos de objeto, utilizando LLVM.

Introducción al sistema LLVM con Microsoft Visual Studio

Descripción general

Para comenzar, primero se necesita conocer cierta información básica.

Hay muchos proyectos diferentes que componen LLVM. La primera pieza es la suite LLVM. Contiene todas las herramientas, bibliotecas y archivos de encabezado necesarios para usar LLVM. Contiene un ensamblador, desensamblador, analizador de código de bits y optimizador de código de bits. También contiene pruebas de regresión básicas que se pueden utilizar para probar las herramientas LLVM y la interfaz de Clang.

La segunda pieza es la parte delantera de Clang . Este componente compilar código C, C ++, Objective C y Objective C ++ en código de bits LLVM. Clang generalmente usa bibliotecas LLVM para optimizar el código de bits y emitir código de máquina. LLVM es totalmente compatible con el formato de archivo de objeto COFF, que es compatible con todas las demás cadenas de herramientas de Windows existentes.

La última parte importante de LLVM, la ejecución de Test Suite, no se ejecuta en Windows, y este documento no lo analiza. Puede encontrar información adicional sobre la estructura de directorios de LLVM y la cadena de herramientas en la página principal Introducción al sistema LLVM.

Requisitos

Antes de comenzar a utilizar el sistema LLVM, revise los requisitos que se indican a continuación. Esto puede ahorrarle algunos problemas al saber de antemano qué hardware y software necesitará.

Hardware

Cualquier sistema que pueda ejecutar adecuadamente Visual Studio 2017 está bien. El árbol de origen de LLVM y los archivos de objeto, las bibliotecas y los ejecutables consumirán aproximadamente 3 GB.

Software

Necesitará Visual Studio 2017 o superior, con la última actualización instalada.

También necesitará el sistema de compilación CMake , ya que genera los archivos del proyecto que utilizará para compilar.

Si desea ejecutar las pruebas LLVM, necesitará Python . Se sabe que la versión 3.6 y posteriores funcionan. También necesitará herramientas GnuWin32 . No instale el árbol de directorio LLVM en una ruta que contenga espacios (p . Ej.) Ya que el paso de configuración fallará.C:\Documents and Settings\...

Para simplificar el procedimiento de instalación, también puede utilizar Chocolatey como administrador de paquetes. Después de la instalación de Chocolatey, ejecute estos comandos en un shell de administración para instalar las herramientas necesarias:

- `choco install -y ninja git cmake gnuwin python3`
- `pip3 install psutil`

También hay un Dockerfile de Windows con toda la cadena de herramientas de compilación. Esto se puede usar para probar la compilación con una cadena de herramientas diferente a la instalación de su host o para crear servidores de compilación.

Actividades semana abril 26-30, 2021

Actividades individuales:

1) Leer la sección 1.3 "Overview of Translation" del capítulo 1 del libro "Engineering a Compiler" de Cooper y Torczon (2012).

Actividades en equipo:

2) Trabajar en la presentación de avance de proyecto final del concepto "Multi-Level Intermediate Representation (MLIR)" consultando las siguientes referencias: "MLIR" página oficial

1.3 Resumen de la traducción

Para traducir código escrito en un lenguaje de programación en código adecuado para ejecución en alguna máquina de destino, un compilador se ejecuta a través de muchos pasos.

NOTACIÓN

Los libros de compilación tratan, en esencia, de notación. Después de todo, un compilador traduce un programa escrito en una notación en un programa equivalente escrito en otra notación. En la lectura de este libro surgirán una serie de problemas de notación. En algunos casos, estos problemas afectarán directamente su comprensión del material.

Expresar algoritmos

Hemos tratado de mantener los algoritmos concisos. Los algoritmos se escriben a un nivel relativamente alto, asumiendo que el lector puede proporcionar detalles de implementación. Están escritos en un slanted, sans-serif font. La sangría es deliberada y significativa; importa más en una construcción si - entonces - si no.

Para que este proceso abstracto sea más concreto, considere los pasos necesarios para generar código ejecutable para la siguiente expresión:

$a \leftarrow a \times 2 \times b \times c \times d$

1.3.1 - El Front End

Antes de que el compilador pueda traducir una expresión en un código de máquina de destino ejecutable, debe comprender tanto su forma o sintaxis como su significado, o semántica. La interfaz determina si el código de entrada está bien formado, tanto en términos de sintaxis como de semántica. Si encuentra que el código es válido, crea una representación del código en la representación intermedia del compilador; si no, informa al usuario con mensajes de error de diagnóstico para identificar los problemas con el código.

Comprobación de la sintaxis

Para comprobar la sintaxis del programa de entrada, el compilador debe comparar la estructura del programa con una definición del lenguaje. Esto requiere una definición formal apropiada, un mecanismo eficiente para probar si la entrada cumple o no con esa definición, y un plan sobre cómo proceder con una entrada ilegal.

Las gramáticas del lenguaje de programación generalmente se refieren a palabras basadas en sus partes del habla, a veces llamadas categorías sintácticas. Basar las reglas gramaticales en partes del discurso permite que una sola regla describa muchas oraciones. Por ejemplo, en inglés, muchas oraciones tienen la forma:

- *Sentence* \rightarrow *Subject* verb *Object* endmark

donde verb y endmark son partes del discurso, y Sentence, Subject y Object son variables sintácticas. La oración representa cualquier cadena con la forma descrita por esta regla. El símbolo "-->" lee "deriva" y significa que una instancia del lado derecho puede abstraerse a la variable sintáctica del lado izquierdo.

Representaciones intermedias

El problema final que se maneja en la interfaz de un compilador es la generación de una forma de código. Los compiladores utilizan una variedad de diferentes tipos de IR, según el idioma de origen, el idioma de destino y las transformaciones específicas que aplica el compilador. Algunos irs representan el programa como un gráfico. Otros se asemejan a un programa de código ensamblador secuencial. El código en el margen muestra cómo podría verse nuestra expresión de ejemplo en un ir secuencial de bajo nivel. El capítulo 5 presenta una descripción general de la variedad de tipos de inicios que utilizan los compiladores.

1.3.2 El Optimizador

Cuando el front end emite ir para el programa de entrada, maneja las sentencias una por una, en el orden en que se encuentran. Por lo tanto, el programa ir inicial contiene estrategias de implementación generales que funcionarán en cualquier contexto circundante que el compilador pueda generar. En tiempo de ejecución, el código se ejecutará en un contexto más restringido y predecible. El optimizador analiza la forma del código para descubrir hechos sobre ese contexto y utiliza ese conocimiento contextual para reescribir el código de modo que calcule la misma respuesta de una manera más eficiente.

La eficiencia puede tener muchos significados. La noción clásica de optimización es reducir el tiempo de ejecución de la aplicación. En otros contextos, el optimizador podría intentar reducir el tamaño del código compilado u otras propiedades como la energía que consume el procesador al evaluar el código. Todas estas estrategias tienen como objetivo la eficiencia.

Análisis

La mayoría de las optimizaciones consisten en un análisis y una transformación. El análisis determina dónde el compilador puede aplicar la técnica de forma segura y rentable. Los compiladores utilizan varios tipos de análisis para respaldar las transformaciones. El análisis de flujo de datos se basa, en tiempo de compilación, en el flujo de valores en tiempo de ejecución. Los analizadores de flujo de datos normalmente resuelven un sistema de ecuaciones de conjuntos simultáneos que se derivan de la estructura del código que se está traduciendo.

Transformación

Para mejorar el código, el compilador debe ir más allá de analizarlo. El compilador debe utilizar los resultados del análisis para reescribir el código en un formato más eficiente. Se han inventado innumerables transformaciones para mejorar los requisitos de tiempo o espacio del código ejecutable. Algunos, como descubrir cálculos invariantes de bucle y moverlos a ejecutados con menos frecuencia ubicaciones, mejorar el tiempo de ejecución del programa. Otros hacen el código más compacto. Las transformaciones varían en su efecto, el alcance sobre el cual operan, y el análisis requerido para apoyarlos.

1.3.3 - El back-end

El back-end del compilador atraviesa la forma IR del código y emite código para la máquina de destino. Selecciona las operaciones de la máquina objetivo para implementar cada operación. Elige un orden en el que las operaciones se ejecutarán de manera eficiente. Decide qué valores residirán en los registros y qué valores residirán en la memoria e inserta código para hacer cumplir esas decisiones.

Selección de instrucción

La primera etapa de la generación de código reescribe las operaciones de infrarrojos en operaciones de máquina de destino, un proceso llamado selección de instrucciones. La selección de instrucciones mapea cada operación de infrarrojos, en su contexto, en una o más operaciones de máquina objetivo.

Asignación de registros

Durante la selección de instrucciones, el compilador ignoró deliberadamente el hecho de que la máquina de destino tiene un conjunto limitado de registros. En su lugar, utilizó registros virtuales y asumió que existían “suficientes” registros. En la práctica, las primeras etapas de compilación pueden crear más demanda de registros de la que puede soportar el hardware. El asignador de registros debe mapear esos registros virtuales en los registros reales de la máquina objetivo. Por lo tanto, el asignador de registros decide, en cada punto del código, cuyos valores deben residir en los registros de la máquina de destino. Luego reescribe el código para reflejar sus decisiones.

Programación de instrucciones

Para producir código que se ejecute rápidamente, el generador de código puede necesitar reordenar las operaciones para reflejar las restricciones de rendimiento específicas de la máquina de destino. El tiempo de ejecución de las diferentes operaciones puede variar. Las operaciones de acceso a la memoria pueden tomar decenas o cientos de ciclos, mientras que algunas operaciones aritméticas, particularmente la división, toman varios ciclos. El impacto de estas operaciones de latencia más prolongada en el rendimiento del código compilado puede ser dramático.

Interacciones entre componentes de generación de código

La mayoría de los problemas realmente difíciles que ocurren en la compilación surgen durante la generación de código. Para hacer las cosas más complejas, estos problemas interactúan.

Actividades semana mayo 3-7, 2021

Actividades individuales:

1) Leer las secciones 2.7, 3.7 y 4.6 "Chapter Summary and Perspective" de los capítulos 2,3 y 4 respectivamente del libro "Engineering a Compiler" de Cooper y Torczon (2012).

Importante: Cada estudiante debe actualizar sus respectivos apuntes personales de todos los temas y actividades semanales vistos hasta esta semana.

Actividades en equipo:

2) Trabajar en el video de presentación de avance del proyecto final del concepto "Multi-Level Intermediate Representation (MLIR)".

2.7 Resumen y perspectiva del capítulo

El uso generalizado de expresiones regulares para buscar y escanear es una de las historias de éxito de la informática moderna. Estas ideas se desarrollaron como una parte inicial de la teoría de los lenguajes formales y los autómatas. Se aplican de forma rutinaria en herramientas que van desde editores de texto hasta motores de filtrado web y compiladores como un medio para especificar de manera concisa grupos de cadenas que resultan ser lenguajes regulares. Siempre que se deba reconocer una colección finita de palabras, los reconocedores basados en dfa merecen una seria consideración.

La teoría de las expresiones regulares y los autómatas finitos ha desarrollado técnicas que permiten el reconocimiento de lenguajes regulares en un tiempo proporcional a la longitud del flujo de entrada. Las técnicas para la derivación automática de dfas a partir de res y para la minimización de dfa han permitido la construcción de herramientas robustas que generan reconocedores basados en DFA. Tanto los escáneres generados como los hechos a mano se utilizan en

compiladores modernos muy respetados. En cualquier caso, una implementación cuidadosa debe ejecutarse en un tiempo proporcional a la longitud del flujo de entrada, con una pequeña sobrecarga por carácter.

3.7 Resumen y perspectiva

Casi todos los compiladores contienen un analizador. Durante muchos años, el análisis sintáctico fue un tema de gran interés. Esto condujo al desarrollo de muchas técnicas diferentes para construir analizadores sintácticos eficientes. La familia de gramáticas LR (1) incluye todas las gramáticas libres de contexto que se pueden analizar de forma determinista. Las herramientas producen analizadores sintácticos eficientes con propiedades de detección de errores demostrablemente más sólidas. Esta combinación de características, junto con la amplia disponibilidad de generadores de analizadores sintácticos para las gramáticas LR (1), LALR (1) y SLR (1), ha disminuido el interés en otras técnicas de análisis automático como los analizadores de precedencia de operadores.

Los analizadores tienen su propio conjunto de ventajas. Podría decirse que son los analizadores sintácticos codificados a mano más fáciles de construir. Proporcionan excelentes oportunidades para detectar y reparar errores de sintaxis. Son eficientes; de hecho, un analizador sintáctico descendente recursivo de arriba hacia abajo bien construido puede ser más rápido que un analizador LR (1) basado en tablas. (El esquema de codificación directa para LR (1) puede superar esta ventaja de velocidad.) En un analizador sintáctico descendente recursivo de arriba hacia abajo, el escritor del compilador puede afinar más fácilmente las ambigüedades en el lenguaje fuente que podrían molestar a un analizador LR (1), como un analizador sintáctico LR (1) idioma en el que los nombres de palabras

clave pueden aparecer como identificadores. Un escritor de compiladores que quiera construir un analizador codificado a mano, por cualquier razón, es recomendable que utilice el método descendente recursivo de arriba hacia abajo. Al elegir entre las gramáticas LR (1) y LL (1), la elección se convierte en una de las opciones disponibles. herramientas. En la práctica, pocas construcciones de lenguaje de programación, si es que hay alguna, caen en la brecha entre las gramáticas LR (1) y las LL (1) gramáticas. Por lo tanto, comenzar con un generador de analizador disponible siempre es mejor que implementar un generador de analizador desde cero.

4.6 Resumen y perspectiva

En los Capítulos 2 y 3, vimos que gran parte del trabajo en la interfaz de un compilador puede automatizarse. Las expresiones regulares funcionan bien para el análisis léxico, mientras que las gramáticas libres de contexto funcionan bien para el análisis de sintaxis. En este capítulo, examinamos dos formas de realizar un análisis sensible al contexto: el formalismo gramatical de atributos y un enfoque ad hoc. Para el análisis sensible al contexto, a diferencia del escaneo y el análisis sintáctico, el formalismo no ha desplazado el enfoque ad hoc.

El enfoque formal, que utiliza gramáticas de atributos, ofrece la esperanza de escribir especificaciones de alto nivel que produzcan ejecutables razonablemente eficientes. problema en el análisis sensible al contexto, han encontrado aplicación en varios dominios, desde los probadores de teoremas hasta el análisis de programas. Para problemas en los que el flujo de atributos es principalmente local, las gramáticas de atributos funcionan bien. Los problemas que pueden formularse enteramente en términos de un tipo de atributo, ya sea heredado o sintetizado, a



menudo producen soluciones limpias e intuitivas cuando se presentan como gramáticas de atributos. Cuando el problema de dirigir el flujo de atributos alrededor del árbol con reglas de copia llega a dominar la gramática, probablemente sea el momento de salir del paradigma funcional de las gramáticas de atributos e introducir un repositorio central de hechos.

Prácticas y Actividades en clase

Actividades en clase

skp-rdiy-dyj - 28 Apr 2021

Establecer fondo | Borrar marco

Compartir

Abrir en un Jamboard

Cthulhu's Sons Team

```

if (x==j)
  z=0;
else
  z=1;

```

Giovanni se esta pirateando mi código xd

Muy bien equipo :3

Estructura de un token:
<tipo o clase, "cadena">

Tipos o clases de tokens:
W: espacio en blanco
K: palabra clave
I: identificador
N: número
O: otros tokens (,),[,>,<,,=;

Ejemplo: <K,"if">

Ketzali <3

```

<K,"if">
<W," ">
<O,"(">
<I,"x">
<O,"=">
<O,"=">
<I,"j">
<O,")">
<W," ">
<W," ">
<I,"z">
<O,"=">
<N,"0">
<O,";">
<W," ">
<K,"else">
<W," ">
<W," ">
<I,"z">
<O,"=">
<N,"1">
<O,";">

```

Miguel <3

```

<k,"if">
<w," ">
<o,"(">
<l,"x">
<o,"=">
<o,"=">
<l,"j">
<o,")">
<w," ">
<w," ">
<l,"z">
<o,"=">
<n,"0">
<o,";">
<w," ">
<k,"else">
<w," ">
<w," ">
<l,"z">
<o,"=">
<n,"1">
<o,";">

```

Jesus <3

```

<K,"if">
<W," ">
<O,"(">
<I,"x">
<O,"=">
<O,"=">
<I,"j">
<O,")">
<W," ">
<W," ">
<I,"z">
<O,"=">
<N,"0">
<O,";">
<W," ">
<K,"else">
<W," ">
<W," ">
<I,"z">
<O,"=">
<N,"1">
<O,";">

```

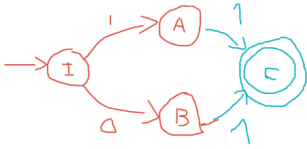
skp-rdiy-dyj - 26 May 2021

Establecer fondo | Borrar marco

Compartir

Abrir en un Jamboard

ER=(1+0)1



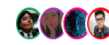
```

graph LR
    Start(( )) --> I((I))
    I -- 1 --> A((A))
    I -- 0 --> B((B))
    A -- 1 --> C(((C)))
    B -- 0 --> C
    style Start fill:none,stroke:none
    style C fill:none,stroke:none

```



skp-rdiy-dyj - 26 May 2021



Compartir



Establecer fondo

Borrar marco

Abrir en un Jamboard

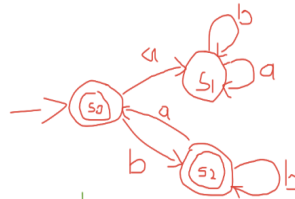
$$A = \{\Sigma, E, \delta, S, F\}$$

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

$$E = \{S_0, S_1, S_2\}$$

$$S = \{S_0\}$$

$$F = \{S_0, S_1\}$$



δ	a	b
S ₀	S ₁	S ₂
S ₁	S ₁	S ₁
S ₂	S ₀	S ₂

AFD

$$L = \{\epsilon, b, bb, bbb, \dots\}$$

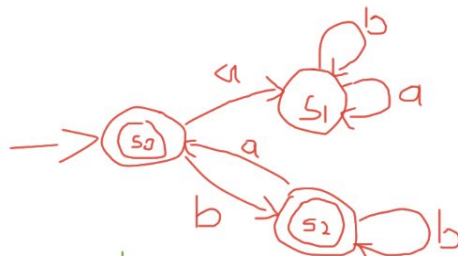
$$A = \{\Sigma, E, \delta, S, F\}$$

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

$$E = \{S_0, S_1, S_2\}$$

$$S = \{S_0\}$$

$$F = \{S_0, S_1\}$$



δ	a	b
S ₀	S ₁	S ₂
S ₁	S ₁	S ₁
S ₂	S ₀	S ₂

AFD

$$L = \{\epsilon, b, bb, bbb, \dots\}$$

Prácticas

Ejercicios - Expresiones Regulares - 12/Mayo/2021

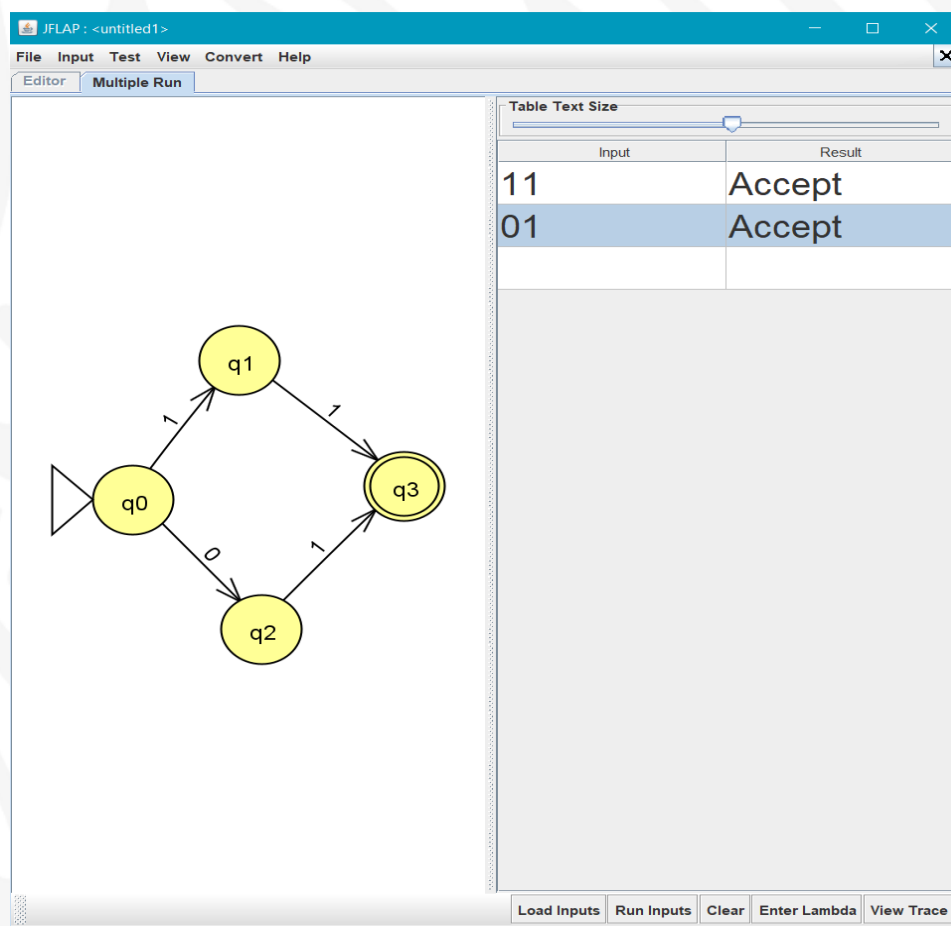
Ejercicios: Con el alfabeto $\Sigma = 0,1$

1) Con la expresión regular: $(1+0)1$ escribe 5 cadenas válidas (La expresión regular tiene 2 elementos)

Elemento 1 - $(1+0)$

Elemento 2 - (1)

R= No existen 5 cadenas válidas de esta expresión regular, las únicas cadenas válidas son: 11 y 01



2) Con la expresión regular: $(1+0)^*1$ escribe 5 cadenas válidas

· 1 · 01 · 0111 · 101101 · 101

JFLAP : <untitled1>

File Input Test View Convert Help

Editor Multiple Run

Table Text Size

Input	Result
1	Accept
01	Accept
0111	Accept
101101	Accept
101	Accept

Diagrama de Máquina de Estados Finitos (MEF):

```

graph LR
    start(( )) --> q0((q0))
    q0 -- 0 --> q0
    q0 -- 1 --> q1(((q1)))
  
```

Buttons: Load Inputs Run Inputs Clear Enter Lambda View Trace

3) $0^* + 1^*$

·0000 ·1111 ·000 ·111 ·11111

JFLAP : <untitled1>

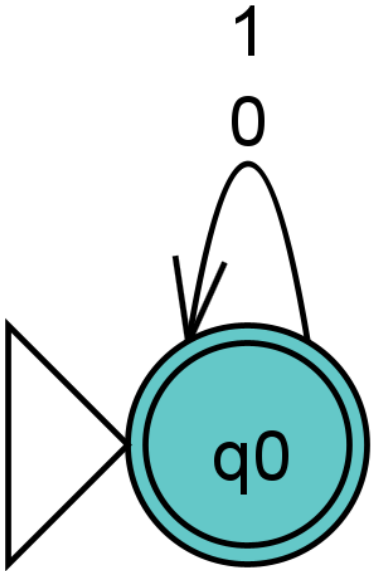
File Input Test View Convert Help

Editor Multiple Run

Table Text Size

Input	Result
0000	Accept
111	Accept
000	Accept
111	Accept
11111	Accept

Load Inputs Run Inputs Clear Enter Lambda View Trace



4) $(0^*1)0$

· 010 · 0010 · 00010 · 000010 · 0000010

JFLAP : <untitled1>

File Input Test View Convert Help

Editor Multiple Run

Table Text Size

Input	Result
010	Accept
0010	Accept
00010	Accept
000010	Accept
0000010	Accept

Diagram illustrating a finite automaton with three states: q0, q1, and q2. q0 is the start state, and q2 is the final state. Transitions are: q0 to q0 on input 0, q0 to q1 on input 1, and q1 to q2 on input 0.

Load Inputs Run Inputs Clear Enter Lambda View Trace

5) $1(1^*010^*)0^*$

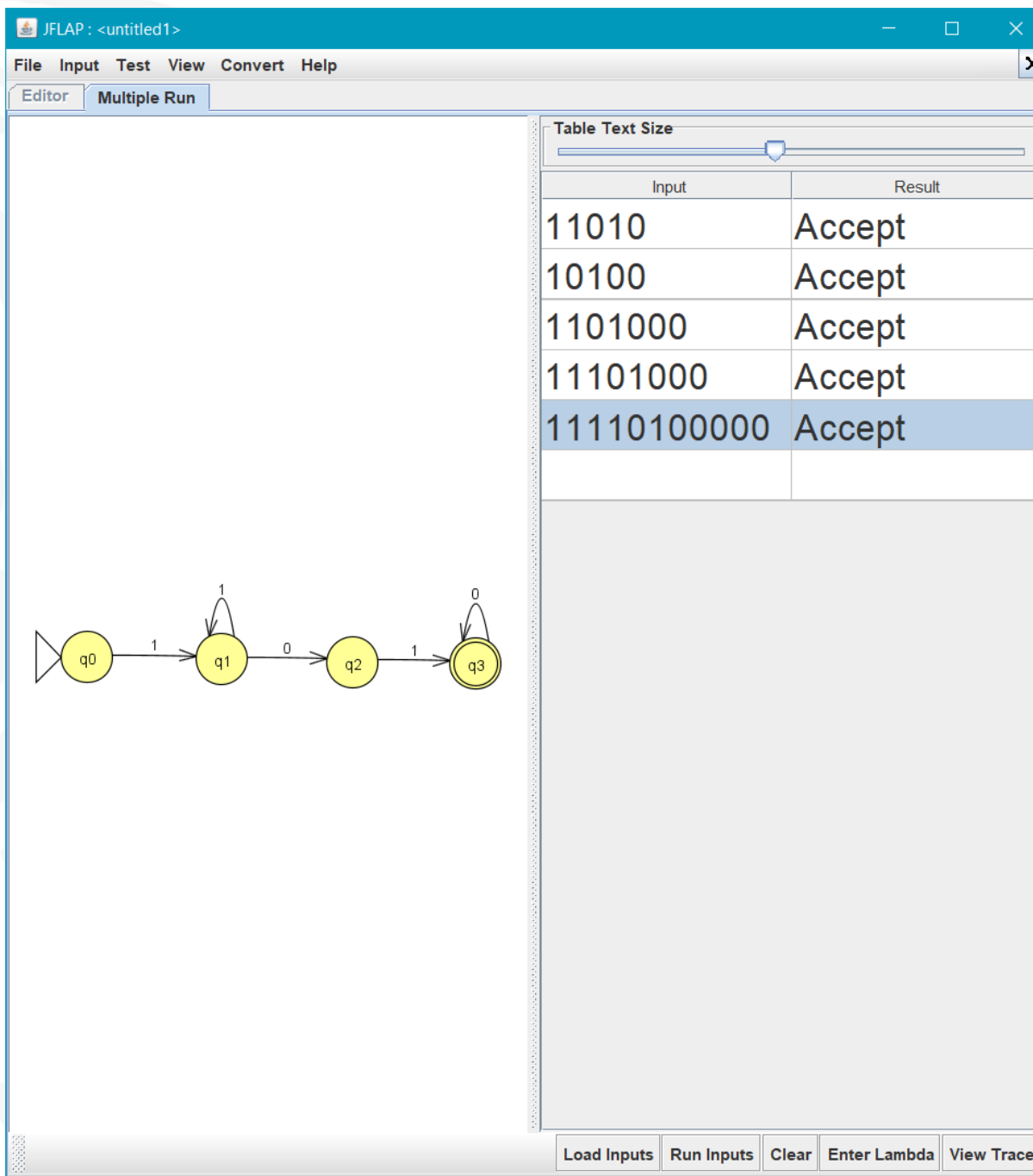
· 11010

· 10100

· 1101000

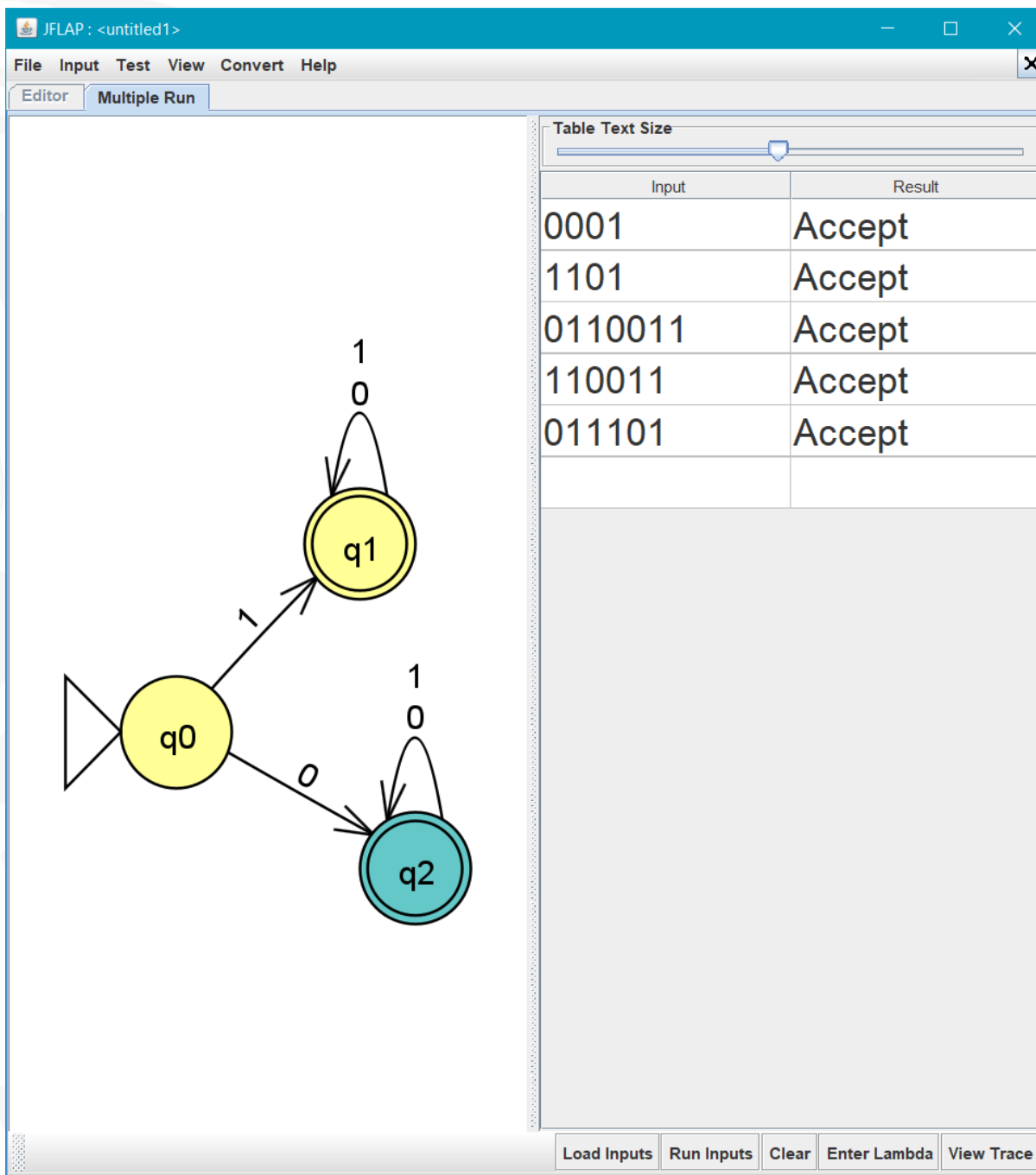
· 11101000

· 11110100000

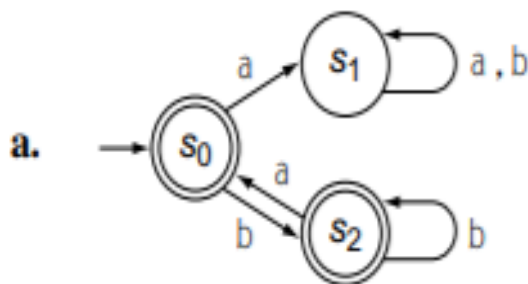


6) $(1+0)+(0+1)^*(0^*1^*)$

· 0001 · 1101 · 0110011 · 110011 · 011101



Ejercicio 1 de la página 80 del libro "Engineering a Compiler"



Solución propuesta:

$$S_0 = a \cdot S_1 + b \cdot S_2 + \lambda \quad (1)$$

$$S_1 = a \cdot S_1 + b \cdot S_1 \quad (2) \rightarrow a^* + b^*$$

$$S_2 = b \cdot S_2 + a \cdot S_0 + \lambda \quad (3)$$

$$(3) \quad S_2 = b^* + a \cdot S_0$$

$$(1) \quad S_0 = a \cdot (a^* + b^*) + b \cdot (b^* + a \cdot S_0) + \lambda$$

$$S_0 = a \cdot a^* + a \cdot b^* + b \cdot b^* + (b \cdot a) \cdot S_0 + \lambda$$

$$S_0 = a \cdot a^* + a \cdot b^* + b \cdot b^* + (b \cdot a)^*$$

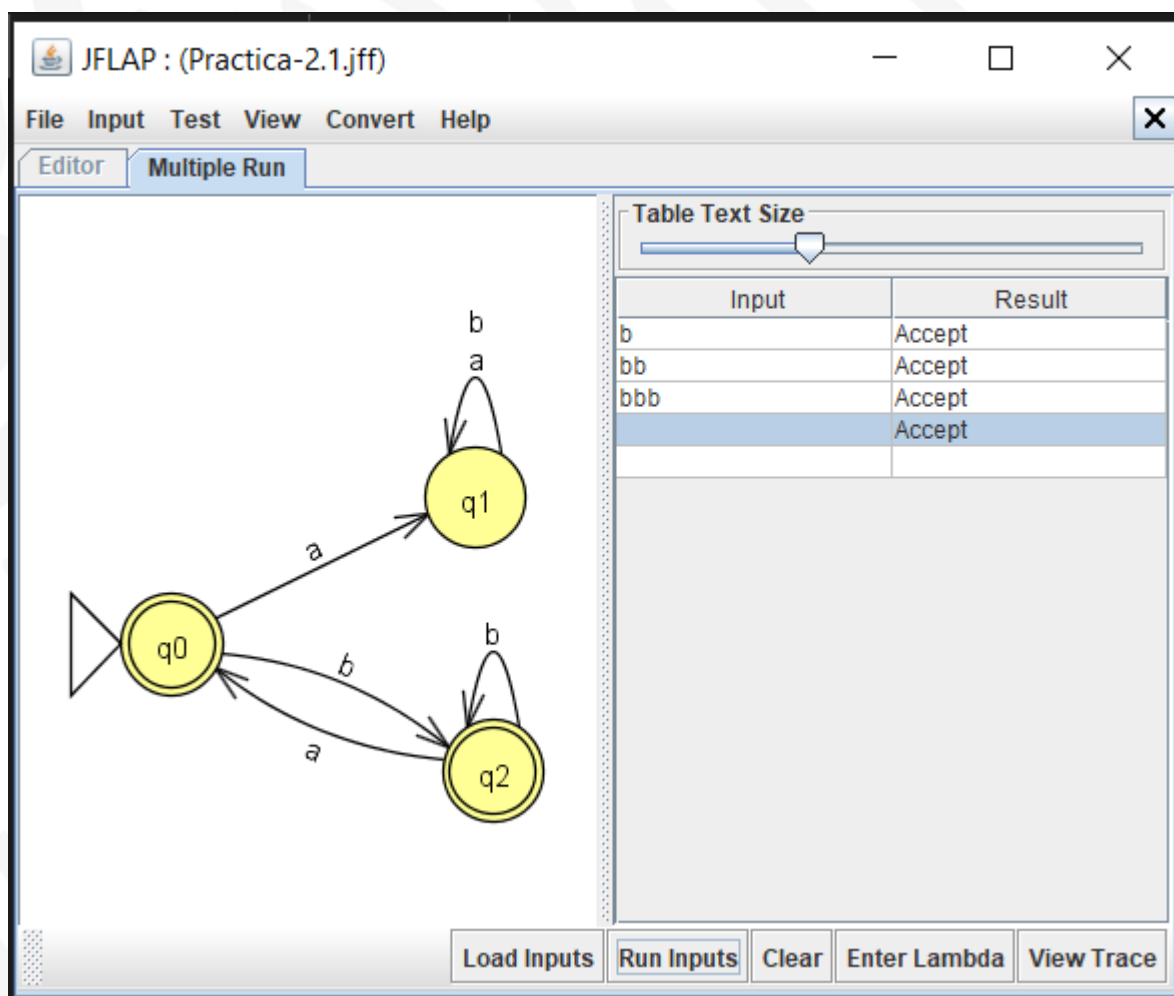
$$S_0 = a(a^* + b^*) + b^* + (b \cdot a)^*$$

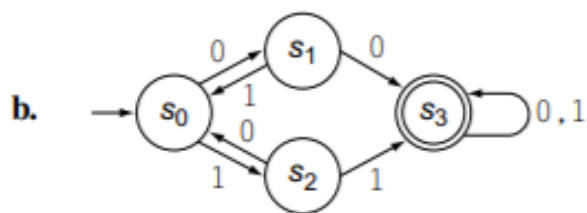
Solución correcta:

A)

S	a	b
S_0	S_1	S_2
S_1	S_1	S_1
S_2	S_0	S_2

$L = (\epsilon, b, bb, bbb, \dots)$





Solución propuesta:

State transition diagram (repeated from above):

$$s_0 = 0 \cdot s_1 + 1 \cdot s_2 \dots \dots (1)$$

$$s_1 = 1 \cdot s_0 + 0 \cdot s_3 \dots \dots (2)$$

$$s_2 = 0 \cdot s_0 + 1 \cdot s_3 \dots \dots (3)$$

$$s_3 = 0 \cdot s_3 + 1 \cdot s_3 \dots \dots (4)$$

(4) $s_3 = 0^* + 1^* + 1$

(1) $s_0 = 0 \cdot (1 \cdot s_0 + 0 \cdot s_3) + 1 \cdot (0 \cdot s_0 + 1 \cdot s_3)$

~~$s_0 = 0 \cdot 1 + 0 \cdot s_0 + 0 \cdot 0$~~

$$s_0 = 0 \cdot 1 + 0 \cdot s_0 + 0 \cdot 0 + 0 \cdot s_3 + 1 \cdot 0 + 1 \cdot s_0 + 1 \cdot 1 + 1 \cdot s_3$$

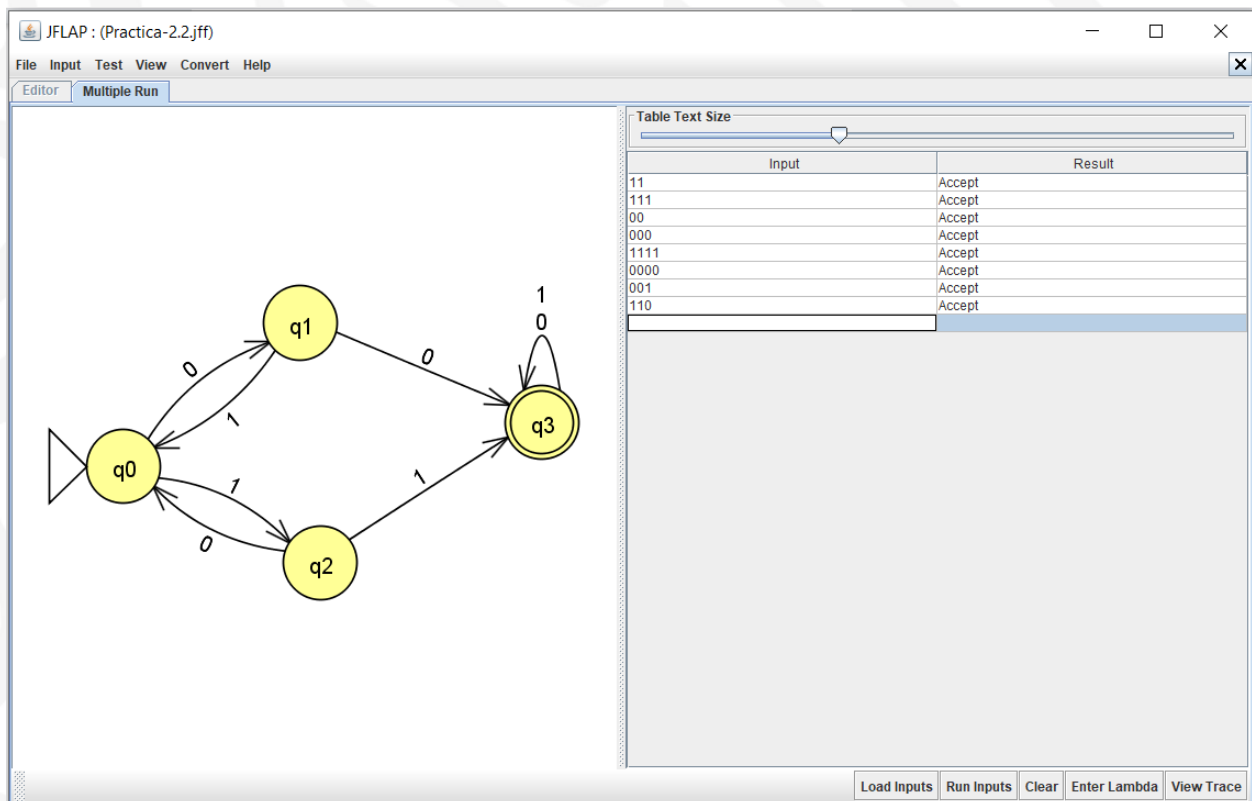
$$s_0 = 0 \cdot 1 + 0^* + 0 \cdot 0 + 0(0^* + 1^*) + 1 \cdot 0 + 1^* + 1 \cdot 1 + 1(0^* + 1^*)$$

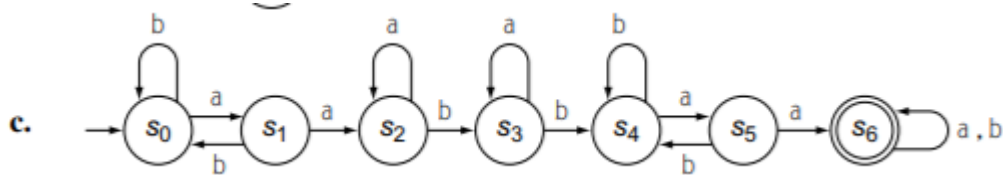
Solución correcta:

b)

S	0	1
S ₀	S ₁	S ₂
S ₁	S ₃	S ₀
S ₂	S ₀	S ₃
S ₃	S ₃	S ₃

$L = (\epsilon, 11, 111, 1111, \dots)$
 $L = (\epsilon, 00, 000, 0000, \dots)$





Solución propuesta:

$$\begin{aligned}
 S_0 &= b \cdot S_0 + a \cdot S_1 & \text{--- (1)} \\
 S_1 &= a \cdot S_2 + b \cdot S_0 & \text{--- (2)} \\
 S_2 &= a \cdot S_2 + b \cdot S_3 & \text{--- (3)} \\
 S_3 &= a \cdot S_3 + b \cdot S_4 & \text{--- (4)} \\
 S_4 &= b \cdot S_4 + a \cdot S_5 & \text{--- (5)} \\
 S_5 &= b \cdot S_4 + a \cdot S_6 & \text{--- (6)} \\
 S_6 &= a \cdot S_6 + b \cdot S_6 + \lambda & \text{--- (7)}
 \end{aligned}$$

(7) $S_6 = a^* + b^* + \lambda \rightarrow S_6 = a^* + b^*$
 (1) $S_0 = b^* + a \cdot (a \cdot S_2 + b \cdot S_0)$
 $S_0 = b^* + a \cdot a + a \cdot S_2 + a \cdot b + a \cdot S_0$
 $\Rightarrow S_0 = b^* + aa + a \cdot S_2 + ab + a \cdot S_0$
 (3) $S_2 = a \cdot S_2 + b \cdot S_3 \rightarrow S_2 = a^* + b \cdot S_3$
 (4) $S_3 = a \cdot S_3 + b \cdot S_4 \rightarrow S_3 = a^* + b \cdot S_4$
 (5) $S_4 = b \cdot S_4 + a \cdot S_5 \rightarrow S_4 = b^* + a \cdot S_5$
 (6) $S_5 = b \cdot S_4 + a \cdot S_6 \rightarrow S_5 = b(b^* + a \cdot S_5) + a \cdot (a^* + b^*)$
 $S_5 = b \cdot b^* + b \cdot a + b \cdot S_5 + a \cdot a^* + a \cdot b^*$
 $S_5 = bb^* + ba + b \cdot S_5 + aa^* + ab^*$

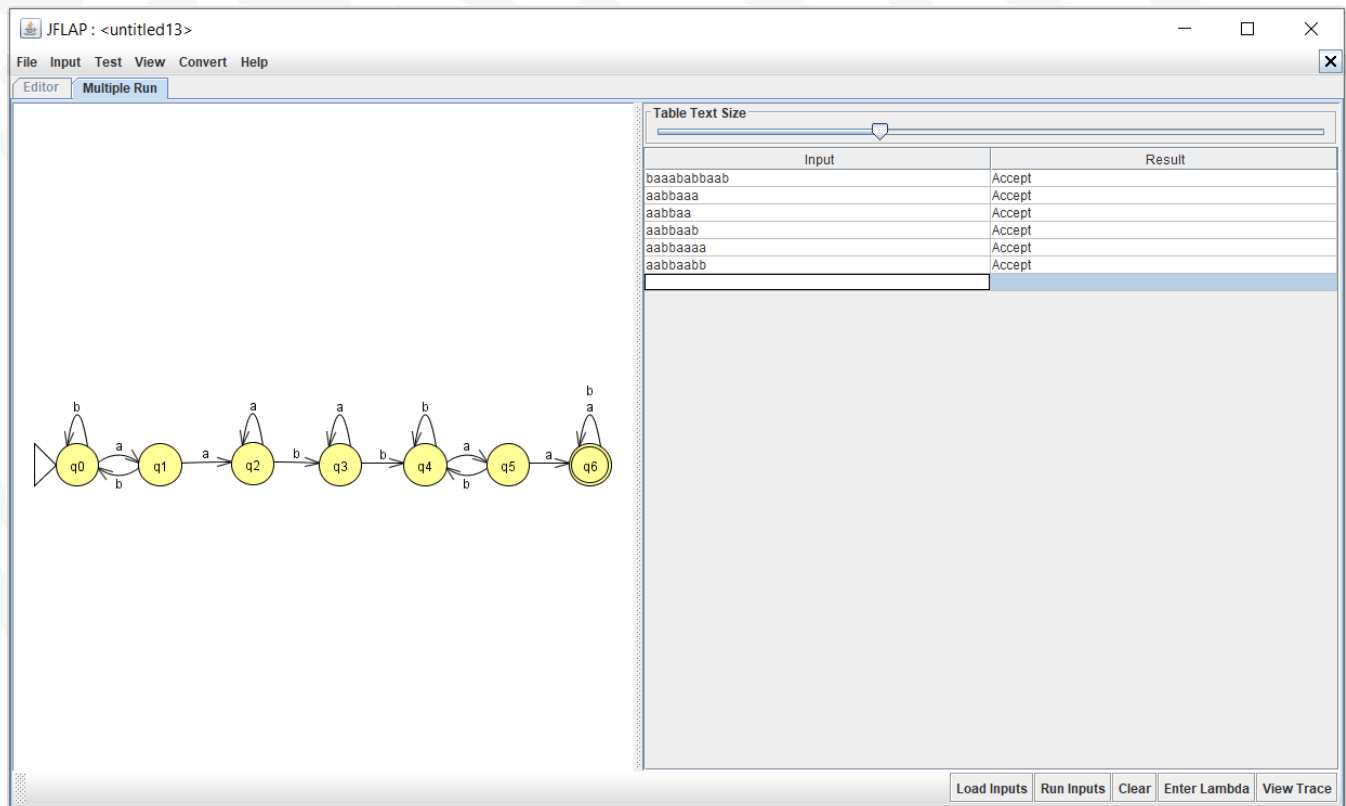
(5) $S_4 = b^* + a \cdot (bb^* + ba + b^* + aa^* + ab^*)$
 $S_4 = b^* + abb^* + aba + ab^* + aaa^* + aab^*$
 (4) $S_3 = a^* + b(b^* + abb^* + aba + ab^* + aaa^* + aab^*)$
 $S_3 = a^* + bb^* + babb^* + baba + bab^* + baaa^* + baab^*$
 (3) ?

Solución correcta:

c)

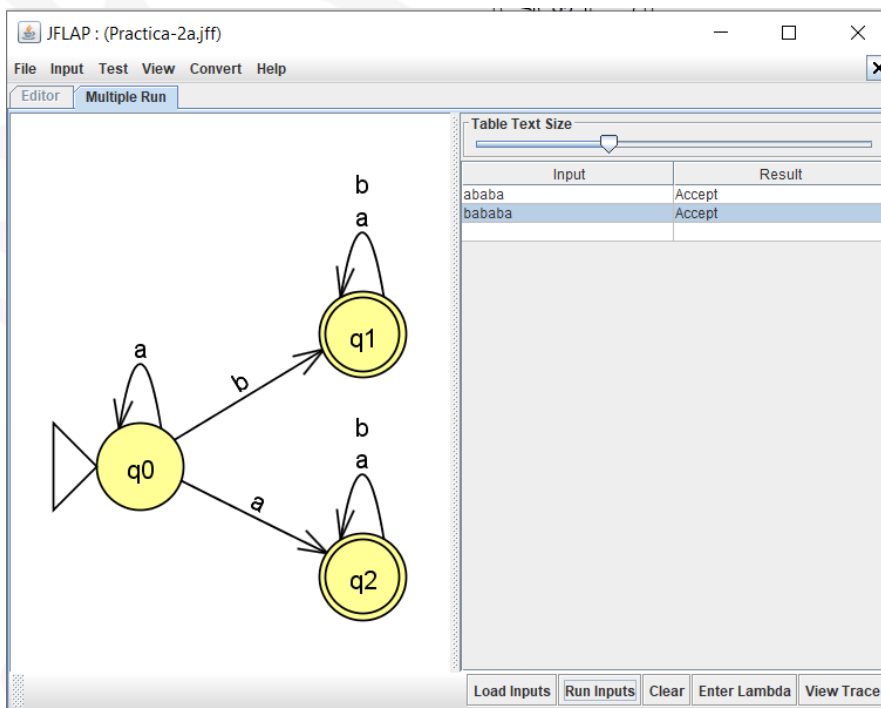
S	a	b
S ₀	S ₁	S ₀
S ₁	S ₂	S ₆
S ₂	S ₂	S ₃
S ₃	S ₃	S ₄
S ₄	S ₅	S ₄
S ₅	S ₆	S ₄
S ₆	S ₆	S ₆

$L = (\epsilon, aabbaa, aabbaaa, aabbaaaa, \dots)$

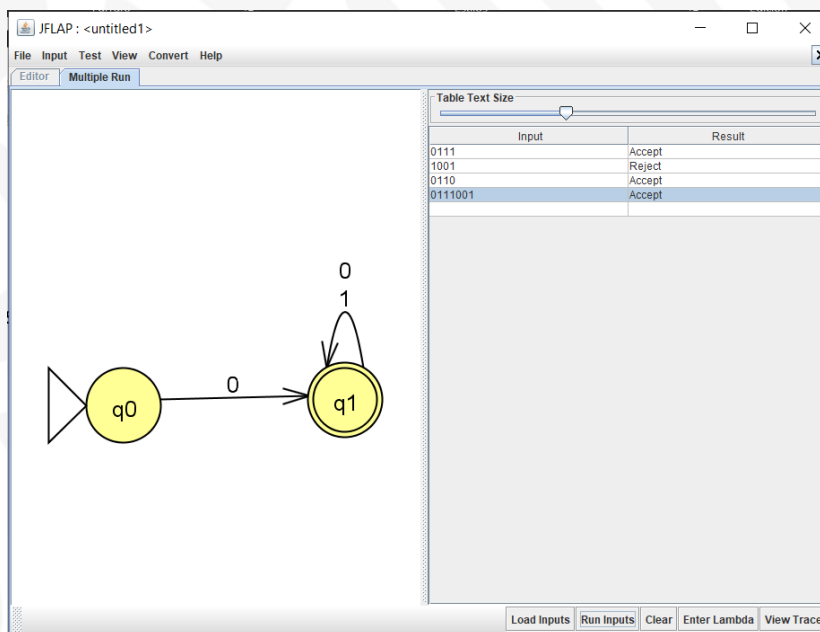


2. Construct an FA accepting each of the following languages:
 - a. $\{w \in \{a, b\}^* \mid w \text{ starts with 'a' and contains 'baba' as a substring}\}$
 - b. $\{w \in \{0, 1\}^* \mid w \text{ contains '111' as a substring and does not contain '00' as a substring}\}$
 - c. $\{w \in \{a, b, c\}^* \mid \text{in } w \text{ the number of 'a's modulo 2 is equal to the number of 'b's modulo 3}\}$

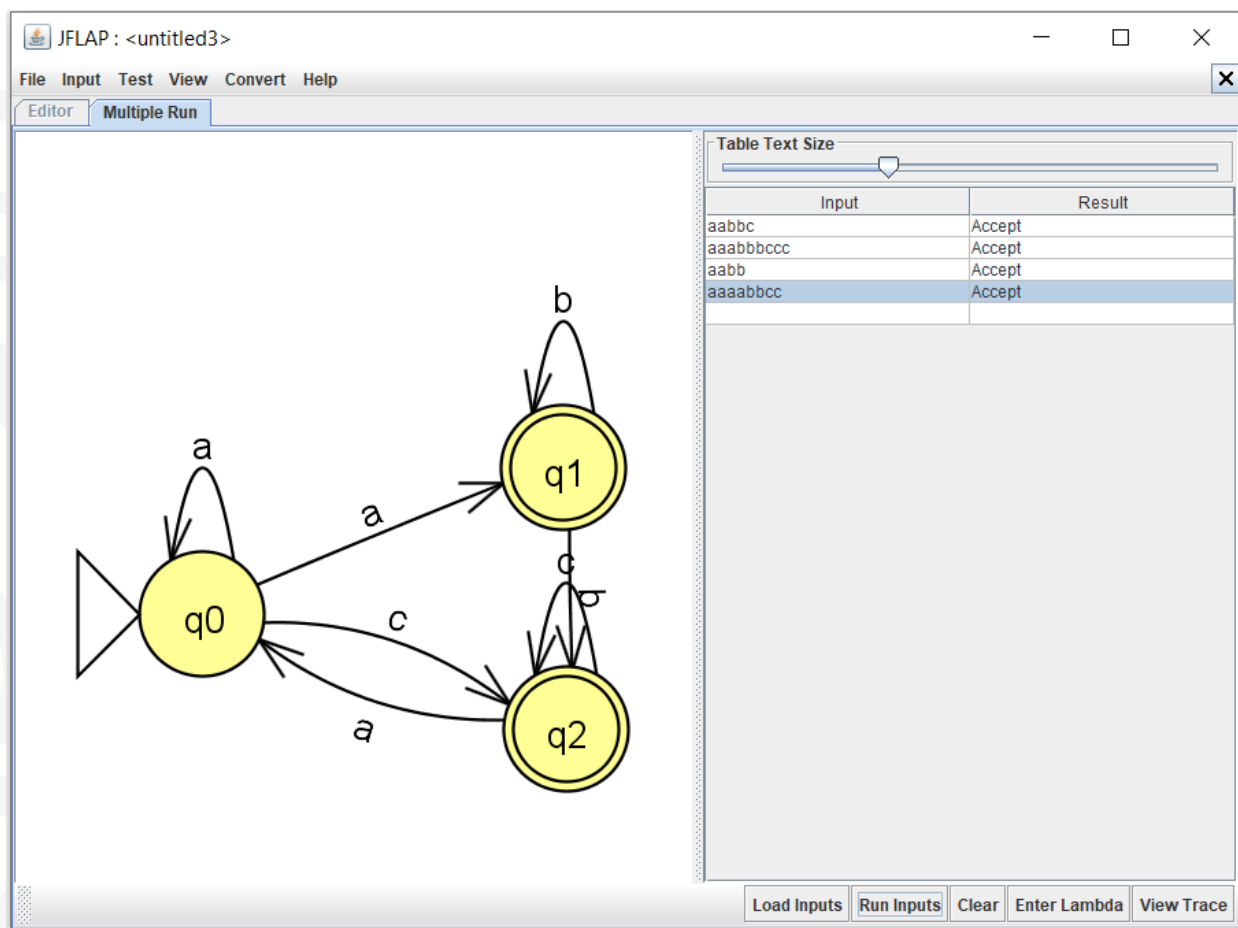
a) Solución propuesta



b) Solución propuesta



c) Solución propuesta



Bibliografía

Aho, A., Lam, M., Sethi, R., & Ullman, J. (1986). *Compilers - Principles, Techniques, & Tools* (Segunda ed.). PEARSON.

Cooper, K., & Torczon, L. (2012). *ENGINEERING A COMPILER* (Segunda ed.). ELSEVIER.

Gitler, I., Gomes, A., & Nesmachnow, S. (2020, 11). *The Latin American Supercomputing Ecosystem for Science*. COMMUNICATIONS OF THE ACM. Recuperado en abril 14, 2021, de <https://cacm.acm.org/magazines/2020/11/248214-the-latin-american-supercomputing-ecosystem-for-science/fulltext>

Grune, D., van Reeuwijk, K., Bal, H., Jacobs, C., & Langendoen, K. (2012). *Modern Compiler Design* (Segunda ed.). Springer.

Jalife-Rahme, A. (2021, 01 16). *Golpe cibernético' global: del 11 de septiembre al 6 de enero*. SPUTNIC Mundo. Recuperado en marzo 10, 2021, de <https://mundo.sputniknews.com/20210116/golpe-cibernetico-global-del-11-de-septiembre-al-6-de-enero-1094134378.html>

LLVM. (12, 01 01). *Introducción/Tutoriales LLVM*. LLVM. Recuperado en abril 19, 2021, de <https://llvm.org/docs/GettingStartedTutorials.html>

LLVM. (12, 01 01). *The LLVM Compiler Infrastructure*. LLVM. Recuperado en abril 19, 2021, de <https://llvm.org/>

MLIR. (n.d.). *Multi-Level Intermediate Representation*. Multi-Level IR Compiler Framework. Recuperado en junio 02, 2021, de <https://mlir.llvm.org/>

Wikipedia. (2021, 01 25). *Translator (computing)*. WIKIPEDIA. Recuperado en abril 12, 2021, de [https://en.wikipedia.org/wiki/Translator_\(computing\)](https://en.wikipedia.org/wiki/Translator_(computing))

Wolfram, S. (15, 12 10). *Untangling the Tale of Ada Lovelace*. STEPHEN WOLFRAM Writings. Recuperado en marzo 17, 2021, de <https://writings.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/>