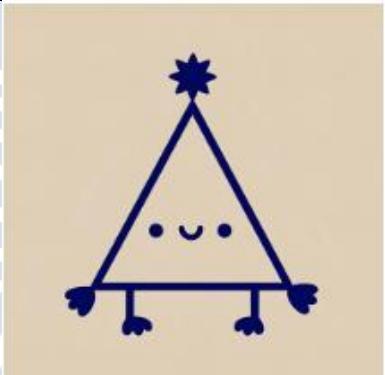


Práctica de laboratorio

CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERIA EN SISTEMAS COMPUTACIONALES	Lenguajes y autómatas 2.
INTEGRANTES-EQUIPO 04	
• Casteñeda Pérez Cristian Eduardo	21030140
• Macías Sevilla Diana Nathasha	21030223

PRACTICA NO.	NOMBRE DE LA PRACTICA	DURACIÓN (HORAS)
Tarea 6	Avance de proyecto	2 hora(max)

INTRODUCCIÓN



CAN es un robot que se desplaza en un mundo bidimensional compuesto por calles y avenidas, formando una cuadrícula. Solo puede moverse hacia adelante, girar 90° a la izquierda y manipular objetos llamados zumbadores.

El entorno de CAN puede contener muros que le impiden el paso y zumbadores que puede recoger o dejar en una casilla específica. Para interactuar con este mundo, CAN recibe instrucciones a través de un lenguaje de programación estructurado que permite controlar su comportamiento.

Las principales acciones que CAN puede realizar son:

- Avanzar: Se mueve una casilla en la dirección en la que está orientado.
- Girar a la izquierda: Rota 90° en sentido antihorario.
- Recoger un zumbador: Si hay un zumbador en la casilla donde está ubicado, lo recoge.
- Dejar un zumbador: Si tiene zumbadores en su mochila, puede colocar uno en la casilla actual.
- Verificar condiciones: Puede saber si hay un zumbador en su casilla, si su mochila está vacía o si hay un muro frente a él.

Para programar a CAN, se pueden usar estructuras de control como:

- Ciclos (repetir, mientras): Para repetir instrucciones un número determinado de veces o mientras se cumpla una condición.
- Condicionales (si, si-sino): Para que CAN tome decisiones según su entorno.
- Funciones: Permiten organizar el código en bloques reutilizables.

Práctica de laboratorio

A través de estos conceptos, CAN puede resolver problemas como recorrer laberintos, recolectar zumbadores en patrones específicos o construir figuras dentro de su mundo.



OBJETIVO

Creación de un compilador de un lenguaje de programación con simulación (parecido a Karel)

Objetivos específicos.

- Creación de una página web que contenga el siguiente menú, acerca de nosotros, documentación y la pagina principal para realizar la compilación
- Crear un análisis léxico del lenguaje regular, con la utilidad de distintos autómatas
- Crear un análisis sintáctico dado el resultado de los tokens
- Crear un análisis semántico
- Ejecución de las instrucciones de karel

FUNDAMENTO

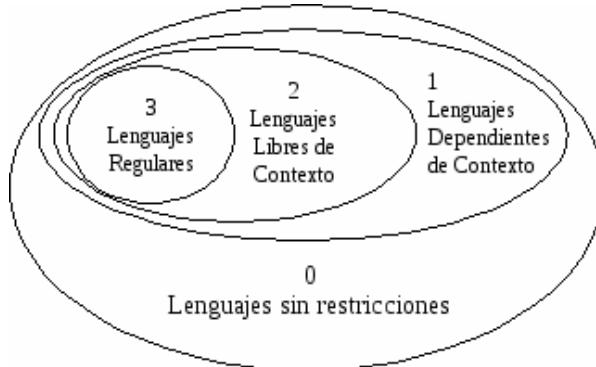
Un compilador es un programa que traduce un programa escrito en lenguaje fuente y produce otro equivalente escrito en un lenguaje destino.

La gramática de Chomsky, o gramática generativa transformacional, es una teoría lingüística propuesta por Noam Chomsky que busca explicar cómo los humanos aprenden y usan el lenguaje. Esta teoría postula que existe una estructura gramatical subyacente y universal en todas las lenguas, y que los humanos tienen una capacidad innata para aprender y usar esta estructura.

Práctica de laboratorio

El punto clave de la gramática de Chomsky es la idea de la competencia lingüística innata. Chomsky argumenta que los humanos no aprenden el lenguaje simplemente a través de la exposición a ejemplos, sino que nacen con una capacidad innata para adquirir y usar la gramática de cualquier lenguaje. Esta capacidad innata se conoce como la competencia lingüística, y se manifiesta en la habilidad de generar oraciones gramaticalmente correctas y entender estructura subyacente del lenguaje.

La Jerarquía de Chomsky es un sistema clasificación de lenguajes formales según su capacidad generativa. Esta jerarquía, propuesta por Noam Chomsky, clasifica las gramáticas en cuatro tipos: tipo 0 (irrestrictos), tipo 1 (dependientes de contexto), tipo 2 (libres de contexto) y tipo 3 (regulares).



la
de

- Tipo 0 (Irrestrictos): Estos lenguajes pueden generar cualquier cadena de símbolos usando una gramática sin restricciones en sus reglas.
- Tipo 1 (Dependientes de Contexto): Estos lenguajes tienen gramáticas que generan cadenas basadas en el contexto de los símbolos, donde cada regla tiene la forma " $\alpha \rightarrow \beta$ " con $\alpha \neq \epsilon$ (cadena vacía) y $\beta \neq \epsilon$, y β es una cadena de símbolos.

3 (Regulares): Estos lenguajes tienen gramáticas que generan cadenas de forma más simple, usando reglas de la forma " $A \rightarrow aB$ " o " $A \rightarrow a$ ", donde A y B son símbolos no terminales, y 'a' es un símbolo terminal.

La gramática de Chomsky no se presenta "en forma de cebolla" en un sentido literal. La forma en que se presenta la gramática de Chomsky es más bien una estructura jerárquica de reglas gramaticales, que se organizan en capas que van desde la estructura profunda (o semántica) a la estructura superficial (o sintáctica). Esta estructura se puede visualizar como una cebolla en capas, donde cada capa representa un nivel de análisis gramatical.

Los compiladores utilizan gramáticas para definir la sintaxis del lenguaje que están procesando. La jerarquía de Chomsky ayuda a clasificar las gramáticas y entender su relación con los lenguajes. Por ejemplo, las gramáticas libres de contexto (tipo 2) son muy comunes en la definición de la sintaxis de lenguajes de programación y se pueden analizar utilizando algoritmos como el análisis descendente o el análisis ascendente.

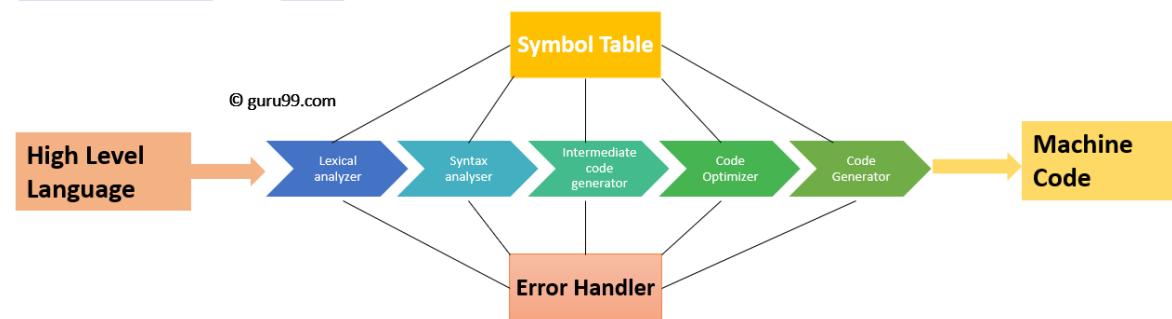
Las implicaciones de la jerarquía de Chomsky para el desarrollo de la computación han sido realmente extraordinarias. La jerarquía permitió que la construcción de lenguajes de

Práctica de laboratorio

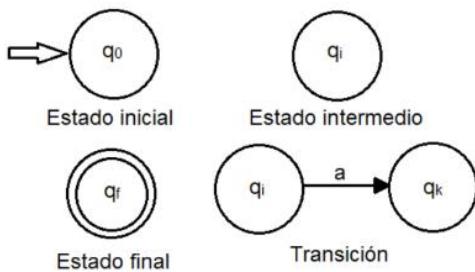
programación con propiedades matemáticas conocidas, compiladores e intérpretes, en particular, los lenguajes de programación tendían a utilizar dos tipos de autómatas: los autómatas de estado finito para llevar a cabo el reconocimiento léxico (la formación de tokens) propios del lenguaje de programación y los autómatas de pila como analizadores sintácticos, es decir, para analizar las expresiones bien formadas del lenguaje.

Compilador funciona en varias fases, cada una de las cuales transforma el programa fuente de una representación a otra. Cada fase toma las entradas de la etapa anterior y envía su salida a la siguiente fase del compilador.

Hay 6 fases en un compilador. Cada una de estas fases ayuda a convertir el lenguaje de alto nivel en código de máquina



Un autómata es una construcción lógica que recibe una entrada y produce una salida en función de todo lo recibido hasta ese instante. En el caso de los procesadores de lenguaje un autómata es una construcción lógica que recibe como entrada una cadena de símbolos y produce una salida indicando si dicha cadena pertenece o no a un determinado lenguaje.



Representación de un autómata

- Donde q_0, q_i, q_k, q_f son estados y se representan con un círculo
- a es un símbolo de entrada y pertenece a un alfabeto.

Un autómata finito determinista (AFD) es un modelo matemático que analiza cadenas de caracteres y decide si las acepta o rechaza. Se compone de un conjunto de estados, un estado inicial, un alfabeto, y una función de transición.

Los AFD se construyen con base en:

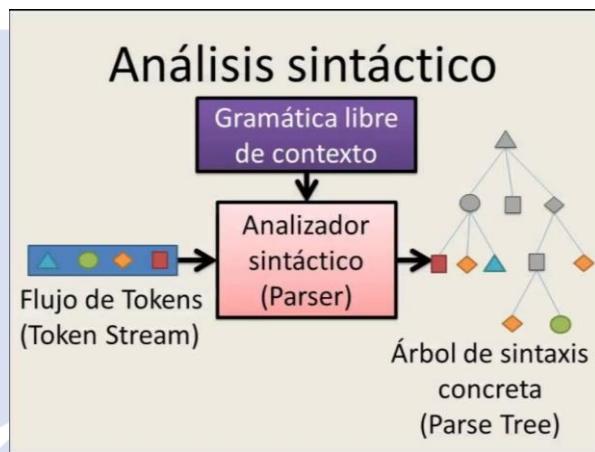
Práctica de laboratorio

- Un conjunto finito de estados, Q
- Un conjunto finito de símbolos de entrada, Σ (sigma)
- Una función de transición, δ o Δ (delta), que toma un estado y un símbolo de entrada y devuelve un estado
- Un estado inicial, uno de los estados de Q

Un conjunto de estados finales o de aceptación, F

El AFD acepta una cadena si, al finalizar de procesarla, el autómata queda en uno de los estados finales de aceptación.

Los AFD son un caso particular de los autómatas finitos no deterministas (AFND).



ANALISIS SINTACTICO

El análisis sintáctico es el proceso de analizar e interpretar datos para comprender su estructura y significado. Esto puede hacerse con distintos tipos de datos, como texto, código u otros formatos de datos estructurados.

En términos más sencillos, el análisis sintáctico consiste en descomponer una información compleja en partes más pequeñas y comprensibles. Es como tomar una frase y descomponerla en sus palabras individuales y componentes gramaticales, para poder entender lo que dice.

El análisis sintáctico se utiliza habitualmente en informática para analizar e interpretar código, como HTML o JavaScript, y garantizar que esté correctamente formateado y estructurado. Por ejemplo, un navegador web utiliza el análisis sintáctico para leer y comprender el código HTML y presentarlo como una página web.

El análisis sintáctico también puede utilizarse en el procesamiento del lenguaje natural, donde se emplea para analizar y comprender el lenguaje escrito o hablado. En este caso, el análisis sintáctico consiste en descomponer una frase u oración en sus componentes gramaticales, como sustantivos, verbos y adjetivos, para comprender su significado.

En general, el análisis sintáctico es un concepto fundamental en informática y otros campos relacionados con el análisis y la interpretación de datos. Consiste en descomponer datos complejos en partes más pequeñas y comprensibles para poder analizarlos e interpretarlos correctamente.

Práctica de laboratorio



- SvelteKit
- Dependencias de taiwaland
- TypeScript
- Conocimientos de compiladores

DESARROLLO



Análisis léxico

El análisis léxico es la primera fase en el proceso de compilación de un programa escrito en el lenguaje Karel. Su objetivo es leer el código fuente como una secuencia de caracteres y agruparlos en unidades significativas llamadas tokens, que representan palabras clave, identificadores, números, operadores, y otros símbolos del lenguaje.

Para esta tarea, el análisis léxico emplea una gramática regular, la cual corresponde al Tipo 3 de la jerarquía de Chomsky. Las gramáticas regulares permiten describir de forma precisa los patrones simples y repetitivos presentes en los tokens del lenguaje. Esta gramática puede definirse mediante expresiones regulares, las cuales son ampliamente utilizadas en herramientas de generación de analizadores léxicos como Flex, Lex, o JLex.

El análisis léxico de Karel utiliza una gramática de Tipo 3 (regular) porque:

- Cada token se escribirá mediante una expresión regular.
- Las producciones tienen la forma $A \rightarrow aB$ o $A \rightarrow a$, donde A y B son no terminales y a es un terminal.
- Esta gramática puede representarse mediante autómatas finitos deterministas (DFA) o no deterministas (NFA), lo cual permite implementar analizadores léxicos eficientes.

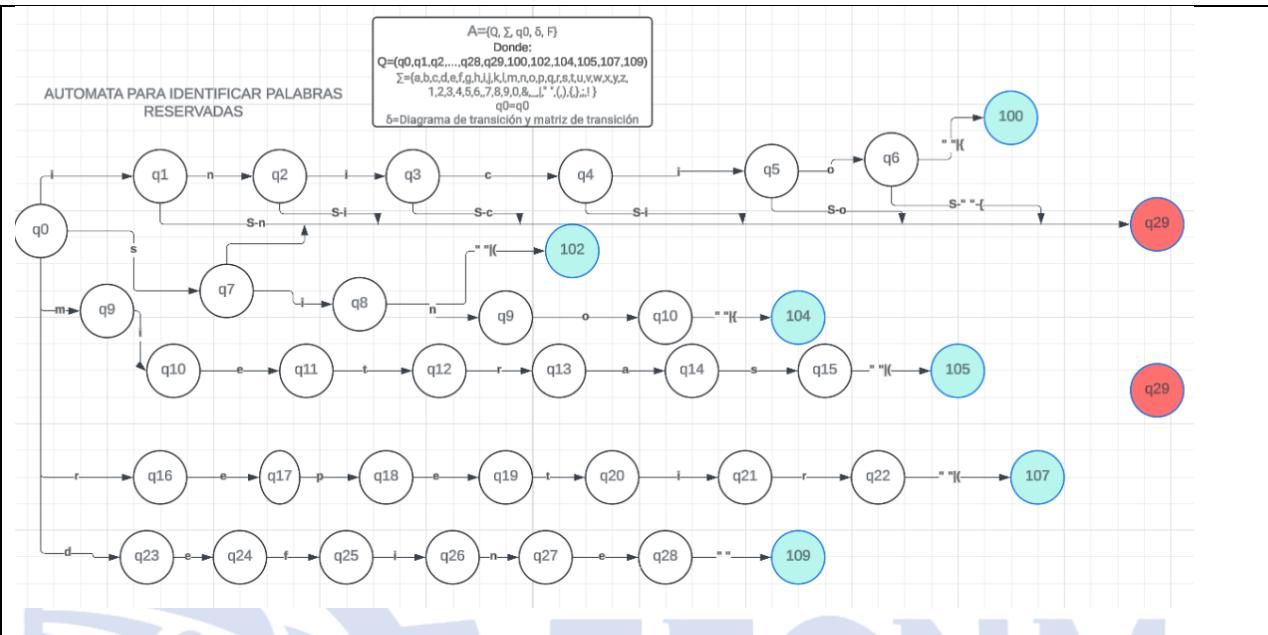
Lenguaje de programación

Palabras reservadas

1. Sentencias de Control

#	Palabra Reservada	Token (Número)	Descripción
1	Inicio	TK_INICIA (1000)	Indica el inicio del programa.
3	Si	TK_SI (1002)	Inicia una estructura condicional.
5	Sino	TK_SINO (1004)	Indica la acción en caso contrario.
6	Mientras	TK_MIENTRAS (1005)	Inicia un ciclo while.
8	Repetir	TK_REPETIR (1007)	Inicia un ciclo de repetición (for).
10	define	TK_DEFINE (1009)	Permite definir nuevas funciones.

Práctica de laboratorio



2. Condiciones

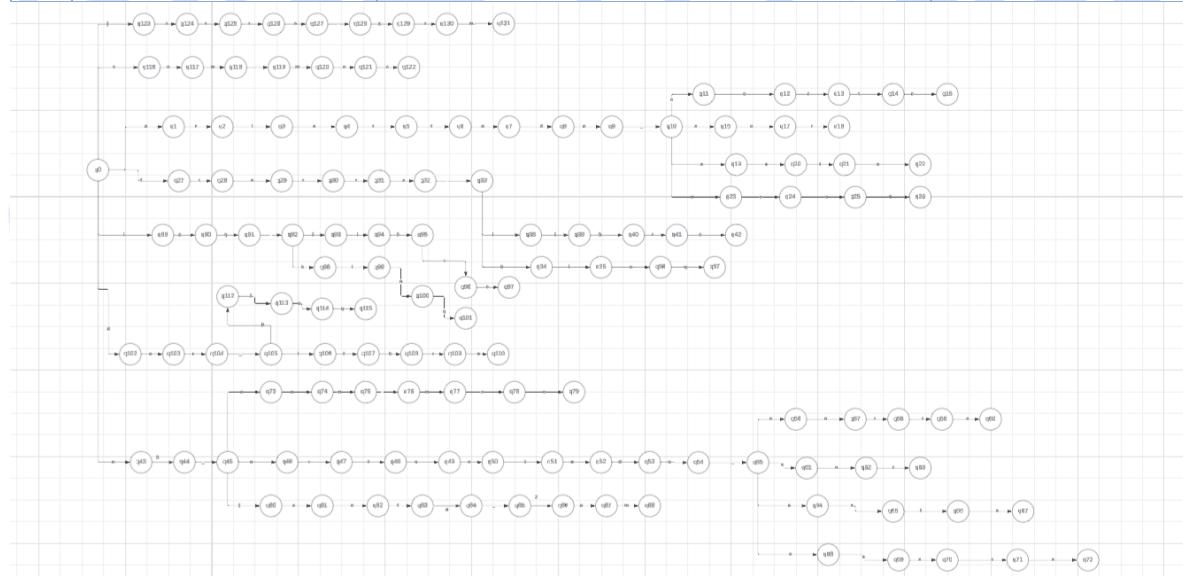
#	Palabra Reservada	Token (Número)	Descripción
1	frente_libre	TK_FRENTES_LIBRE (2000)	Verifica si hay espacio libre adelante.
2	frente_bloq	TK_FRENTES_BLOQUEADO (2010)	Verifica si hay un obstáculo adelante.
3	izq_libre	TK_IZQUIERDA_LIBRE (2020)	Verifica si CAN puede moverse a la izquierda.
4	izq_bloq	TK_IZQUIERDA_BLOQUEADA (2030)	Verifica si hay un obstáculo a la izquierda.
5	der_libre	TK_DERECHA_LIBRE (2040)	Verifica si CAN puede moverse a la derecha.

Práctica de laboratorio

6	der_bloq	TK_DERECHA_BLOQUEADA (2050)	Verifica si hay un obstáculo a la derecha.
7	junto_zum	TK_JUNTO_A_ZUMBADOR (2060)	Verifica si hay un zumbador en la casilla de CAN.
8	no_junto_zum	TK_NO_JUNTO_A_ZUMBADOR (2070)	Verifica si no hay zumbador en la casilla.
9	zum_moc	TK_ALGUN_ZUMBADOR_EN_MOCHILA (2080)	Verifica si CAN tiene al menos un zumbador.
10	no_zum_moc	TK_NINGUN_ZUMBADOR_EN_MOCHILA (2090)	Verifica si CAN no tiene zumbadores.
11	orientado_norte	TK_ORIENTADO_NORTE (2100)	Verifica si CAN está mirando al norte.
12	orientado_sur	TK_ORIENTADO_SUR (2110)	Verifica si CAN está mirando al sur.
13	orientado_este	TK_ORIENTADO_ESTE (2120)	Verifica si CAN está mirando al este.
14	orientado_oest	TK_ORIENTADO_OESTE (2130)	Verifica si CAN está mirando al oeste.

Práctica de laboratorio

15	no_orientado_norte	TK_NO_ORIENTADO_NORTE (2140)	Verifica si CAN no está mirando al norte.
16	no_orientado_sur	TK_NO_ORIENTADO_SUR (2150)	Verifica si CAN no está mirando al sur.
17	no_orientado_este	TK_NO_ORIENTADO_ESTE (2160)	Verifica si CAN no está mirando al este.
18	no_orientado_oeste	TK_NO_ORIENTADO_OESTE (2170)	Verifica si CAN no está mirando al oeste.

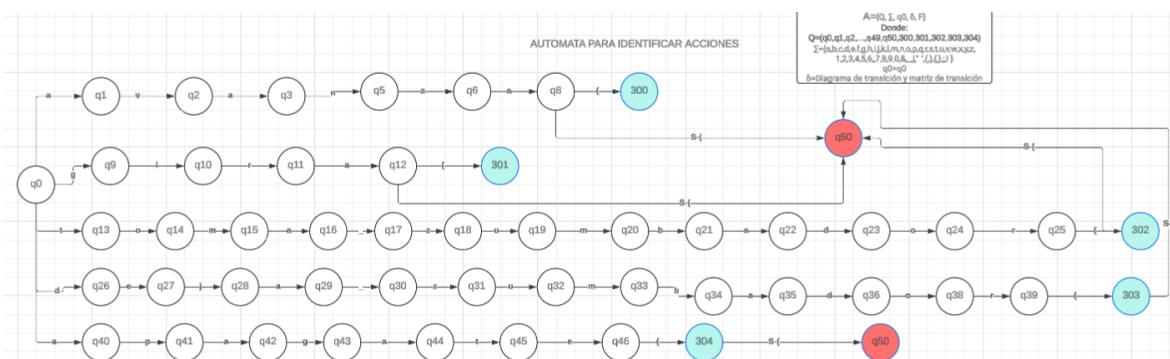


3. Acciones

#	Palabra Reservada	Token (Número)	Descripción
1	avanza	TK_AVANZA (3000)	Mueve a CAN una casilla hacia adelante.
2	gira	TK_GIRA_IZQUIERDA (3001)	Rota a CAN 90° a la izquierda.

Práctica de laboratorio

3	toma	TK_TOMA_ZUMBADOR (3002)	Recoge un zumbador de la casilla actual.
4	deja	TK_DEJA_ZUMBADOR (3003)	Deja un zumbador en la casilla actual.
5	apagate	TK_APAGATE (3004)	Finaliza la ejecución del programa.

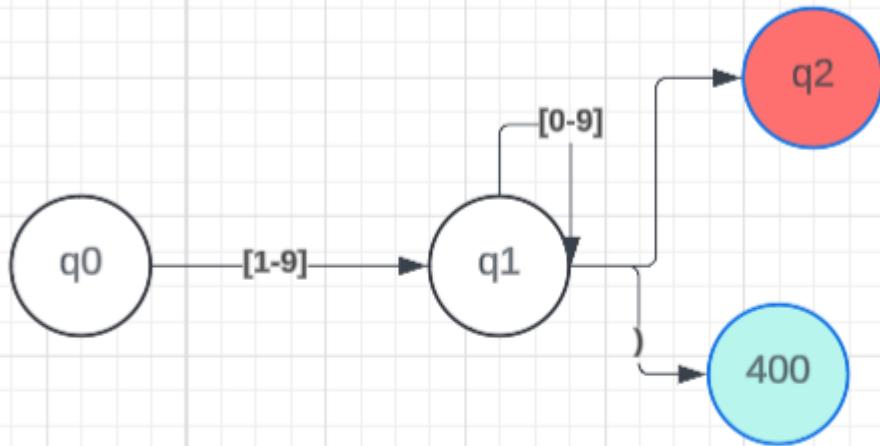


4. Definición de número enteros

Para la expresión de repetir es necesario indicar el número que se va a repetir, en este caso se encuentra la siguiente expresión donde no se incluye número enteros: [1-9]\d*

TK_NUMERO_ENTERO(400).

AUTOMATA PARA IDENTIFICAR NUMEROS ENTEROS



$A = \{Q, \Sigma, q_0, \delta, F\}$
 Donde:
 $Q = \{q_0, q_1, 400, \text{error}\}$
 $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
 $q_0 = q_0$
 $\delta = \text{Diagrama de transición y matriz de transición}$

5. DEFINICION DE SIGNOS DE PUNTUACION

#	Palabra Reservada	Token (Número)
1	(5000
2)	5001
3	;	5002
4	{	5004
5	}	5005
6		5006

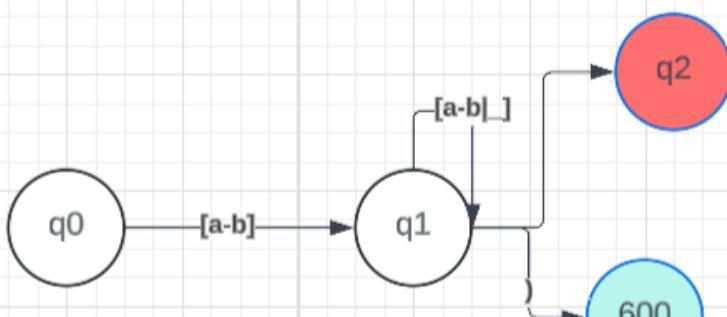
Práctica de laboratorio

7	!	5007
8	&&	5008

6. Definición de nombre de funciones

TK_DEDFINCTION_FUNCIONES(600)

AUTOMATA PARA IDENTIFICAR NOMBRE DE FUNCIONES



$$A = \{Q, \Sigma, q_0, \delta, F\}$$

Donde:

$$Q = \{q_0, q_1, 400, q_2\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$q_0 = q_0$$

δ =Diagrama de transición y matriz de transición

Ejemplo de un programa sencillo

```

inicio {
    repetir(3) { // CAN avanza tres veces
        avanza();
    }
    Si(junto_a_zumbador) {
        toma(); // CAN recoge un zumbador si hay
    }
    derecha(); // Llama a la función personalizada para girar a la
    derecha
}
  
```

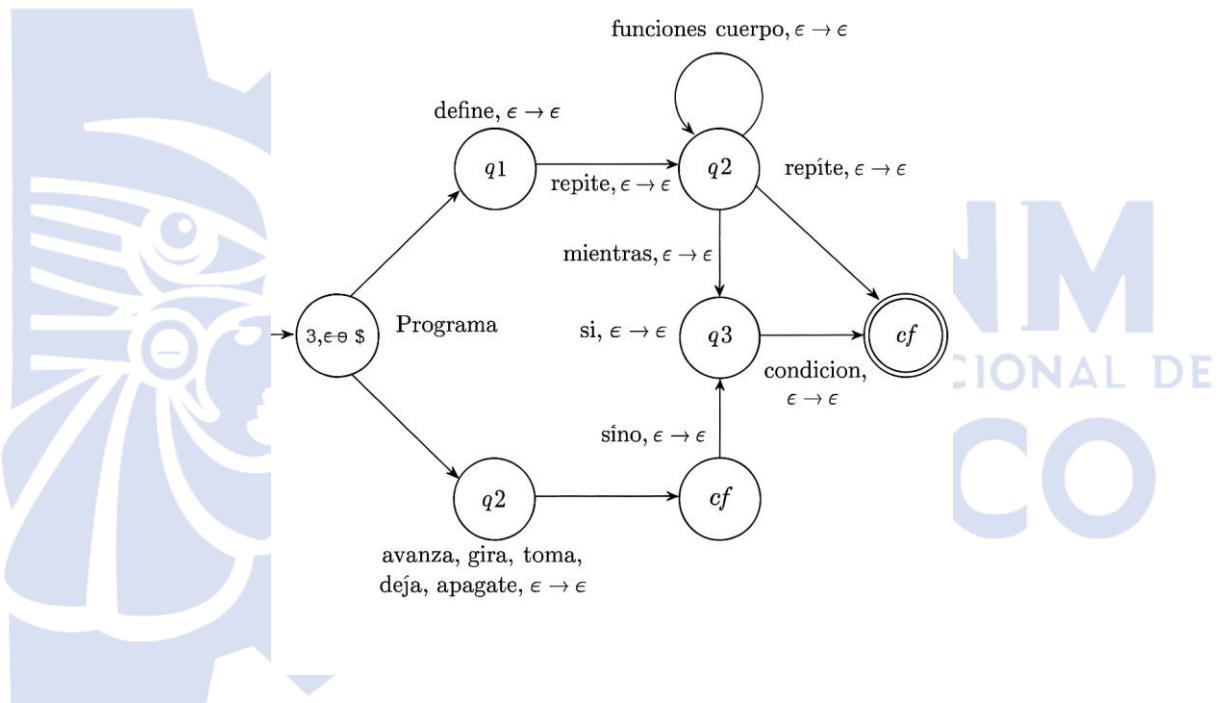
Práctica de laboratorio

}

```
// Función personalizada para girar a la derecha
definir girarDerecha() {
    repetir(3) {
        giraIzquierda();
    }
}
```

APR 2018

Sintáctico



La gramática define cómo se construyen los programas válidos. Se expresa con reglas de producción BNF simplificadas:

programa \rightarrow funciones instrucciones
 funciones \rightarrow define nombre { instrucciones } funciones | ϵ
 instrucciones \rightarrow instrucion instrucciones | ϵ
 instrucion \rightarrow avanza | gira | toma | deja | apagate
 | repite numero { instrucciones }
 | mientras (condicion) { instrucciones }
 | si (condicion) { instrucciones } sino { instrucciones }
 | si (condicion) { instrucciones }
 | llamada_funcion
 llamada_funcion \rightarrow nombre

Práctica de laboratorio

```

condicion      → condicion_base
| not condicion
| condicion and condicion
| condicion or condición

```

Notas:

- nombre y numero representan identificadores y valores numéricos enteros, respectivamente.
- ϵ representa la cadena vacía.

El análisis sintáctico toma una secuencia de tokens generados por el analizador léxico y construye un árbol de sintaxis abstracta (AST) siguiendo las reglas de producción.

Ejemplo de AST

```

{
  tipo: 'programa',
  cuerpo: [
    { tipo: 'avanza' },
    { tipo: 'repite', veces: 3, cuerpo: [
      { tipo: 'gira' },
      { tipo: 'avanza' }
    ]}
  ],
  funciones: [
    { tipo: 'define', nombre: 1, cuerpo: [ { tipo: 'toma' } ] }
  ]
}

```

Semántico

El análisis semántico verifica que el AST sea lógicamente válido. Incluye:

Verificación de funciones:

- Toda llamada a función (llamada_funcion) debe tener una definición previa (define).
- No se permite redefinir una función con el mismo nombre en el mismo ámbito.

INCIDENCIAS

Sin incidencias

OBSERVACIONES

- **Visualización clara y útil:** El uso de SvelteKit con Tailwind para mostrar el árbol de tokens es excelente. Es visual, legible y fácil de extender.
- **Tokens detallados:** Incluye no solo instrucciones sino también símbolos como (,), {, }, lo cual es útil para el análisis sintáctico completo.
- **Diagramas:** [autómatas](#)

Práctica de laboratorio

CONCLUSIONES

El desarrollo del sistema de análisis sintáctico y semántico, junto con su visualización mediante un árbol sintáctico en Svelte, permitió comprender y aplicar de manera integral los conceptos fundamentales de la construcción de compiladores. A través de la implementación del autómata de pila, se validó la estructura jerárquica del lenguaje, facilitando la detección de errores de estructura y coherencia en el código fuente. Además, la representación visual del árbol sintáctico ofreció una herramienta didáctica para entender cómo se construyen y analizan las expresiones en lenguajes de programación estructurados. Este enfoque promueve el aprendizaje interactivo y permite extender fácilmente la arquitectura del compilador hacia análisis más avanzados, como la generación de código intermedio o la optimización semántica.

BIBLIOGRAFIA

- *Tutorial de Karel.* (s. f.). <https://omegaup.com/karel.js/manual/Karel.html>
- Ufpb-Computacao. (s. f.). *compiladores-livro/livro/capitulos/2-lexica.asc at master · ufpb-computacao/compiladores-livro*. GitHub. <https://github.com/ufpb-computacao/compiladores-livro/blob/master/livro/capitulos/2-lexica.asc>
- GeeksforGeeks. (2021, 20 julio). *BNF Notation in Compiler Design*. GeeksforGeeks. <https://www.geeksforgeeks.org/bnf-notation-in-compiler-design/>