UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Estructuras Discretas

Grupo: 07 - Semestre: 2021-1

Proyecto 2:
Programa Tablas de Verdad

FECHA DE ENTREGA: 02/02/2021

Alumnos:

López Luna Demian Kheri

Esquivel Razo Israel

Osnaya Vázquez Josué Vicente

Quintanar Ramírez Luis Enrique

Sánchez Gutiérrez David

Santana Sánchez María Yvette

Téllez González Jorge Luis

Índice

1. Introducción

La aplicación desarrollada por el equipo *Nameless* de acuerdo a los requisitos del segundo proyecto para la asignatura de *Estructuras Discretas* es la siguiente:

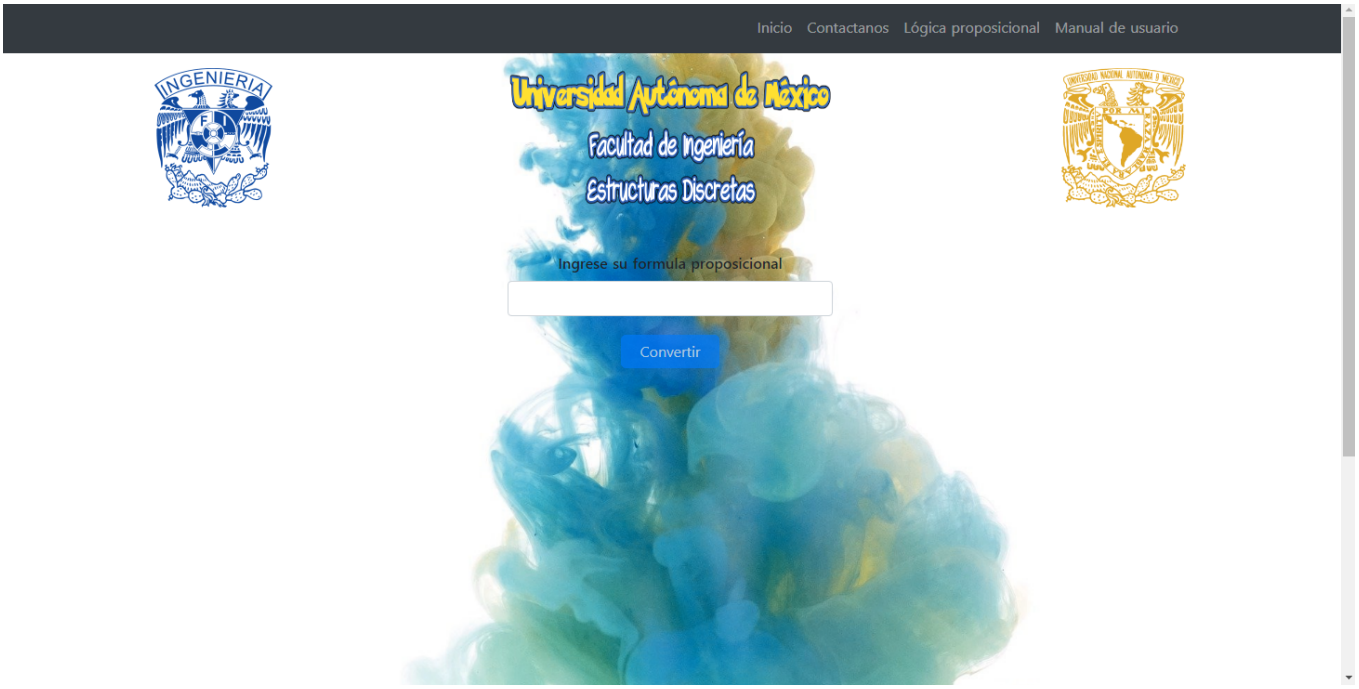


Figura 1: *Vista general de la aplicación.*

La aplicación contiene los siguientes elementos básicos:

- Interfaz responsiva que permite introducir una fórmula proposicional compuesta de hasta 6 atómicas o más, genera la tabla de verdad y señala si la fórmula introducida es una tautología, contengencia o contradicción.
- Una sección con información teórica acerca de la lógica proposicional y los conectores lógicos básicos.
- Un manual de usuario que indica los elementos y consideraciones necesarias para hacer uso de la aplicación web.

2. Uso de la aplicación web

Para utilizar la aplicación web, se ha optado por utilizar una herramienta llamada *Netlify*, la cual se trata de un servicio de *hosting* enlazado al repositorio de *Github* que contiene el código fuente del proyecto. De

este modo, se puede acceder a la aplicación web desde cualquier dispositivo con conexión a Internet sin necesidad de realizar ningún tipo de instalación.

El enlace para acceder a la aplicación web es el siguiente (Se recomienda el uso del navegador *Google Chrome* o un derivado de *Chromium*): `https://proverdad.netlify.app/`

2.1. Requerimientos técnicos

El desarrollo de la aplicación web ha requerido el uso indispensable de las siguientes tecnologías:

- JavaScript.
- Node.js
- Visual Code Studio
- Nuxt.js
- Bootstrap

Se recomienda el uso mínimo de un equipo de cómputo con un procesador **x64** (Intel/AMD) y **4GB** de RAM para trabajar con las herramientas listadas anteriormente. A continuación se detalla el proceso de instalación requerido para poder compìlar y montar la aplicación sobre un servidor local así como realizar modificaciones al código fuente.

2.2. Instalación

El primer elemento necesario para operar con el código fuente consiste en instalar el IDE **Visual Studio Code** en el siguiente enlace: `https://code.visualstudio.com/download`. Una vez descargado e instalado con las opciones predeterminadas de instalación, deberá de descargar e instalar **Node.js** a través del siguiente enlace: `https://nodejs.org/es/download/`.

Puede marcarse la casilla *Automatically install the necessary tools* para tener todos los módulos **npm** que puedan necesitarse posteriormente. Tras realizar la instalación inicial, se abrirá un script que instalará desde Windows PowerShell (Para el caso de W10) el software **Chocolatey**, un gestor de paquetes que automatiza la instalación del software necesario para hacer uso de Node.js y el gestor de paquetes para JavaScript **npm**.

Lo anterior puede omitirse si así se desea para realizar la instalación manual de cualquier dependencia que se pueda requerir; para el proceso actual de instalación no es necesario marcar esa casilla.

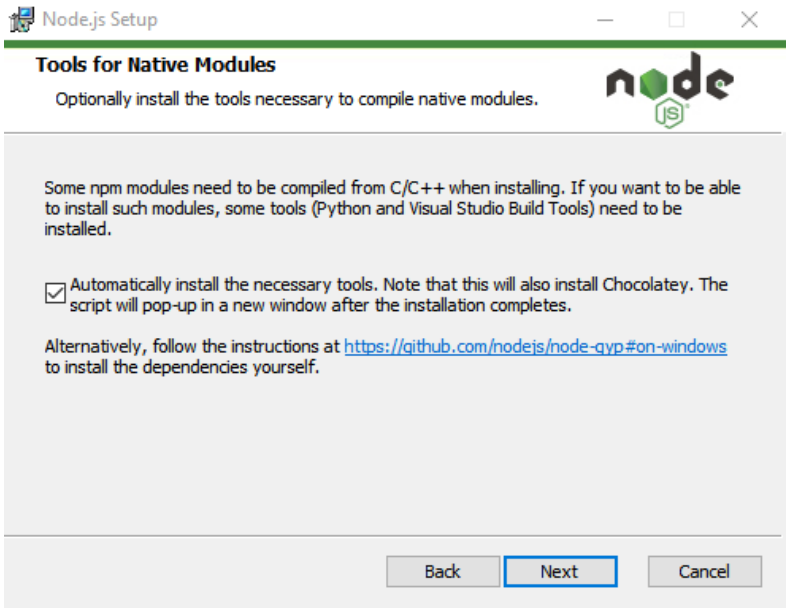


Figura 2: *Instalación de Node.js*

Una vez que se han instalado ambas herramientas, se procede a ejecutar **Visual Studio Code** y abrir el código fuente del proyecto en la pestaña *File* $->$ *Open Folder*. Una vez que se abra la ruta que contenga la carpeta con el código fuente del proyecto, se debe de utilizar la pestaña *Terminal* para iniciar una ventana de PowerShell como puede verse a continuación:

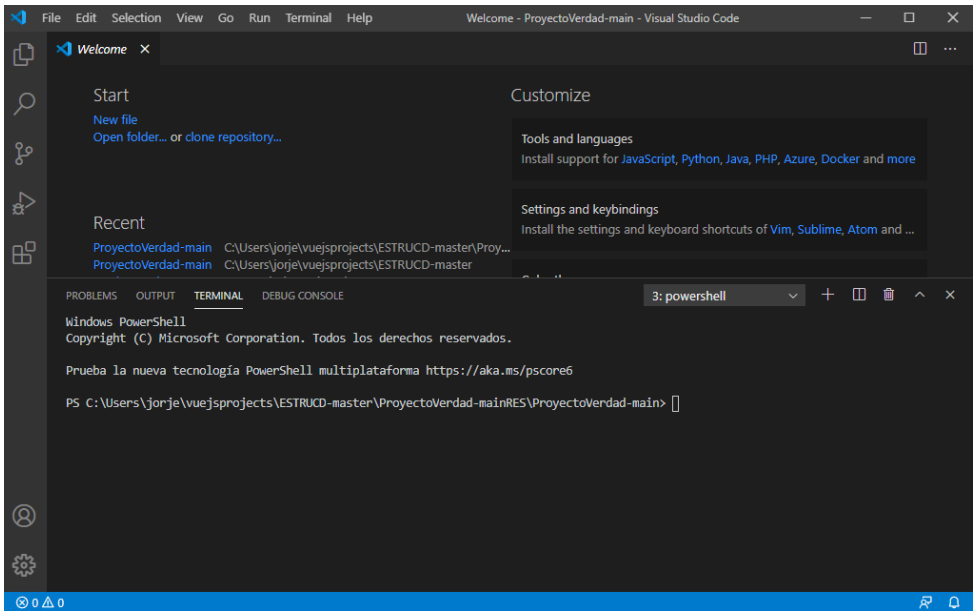


Figura 3: *Terminal de PowerShell en VSCode.*

Finalmente, deberán de escribirse en la consola dos comandos:

1. **npm install**: Este comando instalará todas las dependencias necesarias para la aplicación web de forma automática. Este proceso de instalación puede tomar un par de minutos cuando es realizado por primera vez.

2. **npm run dev**: Por medio de este comando se realizará el proceso de compilado y despliegue de la aplicación web utilizando un servidor local al cual puede accederse utilizando un navegador web.

Si los anteriores procesos son realizados de forma correcta, deberá de verse la siguiente salida en la terminal con el enlace para acceder a la aplicación web utilizando la combinación *ctrl+click* al ubicar el cursor sobre el enlace que se encuentra sombreado en azul oscuro:

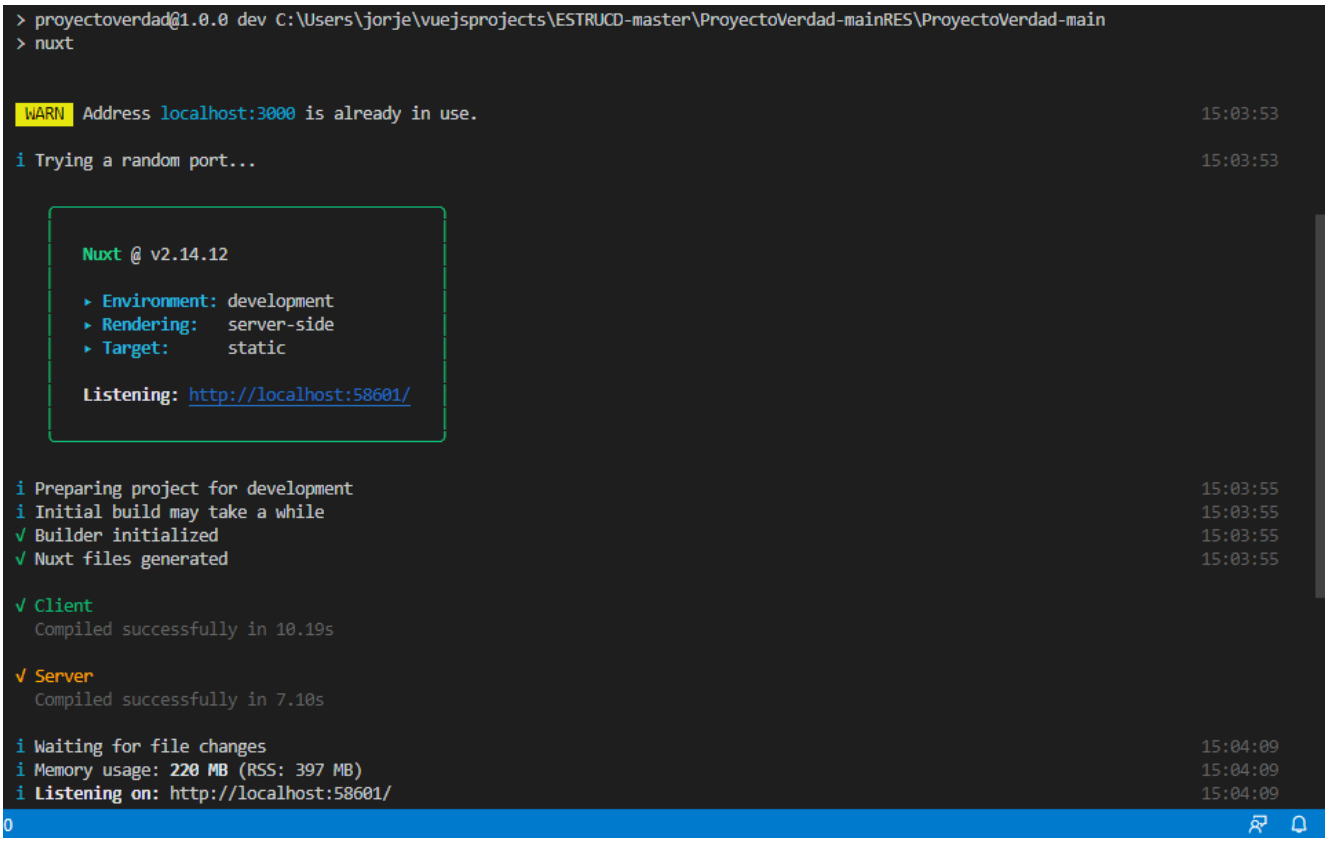


Figura 4: *Despliegue local de la aplicación web.*

3. Diseño del programa

Las páginas web que conformar la aplicación se encuentran en la subcarpeta **pages**. El archivo **index.vue** contiene toda la interfaz gráfica de la aplicación así como los métodos necesarios para el procesamiento de

las fórmulas proposicionales. De forma general, es posible dividir en tres campos principales el segmento de código: **Generación de tablas, diseño gráfico y lógica interna**.

3.1. Aspecto gráfico

El aspecto gráfico del programa se compone de la generación de tablas y el diseño gráfico. A continuación se detallarán cada uno de estos aspectos:

3.2. Estructura general del programa

La página web contiene un estilo modular con diversas carpetas que conforman elementos distintos para definir su aspecto y las subpáginas que la conformen. Muchos elementos del proyecto generado se crean de forma automática con **Nuxt.js**. Para generar este proyecto por medio de la consola de comandos (Con Node.js previamente instalado) se utiliza el comando **npm create nuxt-app *nombre***. De esta forma, la página web será generada y tendrá como carpetas de relevancia las siguientes:

- **components**: Esta carpeta contiene 4 archivos **.vue** que definen los aspectos básicos de la vista de la página web.

    1. **Navbar**: define la barra de navegación en la parte superior de la página web.

    2. **Footer**: Contiene el pie de página de la aplicación web.

    3. **Head**: define las imágenes de fondo y la presentación base de la página web. Es posible configurar la página en esta carpeta de tal forma que al redimensionar el navegador la página modifique su aspecto de forma consecuente.

    4. **Logo**: Carpeta que contiene el logo de nuxt. En este caso no se utiliza.

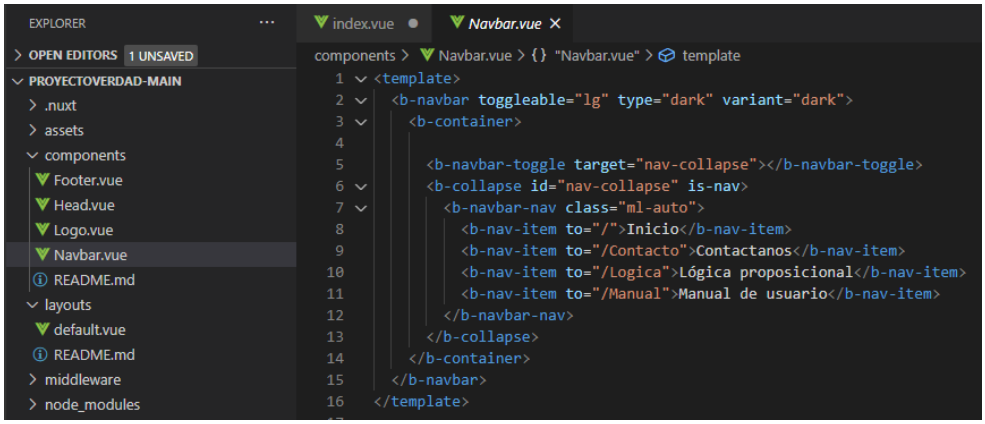


Figura 5: *Carpeta components.*

- **layouts**: En el archivo *default.vue*, la etiqueta **template** guarda la división general que la página web tendrá, así como las subpáginas que la van a conformar, por lo que representa un aspecto clave de la página web. De esta forma, es posible manipular el orden de los elementos de la plantilla web.

  Por otra parte, la etiqueta **style** define con lenguaje CSS el diseño gráfico general de la página web, como la tipografía a utilizar, el ajuste de línea o el tipo de espaciado si así se desea. En este punto también es posible modificar el fondo por defecto y definir cualquier imágen que se desee.

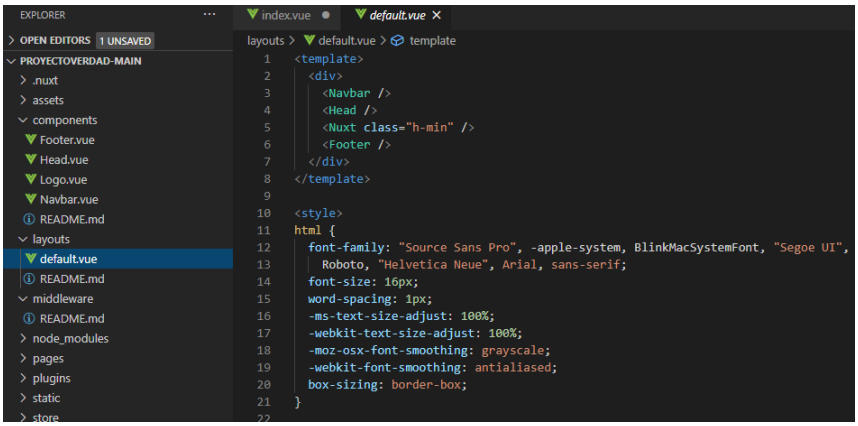


Figura 6: *Carpeta layouts.*

assets: Esta carpeta contiene los elementos gráficos que van a referenciarse y utilizarse al interior de la página web (Como el título, los escudos de la UNAM y la FI así como el fondo de la página).

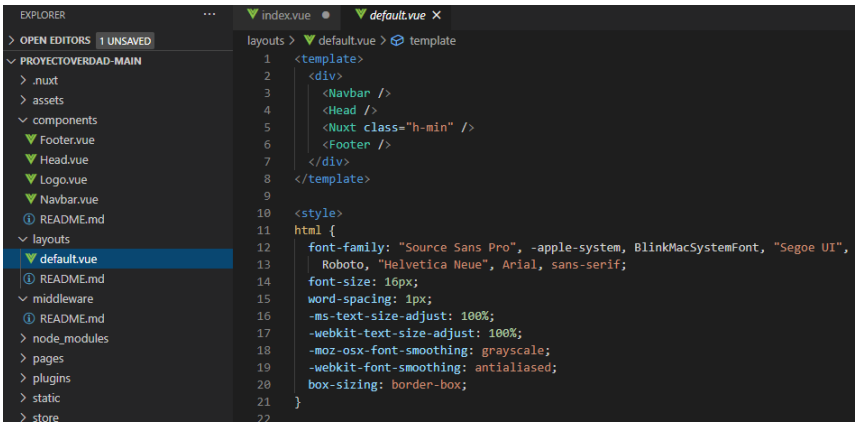


Figura 7: *Carpeta assets.*

pages: Contiene los archivos para las subpáginas web que se definan en la barra de navegación. Para este caso, se consideran los siguientes archivos: **index, Logica y Manual**. El primer archivo contiene toda la página web y su algoritmo de desarrollo, el segundo contiene el código necesario para mostrar

un PDF sobre lógica proposicional en pantalla y el tercero de igual manera muestra un PDF pero que aborda el uso de la aplicación web.

Figura 8: *Carpeta pages.*

3.3. Implementación

El código principal de la página web se encuentra en el archivo **index.vue** y contiene tres aspectos esenciales:

1. La etiqueta **style** como en el caso anterior determina la tipografía y la vista general de la aplicación, sin embargo, en este segmento de código también se definen las propiedades de las tablas a utilizar para la generación de las tablas de verdad por medio de *Bootstrap*.

```
table {
  min-width: 100%;
  margin: 20px;
  background-color: #208fcfc4;
  box-shadow: 0px 0px 10px;
  color: whitesmoke;
}
tr {
  height: 50px;
  border: 2px solid white;
  background-color: #ebff39dc;
}
tbody tr:nth-child(odd) {
  background-color: #edfa75d3;
  border: 2px solid white;
}
```

Figura 9: *Elementos de configuración en las tablas CSS.*

2. La etiqueta **template** contendrá los elementos de *Bootstrap* necesarios para implementar la creación de botones responsivos y tablas según sea necesario.

Bootstrap divide de forma general una página en *containers* de forma que se tiene control de columnas y renglones para definir de forma preciosa los elementos que se quieren añadir en la página y manipular adecuadamente su ubicación. Así mismo, también se puede modificar su alineación así como la alineación del texto en su interior si lo tuviese.

En la siguiente parte del código se define un elemento **b-form group** utilizado para generar un formulario que guardará la entrada escrita por el usuario en una variable denominada **input** que podrá ser enviada al método **dibujarTabla** utilizando el elemento **b-button**. Cabe destacar que, si no hay nada escrito en el formulario, la página no permitirá al usuario enviar absolutamente nada para su revisión y verificación.

```
<template>
  <div>
    <b-container class="text-justify">
      <b-row align-h="center" class="mt-5">
        <b-col cols="12" lg="4">
          <b-form-group
            label="Ingrese su formula proposicional"
            label-class="h2 text-center"
          >
            <b-input type="text" v-model="input"></b-input>
          </b-form-group>
          <div class="text-center">
            <b-button
              :disabled="input === '' || input.includes(' ') ? true : false"
              variant="primary"
              class="mt-1"
              @click="dibujarTabla(input)"
              >Convertir</b-button
            >
          </div>
        </b-col>
      </b-row>
```

Figura 10: *Formulario y botón de la página web.*

En los siguientes renglones definidos con **b-row** se encuentran las definiciones de qué mostrar en caso de que la entrada introducida no se trate de una fórmula proposicional o se encuentre mal escrita de acuerdo a los lineamientos de escritura, así mismo, define el caso de *éxito* cuando la entrada es correcta, procediendo a enviar la entrada para su evaluación y para crear la tabla de verdad correspondiente.

Nótese que es posible modificar los colores de los elementos gráficos utilizando el parámetro *variant*. Por ejemplo, la opción **danger** muestra el elementos gráfico de color rojo, mientras que la opción **success** lo muestra en color verde para indicar que la operación ha sido realizada con éxito. Entre otros parámetros puede encontrarse la alineación del texto.

```
<b-row align-h="center">
  <b-col cols="12" lg="4">
    <b-alert variant="danger" :show="show">{{ error }}</b-alert>
    <b-alert class="text-center my-3" variant="success" :show="showTabla">
      {{ eval }}
    </b-alert>
  </b-col>
</b-row>
<b-row align-h="center">
  <b-col v-if="showTabla" cols="12" lg="7" class="mt-5 mt-lg-0">
    <h5 style="color:yellow !important;" align="center">
      Tabla de Verdad:
    </h5>
    <b-table
      class="text-center"
      striped
      hover
      :items="items"
      :fields="fields"
    ></b-table>
```

Figura 11: *Mensajes de error o éxito.*

3. La etiqueta **script** contiene el algoritmo implementado para actuar como *parser* para leer la fórmula proposicional y manipularla con el fin de generar la tabla de verdad correspondiente.

   En el segmento *data* se encuentran declaradas las variables a utilizar durante el desarrollo del programa, como puede verse a continuación.

```
data() {
  return {
    input: "",
    show: false,
    showTabla: false,
    error: "Error",
    items: [],
    fields: [],
    eval: ""
  };
},
```

Figura 12: *Declaración de variables.*

El código comienza con el método **dibujarTabla**, este recibe la expresión del usuario que desea ejecutar, posteriormente, se utilizan los siguientes métodos para la expresión:

- **validar**: Éste verifica si la entrada es correcta, en conjunto de los métodos **veriCaracs** y **balance**. En el caso de **veriCaracs**,los caracteres son revisados uno a uno dentro de la cadena de expresión, sólo continuará si la expresión dicha contiene letras para representar proposiciones

atómicas u operaciones con los métodos **esLetra** y **esSimbolo** respectivamente consultando el código ASCII.

```
function esLetra(car) {
  let ascii = car.toLowerCase().charCodeAt(0);
  return ascii > 96 && ascii < 123 && ascii != 118;
}
// Devuelve verdadero o falso si es un numero o no
function esNumero(car) {
  return !isNaN(car);
}
// Devuelve verdadero o falso si el símbolo está dentro de los que se usaran
function esSimbolo(car) {
  //ascii [94,118,170,26,29,40,41]
  // ^: conjuncion; v: disjuncion; ~: Negacion; →: condicional; ↔: Bicondicional
  let caracts = ["^", "v", "¬", "→", "↔", "(", ")"];
  for (let i = 0; i < caracts.length; i++) {
    if (car == caracts[i]) {
      return true;
    }
  }
  return false;
}
// Indica si la expresion tiene los caracteres que nos interesan
function veriCaracs(cad) {
  for (let i = 0; i < cad.length; i++) {
    if (!esLetra(cad[i]) && !esNumero(cad[i]) && !esSimbolo(cad[i])) {
      return false;
    }
  }
  return true;
}
```

Figura 13: *Validaciones para revisar que la cadena esté bien formada.*

- **balance**: Es el método dónde se revisan los paréntesis, con el uso de una pila, se recorre la expresión y se agregan los paréntesis izquierdos a la pila, en caso de que se encuentre un paréntesis derecho se vacía la pila, hacemos éste procedimiento hasta que terminemos de leer la expresión, si la pila termina vacía, significa que no tenemos paréntesis de más.

```
function balance(expr) {
  let pila = [];
  let balnceado = true;
  let indice = 0;
  while (indice < expr.length && balnceado) {
    let simbolo = expr[indice];
    if (simbolo == "(") {
      pila.push(simbolo);
    } else {
      if (simbolo == ")") {
        if (pila.length <= 0) balnceado = false;
        else pila.pop();
      }
    }
    indice = indice + 1;
  }
  if (balnceado && pila.length <= 0) return true;
  else return false;
```

Figura 14: *Validación de paréntesis utilizando una pila.*

- **posfija**: Si el balance es correcto, procedemos a convertir la expresión de infija a postfija debido a las ventajas que tiene para este tipo de lecturas. El método **posfija** genera un diccionario donde la llave es la operación y el valor tiene el número de precedencia. Se genera una lista dónde colocaremos el resultado y una pila para acomodar los operadores. Hacemos un ciclo *for* que recorra toda expresión:

  a) Si comienza con una letra, se va directamente a la lista de salida.

  b) Si es un paréntesis izquierdo, se coloca en la pila de operadores.

  c) Si es un paréntesis derecho, se vacía la pila a la lista, hasta que sea un paréntesis izquierdo, así colocamos las operaciones en la lista.

  d) En caso de que sea otro símbolo se compara con el orden de precedencia y si es de mayor precedencia se agrega a la pila.

```
function posfija(exp) {
  let precedencia = { "¬": 4, "^": 3, v: 3, "→": 2, "↔": 1, "(": 0 };
  let pilaOp = [];
  let sufija = [];
  for (let i = 0; i < exp.length; i++) {
    if (esLetra(exp[i]) || esNumero(exp[i])) {
      sufija.push(exp[i]);
    } else {
      if (exp[i] == "(") {
        pilaOp.push(exp[i]);
      } else {
        if (exp[i] == ")") {
          let simTope = pilaOp.pop();
          while (simTope != "(") {
            sufija.push(simTope);
            simTope = pilaOp.pop();
          }
        } else {
          while (
            pilaOp.length > 0 &&
            precedencia[pilaOp[pilaOp.length - 1]] >= precedencia[exp[i]]
          ) {
            sufija.push(pilaOp.pop());
          }
          pilaOp.push(exp[i]);
        }
      }
    }
  }
  while (pilaOp.length > 0) {
    sufija.push(pilaOp.pop());
  }
  return sufija.join("");
```

Figura 15: *Algoritmo RPN implementado.*

- **cambiarAtomicas**: Al usar el método se crea una tabla con ceros y unos denominada *matriz* y también se genera una lista auxiliar y una bandera. El trabajo de la bandera y la lista *aux* es copiar la expresión y sustituir el símbolo por el valor correspondiente ya que este ya se encuentra transformado en notación postfija, después sustituirlo en nuestra tabla o matriz donde 0 se cambiará por un **F** y uno por una **T** según corresponda.

```
// Cambia los valores de las atomicas por 'T' y 'F' segun cada caso y evalua la expresión
function cambiarAtomicas(exp, atomicas) {
  let matriz = crear(atomicas.length);
  let aux;
  for (let h = 0; h < matriz[0].length; h++) {
    aux = [];
    for (let i = 0; i < exp.length; i++) {
      let bandera = false;
      for (let j = 0; j < atomicas.length; j++) {
        if (exp[i].toLowerCase() == atomicas[j].toLowerCase()) {
          aux.push(matriz[j][h]);
          bandera = true;
        }
      }
      if (bandera == false) aux.push(exp[i]);
    }
    matriz[atomicas.length].push(evaluar(aux));
  }
  return matriz;
}
```

Figura 16: *Matriz de valores booleanos.*

- **atoms**: Genera una lista donde almacena cada una de las proposiciones atómicas a partir del la condicional. El método mencionado llama a otro llamado **actualizarAtomicas** dónde se revisa si están repetidas las proposiciones atómicas.

```
// Devuelve las atomicas de la expresion
function atoms(exp) {
  let atomicas = [];
  for (let i = 0; i < exp.length; i++) {
    if (esLetra(exp[i]))
      actualizarAtomicas(atomicas, exp[i].toLowerCase());
  }
  return atomicas;
}
```

Figura 17: *Lista de atómicas en la expresión.*

- **evaluar**: Este método es utilizado en el método **cambiarAtomicas** para evaluar las expresiones sustituidas por **T** o **F** con los operandos que se encuentren en la expresión (Que ya se encuentra en RPN y evita cualquier tipo de ambigüedad en la resolución de cada expresión) sustituyendo cada bloque por **T** o **F** según corresponda y sucesivamente revisar el siguiente elemento operando para continuar determinando los valores de verdad hasta que finalmente la pila quede vacía y la expresión quede evaluada para uno de los posibles casos que se puedan presentar en los valores de verdad de las atómicas.

```
function evaluar(exp) {
  let pilaOP = [];
  for (let i = 0; i < exp.length; i++) {
    if (exp[i] == "T" || exp[i] == "F") {
      pilaOP.push(exp[i]);
    } else {
      if (exp[i] == "¬") {
        let operando = pilaOP.pop();
        pilaOP.push(operando == "T" ? "F" : "T");
      } else {
        let atom2 = pilaOP.pop();
        let atom1 = pilaOP.pop();
        if (exp[i] == "^")
          pilaOP.push(atom1 == "T" && atom2 == "T" ? "T" : "F");
        else if (exp[i] == "v")
          pilaOP.push(atom1 == "T" || atom2 == "T" ? "T" : "F");
        else if (exp[i] == "→")
          pilaOP.push(atom1 == "T" && atom2 == "F" ? "F" : "T");
        else if (exp[i] == "↔") pilaOP.push(atom1 == atom2 ? "T" : "F");
      }
    }
  }
  return pilaOP.pop();
```

Figura 18: *Método de evaluación de los valores de verdad.*

El orden general de ejecución de cada uno de los métodos empleados en el programa es el siguiente:

dibujarTabla(exp)
- *validar(exp)*
  - *veriCaracs(exp)*
  - *balance(exp)*
  - *posfija(exp)*
  - *cambiasAtomicas(posfij)*
  - *Retorna la expresión convertida a posfija y evaluada.*
- *atoms(exp)*
  - *actualizarAtomicas*

Figura 19: *Lista de atómicas en la expresión.*

Finalmente, una vez que este proceso es terminado y se generó la matriz correspondiente con los valores de verdad, únicamente se imprime con las configuraciones de CSS y Bootstrap declaradas previamente. Para esto, se verifica que en primer lugar la tabla se genere correctamente utilizando un **try-catch** al utilizar los métodos **validar** y **atoms** y recuperar los elementos de la tabla generada con un ciclo **for**, Posteriormente, se utiliza un mapeo para ubicar adecuada-

mente cada valor de verdad en la tabla generada en el diseño de la web y finalmente mostrar el resultado en pantalla.

```
try {
  tabla = validar(formul);
  atomicas = atoms(formul);

  for (let i = 0; i < tabla[0].length; i++) {
    tabla.forEach((element, j) => {
      columna.push(element[i]);
    });
    fila.push(columna);
    columna = [];
  }

  this.items = fila;
  this.fields = atomicas.map((e, i) => ({
    key: i.toString(),
    label: e
  }));
  this.fields.push({ key: atomicas.length.toString(), label: formul });

  this.items = fila.map(obj => {
    let rObj = {};
    fila[0].forEach((e, i) => {
      rObj[i] = obj[i];
    });
    return rObj;
  });
```

Figura 20: *Generación de la tabla.*

Cabe señalar que si en alguno de estos puntos se genera una excepción, los valores **showTabla, show** automáticamente tendrán un valor booleano **false** y se generará un mensajo de error a la salida.

4. Pruebas de validación

Para verificar que el programa estuviese funcionando adecuadamente se realizaron 2 sets de pruebas en total durante el desarrollo del programa. La primer prueba será anexada como documento adjunto a la documentación presentada, y en este punto será abordado el segundo set de pruebas.

4.1. Pruebas realizadas y evidencias

A continuación se presentan las siguientes fórmulas proposicionales que se utilizan para probar el programa y verificar que estuviese arrojando los resultados de forma adecuada. Para esto, se utilizaro en total 10 fórmulas proposicionales que fueron estudiadas durante el curso de *Estructuras Discretas*

para comparar la tabla de verdad generada por el programa por la creada durante el desarrollo del curso. Como se observa a continuación, los resultados obtenidos resultan correctos y en concordancia con lo esperado para el programa.

Fórmula proposicional	Resultado esperado	¿Resultado obtenido?	¿Tabla generada correctamente?
(p→q)v(pvq)	Tautología	✔	✔
(pv¬q)^(p→q)	Contingencia	✔	✔
(p^q)^¬(pvq)	Contradicción	✔	✔
pv(¬p→(qv(¬q→r)))	Contingencia	✔	✔
(p^q)v¬(p^q)	Tautología	✔	✔
(p^q)v¬(p^q)	Contingencia	✔	✔
(p^q)^¬(pvq)	Contradicción	✔	✔
(q→p)^(¬p^q)	Contradicción	✔	✔
(¬pvq)^(p^(p^q))↔(p^q)	Tautología	✔	✔
pv(¬p^q)	Contingencia	✔	✔

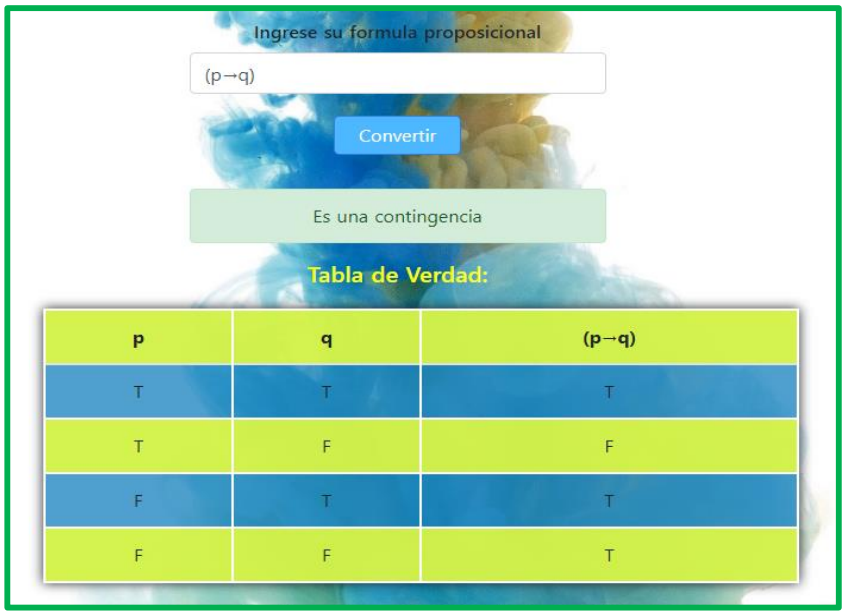
Figura 21: *Lista de atómicas en la expresión.*

La evidencia del desarrollo de estas pruebas se encuentran alojadas en el servidor de *Google Drive* y puede accederse a su visualización por medio del siguiente enlace: `https://drive.google.com/file/d/1x6Y6OJMCAfnRbcOS_wCzYrpu-NDyKmUS/view?usp=sharing`

Pruebas realizadas / Evidencias de pruebas

Las pruebas realizadas en nuestro proyecto son:

- **(p→q)**

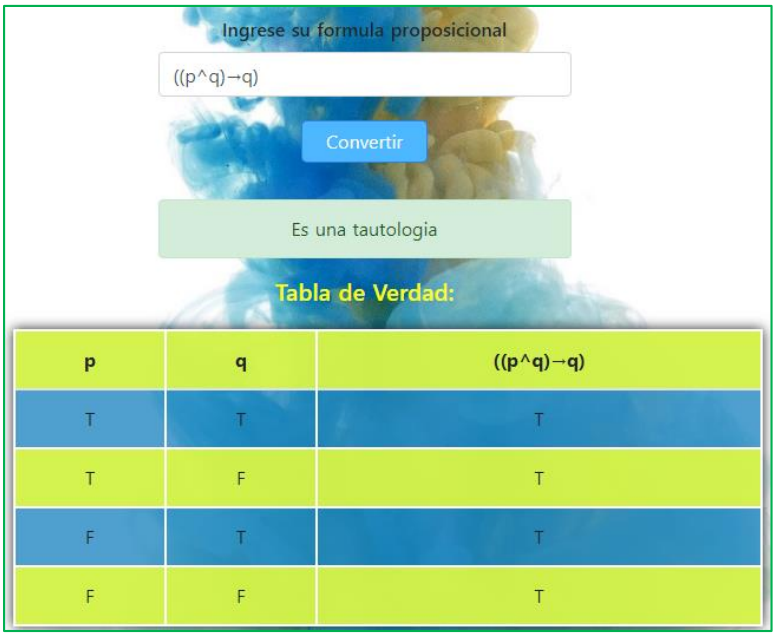


p	q	(p→q)
T	T	T
T	F	F
F	T	T
F	F	T

- **(p→q)^(q^p^¬q)**

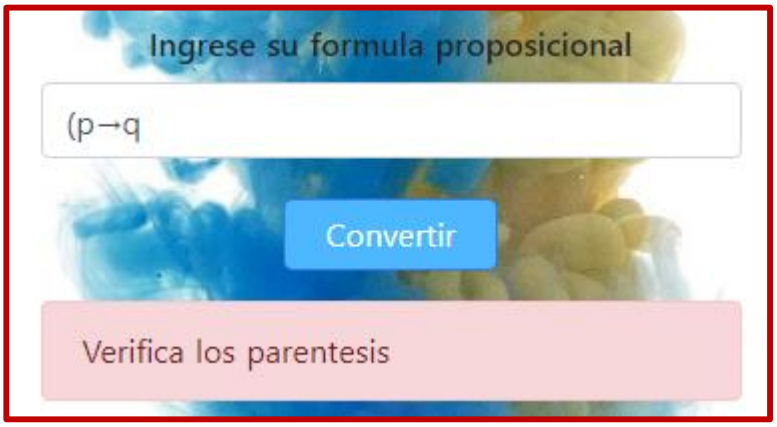
p	q	(p→q)^(q^p^¬q)
T	T	F
T	F	F
F	T	F
F	F	F

- **((p^q)→q)**

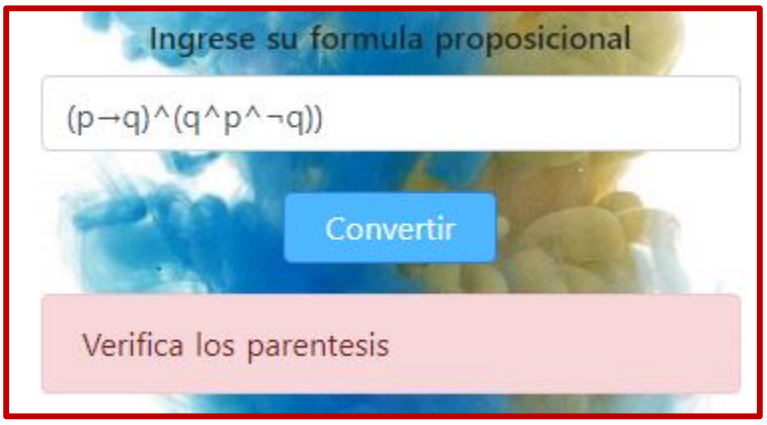


p	q	((p^q)→q)
T	T	T
T	F	T
F	T	T
F	F	T

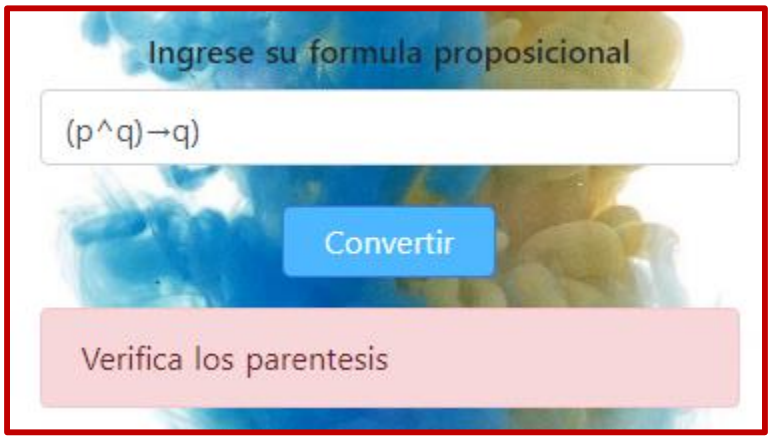
- **(p→q**



- **(p→q)^(q^p^¬q))**

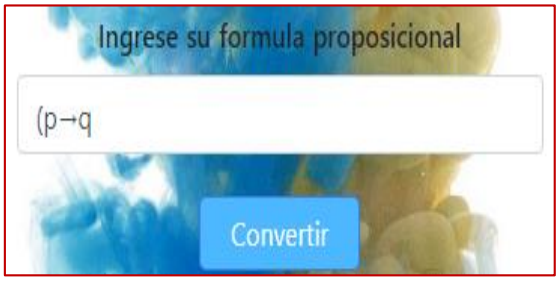


- **(p^q)→q)**



Podemos observar, que sí tecleamos algún espacio nuestro botón **Convertir,** no estará activo, es decir, no se puede dar un click:

SIN ESPACIOS



CON ESPACIOS