Depto. De Cs. e Ing. De La Computación
Universidad Nacional del Sur

Tesis de Licenciatura
en Ciencias de la Computación

Templates Declarativas en React: Simplificando la definición de interfaces gráficas utilizando transpilación

Juan Pablo Sumski
Director: Sebastián Gottifredi

Bahía Blanca Argentina

Diciembre de 2023

Índice

1 Introducción

En sus inicios la web fue concebida como una colección de documentos de texto plano, que ocasionalmente contenía alguna imágen y un poco de color, pero que no disponían de ninguna funcionalidad o capacidad de interacción más que algún enlace a otro documento, si no que eran páginas estáticas. Para añadirle interacción a estas páginas como cambios dinámicos de interface, procesamiento de datos, la posibilidad de realizar solicitudes a un servidor sin recargar la página, entre otras funcionalidades, surge el lenguaje JavaScript, el cuál es interpretado y ejecutado por un navegador junto con la página web.
El uso de este lenguaje de scripting creció tanto en popularidad desde su salida en 1996 al punto que hoy casi todas las páginas webs que visitamos tienen código JavaScript. En este contexto, apoyándonos en su evolución y el poder de cómputo del mismo, hoy tenemos aplicaciones web de todo tipo: suites de ofimática, reproductores de música/películas, programas de edición o servicios de comunicación por video y audio en tiempo real. Hace no mucho tiempo, todas estas aplicaciones requerían que instalemos un software de manera nativa en nuestra computadora, hoy las tenemos disponibles en cualquier dispositivo que tenga un navegador y acceso a internet, lo que logra hacer universales estos servicios.

Uno de los principales motivos del crecimiento de la popularidad de este lenguaje fue que los desarrolladores que resolvían un problema utilizando JavaScript, empaquetaban sus soluciones en librerías para compartirlas y que otras personas pudieran utilizarlas en sus proyectos. Un tipo de librería muy importante hoy en día en el desarrollo de estas aplicaciones web modernas son los *frameworks front-end*, que nos otorgan un marco para la creación de aplicaciones dinámicas e interactivas, el cuál busca lograr escalabilidad, homogeneidad y facilidad de mantención para las mismas, de manera tal que la tarea del desarrollador se centre en resolver los problemas específicos de la aplicación y no en lidiar con cuestiones del lenguaje vanilla. Debe quedar claro que estas librerías no dotan de más funcionalidad a los formatos utilizados en la web, sino que simplifican el desarrollo ahorrándole al programador la tarea de programar características comunes a cualquier aplicación web como pueden ser administrar el estado, seguridad o ruteo, entre otras.

Sin embargo, la feature más interesante de estos frameworks se da en forma de abstracción a las tecnologías subyacentes, principalmente a través de la extensión de los lenguajes ya existentes o la creación de nuevos *lenguajes de dominio específico* que permiten al desarrollador definir componentes de la interfaz gráfica y su interacción de manera declarativa y más sencilla sin entrar en el detalle de cómo se actualizan y mantienen en el ciclo de la aplicación. Estos lenguajes no son interpretados por el navegador, sino que sufren de un proceso denominado *transpilación* que se define como la traducción de un lenguaje de alto nivel a otro también de alto nivel, ya sea de manera interpretada o precompilada, terminan convirtiéndose en HTML, CSS y JavaScript para la ejecución del navegador.

Esta característica se suele conocer como *lenguaje de templating web* o al menos consiste en un lenguaje que reúne algunas características de los mismos. Estos lenguajes no son más que una abstracción que consiste en una forma declarativa de definir, a modo de

plantilla, la estructura básica estática de una página y partes dinámicas que deben ser completadas con información del modelo o generadas en base a parámetros de este. Estos *templates*, al igual que otros archivos de una app desarrollada en el marco de un framework front-end deben pasar por un proceso, en este caso a través de un *motor o procesador de plantillas*, que se encarga de procesar la plantilla y combinar con los datos del modelo de la aplicación de manera adecuada para generar así documentos web.

Si bien hay *frameworks front-end* actuales y populares que soportan plantillas e incluyen una herramienta de transpilación para las mismas como es el caso de Vue o Angular, tenemos otras librerías como React que carecen de este soporte y en su lugar utilizan un híbrido entre JavaScript y HTML, llamado JSX para definir las vistas de una aplicación web y su interacción con el modelo y la lógica. Sabiendo esto, a lo largo de esta tesis se mostrará un lenguaje de templating para incluir en React y además se implementará una herramienta, a modo de librería que sirva de procesador de este lenguaje del lado del cliente de la aplicación web, permitiendo un desarrollo más declarativo y más desligado del resto de la lógica de la aplicación a desarrollar, buscando acercarse al patrón arquitectónico que siguen otros frameworks, *modelo-vista-controlador*. Se detallará el proceso de diseño de este lenguaje y su herramienta traductora, se analizará la solución actual presentada por React al desarrollo de interfaces y se contrastarán los puntos a favor y en contra con la alternativa de la tésis, haciendo foco en aspectos clave como arquitectura u optimización. Por último se describirán mejoras posibles y/o necesarias para lograr tener un lenguaje más completo y de calidad comercial.

2 Resolución de Problemas y Arquitectura

Por lo anteriormente visto en la introducción queda claro que el problema a resolver se puede dividir en dos partes: una, la definición de un lenguaje de *templating* y dos, la creación de la herramienta de traducción que a partir de documentos escritos en este nuevo lenguaje y un modelo y lógica de aplicación pueda generar el código de una página web interactiva que respete la estructura y la funcionalidad definida en términos del lenguaje.

En este capítulo vamos a abstraernos de las tecnologías a utilizar en la implementación final de la herramienta e incluso de la sintaxis del lenguaje y solo nos concentraremos en la resolución de los problemas de manera general y las características que deben tener las soluciones a estos.

2.1 Diseño del lenguaje

El primer punto a resolver es sencillo, recordemos que un **lenguaje de templating web** es un **lenguaje de dominio específico** diseñado para poder escribir plantillas a partir de las cuales se puedan generar páginas webs dinámicas.

Este lenguaje debe permitir describir la **estructura estática** de una página web, con contenido y sus atributos, y además debe permitir **invocar constructores o directivas** que luego serán traducidos a **estructura dinámica** nueva con contenido, funcionalidad y atributos o a modificaciones de la estructura estática antes definida.

Sabiendo que el resultado tiene que ser una página web y contemplando las características y funcionalidades antes mencionadas, definir el lenguaje resulta algo casi trivial, con dos partes muy marcadas: una **estática** y otra **dinámica**.

Veamos la parte estática, considerando el output como un documento HTML, sabemos que este formato se compone de una estructura llamada ***document object model o DOM***, la misma está formada por varios **elementos**, dispuestos en **forma de árbol**, que conforman una página, entonces una característica deseable en nuestro lenguaje a crear sería tener una manera de **definir estos elementos**, **asignarles atributos** que los afecten e **indicar claramente cuál es su contenido** o sus subelementos, con esta información la herramienta traductora podría hacer una traducción casi directa al formato de salida, por lo sería óptimo que nuestro lenguaje siguiera una sintaxis similar.

Por otro lado, para la parte dinámica, el lenguaje debe proveer una sintaxis que permita invocar a los **constructores o directivas** con sus **argumentos**, a partir de la cuál la herramienta de traducción va a realizar una interpretación de estas instrucciones y con la información del modelo de datos, **generará el código de nuevas estructuras** no presentes en la template o **modificará los atributos de manera dinámica** de alguna estructura ya presente.

2.2 Diseño del intérprete

El apartado más difícil es desarrollar la herramienta para interpretar este lenguaje: **un motor o traductor de *templates***, que se encarga de **procesar los archivos** y **completar** donde haga falta **con la lógica y el modelo** de la aplicación de manera acorde para generar la página web final que debe ver el usuario con toda la estructura, datos y lógica de la aplicación. Podemos usar esta división para ver cómo resolver el problema por partes.

2.2.1 Problemáticas

El primer problema a resolver es el **procesado de archivos**, al cuál parece apropiada una solución en dos partes, primero una lectura del mismo con la creación de un ***árbol de sintaxis abstracta o AST***, a partir del cuál en una segunda pasada se van a procesar sus elementos, los cuáles deberían contener información estática sobre la página y directivas o constructores del lenguaje que dotan a los elementos de funcionalidad o cambian la estructura. La mejor estrategia sería tener un AST lo más similar al resultado final esperado, compuesto de los elementos HTML, sus atributos, sus hijos y en caso de tenerlas, directivas del lenguaje que afecten a la creación de ese elemento o a sus atributos.

Para resolver esta segunda etapa de procesado, está claro que si seguimos la idea de mantener un AST similar al resultado final, traducir la parte **estática** resulta en un **proceso directo**, donde se toma el elemento del AST con su información y se crea el elemento HTML correspondiente. Sin embargo para resolver qué sucede con la componente **dinámica** del lenguaje sería útil separar las directivas o constructores que se proveen en dos: por un lado aquellos con **funcionalidad estructural** y por el otro **los de atributos**, basándonos en el resultado o impacto que tienen en el código resultante.

Por un lado, los **constructores de funcionalidad estructural** se traducen a una nueva estructura de elementos HTML generada de manera dinámica basada en alguna variable del estado de la aplicación, como podría ser una repetición de una misma estructura según los elementos de una lista, la creación o no de una estructura basada en una condición o en la inclusión de otros templates instanciados a modo de componentes, entre otras. Además de la estructura HTML, debería generarse código JavaScript necesario para modificar esa estructura luego de su creación inicial según cambien los valores del estado a lo largo de la ejecución de la aplicación.

Mientras que por el otro, las **directivas de atributos** dan como resultado una instanciación de atributos en elementos ya creados, no se debe confundir esta instanciación con una asignación directa como si se se tratase de tomar el valor del atributo del elemento del AST y se copiarlo al mismo atributo del elemento traducido, sino que pueden llevar un proceso para determinar qué valor tomará, qué atributo será el modificado e incluso incluir, código JavaScript para actualizar el elemento acorde al valor que tome un determinado parámetro a lo largo de la ejecución de la aplicación, un ejemplo de directiva de atributo podría ser una que permita acceder a una variable de la aplicación para tomar su valor de manera dinámica y asignarselo a un determinado atributo del HTML.

Una vez resuelta la traducción de un elemento del AST generado, se debería seguir traduciendo los hijos de manera recursiva y una vez terminado un camino, se debería seguir con su siguiente elemento hermano, realizando así un recorrido en profundidad del grafo para lograr el documento web final.

Asumiendo ya resuelto el primer paso, hay que abordar el segundo, el cuál tiene la dificultad de definir cómo se administra este ***estado***, que se compone de funciones y variables que van a ser instanciadas o utilizadas en los templates para **dotar de funcionalidad** a la página resultante, ya sea para ingreso de datos o el display de los mismos, además se debe resolver cómo la página resultante debería encargarse de mantener siempre una **coherencia entre el modelo y la vista** de una manera optimizada.

Una solución sencilla a la parte de administración del estado parecería ser la creación, previa a la traducción, del mismo con todas las funciones y variables necesarias e ir insertándolas en la traducción final a medida que se recorren los elementos del AST y aparezca una directiva que tenga como argumento algún elemento del estado, claramente, para una instrucción por ejemplo, que permita mostrar el contenido de una variable ya inicializada, esta solución alcanzaría.

Sin embargo, esta solución resulta insuficiente cuando alguna de nuestras directivas debe modificar el estado, agregando un nuevo elemento, como podría ser en el caso de un constructor iterativo. Este constructor iterativo podría querer acceder dentro de cada iteración al valor de un elemento de una lista para mostrarlo como texto, con lo cuál deberíamos tener alguna manera de acceder a esa variable de control, si nosotros tenemos el estado fijo desde un principio de la traducción, esto es imposible.

Por ende una solución óptima para este problema consiste en primero resolver para cada elemento de la template en traducción estas directivas que pueden agregar variables al estado, y luego, realizar la traducción de manera recursiva de los hijos con el nuevo estado como parámetro. Este enfoque también nos ayuda a mantener el ***scope*** de nuestras nuevas variables acorde, tomando de ejemplo el constructor iterativo, en la estructura generada por la primer iteración sólo se podría acceder al elemento de la primer posición del iterable en cuestión y no se podría acceder a ninguna instancia de la variable de control en otro lugar de la template que no sea un elemento del constructor invocado o sus elementos descendientes.

El otro punto importante, hasta ahora ignorado, es la mantención de c**oherencia entre vista y modelo** en todo momento, para esto se debe generar un algoritmo en el proceso de traducción para que la aplicación final vaya revisando el árbol HTML generado inicialmente de manera optimizada durante toda la ejecución, **buscando si hay alguna diferencia con el valor que se tomó de la variable del estado**, si esto es así debe actualizar ese elemento y sus atributos de manera acorde, además de repetir el proceso con todos los hijos de ese elemento. Sin este proceso, la página web resultante no sería dinámica y los valores serían los iniciales durante toda la ejecución

Referido al aspecto de generación de la salida de la herramienta traductora, ya se dió parcialmente una solución ya que se discutió que durante el procesamiento de la template y el enlace de esta misma con el modelo y la lógica de la aplicación se podría ir generando

una salida, elemento a elemento, de la estructura final de la aplicación web: de manera directa en los casos donde no hay constructores que agreguen parte dinámica y con relativa facilidad donde si los hay gracias a las soluciones propuestas. Sabemos que esta salida corresponde a un documento web, por lo que solo puede ser HTML, CSS y JavaScript que son los formatos que reconoce un navegador, pero no se definió cuál es específicamente la manera en la que se debe generar este código.

La idea es que todo el proceso ocurra del lado del cliente y en tiempo real, no preprocesado, por lo que la generación de este código que se muestra al usuario final de la aplicación web tiene que ser llevada a cabo de manera eficiente y además, la parte de JavaScript tiene que entre otras cosas, administrar estado y la actualización del DOM, por lo que la mejor opción sería optar por otra abstracción de tecnologías web como lo son los *frameworks front-end* que proveen herramientas sencillas para las funcionalidades requeridas y lo hacen de una manera optimizada.

2.2.2 Arquitectura

Considerando los pasos que debe seguir la herramienta traductora, los problemas que nos encontramos y las soluciones planteadas, podemos considerar la siguiente arquitectura para el desarrollo de la aplicación:

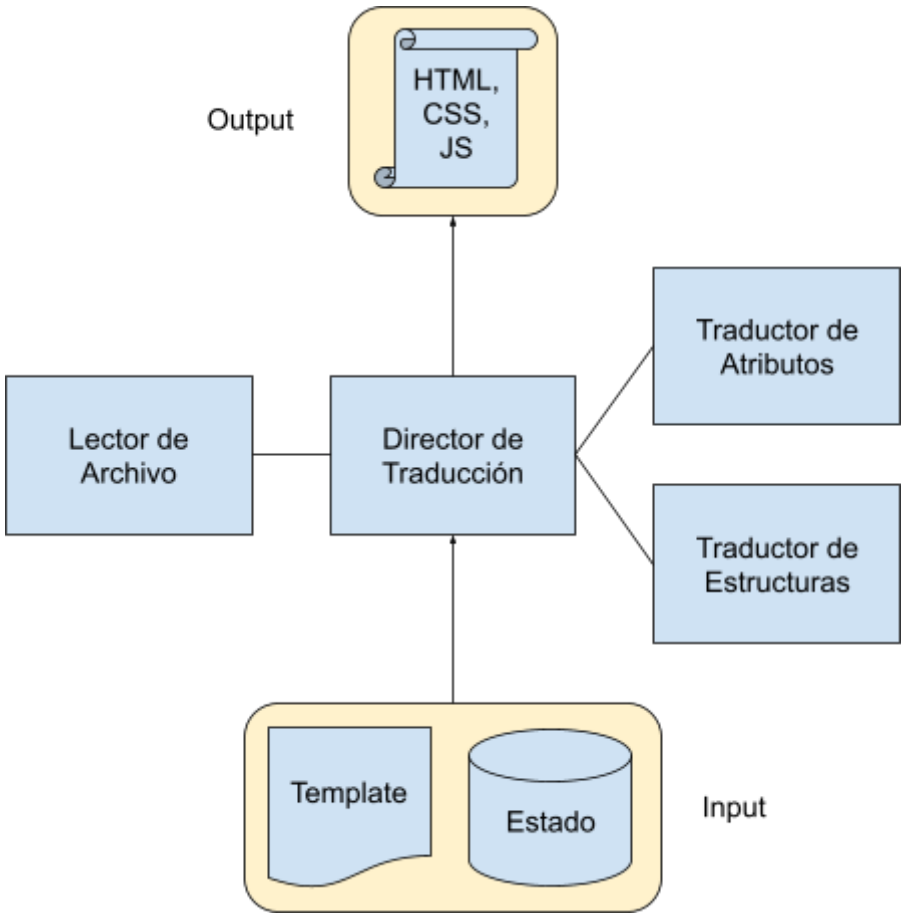

Figura 1: arquitectura de la herramienta traductora

La misma cuenta con un módulo principal, nombrado **Director de Traducción**, que interactúa con el resto de componentes, y además sirve de **interfaz** para nuestra herramienta, recibe la entrada que consta del archivo template y un determinado estado y debe retornar como salida una página web funcional consistente de HTML, CSS y Javascript. Este módulo principal depende de un módulo **Lector de Archivo**, el cuál se va a encargar de **tomar el archivo** de *template* escrito en el lenguaje con las características antes mencionadas, y de **generar** a partir del mismo un **AST**.

El Director de Traducción depende de dos módulos, un **Traductor de Atributos** y un **Traductor de Estructuras**, cada uno separa el trabajo de la traducción de manera acorde a las directivas solicitadas para un elemento. El **Traductor de Estructuras** se encarga de generar una **nueva estructura de elementos** que formarán parte de la estructura del documento web, mientras que la salida del **Traductor de Atributos** se compone de **características** y **funcionalidades** para agregar a estos elementos. Ambos módulos también deberían generar el código necesario para mantener tanto las estructuras como atributos acorde a los valores cambiantes de las variables indicadas inicialmente.

En base al AST inicial, el Director de Traducción debería tomar el primer elemento y llamar tanto al Traductor de Atributos como al Traductor de Estructuras que **en conjunto resolverán la salida para ese elemento**, en los casos que no se vea afectado por ninguna directiva, la traducción es directa.

El Traductor de Estructuras, por lo visto en el paso 2 en el que dividimos el problema que tiene que resolver la herramienta, puede llegar a necesitar expandir el estado que ingresó al traductor, por lo que es en este módulo, que vamos a llamar de **manera recursiva** y **en profundidad** a la resolución de los hijos del elemento del AST que se esté traduciendo, por lo que tenemos una **recursión indirecta cruzada** con el módulo Director de Traducción.

Una vez terminado este **recorrido en profundidad**, la traducción estará completada y el resultado será una página web funcional generada de manera dinámica en base al estado indicado, con la capacidad de también actualizarse a medida que pasa el tiempo en la ejecución.

2.3 En la Práctica

Teniendo ya claro los objetivos a alcanzar, los problemas que hay que considerar al momento de desarrollar, y expandiendo lo comentado en la introducción sobre las tecnologías se propone a continuación una solución concreta para pasar de diseño a implementación en lenguaje e intérprete.

Para la creación del lenguaje de templating que propone esta tésis se va a tomar de ejemplo la sintaxis que utilizan las **plantillas del framework Vue**, del cuál se van a adoptar algunas de las varias funcionalidades de su lenguaje, aunque sea las necesarias para una demo interesante, y también se imitará la estructura de los archivos, siendo estos **documentos tipo XML**, que incluyen el uso de los constructores y directivas del lenguaje como atributos de los elementos en cuestión.

Por otro lado, para la implementación de la herramienta, esta se realizará como una **librería de React** no solo porque queremos agregarle el sistema de templates en lugar de tener que utilizar las herramientas que este nos provee para definir las interfaces gráficas y su interacción, sino que utilizar React nos va a permitir enfocarnos solamente en el desarrollo de la parte de procesado de los archivos, haciéndonos obviar el desarrollo de toda una herramienta personalizada para manejar estado y las actualizaciones dinámicas según estas variables.

3 React

En este capítulo se explicarán los aspectos básicos de **React**, qué es y cómo funciona. Poniendo especial foco en las características que aprovecha nuestra librería para el procesado y traducción de las templates y aquellas que nos ahorran el desarrollo de parte de las características que definimos necesarias para tener un intérprete funcional de nuestro lenguaje de templates.

3.1 ¿Qué es React?

React es una librería JavaScript, no un *framework*, utilizada para desarrollar y mantener interfaces de usuario basadas en componentes, es open-source y fue creada y es mantenida por Meta (antes Facebook).

Si bien de manera general, por el uso dado, se lo considera un *framework front-end*, esto no es técnicamente así ya que React vanilla no exige una estructura o patrón rígido de archivos para desarrollar una aplicación completa con el, se puede utilizar en otros entornos que no sean un navegador y carece de gran parte del ecosistema de herramientas necesarias como podrían ser ruteo, manejos de formularios, entre otras, permitiendo al desarrollador elegir entre varias librerías de terceros para incluir estas funcionalidades, dando más libertad pero perdiendo la base sólida para arrancar que dan los frameworks, que no solo facilita trabajo sino que también logra uniformidad entre distintas aplicaciones, lo cuál es útil para la reutilización de código.

De todas formas, **React** en conjunto de otra librería llamada **ReactDOM** se utiliza para resolver los mismos problemas que resuelve un *framework front-end*, e incluso tenemos opciones como *Next.js* o *Remix*, que si son *frameworks* basados en React que incluyen todas las funcionalidades necesarias para ser considerados como tal. Teniendo todo esto en cuenta, cuando se hable de React como *framework* hay que entender que es de una manera coloquial y puede incluir conceptos de ReactDOM también.

3.2 Componentes

Las aplicaciones React se estructuran por **componentes.** Un componente se define como una **pieza de la interface gráfica** que tiene su **propia lógica y apariencia**. En términos de código, un componente es una función JavaScript que retorna un objeto de tipo **React.FC**, donde **FC** significa ***function component***.

3.2.1 Declaración

Para crear estos componentes tenemos dos alternativas: **JSX** o una llamada a la función **createElement(..)**

La primera opción, **JSX** hace referencia a *JavaScript "eXtended",* esta consiste en un formato que permite escribir **dentro de archivos de JavaScript árboles DOM en una sintaxis tipo-XML**. Esta herramienta no es más que ***azúcar sintáctico***, que luego es ***transpilado*** a una serie de llamadas a la función **createElement(..)** de la segunda alternativa.

Utilizar JSX es sencillo, consiste en un archivo JavaScript normal pero que permite una expresión tipo-XML rodeada por paréntesis que dentro debe tener una estructura con el anidamiento deseado, con la condición de que el elemento *top-level* o padre sea uno solo. Estos elementos pueden ser elementos de HTML u otros Componentes de React, ya sean *built-in,* desarrollados o importados a través de librerías, pueden instanciar sus atributos o como se conocen en los componentes React: *props*.

```
function ComponenteHijo({ prop1 }) {
  const variable = prop1;
  return (
    <div>
      <span></span>
    </div>
  );
}

function Componente() {
  return (
    <>
      <div className="texto-rojo">Hola Mundo!</div>
      <div>
        <ComponenteHijo prop1="hola"></ComponenteHijo>
        <ComponenteHijo prop1="mundo" />
      </div>
    </>
  );
}
```

Figura 2: código de dos componentes básicos, donde uno instancia al otro

JSX no solo permite usar lenguaje de marcado en JavaScript, sino que también te permite dentro de estas estructuras ***"escaparte"* a utilizar JavaScript**, utilizando llaves (*curly braces*). Dentro de estas llaves se pueden ejecutar expresiones, no statements, por lo que podemos acceder a una variable, llamar funciones, utilizar el operador ternario para evaluar condiciones o asignarle una variable a un atributo o *prop*.

En la fig. 3 a continuación se puede ver la utilidad de las llaves dentro del código JSX, pudiendo declarar una interface gráfica dinámica con renders condicionales e iterativos de manera mucho más intuitiva.

```
function ComponenteHijo({ valor }) {
  return (
    <span>
      {valor}
      {valor % 2 === 0 ? "PAR" : "IMPAR"}
    </span>
  );
}

function Componente() {
  const arreglo = [1, 2, 3];
  return (
    <>
      <div className="texto-rojo">Hola Mundo!</div>
      <div>
        {arreglo.map((elemento) => (
          <ComponenteHijo valor={elemento} />
        ))}
      </div>
    </>
  );
}
```

Figura 3: código de dos componentes con expresiones dentro

El ejemplo de la figura 3 muestra un componente (*ComponenteHijo*) que muestra **un número** pasado como *prop* (por parámetro) dentro de un elemento HTML span y **de manera condicional** en el caso de que sea par o impar imprime a su lado texto acorde. El otro componente (*Componente*) **itera sobre un arreglo dentro del JSX**, generando para cada elemento una instancia de ComponenteHijo con el *prop* valor **instanciado** con el valor del elemento.

Visto el poder de JSX, pasamos a la otra alternativa, utilizar la función **createElement**, esta tiene el siguiente encabezado.

```
const element = createElement(type, props, ...children);
```

Figura 4: encabezado de React.createElement

El primer parámetro es ***type***, el cuál podemos instanciar con un *string* de un *tag* común en HTML como puede ser *div* o *img*.

El segundo parámetro es ***props***, el cuál debe ser un **objeto de JavaScript** consistente de **claves** que representan el **nombre del atributo** en cuestión y **valores** que representan los valores que toman esos atributos.

Por último se pueden incluir ***children***, este parámetro usa el *spread operator* de JavaScript, por lo cuál **podemos omitir** este parámetro en el caso de que nuestro elemento no tenga hijos, o **incluir múltiples separados** por comas, en los cuáles por ejemplo podemos llamar de vuelta a **createElement**.

El resultado de la función es un **React.FC** al igual que si escribiéramos JSX.

En base a esto podemos afirmar que las dos opciones son lo mismo, solo que la primera debe pasar por un proceso de transpilación y a uso práctico resulta una abstracción mucho más amigable con el desarrollador. En la figura 5 se muestra de ejemplo una comparativa que genera el mismo output usando JSX y usando las llamadas a la librería de manera directa.

```jsx
import { createElement, Fragment } from "react";

function ComponenteJSX() {
  const arreglo = [1, 2, 3];
  return (
    <>
      <div className="texto-rojo">Hola Mundo!</div>
      <div>
        {arreglo.map((elemento) => (
          <ComponenteHijo valor={elemento} />
        ))}
      </div>
    </>
  );
}

function ComponenteDirecto() {
  const arreglo = [1, 2, 3];
  return createElement(
    Fragment,
    createElement("div", { className: "texto-rojo" }, "Hola Mundo!"),
    createElement(
      "div",
      null,
      arreglo.map((elemento) =>
        createElement(ComponenteHijo, { prop1: elemento })
      )
    )
  );
}
```

Figura 5: comparativa código JSX con llamadas a createElement

3.2.2 Eventos

En la sección anterior se mencionó que tanto los componentes función como los elementos HTML pueden recibir parámetros, en HTML se los conocen como atributos, pero React agrega el concepto de **prop**.

Dentro de estos props, React tiene algunos como **onClick**, **onSubmit**, **onFocus,** entre otros, para poder agregarle a los elementos HTML resultantes funciones conocidas como **manejadores de eventos** que son llamadas en respuesta a eventos o *triggers* como clics, *hovers*, o *submits* sobre el elemento en cuestión.

Para manejar eventos entonces necesitamos crear un manejador e instanciar el *prop* asociado al evento de manera correcta.

Para crear estos *event handler* se declara una función con identificador comenzando en *handle* que haga lo que queramos que pase ante el evento.

```
function Boton() {
  function handleClick() {
    alert('Hola me clickeaste!');
  }

  return (
    <button onClick={handleClick}>
      Click acá
    </button>
  );
}
```

Figura 6: ejemplo de manejo de eventos

Luego para asignarle el manejador de eventos al evento hay que hacerlo pasando el nombre de la función, no llamándola, en la figura 7 se muestra la manera correcta de instanciar el prop.

```
// Esto está mal, se llama a handleClick y se pasa por parametro
// el resultado de la función
<button onClick={handleClick()}>Click acá</button>

// Manera correcta, se está pasando una función con nombre
<button onClick={handleClick}>Click acá</button>

// Alternativa correcta, estamos pasando una funcion anonima
<button onClick={() => {handleClick();}}>Click acá</button>
```

Figura 7: instanciación de manejador de eventos

3.2.3 Hooks

Dentro de estos componentes podemos llamar a funciones conocidas como ***hooks***, estas le permiten a los desarrolladores aprovechar muchas características de React que antiguamente requerían declarar los componentes en forma de clase y no de funciones, y estos componentes debían implementar funciones de la interface Component que determinaban **cómo se comportaba el componente en cada etapa de su ciclo de vida**. El detalle de cómo funcionan tiene que ver principalmente con la ejecución de la aplicación, por ende va a ser tratado más adelante, en la *sección 3.3*, de momento abstraigamonos de cómo funcionan y concentrémonos en la funcionalidad que estos nos otorgan.

Los ***hooks*** son funciones con identificador comenzando en *use,* tenemos varios *built-in* y podemos construir a partir de estos nuestros propios hooks. Tienen **dos reglas** importantes: se tienen que usar **solo en componentes**, no en otras funciones, y dentro de estos componentes solo **se deben usar en *top-level,*** llamarlos dentro de loops o condiciones puede generar efectos no deseados por como funciona React.

Para la implementación propuesta en esta tesis se necesita aprovechar las funcionalidades de dos hooks built-in, **useState** y **useEffect**, procedemos a explicar cómo se pueden aprovechar y para qué nos sirven.

3.2.3.1 Hook useState

Los componentes **necesitan actualizar** lo que está en pantalla en base a interacciones, como puede ser al clickear en un *carousel* de imágenes y también **necesitan recordar** información, como por ejemplo el valor de un campo de texto. Para lograr guardar esta información específica a un componente se usa un concepto llamado **state**.

Por ende toda variable que queramos que afecte a las actualizaciones de UI o que nos interesan conservar en la ejecución del componente deben ser creadas utilizando useState.

```
const [stateVariable, setStateVariable] = useState(defaultValue);
```

Figura 8: uso de *useState*

Como se puede observar en la figura 8, la llamada a useState retorna un arreglo de 2 componentes:

- el primero, el **valor de la variable** en el estado
- y el segundo, la **función setter**, a través de la cuál se va a actualizar el valor de la variable en el estado y React va a saber que necesita actualizar la UI

Además la función toma como parámetro el **valor inicial** a asignar a la variable de estado.

```jsx
import { useState } from 'react';

function Contador() {
  const [contador, setContador] = useState(0);

  function handleClick() {
    setContador(contador + 1);
  }

  return (
    <button onClick={handleClick}>
      Clickeaste {contador} veces.
    </button>
  );
}
```

Figura 9: ejemplo de contador usando *useState*

En la figura 9 se muestra un ejemplo básico de cómo se puede usar ***useState*** para hacer un contador, como vemos, la segunda componente del *array* es el nombre del *setter*, que es solamente una función de un parámetro que cambia el valor al cual podemos acceder referenciando la primera componente con el valor que mandemos como parámetros.
En este caso el *setter* suma 1 a la variable del estado, este setter es llamado dentro de la función *onClick* del botón. React actualiza el texto del botón cuando el valor de la variable contador cambia.

3.2.3.2 Hook useEffect

Los componentes pueden necesitar **sincronizarse con un servicio externo**, por ejemplo llamar a una base de datos o hacer una *request* HTTP y el resultado de estos servicios puede querer ser mostrado, por ejemplo reemplazando una página que simplemente mostraba la leyenda *"Cargando…"*. La particularidad de estas llamadas es que no son causadas por un evento, que es directamente una acción del usuario como presionar un botón o completar un formulario, sino que son ***efectos secundarios*** causados por el simple renderizado de un componente.

Debido al funcionamiento de los componentes de React, cómo se define qué se muestra en pantalla y cuando se actualiza, para lograr el efecto deseado tenemos que utilizar el hook ***useEffect***.

```jsx
useEffect(setup, dependencies?)
```

Figura 10: encabezado de *useEffect*

En la figura 10 se muestra el encabezado del *hook*, que como tal debe ser instanciado siguiendo las dos reglas de los *hooks*. Esta función no retorna nada y toma 1 o 2 parámetros:

- El primero corresponde al **efecto** que queremos lograr, es decir debemos completar con una función que realice una acción, también llamado ***set-up***, y que retorne una función de ***clean-up***, es decir que desconecte o limpie esa interacción externa para evitar posibles problemas en una llamada siguiente al efecto.

- Mientras que el segundo, que puede no estar presente, corresponde a un **arreglo de dependencias** que determinan cuándo se va a ejecutar el efecto.

La función de *set-up* va a ser ejecutada en la primera aparición del componente, y luego cada vez que se ejecute el efecto va a ser limpiada la ejecución anterior con el código de clean-up y luego ejecutado el *set-up*. Al terminar la existencia del componente se llama por última vez a la función *clean-up*.

Tenemos tres opciones para elegir en qué momento se va a ejecutar el efecto, veamos cuáles son y cómo se logran:

Primero tenemos que se ejecute siempre que se actualice la interface del componente, esto se logra sin instanciar el segundo argumento de *useEffect*.

```
import { useEffect } from "react";

function Componente() {
  useEffect(() => {
    // Este código se ejecuta en cada re-render
  });
  return <div />;
}
```

Figura 11: *useEffect* sin dependencias

Escribiendo el código anterior logramos tener un efecto que **se ejecuta siempre que se actualice la interface gráfica**, pero este puede no ser el resultado deseado. Por ejemplo podemos tener un cambio de una variable de estado que no tenga nada que ver con el efecto, pero como esta fuerza una actualización de la gráfica, el efecto se va a volver a ejecutar

Para eso es que podemos **completar el parámetro de dependencias** con un arreglo. En este arreglo podemos completar con las **variables que deberían forzar una ejecución** solo **en el caso que hayan cambiado** desde la última vez que se actualizó la interface, si esto no fue así, el código de *set-up* no se ejecutará.

```
import { useEffect } from "react";

function Componente({ parametro }) {
  useEffect(() => {
    if (parametro) {
      // ...
    } else {
      // ...
    }
  }, [parametro]); // Se va a ejecutar cuando se actualice parametro
  return <div />;
}
```

Figura 12: *useEffect* con dependencias

En la figura 12 vemos un ejemplo que da como resultado la ejecución de un código que podría determinar una llamada a un servicio externo u otro en base a un parámetro, este código, incluída la evaluación de la condición, solo se va a ejecutar si el parámetro cambió. Cabe aclarar que el arreglo debe tener **TODAS** las dependencias que utiliza la función declaradas, caso contrario la aplicación no se va a ejecutar.

Por último, si no tiene dependencias y queremos lograr un efecto que se ejecute una sola vez, ni bien es mostrado el componente y que no se vuelva a re-ejecutar basta con instanciar el parámetro con un arreglo vacío como se muestra en la figura 13.

```
import { useEffect } from "react";

function Componente() {
  useEffect(() => {
    // Este codigo solo se va a ejecutar una vez
  }, []); // Se ejecuta al inicio del componente
  return <div />;
}
```

Figura 13: *useEffect* con dependencias vacías

3.3 Ejecución y funcionamiento

Ya vimos las características más interesantes de React y también cómo utilizarlas para poder lograr crear interfaces gráficas interactivas. Sin embargo, en muchas de las explicaciones quedó abstracto cómo es que se logra el funcionamiento, en esta sección vamos a ver cómo estos componentes con sus *props* y sus *hooks* son mostrados al usuario y actualizados de manera acorde, cómo es el flujo de ejecución que sigue una app con esta librería y cómo abstrae el manejo del *DOM*.

Primero, veamos cómo es que podemos mostrar nuestros componentes en una aplicación web con React. Tenemos que tener un *index.html* y un *index.js*.

En el **index.html** debemos incluir un elemento HTML con un atributo **id**, llamemoslo ***root*** para este ejemplo, incluir el script *index.js* y debemos en algunos casos incluir tanto el *script* de React como de ReactDOM.

En el **index.js** debemos importar la librería de **ReactDOM** y llamar a la función **createRoot** con parámetro que referencie al elemento con id *root* que declaramos en HTML y sobre el resultado de esta función llamamos a la función **render** con nuestro componente principal como parámetro.

Si utilizamos paquetes o frameworks como *create-react-app*, *Next.js* o *Vite* este proceso antes mencionado ya viene hecho, tenemos una estructura de archivos del proyecto prearmada y solo tenemos que completar los .jsx con nuestros componentes y su interacción.

index.js

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

index.html

```
<!doctype html>
<html lang="en">
  ...
  <body>
    <div id="root"></div>
    <script src="/index.js"></script>
  </body>
</html>
```

Figura 14: archivos principales de una aplicación web React.

A partir de ese elemento HTML con id *root* y con la ejecución del método *render*, **ReactDOM** se va a encargar de manipular el DOM.

Antes de explicar cómo funciona el método **render**, el cuál toma un componente como parámetro, veamos cúal es específicamente el objeto que retorna una función componente de React. El retorno de un componente, como se puede ver en la figura 15, tras convertir el JSX de la función a llamadas a *createElement* es un elemento o instancia de otro componente React, con su tipo y sus props, los cuáles incluyen el *prop* especial *children* que puede contener un elemento o un arreglo de elementos formando un árbol.

```
function App() {
  return (
    <div>
      Componente
    </div>
  );
}
```

```
const returnApp = {
  $$typeof: Symbol(react.element),
  key: null,
  props: {children: "Componente"},
  ref: null,
  type: "div",
}
```

Figura 15: componente de aplicación y valor que retorna

La función **render** en conjunto con el componente principal y todos los componentes hijos del mismo terminan armando un árbol de elementos o instancias de componentes de React conocido como **Virtual DOM**. Este **VDOM** es una abstracción ligera, mantenida en memoria, de lo que finalmente se debe mostrar en la página, es decir el DOM HTML, la librería **ReactDOM** realiza una traducción completa, manteniendo sincronizado el primero con el segundo, este proceso es costoso pero no hay manera de evitarlo por lo menos en el render inicial.

Cómo dijimos antes, el **VDOM** se compone de elementos y de instancias de componentes, estas últimas incluyen la estructura de elementos que lo componen y se va a mostrar por pantalla y además poseen un estado interno y manejan su propio ciclo de ejecución, los cuáles son modificados utilizando los hooks explicados en la sección anterior como useState o useEffect. Estas instancias de componentes son llamadas, ya que en sí son funciones, en cada render.

Hasta ahora lo único que sabemos que hace la función *render* es generar el VDOM y traducirlo al DOM para mostrar la página inicial, sin embargo, si solo hiciera esto no tendríamos una aplicación web interactiva, para esto tenemos la **interacción de React con ReactDOM**, ya se mencionó que una función componente no solo retorna elementos, sino que puede ejecutarse y dentro de esta ejecución tenemos los *hooks* y si entramos en el código de estos, tenemos en su funcionamiento **llamadas a un renderer** abstracto, que como estamos trabajando con React en el contexto de una aplicación web, este resulta ser ReactDOM. Al momento de hacer las llamadas para crear el VDOM, se instancia una **referencia al renderer** en los componentes, para que estos puedan **solicitar las actualizaciones cuando sean necesarias**.

Supongamos que tenemos un árbol con una cantidad de elementos grande, y el último elemento decide solicitar un *re-render* para cambiar el más mínimo detalle, primero de **manera rápida y poco costosa** se va **generar un nuevo VDOM**, pero luego **redibujar todo el DOM** para que este coincida con el nuevo VDOM resulta en **muy costoso y lento**, por lo que React utiliza un algoritmo conocido como ***diffing***, el cúal busca la cantidad más pequeña de operaciones para modificar el DOM y que coincida con su versión virtual nueva.

Si hiciéramos este *diffing* de manera directa, para un árbol de n elementos, tendríamos un algoritmo de **orden n^3**, lo cuál es prohibitivo, sin embargo React logra utilizando **2 heurísticas**, reducir el tiempo del algoritmo a **orden n**.

Primero, React asume que **si el componente o elemento cambió de tipo, todo el árbol que sigue se tiene que re-renderizar**, en el caso contrario se comparan los atributos o *props* y se ejecutan las operaciones para actualizar los que deban hacerlos.

La segunda heurística es que los elementos deben tener un **identificador único**, esto se debe a que actualizar la lista de hijos de un elemento para que incluya un elemento al final es un proceso poco costoso, sin embargo, si nosotros no identificamos a los elementos podríamos generar una actualización innecesaria al agregar un elemento nuevo al principio.

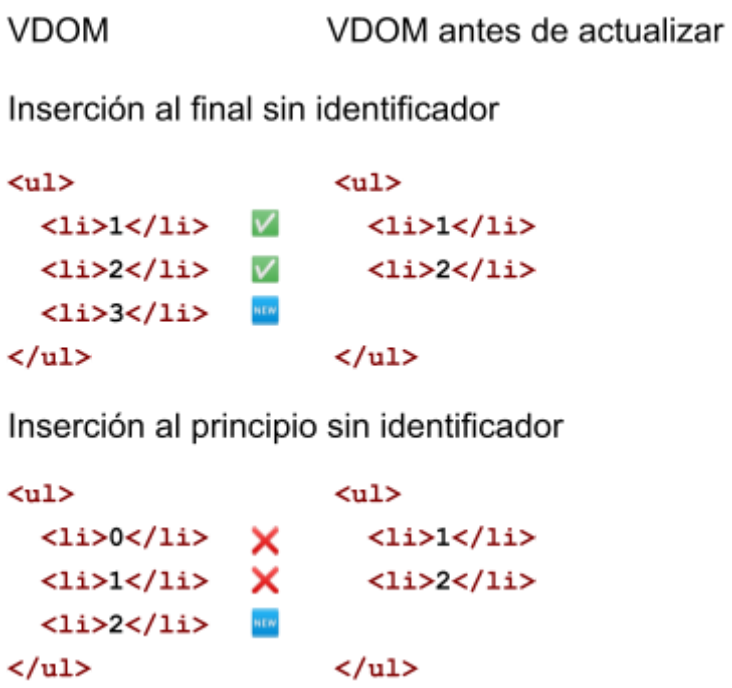

Figura 16: inserción de un elemento en lista sin identificadores

Como podemos ver en la figura 16, si insertamos un elemento nuevo al final de una lista de hijos, es directa la comparación con los elementos que ya estaban y no son modificados, simplemente se agrega en el DOM el nuevo elemento. Sin embargo, si agregamos un elemento al principio, en la comparación el 0 del VDOM es distinto del 1 en el VDOM anterior, el 1 del 2 y el 2 no tiene ningún elemento con el cuál compararse, por lo cúal se van a hacer operaciones para modificar o crear los elementos para que existan el 1 y el 2 cuando ya estaban, solo que en otro orden. Si buscáramos en la lista del VDOM nuevo las coincidencias con el VDOM anterior tomaría un tiempo de cantidad de operaciones n^2.

El uso de un identificador único le permite a React **representar los hijos de un elemento como una *tabla hash*** y no una lista, logrando así **tiempo constante** para comparar los elementos, evitar perder el tiempo **buscando las coincidencias** comparando las dos listas y con estas coincidencias ahorrar las operaciones innecesarias del DOM.

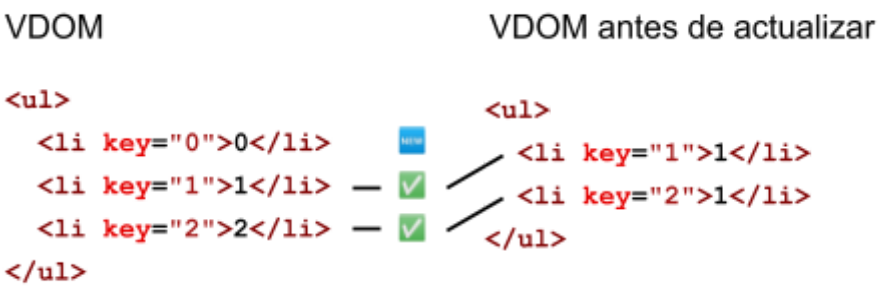

Figura 17: inserción de un elemento en lista usando identificadores.

Como se ve en la figura 17, React busca en el VDOM en tiempo constante la existencia de un elemento con el identificador correspondiente y compara si es nuevo o hubo cambios, en caso de que no, no se ejecuta ninguna operación costosa del manejo del DOM.

Una vez hecho este *diffing*, el DOM y el VDOM nuevo coinciden y ReactDOM queda a la espera de más llamadas de los componentes para actualizar la página.

4 Vue

En este capítulo se introducirá brevemente el ***framework* Vue**, detallando en qué consiste, cómo funciona y explicando algunas de las soluciones que nos ofrece su lenguaje de templating para tomarlas de inspiración y luego adaptarlas a la implementación que propone esta tésis.

4.1 ¿Qué es Vue?

Vue es un *framework front-end* de JavaScript para construir interfaces gráficas de usuario y crear aplicaciones web single page. Fue creado por Evan You tomando como inspiración sus años de experiencia en **Angular**, otro framework frontend, del cuál tomó las características que más le gustaban y le parecían útiles para crear un **producto más ligero**, la librería es de código abierto y es mantenida por la comunidad.

Vue nos provee de dos funcionalidades *core*, la primera: **reactividad**, al igual que React, Vue tiene una metodología para estar al tanto de los cambios de estado en el código JavaScript de nuestra aplicación y al momento que ocurren actualizar la vista de manera eficiente. La segunda, que es la que nos importa para el desarrollo de la tésis, es el **renderizado declarativo**, Vue extiende HTML con una sintaxis de templates para describir declarativamente la salida HTML basada en el estado de la aplicación.

4.2 SFC y Templates

```
<script setup lang="ts">
  import { ref } from 'vue';
  const contador = ref(0);
</script>

<template>
  <div>
    <span v-text:"contador"></span>
    <button class="boton" v-on:click="contador++">SUMAR</button>
  </div>
</template>

<style scoped>
  boton {
    background-color:red;
    color: white
  }
</style>
```

Figura 18: archivo .vue de un componente contador con sus tres secciones default

En los proyectos generados con herramientas como *Vite* o *create-vue*, la manera de declarar componentes es a través de archivos ***single-file components*** o mejor conocidos como ***.vue**, estos se pueden componer de las 3 partes de una página web: HTML, CSS y JavaScript. Los archivos son **estructurados *tipo-XML***, donde la parte de HTML estará entre tags *template*, la parte de JavaScript entre tags *script* y la de CSS entre tags *style*. Cabe destacar que la parte de JavaScript es posible omitirla, permitiendo así declarar archivos que solo representen *vistas* o *templates* y no componentes. Además es posible declarar otras secciones.

Esta sintaxis de template no es más que una **extensión a HTML** que permite *bindear* la interface gráfica en el DOM con las variables de la instancia del componente, esto lo logra permitiendo dictar **directivas** a los elementos a través de atributos con nombres especiales que pueden recibir como valor expresiones de JavaScript. Detrás de todo esto, Vue lo que hace es **compilar estas directivas a código JavaScript optimizado** y junto con el sistema de reactividad mencionado anteriormente lograr las actualizaciones del DOM ante un cambio de estado de la manera óptima.

4.2.1 Directivas

Vue utiliza **directivas** para determinar dónde se deben aplicar de manera reactiva actualizaciones al DOM cuando el valor de su expresión cambia. Como se muestra en la figura 19, una directiva en Vue es un atributo de elemento HTML donde su identificador tiene 3 partes y su valor corresponde un *string* que contiene una expresión JavaScript.

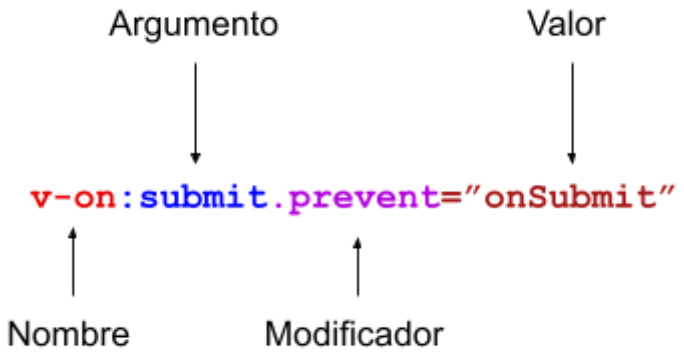

Figura 19: sintaxis de una directiva en Vue

Dentro de esas 3 partes restantes de la directiva, tenemos la primera que corresponde al **nombre**, este indica que directiva se va a ejecutar, luego, tenemos separado por dos puntos un **argumento**, el cuál puede estar o no y representa un parámetro que puede utilizar la directiva, por ejemplo: tenemos la directiva **v-bind** que permite asociar un atributo HTML a una variable para que cuando esta cambie también lo haga el atributo, el argumento representa el nombre del atributo mientras que el valor la variable a asociar, por último, separado por un punto tenemos **modificadores**, que corresponden a pequeñas variaciones opcionales de la directiva. Estas directivas pueden ser creadas por los desarrolladores, sin embargo en esta sección se tratarán algunas de las *built-in* que provee Vue.

Primero una sencilla, la directiva **v-text**, la cuál espera un *string* como valor y se encarga de reemplazar el contenido interno del elemento que recibe la directiva por el texto del valor.

```
<span v-text="contador"></span>
```

Figura 20: elemento span con directiva v-text

En la figura anterior tenemos un *span* que va a cambiar de manera reactiva el valor de texto que muestre según vaya aumentando el valor de la variable *contador*. En este caso tenemos una variable, sin embargo recordemos que el valor que toma la directiva tiene que ser una expresión JavaScript, para ésta en particular el valor debe ser tipo *string*, por lo que podríamos tener incluso concatenaciones o utilizar operadores ternarios que decidan por un string u otro en base a una condición.

Otra directiva interesante para analizar es **v-for,** la cuál sirve para renderizar un elemento múltiples veces basado en los ítems de un arreglo, mapeo u otra estructura iterable. Esta directiva no toma directamente una expresión JavaScript como valor como la mayoría de las directivas sino que define una sintaxis especial que permite declarar un **alias** para poder acceder al valor de la *variable de control* en cada iteración sobre una **estructura iterable** la cuál puede ser indicada por una variable o con el resultado de una expresión.

```
<!-- Sintaxis v-for: alias in expression -->
<div v-for="item in items">
  <span v-text="item"></span>
</div>
```

Figura 21: elemento div con directa v-for, con un acceso al alias.

4.2.2 Slots

En la introducción del capítulo se habló de que Vue permite declarar componentes reutilizables como los de React, estos se instancian de la misma manera utilizando elementos nombrados y se importan en la sección de *script* de los archivos SFC.

Estos componentes permiten al igual que los de React el pasaje de expresiones de JavaScript como *props* o atributos pero también nos interesa pasar o inyectar el contenido de otras templates a modo de fragmento en lugares específicos dentro de otra, para esto existe el sistema de **slots**.

Se puede utilizar el **elemento *slot*** en una *template* para indicar un **espacio reservado** para **renderizar contenido provisto por un componente padre**. El componente padre (App) puede proveer contenido como muestra la figura 22, poniéndolo dentro del elemento que representa al componente hijo (Layout) instanciado. Si agregamos (en la template hijo) más de un elemento *slot,* el contenido provisto por el padre se renderiza en esos lugares también.

Figura 22: uso básico de slot sin nombre

Sin embargo en esa situación estamos limitados, podemos querer tener **más de un *slot* con diferente contenido**, para ello se agrega un atributo ***name*** a los slots, el cuál permite referenciar ese lugar al momento de instanciar el componente. Para poder proveer contenido a un slot nombrado no basta con incluir hijos dentro del elemento componente sino que tenemos que utilizar dentro de este elementos *template* con la directiva **v-slot,** pasándole como argumento el nombre del slot a completar. Dentro de estos elementos *template* con directiva v-slot incluiremos el contenido que queremos proveer al componente en ese lugar. En la figura 23 se muestra un ejemplo básico del uso de slots nombrados.

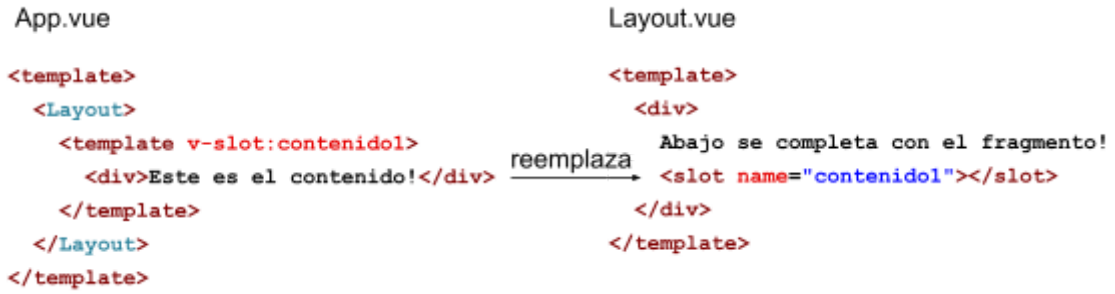

Figura 23: uso de slots nombrados

4.3 Traducción

Al igual que con React usando JSX o cualquier otra herramienta de desarrollo web *front-end* que incluya un tipo de abstracción del lenguaje, sus archivos tienen que ser **traducidos a** los formatos que pueden leer y ejecutar los navegadores: **HTML, CSS y JavaScript**. Para lograr esto, Vue otorga una variedad de opciones acorde a nuestras necesidades de alcance o de optimización.

La manera más común es utilizando herramientas de building como Vite, la cuál además del *scaffolding* de la estructura de una aplicación web Vue, incluye *scripts* para ***buildear*** esta aplicación, en este caso, las templates de Vue son **pre-compiladas** a código JavaScript optimizado y minificado que va a reaccionar de acuerdo al estado de la aplicación.

Sin embargo, si no se está construyendo una *single-page application* o un proyecto muy interactivo se puede optar por usar **Vue sin paso de *building***, para esto, se debe enviar al

usuario final de la página web no solo el contenido necesario para mostrar la misma sino que también el compilador de templates, el cuál pesa aproximadamente 13 kbs adicionales. Este compilador de templates toma la template que ya está en el DOM, la convierte en un string y traduce este último a una función JavaScript de *render* que da como resultado un nuevo DOM reactivo que reemplazará al DOM viejo. Este proceso es más costoso e incluye *overhead*, clara desventaja respecto a la alternativa con *building,* ya que esta se encarga de reconocer las funcionalidades no utilizadas para no enviarle al usuario final la librería completa.

Otra alternativa adicional es utilizar **petite-vue**, que corresponde a una librería aún mas ligera que funciona sin paso de *building*, pero que también ahorra ese paso de la función de *render*, esta opción está ligada al DOM real por lo que para proyectos grandes o que sean multiplataforma resulta limitante.

5 Creación de un Lenguaje de Templates

En el capítulo anterior se introdujeron los aspectos fundamentales de Vue y dentro de estos se hizo foco en el **lenguaje de *templating*** que provee para escribir interfaces reactivas dinámicas de manera **declarativa** para desarrollar aplicaciones web. Tomando como inspiración su formato de archivos SFC, su sintaxis de *templating,* su sistema de *slots* y las directivas *built-in* incluídas en la librería, se propondrá en este capítulo el lenguaje de *template* para ofrecer una manera alternativa a React de declarar sus interfaces y la interacción de las mismas, definiendo su sintaxis, su estructura de archivo para utilizarlo y explicando la funcionalidad de sus directivas.

Para lograr facilidad de uso y de igual manera que se hace en Vue, se propone entonces estructurar el lenguaje de *templates*[1] en estilo **HTML/XML**, decisión también tomada en favor de facilitar el desarrollo de la herramienta de traducción, cuestión en la que se entrará más en detalle en el capítulo siguiente.

Los archivos escritos en el lenguaje tendrán en su más alto nivel solo **un elemento** con tag *template,* si esto no es así, no se considerará un archivo válido, y dentro de este elemento se seguirá con una **sintaxis** muy similar a HTML con algunas **reglas** determinadas y pudiendo realizar llamadas a **directivas** que alteren la vista de la interface de manera dinámica.

5.1 Sintaxis y Reglas

El lenguaje propuesto se puede definir como HTML con ciertas modificaciones a modo de extensión o restricción del mismo pero conservando sus conceptos básicos como el uso de **elementos,** definidos entre **tags** cerrados entre **<...>** o **</…>**, con **atributos** clave-valor.

```
<template>

  ...

</template>
```

Figura 24: estructura de archivo del lenguaje

[1] Cuando se habló de templating en Vue, se trató en el contexto de los archivos single-file component, los cuáles estaban estructurados tipo XML, con secciones como *template*, *script*, *style* y la posibilidad no incluir alguna de las anteriores o de incluir alguna definida por los propios desarrolladores. Sin embargo, para el lenguaje propuesto no vamos a definir otra sección que no sea la de *template*, y vamos a considerar que la parte de código se escribe en React, por este capítulo vamos a asumir la interacción entre las templates y el resto del código de la aplicación resuelta.

Previamente se definió que los archivos de nuestro lenguaje deben contener en su más alto nivel, un elemento *template* como muestra la figura 24. Es dentro de este elemento que se va a declarar la interface gráfica con sus directivas.

Dentro de este elemento *template* se pueden declarar, con cualquier nivel de anidamiento los **elementos HTML** que queramos para estructurar el cuerpo de una interface gráfica como los correspondientes a los *tag div, h1, img*, entre otros. Entre estos elementos, existe la posibilidad de volver a utilizar *template* con ciertos atributos para poder soportar la **composición de plantillas** a modo de componentes, se entrará en mayor detalle en la sección 5.2.2.2.

Dentro de estos elementos podemos tener los **atributos** que queramos, como *href* para agregar un link o *src* para determinar la dirección de una imágen y además en un estilo similar al de Vue, este lenguaje dispondrá de **directivas**, que tendrán el aspecto de un atributo, de hecho se van a instanciar igual, con la salvedad de que el valor de las mismas en la mayoría de los casos hará referencia a alguna variable del estado de la aplicación, con la cuál la directiva ejecutará el código necesario para cambiar ese elemento ante un cambio en el valor de esa variable.

5.2 Directivas

Como se introdujo en la sección anterior, dentro de los elementos podemos utilizar atributos especiales conocidos como **directivas**. La tarea de estas directivas es **cambiar la vista de manera reactiva**, podemos separarlas según el efecto que tienen en ese cambio de vista en dos grandes grupos: de **atributos** o **estructurales**, el primero corresponde a un cambio dinámico de algún atributo del elemento pero conservando este último en la vista, mientras que el segundo grupo corresponde a la creación de nuevos elementos o eliminación de alguno ya existente. Un elemento cualquiera puede tener varias directivas de atributos, pero no puede tener más de una estructural.

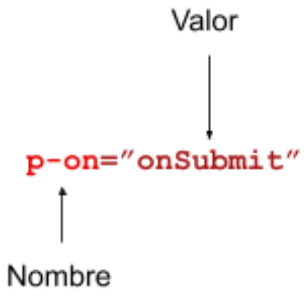

Figura 25: sintaxis de una directiva en el lenguaje propuesto

Independientemente de la categoría a la que pertenezcan, las directivas tienen la misma sintaxis, un **nombre** comenzado en **"p-"** seguido de un identificador representativo de lo que hace la función, este nombre puede contener más identificadores separados por guiones indicando alguna funcionalidad adicional. Luego, como son atributos, las directivas también tienen un **valor**, que a diferencia de Vue, que puede manejar expresiones completas de JavaScript, en este lenguaje se aceptan **literales**, nombres de **variables del**

estado en la mayoría de los casos o se definen **sintaxis especiales** para algunas de las directivas más complejas. Ahora que se definió la sintaxis de atributos y sus generalidades, se verá en detalle las desarrolladas para el lenguaje propuesto.

5.2.1 Directivas de Atributos

Se incluirán en el lenguaje **directivas de atributos** para enlazar variables con atributos HTML, para mostrar variables como texto y para agregar interacción a partir de eventos.

La directiva **p-bind** toma como valor una sintaxis especial que incluye, separado por comas, el **nombre del atributo a enlazar** y el identificador de la **variable** a asignar, tal como se muestra en la siguiente figura.

```
<img p-bind="attributeName,variable">
```

Figura 26: directiva p-bind en un elemento img

Para mostrar una variable como texto dentro de un elemento, se utiliza la directiva **p-text**, a la cuál se le asigna como valor un identificador de la **variable** a mostrar.

```
<h1 p-text="variable"></h1>
```

Figura 27: directiva p-text en un elemento h1

Si queremos mostrar como texto una propiedad de un objeto del estado, se utiliza la directiva **p-text-object**, la cuál permite como valor un identificador de la **variable** del estado de la aplicación y luego de manera opcional, nombres de **propiedades encadenadas por puntos**, para acceder como se hace en JavaScript a las propiedades de un objeto.

```
<h1 p-text-object="variable.prop1.prop1prop"></h1>
```

Figura 28: directiva p-text-object en un elemento h1

La directiva **p-class** es sencilla e interpreta su valor como un ***string***, este *string* contiene por separado los **nombres de las clases** a agregar al elemento. La razón de existir de esta directiva es porque el atributo HTML *class* no funciona por cuestiones de la implementación.

```
<div p-class="class1 class2"></div>
```

Figura 29: directiva p-class en un elemento div

Si queremos asignar clases de manera dinámica, se utiliza la directiva **p-class-bind**, la cuál recibe por valor un identificador de una **variable** que tenga un ***string*** **o arreglo de *strings*** que representen **clases**.

```
<div p-class-bind="variable"></div>
```

Figura 30: directiva p-class-bind en un elemento div

Para agregar una acción que suceda ante eventos que afecten a un elemento se incluyen las directivas **p-on-click**, **p-on-change y p-on-submit,** las 3 toman como valor el identificador de una **función**, a la cuál se va a llamar cuando ocurra la acción. La primera directiva es para aplicar sobre cualquier elemento para llamar una función **ante un click**, la segunda para llamar la función **ante un cambio**, principalmente en elementos de tipo *input*, y la última para llamar una función **ante un *submit*** de un formulario (elemento *form*).

```
<button p-on-click="handleClickFun"></button>
```

Figura 31: directiva p-class-bind en un elemento div

```
<input type="text" p-on-change="handleChangeFun" />
```

Figura 32: directiva p-on-change en un elemento div

```
<form p-on-submit="handleSubmitFun"></form>
```

Figura 33: directiva p-on-submit en un elemento form

Además se incluyen las directivas **p-on-click-prevent**, **p-on-change-prevent y p-on-submit-prevent,** que hacen lo mismo que las 3 nombradas anteriormente con la salvedad de que además de agregar esa funcionalidad ante el evento, desactivan la funcionalidad *default* que tenía elemento ante esos eventos.

5.2.2 Directivas Estructurales

Se incluyen en el lenguaje dos tipos de **directivas estructurales**, unas dependientes del **estado de la aplicación**, mientras que otras están dedicadas a la **componentización** de la aplicación web. El lenguaje restringe su uso a **una directiva estructural por elemento** para simplificar la traducción.

5.2.2.1 Estado de la Aplicación

Primero, se tratarán las dependientes del estado de la aplicación que son aquellas directivas que **cambian la estructura de la interface** gráfica **de acuerdo al** valor de alguna variable del **estado**.

Dentro de esta categoría tenemos la directiva **p-if,** esta se aplica a cualquier elemento y recibe como valor el identificador de una **variable**, esta variable debe tener un valor correspondiente a una **expresión *booleana***, si la misma es **verdadera el elemento existirá** en el DOM, es decir será visible, y **caso contrario** no lo hará.

```
<div p-if="variable"></div>
```

Figura 34: directiva p-if en un elemento div

Además de incluir un **constructor de if** en el sistema de *templates*, es interesante disponer de un **if-else**, este se puede invocar a través de la directiva **p-if-else**, la cuál también toma como valor el identificador de una **variable** que debe corresponder a una **expresión *booleana,*** sin embargo, el elemento al cuál afecta esta directiva debe tener de manera obligatoria **dos hijos**, de los cuales **se mostrará en pantalla uno solo**, el primero, a modo de sentencia *then,* representa el contenido que se va a mostrar si la variable es *true*, mientras que el segundo, a modo de sentencia *else,* si la variable es *false*. Cabe aclarar que estas directivas permiten anidamiento, dejando la posibilidad abierta a utilizar otra directiva **p-if-else** en esos dos hijos.

```
<div p-if-else="variable">
  <div>Then</div>
  <div>Else</div>
</div>
```

Figura 35: directiva p-if-else en un elemento div

Por último, podemos querer *repetir* una estructura dependiendo de una variable, para eso se introduce la directiva **p-for,** la cuál nos permite **iterar sobre una estructura** como un arreglo y poder luego **acceder en cada iteración al valor** correspondiente, esta recibe como valor una expresión con **sintaxis especial** como se muestra en la figura 36, donde nos permite darle un **alias** para referenciar el valor de la iteración dentro del elemento, y luego seguido de la palabra **in,** dar el identificador de una **variable** del estado que represente un iterable.

```
<span p-for="alias in iterable">
  <h1 p-text="alias"></h1>
</span>
```

Figura 36: directiva p-for en un elemento span

5.2.2.2 Componentización

Otorga mucha flexibilidad disponer de una manera de **componentizar** las *templates* que definimos, esto nos permite **separar** por funcionalidades e incluso **reutilizar** partes en otros contextos de manera sencilla. Para lograr esta componentización de una manera similar a la permitida por Vue, se introducen tres directivas: **p-slot**, **p-include** y **p-fill**.

La primera directiva, **p-slot**, permite declarar en una *template*, un **espacio nombrado** por el valor de tipo *string* del atributo, este espacio **debe ser reemplazado por contenido** al instanciar esta *template* en otra. Esta directiva solo se puede declarar en elementos *template,* exceptuando al presente en *top-level.*

```
<template>
  <template p-slot="slotName">
    Espacio a reemplazar
  </template>
</template>
```

Figura 37: directiva p-slot en un elemento template

Para lograr **incluir una template** dentro de otra, se hace uso de la directiva **p-include**, que debe recibir como valor un *string* que corresponde al ***path*** de la *template* a instanciar. Esta directiva al igual que p-slot también solo se puede declarar en elementos *template* que no sean el de *top-level.*

```
<template p-include="template/path/to/file"></template>
```

Figura 38: directiva p-include en un elemento template

Recordemos que los elementos afectados por la directiva p-slot necesitan ser reemplazados por contenido provisto al momento de instanciar la template, para lograr esto, dentro de los elementos *template* con directiva **p-include**, que se encargan de instanciar *templates*, solo pueden existir elementos hijos, de cualquier tipo pero que tengan la directiva **p-fill**, la cuál toma como valor un *string* que debe representar un **nombre de *slot* presente en la *template* incluída**.

```
<template p-include="template/path/to/file">
  <div p-fill="slotName">
    Contenido del slot
  </div>
</template>
```

Figura 39: directiva p-fill en un elemento div

Al momento de la traducción de las *templates* a código comprensible por los navegadores, los espacios definidos con *p-slot* de la *template* traída con *p-include* serán reemplazados por los elementos afectados por *p-fill* tal que coincidan los nombres. A diferencia de Vue y a modo de simplificar la implementación, solo se implementan *slots* nombrados.

6 Implementación del Intérprete de Templates

En el capítulo anterior se definió un lenguaje de *templates* con sintaxis inspirada en la solución de Vue. Para terminar de resolver el problema planteado por esta tesis, resta crear un **motor de templates** que tome archivos escritos en este lenguaje y genere la aplicación web con vista dinámica y reactiva a cambios. Para lograr esto, en este capítulo se propondrá una implementación de una librería de React ***client-side*** para **procesar** estas templates y ***transpilarlas*** a instrucciones React que se encarguen de generar y manejar los cambios de la interface gráfica final acorde al contexto de una aplicación web en la que se decidan instanciar estas *templates*.

6.1 ¿Por qué React?

Existen dos motivos bastante diferentes de por qué se eligió React para la implementación de la herramienta.

El primero quedó claro en el capítulo de introducción, React carece de un sistema de *templates declarativas* para desarrollar interfaces gráficas y en su lugar utiliza JSX, formato que otorga la posibilidad de escribir expresiones en formato HTML con código JavaScript de por medio para agregar reactividad. Se podría decir que más que un motivo, es una motivación ya que el **objetivo perseguido** por esta tesis es agregar esta funcionalidad de *templating* al *framework* React, por lo que decidir resolver este problema con una librería desarrollada para el mismo es una cuestión lógica.

Sin embargo esta elección no solo es una cuestión lógica, sino que también está motivada por el **enfoque de la tesis**, el cuál prioriza el **desarrollo de un lenguaje y su traducción**, si no se utilizara React y se buscara desarrollar un *motor de templating* que devuelva un resultado con una funcionalidad similar, la cantidad de problemas a resolver sería mucho mayor como se vió en la sección 2.2.1, donde se habla de las problemáticas en el diseño arquitectónico de la herramienta traductora. Utilizando React podemos abstraernos de cualquier problema que no sea la traducción ya que nos otorga facilidades como las explicadas en el capítulo 3 para no tener que ocuparnos en desarrollar herramientas para el manejo de estado o para optimizar las actualizaciones de la vista.

6.2 Alcance e Interfaz

El intérprete propuesto se desarrolló en forma de un paquete *Node.js* utilizable como librería en el contexto de una aplicación desarrollada con React. El código de la implementación se encuentra en https://github.com/iamjuanpy/pomjs.

Para poder generar una vista web dinámica y reactiva, como se vió en la sección 2.2, un motor de *templates* necesita de **dos parámetros** igual de importantes, uno, la ***template*** a completar y el otro el **modelo o estado** de la aplicación a partir del cuál **completar los espacios** dejados en la template. Por esto, como se muestra en la figura 40, para instanciar

una *template* del lenguaje propuesto en React, se declara un componente función de tipo PomView de la librería implementada que recibe como *prop* **filePath** un *string* que representa una **dirección** a la cuál buscar la *template* y luego de manera opcional diversos **props nombrados** que referencien a **variables estado** o **funciones**.

```
<PomView filePath={"/template/path"} prop1={prop1} ...></PomView>
```

Figura 40: sintaxis JSX del componente interfaz de la herramienta traductora

Por lo enunciado anteriormente y mirando la figura 41 como referencia, se puede entender el **alcance** que tiene la herramienta traductora implementada, esta se encarga de **descargar** la *template* de la dirección provista, **procesar** el archivo, **enlazando** o **determinando la estructura** con las **variables del estado y funciones** pasadas como *props* al componente ***PomView*****,** para terminar **generando código** correspondiente a un componente React válido.

```
import React from "react";
import { PomView } from "pomjs";

function Component() {
  // Lógica/Modelo del componente (a completar)

  // Vista del componente
  return <PomView filePath={"/template/path"} prop1={prop1} ...></PomView>;
}

export default Component;
```

Figura 41: ejemplo de componente React que utiliza el motor de templates

6.3 Implementación de la Arquitectura propuesta

Siguiendo la arquitectura propuesta en el capítulo 2 para resolver el problema y sabiendo el alcance que tiene la herramienta y qué tareas se delegan a React, se analizarán los archivos del código fuente de la aplicación organizándolos por módulos, detallando la interacción con las herramientas provistas por React y si se utiliza otra librería externa para resolver algún problema en ellos.

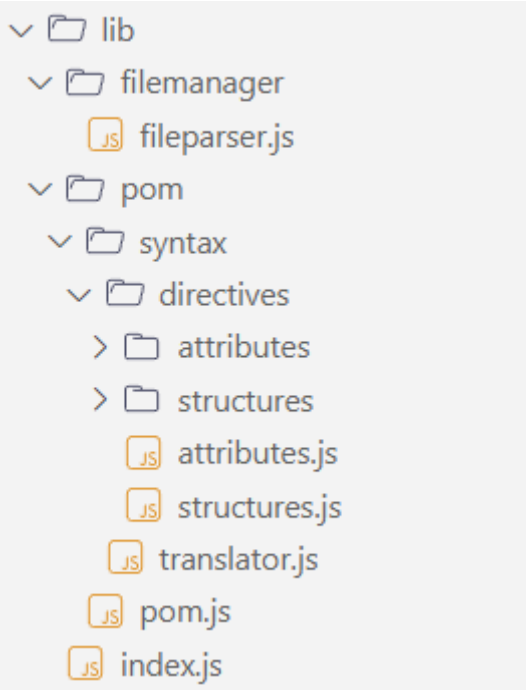

Figura 42: estructura del código fuente

El módulo **Lector de Archivos** se implementó en ***fileparser.js*.**

El módulo **Traductor de Atributos** está implementado en ***attributes.js***, con la traducción para cada directiva en archivos dentro de la **carpeta attributes**.

Sucede algo similar con el **Traductor de Estructuras**, el cuál está implementado en ***structures.js***, con la traducción para cada directiva en archivos dentro de la **carpeta structures**.

Por último el módulo **Director de Traducción** está implementado entre ***pom.js*** y ***translator.js***.

6.3.1 Lector de Archivos

Este módulo corresponde al archivo ***fileparser.js,*** código que tiene una sola función: **parseFile**. Esta función toma como parámetro un *string* que representa el contenido de un archivo del lenguaje y utiliza una librería de *Node.js,* llamada **htmlparser2,** para leer el string pasado por parámetro y convertirlo en un **AST similar al DOM.** Si bien nos abstraemos de cómo se realiza el *parse*, la librería permite **personalización** para determinar el ***output*** de este proceso, por lo que se definió un **AST acorde a las necesidades** del resto de los módulos.

El **AST** que sale de esta función contiene **dos tipos de nodos**, **elementos** o **texto**.

Los primeros, los **elementos**, son objetos JavaScript que tienen un ***type*** que los identifica como elementos, un ***name*** que identifica qué tipo de elemento son acorde al nombre de su *tag* (*div, h1, etc.*), un objeto ***attributes*** que corresponde a pares clave-valor que representan

los atributos del elemento y por último un arreglo ***children*** que va a contener 0 o más nodos del AST. Los nodos **texto** también son objetos JavaScript que tienen un ***type*** que los identifica como tales pero además sólo contienen otra propiedad nombrada ***content,*** la cuál almacena el texto leído.

En este módulo no se hace ningún otro chequeo que no sea propio de sintaxis HTML, así que la salida final de **parseFile,** es un arreglo de nodos del AST.

6.3.2 Traductor de Atributos

Este módulo se encarga de **agregarle funcionalidad a los elementos existentes en la vista**, esto lo hace a través de la definición de *props* en el elemento React final a partir de directivas de atributos.

En el archivo *attributes.js* correspondiente a este módulo se encuentra declarada una función **translateAttribute**, la cuál toma como parámetro un **elemento** del AST a traducir, el **nombre de atributo** a resolver y el **estado** de la aplicación, esta función retorna un objeto con un solo par clave-valor. La idea es que el módulo Director de Traducción llame esta función cuando esté traduciendo un elemento, para todos los **atributos** del mismo que **no sean directivas estructurales**, para poder formar luego un solo objeto con todos los *props* que se deben asignar al elemento React.

Tiene dos alternativas de ejecución esta función, si el atributo traído como parámetro no corresponde a una directiva de atributo, es decir, es un **atributo HTML** como puede ser *class* o *src,* la función retorna el par **atributo-valor** como está ya que no requiere traducción.

El caso contrario es que corresponda a una **directiva de atributo**, por lo que debe existir en un **mapeo nombre de directiva a función traductora**, estas funciones traductoras se encuentran definidas en la carpeta ***attributes,*** reciben por parámetro el **elemento y el estado** de la aplicación y el retorno tiene ser un objeto con un par atributo-valor correspondiente, en la mayoría de los casos se accede al estado para determinar ese valor. Estas funciones pueden notificar un **mensaje de error** a través de *throws,* en caso de que no se respete la sintaxis del valor o no se encuentre una variable en el estado y por ende no se pueda resolver la traducción.

Dentro de esas funciones traductoras tenemos todas las traducciones a las directivas presentadas en la sección 5.2.1, el código de todas arranca igual, buscando dentro del mapeo atributos-valor que tiene el elemento del AST el **valor correspondiente a la directiva a traducir**.

En el caso de la traducción de **p-bind**, como se especificó en el capítulo de definición del lenguaje, el valor debe corresponder a dos *strings* separados por coma*,* uno el **nombre** del atributo a *bindear* y el otro un **identificador** para buscar en el **estado** una variable, si la función no puede interpretar el valor como corresponde notifica un error de sintaxis. Retorna un objeto con propiedad llamada como el **nombre** pasado por parámetro de la directiva, con valor correspondiente a acceder al estado con el **identificador** obtenido.

La función traductora de **p-class-bind** o **p-text** es más directa ya que busca una variable en el **estado** con el **identificador** pasado como valor de la directiva y retorna respectivamente un objeto con una propiedad ***className*** o ***children*** con esa variable como valor. Para **p-class,** es más sencillo ya que esta interpreta el valor como un *string* para asignarle al atributo ***className***.

El caso de **p-text-object** es un poco más complejo ya que el valor de la directiva corresponde a mínimo un **identificador** de una variable del **estado** y luego, opcionalmente, separado por **puntos,** accesos encadenados a **propiedades**. Con el fin de obtener el valor a mostrar, la función intenta separar por puntos el valor de la directiva y a partir del arreglo obtenido va accediendo a una propiedad con el identificador de la iteración actual de un objeto obtenido por la anterior, en el caso base comienza por el objeto estado. Si todo es correcto retorna un objeto con propiedad ***children*** con el **valor** obtenido.

Por último, se tienen todas las directivas que comienzan en **p-on…**, las funciones que las traducen interpretan su valor como el identificador de una **función** en el estado, el objeto que retornan estas traducciones corresponde a una **propiedad de React** para dictar acciones ante eventos específicos con la función correspondiente al identificador como valor. En el caso de estas directivas con sufijo en **…-prevent,** la función traductora interpreta de igual manera el valor, pero a estas propiedades de React retornadas se les asigna una función nueva que antes de ejecutar la determinada por la directiva, **previene el efecto default** de ese elemento.

6.3.3 Traductor de Estructuras

El Traductor de Estructuras tiene la responsabilidad de **generar estructura** de la interfaz gráfica, ya sea aquella que es **estática** durante toda la ejecución de la aplicación como la que es **dinámica o reactiva** junto al código que la crea o modifica **según el estado**.

Dentro de *structures.js* se encuentran definidas dos funciones, una, **isStructureDirective** y la otra, **translateStructure**.

La primera, **isStructureDirective** toma como parámetro el **nombre de un atributo** de un elemento y retorna verdadero o falso si existe dentro de este módulo una traducción posible, es decir, si el atributo **corresponde o no a una directiva estructural**. Este método se usa en el módulo Director de Traducción para decidir si resolver una traducción con este módulo o con el encargado de traducir atributos como se explicó en la subsección anterior.

La segunda función, **translateStructure,** es la que resuelve la traducción de la cuál se debe encargar el módulo, toma como parámetros el nodo **elemento** a traducir, el nombre de la **directiva estructural,** el **estado** de la aplicación hasta el momento y por último los **atributos traducidos** por el módulo explicado en la subsección anterior.

Al igual que la función **translateAttribute,** esta tiene dos caminos de ejecución, si el parámetro del nombre de la directiva estructural no está definido, es decir, el elemento no posee como atributo **ninguna directiva estructural**, la traducción a retornar es la creación

de un **ReactElement** del **tipo** del elemento, con los **atributos traducidos** como *props* y como hijos de este elemento: la llamada de forma **recursiva** indirecta o cruzada al método **translateElement** del módulo Director de Traducción con parámetro cada uno de estos hijos y el estado de la aplicación, este método que será explicado en detalle en la subsección siguiente es el encargado de llamar a los dos módulos de traducción para cada elemento del AST para otorgar el resultado final.

Si el parámetro correspondiente a la directiva estructural **está instanciado**, la función buscará en un **mapeo** nombre de directiva estructural a función, la **función de traducción correspondiente a la directiva utilizada**, la cuál toma como parámetros el **elemento** a traducir, los **atributos ya traducidos** y el **estado** de la aplicación.

Estas funciones se declaran en archivos en la carpeta ***structures*** y deberán retornar un **ReactElement**, además cae en estas funciones la responsabilidad de decidir qué hacer con los **hijos del elemento**, pudiendo optar por no insertar directamente sus traducciones como es el caso de las directivas de componentes, y definiendo qué hacer con el **estado** para las **llamadas recursivas,** ya que como se anticipó en el capítulo 2, una directiva como el *p-for* puede llegar a necesitar modificar el estado antes de la traducción de sus hijos, para que estos accedan a un estado modificado.

Es por estas dos responsabilidades que las llamadas recursivas para resolver la traducción de todo el AST son indirectas o cruzadas, no se podría resolver los hijos en el módulo Director de Traducción y luego resolver la estructura actual si la resolución de la estructura del elemento actual puede afectar las llamadas siguientes. Además de estas dos responsabilidades, estas funciones pueden al igual que las traducciones de atributos, **notificar errores** a través de *throws*.

Al igual que para las directivas de atributos, se definieron todas las funciones que traducen los elementos afectados por las directivas propuestas en 5.2.2.

La función que traduce **p-if** interpreta el valor de la directiva como identificador de una **variable del estado** y si esta variable es verdadera retorna el **ReactElement** correspondiente, con la **traducción de sus hijos** y los **atributos traducidos** por el módulo anterior, caso contrario retorna ***null***.

En el caso de **p-if-else,** la función identifica el valor de la misma manera que la anterior, pero además hace un chequeo adicional para comprobar que el elemento a traducir **solo tenga 2 hijos**, si esto no es así notifica un error, pero si se cumple esta condición, la función traductora retorna el elemento correspondiente con atributos traducidos y dentro del mismo, dependiendo si el valor de la variable es verdadero o falso, la traducción recursiva del primer hijo o del segundo, que representan los bloques *then* y *else* de la directiva.

Para la directiva iterativa **p-for** hay que tener en cuenta varias cosas, primero el valor que recibe debe ser interpretado correctamente y si no respeta la sintaxis como se mostró en la figura 36 del capítulo anterior, notificar el error. Si el valor es interpretado correctamente se tiene un **nombre o alias** para el valor de la variable de control y un identificador para acceder a una **variable de estado** correspondiente a una estructura **iterable**. La función traductora retorna un **arreglo** de **ReactElement**, el cuál es creado recorriendo la estructura

identificada por el valor de la directiva, por cada ítem de la misma, se realiza la traducción a **ReactElement** del **elemento** afectado por la directiva, con los **atributos ya traducidos** por el módulo anterior y la **traducción de sus hijos** como contenido. La particularidad de esta función radica en este último apartado, ya que para cada iteración, la traducción de los hijos se realiza con un **nuevo estado,** el cuál incluye todas las variables y funciones disponibles hasta el momento con la adición del **valor de la iteración actual**, al cuál se va a poder hacer **referencia** dentro utilizando el **alias** proporcionado por la directiva. Por lo hablado en el capítulo de React de los **aspectos de optimización**, si el iterable a recorrer es un objeto se le va a exigir un identificador único, nombrado ***id*** para utilizar de ***key*** para lograr un **tiempo de orden lineal** en la cantidad de actualizaciones de la vista.

Por último tenemos las directivas que nos permiten componentizar las *templates*, donde la traducción que tiene el funcionamiento más complejo es la de **p-include**, esta función interpreta el valor de la directiva como una **dirección** a la cuál buscar la *template* a incluir. Respecto a qué sucede con los hijos del elemento, se exige que estos tengan la directiva **p-fill,** la cuál tiene como valor un identificador del ***slot*** a completar con el elemento afectado, si todos los hijos lo cumplen, serán traducidos sin ser insertados como hijos, sino que serán **sumados al estado actual** utilizando ese identificador como clave. Por último, usando esta dirección y el nuevo estado, se retorna un elemento de tipo **PomView** que traducirá toda la template a incluir encargándose de la inserción de esos hijos agregados al estado.

La función correspondiente a la directiva **p-fill** por el contrario traduce al elemento con sus hijos como si no tuviera ninguna directiva estructural afectandolo. **p-slot** es la que tiene la lógica de reemplazo, la función traductora interpreta su valor como un identificador de la **variable de estado** que tiene el elemento que va a reemplazar el elemento *slot*, esta variable de estado es una de las **generadas por p-include** a partir de los **elementos hijos afectados por p-fill**. Si esa variable existe, la función retorna ese valor que corresponde a un ReactElement. Esta función informa error en el caso que la directiva afecte a un elemento que **no sea un *template*** o que **no esté instanciado el slot** en el estado.

6.3.4 Director de Traducción

Por último se entrará en detalle en el módulo que provee la **interfaz** a través de la cuál se utiliza la herramienta y a su vez se encarga de **orquestar la interacción** con los otros módulos. Ya se vió un poco en la explicación de los otros módulos un poco de esa última característica pero es útil tener una explicación más precisa a modo de conclusión.

Este módulo se compone de código separado en dos archivos: ***pom.js*** y ***translator.js***, el primero enfocado a la interfaz, la interacción con el lector de archivos y también encargado de llamar al segundo que es donde se encuentra todo el código para orquestar la ejecución de los dos módulos traductores.

En *pom.js* se encuentra el único ***export* público** de la librería, la función **PomView**, la cuál sirve para instanciar templates con la sintaxis JSX mostrada en la figura 40. Por lo dicho en la sección 6.2 y el tipo de resultado de la función mostrado en la figura 43, este método es evidentemente nuestro **punto de entrada y de salida de la aplicación**.

```
function PomView({ filePath, ...state }: {
    filePath: string;
    [x: string]: any;
}): React.FunctionComponentElement<{
    children?: React.ReactNode;
}>
```

Figura 43: encabezado con tipos de parámetros y de salida de la función *PomView*

El primer parámetro de entrada que utiliza el método es **filePath,** con el cuál va a hacer una ***request GET,*** la cuál debe retornar un **archivo** escrito en el lenguaje de *template*, este archivo, en forma de *string*, va a ser **leído y convertido** a un **AST** como el que se mostró en 6.3.1 por el módulo **Lector de Archivos** utilizando la función **parseFile**. Una vez resuelto el AST, se procede a ejecutar dos funciones de *translator.js,* primero **checkForTemplateElement** y luego **translateTemplate**.

La primera, **checkForTemplateElement**, toma como parámetro el **AST** provisto por el Lector de Archivos y hace el **chequeo básico de sintaxis** de nuestro lenguaje: que el AST solo tenga un elemento *top-level* y este sea de tipo *template*.

La segunda es la que nos da el componente resultante, si se pasó el chequeo básico anterior, **translateTemplate** se ejecutará para todos los hijos del elemento *template top-level.* Para cada uno de estos hijos, devolverá su traducción a **ReactElement**. Si es texto*,* el proceso de traducción es directo, caso contrario corresponde a un **elemento** y se utilizará la información adicional del mismo, el parámetro correspondiente al **estado** y los módulos **Traductor de Atributos** y **Traductor de Estructuras** para devolver el resultado apropiado.

Como ya sabemos, del módulo Traductor de Estructuras se utilizan dos funciones: **isStructureDirective** y **translateStructure**. Mientras que del módulo Traductor de Atributos solo: **translateAttribute**.

De la información del elemento, la función **translateTemplate** toma la propiedad ***attributes*****,** de la cuál usando la función **isStructureDirective** decide cuáles atributos pueden ser traducidos por **translateAttribute** para lograr que cualquier atributo que no sea una directiva estructural pase por esta función como ya vimos, dando como resultado un objeto que va a corresponder a los *props* del ReactElement final.

Luego, puede haber o no un atributo que sí sea una directiva estructural, si hay se instancia como parámetro en la función **translateStructure** o en caso contrario se llama a la función con ese parámetro *undefined*. Como vimos anteriormente, esta función no solo va a determinar el ReactElement final agregando como *props* el resultado de la función anterior, si no que va a hacer una **llamada recursiva cruzada** a la función **translateTemplate** para traducir los elementos hijos del nodo actual teniendo en cuenta el estado de la aplicación hasta llegar al final de esa rama del AST.

Por último, esta función componente **PomView** retorna como vista un elemento **ReactFragment**, o sea vacío, con un arreglo de hijos correspondientes a las traducciones realizadas por la función anterior de todos esos elementos contenidos en *template*.

7 Conclusiones

A lo largo de esta tesis se introdujeron y estudiaron los conceptos de lenguajes de *templating* y las herramientas para interpretarlos, viendo cómo funcionan ejemplos actuales de los mismos. También, de manera más secundaria pero no menos importante se abordaron de manera general temáticas como el desarrollo web, los *frameworks front-end,* metodologías de diseño, entre otros temas relacionados.

Utilizando todos estos conocimientos, en los capítulos 5 y 6 se terminó desarrollando un lenguaje de *templating* propio junto con su intérprete para utilizar en el desarrollo de aplicaciones web con React, con el objetivo no solo de dotar a esta herramienta de la funcionalidad tratada si no también de ejemplificar los beneficios del enfoque declarativo frente al imperativo, la separación de intereses y el uso de abstracciones en el desarrollo en general. En un aspecto más secundario, otro objetivo perseguido fue poner en práctica o expandir el conocimiento adquirido, correspondiente a diversas áreas de la profesión, a lo largo de varias materias de la carrera, como es el caso de las características de un lenguaje y las maneras de interpretarlos en Lenguajes de Programación y en Compiladores e Intérpretes respectivamente, los fundamentos del desarrollo web en Ingeniería de Aplicaciones de Web o los lineamientos, metodologías y estrategias para desarrollar software de calidad en materias como Tecnología de Programación o Diseño y Desarrollo de Software.

Debido al alcance y enfoque de esta tesis, tanto la creación del lenguaje como la implementación de su intérprete no estuvieron enfocadas en lograr resolver el problema de la manera más óptima, ni tampoco en mejorar el rendimiento de la herramienta subyacente, no obstante, sigue siendo importante recalcar aspectos evidentes del rendimiento y optimización en los que podría haber mejoras o en aquellos que la implementación propuesta sufre en comparación con las alternativas ofrecidas actualmente por React u otras herramientas, por ejemplo:

- Las *templates* no sufren el proceso de *minificación* que sí se aplica sobre los archivos de un proyecto React, proceso encargado de reducir el tamaño de los archivos enviados a los clientes.

- El intérprete es enviado junto con el resto de la aplicación web, en el caso de las templates de Vue, estas se pueden precompilar para ahorrar ancho de banda y tiempo de procesado.

- El intérprete *parsea* la estructura XML del documento a un AST, para luego realizar la traducción, resultando en dos pasadas en tiempo lineal respecto a la cantidad de elementos de la template que se podrían optimizar en una.

Además cómo estos desarrollos propios son en marco de una tesis para concluir estudios de grado, el alcance es limitado y se dejaron pendientes algunos desafíos para el futuro, en cuestión tanto de agregado de funcionalidad como de la búsqueda de mayor robustez ante cualquier uso de la herramienta como la que garantizan las alternativas de uso profesional o

comercial para la resolución de estos problemas. Algunos cambios interesantes o implementaciones relacionadas que se desprenden de los temas de esta tésis podrían ser:

- Cambiar el *fetch* de las *templates* en el intérprete*,* permitiendo personalización sobre dónde y cómo buscar la template, actualmente busca en los archivos públicos del *server* de la página.

- Relacionado al punto anterior, complementar la descarga de las *templates* con un sistema de *caché,* para ahorrar descargas pero siempre teniendo la última versión disponible.

- Mejorar las directivas implementadas, incluyendo más chequeos con sus mensajes de error para no solo lograr mayor robustez sino también permitirle al desarrollador comprender mejor lo que sucede.

- Desarrollar un intérprete de *templates* que permita pre-compilar el resultado.

- Adaptar el intérprete actual a una herramienta adicional para disponer de chequeos de sintaxis en tiempo real en un IDE.

Adicionalmente, y gracias a lo expuesto en el capítulo anterior, se pueden agregar las directivas que uno quiera al lenguaje de manera sencilla, sumándole nuevo código al intérprete siguiendo las interfaces y los lineamientos para las funciones de traducción, teniendo en cuenta las reglas de sintaxis básicas del lenguaje.

Además de todos estos conceptos investigados, el contenido repasado y extendido de las materias de la carrera, el desarrollo de esta tesis de grado supuso no sólo práctica y aprendizaje sobre implementación de software sino también de un proceso de aprendizaje y práctica muy valioso sobre la redacción de un documento y la realización de una presentación, útiles para la vida profesional.

8 Referencias

Introduction to Client-Side Frameworks, Mozilla Developer Network Docs. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction

Frameworks Main Features, Mozilla Developer Network Docs. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Main_features

Web Template System, Wikipedia https://en.wikipedia.org/wiki/Web_template_system

Template Processor, Wikipedia https://en.wikipedia.org/wiki/Template_processor

Domain Specific Language, Wikipedia https://en.wikipedia.org/wiki/Domain-specific_language

Getting Started with React, Mozilla Developer Network Docs. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started

Introduction to React, React Learn https://react.dev/learn/

Advanced React Documentation, React Reference https://react.dev/learn/

Responding to Events, React Learn https://react.dev/learn/responding-to-events

Hooks, React Reference https://react.dev/reference/react/hooks

State: A Component's Memory, React Learn https://react.dev/learn/state-a-components-memory

useState, React Reference, https://react.dev/reference/react/useState

Synchronizing with Effects, React Learn https://react.dev/learn/synchronizing-with-effects

useEffect, React Reference, https://react.dev/reference/react/useEffect

Reconciliation, React Documentation https://legacy.reactjs.org/docs/reconciliation.html

Introduction to Vue, Vue Documentation https://vuejs.org/guide/introduction.html

SFC Syntax Specification, Vue API Documentation https://vuejs.org/api/sfc-spec.html

Template Syntax, Vue Documentation https://vuejs.org/guide/essentials/template-syntax.html

Built-In Directives, Vue API Documentation https://vuejs.org/api/built-in-directives.html

Components Basics, Vue Documentation
https://vuejs.org/guide/essentials/component-basics.html

Slots, Vue Documentation https://vuejs.org/guide/components/slots.html

htmlparser2, npm Registry https://www.npmjs.com/package/htmlparser2