UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DESARROLLO DE UN LENGUAJE PARA LA VISUALIZACIÓN DE ESTRUCTURAS DE DATOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JONATHAN ALEXIS URZÚA URZÚA

PROFESOR GUÍA:
JOSÉ ALBERTO PINO URTUBIA

MIEMBROS DE LA COMISIÓN:
JUAN FERNANDO ÁLVAREZ RUBIO
MARÍA CECILIA RIVARA ZÚÑIGA

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: JONATHAN ALEXIS URZÚA URZÚA
FECHA: 2018
PROF. GUÍA: JOSÉ ALBERTO PINO URTUBIA

DESARROLLO DE UN LENGUAJE PARA LA VISUALIZACIÓN DE ESTRUCTURAS DE DATOS

Las estructuras de datos son la piedra angular en el desarrollo de cualquier algoritmo o programa. El código fuente de un programa suele describir, en esencia, una serie de operaciones y transformaciones sobre distintas estructuras de datos. El código no sólo debe expresar correctamente estas operaciones para que la máquina pueda ejecutarlas, sino que debe ser legible y entendible por aquellos que trabajan con él, tanto añadiendo nuevas funcionalidades como corrigiendo errores.

La motivación principal de esta memoria es mejorar el proceso de documentación de código fuente al incorporar descripciones de las estructuras de datos en juego. Estas descripciones se pueden utilizar posteriormente para la generación de visualizaciones que ayuden a entender de mejor forma el código.

Esta memoria describe el desarrollo de un lenguaje para la visualización de estructuras de datos. Se exploran aspectos teóricos y técnicos detrás de la construcción tanto del intérprete del lenguaje como del motor encargado de generar las visualizaciones. Complementario al intéprete y motor del lenguaje, se desarrolló una aplicación web que permite utilizar el lenguaje para generar visualizaciones de forma interactiva, así como explorar las distintas estructuras soportadas por el lenguaje.

Se concluye publicando la herramienta en Internet y realizando una evaluación con múltiples usuarios, en la que se validó exitosamente la utilidad y facilidad de uso del lenguaje.

Agradecimientos

A mí mamá y hermana. Por ser una fuente inagotable de apoyo, cariño y motivación. Esto es gracias a ustedes. Gracias, mamá, gracias, Catita. Las amo con el alma.

A mi familia, tía Rosa, tío Lucho, primos, primas y amigos. Por siempre creer en mí y querer lo mejor para mí. Por su cariño y apoyo incondicionales.

A mis amigos José Manuel y Paulo, por acompañarme en estos largos años. Sé que no hubiera podido superar los momentos más duros de la carrera de no haber sido por ustedes. Gracias por estar ahí, siempre apoyándome y no dejándome caer. Lo logramos, al fin.

A mis amigos del colegio, y la vida, Sebastián, Esteban y Jeffrey. Por su permanente apoyo. Por una amistad que no depende del tiempo ni el espacio.

A mi amiga Hitomi, por estar ahí siempre, acompañándome en las buenas y en las malas. A mi amigo Kyo y su familia, por apoyarme y recibirme con cariño siempre. A mis amigos Kz y Nico, porque aunque nos separe la distancia, estamos juntos en esto. A mi amiga Ina y su familia, por su cariño tan puro.

A mi hermana Leontina y tía Tina. Por estar junto a mí todo estos años. Por ser un pilar de cariño y apoyo permanente, tanto para mí como para mi mamá y hermana.

A mi tía Lily, tío Lucho, Marco, Rodrigo y familiares, quienes me guiaron, cuidaron y dieron su cariño desde pequeño. Por ser una familia a la que siempre puedo regresar.

A Foris, por una dosis constante de locura, desafíos y genialidad. Gracias por apoyarme durante todos estos años, dándome un espacio para crecer y desarrollarme como profesional y persona. Gracias, Andrés, por enseñarme tanto, por preocuparte y guiarme.

Quisiera poder extender esta lista y agregar a todos y todas quienes, de alguna forma, participaron y colaboraron en esto. Gracias <3.

Finalmente, me gustaría agradecer especialmente al profesor José Alberto Pino. Por permitirme trabajar en esta idea, por sus consejos y apoyo y por tener siempre la mejor disposición para ayudarme.

¡Gracias!

Tabla de contenido

Índice de tablas

Índice de ilustraciones

Índice de códigos

Introducción

Motivación

Al trabajar con código fuente, muchas veces se encuentran trozos de código cuyo funcionamiento no es evidente. Esto, por supuesto, obedece a diversas razones. En principio, puede que la persona simplemente no esté familiarizada con el lenguaje de programación utilizado. También es posible que no esté habituada a los conceptos o estilos de programación empleados en el proyecto. Finalmente, puede que el código no posea comentarios, documentación o convenciones que ayuden a entender su funcionamiento.

No obstante lo anterior, frecuentemente el problema radica en no lograr visualizar o comprender correctamente el flujo del código a medida que el programa se ejecuta. Ante esta situación, se suele recurrir a herramientas de depuración o a diagramar de algún modo los cambios que sufren los datos durante la ejecución.

En efecto, poder visualizar adecuadamente un algoritmo o estructura de datos es clave a la hora de entender una pieza de software. Lamentablemente, los repositorios de código fuente no suelen incluir visualizaciones de sus métodos y estructuras. Sin una visualización de referencia, queda a la interpretación subjetiva de cada persona el cómo visualizar los métodos y estructuras de datos en juego.

Por otro lado, y volviendo al planteamiento inicial, hay un aspecto fundamental que debe ser considerado y es que el código no siempre es escrito por la misma persona que debe mantenerlo. Esto implica cierto tipo de comunicación entre el autor del código y sus futuros lectores. Naturalmente, el código mismo será el medio de comunicación por excelencia a través de su documentación, comentarios y convenciones estilísticas.

Sin embargo, en todo proceso de comunicación existen problemas inherentes a la comunicación en sí. Es probable que lo que el autor haya escrito no necesariamente sea entendido del mismo modo por el futuro lector. También hay que considerar la temporalidad de los eventos. Puede pasar mucho tiempo desde que se escribe el código hasta que alguien más debe entenderlo o modificarlo. Esto podría repercutir en la disponibilidad de conocimiento fuera del código, como por ejemplo, acceso a manuales o a colegas de trabajo que puedan ayudar a entender dicho código.

La motivación personal del memorista es verse, en su labor como desarrollador, enfrentado constantemente a métodos y trozos de código cuyo funcionamiento no es evidente. Incluso

revisando código debidamente documentado no siempre es fácil entender qué es lo que hace un método o una función. En ocasiones, un ejemplo visual con datos concretos podría ser la pieza faltante en el entendimiento. Desde la experiencia del memorista, es raro encontrarse con documentaciones que provean visualizaciones para los métodos y estructuras usadas.

Posibles Soluciones

Una solución muy simple es tomar fotografías de los diagramas o dibujos realizados al diseñar e implementar un algoritmo y guardarlas junto al código fuente, quizá en una wiki u otro medio de documentación. Sin embargo, esto también implica encargarse de mantener al día estos archivos, nombrarlos adecuadamente y preocuparse del almacenamiento y gestión de estos archivos.

Una fotografía podría no ser el mejor formato para almacenar estos diagramas debido a que no siempre son tomadas en las mejores condiciones de iluminación y nitidez. Por otro lado, las fotografías de buena calidad suelen requerir mayor almacenamiento. Si se considera que un algoritmo puede requerir más de una fotografía (por ejemplo, una secuencia de pasos), esto podría volverse rápidamente un problema.

Otra solución sería usar alguna herramienta visual para generar diagramas. Esto remedia gran parte de los problemas asociados a las fotografías, pues estos diagramas suelen ser más entendible y requerir mucho menos espacio de almacenamiento.

Muchos diagramas suelen usar el formato SVG, pues debido a su naturaleza vectorial, permite generar imágenes escalables a cualquier tamaño sin degradación en cuanto a calidad, fenómeno que sí ocurre con formatos *raster*, como JPEG o PNG.

El formato SVG está diseñado para la creación de figuras y elementos visuales de alta calidad y pequeño tamaño de almacenamiento. Hay muchas bibliotecas, por ejemplo D3 [19], que utilizan SVG para sus resultados. Sin embargo, la curva de aprendizaje necesaria para generar una visualización, ya sea con D3 o directamente con SVG, es bastante alta.

Más aún, cualquier solución que involucre la generación de imágenes, también conlleva un costo de gestión y mantenimiento de las mismas. Ya sea junto al código o en algún repositorio de datos externo. Esto, además, genera problemas con el versionamiento, pues mientras el código crece y cambia, estas imágenes o recursos también deben cambiar y ser referenciados correctamente.

Continuando, otra solución muy usada a la hora de querer visualizar el comportamiento de un programa es utilizar entornos de desarrollo integrados (IDEs por sus siglas en inglés) y su funcionalidad de depuración para detener la ejecución del programa en un punto dado. Desde ese punto, se puede inspeccionar el estado de las distintas variables y sus valores en la ejecución. Sin embargo, las IDEs no entregan una visualización, sino una representación de las estructuras más básicas (típicamente las nativas del lenguaje de programación en cuestión). Las sesiones de depuración no suelen terminar con un documento o artefacto que sirva de futura documentación para entender mejor el código fuente. Esto, debido a lo complejo que

resultaría extraer o generar dichos artefactos a partir de los datos ofrecidos por las IDEs en su modo de depuración.

Los lenguajes visuales representan una solución particularmente interesante, pues reflejan de mejor manera ideas más complejas. Un lenguaje visual está diseñado para comunicar de forma efectiva y concisa el conocimiento específico de un dominio. De hecho, cualquier idea que se comunique de manera visual va a requerir que los participantes compartan un mismo lenguaje visual. Si este lenguaje no coincide o presenta diferencias los participantes de la comunicación tendrán que mediar un entendimiento mutuo.

Cualquier instrumento que busque comunicar ideas visualmente debe utilizar un lenguaje consistente y semántico. Esto es especialmente importante si el instrumento busca poder explicar un conocimiento sobre el cual el receptor tiene dudas. Si el lenguaje es ambiguo o poco claro resultará aún más complicado comunicar el conocimiento.

Philip J. Guo, en su proyecto *Online Python Tutor* [32], presenta una herramienta que permite ingresar código fuente escrito en Python y visualizar, paso a paso, su ejecución. Esto reduce enormemente la brecha que existe entre la lectura del código fuente y su estado invisible de ejecución. Esto le permite al usuario entender lo que hace el programa de forma visual e interactiva. Esta herramienta fue especialmente diseñada con fines pedagógicos en el contexto de cursos introductorios de ciencias de la computación. Por ello, se tomó especial cuidado en el lenguaje visual con el que se presenta la ejecución de un programa. Online Python Tutor representa una sólida solución al problema de querer explicar un programa a otra persona. Sin embargo, debido a su orientación más pedagógica, las operaciones posibles en el código están limitadas a las estructuras y operaciones más simples del lenguaje. Por otro lado, el uso de la herramienta exige ingresar a su sitio web, o a otro donde esté incluida una instancia de la aplicación. No es posible descargar la visualización y guardarla en algún repositorio de datos. La herramienta permite generar una URL única. Esta URL permite volver a visualizar el programa en un estado específico. Esta misma URL se podría incluir en la documentación propia del código de fuente para así lograr un enlace entre una pieza de código fuente y su visualización. Sin embargo, esto implica tener que visitar el sitio (a partir de esa URL) cada vez que se quiera visualizar el programa, lo que crea una dependencia dura sobre la aplicación web.

Sin embargo, ninguna de estas soluciones logra conciliar los aspectos detallados en la motivación de esta propuesta. Elaborar diagramas, por muy sencillo que sea, va a involucrar necesariamente tareas de gestión para coordinarse con el código fuente. Además, esto va a requerir que el lector sepa de la existencia de estos diagramas.

Dicho lo anterior, la solución propuesta es un lenguaje de texto plano que pueda ser mantenido junto al código, es decir, versionado, y luego ser interpretado para generar visualizaciones que ayuden a entender mejor su funcionamiento.

Objetivos

Objetivo General

Diseñar e implementar un lenguaje de marcado liviano que permita generar visualizaciones de estructuras de datos.

Por ejemplo, el Código 1 generaría la visualización mostrada en la Figura 1.

```
[1, 2, 3, 4]
[1, 2, 3 !em, 4]
[1, 2, 4]
```

Código 1: Ejemplo de sintaxis para arreglos

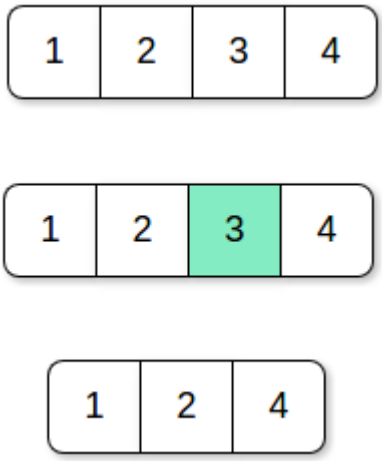


Figura 1: Ejemplo de visualización de arreglos

En este caso, en la segunda línea se puso énfasis en el número 3, lo que se traduce visualmente en un coloreado de la celda respectiva en el arreglo.

Objetivos Específicos

Objetivo	**Forma de medir**	**Entregable**
Definir una gramática para el lenguaje	La gramática puede medirse en función de cuántos aspectos del lenguaje cubre. En específico, se utilizarán las estructuras de datos que soportará el lenguaje como métrica. Así, si la gramática cubre sólo la mitad de las estructuras a cubrir, se dirá que el objetivo se cumplió en un 50 %.	Documento que describa formalmente la gramática del lenguaje.
Construir un intérprete para el lenguaje	El intérprete puede medirse en función de su grado de conformidad respecto a la gramática.	Ejecutable que lea una entrada de texto e imprima un árbol de sintaxis abstracta del lenguaje.
Construir un motor de visualización para el lenguaje	Similar al intérprete, el grado de avance del motor se mide en función de su conformidad al árbol de sintaxis abstracta, en específico, cuántas cláusulas puede representar correctamente.	Ejecutable que lea un árbol de sintaxis abstracta del lenguaje y genere la visualización correspondiente.
Desarrollar una aplicación web que permita utilizar el lenguaje	La aplicación debe ser consistente con el comportamiento del lenguaje (gramática e intérprete) y mostrar correctamente el resultado del motor.	Aplicación web que permita cargar un trozo de texto del lenguaje y genere la visualización correspondiente.

Tabla 1: Objetivos específicos

Capítulo 1

Lenguajes

1.1. Lenguajes y Expresividad

Un lenguaje escrito utiliza símbolos dispuestos en un orden visual con el objetivo de comunicar ideas. Estas ideas son entendidas por otros a través de un proceso de lectura e interpretación de estos símbolos.

Sin embargo, ni los símbolos ni su estructura tendrían utilidad si no hubiera algo que comunicar en sí. Por lo tanto, se puede observar que el lenguaje opera en dos niveles. Por un lado, en lo concreto, a través de manifestaciones físicas (símbolos, sonidos, etc) que hacen referencia a un nivel conceptual. Por ejemplo, se puede utilizar la palabra "casa" para hacer referencia, en el lenguaje español, a la idea de una casa u hogar. Esta misma idea de casa se puede expresar, en japonés, con el kanji 家 (ie).

Por otro lado, el lenguaje debe ser capaz de comunicar las ideas que sus usuarios necesitan. La expresividad del lenguaje es la capacidad de éste para expresar determinadas ideas. Si un lenguaje puede expresar más y de mejor forma estas ideas, entonces se le considera más expresivo. Definir un lenguaje, entonces, implica necesariamente atender el problema de la expresividad.

En japonés existe la palabra "komorebi" que se traduce literalmente como "los rayos de sol que se filtran a través de las hojas". Por supuesto, los japoneses no necesitan decir esta frase cada vez que quieren hablar de esto, ellos tienen una palabra específica para ese significado. Esto conduce a la idea de que en ciertos lenguajes resulta más sencillo comunicar ciertas ideas. Por otro lado, un lenguaje no sólo define lo que se *puede* comunicar, sino también lo que se *quiere* comunicar. Que esta palabra, "komorebi", exista en el vocablo japonés no es una coincidencia, sino que responde a una actividad cultural (hablar de ello).

La expresividad de un lenguaje, entonces, no es una propiedad intrínseca del mismo, sino una propiedad que *emerge del uso* que se le da en un contexto específico.

Por lo tanto, cobra importancia definir qué es lo que se quiere expresar, para así poder

diseñar un lenguaje que efectivamente pueda comunicarlo.

En el contexto de esta memoria, se busca expresar estructuras de datos y sus elementos, así como características o estados presentes en estos datos. En el capítulo 5, *"Estructuras de Datos"*, se revisan múltiples estructuras de datos y se discute la sintaxis para cada una de ellas.

1.2. Lenguajes de Marcado

Los lenguajes de marcado son un sistema mediante el cual se pueden agregar anotaciones o "marcas" a un texto. El Código 1.1 en lenguaje HTML[1] mostra cómo se le puede añadir una semántica a un párrafo de texto y una palabra (enfatizada).

```
1  <p>This <em>word</em> is emphasized.</p>
```

Código 1.1: Ejemplo de HTML

Esta información puede ser rescatada posteriormente a través de un análisis sintáctico del texto.

Como parte de esta memoria se estudiaron algunos lenguajes de marcado con el objetivo de aprender de ellos y así mejorar el diseño del lenguaje en desarrollo.

A continuación, se presentan dichos lenguajes. La estructura de la información es, primero, una breve descripción y luego una serie de observaciones relevantes al objetivo de esta memoria.

setext

setext (*Structure Enhanced Text*) [28] es un lenguaje de marcado liviano cuya primera versión data del año 1992. A diferencia de otros lenguajes de marcado, como HTML, *setext* fue uno de los primeros en no requerir el uso de software especializado para su visualización o interpretación. De aquí nace la distinción de "lenguaje de marcado liviano".

Observaciones: *setext* nació en el contexto de listas de correos con el objetivo de otorgar semánticas y estilo a documentos sin la necesidad de software especializado. *setext* sería la inspiración futura para lenguajes como Markdown, reStructuredText y Textile, entre otros.

reStructuredText

reStructuredText, usualmente abreviado como *RST*, *ReST* o *reST*, es un lenguaje de marcado usado principalmente en el ecosistema de Python para documentación técnica.

[1] Para más información sobre HTML ver el capítulo 3

Aunque *reST* es sólo un lenguaje, está enmarcado en un conjunto de herramientas orientadas a la generación de documentación, similar a JavaDoc en Java.

Observaciones: reST, inspirado por setext, emplea una sintaxis liviana, pero también incluye "directivas", que permiten extender el lenguaje con nuevas funcionalidades sin la necesidad de añadir una nueva sintaxis. Por ejemplo, en el Código 1.2 se utiliza la directiva `replace` para realizar el reemplazo de una cadena de texto por otra en el documento.

```
1 .. |reST| replace:: reStructuredText
2
3 Yes, |reST| is a long word, so I can't blame anyone for wanting to
4 abbreviate it.
```

Código 1.2: Ejemplo de reST

La inclusión de mecanismos para la extensión del lenguaje parece ser una característica importante [50]. Los lenguajes evolucionan según los usos y necesidades de los usuarios, por lo tanto, es importante considerar cómo el lenguaje se podría extender eventualmente para cubrir nuevos usos y necesidades. En particular, si un nuevo caso de uso requiere añadir nueva sintaxis, esto podría generar problemas de retrocompatibilidad. Esto, pues los intérpretes para versiones antiguas del lenguaje no reconocerían esta nueva sintaxis y generarían errores (o actualizaciones forzadas). Un mecanismo de extensión como el descrito permitiría, por ejemplo, la incorporación paulatina de nuevos aspectos al lenguaje (primero como "directivas" y luego como componentes estables del lenguaje).

Finalmente, otro aspecto destacable de *reST* es su objetivo de mantener el marcado independiente de la plataforma o formato de destino. Es decir, el hecho de que un documento vaya a convertirse finalmente en HTML o PDF no debería influir en el tipo de marcado utilizado.

XML

Extensible Markup Language (XML) es lenguaje de marcado que define una serie de reglas para codificar documentos de manera que sean legibles tanto por personas como por máquinas. Es un estándar abierto y ampliamente usado en diversas industrias.

XML es el lenguaje a través del cual se han implementado innumerables lenguajes, tales como XHTML, RSS, Atom, SOAP y SVG.

Un documento XML se estructura en forma de árbol y utiliza etiquetas (como `<question>`) para marcar el texto y darle significado.

El Código 1.3 es un ejemplo de documento en XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<mail>
  <to>USACH</to>
  <from>DCC</from>
  <body>Si este mail te llega abramos una botella de champaa</body>
</mail>
```

Código 1.3: Ejemplo de XML

Uno de los aspectos más destacables de este lenguaje son los conceptos de identificador, clase y atributo.

1. **Identificador**: todos los elementos del documento pueden tener un atributo especial, llamado `id` que permite asignar un identificador único a un elemento. Por ejemplo: `<pet id="mike"></pet>`.

2. **Clase**: en XML un elemento puede tener cero o más clases. Así, una serie de elementos de la misma naturaleza o que compartan algún rasgo, pueden ser fácilmente encontrados o referenciados si se les asigna la misma clase. El Código 1.4 muestra un ejemplo de varios platos de comida, algunos de los cuales están marcados con la clave `veggie`.

3. **Atributo**: un elemento puede tener una serie de atributos con valores asociados. Por ejemplo: `<person gender="female" age="27"></person>`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
  <food class="veggie">
    <name>Strawberry Belgian Waffles</name>
    <description>Light Belgian waffles covered with strawberries and whipped cream</description>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <description>Two eggs, bacon or sausage, toast, and our ever-popular hash browns</description>
  </food>
  <food class="veggie">
    <name>French Toast</name>
    <description>Thick slices made from our homemade sourdough bread</description>
  </food>
</breakfast_menu>
```

Código 1.4: Ejemplo de clases en XML

Estos aspectos sumados a la estructura jerarquizada del documento hacen que XML sea un formato muy fácil de manipular y transformar mediante programas.

XML además está acompañado de una serie de herramientas que permiten, entre otras cosas, validar la estructura y semántica de un documento dado.

MathML

Mathematical Markup Language (MathML) es una aplicación de XML para notación matemática, tanto en estructura como en contenido. Uno de los objetivos de MathML es lograr incorporar fórmulas matemáticas en páginas web y otros documentos.

Observaciones: Un aspecto destacable de MathML es que tiene mecanismos para especificar la forma en que se presenta una fórmula matemática, así como mecanismos para especificar la semántica o significado de la misma.

Por ejemplo, el Código 1.5 muestra cómo la expresión $ax^2 + bx + c$ se puede especificar en MathML "presentacional".

```xml
<mrow>
  <mi>a</mi> <mo>&InvisibleTimes;</mo> <msup><mi>x</mi><mn>2</mn></msup>
  <mo>+</mo><mi>b</mi><mo>&InvisibleTimes;</mo><mi>x</mi>
  <mo>+</mo><mi>c</mi>
</mrow>
```

Código 1.5: Ejemplo de MathML presentacional

En este ejemplo, la etiqueta `<mrow>` es un elemento que comunica la forma en que se dispondrá el contenido (en este caso, en una "fila"). El resto de etiquetas, como `<mi>` o `<mo>`, expresan el tipo de dato especificado (un identificador o un operador, respectivamente).

Por otro lado, el Código 1.6 muestra cómo la misma expresión se escribiría en MathML "semántico".

```
<math>
    <apply>
        <plus/>
        <apply>
            <times/>
            <ci>a</ci>
            <apply>
                <power/>
                <ci>x</ci>
                <cn>2</cn>
            </apply>
        </apply>
        <apply>
            <times/>
            <ci>b</ci>
            <ci>x</ci>
        </apply>
        <ci>c</ci>
    </apply>
</math>
```

Código 1.6: Ejemplo de XML semántico

En este ejemplo, se especifica la semántica de la expresión, que se lee como la aplicación de una suma a tres elementos. Se puede ver cómo esta sintaxis, a diferencia de la anterior, evoca la aplicación de funciones y argumentos.

Resulta destacable la incorporación de elementos semánticos y de presentación en el lenguaje. En el caso de la etiqueta `<mrow>` se esperaría que las implementaciones del lenguaje desplieguen el contenido en una "fila".

CSS

Cascading Style Sheets (CSS) es un lenguaje de estilos utilizado para describir la presentación de documentos en un lenguaje de marcado. Aunque es mayormente utilizado para configurar el estilo visual de páginas web escritas en HTML, el lenguaje puede aplicarse a cualquier documento XML, incluidos, por ejemplo, HTML Y SVG.

CSS emplea selectores y reglas para aplicar estilo a un documento. Los selectores permiten seleccionar o identificar elementos en base a una serie de condiciones. Una vez seleccionados los elementos, las reglas definen qué cambios sufrirán estos elementos en sus propiedades.

Uno de los aspectos más destacables de este lenguaje es su capacidad para referenciar elementos a través de selectores. Esta capacidad deviene de los conceptos de identificador, clase y atributos presentes en XML y lenguajes derivados (como HTML).

1. **Identificador**: CSS permite seleccionar un elemento si se conoce el valor de su atributo `id`, mediante la sintaxis `#valor`. Esto le permite a CSS asignar reglas de estilo

directamente a un único elemento.

2. **Clase**: Utilizando el atributo `class`, una regla CSS puede afectar a todos aquellos elementos del documento que compartan esa clase (ver Código 1.9).

3. **Atributo**: en XML, un elemento puede tener una serie de atributos con valores asociados. El Código 1.7 muestra un documento en el que el elemento `span` tiene un atributo `lang`. Con esto, se podría se podría utilizar el selector `[lang="fr"]` para encontrar todos los elementos, independiente de su tipo, identificador o clase, que contengan el atributo "`lang`" y cuyo valor sea "`fr`".

```
<p>Oh well, <span lang="fr">c'est la vie</span>, as they say in France.</p>
```

Código 1.7: Ejemplo de HTML

El Código 1.8 muestra una regla CSS en la cual se seleccionan todos los elementos de tipo `h1` (encabezados de primer nivel) y se modifica su color de texto a rosado.

```
h1 {
    color: pink;
}
```

Código 1.8: Ejemplo CSS selector h1

En el Código 1.9 se se modifica el tamaño de la tipografía para todos los elementos de la clase `product`.

```
.product {
    font-size: 14px;
}
```

Código 1.9: Ejemplo CSS selector de clase

Estos conceptos (identificador, clase y atributo) permiten **referenciar** elementos. En el caso de HTML y CSS, esto permite una separación de responsabilidades, entre la estructuración y semántica de un documento (HTML) y su presentación (CSS). Dado que en esta memoria se pretende usar SVG como formato de visualización, tiene sentido poder asignar identidad, clase y atributos a los elementos a visualizar para posteriormente aplicar estilos o algún procesamiento.

1.3. Conclusiones y Decisiones

A partir de las reflexiones realizadas y los lenguajes estudiados, se realizan las siguientes conclusiones y decisiones de diseño:

- El lenguaje a desarrollar debería concentrarse sólo en la generación de visualizaciones de estructuras datos y **no** en la visualización de algoritmos o programas. Para visualizar un programa se puede generar un documento, por ejemplo HTML, en el cual se inserten explicaciones y se invoque al lenguaje propuesto múltiples veces. En otras palabras, esta

memoria no busca generar un lenguaje de documentación (como reST o JavaDoc), sino ser una herramienta que permita su utilización en cualquiera de estos lenguajes (y fuera de ellos) para la representación de estructuras de datos.

- Inicialmente, el lenguaje sólo permitirá la representación de **una** estructura de datos. Componer varias estructuras en un mismo documento debería poder realizarse con tecnologías ya establecidas (HTML, por ejemplo).

- Dado que existen muchos dispositivos y tipos de documentos en los cuales se podría incluir una visualización de estructuras de datos, el lenguaje debería mantener una sintaxis que no incluya comandos, parámetros o directivas específicas para alguno de ellos. Esto es, la sintaxis del lenguaje debe ser **independiente del formato** para el cual van a generarse las visualizaciones. Esto no excluye que las implementaciones del lenguaje puedan tener configuraciones propias, por ejemplo, al momento de ejecutar el intérprete o motor de visualización.

- Aunque la sintaxis del lenguaje estará definida, las visualizaciones generadas no deberían considerarse autoritativas o definitivas. Esto, como se mencionó, radica en el hecho de que una idea puede representarse de múltiples formas. Por lo tanto, el usuario final debería poder modificar y/o estilizar las visualizaciones según estime conveniente.

- El lenguaje debería permitir la inclusión de metadata en el documento. Esta metadata puede ser, por ejemplo, utilizada por el motor de visualización para aplicar post-procesamiento.

Capítulo 2

Interpretación

Interpretar un lenguaje es reconocer el significado detrás de una serie de símbolos o vocablos. Esto puede ocurrir en distintos niveles, por supuesto.

Por ejemplo, la cadena de texto "la brisa marina", podría interpretarse de las siguientes maneras:

- La **consonante** "l", la **vocal** "a", un **espacio en blanco**, etc.
- La **palabra** "la", la **palabra** "brisa", la **palabra** "marina".
- El **artículo** "la", el **sustantivo** "brisa", el **adjetivo** "marina".
- *Una brisa, particularmente una brisa originada o relativa al mar.*

Se podría hacer un pequeño cambio al texto, por ejemplo: "la brisa, Marina", el que se podría interpretar como:

- El **artículo** "la", el **sustantivo** "brisa" y el **nombre** "Marina".

Más aún, se podría decir que en la oración anterior "Marina" cumple la función de vocativo, la persona a la cual va dirigida la comunicación, mientras que "la brisa" es aquello que se desea comunicar.

En este capítulo se aborda el problema de que un computador pueda extraer y derivar unidades significativas a partir de un texto de entrada (secuencia de símbolos). Esto con objeto de diseñar e implementar un intérprete para el lenguaje propuesto en esta memoria.

El capítulo está organizado de la siguiente manera: primero se estudian los conceptos de análisis lexicográfico, análisis sintáctico y análisis semántico [52]. Luego se realiza una introducción a las gramáticas libres de contexto [36], para finalizar explorando diversas formas de producir un intérprete.

2.1. Análisis Lexicográfico

En computación, el análisis lexicográfico es un proceso mediante el cual una secuencia de caracteres es convertida a una secuencia de "*tokens*" (unidades léxicas mínimas). Por ejemplo, el siguiente texto:

```
1 + 2 * 34
```

entendido como una expresión aritmética se podría convertir en la siguiente secuencia de tokens:

```
[ (LPAREN,) (NUMBER, 1) (OP, +) (NUMBER, 2) (RPAREN,) (OP, *) (NUMBER, 34) ]
```

Un *token* es un par que asocia un tipo léxico y un valor literal opcional. Sin embargo, esta estructura depende finalmente de la implementación del analizador lexicográfico, o lexer. Por ejemplo, se podría incluir en el token la posición dentro del texto en la cual ocurre (ver Código 2.1).

```
{
    "token": "NUMBER",
    "text": "34",
    "start": 10,
    "end": 12
}
```

Código 2.1: Estructura de datos para guardar un token

Un token no está limitado a la representación de un solo caracter ("+", "1" o "a"), sino que puede abarcar varios de ellos e incluso varias líneas.

Una categoría especial de símbolos corresponde a aquellos que introducen *espacios en blanco*. Los lexers de muchos lenguajes suelen ignorar la mayoría de los espacios en blanco y no generan tokens que reflejen su presencia en el texto de entrada.

Así, por ejemplo, los Códigos 2.2 y 2.3 resultan equivalentes después del análisis lexicográfico.

```
if (a == 2) {
    a = a + 1;
}
```

Código 2.2: Condicion en C con espacios en blanco

```
if(a==2){a=a+1;}
```

Código 2.3: Condición en C sin espaciios en blanco

Sin embargo, no todos los espacios en blanco pueden ser ignorados. El Código 2.4 es válido si se mantienen los espacios, pero el Código 2.5 deja de serlo si se remueven.

```c
int function test() { return 0; }
```

Código 2.4: Functión en C con espacios en blanco

```c
intfunctiontest(){return0;}
```

Código 2.5: Función en C sin espacios en blanco

En este caso el lenguaje exige una separación entre el tipo (`int`), la palabra clave (`function`) y el nombre de la función (`test`). Un *lexer* podría tomar el texto de entrada y reemplazar todas las ocurrencias de múltiples espacios contiguos "␣␣" por un solo espacio en blanco "␣". También se podría realizar una separación del texto en varias partes utilizando como separador un espacio en blanco. Lo anterior se ilustra en el siguiente ejemplo:

```
["int", "function", "test()", "{", "return", "0;", "}"]
```

el *lexer* podría, entonces, operar sobre esta lista de sub-cadenas generando tokens a medida que los encuentra, de izquierda a derecha.

Sin importar cómo se implemente un lexer, su responsabilidad es convertir el texto de entrada en una secuencia de tokens.

Una observación importante en este punto es que el *lexer* **sólo se encarga del reconocimiento de tokens**, no de la **validez sintáctica** ni de que el texto se considere "correcto" (según sea el caso).

2.2. Análisis Sintáctico

El análisis sintáctico, *parsing* o "*parseo*" corresponde al proceso de extraer el significado lógico a partir de una secuencia de símbolos dados. En esta etapa se decide si un texto de entrada tiene sentido desde un punto de vista sintáctico.

Por ejemplo, las siguientes expresiones matemáticas tienen sentido léxico ya que todas ellas utilizan símbolos válidos, no obstante sólo dos de ellas son válidas sintácticamente:

A $(1 + 1) * 2$

B $1 * +45$

C $452 + 2$

D $+ * +(2$

E $7/0$

la expresión **D** aplica incorrectamente los operadores de adición y multiplicación, además de tener mal balanceados los paréntesis. Nótese que la expresión **E** es válida sintácticamente, pues es la división entre dos números, sin embargo, produciría un error al ser evaluada (división por cero).

En computación, el análisis sintáctico suele ocurrir como una etapa posterior al análisis lexicográfico, operando sobre la secuencia de tokens generada por el lexer.

El objetivo de un parser es construir una estructura, por lo general algún tipo de árbol, que represente los elementos sintácticos encontrados. El Código 2.7 es el árbol sintáctico generado a partir del Código 2.6.

```python
if a == 15:
    print("Valid")
else:
    print("Not valid!")
```

Código 2.6: Condición en Python

```
{
    "type": "Module",
    "body": [
        {
            "type": "If",
            "test": {
                "type": "Compare",
                "left": {"type": "Name", "id": "a"},
                "ops": [ {"type": "Eq"} ],
                "comparators": [ {"type": "Num", "n": 15} ]
            },
            "body": [
                {
                    "type": "Print",
                    "values": [
                        {"type": "Str", "s": "valid"}
                    ]
                }
            ],
            "orelse": [
                {
                    "type": "Print",
                    "values": [
                        {"type": "Str", "s": "not valid!"}
                    ]
                }
            ]
        }
    ]
}
```

Código 2.7: AST de Condición en Python

Este tipo de estructura se conoce típicamente como *árbol de sintaxis abstracta* o *AST*. Posteriormente, este AST suele pasarse al compilador para la construcción del programa o directamente a un sistema en tiempo de ejecución.

2.3. Análisis Semántico

El análisis semántico es una etapa posterior al análisis sintáctico en la cual se busca recabar información adicional que permita detectar errores tempranamente. En lenguajes compilados esta etapa típicamente determina el tipo de cada variable y su uso adecuado en distintos contextos. En lenguajes interpretados no suele existir esta etapa, pues habría que ejecutar el programa para determinar el valor y tipo de cada variable con anticipación.

Los Códigos 2.8 y 2.9 son trozos de código Python válido tanto léxica como sintácticamente, sin embargo, el segundo is inválido según la semántica del lenguaje. En particular, es incorrecto intentar imprimir el valor de una variable antes de haberla definido. Este es un error de tipo semántico que resultaría difícil de detectar en la etapa sintáctica.

```
1 a = 15
2 print(a)
```

Código 2.8: Código Python válido semánticamente

```
1 print(a)
2 a = 15
```

Código 2.9: Código Python inválido semánticamente

2.4. Gramáticas

En lingüística, la gramática es el conjunto de reglas estructurales que definen la composición de cláusulas, frases y palabras en un lenguaje natural. El estudio de los lenguajes naturales y sus reglas gramaticales es un área amplia, con muchas excepciones y particularidades según cada lenguaje. [1]

En computación, los lenguajes de programación suelen definirse en términos de una gramática. Por ejemplo, cómo se estructuran las cláusulas condicionales, los bucles, la puntuación o qué expresiones son válidas.

Para poder estudiar y facilitar la interpretación de un lenguaje es necesario algún tipo de formalismo o modelo que permita describir de manera precisa la gramática del mismo. Con esto se puede construir software que permita, entre otras cosas, reconocer si un texto dado es válido según la gramática del lenguaje o no.

Entre los modelos matemáticos relacionados al estudio y definición de lenguajes, destacan las gramáticas libres de contexto, ampliamente usadas en la definición de lenguajes de programación.

[1] Estos conceptos son estudiados a partir de Lyons[44], O'Grady et al.[45] y Chomsky[23]

2.4.1. Gramáticas Libres de Contexto

Una gramática libre de contexto, o CFG por sus siglas en inglés, es un conjunto de reglas recursivas que permiten generar patrones de texto. Una CFG se compone de los siguientes elementos:

1. Un conjunto de símbolos **terminales**, que corresponden a caracteres del alfabeto del lenguaje.

2. Un conjunto de símbolos **no-terminales**, que funcionan como marcadores para la generación de símbolos terminales.

3. Un conjunto de **reglas de producción** que indican cómo reemplazar o re-escribir los símbolos no-terminales.

4. Un **símbolo de inicio**, que es un símbolo no-terminal especial que genera todos los demás patrones del lenguaje.

Por ejemplo, la siguiente es una gramática que describe expresiones aritméticas:

```
<exp> -> number
<exp> -> (<exp>)
<exp> -> <exp> + <exp>
<exp> -> <exp> - <exp>
<exp> -> <exp> * <exp>
<exp> -> <exp> / <exp>
```

acá, `<exp>` es el único símbolo *no-terminal* y además es el *símbolo de inicio*. A partir de este símbolo, aplicando las reglas de producción se pueden producir todas las cadenas de texto válidas en este pequeño lenguaje. La primera regla indica que una expresión `<exp>` puede reemplazarse o re-escribirse como un número (es decir, un número es una expresión válida del lenguaje). La segunda regla indica que una expresión encerrada entre paréntesis también es una expresión válida. Nótese cómo esta regla se define de manera recursiva. El resto de las reglas indica que la suma, diferencia, producto y división de dos expresiones es también una expresión.

2.4.2. Backus-Naur Form

La notación de Backus-Naur, o *Backus-Naur Form (BNF)*, es un metalenguaje para la expresión de gramáticas libres de contexto. Fue propuesta por John Backus en el año 1959 como parte del desarrollo y definición del lenguaje ALGOL 58 [17]

El siguiente es un ejemplo, en notación BNF, de una gramática que describe todas las cadenas de texto formadas sólo por ceros y unos:

```
<binary-string> ::= <binary-string> 0 | <binary-string> 1 | 0 | 1
```

Algo destacable de la notación BNF es que permite definir múltiples reemplazos en la misma regla de producción, separados por el símbolo "`|`" (barra). De este modo, la regla anterior equivale a decir que, una cadena binaria puede ser:

1. Una cadena binaria seguida de un cero.
2. Una cadena binaria seguida de un uno.
3. Un cero.
4. Un uno.

La siguiente es una gramática para la descripción de enteros con signo:

```
<signed> ::= + <nosign> | - <nosign>
<nosign> ::= <digit> <nosign> | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

estas reglas indican que un número con signo se compone de un signo ("`+`" o "`-`") seguido de un número sin signo. Nótese cómo "`+0`", "`-0`", "`-00000`" serían números válidos según esta gramática.

2.5. CFG vs PEG

En el contexto de los intérpretes una propiedad muy importante es el soporte para reglas de producción recursivas por la izquierda. Esto quiere decir que una regla de producción puede empezar por una referencia a sí misma, directa o indirectamente.

Considérese el siguiente ejemplo de expresiones aritméticas:

```
add  ::= exp '+' exp
mul  ::= exp '*' exp
exp  ::= add | mul | num
```

Esta gramática reconoce adiciones múltiples como "`5 + 4 + 3`". Intuitivamente, esto podría entenderse como `(5 + 4) + 3` o como `5 + (4 + 3)`. Esto indica que la gramática permite dos árboles de interpretación para una misma expresión. Aunque en este ejemplo el resultado es el mismo, es fácil ver cómo esta ambigüedad podría traer problemas si el orden de las operaciones es importante.

Las gramáticas de análisis sintáctico de expresiones o *PEG* (*Parsing Expression Grammar*) son un tipo de gramática formal. PEG fue descrito por Bryand Ford en 2004 [30]. Este tipo de gramáticas son tan poderosas como las gramáticas CFG, pero, de acuerdo a sus autores, describen de manera más natural los lenguajes de programación.

La principal diferencia entre CFG y PEG es que el ordenamiento de las opciones no es relevante en CFG, pero sí en PEG. Si hay múltiples formas de interpretar un texto de entrada,

una CFG sería ambigua y por ende incorrecta. En cambio, una PEG aplicaría la primera opción disponible, lo que soluciona automáticamente algunas ambigüedades. Otra diferencia es que las gramáticas PEG no necesitan un lexer o etapa de análisis lexicográfico.

Históricamente tanto PEG como algunas CFG han sido incapaces de procesar reglas recursivas por la izquierda, aunque existen herramientas que logran evitar esto, ya sea modificando el algoritmo de interpretación o re-escribiendo automáticamente las reglas recursivas por versiones no recursivas.

2.6. Generadores de Intérpretes

Un generador de intérpretes (*parser generator*) es una herramienta que permite, ya sea a través de un lenguaje propio o de una API[2], describir una gramática y generar automáticamente el código necesario para reconocerla [3]. Esto se diferencia de programar un intérprete de forma manual, en el que si bien podría lograrse un mejor rendimiento o flexibilidad (dependiendo del lenguaje), también deben proveerse todas las rutinas encargadas del manejo de texto.

Los generadores de intérpretes pueden generar un intérprete autónomo o requerir la inclusión de una biblioteca adicional que permite la ejecución del intérprete generado (*runtime*).

2.7. Combinadores de Intérpretes

Un combinador de intérpretes (*parser combinator*) es una herramienta que permite combinar varios intérpretes individuales en una gran intérprete global [37]. La ventaja de esto es que permite la creación de complejos intérpretes a partir de intérpretes más simples. Estos intérpretes, por ser más sencillos, son también más legibles, entendibles y reusables. En la práctica, un combinador de intérpretes en realidad combina múltiples funciones "reconocedoras" que cooperan en la interpretación de un texto de entrada compartido. Por ejemplo, una función puede reconocer el caracter "a" y otra función reconocer el caracter "b". Se podrían combinar estos dos reconocedores en un intérprete que puede, por ejemplo, reconocer "a" seguido de "b".

Dado que, en general, un combinador de intérpretes es más lento que un generador de intérpretes suelen ser usados para interpretar pequeños sub-lenguajes o lenguajes de dominio específico.

[2]Application Programming Interface

[3]La idea de un generador de intérpretes es similar a la idea de un compilador de compiladores [21]

2.8. Tecnologías y Herramientas

2.8.1. Problema

Para la construcción del intérprete del lenguaje se evaluaron varias tecnologías que podrían asistir en este proceso. A continuación se detallan varias soluciones tecnológicas para este problema, acompañadas de sus ventajas y desventajas.

Solución 0: LEX/YACC

Sitio web: `http://dinosaur.compilertools.net`

lex es un programa que genera analizadores lexicográficos ("lexers"). Fue escrito originalmente por Mike Lesk y Eric Schmidt y publicado oficialmente en 1975 [43]. Aunque inicialmente era un software de código propietario, se han escrito muchas implementaciones de código abierto a lo largo de los años. Es el analizador lexicográfico por defecto en muchas distribuciones UNIX y su especificación es parte del estándar POSIX.

lex funciona leyendo una especificación del analizador lexicográfico y genera el código fuente en C que lo implementa.

En la especificación del lexer se pueden incorporar trozos de código C que son ejecutados cada vez que una regla de producción es activada. El Código 2.10 ejecuta una instrucción en C cada vez que un entero se reconoce en el texto de entrada.

```
%%
[0-9]+  {
            /* yytext is a string containing the matched text. */
            printf("Saw an integer: %s\n", yytext);
        }
%%
```

Código 2.10: Ejemplo de lex

yacc (Yet Another Compiler-Compiler) [38] es un software para sistemas operativos UNIX. Es un generador de intérpretes de tipo LALR ("Look Ahead Left-to-Right"). yacc funciona leyendo una especificación de la gramática en formato BNF y produciendo un intérprete en código C. Al igual que lex, permite la inclusión de "acciones" o trozos de código junto a las reglas que son ejecutadas cuando éstas se activan.

El Código 2.11 muestra una regla de producción en *yacc*. En este caso, la acción añade un nodo al AST en memoria.

```
expr : expr '+' expr  { $$ = node('+', $1, $3); }
```

Código 2.11: Ejemplo de yacc

yacc ha gozado de mucha popularidad y ha sido el generador de intérpretes en UNIX durante mucho tiempo. Ha sido sustituido por re-implementaciones más modernas, notablemente, por `GNU bison` [25] .

Hay que observar que yacc sólo realiza el análisis sintáctico y requiere de un analizador léxico que le provea los tokens. Debido a esto, la combinación lex+yacc ha sido muy popular a lo largo de los años.

La metodología de incluir acciones junto a las reglas es bastante común y se ve en otras tecnologías mencionadas más adelante.

Ventajas	**Desventajas**
No aplica	*No aplica*

Tabla 2.1: Análisis de lex

Ni *lex*, *yacc* ni *bison* son alternativas realistas para esta memoria, sin embargo representan puntos de partida interesantes a tomar en cuenta. Muchas bibliotecas en JavaScript han tomado, de una u otra manera, inspiración a partir de estas bibliotecas, por lo que vale la pena estudiarlas.

Solución 1: ANTLR

Sitio web: `http://www.antlr.org/`

ANTLR [47] es un generador de intérpretes escrito en Java que además permite generar intérpretes para JavaScript y muchos otros lenguajes. ANTLR está basado en un nuevo algoritmo para procesar gramáticas LL(*) desarrollado por el autor de la biblioteca [46].

La sintaxis de ANTLR es sencilla y permite definir separadamente las gramáticas del lexer y del intérprete.

Es bastante popular debido a sus muchas opciones y la gran cantidad de gramáticas desarrolladas por la comunidad y disponibilizadas abiertamente. Además, la última versión (v4) soporta reglas de producción recursivas por la izquierda

En vez de incluir acciones directamente en las reglas como yacc o lex, ANTLR provee mecanismos para visitar el AST generado. En particular, se genera una clase que implementa el patrón de diseño "Visitor" sobre el AST, permitiendo definir en ella toda la lógica del programa. Esto mantiene la gramática y la semántica (acciones) separadas.

El Código 2.12 es un ejemplo de gramática en ANTLR.

```
1 /*
2  * Parser Rules
3  */
4
5 operation   : NUMBER '+' NUMBER ;
```

```
6
7 /*
8  * Lexer Rules
9  */
10
11 NUMBER      : [0-9]+ ;
12 WHITESPACE : ' ' -> skip ;
```

Código 2.12: Ejemplo de ANTLR

acá `NUMBER` y `WHITESPACE` son definiciones para el lexer y operation es una regla para el intérprete. Como se puede ver, la sintaxis es similar a BNF, con algunas extensiones. Por ejemplo, el token `NUMBER` está descrito por una expresión regular `[0-9]+` y el token `WHITESPACE` tiene asociada una "acción" propia de ANTLR. Esta acción, sin embargo, es puramente opcional y ayuda a simplificar el trabajo con la gramática.

La Tabla 2.2 ofrece un análisis de las ventajas y desventajas que ofrece ANTLR respecto a esta memoria.

Ventajas	**Desventajas**
1. Biblioteca madura y consolidada. 2. Excelente documentación y soporte de la comunidad. 3. Implementa LL(*) con reglas de producción recursivas por la izquierda. 4. Muchos ejemplos implementando gramáticas conocidas, por ejemplo: JSON, CSS, Python, etc. Esto resulta particularmente útil, pues se pueden tomar estas gramáticas como referencias y extraer de ellas los elementos que resulten útiles, sin tener que reinventar las reglas ni tener que validar todo nuevamente. 5. Permite generar el intérprete para muchos lenguajes (Java, JavaScript, C#, C++, Go, Python, etc).	1. Requiere software en otro lenguaje, Java, para la generación del intérprete. 2. Requiere la inclusión de software adicional para la ejecución del intérprete en su entorno final.

Tabla 2.2: Análisis de ANTLR

Solución 2: JISON

Sitio web: `http://zaa.ch/jison/`

Jison es una reimplementación de `bison` (y por derivación, de `yacc`), pero en JavaScript. A diferencia de `bison` y `yacc`, Jison incluye su propio analizador lexicográfico inspirado por `flex` (otra reimplementación de `lex`).

El intérprete que Jison genera no necesita de una biblioteca adicional para funcionar, sino que opera de manera autónoma.

Una gramática Jison puede especificarse en formato JSON o en un estilo similar a `bison`/`yacc`. Ambas formas requieren la inclusión de acciones (código) en las reglas, que luego son ejecutadas cuando estas se activan.

El Código 2.13 es un ejemplo extraído de la documentación de Jison.

```
{
   "comment": "JSON Math Parser",
   // JavaScript comments also work

   "lex": {
      "rules": [
         ["\\s+",                    "/* skip whitespace */"],
         ["[0-9]+(?:\\.[0-9]+)?\\b", "return 'NUMBER'"],
         ["\\*",                     "return '*'"],
         ["\\/",                     "return '/'"],
         ["-",                       "return '-'"],
         ["\\+",                     "return '+'"],
         [..] // skipping some parts
      ]
   },

   "operators": [
      ["left", "+", "-"],
      ["left", "*", "/"],
      ["left", "UMINUS"]
      [..] // skipping some parts
   ],

   "bnf": {
      "expressions": [["e EOF",   "return $1"]],

      "e" :[
         ["e + e",  "$$ = $1+$3"],
         ["e - e",  "$$ = $1-$3"],
         ["- e",    "$$ = -$2", {"prec": "UMINUS"}],
         ["NUMBER", "$$ = Number(yytext)"],
         [..] // skipping some parts
      ]
   }
}
```

Código 2.13: Ejemplo de Jison

Como se ve, la gramática está especificada en formato JSON. Existen definiciones para el

análisis lexicográfico y para el análisis sintáctico. También se pueden declarar operadores, que ayudan al intérprete a decidir sobre la precedencia y desambiguar algunas expresiones.

La Tabla 2.3 ofrece un análisis de las ventajas y desventajas de Jison respecto a esta memoria.

Ventajas	**Desventajas**
1. Muy popular en el ecosistema JavaScript. CoffeeScript (una variante de JavaScript) utiliza Jison para la generación de su intérprete. 2. Sintaxis inspirada en yacc/bison, por lo que resulta familiar para aquellas personas que hayan trabajado con dichas bibliotecas. 3. Lo anterior también permite reutilizar gramáticas ya definidas para yacc/bison. 4. Al igual que bison, jison puede reconocer lenguajes descritos por gramáticas LARL(1), aunque tiene modos de operación para reconocer LR(0), SLR(1) y LR(1). 5. Separa la gramática en léxico y sintaxis. 6. No requiere la inclusión de software adicional para ejecutar el intérprete.	1. En ocasiones remite a la documentación de bison/yacc para aquellas secciones no documentadas. 2. No tiene tutoriales. 3. Las gramáticas no son particularmente legibles ni fáciles de mantener.

Tabla 2.3: Análisis de Jison

Solución 3: NEARLY

Sitio web: `https://nearley.js.org/`

Early es un algoritmo para la interpretación de cualquier gramática libre de contexto, incluso aquellas que son ambiguas o que utilizan reglas de producción recursivas por la izquierda. El algoritmo toma su nombre a partir de su inventor, Jay Early, quien lo describiera en el año 1968 [26]. El algoritmo utiliza programación dinámica para atacar el problema de la ambigüedad gramatical. Sin embargo, como desventaja, suele ser más lento que otros algoritmos.

Nearly es una implementación del algoritmo Early en JavaScript. Nearly es capaz de detectar cuando una gramática es ambigua y reportar este tipo de errores.

Una gramática en Nearly se escribe en un archivo "`.ne`", en una sintaxis propia que permite

la inclusión de código JavaScript.

El Código 2.14 es un ejemplo de una gramática en Nearly.

```
# from the examples in Nearly
# csv.ne

@{%
var appendItem = function (a, b) {
  return function (d) { return d[a].concat([d[b]]); }
};
var appendItemChar = function (a, b) {
  return function (d) { return d[a].concat(d[b]); }
};
var empty = function (d) { return []; };
var emptyStr = function (d) { return ""; };
%}

file          -> header newline rows
{% function (d) { return { header: d[0], rows: d[2] }; } %}

# id is a special preprocessor function that return the first token in the
    match
header        -> row                                {% id %}

rows          -> row
                 | rows newline row                 {% appendItem(0,2) %}

row           -> field
                 | row "," field                    {% appendItem(0,2) %}

field         -> unquoted_field                 {% id %}
                 | "\"" quoted_field "\""        {% function (d) { return d[1]; } %}

quoted_field  -> null                                {% emptyStr %}
                 | quoted_field quoted_field_char    {% appendItemChar(0,1) %}

quoted_field_char -> [^"]                      {% id %}
                     | "\"" "\""               {% function (d) { return "\""; } %}

unquoted_field    -> null                                  {% emptyStr %}
                     | unquoted_field char                 {% appendItemChar(0,1) %}

char              -> [^\n\r",]                             {% id %}

newline           -> "\r" "\n"                             {% empty %}
                     | "\r" | "\n"                         {% empty %}
```

Código 2.14: Ejemplo de Nearly.js

Esta gramática describe un archivo en formato CSV. Como se puede ver, al igual que Jison, junto a cada regla se escribe una acción que es ejecutada cuando la regla es válida. Nearly permite definir un encabezado con funciones JavaScript, lo que simplifica reusar lógica a lo largo de las expresiones.

Nearly también incluye herramientas para la depuración y entendimiento del intérprete. Por

ejemplo, permite "invertir" el intérprete y convertirlo en un generador de textos aleatorios que son reconocidos por la gramática. Esto facilita las pruebas. También incluye una herramienta para generar de diagramas de sintaxis, también conocidos como *railroad diagrams* (ver Figura 2.1).

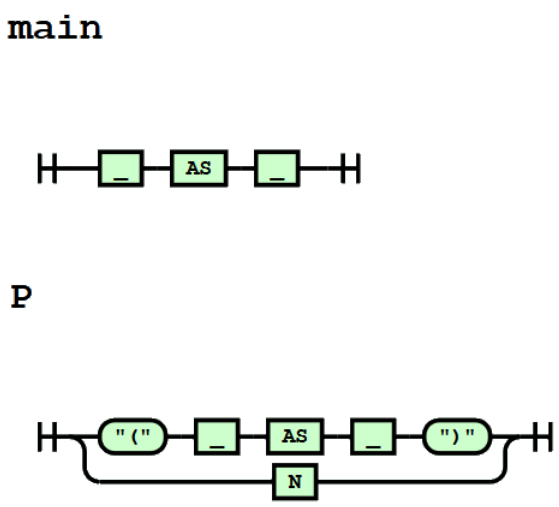


Figura 2.1: Diagrama *railroad* generado por Nearly

Al igual que ANTLR, Nearly requiere la inclusión de una biblioteca adicional para la ejecución del intérprete (*runtime*).

La Tabla 2.4 ofrece un análisis de las ventajas y desventajas de Nearly respecto a esta memoria.

Ventajas	Desventajas
1. Emplea el algoritmo Early, por lo que soporta todas los tipos de CFG 2. Procesa reglas recursivas por la izquierda sin problema. 3. Procesa gramáticas ambiguas sin problema. 4. Tiene herramientas para probar y visualizar la gramática. 5. Editor online. 6. Funciona tanto en el servidor (NodeJS) como en navegadores. 7. Muy popular. 8. Buen soporte de la comunidad. 9. La documentación cubre la mayoría de los casos de usos, dispuesta en modo de guía o tutorial.	1. Requiere la inclusión de software adicional para la ejecución del intérprete en su entorno final. 2. La documentación no cubre todas las funcionalidades. 3. No tiene muchas gramáticas de ejemplo.

Tabla 2.4: Análisis de Nearly

Solución 4: PEG.js

Sitio web: `https://pegjs.org/`

PEG.js es generador de intérpretes para JavaScript basado en el modelo formal PEG. Según sus autores, produce intérpretes rápidos con excelente reporte de errores. Funciona de manera similar al resto de los generadores, escribiendo primero la gramática en un archivo y utilizando luego una herramienta para generar el intérprete a partir de esta. Adicionalmente, se puede describir la gramática a través de código, utilizando los métodos provistos por la API de PEG.js.

Al igual que otros generadores, la gramática puede incluir acciones a ejecutar junto a las reglas. Adicionalmente, posee predicados semánticos, que funcionan como "puertas" lógicas. Estas funciones determinan si una regla puede o no activarse, seguń el contexto o estado interno del intérprete.

La gramática se guarda en un archivo en formato ".pegjs" y permite la inclusión de código JavaScript. El Código 2.15 es un ejemplo de gramática para expresiones aritméticas.

```
1 // math.pegjs
2 {
3   function makeInteger(o) {
4     return parseInt(o.join(""), 10);
5   }
6 }
7
8 start
```

```
  = additive

additive
  = left:multiplicative "+" right:additive { return left + right; }
  / multiplicative

multiplicative
  = left:primary "*" right:multiplicative { return left * right; }
  / primary

primary
  = integer
  / "(" additive:additive ")" { return additive; }

integer "integer"
  = digits:[0-9]+ { return makeInteger(digits); }
```

Código 2.15: Ejemplo de PEG.js

La Tabla 2.5 ofrece un análisis de las ventajas y desventajas de PEG.js respecto a esta memoria.

Ventajas	**Desventajas**
1. Editor online. 2. Muy popular. 3. Buen soporte de la comunidad.	1. Documentación buena, pero no excelente. 2. No hay suficientes gramáticas de ejemplo. 3. No hay tutoriales.

Tabla 2.5: Análisis de PEG.js

Solución 5: Parsimmon

Sitio web: `https://github.com/jneen/parsimmon`

Parsimmon es un pequeño combinador de intérpretes escrito en JavaScript. Su API está inspirada en Parsec, otro combinador de intérpretes escrito en Haskell, con más historia y desarrollo [42].

El Código 2.16 es un ejemplo de gramática en Parsimmon para interpretar JSON

```
let whitespace = P.regexp(/\s*/m);

let JSONParser = P.createLanguage({
  // This is the main entry point of the parser: a full JSON value.
  value: r =>
    P.alt(
      r.object,
      r.string,
      [..]
```

```
    ).thru(parser => whitespace.then(parser)),

  [..]

  // Regexp based parsers should generally be named for better error reporting
    .
  string: () =>
    token(P.regexp(/"((?:\\.|.)*?)"/, 1))
      .map(interpretEscapes)
      .desc('string'),

  [..]

  object: r =>
    r.lbrace
      .then(r.pair.sepBy(r.comma))
      .skip(r.rbrace)
      .map(pairs => {
        let object = {};
        pairs.forEach(pair => {
          let [key, value] = pair;
          object[key] = value;
        });
        return object;
      }),
});

let text = // JSON Object
let ast = JSONParser.value.tryParse(text);
```

Código 2.16: Ejemplo de Parsimmon

Como se puede ver, se utilizan pequeños intérpretes, como `r.comma`, `r.pair.sepBy` y `r.lbrace`, que son combinados para producir un intérprete mayor.

La Tabla 2.6 ofrece un análisis de las ventajas y desventajas de Parsimmon respecto a esta memoria.

Ventajas	Desventajas
1. La sintaxis es JavaScript, sólo se requiere conocer la API de la biblioteca. 2. Excelente para descripción de pequeños lenguajes de dominio específico. 3. No requiere una biblioteca adicional para ejecutar el intérprete.	1. La documentación es buena, pero no excelente. 2. Tiene pocos ejemplos de gramáticas en los cuales basarse. 3. No tiene ejemplos de gramáticas más grandes y complejas, como CSS o Python, que puedan servir de referencia. 4. Es un proyecto relativamente nuevo, por lo que el soporte y preguntas es escaso. Por lo general se busca la forma de solucionar, en la biblioteca, problemas ya resueltos en otras.

Tabla 2.6: Análisis de Parsimmon

2.8.2. Decisión

Se escogió la solución 1: **ANTLR**.

Esto, debido a las siguientes ventajas que presenta:

1. Al ser una herramienta consolidada existe una gran masa de conocimiento colectivo en forma de preguntas y discusiones en foros. Muchos problemas típicos ya tienen solución.

2. Implementa LL(*) con reglas de producción recursivas por la izquierda. La gran mayoría de las bibliotecas no pueden ofrecer esta funcionalidad, que permite expresar gramáticas muy fácilmente.

3. Posee muchos ejemplos implementando gramáticas conocidas, por ejemplo: JSON, CSS, Python, etc. Esto resulta particularmente útil, pues se pueden tomar estas gramáticas como referencias y extraer de ellas los elementos que resulten útiles, sin tener que reinventar las reglas ni tener que validar todo nuevamente.

4. Permite generar intérpretes para distintos lenguajes (Java, JavaScript, C++, C#, Go, Swift, Python, etc). En el futuro esto permitiría implementar el lenguaje en otras plataformas de manera más sencilla.

5. Al utilizar un generador de intérpretes se fuerza la formalización de la gramática. El efecto de esto es que el intérprete generado acepta exclusivamente las cadenas de texto reconocidas por la gramática. Dicho de otro modo, si se construyera manualmente el intérprete no sería fácil demostrar que reconoce formalmente la gramática del lenguaje.

Respecto a las desventajas:

1. Se necesita utilizar Java para generar el intérprete, sin embargo, esto es un problema sólo para la persona que desarrolla el lenguaje, no para el usuario final.

2. Lo anterior también involucra una nueva etapa en el ciclo de desarrollo y mantenimiento del proyecto, sin embargo, se considera que esto es una merma menor en contraste a la ventajas de productividad que ofrece la biblioteca.

3. Requiere incluir el sistema de ejecución de ANTLR. Dependiendo del tamaño y complejidad de esta pieza de software esto podría ser un problema. Por ejemplo, para utilizar el lenguaje en un navegador web el usuario tendrá que descargar no sólo el intérprete generado, sino también este sistema acompañante. No obstante, estas piezas de código se pueden concatenar, minificar y comprimir usando tecnologías ya establecidas que logran excelentes resultados. Finalmente, y aunque existen soluciones como la recién descrita, se concluyó que el tamaño final de la biblioteca no es prioridad, por ahora.

En síntesis, se valoró mucho la madurez y calidad de documentación y ejemplos, así como el soporte de la biblioteca para distintos tipos de gramáticas. Como esta es la primera versión de un lenguaje nuevo se consideró prioritario poder producir un intérprete rápidamente.

2.9. Dificultades y Reflexiones

2.9.1. Conocimiento sobre lenguajes e intérpretes en general

El área de la computación relacionada a lenguajes, compiladores, intérpretes y formalismos asociados es bastante amplia. Como esta es una memoria dedicada al desarrollo de un lenguaje, naturalmente resulta necesario ahondar en estos temas.

Como base teórica el memorista contaba con lo aprendido en los cursos "CC3102 Teoría de la Computación" y "CC4101 Lenguajes de Programación". En ambos cursos se estudian temáticas como lenguajes formales, gramáticas formales, compiladores, modelos de computación, etc. Esta base teórica resultó invaluable a la hora de indagar en las soluciones existentes en la industria. Sin embargo, algunos conceptos que se manejaban sólo a un nivel conceptual debieron ser profundizados.

Por ejemplo, ante la pregunta "¿es necesario que el intérprete soporte gramáticas LL(k)?" Este concepto en particular, LL(k) representa el conjunto de intérpretes de gramáticas libres de contexto que procesan el texto de izquierda a derecha, realizando derivaciones por la izquierda. Además, se dice que un intérprete es LL(k) si puede mirar k tokens adelante del token en el que se encuentra actualmente (lookahead). Se denomina LL(*) a aquellos intérpretes que no están restringidos por un número finito k de tokens de lookahead, sino que pueden basar sus decisiones al reconocer si los siguientes tokens pertenecen a un lenguaje regular.

En general, se presentó la siguiente situación en repetidas ocasiones: entender conceptual-

mente una idea, como la anterior, pero no saber cómo aplicarla al desarrollo del lenguaje en concreto. Esto produjo una especie de "*deadlock*" conceptual en el cual no se podía avanzar con el diseño del lenguaje debido a la falta de profundidad en algunos conceptos.

2.9.2. Diseño de la sintaxis y elección de tecnología

Diseñar un lenguaje resulta ser un proceso cíclico en el que se re-evalúan constantemente las decisiones tomadas respecto a la sintaxis y las funcionalidades ofrecidas. Decidir una sintaxis en particular podría requerir cambiar la biblioteca utilizada. Sin embargo, si esto no fuera posible, se tendría que acomodar la sintaxis, lo cual obviamente no es deseable.

El proceso anterior requiere conocimientos en dos canales paralelos, por un lado, la sintaxis a desarrollar y, por otro, las capacidades de las bibliotecas utilizadas. Entre ambos canales se produce una especie de contención, pues no es hasta que se conoce mejor las capacidades de la biblioteca que se pueden considerar viables algunos elementos de la sintaxis. Inversamente, sin tener una decisión tomada sobre la sintaxis es difícil elegir una biblioteca por sobre otra. En síntesis, es el problema de "no saber lo que no se sabe".

Al buscar bibliotecas para construir el intérprete se halló una enorme variedad de ellas. Sin embargo, estudiar cada una de estas bibliotecas para conocer sus ventajas y desventajas es un proceso que requiere bastante tiempo. Esto debido a que muchas bibliotecas requieren aprender una sintaxis o lenguaje específico para poder utilizarlas. De hecho, se utilizaron dos bibliotecas antes llegar a la solución definitiva.

Dado que la sintaxis del lenguaje no estaba definida, resultó difícil determinar qué biblioteca sería la más adecuada.

Otra variable en este problema es el tiempo. Como el desarrollo del intérprete es sólo una parte del desarrollo de esta memoria, el tiempo que se le podía dedicar a esta etapa era limitado. En particular, no se pudo indagar completamente en todas las diferencias técnicas que ofrecía cada biblioteca, ni en los alcances teóricos de todas ellas, pues habría requerido demasiado tiempo.

Capítulo 3

Visualización

3.1. Historia

Además de estudiar lenguajes de marcado, también resulta interesante considerar nociones de visualización de algoritmos y estructuras de datos. Cabe hacer la distinción entre visualización de algoritmos y visualización de programas. La primera hace relación a la visualización de un algoritmo abstracto (por ejemplo, Quicksort), mientras la segunda hace relación a la implementación específica de un programa en código fuente.

troff

`troff` es un sistema de procesamiento de documentos desarrollado por AT&T para UNIX. `troff` tiene comandos que permiten designar tipografías, espaciado, párrafos, márgenes, pies de páginas, etc.

La historia de `troff` es larga y su origen se puede trazar hasta un programa llamado `RUNOFF`, que permitía la visualización con formato de archivos de texto plano. Por ejemplo:

ENTRADA

```
When you're ready to order,
call us at our toll free number:
.SK
.CENTER
1-800-555-xxxx
.SK
Your order will be processed
within two working days and shipped
```

SALIDA

```
When you're ready to order, call us at our toll free number:
                              1-800-555-xxxx
Your order will be processed within two working days and shipped
```

En esta época, no existían monitores de alta resolución ni la profundidad de color con la que hoy se cuenta, sino que se utilizaban terminales, por lo general bastante limitadas en sus capacidades de despliegue de información.

3.2. Técnicas de Visualización

En el año 1982 Rob Pike y Bart Locanthi Jr., de Bell Labs, diseñaron una terminal gráfica llamada *Blit* [48] (ver Figura 3.1). Esta terminal operaba con un mapa de bits y permitía la ejecución simultánea y asíncrona de múltiples ventanas. En particular, esta terminal fue una de las precursoras en cuanto al concepto de GUI (*Graphical User Interface*).

Jon L. Bentley y Brian W. Kernighan [18] diseñaron, utilizando diversas herramientas de la colección de programas de `troff`, un sistema rudimentario para la visualización de algoritmos en esta terminal.

Figura 3.1: Terminal Blit

El sistema permitía, por ejemplo, la visualización de algoritmos de ordenamiento. La Figura 3.2 muestra una visualización del algoritmo `Insertion Sort`:

El sistema hacía uso de varios programas de troff, entre ellos pic y grap, para lograr generar estas visualizaciones. Todo esto ocurría a través de archivos de texto que eran alimentados a un programa, para luego alimentar con el resultado de este al siguiente. La Figura 3.3 muestra un ejemplo de visualización, en este caso, de árboles binarios (no balanceados):

Estas técnicas representan una de las primeras formas de visualizar, e incluso depurar,

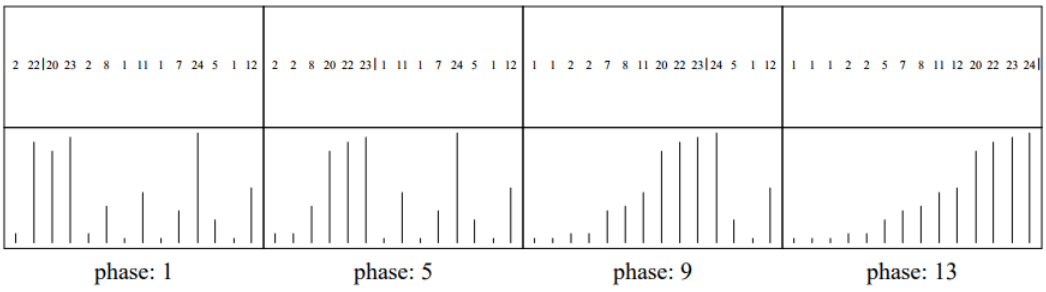


Figura 3.2: Visualizacion de Insertion Sort

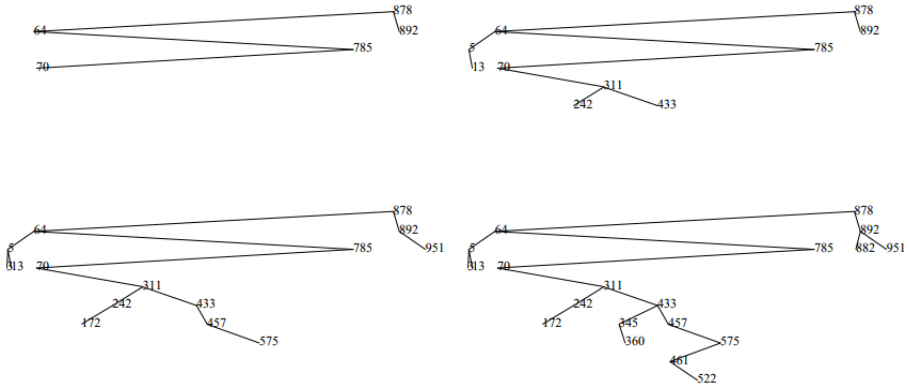


Figura 3.3: Visualización de árboles binarios

programas. Adicionalmente, este sistema permitía "reproducir" la ejecución del programa, avanzando pasos, pausando, retrocediendo, etc.

Marc H. Brown y Robert Sedgewick en "*Techniques for Algorithm Animation*" [22] detallan diversas técnicas y exploran, en general, la problemática de visualizar algoritmos (ver Figura 3.4).

De estas lecturas, se extraen las siguientes conclusiones:

1. **Mantenerse "cerca de los datos"**: si la visualización resulta demasiado "extravagante" o poco intuitiva para el usuario, puede generar más confusión que entendimiento. Es importante que las visualizaciones no se alejen demasiado de las estructuras de datos subyacentes, de modo que el usuario pueda establecer vínculos semánticos entre lo que está viendo y lo que intuye que está pasando.

2. **Eventos Interesantes**: las visualizaciones de programas típicamente hacen uso de "snapshots" o instantáneas que reflejan el estado de la ejecución en un momento dado. Una tarea crucial en este aspecto es escoger qué eventos, dentro del flujo del programa, resultan interesantes para la visualización. En el ejemplo del ordenamiento utilizando Insertion Sort, lo autores utilizaron dos eventos, el momento en que el algoritmo realiza una comparación y el momento en que realiza un intercambio de datos.

3. **Múltiples vistas**: al mostrar diferentes "vistas" (o visualizaciones) de la misma estructura simultáneamente, se genera un "hilo" conceptual que conecta ambas visualizaciones

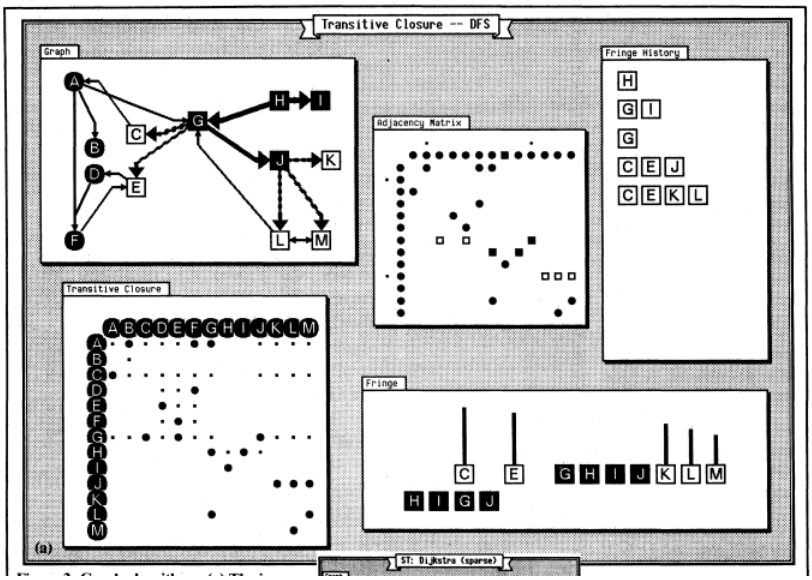


Figura 3.4: Distintas vistas de un grafo

y le permite al usuario entender más simplemente los cambios u operaciones en cuestión. Por ejemplo, una visualización puede entregar una mirada de alto nivel sobre lo que está ocurriendo, mientras que otra puede entregar una perspectiva de bajo nivel que permita indagar en los detalles.

4. **Uso de "hints" o signos que indiquen el estado de los datos**: se pueden usar formas, colores, bordes, etc, en las visualizaciones que ayuden a comunicar el estado particular de cada dato. Por ejemplo, en la Figura 3.4 se puede ver un grafo cuyos vértices y arcos presentan diferentes estilos (cuadrados, redondos, con fondo negro, con fondo blanco, etc). De esto se rescata que una estructura de datos, además de tener datos, tiene ciertos "estados" semánticos que se desea comunicar. Por ejemplo, se podría querer comunicar la idea de que un dato está "marcado", y por lo tanto, sería esperable que su visualización indicara, de algún modo, esta información.

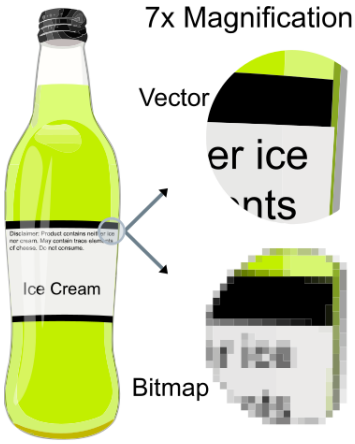


Figura 3.5: Ampliación de imágenes raster y vectorial

3.3. Tecnologías de Visualización

En esta sección de analizan la diferencias entre los formatos de imagen raster y vectorial, así como también una comparación entre las distintas capacidad que ofrecen HTML y SVG para la generación de visualizaciones. Se discuten algunos aspectos esenciales para la visualización de estructuras de datos como, por ejemplo, el anidamiento y posicionamiento de elementos. Finalmente, y al igual que para la construcción del intérprete del lenguaje, se utiliza el enfoque ingeniril para determinar la mejor solución tecnológica a las necesidades de visualización del lenguaje.

3.3.1. Raster *vs.* Vector

Un bitmap, o mapa de bits, es una estructura matricial que representa, típicamente, una grilla de pixeles o puntos de color. En estas imágenes existe una correspondencia uno a uno entre los bits de la matriz y los píxeles de la imagen. Esta forma de almacenar las imágenes es particularmente adecuada para el trabajo con fotografías e imágenes realistas. Una desventaja del formato raster es que depende fuertemente de la resolución de la imagen y, si ésta se modifica, resulta en una pérdida de calidad. Específicamente, una imagen raster no puede ser escalada sin una pérdida de calidad.

Por otro lado, una imagen vectorial almacena su información como un conjunto lógico de polígonos, trazos y estilos. Una imagen vectorial necesita ser interpretada posteriormente para producir una imagen imprimible. A diferencia de las imágenes raster, una imagen vectorial puede ser escalada infinitamente sin pérdida de calidad (ver Figura 3.5).

En la Tabla 3.1 se detallan ventajas y ventajas de cada formato:

	Ventajas	**Desventajas**
Raster	1. Fácil de visualizar en la mayoría de los dispositivos o sistemas. 2. La gran mayoría de procesadores de imágenes puede trabajar o exportar sus resultados a algún formato raster.	1. La imagen tiene una resolución fija y pierde calidad si esta es alterada, tanto si se aumentan o disminuyen sus dimensiones. 2. Dependiendo del uso que se le vaya a dar a la imagen, habría que decidir a priori con qué resolución generarla. 3. Entre mayores son las dimensiones de la imagen, mayor es el tamaño en disco que utiliza el archivo.
Vectorial	1. Escala infinitamente sin perder calidad. 2. No es necesario decidir la resolución.	1. Requiere interpretación por parte del software visualizador. 2. Resulta difícil reproducir imágenes realistas o fotografías. 3. Las imágenes deben ser "rasterizadas" si desean incluirse en un medio impreso (lo que fuerza a elegir una resolución específica).

Tabla 3.1: Comparación entre formatos raster y vectorial

3.3.2. SVG y HTML

En esta sección se estudiarán dos tecnologías, SVG y HTML. Luego, se estudiará cómo ambas pueden usarse para generar visualizaciones de estructuras de datos y las diferencias, ventajas y desventajas que presentan.

SVG

Scalable Vector Graphics (SVG)[29] es un formato para describir imágenes 2D en formato vectorial. Es decir, las imágenes producidas en SVG no utilizan una matriz definida de píxeles, sino que utilizan un conjunto de figuras. Los archivos SVG se guardan formato XML como archivos de texto plano. Utilizar XML permite que sea muy fácil buscar y modificar programáticamente un documento en SVG. De hecho, se puede utilizar un editor de texto simple para abrir y modificar un documento SVG. También se pueden utilizar editores de imágenes, como Inkscape o Adobe Illustrator que soportan este formato.

El Código 3.1 y La Figura 3.6 muestran cómo, en SVG, se puede dibujar un círculo con diferentes colores de relleno y borde.

```
<svg width="100" height="100">
   <circle cx="50" cy="50" r="40"
      stroke="green"
      stroke-width="4"
      fill="yellow" />
</svg>
```

Código 3.1: Código SVG para un círculo

Figura 3.6: Círculo generado con SVG

Como se puede ver, el documento SVG está escrito en formato XML. Se utiliza la etiqueta `<circle>` para dibujar un círculo centrado en las coordenadas especificadas (`cx` y `cy`) con un radio de largo `r`. También se pueden especificar el color y ancho del trazo (circunferencia) así como el color de relleno del círculo. Esta definición produce la imagen (Figura 3.6) al ser interpretada posteriormente por un software apropiado.

SVG es un estándar abierto desarrollado por la World Wide Web Consortium (W3C) desde 1999. Actualmente es soportado por todos los navegadores modernos.

Los documentos SVG, al estar escritos en XML, pueden ser fácilmente consultados, buscados, indexados y comprimidos. Que la imagen SVG sea un archivo en texto plano permite hacer modificaciones sin necesidad de un software especializado. En el ejemplo anterior, si se quisiera cambiar el color del círculo bastaría con modificar el valor de la propiedad `fill`.

SVG cuenta con diversos elementos de dibujo, como líneas (rectas, curvilíneas y segmentadas), rectángulos, elipses, polígonos, etc. Además, cuenta con transformaciones, filtros, oclusiones y patrones, entre otras funcionalidades de dibujo.

El Código 3.2 y la Figura 3.7 muestran cómo dibuar curvas de Bézier.

```
<path d="M100,250 C100,100 400,100
    400,250" />
```

Código 3.2: Código SVG para curva de Bézier

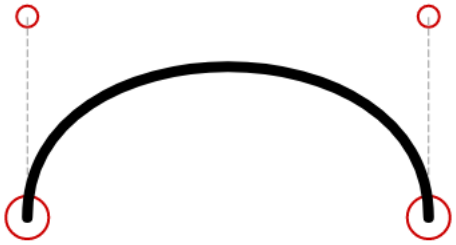


Figura 3.7: Curva generada con SVG

La etiqueta `<path>` recibe un argumento `d`, en el que se describe, con una serie de comandos y puntos, el camino a dibujar. En este ejemplo se añadieron algunas guías visuales para indicar el funcionamiento de la curva de Bézier.

Los documentos SVG puede ser estilizados usando hojas de estilo CSS. Esto permite separar estructura y presentación, favoreciendo el reuso de los estilos. SVG también ofrece interactividad a través de eventos (como hacer "click" en una figura) y capacidad de animación, utilizando SMIL (*Synchronized Multimedia Integration Language*). También se puede utilizar JavaScript para añadir lógica al documento, aunque esto no está soportado por todos los dispositivos.

HTML

Hypertext Markup Language (HTML) es el lenguaje de marcado estándar utilizado para la creación de páginas y aplicaciones web. Junto a CSS y JavaScript se han convertido en piedras angulares para la web moderna.

HTML describe una página a través de elementos semánticos estructurados jerárquicamente. En el Código 3.3 se describe un párrafo (`<p>...</p>`). Este párrafo contiene texto y un elemento de tipo `<span>`, utilizado en este caso para demarcar un texto en idioma francés.

```
<p>Oh well, <span lang="fr">c'est la vie</span>, as they say in France.</p>
```

Código 3.3: Ejemplo de HTML

Es importante notar que HTML es un lenguaje para añadir semánticas apropiadas para la web a un cuerpo de texto, no para darle estilo. Para ello se utiliza CSS.

Los navegadores han incluido históricamente algunos estilos por defecto para los elementos HTML. Por ejemplo, una lista no ordenada de elementos muestra cada elemento en su propia línea acompañado de un punto (ver Código 3.4 as y Figura 3.8.

```html
<ul>
    <li>Skip List</li>
    <li>Bloom Filter</li>
    <li>Iliffe vector</li>
</ul>
```

Código 3.4: Lista HTML

- Skip List
- Bloom Filter
- Iliffe vector

Figura 3.8: Lista HTML

Con CSS se puede modificar completamente la forma en que el navegador despliega los elementos HTML. El Código 3.5 y la Figura 3.9 muestran la aplicación de reglas de estilo a la lista mencionada anteriormente.

```html
<style type="text/css">
    li {
        display: inline;
        border: 1px solid teal;
        border-bottom-width: 5px;
        padding: 5px;
        text-transform: uppercase;
    }
    li:hover {
        border-color: cornflowerblue;
    }
</style>
<ul>
   <li>Skip List</li>
   <li>Bloom Filter</li>
   <li>Iliffe vector</li>
</ul>
```

Código 3.5: Lista HTML estilizada con CSS

SKIP LIST BLOOM FILTER ILIFFE VECTOR

Figura 3.9: Lista HTML estilizada

Como se puede ver, no sólo se añadieron bordes y rellenos, sino que se modificó la forma en que se despliega la lista. En vez de disponer de cada elemento de manera vertical, ahora se disponen horizontalmente. También se muestra una regla interactiva, que se aplica cuando se posa el puntero sobre un elemento, cambiando el color del borde.

HTML, por sí mismo, no posee las capacidades para considerarse una herramienta de visualización, sin embargo, al usarlo en combinación con CSS se vuelve una opción sencilla y eficaz para comunicar ideas visualmente.

3.3.3. Rasterización

La rasterización (o *rasterization*) es el proceso de transformar una imagen en formato vectorial a una en formato raster para poder visualizarla en dispositivos gráficos o imprimirla físicamente. El concepto también aplica al proceso de mostrar en una superficie 2D, como la mayoría de los monitores, una figura en 3D.

De interés para esta memoria es el proceso en el cual una imagen SVG puede transformarse, por ejemplo, en una imagen PNG. La mayoría de los programas de edición gráfica que trabajan con SVG, como Inkscape o Adobe Illustrator soportan la opción de exportar la imagen a formatos raster. Inkscape, además, permite su invocación desde la línea de comandos, lo que facilita su integración en otras tareas o programas donde se requiera rasterización.

Para el caso de HTML, la primera solución que viene a la mente es tomar una captura de pantalla y guardarla en un formato raster (como PNG). El proceso de captura de pantalla suele ser algo manual, pues es el usuario el que debe presionar una tecla en su computador y proceder a guardar la imagen. Hoy en día los navegadores más modernos permiten automatizar este proceso, pudiendo invocar esta funcionalidad desde JavaScript.

Lo anterior, sin embargo, requiere ejecutar un navegador web para la rasterización HTML. Como los navegadores web utilizan una interfaz gráfica de usuario (GUI) para presentar el contenido web al usuario, esto sería un gran problema, pues la rasterización no podría realizarse en servidores sin interfaz gráfica. Afortunadamente, existen proyectos, como PhantomJS [35], que permiten ejecutar un navegador web sin su interfaz gráfica. Esto se conoce como *headless browser* y se puede usar para realizar exploración automática de sitios web y pruebas de aceptación de interfaces de usuario.

3.4. Aspectos de Visualización de Estructuras de Datos

En esta sección se estudiarán algunos aspectos que se consideran relevantes en cuanto a la visualización de estructuras de datos. Para esto se plantean varios problemas y se ve la forma en que se solucionan utilizando HTML y SVG. Se eligieron estas dos tecnologías por ser actualmente las formas más populares de generar contenido visual en la web.

3.4.1. Aspecto 1: Manejo de Texto

Un requerimiento común a la hora de visualizar datos es incorporar información textual junto a otras formas. Un ejemplo típico de esto es dibujar cajas o círculos con un texto en su interior (para representar arreglos, nodos, listas, etc). Por lo tanto, la primera observación tiene que ver con cómo HTML y SVG manejan el texto y sus dimensiones. Se plantea el siguiente problema:

Problema: *dibujar un trozo de texto rodeado por un borde de color*

En HTML se podría escribir el texto dentro de una etiqueta span y luego darle un borde utilizando CSS (Código 3.6 y Figura 3.10).

```
<style type="text/css">
  span {
    border: 5px solid red;
  }
</style>

<span>Texto</span>
```

Código 3.6: HTML para un texto con borde

Figura 3.10: Texto con borde en HTML

La observación clave aquí es que si el contenido cambia (ver Código 3.7), la caja que lo contiene automáticamente se expande para acomodar este cambio (ver Figura 3.11).

```
<style type="text/css">
  span {
    border: 5px solid red;
  }
</style>

<span>Texto de ejemplo</span>
```

Código 3.7: HTML para un texto bordeado pero más largo

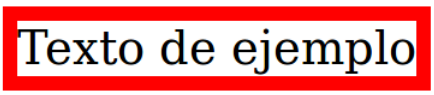


Figura 3.11: Texto más largo con borde en HTML

Esto funciona así debido a que en HTML y CSS los objetos se modelan a través de cajas. Estas cajas toman en cuenta las dimensiones del contenido y definen, adicionalmente, los

bordes, rellenos y márgenes del elemento. Esto es conocido como CSS Box Model y se puede apreciar en la Figura 3.12.

Figura 3.12: CSS Box Model

Los navegadores web modernos aplican estilos por defecto a la mayoría de los elementos para asegurar una experiencia de uso consistente a lo largo de la web. Las siguientes dos imágenes ilustran la caja del cuerpo del documento HTML (`<body>`) y del elemento creado (`<span>`).

Figura 3.13: Relleno por defecto del elemento body

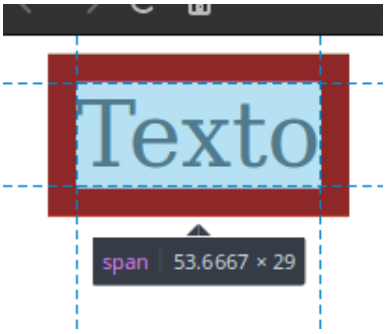


Figura 3.14: Relleno por defecto del elemento span

En la Figura 3.13 se ve cómo el navegador automáticamente.define un relleno (`padding`) para el cuerpo principal del documento, lo que asegura que los elementos no se yuxtapongan inmediatamente a la interfaz del navegador. Por otro lado, en la Figura 3.14 el navegador dibujó el borde del elemento span, pero como este tipo de elemento no define un relleno, el texto aparece sin ninguna separación de su contenedor. Esto es fácil de solucionar y tanto el relleno como el borde se ajustarán automáticamente según se modifique el texto.

En SVG, por otro lado, las entidades no se modelan a través de cajas que las contienen, sino simplemente como objetos en un plano cartesiano. Por ende, si se quiere recrear lo anterior

(un texto bordeado), se tendría que dibujar el texto y luego dibujar un rectángulo que lo rodee. Sin embargo, esto resulta tedioso, pues es difícil calcular a priori las dimensiones que ocupará un trozo de texto. Esto depende de la tipografía utilizada, del tamaño de letra, del espacio entre caracteres, etc. Incluso si se logran determinar con certeza las dimensiones del texto, no es posible configurar de manera automática que la caja se ajuste al texto. Es decir, si se modifica el texto en el archivo fuente el rectángulo mantendrá sus dimensiones originales y se verá mal.

Estas situaciones se pueden observar en el Código 3.8 y Figura 3.15 y en el Código 3.9 y Figura 3.16, respectivamente.

```
<svg height="400" width="400">
  <rect x="5" y="5" width="70"
    height="50"
          style="fill: none; stroke
    -width: 5;stroke: red" />

  <text x="20" y="35">Texto</text>
</svg>
```

Código 3.8: SVG para un texto bordeado

Figura 3.15: Texto y rectángulo en SVG

```
<svg height="400" width="400">
  <rect x="5" y="5" width="70"
    height="50"
          style="fill: none; stroke
    -width: 5;stroke: red" />

  <text x="20" y="35">Texto de
    ejemplo</text>
</svg>
```

Código 3.9: SVG para un texto bordeado pero más largo

Figura 3.16: Texto más largo y rectángulo en SVG

3.4.2. Aspecto 2: Conectividad entre elementos

Una actividad recurrente a la hora de diagramar o visualizar estructuras de datos es conectar, de algún modo, sus elementos. Esto es evidente en estructuras como árboles o grafos, pero también está presente en estructuras como las listas enlazadas o aquellas que utilizan punteros. Para ver cómo se comportan HTML y SVG ante este requerimiento se plantea el siguiente problema:

Problema: *trazar una línea desde un objeto a otro*

En SVG se utilizaría directamente un camino con algún estilo y las coordenadas de los objetos. El Código 3.10 y la Figura 3.17 ilustran esta solución, conectando dos vértices de un grafo.

```
<svg>
    ...
    <path fill="none" stroke="black
    " d="M27,-71.6966C27,-63.9827
    27,-54.7125 27,-46.1124"/>
    ...
</svg>
```

Código 3.10: Elemento path con una definición de camino

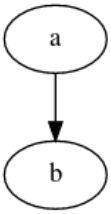


Figura 3.17: Camino entre dos vértices

En HTML no existe el concepto de línea (desde un punto cualquier a otro) como tal, por lo que habría que recurrir a alguna técnica que permita incorporar líneas al documento. Por ejemplo, se podría utilizar un elemento de un alto pequeño (pocos píxeles) con un ancho igual a la separación entre los elementos y luego utilizar posicionamiento absoluto junto a una rotación (transformación CSS). Esta solución se ilustra en el Código 3.11 y la Figura 3.18.

```
<style>
    #conector {
        position: absolute;
        background: black;
        height: 5px;
        width: 115px;
        left: 105px;
        top: 20px;
        transform: rotate(10deg);
    }
</style>
<div>
    <div id="a">A</div>
    <div id="b">B</div>

    <div id="conector"></div>
</div>
```

Código 3.11: HTML para conectar dos elementos

Figura 3.18: Conexión entre dos elementos HTML

Como se ve, esta solución necesita calcular las posiciones y dimensiones de los elementos con respecto a un contenedor común y luego calcular el ángulo de rotación y el ancho de la línea. Este es un procedimiento intrincado y propenso a errores. Finalmente, es debatible si HTML fue diseñado para este tipo de presentaciones.

Una mejora respecto a la solución anterior es utilizar un elemento SVG incrustado en el documento HTML, en el cual se dibuja sólo esta línea y luego se posiciona de manera absoluta en relación a los elementos. Esta tarea es precisamente la que automatiza la biblioteca *jsPlumb* [49] (ver Figura 3.19).

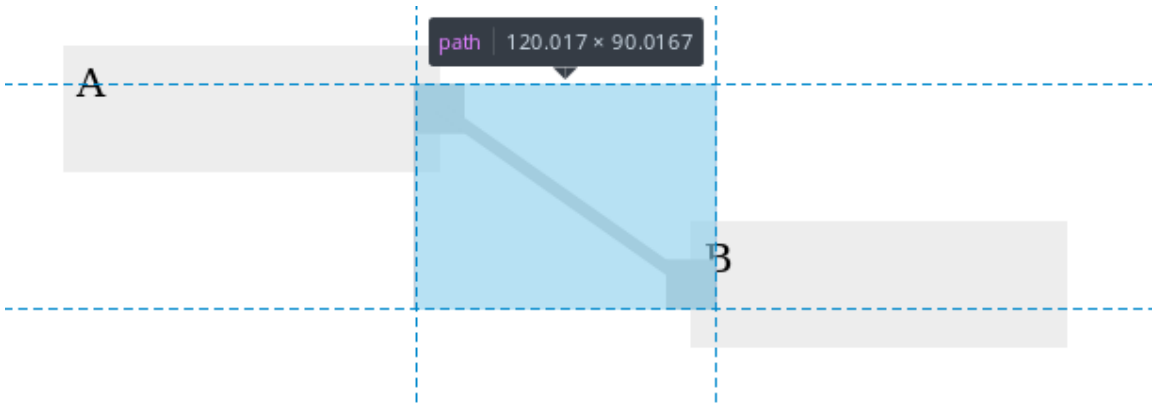


Figura 3.19: Conexión usando jsPlumb

En cuanto a conectividad, una ventaja de SVG es que soporta el dibujo de curvas de Bézier, lo que permite crear conexiones que rodeen o eviten solaparse con otros elementos (u otras conexiones).

3.4.3. Aspecto 3: Posicionamiento de elementos

Si se desea visualizar un arreglo de elementos es usual querer visualizarlos en una misma "fila" (todos alineados horizontalmente), esto implica que cada elemento debe estar posicionado de manera relativa. Esto es, cada elemento debe empezar a dibujarse en la posición en la que terminó el elemento anterior. También se querría que el texto mostrado en cada casilla esté centrado horizontal y verticalmente. Se plantean algunos problemas que permitirán elaborar más en estos temas:

Problema: *dibujar una serie de textos bordeados de manera continua*

En HTML esto es fácil de implementar utilizando tablas y dándole estilo a las celdas, como se ilustra con el Código 3.12 y la Figura 3.20.

```html
<style type="text/css">
    td { border: 1px solid black;
    padding: 5px;}
</style>
<table cellpadding="0" cellspacing=
    "0">
    <tbody>
        <tr>
            <td>A</td>
            <td>B</td>
            <td>C</td>
            <td>Texto</td>
        </tr>
    </tbody>
</table>
```

Código 3.12: HTML para varios textos bordeados

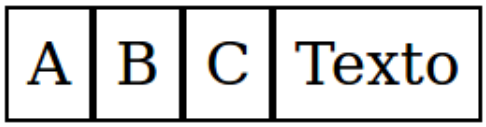


Figura 3.20: Tabla HTML para representar varios textos bordeados

En este caso el posicionamiento de las cajas lo provee automáticamente el navegador, pues es el estilo por defecto con el que se muestran las tablas.

Hacer lo mismo en SVG sería bastante más difícil, pues no se saben a priori las dimensiones de los textos y por lo tanto, las coordenadas en las cuales dibujar los rectángulos (bordes). Se podría utilizar JavaScript (soportado en SVG) para obtener las dimensiones de las cajas de texto y luego posicionar los contenedores de manera apropiada.

Sin embargo, si el esquema de distribución no fuera horizontal, sino escalonado, como se muestra en la Figura 3.21, sería bastante más difícil de implementar en HTML. Una solución podría ser forzar un margen superior negativo para cada elemento y luego agregar un margen izquierdo proporcional a la posición en la lista. Si bien es una solución, resulta más un artilugio que una solución mantenible.

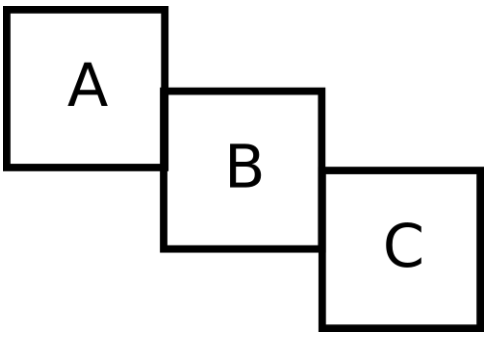


Figura 3.21: Cajas de texto escalonadas

Por otro lado, pese a que SVG no tiene un gran manejo de los textos y posicionamiento, sí tiene muchas características que lo hacen una excelente opción para todo tipo de visualizaciones. Por ejemplo, dibujar un grafo (ver Figura 3.22) no es particularmente difícil en SVG, pero sería todo un reto intentar hacerlo en HTML (y aunque se pudiera, probablemente tampoco sería una solución muy mantenible).

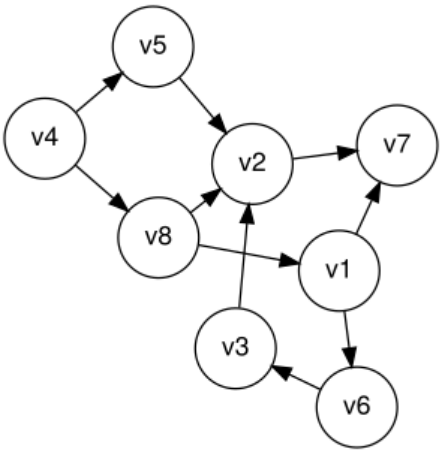


Figura 3.22: SVG conectando muchos elementos

3.4.4. Aspecto 4: Anidamiento de elementos

Muchas estructuras de datos soportan anidamiento. Por ejemplo, es común que una lista contenga otras listas como elementos. También puede ser que una estructura tenga referencias a sí misma, como suele ser el caso de los nodos en las listas enlazadas. Visualmente este anidamiento podría verse de varias formas, por ejemplo, el elemento podría encerrar o contener al otro, o podría apuntarlo con un conector. Específicamente, se podría decir que el primero es realmente *anidamiento* y el segundo es *referenciamiento*:

Figura 3.23: Arreglos anidados

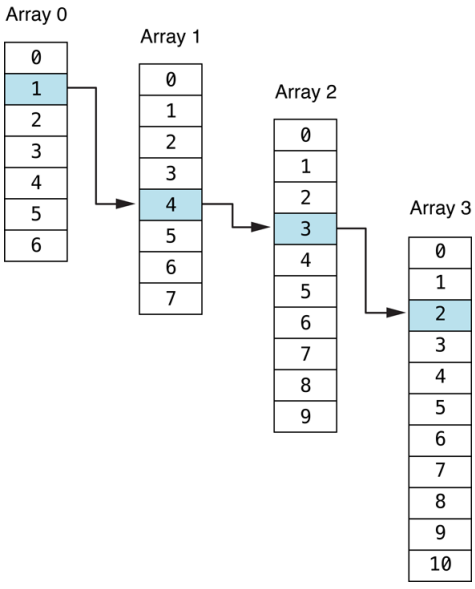


Figura 3.24: Arreglos conectados por flechas

En la Figura 3.23 se dibuja una estructura dentro de otra (anidamiento), mientras que en la Figura 3.24 se conectan con flechas (referenciamiento).

Ni el anidamiento ni el referenciamiento necesariamente tienen que ser hacia otro elemento del mismo tipo. Es válido, por ejemplo, que una lista tenga como elementos a objetos más complejos, como tablas de *hash* o estructuras.

El anidamiento es muy fácil de lograr con HTML, el lenguaje es en sí una estructura de árbol en la que los elementos se disponen de manera anidada y es el motor del navegador el que se encarga de la visualización. SVG no soporta anidamiento de los elementos directamente: cada elemento se dibuja de manera relativa al contenedor SVG. Es por ello que, en el ejemplo del texto rodeado de un borde, en SVG había que dibujar primero un rectángulo y luego, sobre el anterior, el texto. No obstante, SVG soporta anidamiento de otros documentos SVG, los que son dibujados en una posición relativa a su padre. Esto permite, de manera limitada, reusar y posicionar elementos de manera relativa.

El referenciamiento es mucho más fácil en SVG, pues basta conocer las coordenadas de los elementos para dibujar un conector entre ellos. En HTML esto es más difícil, como se vio en el "Aspecto 3: Conectividad entre Elementos" 3.4.2.

3.4.5. Aspecto 5: Disposición de grafos

Muchas estructuras de datos pueden diagramarse de una manera relativamente no ambigua. Por ejemplo, un arreglo suele ser una tabla, una lista suelen ser cajas u otras formas conectadas y dispuestas horizontalmente, una matriz suele ser una grilla de números con algún tipo de borde lateral, etc.

Para los grafos, la *representación* de los vértices y los arcos no suele ser un problema. Típicamente se emplean formas, como elipses o recuadros, acompañadas de conectores, como flechas con distintos tipos de puntas o terminaciones.

Sin embargo, la *disposición* de los vértices en el espacio puede variar mucho, incluso para un mismo grafo. Las Figuras 3.25 y 3.26 son ejemplos del mismo grafo, pero dispuesto de maneras diferentes.

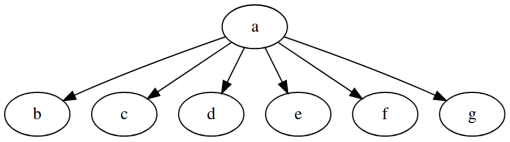


Figura 3.25: Grafo dispuesto jerárquicamente

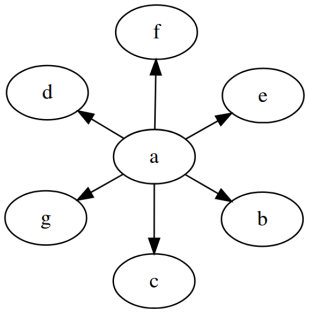


Figura 3.26: Grafo dispuesto circularmente

En este caso, el grafo es *planar*, lo que significa que se puede dibujar sobre un plano sin que haya dos arcos solapándose. Las Figuras 3.27 y 3.28 muestran distintas disposiciones de un grafo no planar (en específico, el grafo completo de 5 vértices, conocido como `K5`).

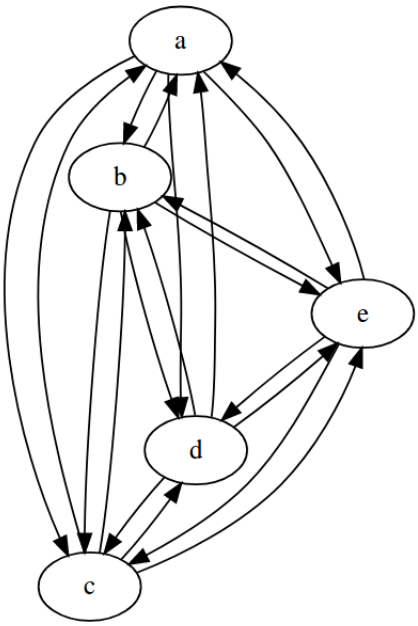


Figura 3.27: Grafo completo minimizando solapamientos

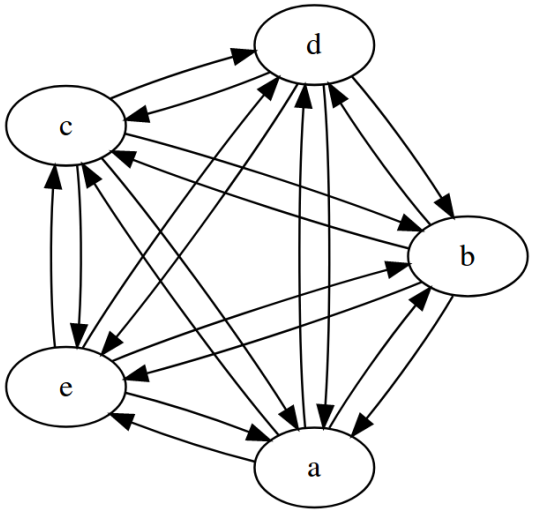


Figura 3.28: Grafo completo dispuesto circularmente

Ambas diagramaciones son válidas y aunque la primera disminuye la cantidad de arcos solapados la segunda tiene una mejor disposición visual (para este caso).

Dibujar grafos es un problema difícil. Existen muchos algoritmos que buscan determinar una buena disposición para un conjunto de vértices y arcos dados. Para un caso, podría resultar mejor aplicar un algoritmo en particular, mientras que en otros casos podría ser diferente. Diseñar una solución que entregue una buena visualización para el 100 % de los casos parece ser algo demasiado ambicioso por ahora.

Este es un aspecto en el que ni HTML ni SVG ofrecen realmente algún tipo de ventaja, pues es un problema de diagramación.

Por otro lado, ubicar objetos en posiciones específicas no es difícil en SVG ni en HTML, sin embargo, dibujar las conexiones es algo más complicado en HTML que en SVG.

3.4.6. Conclusiones

Los aspectos de visualizaciones de estructuras de datos estudiados en esta sección representan problemas fundamentales que deben ser tratados por el motor de visualización del lenguaje. Entender cómo se pueden resolver permite realizar una mejor evaluación de las tecnologías y plantear una estrategia de desarrollo más robusta.

3.5. Motor de Visualización

El motor de visualización del lenguaje será el responsable de generar visualizaciones adecuadas para cada estructura de datos. Como se vio en las secciones anteriores, existen varias dificultades técnicas que deben considerarse, así como ventajas y desventajas presentes en cada tecnología.

3.5.1. Problema

Determinar la o las tecnologías a utilizar para generar las visualizaciones de las estructuras de datos.

Restricciones

1. Deben ser tecnologías idealmente usables tanto en un navegador web como en un servidor.

2. Deben ser utilizables desde la línea de comandos sin requerir abrir un navegador u otro programa con interfaz gráfica de usuario.

Estas restricciones nacen de los objetivos del lenguaje: ser portable y fácil de usar e incorporar a cualquier flujo de trabajo. La web cumple con ambos de estos objetivos.

Solución 1: HTML

Una forma de implementar el motor sería utilizando plantillas pre-diseñadas en HTML y componerlas adecuadamente. Esto aprovecharía el motor de layout del navegador, el cual resolvería los problemas de manejo de texto y posicionamiento. También facilitaría el desarrollo de nuevas visualizaciones por parte de terceros, ya que resultaría muy fácil diseñar un componente con HTML y CSS e incorporarlo al lenguaje.

En la actualidad existen APIs de HTML5, como *Shadow DOM* y *Web Components* que facilitan la distribución y encapsulamiento de estas plantillas.

El principal problema de este enfoque es que sólo funciona para estructuras anidables y sin conexiones (líneas) entre ellas. Dibujar un grafo, una lista enlazada, o cualquier estructura que represente algún tipo de conexión (puntero) sería difícil.

Solución 2: SVG

Otra solución podría ser utilizar SVG completamente. En este caso resulta mucho más fácil dibujar estructuras enlazadas, como colas, árboles o grafos. Al igual que en el caso de HTML, se podrían diseñar plantillas para cada estructura de datos y estilizarlas con CSS.

El principal problema es que habría que calcular, de algún modo, las dimensiones de cada elemento, desde el hijo más profundo hasta el padre superior en la jerarquía. Es muy difícil saber *a priori* qué dimensiones específicas tendrá un texto. En particular, no es un dato que pueda obtenerse en tiempo de *generación* del documento, sino que se debe calcular con ayuda de JavaScript en tiempo de *visualización*. Basta con que la tipografía especificada en el documento no se encuentre en el sistema donde se visualiza para que el cálculo ya no sea válido.

Solución 3: HTML + SVG

Utilizar HTML y SVG de manera combinada se presenta como una opción muy robusta. Esto, porque combina lo mejor de ambas soluciones. Se puede utilizar HTML para posicionar y dimensionar los elementos de manera automática y crear conectividad entre ellos utilizando SVG. Para las formas, por ejemplos círculos u otros polígonos regulares, se podría también utilizar SVG. En casos donde el posicionamiento sea más complejo, se puede recurrir a dibujar toda la visualización con SVG.

En este caso la principal desventaja sería tener que posicionar y conectar adecuadamente los elementos. Como se vio, HTML brinda buen posicionamiento para estructuras simples como listas, arreglos o matrices, pero no tanto así para grafos o árboles.

Solución 4: D3.js

D3.js es una biblioteca para la creación de visualizaciones de datos. Está escrita en JavaScript y utiliza estándares web como HTML, SVG y CSS para la creación de documentos visuales. Posee abstracciones que permiten simplificar el manejo de datos, así como la construcción progresiva y reactiva de documentos en base a la estructura y cambios en los datos.

Las Figuras 3.29 y 3.30 muestran ejemplos de visualizaciones generadas con D3.js.

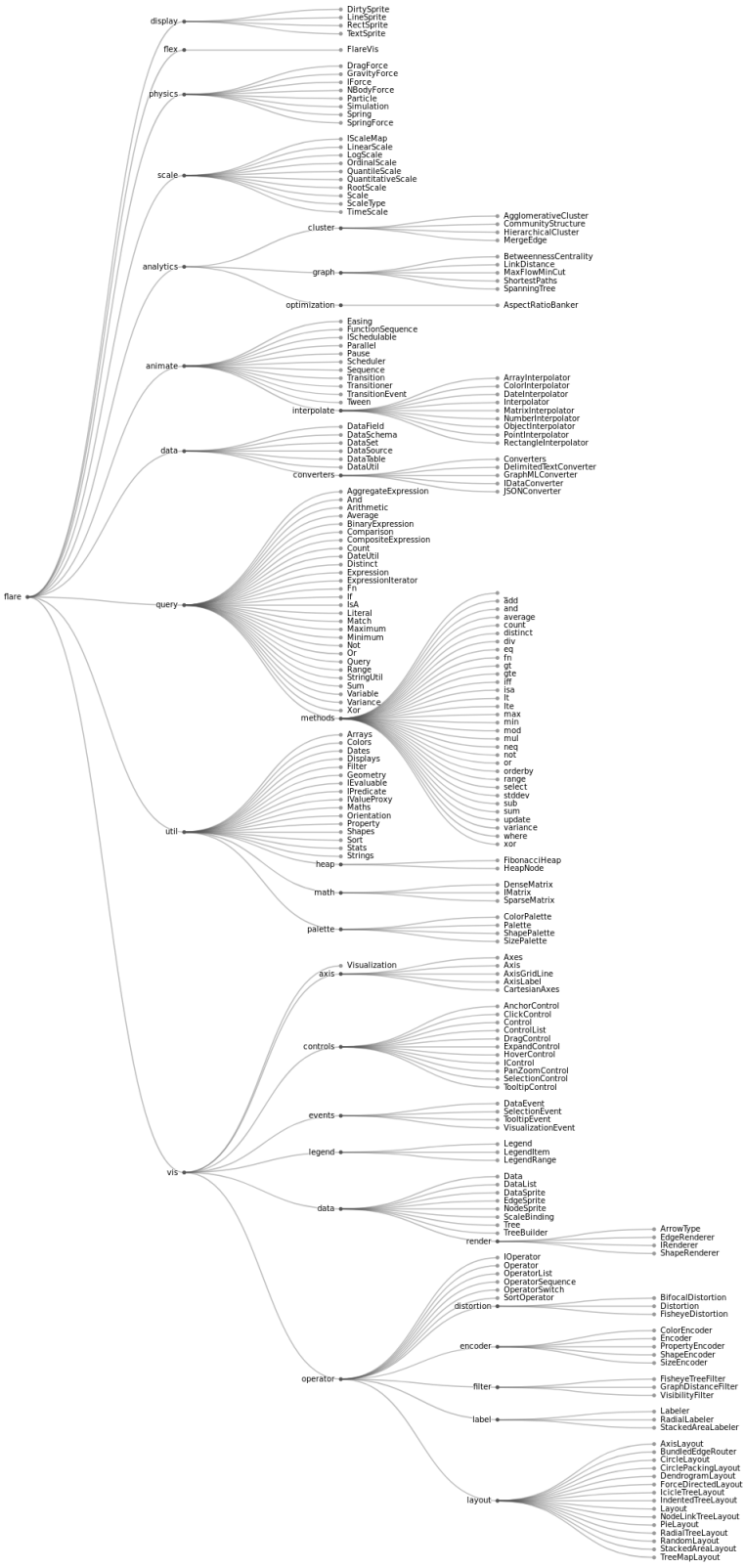


Figura 3.29: Diagrama de árbol utilizando D3

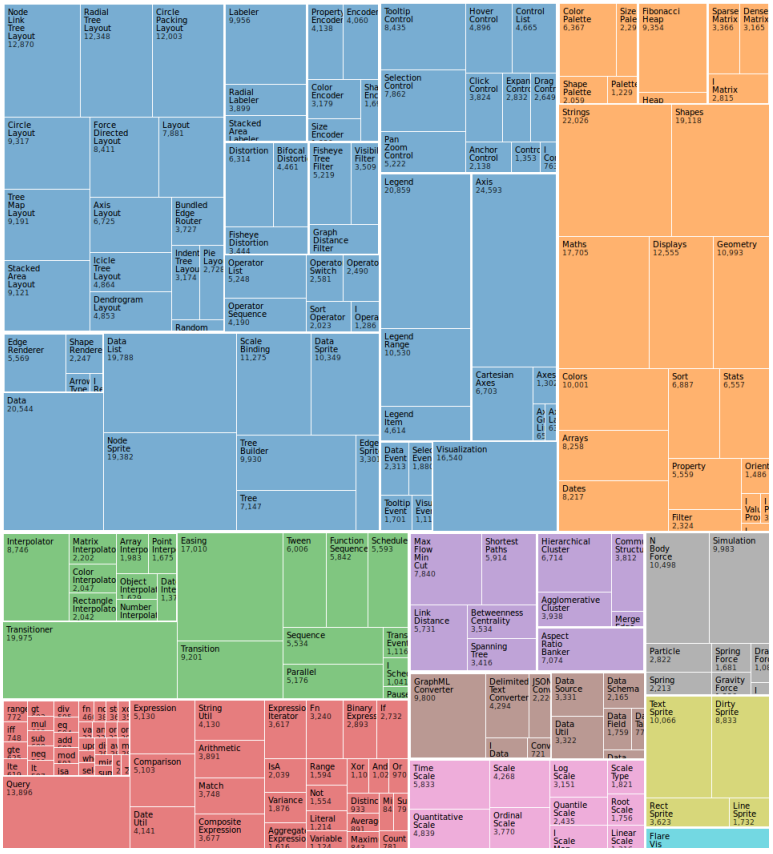


Figura 3.30: Árbol en forma de mapa utilizando D3

D3.js es actualmente la biblioteca más popular y robusta en cuanto a generar visualizaciones de datos en la web. Sin embargo, su curva de aprendizaje también es bastante alta. Por otro lado, D3.js no tiene un gran soporte para el reuso o anidamiento de otras visualizaciones. Lo anterior se traduce en que resulta muy apropiado para diseñar visualizaciones *ad hoc* para un conjunto de datos particulares, pero sería difícil utilizarla de manera general para el desarrollo del motor de visualización.

Otra desventaja de D3.js, en cuanto a esta memoria, es que involucra la ejecución y evaluación de JavaScript en una navegador. Esto puede aliviarse con la rasterización HTML que se comentó en la sección de "Tecnologías de Visualización" 3.3.

Solución 5: Graphviz + Emscripten

Graphviz [27] es un biblioteca muy popular para la visualización de grafos. Fue creada por AT&T Labs en el año 1991 y ha continuado su desarrollo hasta ahora. Utiliza una sintaxis, llamada *DOT* [39], para especificar los grafos (con sus vértices y arcos). Graphviz cuenta con una serie de motores que pueden tomar, como entrada, un archivo en formato *DOT* y producir una visualización del grafo allí descrito. Cada motor implementa un algoritmo diferente, lo que provee varias opciones según la naturaleza y complejidad del grafo.

Graphviz soporta tanto grafos dirigidos como no dirigidos. También ofrece muchas opciones para personalizar la forma en que se dibujan los vértices y arcos. Además de lo anterior, Graphviz permite exportar sus resultados en varios formatos, entre ellos, PNG, SVG y PDF.

La principal desventaja de Graphviz es que está escrito en C y por lo tanto no puede utilizarse directamente en la web. Habría que disponer de un servidor que, a través de llamadas web, invoque a Graphviz internamente (usando el sistema operativo y capturando su salida, por ejemplo).

Afortunadamente, existe una manera más sencilla. *Emscripten*[53] es una biblioteca que permite la compilación cruzada desde C/C++ a JavaScript. Esto permite ejecutar código escrito en C/C++ directamente en el navegador. Esto es posible debido a dos tecnologías:

1. **`LLVM`**[41]: es una serie de herramientas que conforman una infraestructura bajo la cual compilar código escrito en diversos lenguajes hacia una única representación de bajo nivel que luego puede ser ejecutada eficientemente. Actualmente LLVM tiene compiladores para muchos lenguajes, entre ellos, C, C++, Haskell, Julia, Lua, Objective-C, Python, Rust, Ruby, Scala, Swift, etc. En la actualidad, LLVM ha implementado la mayor parte de la biblioteca estándar de C++ y se ha vuelto un reemplazo viable de GCC para muchas plataformas.

2. **`asm.js`**[34]: es un lenguaje de programación intermedio, diseñado para permitirle a aplicaciones escritas en lenguajes como C ser ejecutadas en web, manteniendo un rendimiento considerablemente mejor que JavaScript. *asm.js* es un subconjunto de la sintaxis de JavaScript, por lo que no introduce nuevos elementos, sin embargo, un intérprete del navegador podría detectar y optimizar agresivamente una aplicación compilada hacia *asm.js*.

En la actualidad, cualquier lenguaje que pueda compilarse a la representación intermedia (*IR*) de *LLVM* podría ser compilado a *asm.js* y por lo tanto ejecutado en la web.

Graphviz es una de las bibliotecas que ha sido re-compilada en JavaScript utilizando Emscripten. De este modo, se pueden producir visualizaciones en SVG utilizando el lenguaje DOT y luego mostrarlas, ya sea directamente en el navegador o *rasterizarlas* para otros usos.

Una desventaja de Graphviz es que no permite, de manera sencilla, definir componentes o plantillas que puedan reutilizarse para modificar la apariencia de los elementos del grafo.

Por otro lado, contar con algoritmos de posicionamiento automático de vértices y arcos significa que es más difícil forzar a la biblioteca a seguir un esquema manual. Sin embargo, esto se podría modificar después de la invocación de Graphviz, directamente en el SVG, aunque habría que acomodar también las conexiones.

3.5.2. Decisión

Se decidió utilizar la solución 5: **Graphviz + Emscripten**.

Ventajas:

1. Algoritmos de grafos ya incorporados. Esta es una funcionalidad que es muy difícil de replicar o de programar directamente, por lo que su ponderación es muy elevada en la evaluación.

2. Salida en formato SVG. Resulta muy conveniente que la biblioteca exporte sus resultados en SVG, pues estos se pueden: 1) mostrar en un navegador web, 2) rasterizar e imprimir, 3) estilizar con CSS y 4) inspeccionar y modificar programáticamente, si se quisiera.

3. Al cargar Graphviz en el navegador y utilizarlo "localmente" no se necesita la implementación de un servidor o servicio que responda llamadas web. Es decir, un usuario puede descargar un script en JavaScript y eso es todo lo que necesita. Si el motor de visualización se incorpora en una página web esto permitirá utilizar el lenguaje sin una conexión a internet.

4. No necesita una etapa de interpretación de JavaScript, simplemente se invoca el motor (una vez) y la salida de este es una imagen SVG. Si la salida fuera HTML se necesitaría levantar un navegador web para poder rasterizar la imagen.

Desventajas:

1. Si bien Graphviz es poco flexible en cuanto a personalización, se considera suficiente el grado de personalización ofrecido. Debido al tiempo y a la complejidad de las tareas que habría que desarrollar si se utilizara otra solución, perder algo de estética por funcionalidad resulta un compromiso aceptable.

2. Graphviz, aunque es un proyecto maduro y robusto, no parece estar bajo un desarrollo particularmente activo. Sin embargo, se cree que, al utilizar estándares web (SVG y CSS) es muy difícil que en el corto o mediano plazo esto vaya a significar un problema.

3. Introduce una dependencia sobre una tecnología relativamente nueva, *asm.js*. Y aunque esta tecnología ya forma parte de las últimas versiones en la mayoría de los navegadores modernos, debe alcanzar aún una especificación formal y lograr, idealmente, convertirse en un estándar. Sin embargo, se cree que es una tecnología que sólo irá al alza en los años venideros, debido a la portabilidad que ofrece la web y el buen rendimiento obtenido.

3.6. Dificultades y Reflexiones

Conocimiento experto

El memorista tuvo que indagar mucho más allá de lo que conocía respecto a tecnologías web (HTML, SVG, CSS y JavaScript). Esto fue necesario ya que resulta muy difícil evaluar una tecnología y dirimir sobre lo que se puede o no hacer con ella sin conocerla a cabalidad. Se tuvo que ahondar en especificaciones de cada estándar y revisar el soporte actual en navegadores. La solución escogida, utilizar Graphviz re-compilado para JavaScript surgió después de agotar cada otra alternativa.

Ciclo de desarrollo

Inicialmente, se desarrolló una versión del motor de visualización usando HTML y CSS, con algo de JavaScript para manejar la lógica. Esto resultó fácil y rápido para las primeras estructuras desarrolladas. Sin embargo, al llegar a la etapa de dibujar grafos, resultó muy difícil lograr una visualización adecuada. Esto implicó re-implementar el motor de visualización usando otra tecnología.

Como reflexión, quizá de haber empezado por la estructura más compleja (en este caso grafos), probablemente se hubiera indagado antes en las soluciones, conduciendo a un mejor resultado desde el comienzo. Sin embargo, respecto a los plazos, se considera que fue un contratiempo aceptable que no comprometió el resultado final de manera significativa.

Evaluación y compromisos

Como cada tecnología ofrece capacidades de visualización diferentes, esto pone en una fuerte dependencia la estética y el diseño de las visualizaciones. Una tecnología podría ser muy completa, pero si sólo permite dibujar vértices cuadrados, quizá no resulte atractiva para un usuario, siendo de poco valor finalmente. Por lo tanto, cada evaluación que se hizo implicó también explorar las posibilidades visuales, y no sólo técnicas, que ofrecía cada solución. Ante esto, también hubo que hacer compromisos entre estética, facilidad de desarrollo, mantenibilidad, etc.

Capítulo 4

El Lenguaje

4.1. Descripción del Lenguaje

El lenguaje está orientado a la *descripción* de distintas estructuras de datos.

Hay dos aspectos que conviene distinguir en este punto. Por un lado está describir la estructura en sí y los elementos que la componen (por ejemplo, los valores de un arreglo). Sin embargo, también es importante, como se mencionó en la sección 3.2, poder indicar algún tipo de "estado" o particularidad que presenten los datos. Por ejemplo, si se está describiendo un algoritmo de ordenamiento, podría ser útil marcar visualmente los índices de los elementos en operación.

Respecto a las estructuras de datos, el lenguaje, en su primera versión, soporta las siguientes estructuras:

1. Arreglos
2. Grillas
3. Árboles
4. Grafos
5. Mapas
6. Listas Enlazadas

Cada estructura tiene asociada una sintaxis específica en el lenguaje. También se ofrece una sintaxis para poder introducir tipos definidos por el usuario (ver sección 4.6).

Como principio guía, el lenguaje debe ser simple y sencillo de escribir y de entender. No debe exigir caracteres especiales o que no estén disponibles en los esquemas de teclados convencionales.

4.2. Sintaxis

En esta sección se analizarán algunos aspectos de la sintaxis del lenguaje. Esto se hará mediante algunos ejemplos, que si bien no son detallados, son ilustrativos. En el Capítulo 5 se presentan ejemplos más completos de uso del lenguaje para cada estructura de datos.

Los Códigos 4.1, 4.2, 4.3 y 4.4 muestran la sintaxis utilizada para describir arreglos, mapas, árboles y grafos, respectivamente. Las demás estructuras pueden describirse en función de éstas asociándoles metadatos (ver sección 4.3).

```
[1, 2, 3]
```

Código 4.1: Ejemplo de sintaxis de arreglos

```
{"name": "Bob", "age": 16}
```

Código 4.2: Ejemplo de sintaxis de mapas

```
a {
  b
  c
}
```

Código 4.3: Ejemplo de sintaxis de árboles

```
(a) --> (b) --> (c)
```

Código 4.4: Ejemplo de sintaxis de grafos

El lenguaje tomó elementos de los siguientes lenguajes: JSON, CSS, Python, Elixir, GraphQL y Cypher (ver sección 6.0.1 para más detalles.)

El lenguaje ofrece compatibilidad completa con documentos escritos en JSON. De hecho, la gramática del lenguaje extiende la gramática de JSON, por lo tanto, cualquier documento escrito en JSON se puede visualizar utilizando el lenguaje.

Algunas consideraciones sobre la sintaxis:

1. El lenguaje ignora los espacios en blanco y cualquier tipo de salto de línea.
2. El lenguaje no impone un estilo de identación o formato, pero sí recomienda disponer los datos de manera que resulte más fácil de entender para el usuario o potenciales lectores.
3. El lenguaje utiliza los siguientes símbolos para su interpretación:
   (a) Corchetes cuadrados y curvilíneos: `[ ] { }`.
   (b) Paréntesis: `( )`
   (c) Coma: `,`
   (d) Comilla doble: `"`
   (e) Arroba: `@`
   (f) Signo de exclamación: `!`

(g) Porcentaje: `%`

(h) Guión: `-`

(i) Signos de menor y mayor que: `< >`

4. El lenguaje es agnóstico a cualquier idioma natural, no teniendo ni definiendo palabras claves o reservadas en ninguno de ellos.

5. Como excepción al punto anterior, se definen las palabras `true`, `false`, `null` y `nil` (sin sensibilidad de mayúsculas). Esto se decidió por la ubicua presencia de estos identificadores en los lenguajes de programación y porque de lo contrario el lenguaje no sería compatible con JSON. Finalmente, se consideró que la mayor parte del tiempo, el usuario efectivamente quiere introducir estos valores al escribir estas palabras. En caso contrario, se pueden utilizar comillas para definir estas palabras como cadenas de texto.

En un principio se consideró la idea de una sintaxis determinada por bloques identados, como en Python. Esto permitiría definir algunas estructuras de manera más sencilla y natural. Sin embargo, se descartó esta idea por las siguientes razones:

1. Restringe al usuario, al verse obligado a escribir de manera identada ciertas estructuras. Si el usuario quisiera, por determinada razón, escribir de otra forma su documento, no podría hacerlo (y se consideraría un error).

2. Complica la implementación del intérprete. En particular, el *lexer* debe guardar, en un estado interno, el nivel de identación actual y emitir *tokens* `INDENT` y `DEDENT` cuando hayan cambios de identación. El intérprete luego utiliza estos *tokens* para determinar el alcance léxico de cada bloque. Aunque ANTLR, la biblioteca utilizada para construir el intérprete, disponía de aportes de la comunidad para resolver este problema, no se consideró que fueran soluciones suficientemente mantenibles.

3. Sólo una de las estructuras del lenguaje (árboles) se beneficiaría de una sintaxis identada. Las otras estructuras, como arreglos o mapas, sólo dependen de los símbolos utilizados para su delimitación, por lo que se consideró que los beneficios no superaban los costos.

De cualquier modo, el lenguaje se podría beneficiar de la identación presente en el archivo. Dado que los *tokens* generados por el lexer del lenguaje contienen la posición en la que aparecen en el archivo fuente (filas y columnas de inicio y fin), el intérprete podría realizar inferencias respecto al formato deseado. Por ejemplo, los Códigos 4.5 y 4.6 muestran un arreglo escrito de distintas formas. Aunque para el lenguaje son la misma estructura, el intérprete podría analizar la información presente en los *tokens* e inferir correctamente que un arreglo es horizontal y el otro vertical. Esto no es algo que se haya implementado en el lenguaje, pero es una funcionalidad muy atractiva para implementar en un trabajo futuro.

```
[1 2 3]
```

Código 4.5: Elementos de un arreglo con la misma fila de inicio

```
[
  1
  2
  3
]
```

Código 4.6: Elementos de un arreglo con la misma columna de inicio

4.3. Metadatos

El lenguaje permite asociar metadatos a las estructuras definidas. Así, por ejemplo, se puede especificar que un elemento se considere "enfatizado" o que un arreglo se muestre verticalmente. Esta asociación entre estructura y metadato se realiza a través de anotaciones que se escriben junto a la estructura.

El lenguaje ofrece dos variantes para escribir las anotaciones, la primera, utilizando el símbolo arroba (`@`) y la segunda utilizando el signo de exclamación (`!`). Las anotaciones que utilizan una arroba se escriben a la *izquierda* del objeto, mientras que la variante con el signo de exclamación se escribe a la *derecha*. Esto se ejemplifica en los Códigos 4.7 y 4.8, respectivamente.

```
@vertical [1, 2, 3]


@vertical
[1 2 3]
```

Código 4.7: Metadata por la izquierda (o arriba)

```
[1, 2, 3] !vertical


[1 2 3]
!vertical
```

Código 4.8: Metadata por la derecha (o abajo)

Esta diferencia es netamente estética y le permite al usuario elegir la variante que le resulte más apropiada según su estructura. Hay muchos lenguajes que utilizan el símbolo arroba como descriptor de metadatos. Por ejemplo, en Java se utilizan para añadir anotaciones a los métodos y clases de un programa. En Python se utilizan para decorar funciones y clases.

```
@Override
public void speak() {
    System.out.println("Meow.");
}
```

Código 4.9: Anotación en Java

```
@classmethod
def foo(cls, arg1):
    pass
```

Código 4.10: Decorador en Python

Una observación importante en este punto es que el lenguaje no define ningún metadato por defecto, sino que simplemente define el *mecanismo* mediante el cual se pueden definir.

4.4. Arquitectura del Sistema

La representación de una idea y la idea en sí misma pueden llegar a ser cosas muy diferentes. Una visualización que puede resultar atractiva para una persona podría no serlo para otra. Esto motivó la siguiente decisión: el sistema contará con dos componentes, un **intérprete** (o *parser*) y un **motor** (o *engine*).

El **intérprete** es el responsable de tomar un texto de entrada y transformarlo a una representación intermedia.

El **motor** es el responsable de generar una visualización apropiada a partir de esta representación intermedia.

Adicionalmente, se definen **filtros** que operan entre el intérprete y el motor, transformando la representación intermedia.

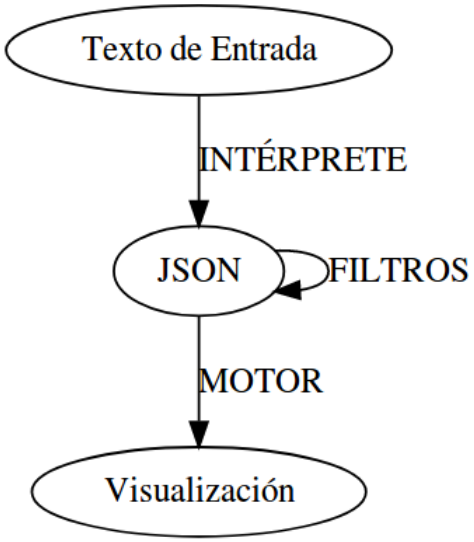


Figura 4.1: Arquitectura del Sistema

La representación intermedia es un documento en formato JSON.

La salida del motor puede ser una imagen o algún otro documento que permita una visualización. Por ejemplo, se puede generar un documento HTML que incluya animaciones e interacciones o simplemente generar una imagen en formato JPEG o similares.

4.5. Representación Intermedia

La representación intermedia en formato JSON permite el desacoplamiento entre intérprete y motor. Este desacoplamiento tiene varias ventajas:

1. Se pueden implementar distintos motores de visualización manteniendo constante la representación intermedia (o entrada, desde el punto de vista de los motores).

2. No es necesario que el intérprete y el motor estén escritos en el mismo lenguaje de programación.

3. Permite desarrollar y mejorar el intérprete y el motor de manera separada.

4. Permite a un usuario utilizar sólo el motor de visualización si así lo prefiere. Esto podría llegar a ser muy útil si las visualizaciones son generadas dinámicamente por una máquina.

Como desventaja, la representación intermedia introduce un contrato o interfaz que debe definirse y mantenerse entre el intérprete y motor.

La representación intermedia obedece a la estructura definida en el Código 4.11.

```
{
  "type": <string: tipo de estructura>,
  "data": <any: el dato>,
  "meta": <object: metadatos asociados por la izquierda>,
  "over": <object: metadatos asociados por la derecha>
}
```

Código 4.11: Estructura JSON intermedia

Como un documento puede tener muchas estructuras, el lenguaje define un tipo, llamado `document`, que actúa como contenedor global (ver Código 4.12).

```
{
  "type": "document",
  "data": [
    {
      "type": "array",
      "data": [1, 2, 3],
      "meta": {},
      "over": {}
    }
  ],
  "meta": {},
  "over": {}
}
```

Código 4.12: Estructura JSON de un documento

4.6. Extensibilidad

La sintaxis define un mecanismo de extensión del lenguaje para incluir estructuras definidas por el usuario. La sintaxis de este mecanismo está inspirada en el tipo `struct` del lenguaje Elixir [8]. El Código 4.13 define un tipo de dato llamado `User`, el que posee dos atributos, `name` y `age`.

```
%User{
  name: "Alexis",
  age: 25
}
```

Código 4.13: Sintaxis para registros

El intérprete internamente denomina a esta estructura un "`record`" (registro) y aplica la misma gramática que para los mapas (objetos JSON).

Para que estas estructuras puedan visualizarse el motor debe soportarlas. En primera instancia el usuario tendría que extender el motor y definir su propia función de visualización. En un futuro, podrían existir muchas estructuras y visualizaciones contribuidas por terceros y el usuario sólo tendría que escoger el tipo adecuado.

Actualmente el motor visual no reconoce ningún tipo definido por el usuario, por lo que estas estructuras no llegan a visualizarse ni fueron incluídas en la plataforma web.

Capítulo 5

Estructuras de Datos

5.1. Arreglos

Descripción de la Estructura

Los arreglos son una de las estructuras de datos más utilizadas en computación. Almacenan un conjunto de elementos, normalmente del mismo tipo, de manera contigua en memoria. Esto hace que recorrer un arreglo sea muy eficiente, pues la CPU no necesita hacer cálculos adicionales, sino simplemente recorrer la memoria. Los arreglos suelen ser de un tamaño fijo determinado al momento de crear el arreglo. Dada la forma en la que se almacena esta estructura, resulta muy eficiente obtener un elemento dada su posición en el arreglo (índice).

Sintaxis en el lenguaje

La sintaxis para describir arreglos en el lenguaje está inspirada en JSON, utilizando corchetes (`[ ]`) y comas (`,`) para delimitar el inicio, el fin del arreglo y separar los elementos, respectivamente. Como extensión propia del lenguaje, las comas se vuelven opcionales, permitiendo separar los elementos sólo con espacios en blanco.

```
1 [1, 2, 3]
2
3 [1 2 3]
```

Código 5.1: Sintaxis de Arreglos

Ejemplos de Visualizaciones

```
[1, 2, 3]
```

Código 5.2: Arreglo simple

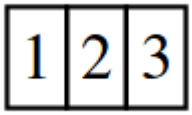


Figura 5.1: Visualización de arreglo simple

```
[cat dog bird]
```

Código 5.3: Arreglo de palabras

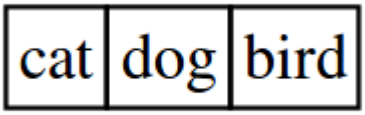


Figura 5.2: Visualización de arreglo de palabras

```
[1 2 3 [a b c]]
```

Código 5.4: Arreglo de distintos tipos

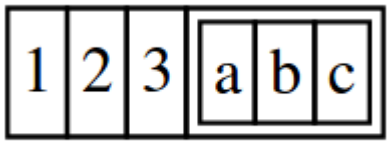


Figura 5.3: Visualización de arreglo de distintos tipos

```
@vertical
[a b c]
```

Código 5.5: Arreglo vertical

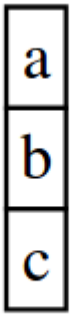


Figura 5.4: Visualización de arreglo vertical

```
[
  a
  b
  c
  [4 5 6] !vertical
]
```

Código 5.6: Arreglo vertical anidado

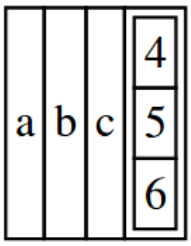


Figura 5.5: Visualización de arreglo vertical anidado

```
@indexedBy(one)
[
  a
  b
  c
  d
  e
]
```

Código 5.7: Arreglo vertical con índices

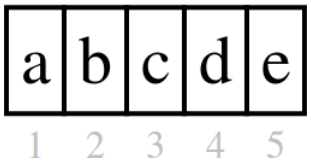


Figura 5.6: Visualización de arreglo con índices

5.2. Grillas

Descripción de la Estructura

Las grillas o matrices son, típicamente, arreglos bidimensionales. Una grilla utiliza dos índices para referenciar a cada uno de sus elementos (fila y columna).

Sintaxis en el lenguaje

En cuanto al lenguaje, las grillas se consideran un caso particular de la sintaxis para arreglos. En particular, una grilla se describe utilizando un arreglo de arreglos. Sin embargo, el lenguaje añade un metadato (`@grid`)[1] que puede utilizarse para indicarle al motor de visualización que muestre la estructura de una manera más conveniente.

```
@grid
[[1 , 2] ,
 [3 , 4]]
```

Código 5.8: Sintaxis de Grillas

Ejemplos de Visualizaciones

```
@grid
[[1 , 2 , 3] ,
 [4 , 5 , 6] ,
 [7 , 8 , 9]]
```

Código 5.9: Grilla simple

1	2	3
4	5	6
7	8	9

Figura 5.7: Visualización de grilla simple

[1]Como nota, el metadato `@grid` no es algo que defina el lenguaje en sí, sino más bien algo que el motor reconoce. El intérprete sólo pasa este valor a través de la representación intermedia.

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

Código 5.10: Grilla sin @grid

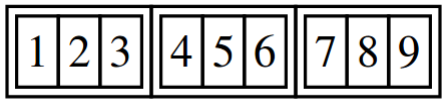


Figura 5.8: Visualización de grilla sin @grid

```
@grid
[[1 2 3]
 [a b c]
 [dog cat egg]]
```

Código 5.11: Grilla sin comas

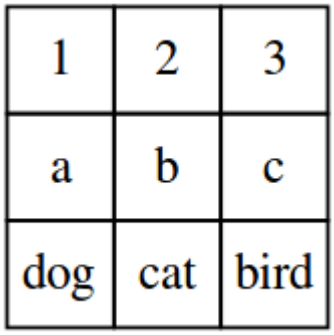


Figura 5.9: Visualización de grilla sin comas

```
@grid @indexedBy("one")
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Código 5.12: Grilla indexada

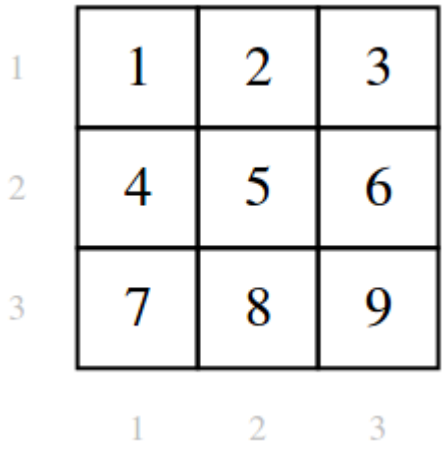


Figura 5.10: Visualización de grilla indexada

```
@grid
[
   [1  2  3]
   [4  5  6]
   [7  8  @grid
          [[1  2]
           [3  4]]]
]
```

Código 5.13: Grilla anidada

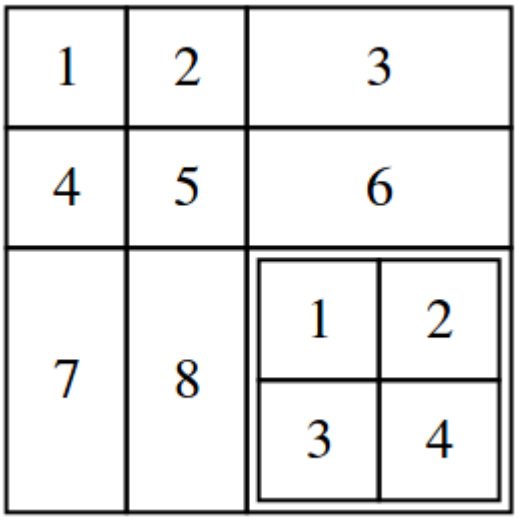


Figura 5.11: Visualización de grilla anidada

```
@grid
[
      [♜ ♞ ♝ ♚ ♛ ♝ ♞ ♜]
      [♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟]
      [- - - - - - - -]
      [- - - - - - - -]
      [- - - - - - - -]
      [- - - - - - - -]
      [♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙]
      [♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖]
]
```

Figura 5.12: Grilla de ajedrez

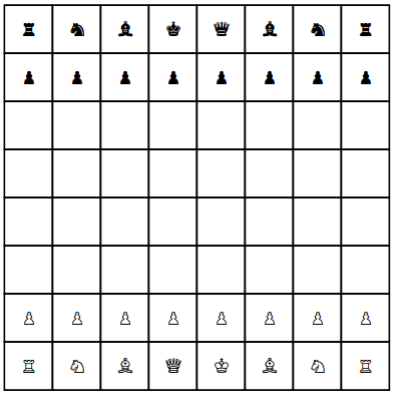


Figura 5.13: Visualización de grilla de ajedrez

5.3. Árboles

Descripción de la Estructura

Los árboles son otra de las estructuras más utilizadas en computación. Un árbol se compone de un nodo raíz del cual nacen una serie de nodos hijos.

Sintaxis en el lenguaje

La sintáxis para diagramar árboles utiliza corchetes curvilíneos (`{ }`) y literales (números, palabras y símbolos). Cada nodo del árbol puede tener una serie de hijos, que se escriben dentro de las llaves respectivas. El Código 5.14 muestra un árbol en el que nodo raíz es `a` y sus hijos, en orden, son `b` y `c`.

```
a {
    b
    c
}
```

Código 5.14: Sintaxis de Árboles

Como el lenguaje no toma en cuenta los espacios en blancos, este mismo árbol puede escribirse, compactamente, como se muestra en el Código 5.15.

```
a { b c }
```

Código 5.15: Sintaxis de Árboles 2

Ejemplos de Visualizaciones

```
a {
    b
    c
}
```

Código 5.16: Árbol simple

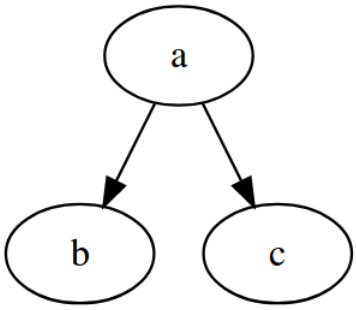


Figura 5.14: Visualización de árbol simple

```
@horizontal
5 {
    1 {
        7
        11
    }
    40 {
        20
    }
}
```

Código 5.17: Árbol horizontal con varios hijos

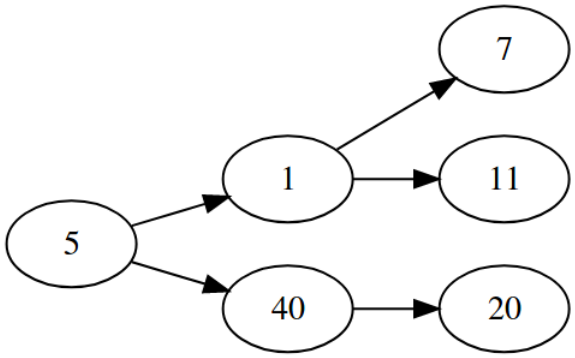


Figura 5.15: Visualización de arbol horizontal con varios hijos

```
@inverted
5 {
    1 {
        7
        11
    }
    40 {
        20
    }
}
```

Código 5.18: Árbol invertido

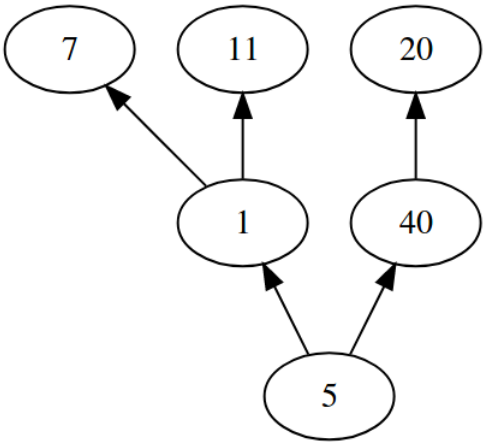


Figura 5.16: Visualización de árbol invertido

```
@horizontal@inverted
5 {
    1 {
        7
        11
    }
    40 {
        20
    }
}
```

Código 5.19: Árbol horizontal e invertido

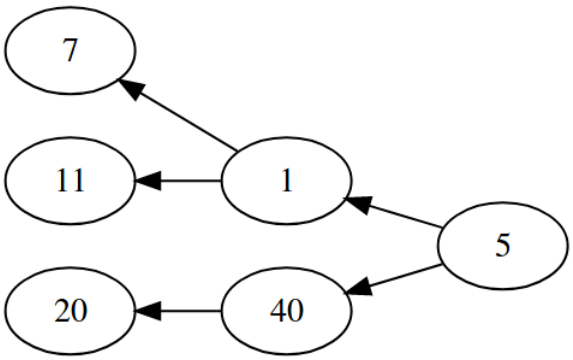


Figura 5.17: Visualización de árbol horizontal e invertido

```
2 {
    7 {
        2 { - - }
        6 {
            5 { - - }
            11 { - - }
        }
    }
    5 {
        9 {
            4 { - - }
            nil
        }
    }
}
```

Código 5.20: Árbol con nodos NULL

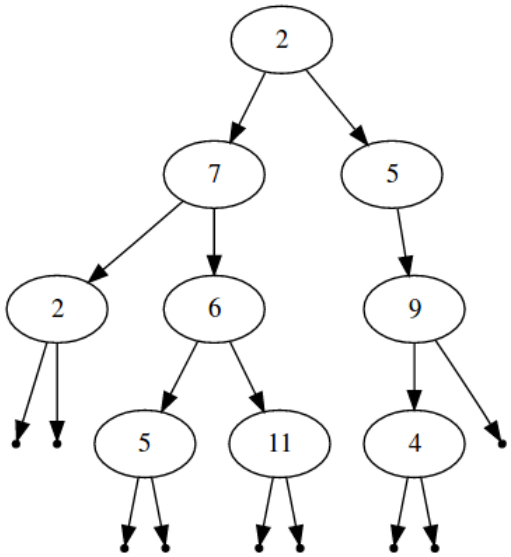


Figura 5.18: Visualización de árbol con nodos NULL

5.4. Grafos

Descripción de la Estructura

Un grafo es una estructura que se compone de un conjunto de nodos y un conjunto de arcos, que conectan dichos nodos entre sí.

Sintaxis en el lenguaje

La sintaxis de grafos utiliza varios símbolos para lograr comunicar visualmente la idea de nodos y arcos. El Código 5.21 muestra la sintaxis para describir un nodo `a` conectado a un nodo `b`.

```
1 (a) --> (b)
```

Código 5.21: Sintaxis de Grafos

Como se puede ver, los nodos se escriben entre paréntesis. Los nodos en un grafo se dibujan típicamente usando círculos o elipses, por lo que usar esta sintaxis resulta intuitivo y natural. Similarmente, los arcos se describen utilizando guiones (`-`) y los símbolos `<` y `>`, lo que crea visualmente una conexión y dirección entre los nodos.

Ejemplos de Visualizaciones

```
1 (a) --> (b)
```

Código 5.22: Grafo simple

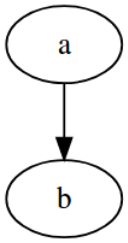


Figura 5.19: Visualización de grafo simple

```
(a : "Label for A")
(b : "Label for B")

(a) --> (b)
```

Código 5.23: Grafo con nodos etiquetados

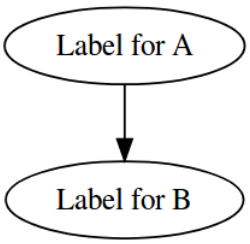


Figura 5.20: Visualización de grafo con nodos etiquetados

```
(cat) -[DISLIKES]-> (dog)
```

Código 5.24: Grafo con arcos etiquetados

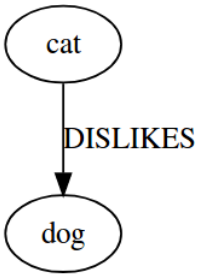


Figura 5.21: Visualización de grafo con arcos etiquetados

```
(a) --> (b) <-- (c) <--> (d)
```

Código 5.25: Grafo con arcos en varias direcciones

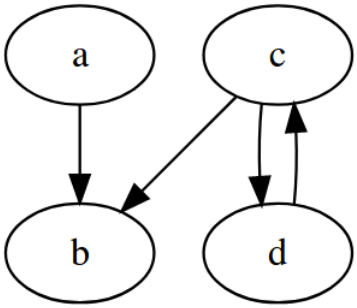


Figura 5.22: Visualización de grafo con arcos en varias direcciones

5.5. Mapas

Descripción de la Estructura

Los mapas son una estructura que permite almacenar pares asociados entre una llave y un valor (`key-value`).

Existen diversas implementaciones de este tipo de dato. Por ejemplo, en C existe el tipo `struct` que permite crear estructuras con campos nombrados. En Python existe el tipo `dict`, que permite crear un diccionario, un objeto que asocia dinámicamente nombres a llaves. En JavaScript los objetos, por defecto, son estructuras asociativas entre llaves y valores.

Sintaxis en el lenguaje

La sintaxis para describir mapas está inspirada en la notación JSON y, de hecho, es compatible con ella. Por lo tanto, un objeto escrito en notación JSON es inmediatamente un mapa en el lenguaje. El Código 5.26 muestra un ejemplo de la sintaxis para describir mapas.

```
{
    "name": "Catalina",
    "age": 8,
    "likes": "flowers and princesses"
}
```

Código 5.26: Sintaxis de Mapas

Ejemplos de Visualizaciones

```
{
    "name": "Catalina",
    "age": 8,
    "likes": "flowers and
   princesses"
}
```

Código 5.27: Mapa simple

name	Catalina
age	8
likes	flowers and princesses

Figura 5.23: Visualización de mapa simple

```
{
    "name": "Catalina",
    "age": 8,
    "likes": {
        "most": "pancakes",
        "least": "veggies"
    }
}
```

Código 5.28: Mapa anidado

name	Catalina
age	8
likes	most: pancakes least: veggies

Figura 5.24: Visualización de mapa anidado

```
@indexedBy("zero")
@vertical
[
    {
        "name": "Catalina",
        "age": 8
    },
    {
        "name": "Jonathan",
        "age": 27
    }
]
```

Código 5.29: Mapa dentro de un arreglo

0	name: Catalina age: 8
1	name: Jonathan age: 27

Figura 5.25: Visualización de mapa dentro de un arreglo

```
{
    "name": "Jonathan",
    "age": 27,
    "likes": [
        "dragons",
        "magic"
    ] !indexedBy("zero"),
    "matrix": [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ] !grid
}
```

Código 5.30: Mapa complejo

name	Jonathan
age	27
likes	dragons (0) | magic (1)
matrix	1 2 3 4 5 6 7 8 9

Figura 5.26: Visualización de mapa complejo

5.6. Listas Enlazadas

Descripción de la Estructura

Una lista enlazada representa una colección lineal de elementos. A diferencia de los arreglos, el orden de los elementos no está determinado por la posición que ocupan en la memoria del sistema, sino que cada elemento de una lista enlazada "apunta" al siguiente. De esta manera, es muy fácil añadir elementos a la lista, pues basta con crear un nuevo nodo y hacer que el último elemento de la lista apunte hacia él. Esto contrasta con los arreglos, en los que agregar un elemento adicional suele ser una operación costosa.

Sintaxis en el lenguaje

Similar al caso de las grillas (ver sección 5.2), la sintaxis para listas enlazadas es un caso particular de la sintaxis para grafos. Del mismo modo, el lenguaje ofrece el metadato `@linkedList` para indicarle al motor de visualización que muestre de una manera más conveniente esta estructura de datos. Esto se ejemplifica en el Código 5.31

```
1 @linkedList
2 (a) --> (b)
```

Código 5.31: Sintaxis de Listas Enlazadas

Ejemplos de Visualizaciones

```
1 @linkedList
2 (a) --> (b) --> (c) --> null
```

Código 5.32: Lista enlazada simple

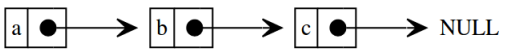


Figura 5.27: Visualización de lista enlazada simple

Capítulo 6

Intérprete

El desarrollo del intérprete se realizó en dos etapas:

1. Diseñar la gramática del lenguaje

2. Programar el intérprete

Para esto se utilizó la biblioteca ANTLR, escogida al final del capítulo 2.

El proceso de desarrollo fue algo cíclico, pues una vez que se generó el primer intérprete se siguieron realizando modificaciones a la gramática, que luego debieron ser incorporadas al intérprete.

En lo siguiente, se describen en mayor detalle estas etapas.

6.0.1. Diseño de la gramática

Para el diseño de la gramática se utilizaron gramáticas ya escritas en ANTLR. En particular, se utilizaron los siguientes gramáticas:

1. **JSON**: La gramática JSON se utilizó para el reconocimiento de los arreglos y mapas del lenguaje, así como las cadenas de texto, números y valores booleanos y nulos.

2. **GraphQL**: La gramática de GraphQL se utilizó para el reconocimiento de árboles, pues la forma en que GraphQL describe los documentos que va a consultar es exactamente la misma sintaxis que se requería (con modificaciones menores).

3. **Cypher**: La gramática de Cypher se utilizó para la descripción de nodos y arcos en grafos.

Todas estas gramáticas están disponibles en el repositorio de ejemplos de ANTLR [3], salvo la de Cypher, que se encuentra publicada en el sitio oficial del lenguaje [12].

Un aspecto que se destaca de estas gramáticas es que están libres de "acciones" o códigos que se ejecuten cuando se reconoce una expresión. Esto permite un grado de portabilidad al que otras bibliotecas, con acciones definidas en un lenguaje de programación específico, no pueden aspirar. Se considera que este es uno de los puntos más fuertes de la biblioteca ANTLR, pues permite el reuso de las gramáticas en otros contextos.

Una adición a la gramática de JSON fue hacer opcionales las comas entre los elementos de un arreglo. Esto facilita el ingreso y edicición de datos, pues el usuario no debe preocuparse de que las comas estén correctamente ubicadas. Si un arreglo en JSON tiene n elementos, entonces debe tener $n - 1$ comas. Esto es un problema, pues cada vez que se quiere agregar un elemento al arreglo hay que agregar, además, una coma. Algunos lenguajes, como Python y PHP, permiten una coma "colgante", al final del último elemento del arreglo. Esto resuelve el problema, pero añade una coma innecesaria realmente. Esta opción fue considerada, pero se descartó porque las comas no juegan un papel en las visualizaciones, pues son descartadas visualmente y sólo se utilizan como separador entre elementos sintácticos. Separar con espacios los elementos no sólo resulta más sencillo, sino que también más intuivo, porque son traducidos, efectivamente, a "espacios en blanco" en la visualización.

Sin embargo, separar los elementos con espacios también introdujo un problema. Inicialmente, el lenguaje permitía asociar metadatos utilizando una arroba (`@`) tanto por la izquierda como por la derecha. Esto generaba una ambigüedad gramatical, como se muestra en los Códigos 6.1 y 6.2. En el primero, el metadatao `@em` pareciera estar asociado al número 2, mientras que en el segundo ejemplo pareciera estar asociado al número 3 (por estar "encima"). Sin embargo, como el intérprete ignora los espacios en blanco, ambos documentos generaban el mismo árbol sintáctico.

```
[
  1
  2 @em
  3
]
```

Código 6.1: Anotación por la derecha

```
[
  1
  2
  @em
  3
]
```

Código 6.2: Anotación por arriba

La solución que se desarrolló a este problema fue agregar otra forma de anotar las estructuras, utilizando un símbolo de exclamación (`!`)[1]. Las anotaciones con `!` se escriben a la derecha (o abajo) de la estructura, mientras que las que utilizan una arroba se escriben a la izquierda (o arriba).

El ejemplo anterior queda representado, con la nueva sintaxis, en los Códigos 6.3 y 6.4.

[1]Esta sintaxis se tomó de CSS, donde se usa la expresión `!important` para sobreescribir un regla.

```
[
  1
  2 !em
  3
]
```

Código 6.3: Anotación por la derecha usando !

```
[
  1
  2
  @em
  3
]
```

Código 6.4: Anotación por arriba usando @

Esta solución resuelve el problema, pero al costo de tener dos sintaxis para la especificación de metadatos. Esto es un problema. Se considera que en futuras versiones del lenguaje, en el que se incorporen mejores formas de anotar el código se podría llegar a una mejor solución.

Un ejemplo de lo anterior puede ser adoptar una parte de la sintaxis de Markdown, en el que se emplean asteriscos para rodear un elemento y así enfatizarlo (`*texto*`). El ejemplo anterior podría reescribirse, nuevamente, como se ejemplifica en los Códigos 6.5 y 6.6. De esta manera no existe ambiguëdad respecto a qué elemento está siendo demarcado, pero se incorpora un significado implícito (el aportado por los asteriscos).

```
[
  1
  *2*
  3
]
```

Código 6.5: Enfatizando el número 2

```
[
  1
  2
  *3*
]
```

Código 6.6: Enfatizando el número 3

Se piensa que el uso del lenguaje por parte de usuarios irá señalando la mejor ruta en cuanto a este tipo de decisiones.

Otra adición que se hizo por encima de la gramática de JSON fue hacer opcionales las comillas para cadenas de texto formadas por una sola palabra. Esto se ilustra en los Códigos 6.7 y 6.8. En el primer caso, las comillas representan la mayor parte del texto escrito, añadiendo "ruido" al documento. El segundo caso las elimina por completo, comunicando de manera mucho más directa los datos.

```
[
  ["a" "b" "c" "d" "e"]
  ["f" "g" "h" "i" "j"]
  ["k" "l" "m" "n" "o"]
  ["p" "q" "x" "y" "z"]
]
```

Código 6.7: Grilla con uso de comillas

```
[
  [a b c d e]
  [f g h i j]
  [k l m n o]
  [p q x y z]
]
```

Código 6.8: Grilla sin uso de comillas

Afortunadamente, esta adición no introdujo ambiegüedades ni otros problemas en la gramática.

Para el desarrollo de la gramática se utilizó la herramienta `grun` (*grammar runner*), provista por ANTLR. Esta herramienta permite probar la gramática contra un texto de entrada y analizar el árbol sintáctico generado, así como los *tokens* reconocidos en el proceso. Adicionalmente, el árbol sintáctico se puede ver y explorar a través de una herramienta gráfica también provista por la biblioteca. Un ejemplo de esto se muestra en la Figura 6.1.

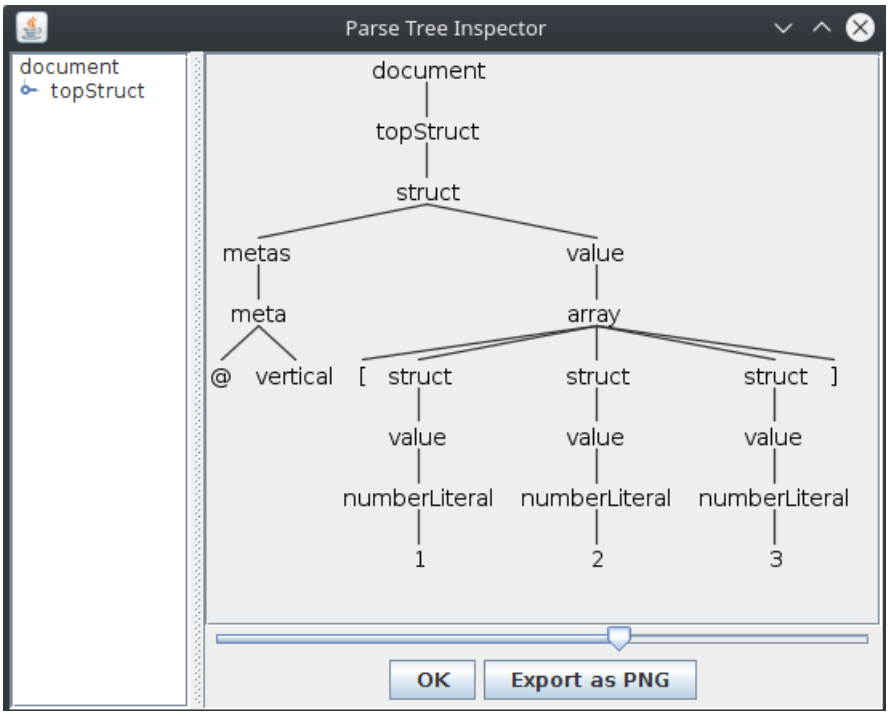


Figura 6.1: Inspector gráfico del árbol sintáctico

La gramática del lenguaje se puede revisar en los anexos del documento (ver Anexo "Gramática del lenguaje").

6.0.2. Programación del intérprete

Para la programación del intérprete se utilizó, nuevamente, el soporte de la biblioteca ANTLR.

Dada una gramática, en este caso llamada `Ve.g4`, se puede generar un intérprete con ANTLR utilizando el siguiente comando:

```
antlr4 -Dlanguage=JavaScript Ve.g4 -visitor -encoding UTF-8
```

Esto genera una serie de archivos que luego son utilizados por ANTLR para la ejecución del intérprete. De todos estos archivos, se destaca el responsable de "visitar" el AST generado por el intérprete de ANTLR, llamado `VeVisitor.js`. Un fragmento de este archivo se muestra en el Código 6.9.

```
1  // Generated from Ve.g4 by ANTLR 4.7
2  // jshint ignore: start
3  var antlr4 = require('antlr4/index');
4
5  // This class defines a complete generic visitor for a parse tree produced by
       VeParser.
6
7  function VeVisitor() {
8    antlr4.tree.ParseTreeVisitor.call(this);
9    return this;
10 }
11
12 VeVisitor.prototype = Object.create(antlr4.tree.ParseTreeVisitor.prototype);
13 VeVisitor.prototype.constructor = VeVisitor;
14
15 // Visit a parse tree produced by VeParser#document.
16 VeVisitor.prototype.visitDocument = function(ctx) {
17   return this.visitChildren(ctx);
18 };
```

Código 6.9: Visitor generado por ANTLR

Por defecto, este visitador lo que hace es visitar cada nodo y luego, recursivamente, visitar cada hijo, sin alterar el orden ni el recorrido.

El principal desarrollo en esta etapa fue extender este visitador generado por la biblioteca por uno propio que pudiera generar el documento JSON con la representación intermedia del lenguaje.

El objeto visitador tiene un método de visita para cada regla sintáctica definida en la gramática. Así, por ejemplo, existen los métodos `visitNumberLiteral`, `visitArray`, `visitEdgeConnector`, etc. Todos los métodos reciben un objeto de "contexto", que contiene los datos del nodo sintáctico actualmente visitado.

La implementación del visitador propio fue relativamente sencilla, pues el intérprete asegura la validez del árbol sintáctico, lo que evita tener que hacer validaciones adicionales.

El Código 6.10 muestra la implementación del método `visitNilLiteral`, que visita todos los valores `NULL` encontrados en el documento. En este caso, el método retorna una estructura en el formato esperado por la representación intermedia del lenguaje. Adicionalmente, verifica si el usuario escribió "null" o utilizó un guión. El motor de visualización hace un distinción en este caso, mostrando la palabra *null* en el primer caso, y no mostrando nada en el segundo.

```
Visitor.prototype.visitNilLiteral = function(ctx) {
  var ret = {
    'type': 'null',
    'data': null,
    'meta': {},
    'over': {}
  };

  if (ctx.dash() != null) {
    ret.meta.dash = true;
  }

  return ret;
};
```

Código 6.10: Implementación de `visitNilLiteral`

El Código 6.11 muestra la implementación del método `visitEdgeConnector`, responsable de construir las conexiones entre los nodos de un grafo. En este caso la regla gramatical es algo más compleja, pues un conector puede escribirse de varias formas (`<--`, `-->`, `<-->` y `--`), lo que obliga verificar la presencia de estos elementos en el nodo del AST.

```
Visitor.prototype.visitEdgeConnector = function(ctx) {
  var ret = {};

  // Possible directions are
  // left: only to the left
  // right: only to the right
  // both: both to the left and to right
  // undirected: no direction, but connected

  // Does this connection point to the left?
  var left = ctx.leftArrowHead() != null;

  // Does this connection point to the right?
  var right = ctx.rightArrowHead() != null;

  var direction = null;
  if (left && right) {
    direction = 'both'
  } else if (left && !right) {
    direction = 'left';
  } else if (!left && right) {
    direction = 'right';
  } else {
    direction = 'undirected';
  }

  ret.direction = direction;
```

```
28
29    // Calculate label
30    if (ctx.edgeLabel() != null) {
31      ret.label = this.visitEdgeLabel(ctx.edgeLabel());
32    }
33
34    return ret;
35  };
```

Código 6.11: Implementación de `visitEdgeConnector`

El desarrollo de `Visitor` resultó fluido y sin mayores problemas. Una dificultad encontrada fue que la documentación de ANTLR y las preguntas y respuestas (*Q&A*) de la comunidad estaban mayormente escritas en Java, dado que es lenguaje por defecto para el cual ANTLR genera intérpretes. Sin embargo, no resultó difícil iniciar una sesión de depuración con el navegador web para explorar la estructura y valores de cada nodo del AST.

Finalmente, resultó sencillo iterar en el ciclo de hacer modificaciones a la gramática, generar un nuevo intérprete con ANTLR y luego modificar el visitador para acomodar estos cambios.

Capítulo 7

Motor de Visualización

El motor de visualización es el responsable final de generar las visualizaciones del lenguaje. Para esto, recibe como entrada la representación intermedia producida por el intérprete y produce como salida una visualización.

El motor desarrollado puede producir imágenes en formato SVG, PNG y PS. Esto está determinado, principalmente, por las capacidades de las bibliotecas utilizadas internamente.

Como se mencionó al final del capítulo 3, se decidió utilizar la biblioteca **Graphviz** para la generación de visualizaciones. Esto es posible gracias a la biblioteca **Viz.js**[15], que realiza la compilación cruzada desde el código fuente de Graphviz, escrito en C, a JavaScript[1] permitiendo ejecutar la bibliteca en un navegador web.

El desarrollo del motor se dividió en dos etapas:

1. Diseño de las visualizaciones

2. Programación del motor

En lo siguiente, se describen en mayor detalle estas etapas.

7.0.1. Diseño de las visualizaciones

Para utilizar Graphviz se debe escribir un archivo en formato `DOT`, que es la sintaxis con la que Graphviz describe grafos. El Código 7.1 y la Figura 7.1 muestran un ejemplo de sintaxis `DOT` con el respectivo grafo generado por Graphviz.

[1]Específicamente, Viz.js utiliza Emscripten para compilar el código C de Graphviz a la representación intermedia de LLVM, la que luego es compilada a asm.js

```
digraph G {
  Hello -> World
}
```

Código 7.1: Ejemplo de DOT

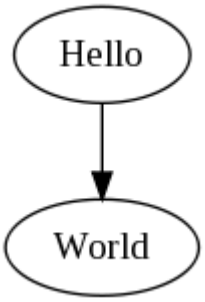


Figura 7.1: Ejemplo de Graphviz

El principal desarrollo en esta etapa fue elaborar los documentos DOT que generaran visualizaciones apropiadas para cada estructura de datos.

El Código 7.2 y la Figura 7.2 muestran un ejemplo del documento DOT utilizado para generar la visualización de un arreglo.

```
digraph G {
  arr [
    shape=plain
    label=<
      <table border="0"
             cellborder="1"
             cellspacing="0">
        <tr>
          <td>1</td>
          <td>2</td>
          <td>3</td>
        </tr>
      </table>
    >
  ]
}
```

Código 7.2: Ejemplo de etiqueta en DOT

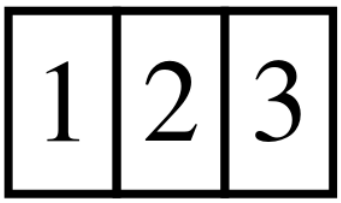


Figura 7.2: Ejemplo de arreglo en Graphviz

Como se puede ver, Graphviz permite el ingreso de código HTML para la definición de las etiquetas de cada nodo. Aunque esta visualización pareciera representar 3 elementos, en realidad es sólo un nodo con una etiqueta más "compleja". El soporte de Graphviz para la inclusión de HTML es bastante primitivo, pues no implementa realmente un motor de visualización como los navegadores web, sino que adopta un subconjunto de la sintaxis de HTML (esencialmente tablas) y lo interpreta internamente. No hay, por ejemplo, una manera sencilla de diagramar componentes o de aplicar hojas de estilo CSS, como permitiría un documento HTML verdadero.

No obstante lo anterior, el espectro de opciones que Graphviz ofrece para estas etiquetas "tipo HTML" es suficiente para los própositos del motor de visualización. Incluso, hay muchas opciones disponibles en Graphviz que no se están usando aún y generarían visualizaciones más atractivas, como la personalización de tipografías, el color de texto, estilización de bordes

y márgenes, etc.

Graphviz representa una solución buena y suficiente, pero no ideal. Esto debido a las siguientes razones:

1. Como el soporte de `HTML` es incompleto, esto limita enormemente las posibilidades de diseñar visualizaciones más complejas.

2. No permite definir componentes que luego puedan reutilizarse múltiples veces. Cada vez que se escribe un arreglo, por ejemplo, éste debe escribirse con toda su definición, lo que se vuelve repetivo y verboso.

3. La mayor ventaja de Graphviz es que posee algoritmos para determinar una buena disposición de los nodos (*layout*). Sin embargo, esta ventaja también se vuelve una desventaja, pues es muy difícil posicionar libremente los elementos.

7.0.2. Programación del motor

El motor de visualización realiza, principalmente, dos tareas:

1. Producir un documento `DOT` a partir de la representación intermedia de entrada.

2. Producir una imagen utilizando `Viz.js` a partir del documento `DOT`. Actualmente el formato de imagen es SVG.

La sencuencia anterior se puede ver en la Figura 7.3.

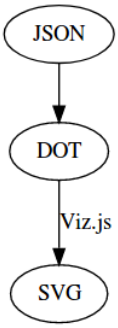


Figura 7.3: Funcionamiento del motor

La estructura de un documento `DOT` es jerárquica, pues en el nivel superior está el "grafo", que posee una serie de "hijos", que pueden ser definiciones de nodos o definiciones de arcos.

Dado que la estructura de la representación intermedia también es jerárquica, se decidió crear una función que, actuando como un "visitador", convirtiera el árbol producido por el intérprete y lo convirtiera en un árbol sintáctico de Graphviz.

Para realizar lo anterior, se definió una clase `GraphvizAST`, que permite definir un nodo del árbol resultante. La implementación de esta clase se muestra en el Código 7.3.

```
class GraphvizAST {
    constructor () {
        this.children = [];
    }

    open () {
        return '';
    }

    close () {
        return '';
    };

    appendChild (child) {
        this.children.push(child);
    };

    prependChild (child) {
        this.children.unshift(child);
    }

    toString () {
        var lines = [];

        lines.push(this.open());

        this.children.forEach(function (child) {
            lines.push(child.toString());
        });

        lines.push(this.close());

        return lines.join('');
    };
}
```

Código 7.3: Clase `GraphvizAST`

El motor define muchas clases que heredan de `GraphvizAST`, como por ejemplo la clase `GraphvizTable`, que se muestra en el Código 7.4. Esta clase es la responasble de producir las etiquetas "tipo HTML" que permiten visualizar muchas de las estructuras del lenguaje.

```
class GraphvizTable extends GraphvizAST {
    constructor () {
        super();

        this.attrs = {
            "border": "0",
            "cellborder": "1",
            "cellspacing": "0",
            "color": "black"
        };
    }

    open () {
        return "<table" + html_attrs(this.attrs) +  ">\n";
    }

    close () {
        return "</table>\n";
    }
}
```

Código 7.4: Clase `GraphvizTable`

Para ilustrar de mejor forma el funcionamiento del motor, el Código 7.5 muestra un arreglo en el formato de la representación intermedia y la Figura 7.4 muestra el árbol sintáctico producido al interior el motor.

```
{
  "type": "array",
  "data": [1, 2, 3],
  "meta": {},
  "over": {}
}
```

Código 7.5: Arreglo en representación intermedia

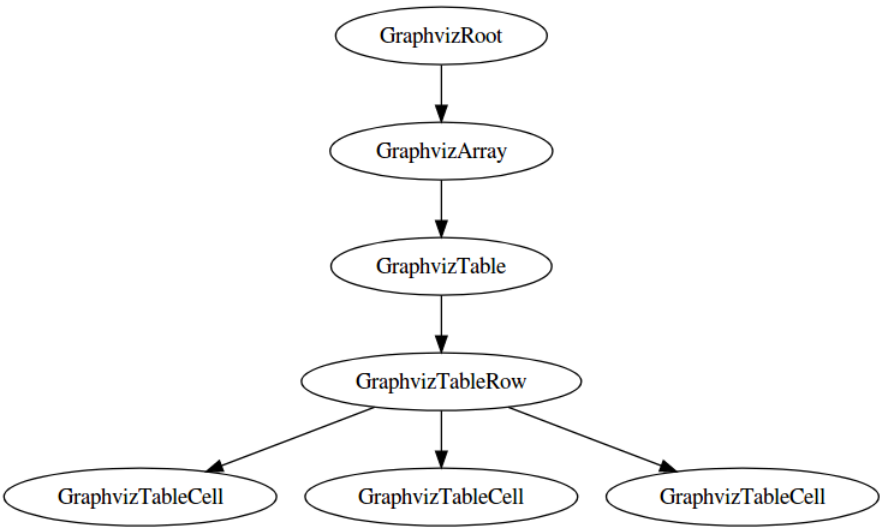


Figura 7.4: Árbol sintáctico producido por el motor

Una vez recorrido completamente el documento de entrada y producido el árbol sintáctico, se invoca el método `toString` de la clase `GraphvizAST`, el que recursivamente construye el resultado final.

Este diseño basado en clases y nodos de un árbol sintáctico permite fácilmente reutilizar componentes, por ejemplo las tablas, en otras estructuras, además de permitir la modificación del árbol durante su generación.

Una vez producido el documento en formato `DOT` éste se pasa a la biblioteca `Viz.js` para la producción final de la visualización por parte de Graphviz.

Capítulo 8

Aplicación Web

8.0.1. Motivación y descripción

Un eje motivacional importante de esta memoria fue poder generar una herramienta útil y aplicable a la vida real.

Con esto en mente, se desarrolló una aplicación web que permite utilizar el lenguaje de manera interactiva para generar visualizaciones en tiempo real. Esta aplicación cuenta con una serie de ejemplos que permiten explorar las distintas estructuras de datos soportadas por el lenguaje.

La Figura 8.1 muestra una captura de la pantalla inicial de la aplicación.

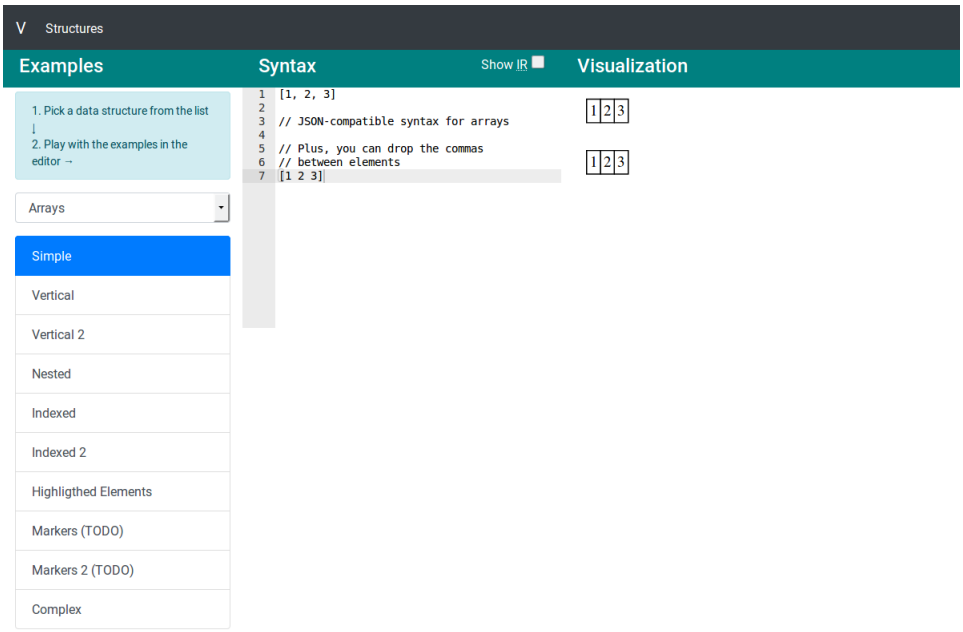


Figura 8.1: Pantalla de inicio de la aplicación

La aplicación web se compone, visualmente, de 4 componentes:

1. Selector de ejemplos
2. Editor de código fuente
3. Editor de la representación intermedia (oculto por defecto)
4. Visualizador

La Figura 8.2 muestra estos componenten según la enumeración previa.

Figura 8.2: Componentes de la aplicación

Los ejemplos se presentan agrupados según estructura de datos. El usuario puede elegir de entre todas las estructuras soportadas por el lenguaje a excepción de los tipos definidos por usuario, que no están soportados por el motor de visualización en esta primera versión.

Una vez seleccionado un ejemplo, la aplicación carga el código del ejemplo en el área de edición, ejecuta el intérprete y genera la visualización correspondiente.

La aplicación carga por defecto la categoría de "Arreglos" y muestra el primer ejemplo asociado.

El usuario puede realizar cambios en el área de edición de código y ver estos cambios reflejados en tiempo real en la visualización.

Opcionalmente, el usuario puede elegir mostrar una segunda área de edición en la que se encuentra la representación intermedia del lenguaje (formato JSON). Si lo desea, el usuario también puede realizar cambios en esta área y ver reflejados los cambios en el visualizador.

Una observación importante es que, dado que tanto el intérprete como el motor de

visualización están escritos enteramente en JavaScript la aplicación es, efectivamente, una aplicación de una sola página (*Single Page Application*) que no necesita de un servidor web para funcionar, sino sólo ser cargada en el navegador de manera estática una vez.

Adicionalmente, la aplicación no necesita realizar conexiones adicionales ni disponer de Internet para interpretar un trozo de código o generar una visualización, pues todas estas tareas se realizan localmente en el navegador del usuario.

8.0.2. Desarrollo de la aplicación

Inicialmente la aplicación era sólo una página HTML con código JavaScript inscrustado e implementaba la versión más primitiva del motor de visualización, sin tener soporte para la interpretación del lenguaje.

Progresivamente se fueron incorporando tecnologías que mejoraban el funcionamiento del sistema. Entre ellas se destacan: Bootstrap[5], Ace Editor[1] y Webpack[16].

1. **Bootstrap**: Para la creación de componentes visuales como selectores, listas y barras de navegación. Así como para la diagramación de la página.

2. **Ace Editor**: Para el manejo del componente de edición de código.

3. **Webpack**: Para concatenar todos los archivos fuentes y cargarlos de manera edificiente en el navegador.

Como última actividad de desarrollo, la aplicación fue cargada a Heroku[10], una plataforma *cloud* que permite el despliegue de aplicaciones de manera sencilla y gratuita (en su plan más básico).

Al estar disponible en Internet, la aplicación pudo ser utilizada en la experimentación y validación con usuarios (ver sección 9.2).

8.0.3. Arquitectura de la aplicación

Como la aplicación es sólo una página HTML estática (enriquecida con JavaScript) que se ejecuta en el navegador del usuario, no se requiere la presencia de un servidor web para programación *backend* o de un servidor de gestión de bases de datos para persistencia de datos.

Sin embargo, para el despliegue de la aplicación en Internet se requirió el uso de un servidor web para servir, de manera estática, los archivos HTML y JavaScript utilizados por la aplicación. Este funcionamiento se puede ver en la Figura 8.3.

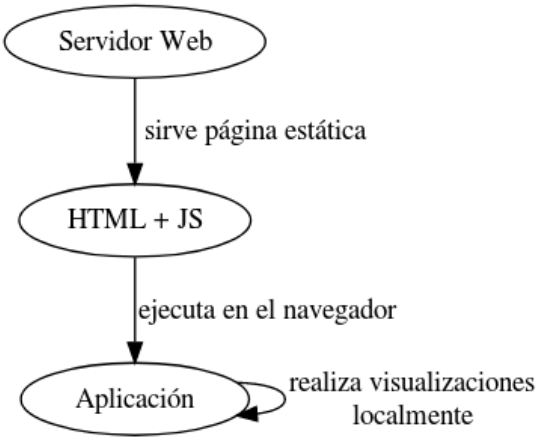


Figura 8.3: Funcionamiento de la aplicación web

Capítulo 9

Aceptación

9.1. Aceptación Tecnológica

Toda herramienta tecnológica busca solucionar una problemática o necesidad específica de un usuario. Esto es, una herramienta debe ser de utilidad al usuario. Por lo mismo, no debe significar un costo o esfuerzo mayor que el que ya emplea en su problemática o necesidad. En otras palabras, el beneficio logrado con la herramienta debe superar el costo percibido por el usuario para su dominio y correcta utilización. Esta idea se toma del modelo de aceptación de tecnología, TAM, propuesto por Fred Davis D. [24].

Por lo tanto y aplicando lo anterior al tema propuesto en esta memoria, se detallan algunos aspectos del lenguaje que, bajo un análisis, justificarán y facilitarán la adopción del mismo una vez implementado.

En primer lugar, el lenguaje debe resultar natural, fácil de entender y escribir (similar en espíritu a Markdown [31]). Esto, al utilizar una sintaxis de texto plano sin demasiados marcadores ni caracteres especiales.

La sintaxis del lenguaje guarda cierta analogía con la estructura representada y su visualización. Así, por ejemplo, la forma de escribir un árbol binario resulta visualmente parecida a la forma en que después se visualiza. Esto permite que el código mismo, escrito en texto, sirva como un puente entre el código fuente y su visualización.

Para que un lenguaje sea adoptado finalmente, no sólo debe resultar útil y fácil de usar, sino también fácil de integrar a los procesos de trabajo actuales del usuario. Esto es, el "roce" al ser implantado en un flujo de trabajo debe ser mínimo, sin requerir grandes inversiones de tiempo, esfuerzo ni negociaciones.

Una ventaja competitiva de este lenguaje es que no sólo se presenta como una sintaxis, sino como un conjunto de herramientas que el usuario podrá utilizar tanto para aprender el lenguaje en sí, como para apoyarse en su uso continuado.

En particular, la aplicación web desarrollada permite que un usuario, de manera sencilla y rápida, genere visualizaciones utilizando el lenguaje, sin requerir descargas, instalaciones o permisos especiales.

Es importante que una herramienta demuestre tempranamente su valor real y sus costos. Permitiendo que el usuario utilice el lenguaje a través de la aplicación web, se calibra tempranamente el valor de la solución versus el costo o esfuerzo percibido por su uso.

Por otro lado, la integración del lenguaje a repositorios de código fuente existentes no debería presentar dificultades. Dado que el lenguaje emplea texto plano, es muy fácil incorporarlo como comentarios junto al código fuente. Este puede ser el mejor lugar en el que se podría documentar el funcionamiento del código, pues resulta inmediato e inevitable. Al mismo tiempo, es el lugar por defecto en el que un lector buscaría ayuda a la hora de entender el funcionamiento de dicho código. Además, al estar junto al código fuente, el lenguaje se beneficia del versionamiento del mismo. Así, las visualizaciones del código evolucionan junto a él de modo natural y cohesivo.

Finalmente y desde la perspectiva del memorista, el costo requerido para utilizar el lenguaje, esto es, aprender la sintaxis y el uso de la herramienta web, es equivalente y probablemente mucho menor al de cualquier solución ya discutida. Esto, debido a que los costos de integración y la curva de aprendizaje son mucho menores.

Los beneficios, por otro lado, se perciben en la escalabilidad y permanencia que presenta la solución. Fácilmente pueden documentarse todos los métodos y estructuras de un proyecto sin que eso signifique un costo de gestión o un mayor almacenamiento. Al mismo tiempo, todos estos "artefactos" acompañarán al código durante su vida útil, ayudando a cada persona que transite por él a entenderlo de mejor forma.

9.2. Experimentación con Usuarios

Para poner a prueba el lenguaje desarrollado se realizó una experimentación con usuarios reales a través de una encuesta en línea.

Esta encuesta se distribuyó entre pares del memorista, colegas de trabajo, estudiantes de la carrera y algunos docentes del departamento. Se considera que esta población, ligada en su quehacer a la computación, es una muestra representativa de los usuarios que podrían llegar a usar la herramienta.

La encuesta instruye al usuario a ingresar a la aplicación web y realizar algunas tareas.

Las instrucciones y preguntas de la encuesta se detallan a continuación.

9.2.1. Encuesta

```
Aplicación: http://ve-app.herokuapp.com/

Para responder esta encuesta, por favor, use la herramienta para:
    - Ver algunos ejemplos
    - Modificar alguna estructura (ej: agregar elementos a un arreglo,
      nodos a un grafo, etc)
```

1. Su principal actividad:
   (a) Desarrollador o Desarrolladora
   (b) Docente
   (c) Estudiante
   (d) Documentador o Documentadora
   (e) Analista QA
   (f) *Otro...*
2. ¿Encuentra usted que la herramienta...
   (a) No resuelve ningún problema
   (b) Quizá en algún momento me podría servir
   (c) Creo que me sería bien útil
   (d) *Otro...*
3. Feedback (opcional)
   Si cree que la herramienta le sería útil, por favor, diga para qué.
   (a) *Campo de texto libre*
4. El lenguaje:
   (a) Es bastante claro
   (b) Es suficientemente claro
   (c) Es un poco confuso
   (d) Es muy confuso
5. Feedback (opcional)
   Si el lenguaje le pareció confuso, por favor, escriba de qué manera.

(a) *Campo de texto libre*

6. Con respecto a las visualizaciones:

   (a) No hay relación entre lenguaje y visualización

   (b) Hay alguna relación

   (c) Hay suficiente relación

   (d) Hay mucha relación

7. Feedback (opcional)
   Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.

   (a) *Campo de texto libre*

8. ¿Qué más podría tener el lenguaje?

   (a) *Campo de texto libre*

9. ¿Qué está de más en el lenguaje?

   (a) *Campo de texto libre*

10. ¿Cómo es la carga inicial del sistema?

    (a) Rápida

    (b) Regular

    (c) Lenta

11. ¿Cómo es la velocidad del sistema una vez cargado?

    (a) Lenta

    (b) Regular

    (c) Rápida

12. El lenguaje... *(puede responder todas las que quiera)*

    (a) Le recuerda a otros lenguajes que utiliza

    (b) Reemplazaría un lenguaje que ya usa (para visualizar estructuras)

    (c) Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código)

    (d) Me interesa y me gustaría colaborar

(e) Lo usaría para mi trabajo

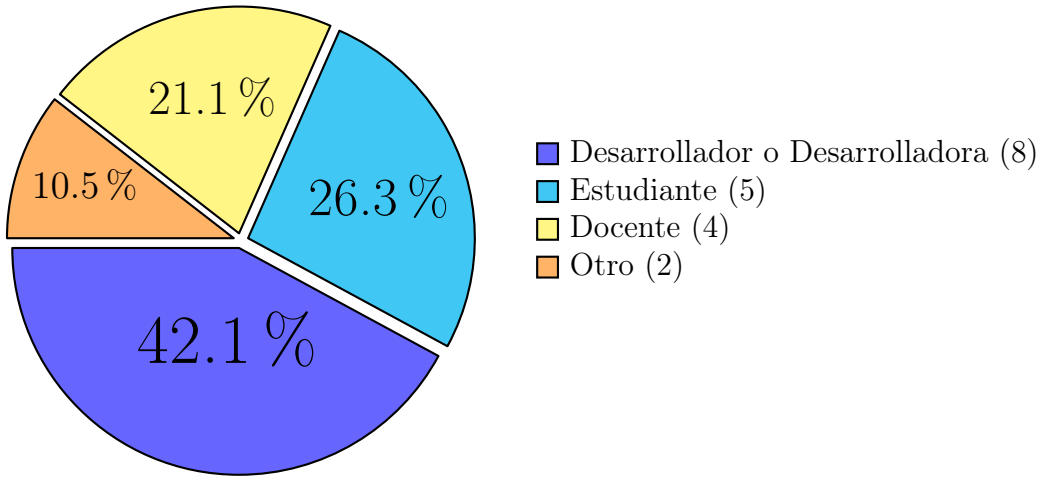
(f) *Otro...*

9.2.2. Resultados

La encuesta fue respondida por 19 usuarios. Se considera que estas respuestas, tanto por cantidad como el nivel de retroalimentación, son suficientes para realizar un análisis significativo sobre la herramienta.

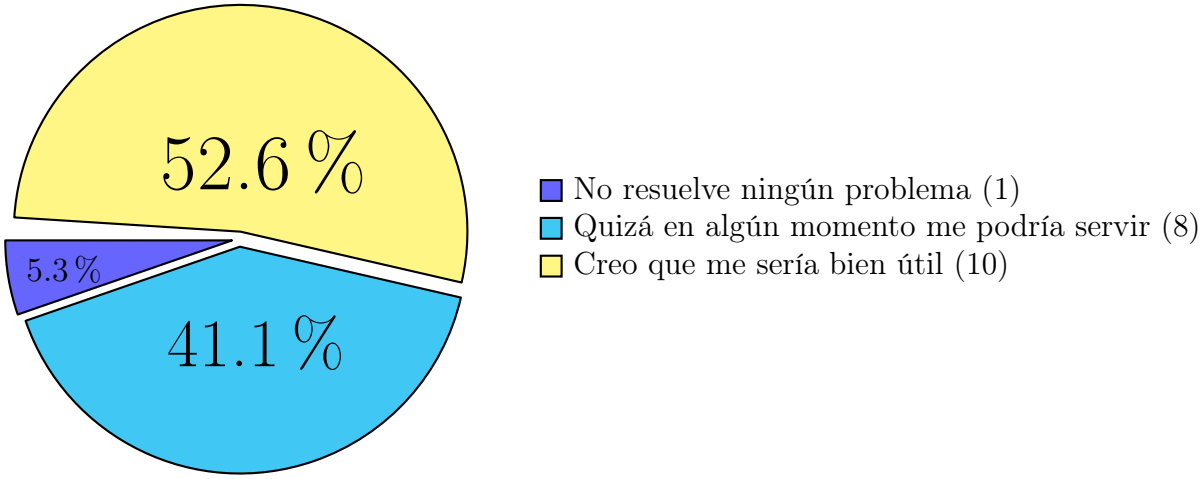
El detalle individual de cada respuesta se puede encontrar en los anexos del documento (ver Anexo "Resultados de la encuesta").

1. **Su principal actividad**



Esta pregunta busca ligar el resto de las respuestas a un patrón común, en este caso, la actividad principal que realiza la persona.

2. **¿Encuentra usted que la herramienta...**



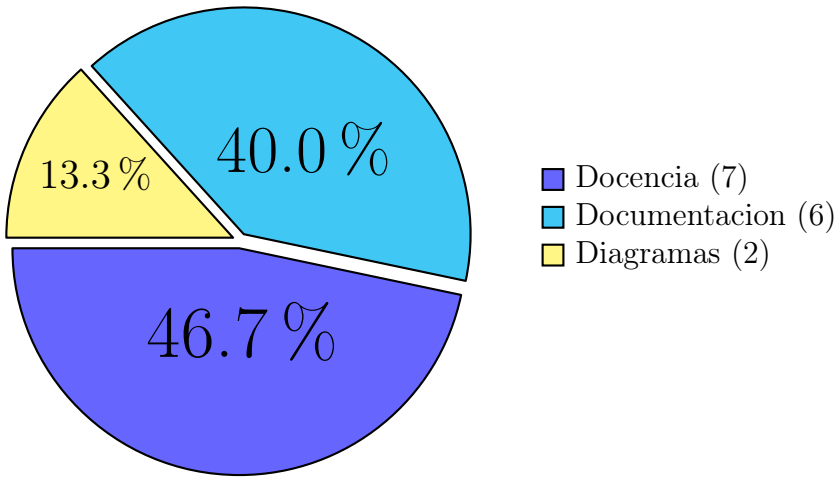
En total, el 93,7 % de los encuestados consideró que la herramienta le podría ser útil. Este es un excelente resultado y confirma las motivaciones detrás del trabajo realizado en esta memoria.

3. *Feedback (opcional)*

   Si cree que la herramienta le sería útil, por favor, diga para qué.

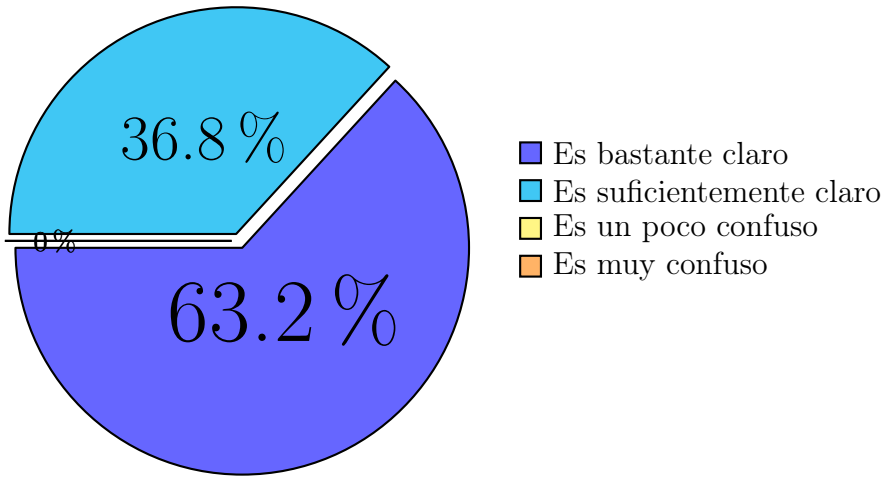
   Para esta pregunta 15 personas decidieron escribir una retroalimentación opcional.

   En este caso, la pregunta es de texto libre, por lo que se realizó una agrupación según la idea principal detrás de cada respuesta.



   Los resultados de esta pregunta son tremendamente valiosos, pues indica que los usuarios pudieron encontrarle un propósito personal a la herramienta de manera casi inmediata. Por otro lado, también se destaca el hecho de que, de los 19 encuestados 15 decidieron escribir en mayor detalle el contexto en el que les resultaría útil la herramienta.

4. **El lenguaje**



Ningún encuestado consideró que el lenguaje fuera confuso. Por el contrario, la gran mayoría consideró que el lenguaje era bastante claro.

5. *Feedback (opcional)*

Si el lenguaje le pareció confuso, por favor, escriba de qué manera

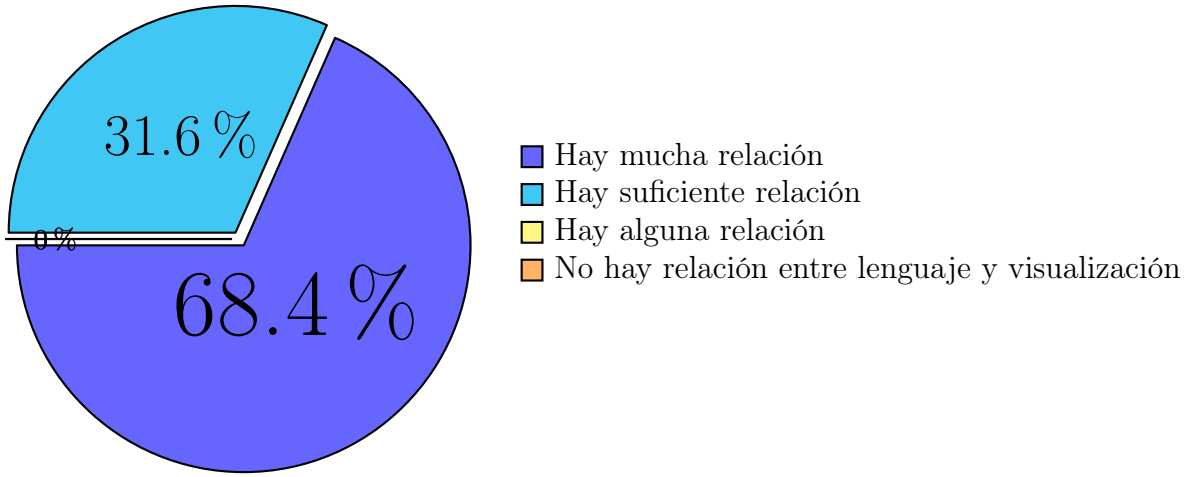
Para esta pregunta 6 personas decidieron escribir una retroalimentación opcional.

Se destaca la siguiente respuesta, pues representa, en esencia, el mayor punto de confusión sentido por los encuestados:

> *"Al lenguaje lo encuentro claro, aunque tiene muchas construcciones sintácticas que sin la ayuda de los ejemplos, llevaría tiempo adoptar. Encuentro además que utiliza "metáforas" diferentes que podrían unificarse. Por ejemplo, utiliza* `@horizontal` *como prefijo e* `!indexedBy` *como sufijo. Indica que la metadata puede incluirse de las dos formas, pero me pregunto si esa flexibilidad favorece al usuario del lenguaje o si hace más compleja la comprensión."*

Que el lenguaje tenga dos sintaxis para la inclusión de metadatos resulta ser un punto de confusión para los usuarios. Pese a que esto resuelve algunas ambiguedades gramaticales, los usuarios consideran contra intuitivo que haya dos formas de realizar la misma acción, sin una diferencia evidente entre ellas.

6. **Con respecto a las visualizaciones**



Todos los encuestados consideraron que existía suficiente relación entre las visualizaciones generadas por el lenguaje y la sintaxis empleada para ello. Más aún, el 68,4 % de los encuestados consideró que esta relación no sólo era suficiente, sino que significativa.

7. *Feedback (opcional)*

Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.

Para esta pregunta 6 personas decidieron escribir una retroalimentación opcional. Las respuestas fueron suficientemente variadas como para que no fuera posible agruparlas en ideas similares y poder graficar aquí los resultados, por lo que citarán algunas de las respuestas para realizar algún análisis.

> *"Me pareció muy genial el poder mostrar las tablas. Lenguajes parecidos requieren ascii muy engorroso de alinear."*

> *"Hay mucha cercanía entre lo que uno escribe, y lo que se ve. Se siente como si el lenguaje entendiera lo que quiero decir."*

> *"Más allá de las potenciales mejoras, creo que el lenguaje ya está en condiciones de ser utilizado."*

> *"Podría cambiarse el color a los arreglos y nodos, por ejemplo pensando en crear estructuras como arboles rojo negro. Asímismo los nodos podrían contener pequeñas listas para dibujar árboles 2-3 o B-Trees."*

Estos comentarios señalan un alto grado de satisfacción con la herramienta.

8. **¿Qué más podría tener el lenguaje?**

   Esta pregunta fue respondida por 13 personas. A continuación se detallan las ideas generales sugeridas por los encuestados junto a algunas observaciones.

   (a) **Transiciones animadas**: esto podría entenderse de dos modos. Por un lado, se podrían animar con transiciones los cambios que va sufriendo la visualización a medida que el usuario modifica el código en el el editor online. También se podría animar la transición entre dos estructuras (calculando su diferencia).

   (b) **Personalización de estilos**: varios usuarios sugirieron la opción de poder cambiar los tipos y tamaños de tipografía, así como personalizar colores y formas de las estructuras.

   (c) **Etiquetado de ramas**: actualmente el lenguaje no define una manera de configurar una etiqueta para las ramas de un árbol (sólo se puede en grafos).

   (d) **Reporte de errores sintácticos**: mostrar los errores sintácticos al momento de editar el código. Actualmente la herramienta no tiene ningún indicador respecto a la validez del texto ingresado y es el usuario el que debe validar, visualmente, que lo que escribió estuvo correcto.

   (e) **Visualización de algoritmos**: como ciclos `for` u ordenamientos. Esta es una sugerencia interesante, pues uno de los objetivos explícitos del lenguaje es no visualizar algoritmos o programas (ejecución), sino estructura de datos (valores). Sin embargo, parece ser una necesidad bastante real. Si se añaden transiciones entre estructuras al lenguaje, no sería difícil simular la ejecución de un algoritmo sencillo.

   (f) **Soporte de estructuras dentro de nodos**: esto permitiría visualizar árboles 2-3, por ejemplo. Actualmente el lenguaje soporta sólo *anidamiento* entre algunas estructuras y, aún así, es algo limitado, pues no todas las estructuras pueden anidarse dentro de otras. Resulta discutible si algunas estructuras deban poder anidarse dentro de otras (por ejemplo, un árbol dentro de un grafo). Sin embargo, en el caso de esta sugerencia, tiene sentido poder anidar arreglos, mapas y grillas dentro los nodos de un grafo o un árbol.

   (g) **Punteros entre estructuras**: esta sugerencia es interesante, pues responde a uno de los aspectos de visualización discutidos en el capítulo 3, específicamente, el concepto de *referenciamiento*. Actualmente, si un arreglo aparece dentro otro, el segundo se dibuja anidado dentro del primero. Sin embargo, se podrían dibujar por separado y luego conectar con alguna línea o flecha apropiadamente.

9. **¿Qué está de más en el lenguaje?**

   Esta pregunta fue respondida por 10 personas.

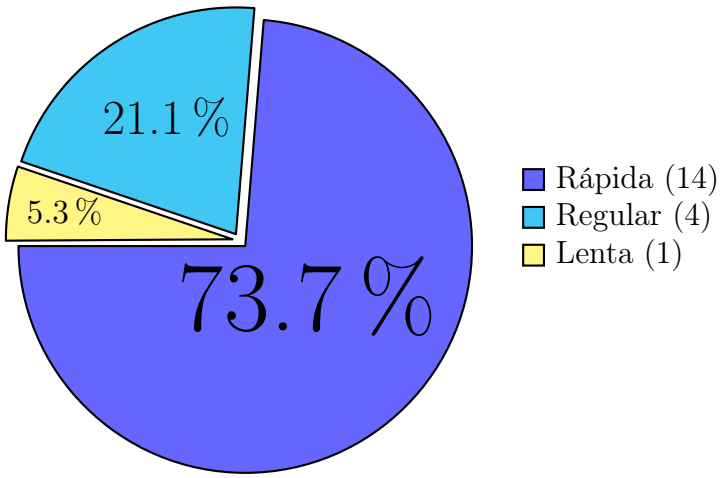
   Se destacan los siguientes comentarios:

   > *"Se me hizo un poco confuso que "!" sirva para agregar meta datos, en ese sentido, la sintaxis para agregar meta datos me parece que hace más complejo el lenguaje."*

   > *"Creo que las decoraciones ensucian un poco la lectura humana del input. Tal vez si hubiera aliases más breves para los metadatos (!v en vez de !vertical, !i en vez de indexedBy)."*

   Nuevamente, la sintaxis para la inclusión de metadatos resultó un punto de confusión. Por un lado, el hecho de que hayan dos mecanismos para su definición (`@` y `!`) y que algunos metadatos sean muy verbosos.

10. **¿Cómo es la carga inicial del sistema?**



    Actualmente, toda la lógica de la aplicación se empaqueta en un solo gran archivo (`bundle.js`), que utiliza aproximadamente `3MB` de almacenamiento en disco. Este archivo debe ser descargado en su totalidad por el navegador para que la aplicación pueda inciar su funcionamiento.

    Esta pregunta se realizó principalmente para medir si esto significaba un problema. Pese a que siempre es mejor reducir el tamaño de las transferencias, no parerece haber impactado negativamente a la mayoría de los usuarios.

11. **¿Cómo es la velocidad del sistema una vez cargado?**

    El 100 % de los encuestados (19) respondió que la velocidad del sistema, una vez cargado, era **rápida**.

12. **El lenguaje...**

	Respuestas
Le recuerda a otros lenguajes que utiliza	12
Reemplazaría un lenguaje que ya usa (para visualizar estructuras)	4
Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código)	14
Me interesa y me gustaría colaborar	5
Lo usaría para mi trabajo	12
No enseño cursos de estructuras de datos, pero si lo hiciera, creo que esta herramienta me ayudaría bastante a comunicar los conceptos detras de estas estructuras.	1
Podría permitir diferentes tipos de salida, como ser html, latex, png, etc.	1

9.3. Análisis de Aceptación

El modelo de aceptación tecnológica *TAM* sugiere que la utilidad y la facilidad de uso son determinantes en la intención que tenga un individuo para usar un sistema.

Este modelo se basa en dos características principales:

1. Utilidad percibida (*Perceived Usefulness*)

2. Facilidad de uso percibida (*Perceived Ease of Use*)

La utilidad percibida (PU) se refiere al grado en que una persona cree que usando un sistema en particular mejorará su desempeño en el trabajo, y la facilidad de uso percibiba (PEOU) señala hasta qué grado una persona cree que usando un sistema en particular realizará menos esfuerzo para desempeñar sus tareas.

Aunque acá no se aplicará el modelo TAM, resulta útil evaluar estas características en función de la experimentación realizada con usuarios.

9.3.1. Utilidad percibida

Respecto a la utilidad percibida, la mayoría de los encuestados contestó que la herramienta les resultaría útil o bastante útil para su actividad (ver pregunta 2).

En particular, se destacan los siguientes comentarios:

> *"Actualmente para mandar diagramas tendría que o sacar una foto a un dibujo o usar ppt para hacerlo, esto parece una alternativa fácil de usar."*

"Para documentación sería útil, dado que se podría dejar ese lenguaje en los comentarios y con un script generar la visualización."

"Encuentro que la herramienta es útil para preparar materiales de los cursos básicos de la carrera de pregrado, principalmente en los cursos de programación, algoritmos y estructuras de datos. También lo encuentro útil para facilitar la creación de diagramas en artículos de investigación. Sin embargo, en este último caso, sería muy útil que la salida sea Latex en lugar de svg."

La utilidad percibida por parte de los usuarios aplica a las áreas de generación de diagramas, documentación y docencia. Al mismo tiempo, estos comentarios están en sincronía con las motivaciones y posibles aplicaciones que se propusieron en esta memoria.

Adicionalmente, la pregunta 12 señaló que 12 de los 19 encuestados utilizaría la herramienta para su trabajo.

Como se mencionó en la introducción de este capítulo, una herramienta tecnológica es útil cuando permite realizar con menor esfuerzo una tarea dada. Y será utilizada en cuanto la utilidad percibida sea mayor que la que le ofrece su herramienta actual.

Por lo tanto, vale la pena analizar algunas alternativas (competencia) utilizadas actualmente por los usuarios para realizar estas tareas:

1. **Fotografías**: un caso es realizar diagramas en papel o pizarra y luego tomar fotografías, que pueden ser adjuntadas en un correo o añadidas a otro documento.

2. **Microsoft PowerPoint** (o Google Drawings): PowerPoint permite incluir en un documento diversas figuras predefinidas. El usuario puede posicionar libremente estas figuras y crear los diagramas que necesite, personalizando colores y estilos. Adicionalnalmente, el usuario podría crear distintas diapositivas y generar transiciones entre ellas que ayuden a explicar el diagrama.

3. **Microsoft Paint** (o similares): Paint ofrece un canvas en el que el usuario puede dibujar libremente utilizando pinceles y figuras predefinidas. Es la opción más básica y primitiva en cuanto a diseño gráfico, pero también es muy sencilla de utilizar y está disponible en todos los sistemas operativos.

Todas estas alternativas ofrecen una flexibilidad mucho mayor a la que ofrece actualmente la herramienta, pues permiten dibujar, ya sea física o digitalmente, de manera libre. Sin embargo, también tienen algunos costos asociados:

1. **Reproducibilidad**: en el caso de una fotografía, es prácticamente imposible volver a generar la imagen si el dibujo o diagrama original se perdió. Más aún, si se quisiera hacer un nuevo diagrama con algunas modificaciones entonces habría que dibujar todo de nuevo (o borrar sobre el original) para poder tomar una nueva fotografía. En el caso de PowerPoint el usuario podría guardar los archivos fuentes (`.ppt/.pptx`) y realizar allí los cambios cada vez que quiera exportar o generar un nuevo diagrama.

2. **Mantenimiento**: Si un usuario quisiera utilizar PowerPoint para documentar su código

fuente tendría, entonces, dos alternativas. La primera es realizar una exportación a algun formato de imagen y almacenar dichas imágenes en el repositorio junto al código fuente. La segunda sería guardar directamente las presentaciones fuente en el repositorio y abrirlas cuando se deseen visualizar.

3. **Programabilidad**: ninguna de estas alternativas permite un acceso programático a través de `APIs` u otros mecanismos. Esto dificulta que puedan ser utilizadas como pasos en una cadena de producción mayor (por ejemplo, en generación automática de documentación).

4. **Especialización**: Al ser alternativas con un propósito más general, no simplifican de ninguna forma la realización de tareas más específicas (como generar un árbol). Esto obliga al usuario a invertir tiempo y trabajo en tareas secundarias (como posicionar correctamente las figuras).

5. **Datos**: Ninguna de estas herramientas está manejada por los datos. Esto, en el sentido de que si el usuario modifica sus datos de origen (por ejemplo, los elementos de un arreglo), debe manualmente actualizar el diagrama o dibujo.

9.3.2. Facilidad de uso percibida

Respecto a la facilidad de uso un punto a destacar es que la herramienta no cuenta con documentación ni tutoriales, ni se realizó ningún tipo de capacitación a los usuarios previa a la encuesta.

Pese a lo anterior, ningún usuario señaló dificultades en su uso. Esto indica que el lenguaje resulta fácil e intuitivo de utilizar de forma natural.

Otro aspecto que se considera clave en este punto es que la aplicación se distribuye por Internet, sin requerir que los usuarios instalen o descarguen nada. Una vez que el sistema carga el usuario puede utilizarlo sin necesitar una conexión a Internet ni tener que esperar por tareas en segundo plano para la generación de las imágenes.

Por otro lado, 12 de los 19 encuestados señalaron que el lenguaje les recuerda a otros lenguajes que utilizan (ver pregunta 12). Esto indica que la decisión de basar el lenguaje en el formato JSON (y extender desde ahí) fue acertada.

Capítulo 10

Aplicaciones

10.1. Documentación de código fuente

El lenguaje podría ser incluido junto a la documentación propia del código fuente, por ejemplo, acompañando la documentación de una clase o método. Luego, se podría configurar una herramienta de documentación automática para que al encontrar fragmentos del lenguaje, los convierta a visualizaciones acordes al formato de la documentación. Algunas herramientas que se pueden utilizar para este fin: Javadoc [40], RDoc [14], Sphinx [20], etc. Esto se realizaría a través de *plugins* específicos para cada herramienta. Debido a que estas herramientas manejan múltiples formatos (por ejemplo, HTML, PDF, EPUB, TEX, etc) es importante que el motor del lenguaje pueda responder acordemente.

El Código 10.1 muestra un ejemplo hipotético en el cual se introdujo código del lenguaje en la declaración de una función en Python (Método `ravel` del módulo de matrices de NumPy [51]). En este caso, se utilizaron las secuencias de caracteres "`###`" para declarar el inicio y fin del fragmento del lenguaje.

```
def ravel(m):
    """
    A 1-D array, containing the elements of the input, is returned. A copy is
    made only if needed.
    ...

    ###
    // input matrix
    [[1, 2, 3]
     [4, 5, 6]
     [7, 8, 9]] !grid
    // apply transformation
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
    ###
    """
    ...
```

Código 10.1: Uso del lenguaje para documentar el método `ravel`

Problema que soluciona: el código fuente suele tener una documentación que explica, en palabras, lo que hace una función. El lenguaje permitiría acompañar estas explicaciones con ejemplos textuales (escritos en el formato del lenguaje), que luego pueden ser visualizados a la hora de generar la documentación del software en cuestión. Esto reduce la brecha mental entre el código fuente (literal y estático) y la ejecución (diagramática y dinámica).

10.2. Docencia

Al contar con una herramienta online que permita ingresar fragmentos del lenguaje y visualizarlos de manera inmediata se podría utilizar el lenguaje para mostrar cómo opera un algoritmo específico. Puede usarse en cursos introductorios a la computación para mostrar operaciones básicas sobre arreglos, listas, tablas de *hash*, etc. En cursos más avanzados se podría usar para ilustrar recorridos y búsquedas sobre árboles y grafos, así como operaciones vectoriales sobre matrices. Uno de los aspectos importantes en el diseño del lenguaje es que se debe poder extender añadiendo más estructuras de datos, como por ejemplo R-Trees [33], con lo cual se podría utilizar para mostrar la operación de algoritmos más específicos.

Problema que soluciona: la docencia suele requerir ejemplos visuales para explicar el funcionamiento de algoritmos. Esto se realiza, usualmente, dibujando sobre una pizarra, preparando diapositivas con ejemplos ya diagramados, utilizando videos o haciendo uso de alguna aplicación interactiva. El lenguaje más su herramienta web podrían acompañar en este proceso, haciendo más mantenibles los ejemplos y permitiendo un trabajo *in situ* con los estudiantes.

10.3. Documentos

Un posible uso del lenguaje, a través de *plugins*, es integrarse a herramientas de generación y publicación de documentos, por ejemplo, LaTeX. En un contexto de docencia, esto podría utilizarse a la hora de generar enunciados de pruebas o trabajos e incluso, en el ámbito de la investigación, publicaciones. En general, se podría incrustar el lenguaje en cualquier otro lenguaje para el cual se haya desarrollado alguna integración. Esto, naturalmente, dependerá en gran medida del soporte del *plugin* y del motor de visualización utilizado.

Problema que soluciona: insertar visualizaciones de estructuras de datos en documentos. Aunque es evidente que ya existen formas de realizar esto, no siempre resulta fácil de hacer, pues la sintaxis puede volverse engorrosa o, directamente, requerir el uso de instrucciones de dibujo ("dibujar" un círculo en las coordenadas x-y, luego una línea entre este círculo y el siguiente, etc").

El Código 10.2 es un trozo de código LaTeX que permite dibujar un árbol (ver Figura 10.1).

```
\begin{tikzpicture}
\Tree
[.50
   [.60 ]
   [.30
      \edge[blank]; \node[blank]{};
      \edge[]; [.40
            \edge[]; {20}
            \edge[blank]; \node[blank
     ]{};
         ]
   ]
]
\end{tikzpicture}
```

Código 10.2: Sintaxis TikZ para describir árboles

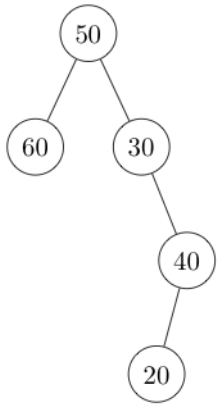


Figura 10.1: Árbol generado por TikZ

Aunque esta sintaxis de TikZ no tiene ningún problema de por sí, plantea el hecho de tener que aprender una sintaxis específica para dibujar árboles, pudiendo haber otras para otras estructuras de datos. Finalmente, este "artefacto" (el trozo de código) sólo es usable en este contexto y pierde valor si la biblioteca subyacente deja de ser soportada.

Conclusión

El objetivo general de esta memoria era desarrollar un lenguaje para la visualización de estructuras de datos.

Dentro de los objetivos específicos se contempló:

1. Definir una gramática para el lenguaje
2. Construir un intérprete para el lenguaje
3. Construir un motor de visualización para el lenguaje
4. Desarrolar una aplicación web que permita utilizar el lenguaje

El entregable del objetivo 1 se adjunta en los anexos 10.3. Los entregables de los demás objetivos son muy extensos como para incluirlos en este documento, sin embargo, se elabora en detalle sobre ellos en sus capítulos respectivos (ver Capítulos 6, 7 y 8).

Dicho esto, se concluye que todos los objetivos de la memoria fueron cumplidos.

Respecto al desarrollo realizado, tanto del intérprete como del motor, se siguió un proceso en el cual se investigaron aspectos teóricos y técnicos detrás de cada tema y se realizó una evaluación ingenieril de las posibles soluciones. Un desafío encontrado en esta etapa fue que la búsqueda y exploración de diversas tecnologías resultó más extensa de lo planificado, lo que retrasó el avance del proyecto en otras áreas. Como consecuencia, gran parte del desarrollo se tuvo que realizar en la etapa final del proyecto, lo que puso una tensión respecto a la completitud y entrega de la memoria.

Respecto a la validación y experimentación con usuarios, se realizó un encuesta a diversas personas relacionadas al área de la computación, entre ellos colegas del memorista y compañeros y docentes de la facultad. Los resultados de esta encuesta validaron positivamente la herramienta desarrollada. Se encontraron aspectos de la herramienta que generaron confusión en algunos usuarios, lo que propone un trabajo futuro en el que se puedan mejorar. Asímismo, la experimentación con usuarios sugirió mejoras y abrió líneas de desarrollo futuro para la herramienta.

Como resumen, el resultado de esta memoria es una herramienta útil, conforme a los objetivos planteados, desarrollada utilizando un enfoque ingenieril, validada con usuarios y con proyecciones de crecimiento.

Trabajo Futuro

Respecto a la memoria no quedan tareas pendientes ni objetivos por cumplir. Sin embargo, aún hay mucho espacio para mejoras. En lo que sigue, se detallan algunas tareas o áreas para trabajo futuro.

Mejorar mecanismo de metadatos

Un punto de confusión en el uso del lenguaje, reportado a través de la experimentación con usuarios, fue la inclusión de dos mecanismos para la definición de metadatos.

Como trabajo futuro se evaluará una mejor forma de asociar metadatos a las estructuras del lenguaje.

Una posible mejora a esta situación sería utilizar la identación presente en el archivo fuente, como se discutió en el capítulo 6. Para esto se podría considerar a los metadatos como otro valor más lenguaje y asociarlos a las estructuras que estén "cerca" en el documento según alguna heurística. Esto podría implementarse como un "filtro" a nivel de la arquitectura del sistema. Esto permitiría disponer de un único mecanismo para la definición de metadatos lo que, se espera, disminuiría la confusión por parte de los usuarios.

Otras mejoras

A continuación algunas funcionalidades que se pretende incorporar al lenguaje en un futuro:

1. Estilización de estructuras
2. Reporte de errores sintácticos
3. Poder descargar o exportar las visualizaciones

Sin duda hay muchas mejoras posibles, pero se considera que estas podrían mejorar considerablemente la experiencia de uso.

Aplicación de escritorio

Electron[7] es una biblioteca que permite la construcción de aplicaciones multi-plataforma utilizando tecnologías web (HTML, JavaScript y CSS). Dado que la herramienta fue desarrollada enteramente con tecnologías web, se podría utilizar Electron para generar ejecutables de escritorio que el usuario podría instalar en su computador sin tener que acceder a una página

web en particular o disponer de una conexión a Internet. Se podría utilizar del mismo modo en que se utiliza cualquier aplicación de escritorio.

Continuidad del proyecto

Como se ha mencionado en diversas ocasiones, una motivación importante de esta memoria era generar una herramienta útil para las personas y que pudiera vivir más allá de la entrega de esta memoria, aportando valor a los usuarios en sus distintas labores.

Para sostener la continuidad del proyecto, se considera que una buena opción es liberar el código fuente de cada componente del lenguaje y continuar su desarrollo como un proyecto *open-source*.

Esto facilitará la obtención de la herramienta por parte de los usuarios, al mismo tiempo que permitirá incorporar retroalimentación y aportes de la comunidad al desarrollo del lenguaje.

Bibliografía

[1] Ace - The High Performance Code Editor for the Web. https://ace.c9.io/. Accessed: 2017-12-8.

[2] Algorithm Visualizer. http://algo-visualizer.jasonpark.me. Accessed: 2017-11-28.

[3] ANTLR - Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4/. Accessed: 2017-12-7.

[4] Binary Search Tree Visualization. https://www.cs.usfca.edu/~galles/visualization/BST.html. Accessed: 2017-11-28.

[5] Bootstrap - The most popular HTML, CSS, and JS library in the world. http://getbootstrap.com/. Accessed: 2017-12-8.

[6] Dillinger - Online Markdown Editor. https://dillinger.io/. Accessed: 2017-11-28.

[7] Electron - Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org/. Accessed: 2017-12-8.

[8] Elixir - a dynamic, functional language designed for building scalable and maintainable applications. https://elixir-lang.org/. Accessed: 2017-12-7.

[9] GraphQL - A query language for your API. http://graphql.org/. Accessed: 2017-12-6.

[10] Heroku - Cloud Application Platform. https://www.heroku.com/. Accessed: 2017-12-8.

[11] JSAV: The JavaScript Algorithm Visualization Library. http://jsav.io. Accessed: 2017-11-28.

[12] openCypher - Full and open specification of the graph database query language Cypher. http://www.opencypher.org/. Accessed: 2017-12-7.

[13] Python3 - Lexical Analysis. https://docs.python.org/3/reference/lexical_analysis.html. Accessed: 2017-12-7.

[14] RDoc - Ruby Documentation System. https://ruby.github.io/rdoc/. Accessed: 2017-11-28.

[15] Viz.js - A hack to put Graphviz on the web. https://github.com/mdaines/viz.js/. Accessed: 2017-12-8.

[16] Webpack - webpack is a module bundler. https://webpack.js.org/. Accessed: 2017-12-8.

[17] John W Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959.

[18] J. L. Bentley and B. W. Kernighan. A System for Algorithm Animation. Technical Report 132, Bell Labs, Murray Hill, New Jersey 07974, January 1987.

[19] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[20] Georg Brandl. Sphinx: Python Documentation Generator. *URL: http://sphinx. pocoo. org/index. html (13.8. 2012)*, 2009.

[21] RA Brooker, IR MacCallum, Derrick Morris, and JS Rohl. The compiler compiler. *Annual review in automatic programming*, 3:229–275, 1963.

[22] Marc H Brown and Robert Sedgewick. Techniques for algorithm animation. *Ieee Software*, 2(1):28, 1985.

[23] Noam Chomsky. *Syntactic structures*. Walter de Gruyter, 2002.

[24] Fred D Davis. User acceptance of information technology: system characteristics, user perceptions and behavioral impacts. *International journal of man-machine studies*, 38(3):475–487, 1993.

[25] Charles Donnelly and Richard Stallman. Bison. the yacc-compatible parser generator. 2000.

[26] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[27] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[28] Adam C Engst. comp.sys.mac.announce / tidbits file server available usenet. https://groups.google.com/d/msg/comp.sys.mac.announce/huItllyV_p4/-X8pBRvRSPcJ. Accessed: 2017-11-28.

[29] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse, 2000.

[30] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[31] John Gruber. Markdown: Syntax. *URL http://daringfireball. net/projects/markdown/syntax. Retrieved on June*, 24, 2012.

[32] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[33] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[34] David Herman, Luke Wagner, and Alon Zakai. asm. js: Working draft 17 march 2013. *Available online at asmjs.org/spec/latest*, 2013. Accessed: 2017-11-28.

[35] Ariya Hidayat. Phantomjs: Headless webkit with javascript api. *WSEAS Transactions on Communications*, 2013.

[36] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[37] Graham Hutton and Erik Meijer. Monadic parser combinators. 1996.

[38] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[39] Eleftherios Koutsofios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.

[40] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM, 1999.

[41] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[42] Daan Leijen. Parsec, a fast combinator parser, 2001.

[43] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator. `http://dinosaur.compilertools.net/lex/index.html`, 1975.

[44] John Lyons. *Introduction to theoretical linguistics*. Cambridge university press, 1968.

[45] William O'Grady, Michael Dobrovolsky, and Francis Katamba. *Contemporary linguistics*. St. Martin's, 1997.

[46] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*, volume 49, pages 579–598. ACM, 2014.

[47] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[48] Rob Pike. The UNIX system: The blit: A multiplexed graphics terminal. *Bell Labs Technical Journal*, 63(8):1607–1631, 1984.

[49] S Porritt. The jsplumb javascript library.

[50] Thomas A Standish. Extensibility in programming language design. *ACM Sigplan Notices*, 10(7):18–21, 1975.

[51] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[52] Reinhard Wilhelm, Dieter Maurer, and Stephen S Wilson. *Compiler design*, volume 610. Springer, 1995.

[53] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

Anexos

Gramática del lenguaje

El Código 10.3, presentado a continuación, describe la gramática del lenguaje en el formato requerido por ANTLR (ver Subsección 6.0.1). Este documento se puede utilizar con las herramientas provistas por ANTLR para generar intérpretes en diversos lenguajes de programación, siendo JavaScript el utilizado en esta memoria.

```
//
// PARSER
//

grammar Ve;

//
// V - DOCUMENT
//

/*
 * Document
 *
 */
document
    : topStruct+
    ;

topStruct
    : struct
    | treeStruct
    | graphStruct
    ;

/* Metadata attachments */

metas
    : meta +
    ;

meta
    : '@' NAME ( '(' scalar (',' scalar)* ')' )?
    ;

```

```
overs
    : over +
    ;

over
    : '!' NAME ( '(' scalar (',' scalar)* ')' )?
    ;

// TREE

treeSet
    : '{' treeStruct ( ','? treeStruct )* '}'
    | '{' '}'
    ;

treeStruct
    : treeNode
    // structs has precedence over tree, so an empty tree "-" will always match the
    // null value first
    | nullNode
    ;

nullNode
    : nilLiteral
    ;

treeNode
    : metas? treeName overs? treeSet?
    ;

treeName
    : NAME
    | NUMBER
    | STRING
    ;

// GRAPH STRUCT

graphStruct
    : graphDef+
    ;

graphDef
    : metas? nodeDef overs?
    | metas? edgeDef overs?
    ;

edgeDef
    : nodeRef ( edgeChain )*
    ;

nodeDef
    : '(' NAME nodeLabel? ')'
    ;

```

```
nodeRef
    : '(' NAME ')'
    ;

edgeChain
    : edgeConnector nodeRef
    ;

edgeConnector
    : ( leftArrowHead dash edgeLabel? dash rightArrowHead )
    | ( leftArrowHead dash edgeLabel? dash )
    | ( dash edgeLabel? dash rightArrowHead )
    | ( dash edgeLabel? dash )
    ;

edgeLabel
    : '[' labelName ']'
    ;

nodeLabel
    : ':' labelName
    ;

labelName
    : STRING | NAME
    ;

leftArrowHead
    : '<'
    ;

rightArrowHead
    : '>'
    ;

dash
    // U+002D HYPHEN-MINUS
    : '-'
    // NOTE: the following symbols are not included in the file due to encoding
    // issues of the document.
    // U+2010 HYPHEN
    // U+2011 NON-BREAKING HYPHEN
    // U+2012 FIGURE DASH
    // U+2013 EN DASH
    // U+2014 EM DASH
    // U+2015 HORIZONTAL BAR
    // U+2212 MINUS SIGN
    // U+FE58 SMALL EM DASH
    // U+FE63 SMALL HYPHEN-MINUS
    // U+FF0D FULLWIDTH HYPHEN-MINUS
    ;

// JSON STRUCT

record
    : RECORD_OPEN pair (',' pair)* RECORD_CLOSE
```

```
146     | RECORD_OPEN RECORD_CLOSE
147     ;
148
149 map
150     : '{' pair (',' pair)* '}'
151     | '{' '}'
152     ;
153
154 pair
155     : NAME ':' struct
156     | STRING ':' struct
157     ;
158
159 array
160     : '[' struct (','? struct)* ']'
161     | '[' ']'
162     ;
163
164 struct
165     : metas? value overs?
166     ;
167
168 value
169     : stringLiteral
170     | numberLiteral
171     | booleanLiteral
172     | nilLiteral
173     | map
174     | array
175     | record
176     ;
177
178 scalar
179     : stringLiteral
180     | numberLiteral
181     | booleanLiteral
182     | nilLiteral
183     ;
184
185 stringLiteral
186     : STRING
187     | NAME
188     | WORD
189     ;
190
191 numberLiteral
192     : NUMBER
193     ;
194
195 booleanLiteral
196     : TRUE
197     | FALSE
198     ;
199
200 nilLiteral
201     : NULL
```

```
202     | NIL
203     | dash
204     ;
205
206 //
207 // LEXER
208 //
209
210 TRUE : ( 'T' | 't' ) ( 'R' | 'r' ) ( 'U' | 'u' ) ( 'E' | 'e' )  ;
211
212 FALSE : ( 'F' | 'f' ) ( 'A' | 'a' ) ( 'L' | 'l' ) ( 'S' | 's' ) ( 'E' | 'e' )
        ;
213
214 NULL : ( 'N' | 'n' ) ( 'U' | 'u' ) ( 'L' | 'l' ) ( 'L' | 'l' ) ;
215
216 NIL : ( 'N' | 'n' ) ( 'I' | 'i' ) ( 'L' | 'l' ) ;
217
218 STRING
219     : '"' (ESC | ~ ["\\])* '"'
220     ;
221
222 fragment ESC
223     : '\\' (["\\/bfnrt] | UNICODE)
224     ;
225
226
227 fragment UNICODE
228     : 'u' HEX HEX HEX HEX
229     ;
230
231
232 fragment HEX
233     : [0-9a-fA-F]
234     ;
235
236
237 NUMBER
238     : '-'? INT ('.' [0-9] +)? EXP?
239     ;
240
241
242 fragment INT
243     : '0' | [1-9] [0-9]*
244     ;
245
246 // no leading zeros
247
248 fragment EXP
249     : [Ee] [+\-]? INT
250     ;
251
252 COMMENT
253     :    ( '//' ~[\r\n]* '\r'? '\n'
254          | '/*' .*? '*/'
255          ) -> channel(HIDDEN)
256     ;
```

```
257
258 NAME
259     : ID_START ID_CONTINUE*
260     ;
261
262 RECORD_OPEN : '% [a-zA-Z]+ '{' ;
263 RECORD_CLOSE : '}' ;
264
265 WORD
266     : NAME
267     // Throw in some Unicode Symbols as valid words too
268     | [\u2600-\u26FF]
269     ;
270
271 WS
272     : [ \t\n\r] + -> channel(HIDDEN)
273     ;
274
275 /// id_start     ::= <all characters in general categories Lu, Ll, Lt, Lm, Lo
        , Nl, the underscore, and characters with the Other_ID_Start property>
276 fragment ID_START
277     : '_'
278     | [A-Z]
279     | [a-z]
280     | // rest omitted for brevity
281     ;
282
283 /// id_continue  ::= <all characters in id_start, plus characters in the
        categories Mn, Mc, Nd, Pc and others with the Other_ID_Continue property>
284 fragment ID_CONTINUE
285     : ID_START
286     | [0-9]
287     // rest omitted for brevity
288     ;
```

Código 10.3: Gramática del lenguaje

Resultados de la encuesta

A continuación se detallan las respuestas individuales de la encuesta (ver Subsección 9.2.1). Pese a que se realizó un análisis de estas respuestas (ver Subsección 9.2.2), se adjuntan acá como testimonio de veracidad.

Respuesta #1

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Lo utilizaría para explicar ciertos pasos de un método o estructura de datos custom
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	Quizás transiciones animadas, pero eso ya es muy complicado de maquetear en un texto leíble
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Regular
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Lo usaría para mi trabajo

Respuesta #2

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay suficiente relación

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código)

Respuesta #3

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	El hecho de que algunas keyword (por ejemplo vertical) se aplican con @ o con ! en contextos diferentes puede resultar algo confuso.
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	
9) ¿Qué está de más en el lenguaje?	Creo que utilizaría todas las estructuras para documentar de alguna manera mi código salvo los grafos.
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código)

Respuesta #4

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Actualmente para mandar diagramas tendría que o sacar una foto a un dibujo o usar ppt para hacerlo, esto parece una alternativa fácil de usar.
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	No se explicó que @ y ! Son equivalentes excepto por la asociación
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	Me pareció muy genial el poder mostrar las tablas. Lenguajes parecidos requieren ascii muy engorroso de alinear
8) ¿Qué más podría tener el lenguaje?	Me gustaría poder ponerle una etiqueta a las ramas de un árbol para dibujar árboles de decisión
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Me interesa y me gustaría colaborar, Lo usaría para mi trabajo

Respuesta #5

Pregunta	**Respuesta**
1) Su principal actividad	Consultora
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Para ver el comportamiento de redes neuronales
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay suficiente relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	
9) ¿Qué está de más en el lenguaje?	

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Reemplazaría un lenguaje que ya usa (para visualizar estructuras), Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Lo usaría para mi trabajo

Respuesta #6

Pregunta	**Respuesta**
1) Su principal actividad	Docente
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Como material auxiliar en para clases de EDA
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	Visualización de errores sintácticos
9) ¿Qué está de más en el lenguaje?	Nada
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Me interesa y me gustaría colaborar

Respuesta #7

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Para documentar código, métodos, y para visualizar datos en herramientas web

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	Hay mucha cercanía entre lo que uno escribe, y lo que se ve. Se siente como si el lenguaje entendiera lo que quiero decir.
8) ¿Qué más podría tener el lenguaje?	Customización de estilo
9) ¿Qué está de más en el lenguaje?	1. El doble hyphen para las flechas de los grafos creo que está de más. Podría ser uno. 2. Creo que las decoraciones ensucian un poco la lectura humana del input. Tal vez si hubiera aliases más breves para los metadatos (!v en vez de !vertical, !i en vez de indexedBy). 3. A las linked lists tampoco les veo mucho uso, pero de más que alguien más sí.b 4. Ahora, esto no es que "esté de más en el lenguaje", pero creo que está de más Cualquier avance que permita limpiar el input me parece interesante. Por ejemplo, declarar los conjuntos de índices de antemano en vez de en el mismo lugar de invocación (si quiero usar índices custom en varias partes, por ejemplo). También sería choro poder declarar que los índices loopean de un conjunto finito: ("A", "B") ->(A B A B A B...).
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Me interesa y me gustaría colaborar, Lo usaría para mi trabajo

Respuesta #8

Pregunta	**Respuesta**
1) Su principal actividad	Estudiante

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Para docencia
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	Hay suficiente relación
8) ¿Qué más podría tener el lenguaje?	
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Reemplazaría un lenguaje que ya usa (para visualizar estructuras), Lo usaría para mi trabajo

Respuesta #9

Pregunta	**Respuesta**
1) Su principal actividad	Docente
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Como profesor, enseño cursos básicos y siempre me cuesta hacer figuras ilustrativas (árboles, grafos, arrays, etc.)
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay suficiente relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Regular
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
12) El lenguaje... (puede responder todas las que quiera)	Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Lo usaría para mi trabajo

Respuesta #10

Pregunta	**Respuesta**
1) Su principal actividad	Docente
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Se ve simple e intuitivo para armar estructuras de datos. Podría servir a apoyar la enseñanza de esta temática en cursos iniciales.
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	a veces los comandos como emphasize o indexedby van dentro de los corchetes y otras veces no... Eso es un poco confuso. Por otra parte las agrupaciones a veces se hacer con [, otras veces con {, y otras veces no hay simbolos que agrupen. Eso también es un poco confuso, pero el resto se ve muy bien.
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	Más allá de las potenciles mejoras, creo que el lenguaje ya está en condiciones de ser utilizado.
8) ¿Qué más podría tener el lenguaje?	Podrían implementarse operaciones sobre las estructuras de datos, por ejemplo ordenamientos y búsquedas... eso serviría para enseñar algoritmos básicos sobre esas estructuras.
9) ¿Qué está de más en el lenguaje?	Nada que yo haya notado.
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	No enseño cursos de estructuras de datos, pero si lo hiciera, creo que esta herramienta me ayudaría bastante a comunicar los conceptos detras de estas estructuras.

Respuesta #11

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	Ejemplo basico: personalización de colores
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Me interesa y me gustaría colaborar, Lo usaría para mi trabajo

Respuesta #12

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Para algunos proyectos que utilizan estructuras de datos esta herramienta permitiría visualizar dichas estructuras para procesos como debugging
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	Sería interesante poder visualizar operaciones sobre las estructuras (sumas de array por ejemplo)
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Rápida

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código)

Respuesta #13

Pregunta	**Respuesta**
1) Su principal actividad	Desarrollador o Desarrolladora
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Para documentación sería útil, dado que se podría dejar ese lenguaje en los comentarios y con un script generar la visualización.
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	En Arrays, en Nested: cambiar "metadaba"por "metadata"
8) ¿Qué más podría tener el lenguaje?	Más que el lenguaje en sí, falta que el intérprete te diga que encontró errores al procesar el lenguaje. Por ejemplo, si en los maps uso usa comillas simples en vez de comillas dobles, si en los grafos uno pone —>(con un guión de más), sería bueno que la página informe que hubo error al procesar el código.
9) ¿Qué está de más en el lenguaje?	Se pueden omitir algunas comillas dobles en @indexedBy(.ºne"), @indexedBy(one) funciona igual.
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Lo usaría para mi trabajo

Respuesta #14

Pregunta	**Respuesta**
1) Su principal actividad	Estudiante
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Podría usarse en documentos para explicar algoritmos por ejemplo de una forma más simple que las soluciones existentes.
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	Debería unificarse la forma en que se construyen árboles y grafos dado que tendia a ser confuso que uno fuera con llaves y el otro con flechas.
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	Podría cambiarse el color a los arreglos y nodos, por ejemplo pensando en crear estructuras como arboles rojo negro. Asímismo los nodos podrían contener pequeñas listas para dibujar árboles 2-3 o B-Trees
8) ¿Qué más podría tener el lenguaje?	Texto en los arcos de los árboles, Punteros entre estructuras, explicar mejor los marcadores sobre arreglos, marcar entre elementos (por ejemplo para explicar un swap de un algoritmo de ordenación)
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Regular
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Lo usaría para mi trabajo

Respuesta #15

Pregunta	**Respuesta**
1) Su principal actividad	Estudiante
2) ¿Encuentra usted que la herramienta	No resuelve ningún problema
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	
4) El lenguaje:	Es suficientemente claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay mucha relación

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Lenta
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza

Respuesta #16

Pregunta	**Respuesta**
1) Su principal actividad	Consultor Analytics
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Útil para generar visualizaciones rápidas y agregarlas a los informes (grafos y árboles mayoritariamente)
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	Me pareció muy claro, siempre y cuando el usuario sea alguien que conoce de código
6) Con respecto a las visualizaciones:	Hay mucha relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	Representar instrucciones y algoritmos, por ejemplo, "¿cómo se ve un for loop?"
9) ¿Qué está de más en el lenguaje?	Se me hizo un poco confuso que ¡"sirva para agregar meta datos, en ese sentido, la sintaxis para agregar meta datos me parece que hace más complejo el lenguaje.
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código)

Respuesta #17

Pregunta	**Respuesta**
1) Su principal actividad	Estudiante
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Crear diagramas para explicar estructuras de datos
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay suficiente relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	Encuentro incómodo que para eliminar algo de la visualización hay que llegar a un estado válido del código.
8) ¿Qué más podría tener el lenguaje?	Soportar más de un elemento en un nodo, soportar estructuras dentro de nodos (aunque lo primero es más importante, ya que sirve para árboles 2-3). Mostrar en la visualización el sector que se está modificando.
9) ¿Qué está de más en el lenguaje?	Nada en mi opinión.
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Reemplazaría un lenguaje que ya usa (para visualizar estructuras)

Respuesta #18

Pregunta	**Respuesta**
1) Su principal actividad	Estudiante
2) ¿Encuentra usted que la herramienta	Creo que me sería bien útil
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Para el esclarecimiento de los códigos que trabajen con estructuras y la verificación del funcionamiento de estos códigos
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	
6) Con respecto a las visualizaciones:	Hay suficiente relación
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
8) ¿Qué más podría tener el lenguaje?	En la parte de grafos se podrían incluir ejes no dirigidos y la posibilidad de especificar la cantidad de ejes entre dos nodos a modo de ahorrar y simplificar código al momento de elaborar multigrafos no dirigidos.
9) ¿Qué está de más en el lenguaje?	
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Reemplazaría un lenguaje que ya usa (para visualizar estructuras), Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Lo usaría para mi trabajo

Respuesta #19

Pregunta	**Respuesta**
1) Su principal actividad	Docente
2) ¿Encuentra usted que la herramienta	Quizá en algún momento me podría servir
3) Feedback (opcional). Si cree que la herramienta le sería útil, por favor, diga para qué.	Encuentro que la herramienta es útil para preparar materiales de los cursos básicos de la carrera de pregrado, principalmente en los cursos de programación, algoritmos y estructuras de datos. También lo encuentro útil para facilitar la creación de diagramas en artículos de investigación. Sin embargo, en este último caso, sería muy útil que la salida sea Latex en lugar de svg.
4) El lenguaje:	Es bastante claro
5) Feedback (opcional). Si el lenguaje le pareció confuso, por favor, escriba de qué manera.	Al lenguaje lo encuentro claro, aunque tiene muchas construcciones sintácticas que sin la ayuda de los ejemplos, llevaría tiempo adoptar. Encuentro además que utiliza "metáforas"diferentes que podrían unificarse. Por ejemplo, utiliza @horizontal como prefijo e !indexedBy como sufijo. Indica que la metadata puede incluirse de las dos formas, pero me pregunto si esa flexibilidad favorece al usuario del lenguaje o si hace más compleja la comprensión.
6) Con respecto a las visualizaciones:	Hay mucha relación

Continúa en la siguiente página

Continúa de la página anterior

Pregunta	**Respuesta**
7) Feedback (opcional). Si tiene un comentario adicional sobre la relación entre lenguaje y visualización, puede escribirlo acá.	
8) ¿Qué más podría tener el lenguaje?	Permitir la configuración de estilos, como ser tipos y tamaños de letra, color, y forma de las figuras.
9) ¿Qué está de más en el lenguaje?	Múltiples metáforas para el mismo propósito.
10) ¿Cómo es la carga inicial del sistema?	Rápida
11) ¿Cómo es la velocidad del sistema una vez cargado?	Rápida
12) El lenguaje... (puede responder todas las que quiera)	Le recuerda a otros lenguajes que utiliza, Sería bueno incorporarlo dentro otro lenguaje (ej. documentación de código), Me interesa y me gustaría colaborar, Lo usaría para mi trabajo, Podría permitir diferentes tipos de salida, como ser html, latex, png, etc.