UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aplicación web para ayuda en el aprendizaje de la gestión de memoria dinámica en programación con el lenguaje C

Iván Serrano Sagredo
Tutor: Marina de la Cruz Echeandía
Ponente: Alfonso Ortega de la Puente

Julio 2018

Aplicación web para ayuda en el aprendizaje de la gestión de memoria dinámica en programación con el lenguaje C

AUTOR: Iván Serrano Sagredo
TUTOR: Marina de la Cruz Echeandía

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2018

Resumen

Este trabajo junto con el de Óscar Martín Sanz, componen un producto único, el cual se ha dividido por una parte en web y generación de código, y en una segunda parte de interpretación de código y representación gráfica. Este documento contiene la primera parte del proyecto.

El objetivo de este producto es crear una aplicación web que, mediante el lenguaje de bloques Google Blockly, permita a los usuarios crear un programa C de manera visual y combinarlo con una herramienta de análisis de código que permita extraer la información de la gestión de la memoria dinámica para dibujarla en un canvas gráfico, así como las modificaciones que las diferentes operaciones realicen sobre ella, todo ello sin la necesidad de usar un servidor externo. El objetivo es que el sistema sea útil para las prácticas de la asignatura programación 2 del grado de ingeniería informática y de telecomunicación. Para ello, se han elaborado distintos módulos, tratándose en esta memoria aquellos comunes, así como los específicos de este trabajo.

El desarrollo de Blockly consiste en la definición de los bloques, los cuales se usarán en su propio canvas gráfico y, para cada uno de ellos, se generará su código equivalente en C, estando este correctamente formateado para ser tratado por el intérprete del trabajo de Óscar. Sin embargo, para poder adaptar Blockly mejor a un lenguaje con definición de tipos, punteros y arrays, se han realizado una serie de mejoras a su código fuente, así como funciones que implementan funcionamiento avanzado necesario para su funcionamiento sobre la base de Blockly.

Todos los módulos del producto han sido posteriormente integrados en una aplicación web, la cual se encarga de la comunicación entre cada uno de ellos.

Abstract

This Thesis, along with that of Óscar Martín Sanz, make up a unique product, which has been divided on one hand into web and code generation, and into a second part of code interpretation and graphic representation. This document contains the first part of the project.

The goal of this Bachelor Thesis is to create a web app that, through Google Blockly block language, allows users to create a C program visually and combine it with a tool of code analysis that allows to represent graphically all the information in it. Every modification made in the memory must be represented in the canvas, all of it without the need of an external server. The objective is to make the system useful for the practices of the subject programming 2 of the degree of computer engineering and telecommunications. For this purpose, different modules have been developed, being treated in this memory those common, as well as the specific ones of this work.

The development of Blockly is based on the definition of blocks, which will be used in their own graphic canvas and, for each of them, their equivalent code will be generated in C, being this correctly formatted to be treated by the interpreter of the work of Óscar.

Nevertheless, in order to better adapt Blockly to a language with definition of types, pointers and arrays, a series of improvements have been made to its source code, as well as functions that implement advanced operation necessary for its operation on the basis of Blockly.

All product modules have been subsequently integrated into a web application, which is responsible for communication between each one of them.

Palabras clave

Blockly, bloques, intérprete, JavaScript, web, d3.js, vue.js, Boostrap Vue, interpretar, canvas, memoria dinámica, C, educacional, análisis de código, gestión de memoria, funciones, condicionales, bucles, punteros, arrays, TADs, estructuras, workspace, mutadores.

Keywords

Blockly, block, interpreter, JavaScript, web, d3.js, Bootstrap, Vue, parser, canvas, Dynamic memory, C, educational, code analysis, memory management, functions, conditional, loop, pointers, arrays, ADTs, structures, workspace, mutator.

Agradecimientos

Quiero agradecer su ayuda y apoyo a ese grupo de personas locas que nos queremos hacer llamar informáticos y nuestros grandes momentos entre clases. Si Miguel, te estoy mirando a ti, ¡vuelve ya, bribón!

Gracias a los profesores, ya que, gracias a ellos estamos aquí hoy con los conocimientos que tenemos. Agradecer sus horas de dedicación y su trato personal.

Quiero agradecérselo también a mi familia, pues han sido uno de los soportes más grandes durante todos estos años.

Gracias a Óscar, mi compañero del Trabajo de Fin de Grado, por realizar ayudarme a realizar esta última etapa de la universidad.

Gracias a David por estar siempre ahí, su locura y ayudar en esas prácticas difíciles.

Gracias a Pablo por darme dos de los mejores años de mi vida y enseñarme a vivir la vida.

Y, sobre todo, gracias a ti Adri, por soportarme todos estos años, incluirme en tu vida y sacarme una sonrisa en los momentos difíciles. Te aprecio mucho y sé que nuestra amistad seguirá existiendo por mucho tiempo. Eres increíble.

INDICE DE CONTENIDOS

1 Prefacio 1
2 Introducción 3
2.1 Motivación 3
2.2 Objetivos 4
2.3 Organización de la memoria 5
3 Estado del arte 7
3.1 Introducción 7
3.2 Blockly - Generación de código 11
3.3 Web… 13
3.4 Conclusiones 14
4 Diseño 15
4.1 General 15
4.1.1 Requisitos funcionales y no funcionales 15
4.1.2 Casos de uso 17
4.1.3 Matriz de trazabilidad 18
4.1.1 Ciclo de vida del desarrollo del proyecto 20
4.2 Web… 21
4.2.1 Introducción 21
4.2.2 Blockly 22
4.3 Blockly 23
4.3.1 Core 23
4.3.2 Blocks 24
4.3.3 Generator 26
5 Desarrollo 27
5.1 Web… 27
5.1.1 Introducción 27
5.1.2 Blockly 28
5.2 Blockly 29
5.2.1 Introducción 29
5.2.2 Core 29
5.2.3 Blocks 31
5.2.4 Generator 33
5.2.4.1 Prioridades 35
5.2.4.2 Bloques dinámicos 36
5.2.4.3 Compatibilidad con el intérprete 36
6 Integración, pruebas y resultados 37
6.1 Integración 37
6.2 Pruebas y resultados 37
6.2.1 Web 37
6.2.1.1 Visualización 37
6.2.2 Blockly 38
6.2.3 Producto final 38
7 Conclusiones y trabajo futuro 39
7.1 Conclusiones 39
7.2 Trabajo futuro 40
Referencias 41
Glosario 43
Anexos I

A Manual de uso... I
B Manual del programador III
7.2.1 Cómo compilar Blockly... III
7.2.2 c_bucles.js V
7.2.3 c_functions.js VI
7.2.4 c_logica.js VIII
7.2.5 c_operaciones.js..VIII
7.2.6 c_punteros.js X
7.2.7 c_salida.js XIII
7.2.8 c_tads.js XIV

INDICE DE FIGURAS

Figura 3-1: Scratch, canvas gráfico ... 7
Figura 3-2: Scratch, bloques ... 8
Figura 3-3: Snap! ... 9
Figura 3-4: Proyectos en Blockly ... 10
Figura 3-5: Generación de JavaScript ... 11
Figura 3-6: Frameworks web ... 13
Figura 4-1: Caso de uso Web ... 17
Figura 4-2: Caso de uso Blockly ... 17
Figura 4-3: Caso de uso Código ... 18
Figura 4-4: Diagrama módulos ... 18
Figura 4-5: Ciclo de vida ... 20
Figura 4-6: Página principal ... 21
Figura 4-7: Diagrama de Blockly ... 23
Figura 4-8: Bloque de tipo output con un value input ... 24
Figura 4-9: Bloque con conexiones superior e inferior que acepta un statement input ... 24
Figura 4-10: Ejemplos de campos ... 25
Figura 4-11: Dropdown de tipos ... 25
Figura 4-12: Toolbar de Blockly ... 26
Figura 5-1: Creación de TAD y punteros y su visualización en las variables ... 30
Figura 5-2: Creación de variable ... 31
Figura 5-3: Comportamientos dinámicos mediante eventos ... 31
Figura 5-4: Mutator ... 32
Figura 5-5: Bloques con diferente número de inputs ... 36

INDICE DE TABLAS

Tablas 2.3-1: División de trabajo 5
Tabla 3.2-1: Editores de código fuente..... 12
Tabla 4.1-1: Matriz de trazabilidad 19
Tabla 5.2-1: Funciones principales del generador..... 33
Tabla 5.2-2: Funciones auxiliares del generador..... 34
Tablas B-1: Bucles V
Tablas B-2: Funciones..... VI
Tablas B-3: Lógica VIII
Tablas B-4: Operaciones VIII
Tablas B-5: Punteros X
Tablas B-6: Salida XIII
Tablas B-7: TADs..... XIV

1 Prefacio

Este TFG se ha desarrollado desde un inicio como un trabajo único. De hecho, los dos alumnos involucrados han formado un único equipo de trabajo. Junto con el tutor y ponente han formado un único equipo que ha abordado partes del trabajo de manera conjunta, aunque también se han repartido algunas tareas de manera independiente.

La manera de documentar el trabajo, desde nuestra perspectiva, más adecuada a la realidad del desarrollo y a la autoría de cada alumno, habría sido la elaboración de una memoria única de la que fueran autores ambos alumnos. Sin embargo, por requerimientos formales y legales de la normativa de los trabajos de fin de grado cada alumno debe presentar una memoria independiente.

Esto genera una situación un poco compleja respecto a la autoría de los textos que describen la parte común. La realidad es que han sido escritos por los dos alumnos como coautores y, para garantizar la coherencia de las memorias independientes deben aparecer en ambas; de otra forma resulten incompletas. Se ha intentado utilizar algún mecanismo de referencia de un documento al otro, pero aun así quedan párrafos que es necesario que estén en las dos para que cada una de las memorias sea autocontenida. En ese sentido la memoria del trabajo de fin de grado de Óscar Martín Sanz se citará como [27].

Tampoco pueden aparecer como autores de cada memoria los dos. La única manera en la que podemos explicitar que la coincidencia de ciertos textos en ambas memorias constituye la constatación de que ambos alumnos son los coautores de los dos es a través de este prefacio que esperemos sea suficiente para satisfacer tanto los requisitos formales de los trabajos de fin de grado como los de autoría de documentos públicos y ausencia de plagio en los mismos.

A continuación, enumeramos los párrafos que aparecen en ambas memorias y de los que son autores ambos alumnos puesto que tanto la redacción como la revisión la han realizado de manera conjunta.

- Este prefacio.
- La introducción del trabajo.
- Párrafos previos a las figuras de la sección “Introducción” del “Estado del arte”.
- En esa misma sección los textos finales desde la descripción de Snap!
- Primer párrafo de la sección “Conclusiones” del “Estado del arte”.
- En el capítulo dedicado al diseño, las secciones iniciales hasta, e incluyendo, el ciclo de vida del desarrollo del proyecto.
- En el capítulo dedicado al desarrollo, el primer párrafo que describe la estructura del sitio web completo.
- En el capítulo dedicado a la integración y las pruebas, la sección de integración.

- En el capítulo de conclusiones y trabajo futuro, los tres primeros párrafos y el último de la sección de conclusiones.

- En ese mismo capítulo algunas frases de algunos párrafos de la sección de trabajo futuro.

Con este prefacio ambos alumnos manifiestan para que conste a todos los efectos su conformidad tanto en la autoría de los párrafos comunes como su presencia en ambas memorias en los términos que se ha intentado justificar en este prefacio.

2 Introducción

2.1 Motivación

En los últimos años hemos visto cómo la tecnología está mucho más presente en nuestra vida. Todos tenemos un teléfono móvil conectado a internet, donde podemos saber el tiempo que hará mañana o cuánto tardará el autobús en llegar a nuestra parada. Esta tecnología también ha permitido que el teletrabajo se vuelva algo mucho más común, algo que hace unos años era impensable. Recientemente hemos visto un aumento de dicha tecnología en los jóvenes estudiantes, y por ello muchos colegios, institutos y universidades han empezado a subirse a la ola tecnológica e implementar nuevas soluciones educativas, lo que ha provocado una proliferación de herramientas web de carácter educativo, así como de librerías que permiten desarrollarlas de una manera mucho más sencilla, como es el caso de Blockly [1]. Una de las principales aplicaciones de estas nuevas herramientas ha sido la introducción a personas no formadas en informática en la disciplina de la programación.

Gracias a las nuevas herramientas existentes, este proyecto nace con el objetivo de incrementar la competencia en la correcta gestión de la memoria dinámica en personas que se están introduciendo en el mundo de la programación, apoyándose para ello en las herramientas gráficas que permiten ayudar a imaginar visualmente el proceso de gestión de memoria dinámica mediante un formalismo de alto nivel distinto del lenguaje de programación C. Para ello, el proyecto consistirá en el desarrollo de una aplicación web que facilite la adquisición de la imagen visual, proporcionando un canvas en el que diseñar visualmente el proceso mediante el lenguaje de bloques Google Blockly, y un canvas gráfico en el que se representará la memoria del sistema, así como las modificaciones que las operaciones realice sobre ella. Así mismo, existirá un área de texto en la que se mostrará la equivalencia en lenguaje C de las operaciones diseñadas de manera visual.

2.2 Objetivos

Para la elaboración de este proyecto completo se definen los siguientes objetivos:

- Estudiar el estado del arte.
- Estudio del funcionamiento de Blockly
- Estudio de las herramientas de depuración de memoria.
- Estudio de las herramientas de diseño gráfico
- Desarrollo de un módulo Blockly con generador de código C que permita gestionar los programas típicos de las prácticas de la asignatura programación 2 del grado de ingeniería informática y telecomunicación.
- Desarrollo de un módulo de representación gráfica de memoria.
- Desarrollo de un módulo que interprete el código C en JavaScript.
- Desarrollo de una aplicación web que integre todos los módulos.
- Creación de una demo sobre el funcionamiento final de la aplicación.

Por razones de tiempo y simplificación se han omitido en el proyecto ciertas funcionalidades de C:

- Parámetros de entrada al programa.
- Entrada o salida estándar.
- Entrada o salida por fichero.
- Inclusión de librerías externas.
- Definición de cabeceras en módulo aparte.
- Enumeradores.
- Definiciones.
- Punteros a función.
- Variables globales.

2.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1**: Motivación, objetivos y organización de la memoria.
- **Capítulo 2**: Estado del arte de herramientas educativas enfocadas a la generación del código y herramientas de depuración de memoria.
- **Capítulo 3**: Diseño de la aplicación y sus módulos principales.
- **Capítulo 4**: Desarrollo de los módulos principales y sus componentes.
- **Capítulo 5**: Integración de los diferentes módulos y pruebas realizadas
- **Capítulo 6**: Conclusiones y trabajo futuro.
- **Apéndice A**: Manual de instalación.
- **Apéndice B**: Manual del programador.
- **Apéndice C**: Funciones del intérprete.

Al ser un Trabajo de Fin de Grado que, junto con otro, desarrolla un sistema único, cada uno ha desarrollado una parte. Oscar Martín se ha encargado principalmente del diseño, desarrollo y pruebas del intérprete. Iván Serrano se ha encargado principalmente del diseño, desarrollo y pruebas de la Web y Blockly, así como del apéndice B. En las siguientes tablas se detalla más cada parte:

	Diseño y análisis general	**Diseño y análisis Web**	**Diseño y análisis Blockly**	**Diseño y análisis Intérprete**	**Diseño y análisis Representa ción gráfica**
Iván Serrano Sagredo	X	X	X		X
Óscar Martín Sanz	X		X	X	X

	Desarrollo Web	**Desarrollo Blockly**	**Desarrollo Intérprete**	**Desarrollo Representación gráfica**
Iván Serrano Sagredo	X	X		
Óscar Martín Sanz			X	X

Tablas 2.3-1: División de trabajo

En base a esta distribución cada memoria contendrá una parte del análisis común y la parte de desarrollo única de cada trabajo. Para entender el funcionamiento por completo es necesario leer ambos documentos.

3 Estado del arte

3.1 Introducción

Existen diversas herramientas de lenguajes de bloques, como pueden ser *Blockly*, *Snap!* o Scratch, cada una de ellas con sus características. Tuvimos que considerar todas ellas para seleccionar la más adecuada a este proyecto.

Scratch, por ejemplo, es una herramienta enfocada a introducir a los usuarios (habitualmente estudiantes de ciclos anteriores a la universidad) a la programación y diseño de algoritmos. Por ello se ofrece un entorno de desarrollo enfocado en contar historias de forma interactiva o desarrollar videojuegos.

La acción se desarrolla en el canvas gráfico, donde el programador determina el comportamiento de todos los elementos dinámicos de la historia o el juego mediante algoritmos expresados con bloques. Todos los personajes se ejecutan en paralelo interactuando unos con otros habitualmente mediante eventos.

El principal inconveniente es que es un paquete cerrado que dificulta el desarrollo de entornos más abiertos seleccionando sólo algunos elementos del paquete, como es nuestro caso. La figura 2-1 muestra un típico estado del canvas gráfico de Scratch en el que se desarrolla un juego o historia interactiva en el cual los personajes son animales.

Figura 3-1: Scratch, canvas gráfico

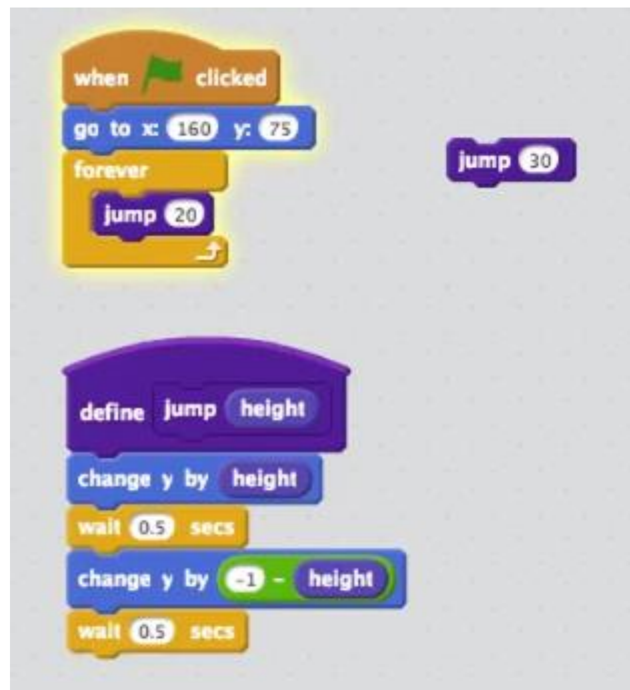
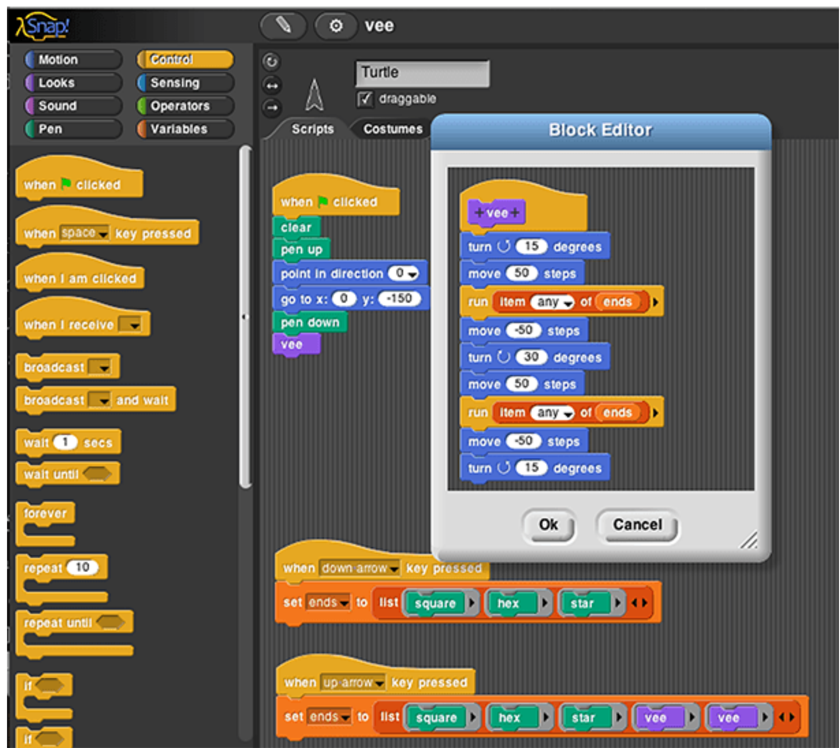


Figura 3-2: Scratch, bloques

La figura 2-2 muestra el código de bloques Scratch para uno de ellos, en concreto es el código para la función *jump* y el comportamiento que consiste en saltar 20 unidades eternamente tras acudir a la posición 160:75.

Snap!, por otro lado, es una herramienta orientada a un público más amplio. De hecho, realiza una tarea similar a Scratch (introducir al mundo de la programación) pero a estudiantes de mayor edad, ya que está siendo utilizado especialmente en la universidad. El sistema incluye un canvas gráfico similar al de Scratch. En apariencia parecen entornos con la misma funcionalidad, pero el código de Snap! es abierto y es JavaScript. El problema es que no está muy documentado y, aunque existen foros que comunican directamente con los autores, realmente es un espacio común para facilitar la continuidad de desarrolladores. Existe también un manual [23], pero este excluye los aspectos técnicos útiles cuando quieres incorporar sus elementos a un nuevo proyecto. No parece que los autores tengan planes de facilitar esa documentación. De haber sido seleccionada para nuestro proyecto habríamos tenido que partir del código directamente.



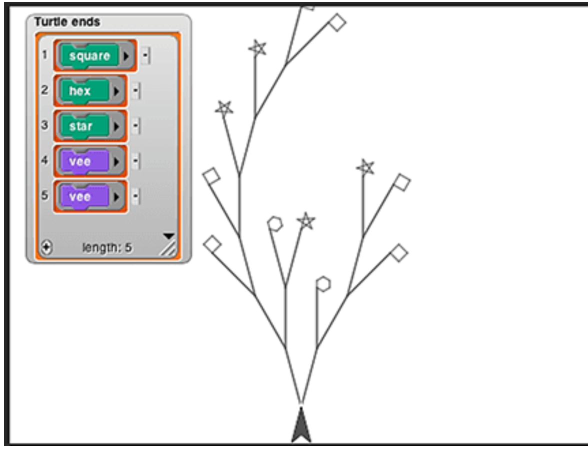


Figura 3-3: Snap!

En la figura 2-3 observamos un ejemplo de un programa en el cual se declara mediante bloques una serie de eventos, como el click del ratón o la pulsación de las teclas de flecha, los cuales se tienen asociados cambios en el canvas gráfico.

Por otro lado, Blockly [1] se ha convertido en un referente de hecho para la inclusión en cualquier sitio web que quiera tener la capacidad de expresar un algoritmo mediante un lenguaje de bloques. Por ello se está utilizando mucho para la creación de herramientas educativas gráficas gracias a la facilidad e intuitividad que ofrece al usuario final. Existen multitud de proyectos que lo usan como base, como son **MIT App Inventor** [2], herramienta del *Massachusetts Institute of Technology* la cual abstrae de tal manera la creación de aplicaciones para dispositivos móviles que hasta niños son capaces de desarrollarlas, o **MakeCode**, desarrollada por Microsoft y sirve como introducción en la informática, así como cientos de proyectos más. En cuanto a proyectos enfocados a lenguajes de programación similares al descrito en esta memoria, existe un ejemplo desarrollado por Blockly [3] en el cual tratan la generación de código en lenguajes no tipados.

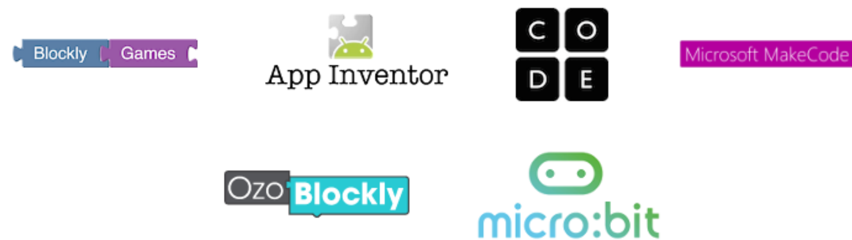


Figura 3-4: Proyectos en Blockly

En nuestro caso concreto, queremos introducir una herramienta educacional específica para el lenguaje C, más en concreto su gestión de memoria, donde tenemos a nuestra disposición una serie de herramientas que nos permiten ver desde diferentes puntos de vista el manejo de la memoria de un programa C. Estas herramientas otorgan una gran cantidad de información al usuario, tanto para controlar posibles errores durante una ejecución como para obtener información que permita mejorar el rendimiento de un programa.

3.2 Blockly - Generación de código

Podemos encontrar varios casos de generadores de código en Blockly. Un ejemplo bastante cercano a nuestro caso de uso es el proyecto llamado ***Cake*** [4] el cual implementaba un generador de C. Sin embargo, este proyecto está basado en una versión bastante antigua de Blockly, cuando aún en su estructura interna no existían las variables tipadas.

Tras un proceso de actualización del Core de Blockly por parte de sus desarrolladores, implementaron un nuevo modelo de variable, el cual incluía los tipos. Para demostrar el uso de este nuevo modelo, implementaron una serie de generadores para diversos lenguajes, todos ellos no tipados, y los mostraron en una demo [3]. Como podemos ver en la imagen 2-5, estos lenguajes fueron: Dart, JavaScript, Lua, PHP y Python

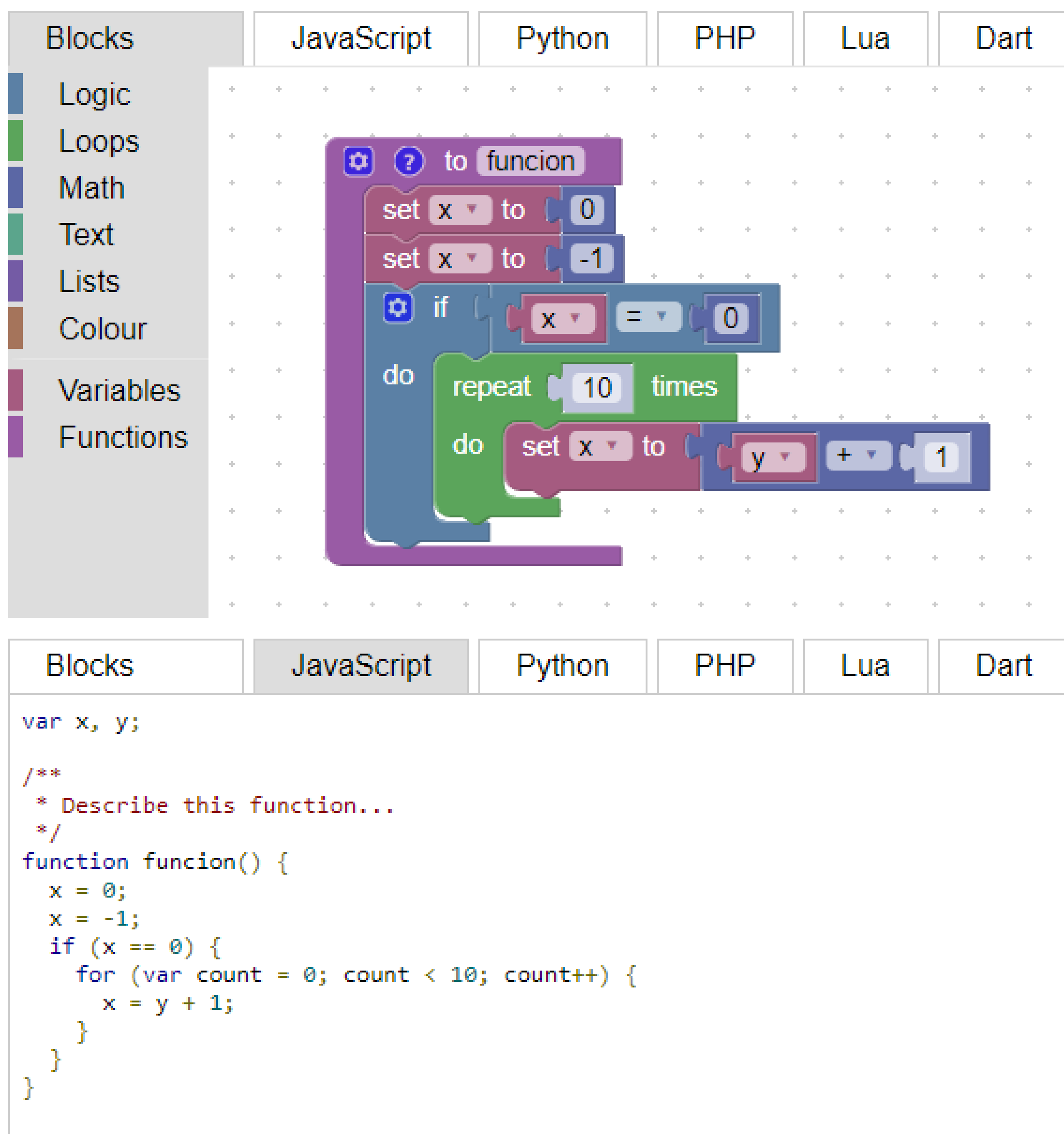


Figura 3-5: Generación de JavaScript

Además, necesitaremos una librería que actúe como editor de texto incrustado en el navegador y sea compatible con C. Como podemos ver en la siguiente lista comparativa de editores de código, CodeMirror es el único editor de texto que se basa exclusivamente en texto plano, el cual se encarga de formatear adecuadamente, hecho que facilita en gran manera su implementación en la página web, ya que no buscamos funcionalidades avanzadas propias de un editor de texto. Además, posee nativamente soporte para sintaxis C y está siendo desarrollado activamente.

Editor	Latest version	Style, clone of	Cost	Open source	Browser support
MDK-Editor	2.10, 2008	Microsoft Visual Studio	Depends on use	Code is readable	tested to work on: IE 6, 7 - Firefox 2, 3 - Chrome
Monaco Editor	0.8.3, 2017-03-03	Visual Studio Code	Free	Yes	IE8+, Firefox 4+, Chrome
Markitup	1.1.14, 2013-02-04	Markup editor, no syntax highlight	Free	Yes	IE 6 & 7, Firefox 2 & 3, Safari 3.1, Opera 9+[3]
Ymacs	0.5, 2012-03-28	Emacs	Free	Yes	Firefox, Chrome, Safari
Orion	8.0, 2015-03-04	Eclipse SWT StyledText, regular textarea	Free	Yes	Firefox 37+, Chrome 40+, Safari7+, Internet Explorer 11+[2]
LDT	2012-02-19	regular textarea	Free	Yes	Firefox 3.6+, IE8, Chromium 16, Midori 4.1, Opera 11, Epiphany
Ace	1.3.3, 2018-03-26	Sublime Text / Microsoft Visual Studio	Free	Yes	Firefox 3.5+, Safari 4+, Chrome, IE 8+, Opera 11.5+
Codenvy Editor	2.10.17, 2014-01-17	Eclipse	-	Yes	Firefox 3+, Chrome, Safari 3+, Internet Explorer 8+, Opera 9+
CodeMirror	5.39.0, 2018-06-20	plain textarea	Free	Yes	Firefox 3+, Chrome, Safari 3+, Internet Explorer 8+, Opera 9+[1]
Codeanywhere	6.0	SublimeText	-	No	Firefox 3+, Chrome, Safari 3+, Internet Explorer 8+, Opera 9+

Tabla 3.2-1: Editores de código fuente

3.3 Web

Para el desarrollo web existen multitud de *frameworks*, como son *Vue.js* [14], *React* [24] o *AngularJS* [25]. Este último, *AngularJS*, es bastante diferente a los otros dos, ya que se basa fuertemente en *TypeScript* [26], un superconjunto de JavaScript, el cual es efectivo para aplicaciones de gran escala. Sin embargo, este no es nuestro caso, ya que la aplicación que se va a desarrollar es simple en número de módulos, pudiendo incluso pasar a formar su propio módulo. La complejidad de *AngularJS* para el alcance de nuestro proyecto es demasiada.

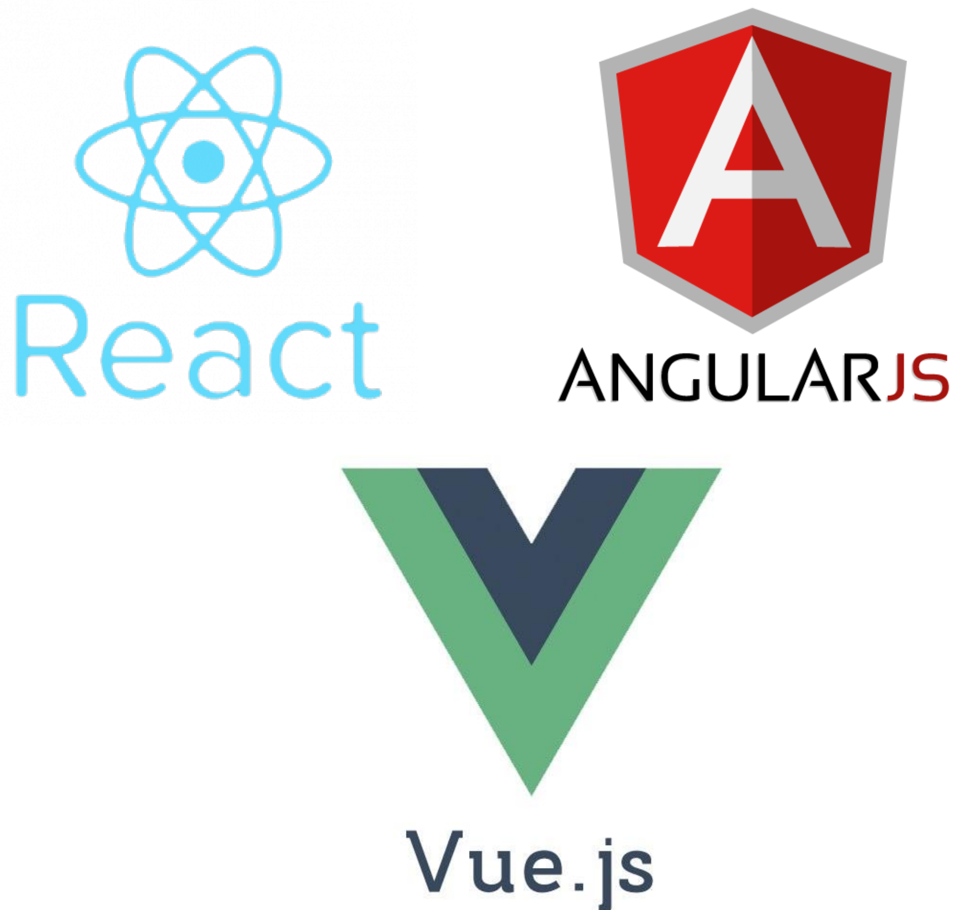


Figura 3-6: Frameworks web

Vue.js y *React* son herramientas muy parecidas, ya que ambos se basan en la virtualización de DOM, proporcionando una gran modularidad gracias a su separación en diferentes librerías independientes. Sin embargo, React define todo en formato JavaScript, incluido el HTML, por lo que dados nuestros conocimientos veíamos más natural un *framework* en el que parte de la sintaxis de los ficheros HTML se respetase. Sobre este *framework* hemos usado la librería de *Boostrap Vue* [16] que, si bien no se adaptaba a nuestras necesidades completamente, era la opción más madura para facilitar el trabajo en *Vue.js* debido a la ausencia de alternativas compatibles con nuestras necesidades.

3.4 Conclusiones

Basándonos en el estudio del arte sobre el producto completo, observamos cómo, aunque existen herramientas de depuración de memoria a nuestra disposición, estas son difíciles de entender y aprender a manejar dado su carácter técnico. Por ello, aprovechando el punto de apoyo que nos da el proyecto C '*Cake*' sobre la versión antigua de Blockly y la actualización de esta plataforma, hemos decidido desarrollar una versión más user-friendly para la creación de programas C y la visualización su gestión de memoria, permitiendo que aquellas personas que se estén introduciendo en el mundo de la programación tengan una base desde la que desarrollarse.

Concretamente para este proyecto, se han utilizado aquellas herramientas que, tras un estudio en profundidad previo, permiten implementar de una manera agradecida los requisitos de los módulos, así como las conexiones necesarias para el funcionamiento correcto de la aplicación web, todo ello de manera que el usuario tenga una sensación de fluidez a la hora de usarla.

4 Diseño

4.1 General

4.1.1 Requisitos funcionales y no funcionales

Requisitos funcionales de la web

FN1: Permitir la navegación entre las distintas pestañas manteniendo los datos.

Requisitos funcionales de Blockly

FN2: Crear bloques que contienen significado en C, como funciones, punteros o TADs.

FN3: Rellenar y modifican los bloques con los datos que se quieren aplicar.

FN4: Borrar bloques o grupos de bloques.

FN5: Mover y situar los bloques en el lugar que se quiera.

FN6: Permitir la interacción con el workspace, ampliar, reducir o desplazar el mismo.

FN7: Guarda el estado actual del programa como XML.

FN8: Seleccionar un archivo XML que contenga la estructura del Blockly y cargarlo.

FN9: Habilitar o deshabilitar bloques.

FN10: Acceder al menú contextual de un bloque.

Requisitos funcionales del código y la memoria

FN11: Simular la ejecución paso a paso del código mostrando la evolución de la memoria.

FN12: Modificar el espacio que ocupa cada una de las secciones de la página.

FN13: Permitir la interacción con la parte visual de cada memoria, ampliar, reducir o desplazar el contenido.

Requisitos no funcionales de la web

NFN1: Que la página sea “responsive”.

NFN2: Que el uso de la web sea intuitivo y sencillo.

Requisitos no funcionales de Blockly

NFN3: Capacidad de definición de tipos propios internamente.

NFN4: Correcto funcionamiento interno de las variables.

NFN5: Adecuar la funcionalidad de cada bloque a su contexto, de forma que un mismo bloque cambie dinámicamente dependiendo del lugar donde esté situado.

NFN6: Permitir la creación de tipos ilimitados.

NFN7: Mantener el estado de todos los bloques tras su guardado y cargado.

NFN8: Desarrollo de los bloques internamente.

Requisitos no funcionales del código y la memoria

NFN9: Generar correctamente el código C a partir de los bloques creados.

NFN10: Interpretar todo el código almacenando la información a mostrar en el paso a paso.

NFN11: Resaltar la línea de código que se ejecuta en el paso a paso.

NFN12: Mostrar adecuadamente el contenido de las memorias del programa en cada paso.

4.1.2 Casos de uso

El siguiente caso de uso muestra las funcionalidades disponibles en el entorno de la web:

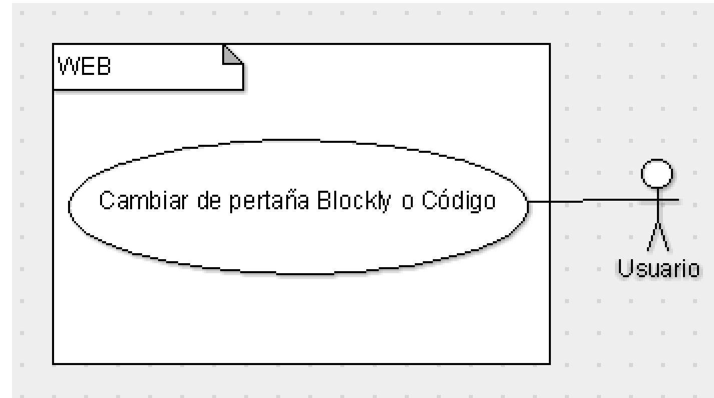


Figura 4-1: Caso de uso Web

Este caso de uso se centra en las funcionalidades presentes en Blockly:

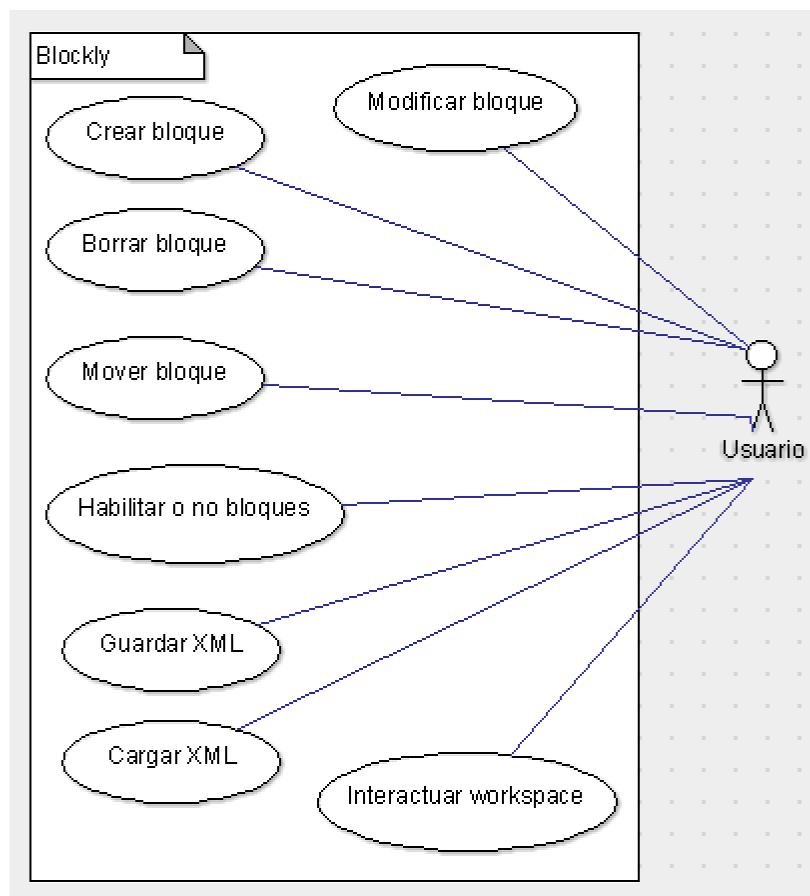


Figura 4-2: Caso de uso Blockly

El siguiente caso de uso trata de las acciones disponibles para el usuario desde la pestaña de código:

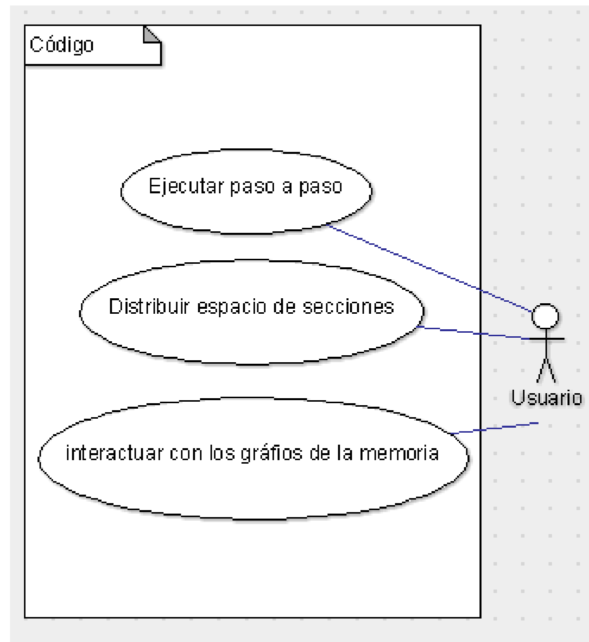


Figura 4-3: Caso de uso Código

4.1.3 Matriz de trazabilidad

El diseño general de nuestra aplicación es el siguiente:

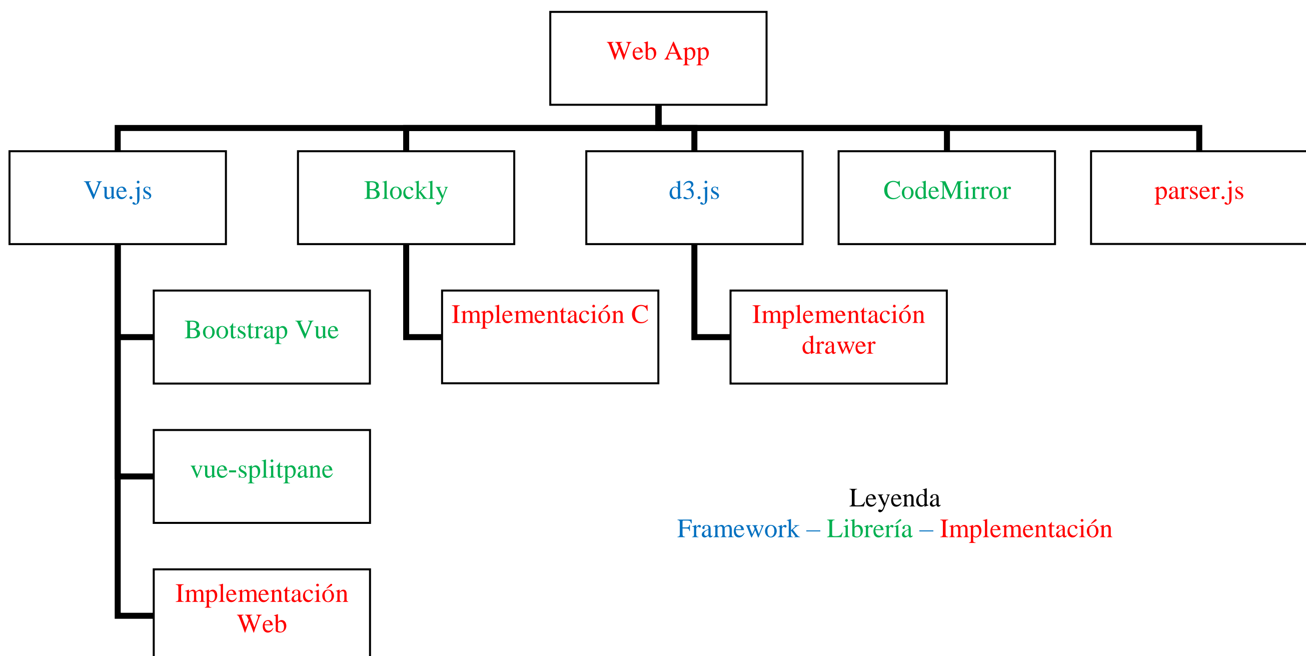


Figura 4-4: Diagrama módulos

En él podemos observar como nuestra aplicación se ha desarrollado en cuatro módulos principales:

- Mediante el framework **Vue.js**, hemos implementado el módulo web que da visibilidad al resto de ellos.
- El módulo de **Blockly**, en el cual especificamos nuestra implementación para C, así como algunos cambios personalizados sobre su librería.
- El módulo del '***parser***', en el cual analizamos el código generado por Blockly para asignarle una representación gráfica paso a paso.
- El módulo de la **representación gráfica**, donde nos encargamos de dibujar la memoria dinámica dada la salida del módulo del parser.

Dados los módulos anteriores, a continuación mostramos la matriz trazabilidad del proyecto completo.

	Web	**Blockly**	**Intérprete**	**Representación gráfica**
FN1	X			
FN2		X		
FN3		X		
FN4		X		
FN5		X		
FN6		X		
FN7		X		
FN8		X		
FN9		X		
FN10		X		
FN11			X	X
FN12	X			
FN13				X
NFN1	X			
NFN2	X			
NFN3		X		
NFN4		X		
NFN5		X		
NFN6		X		
NFN7		X		
NFN8		X		
NFN9		X		
NFN10			X	
NFN11			X	X
NFN12			X	X

Tabla 4.1-1: Matriz de trazabilidad

4.1.1 Ciclo de vida del desarrollo del proyecto

Para el desarrollo del proyecto se ha seguido un ciclo de vida en espiral como se muestra en la siguiente figura:

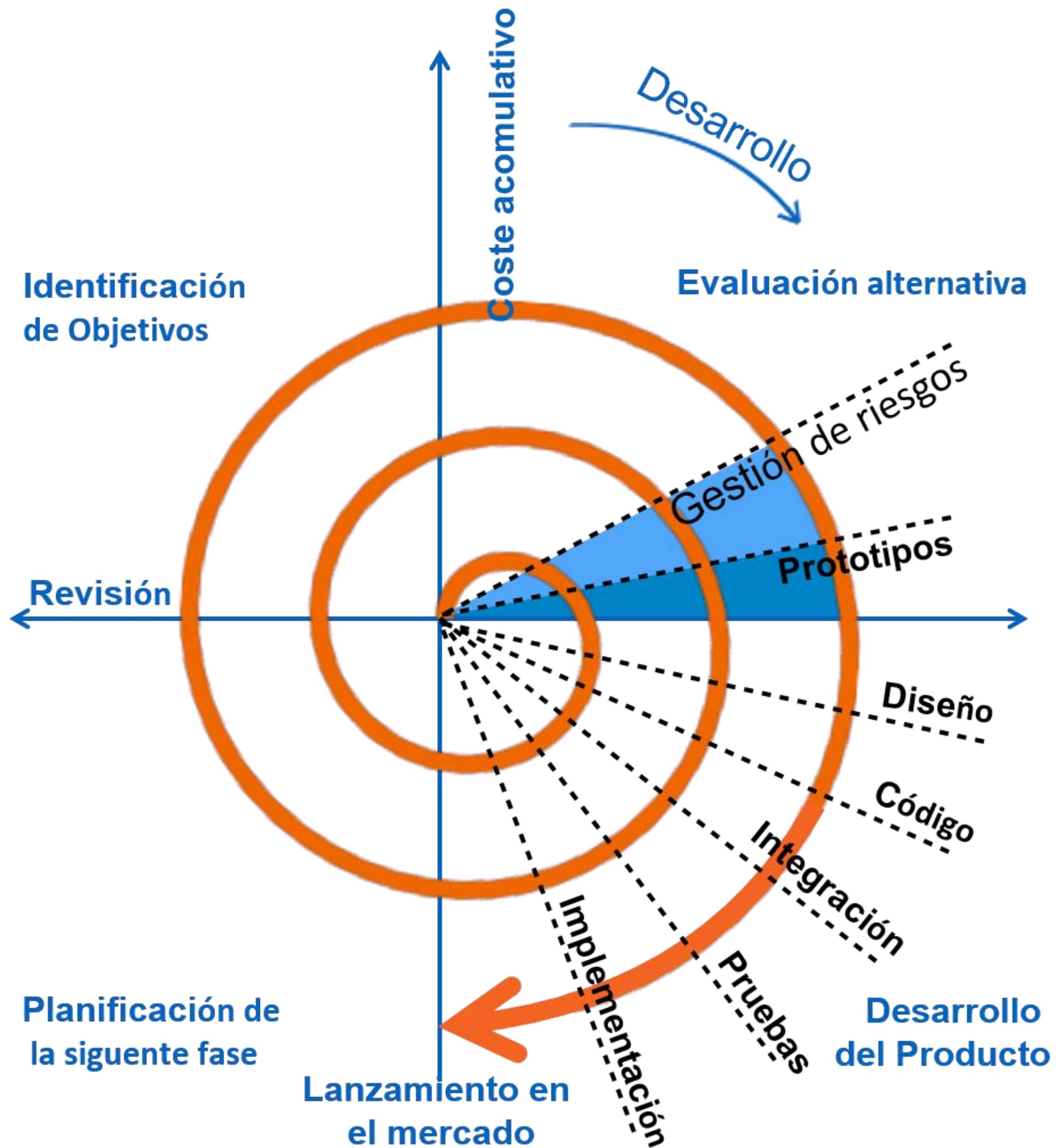


Figura 4-5: Ciclo de vida

Debido a la existencia de módulos tan distintos y que requieren de una correcta comunicación entre sí, se ha optado por un ciclo de vida basado en prototipos, para poder ir realizando pruebas conjuntas a medida que avanza el desarrollo. Y de entre las múltiples opciones se ha elegido éste por ser dinámico y flexible.

4.2 Web

4.2.1 Introducción

El diseño de la página web, como muestra la siguiente figura, está basado en pestañas, teniendo cada una de ellas su contenido de manera independiente. La parte común de la página web entre pestañas está situada en la barra superior, donde podemos observar (1) 2 pestañas entre las que navegar y el botón de realizar paso a paso del intérprete. A su derecha (2) tenemos la funcionalidad de guardado y cargado del workspace de Blockly. Dicha funcionalidad viene dada por el navegador, por lo que según cual se esté usando, se verá de una forma concreta u otra.

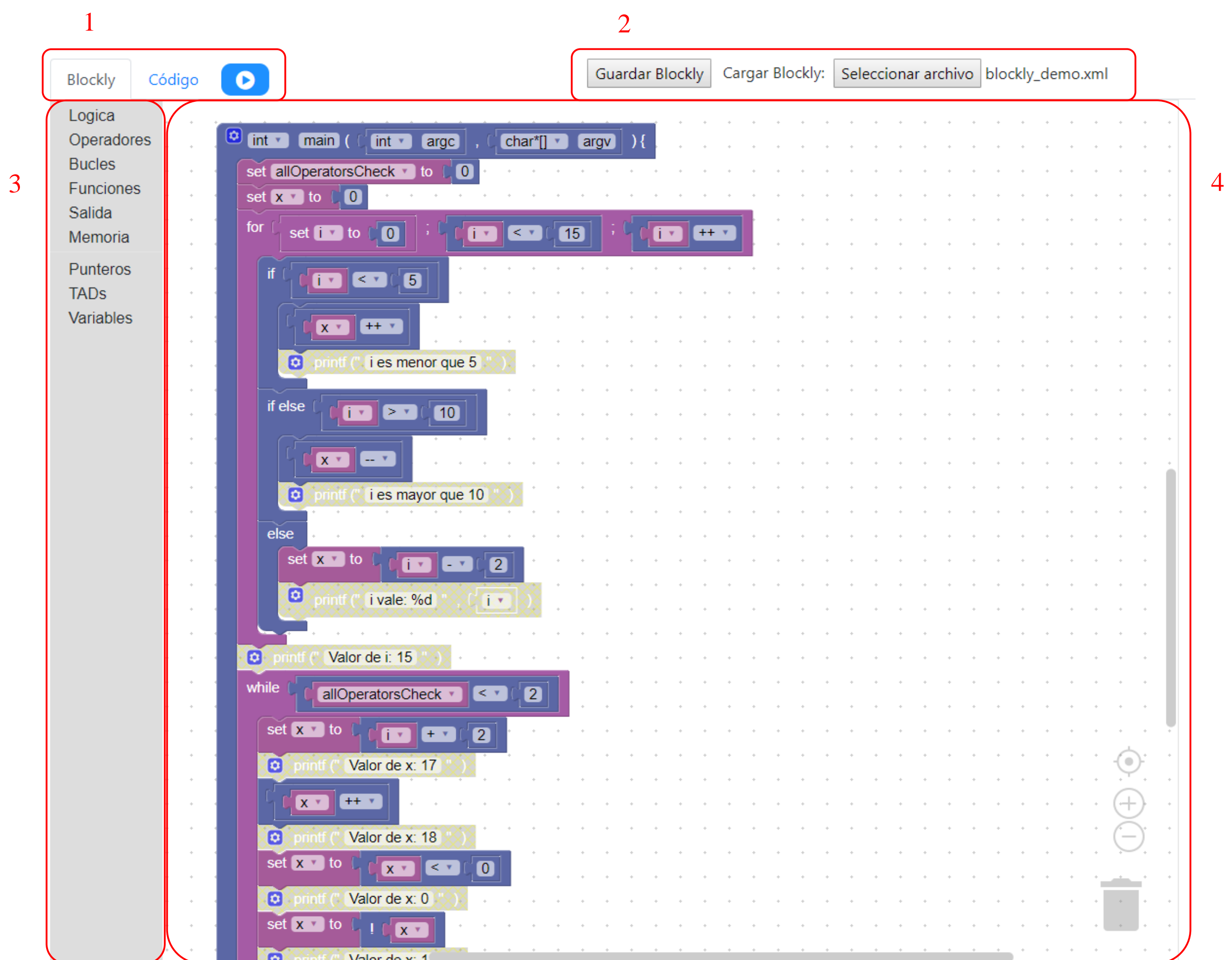


Figura 4-6: Página principal

4.2.2 Blockly

En la pestaña de Blockly podemos observar un menú lateral (3) [11] donde están situadas todas las categorías de bloques disponibles. Haciendo click en cada una de ellas obtendremos una lista de bloques desde donde elegir y arrastrarlos al workspace (4) [12], el cual soporta tanto zoom, como un desplazamiento infinito por él, ya que cuantos más bloques estén presentes en él, más grande se irá haciendo. El diseño de cada bloque, tanto la forma como el color, viene dado por su configuración.

4.3 Blockly

4.3.1 Core

El funcionamiento interno de Blockly no es de demasiada relevancia en este caso, así que mencionaremos aquellos apartados que hemos necesitado modificar. Para obtener una información mucho más detallada de la API de Blockly, consultar su página de referencia [13].

El uso de la librería Blockly se divide entre la generación de los bloques necesarios y sus respectivos generadores de código. Sin embargo, para poder desarrollar más fácilmente nuestra implementación, hemos necesitado modificar ligeramente el Core de Blockly, en concreto el manejo de tipados, datos que explicaremos más a fondo en el apartado de desarrollo. En la siguiente figura podemos observar la estructura del módulo de Blockly.

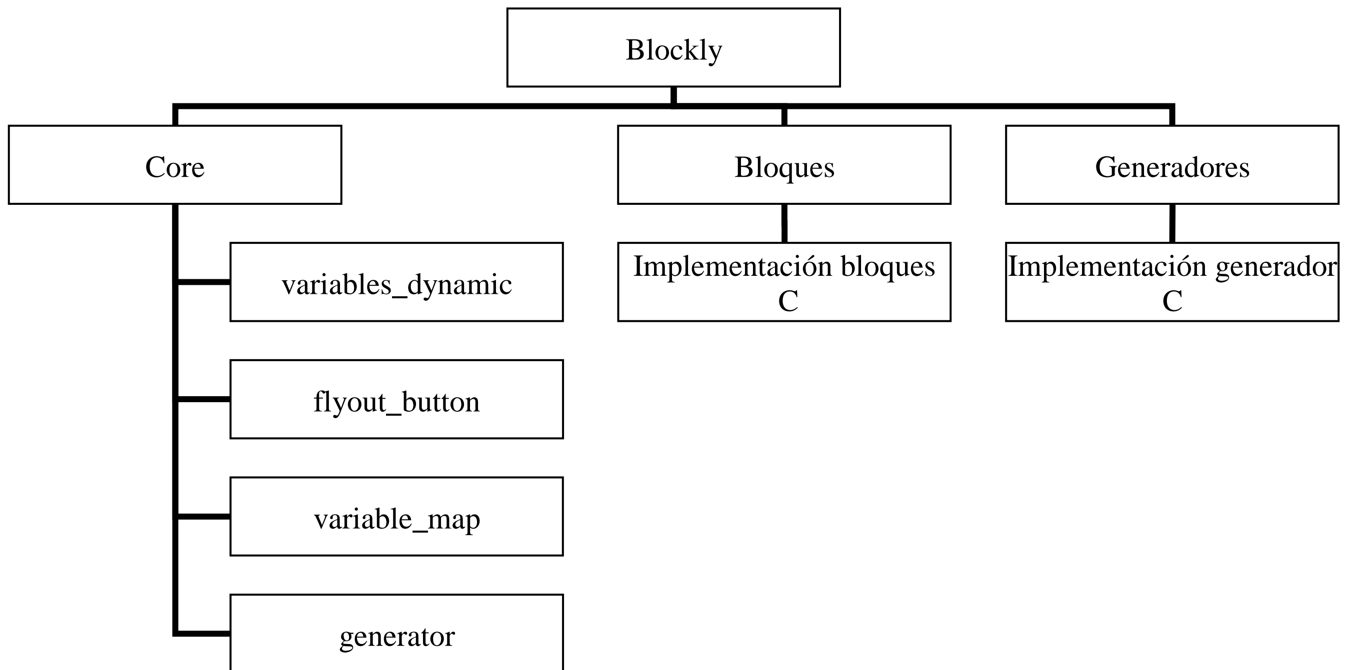


Figura 4-7: Diagrama de Blockly

La modificación de la clase '*generator*' se realizó para facilitar el trabajo del intérprete, de forma que la salida tiene, de manera asegurada, un espacio por delante y otro por detrás de cada paréntesis. En caso de mejorar su funcionamiento, este cambio se podría revertir.

4.3.2 Blocks

Para entender mejor tanto Anexo I como los siguientes puntos, vamos a detallar cómo se realizan las conexiones en Blockly, qué son los inputs y qué tipos hay. Existen 3 tipos de conexiones entre bloques que permiten conectarlos entre sí, pudiendo un mismo bloque tener varios de ellos. Estas conexiones son '**output',** la cual tiene una conexión por la izquierda, **conexión superior**, la cual es una conexión entrante por arriba y **conexión inferior**, que es una conexión saliente por debajo del bloque. Se puede aplicar uno o varios tipos de salida a estas conexiones.

Un **input** define cómo se conectarán los bloques entre sí. En Blockly existen 3 tipos de input:

- **Value input** (ver figura 3-8): En este input se podrán colocar todos aquellos bloques que tengan un 'output'. Se pueden aplicar restricción de tipos sobre el bloque que se conectará, validandose esta restricción con que coincida con uno de ellos.

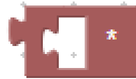


Figura 4-8: Bloque de tipo output con un value input

- **Statement input** (ver figura 3-9): Este input acepta aquellos bloques que tengan una conexión superior. Se pueden aplicar restricción de tipos sobre el bloque que se conectará, validándose esta restricción con que coincida con uno de ellos.

Figura 4-9: Bloque con conexiones superior e inferior que acepta un statement input

- **Dummy input**: Este input no varía la apariencia del bloque ni añade ninguna conexión. Es usado para introducir un campo sin necesidad de uno de los bloques anteriores.

Todos los inputs anteriores aceptan campos. Aunque en Blockly existen más tipos de campos, nosotros usamos los siguientes (puede verse su representación gráfica en las figuras 3-10 y 3-11):

- **Campo de texto**: Tal como su nombre indica, se usa para introducir un campo de texto.
- **Campo numérico**: Utilizado para introducir un número, al cual se le pueden aplicar restricciones de rango.
- **Dropdown opciones**: Es un menú desplegable al cual le insertas manualmente las opciones que quieras que aparezcan.
- **Dropdown variables**: Es un menú desplegable el cual se rellena automáticamente con todas las variables existentes.

Figura 4-10: Ejemplos de campos

Para organizar la estructura de generación de bloques hemos decidido incorporar nuestros cambios dentro de la compilación propia de Blockly, consiguiendo de esta forma un mayor rendimiento en el programa, ya que se hace una petición a los servidores de Google para comprimir los archivos JavaScript al menor número de caracteres y tamaño. Dentro de dicha carpeta, '*blockly/blocks*', hemos dividido nuestros ficheros por categorías, usando en todos ellos el prefijo '*c_*' para diferenciarlos claramente de aquellos originales de Blockly.

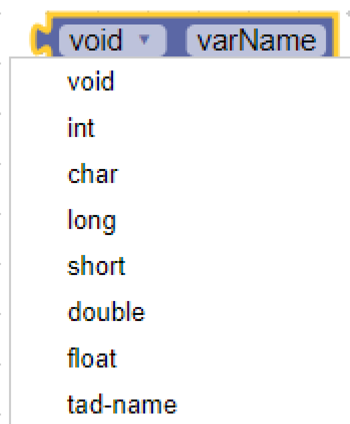


Figura 4-11: Dropdown de tipos

Dentro de cada fichero se sitúan las declaraciones de los bloques y cualquier función específica para su ámbito. Sin embargo, usamos una serie de funciones auxiliares generales comunes a todos ellos, las cuales hemos situado en un fichero común, ‘*blockly/generators/c.js*’.

Figura 4-12: Toolbar de Blockly

Por su parte hemos dividido los bloques en la interfaz gráfica de una manera similar a la organización de ficheros, como se muestra en la figura 3-12.

4.3.3 Generator

En Blockly existe un módulo, el generador, que se encarga de transformar el contenido de un bloque en código. Un mismo bloque puede ser traducido a diversos lenguajes usando generadores distintos, sin embargo, como los bloques originales estaban orientados a lenguajes no tipados, hemos tenido que añadir los nuestros propios y hacer su correspondiente generador.

El diseño del generador sigue la siguiente estructura:

Blockly.GenName.blockName = function (block) { … }

Donde ‘*Blockly*’ es el paquete base, ‘*GenName*’ es el nombre del generador que se desea implementar, en nuestro caso C, y ‘*blockName*’ es el nombre del bloque al que quieres implementar la generación de código, la cual viene dada por una función que recibe el bloque. Toda la generación de código sigue este esquema.

Además, es necesario implementar en el paquete base del generador ‘*Blockly.GenName*’ los parámetros del lenguaje escogido, como son la prioridad de los operadores, las palabras reservadas, o las funciones de inicio y fin de generación del código.

5 Desarrollo

5.1 Web

5.1.1 Introducción

La web, como ya hemos visto, está compuesta por 2 pestañas, Blockly y código. Para su implementación hemos usado el framework *vue.js* [14] para el diseño gráfico de la web, el framework *d3.js* [15] para el desarrollo gráfico de la gestión de memoria, las funciones adecuadas de *Blockly* para inicializar el workspace y, por último, el intérprete.

La lógica principal de la web está situada en el fichero *vue.js*, donde, con el apoyo de *Boostrap Vue*,[16] una librería front-end CSS, la cual hemos modificado ligeramente para adaptarla a nuestro caso de uso. Para explicar este cambio, vamos a explicar un poco el proceso que sigue *Boostrap Vue* para actualizar la aplicación.

Boostrap añade la capacidad de generar comportamientos dinámicos en la aplicación web simplemente añadiendo etiquetas en el HTML, en nuestro caso *<b-tab>* y *<b-tabs>*. Estos dos elementos forman las pestañas, y lanzan eventos de click cuando son pulsados. Sin embargo, estos eventos son lanzados antes de que terminen de renderizar el contenido de cada pestaña, y dado que nuestros renderizados necesitaban que los elementos padre estuviesen completos no podíamos crear correctamente ninguno de nuestros módulos al cambiar entre tablas.

Dado que ninguna de las alternativas nos proporcionaba un modo de ejecutar algo después del renderizado de manera nativa, para poder solucionar este problema, modificamos el código base de *Boostrap* para añadir un callback, '*manageTabClick*' justo al final de su renderizado, de manera que en ese punto todos los datos que necesitamos están listos. Para asegurarnos que se ejecuta exclusivamente cuando está todo renderizado, activamos una bandera mediante la bandera *enterTab* cuando lo comprobamos. Para saber a qué pestaña tenemos que hacer cambio, capturamos el index del evento de click en la pestaña.

Para realizar la **carga** de ficheros de Blockly usamos un input HTML de tipo file, por lo que la lógica de carga de ficheros viene dada por el navegador correspondiente, aceptando exclusivamente ficheros de tipo .XML. Para realizar la **descarga** de ficheros de Blockly usamos la implementación en HTML5 de FileSaver [21].

5.1.2 Blockly

La parte visual de Blockly viene fijada por su propia implementación, sólo es necesario indicarle un elemento HTML donde inyectar su *workpsace*. En nuestro caso, Blockly se generará dentro del *<div>* con id '*content_blocks*'. Es necesario especificar un elemento extra, el cual debe ser elemento DOM conteniendo el código XML a ser usado para generar la '*toolbox*' [11], es decir, el menú lateral donde se mostrarán los bloques disponibles. Para ello, hemos creado el elemento *<xml>* con id '*content_xml*', donde separamos los bloques por categorías. No todos los bloques implementados en código están definidos en el toolbox. Para más información acerca de este proceso, referirse a la documentación de Blockly [18].

Una vez ha sido inicializado el entorno, para no empezar vacío cargamos un preset con una función main ya definida en forma de bloques. Esta función está definida en una etiqueta *<xml>* con id '*startBlocks*', situada dentro del propio *index.html*, donde siguiendo la sintaxis de Blockly se genera el bloque raíz de la función main con todos los elementos necesarios.

5.2 Blockly

5.2.1 Introducción

Para desarrollar el apartado de Blockly, hemos necesitado modificar su Core para que permitiese una mejor adaptación a los tipos de C, así como la incorporación de arrays, punteros y TADs. A partir de este código base, el desarrollo de la aplicación Blockly se divide en la declaración de los bloques, incluyéndose tanto su forma física como toda funcionalidad dinámica necesaria, y la especificación del código generado por cada bloque.

Se dividirá en tres apartados, explicando primeramente los cambios en el Core, ya que son necesarios para el funcionamiento del conjunto del módulo. A continuación, se comenta en qué consiste la creación de bloques y el funcionamiento de aquellos dinámicos, datos necesarios para comprender el funcionamiento del generador del código. La definición específica de cada bloque está presente en el anexo B. Por último, explicaremos el funcionamiento de la generación de código, así como del de todas aquellas funciones que han sido desarrolladas alrededor de Blockly para implementar el funcionamiento avanzado necesario para la creación de tipos, punteros y arrays. A continuación, explicaremos con más detalle el contenido de cada apartado.

5.2.2 Core

La razón principal de las modificaciones realizadas en el Core de Blockly ha sido mejorar el nivel de abstracción en lo relativo a los tipos de las variables. Para lograrlo, el primer paso fue modificar el fichero '*variable_map.js*' para incorporar una serie de tipos primitivos existentes desde el inicio de la ejecución de Blockly. Es de este fichero desde donde se alimenta el resto de Blockly a la hora de cualquier tipo de modificación sobre una variable, ya sea su creación, modificación o borrado. La estructura de datos de las variables es la que sigue:

Clase **variable_map**: Contiene un mapa que guarda las variables declaradas, en el cual la clave viene dada por un tipo, y su valor es una lista con las variables de dicho tipo. También guarda una referencia al *workspace* al que pertenece. Antes de crear ninguna variable, se comprueba que tanto su ID (opcional), como su nombre son únicos, incluso entre tipos. Cada variable viene dada por la clase *variable_model*.

Dada que esta variable solo devuelve los tipos de las variables existentes, hemos añadido una serie de tipos primitivos coincidentes con los de C para permitir su creación desde el principio. Los nuevos tipos de datos definidos en los *typedef*, así como punteros y arrays se analizan de forma externa a esta clase.

Clase **variable_model**: El modelo usado por la variable guarda cuatro valores:
• El *workspace* donde ha sido creada.
• El nombre de la variable
• El tipo de la variable
• La id de la variable

Si bien se puede acceder a toda la estructura de estas clases de manera manual, para garantizar la estabilidad del programa se debe usar exclusivamente las funciones de acceso a dichas variables.

En la siguiente figura podemos observar cómo, declarando tanto un TAD '*point*', y un puntero a dicho TAD, estos aparecen disponibles para la creación de una variable en su categoría lateral dinámica:

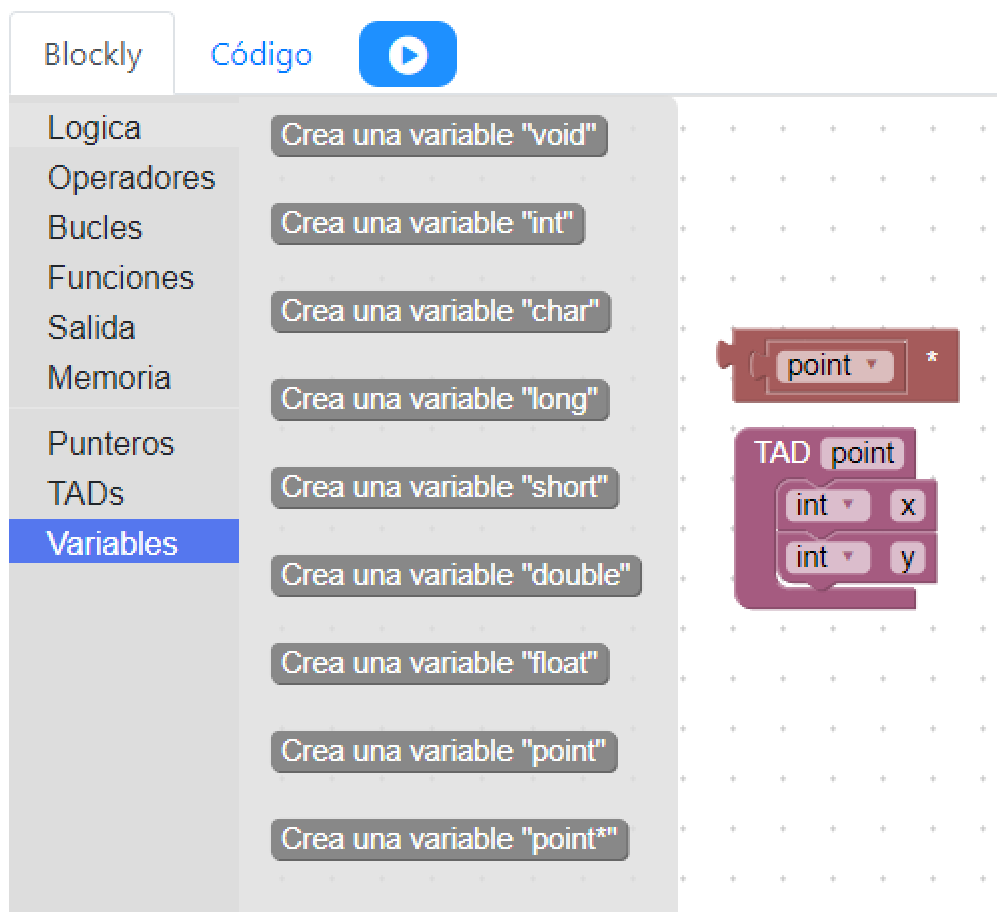


Figura 5-1: Creación de TAD y punteros y su visualización en las variables

Clase **variables_dynamic**: Esta clase se encarga de rellenar la categoría personalizada de las variables, localizada en el lateral izquierdo del panel de Blockly, dinámicamente con todos los tipos disponibles, tanto los primitivos, como TADs, punteros o arrays. Para lograr una mayor generalización y permitir que se incluyan todos los tipos creados, se han modificado las funciones encargadas de rellenar el XML. añadiendo un nuevo parámetro al botón (ver clase *flyout_button*) indicando el tipo de la variable que se va a crear.

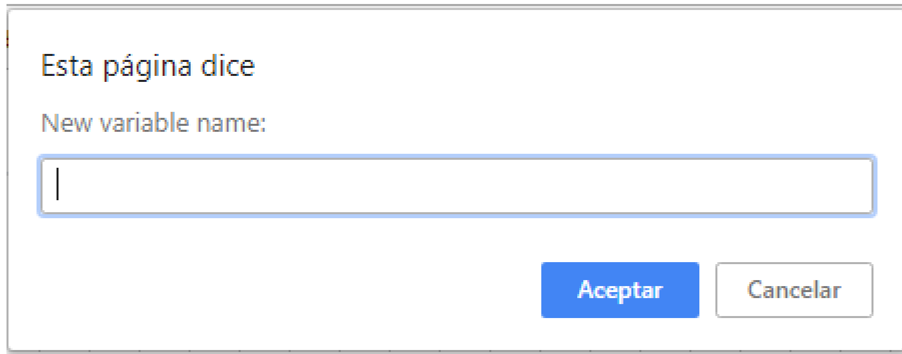


Figura 5-2: Creación de variable

Clase **flyout_button**: Para poder añadir el tipo al botón, ha sido necesario añadir un nuevo parámetro a esta clase, el tipo de la variable a crear. Esto permite un nivel de abstracción superior en la clase *variables_dynamic.js* al hacer independiente la creación de una variable del tipo de variable que es. En la figura 4-2 podemos ver el recuadro modal resultante de hacer click en el botón *flyout_button*.

5.2.3 Blocks

Blockly admite la definición de bloques tanto en formato JSON como en JavaScript. Dado que la funcionalidad avanzada sólo está disponible mediante JavaScript, hemos decidido usar esta forma.

En la carpeta 'blockly/blocks' se maneja la declaración de los bloques de Blockly, así como todas las funciones necesarias para obtener su funcionamiento dinámico. Las características dinámicas se pueden codificar de dos maneras:

- Escuchando **eventos** de Blockly en la función '*onChange(event)*' de cada definición de bloque. Esta función se llama con cada evento de cualquier tipo, y es nuestra responsabilidad filtrar los eventos. En la siguiente figura podemos apreciar cómo un mismo bloque, dependiendo de dónde esté situado o qué parámetro se haya escogido cambia de forma.

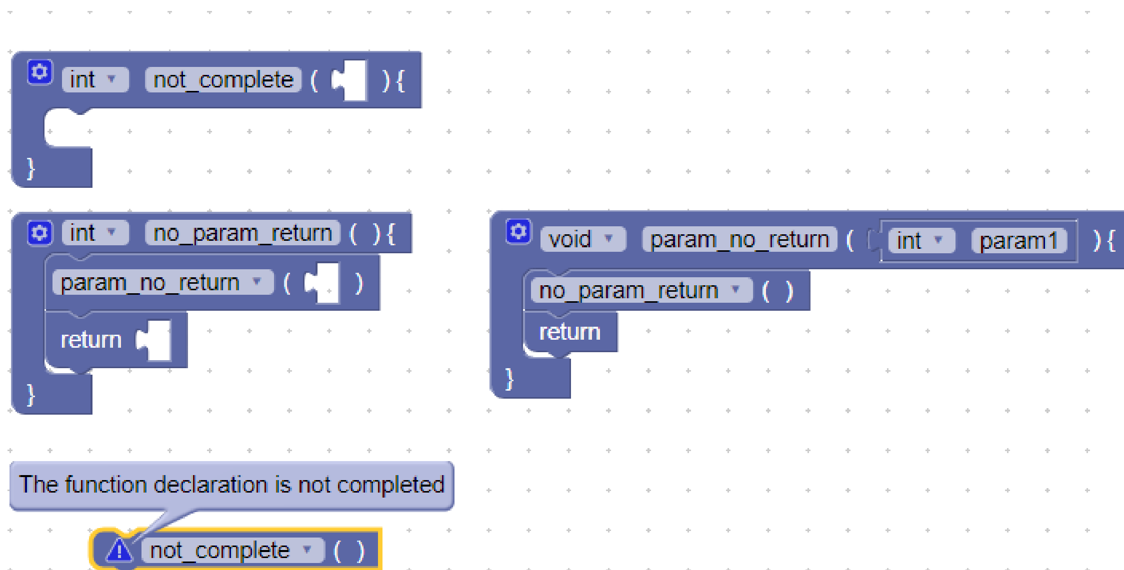


Figura 5-3: Comportamientos dinámicos mediante eventos

- **Mutators**: Implementando las funciones '*decompose(workspace)*' y '*compose(containerBlock)*' se consigue generar los mutator. Estos consisten en un workspace en miniatura mediante los cuales se modifica el comportamiento. En este ejemplo vemos como se usa para modificar los parámetros de una función.

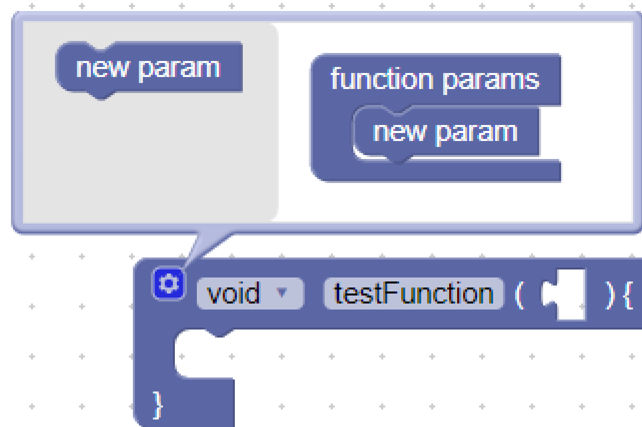


Figura 5-4: Mutator

Para permitir guardar y cargar un estado de Blockly es necesario implementar en todas las funciones que implementen un comportamiento dinámico mediante las siguientes funciones:

- La función '*mutationToDom( )*' se usa para generar un documento XML que contenga todos los datos necesarios para reconstruir el bloque posteriormente. Estos parámetros son guardados como atributos XML, cuyas claves deben estar definidas en minúsculas.
- La función '*domToMutation(xmlElement)*' realiza el proceso contrario. Dado un documento en XML, debemos leer sus atributos y interpretarlos correctamente para reconstruir el bloque, consiguiendo que esté en el mismo estado que anteriormente.

Blockly se encarga de llamar a estas funciones de todos los bloques presentes en el workspace, guardando los datos internos necesarios y sumando los que hemos especificado nosotros. Usando las funciones anteriores, hemos modificado o añadido los siguientes ficheros por categorías:

- c_bucles.js
- c_functions.js
- c_logica.js
- c_operaciones.js
- c_punteros.js
- c_salida.js
- c_tads.js
- variables_dynamic.js

Para ver las características concretas de cada uno de los bloques referidos en estos ficheros, ver el anexo B.

A la hora de definir nuevos tipos, distinguimos entre dos casos: la creación de punteros y arrays, y la creación de TADs. Para la **creación de punteros y arrays**, usamos bloques lógicos que no generan código ninguno, si no que crean dentro de la estructura de Blockly unos tipos que simulan ser arrays o punteros. Estos tipos simulados son tal como *int** o *char[5]*, tipos que claramente en C no existen, pero son necesarios para poder diferenciarlos de las variables normales. Sin embargo, para mantener la compatibilidad tanto con las definiciones correctas de código C, '*char var[5];*', como con la comprobación de tipos interna, se ha implementado un wrapper que permite, dada una tupla [tipo, valor], formatear esa misma tupla de manera correcta. Por ejemplo, la tupla [char[5], var] se formatearía como [char, var[5]]. Dicha función es '*formatTypeVar*' y está definida en el generador.

A la hora de **crear TADs** la lógica interna es similar, dado que también es creado usando bloques lógicos. Al igual que en el caso de punteros y arrays, internamente no se inserta ningún tipo en el mapa de variables. En su lugar, cuando un bloque hace una petición para obtener los tipos existentes, se realiza un escaneo del tablero en busca de definiciones de TAD, obteniendo una lista con todos ellos y ofreciendo al usuario la posibilidad de crear una variable con dicho tipo. Es en ese momento donde el TAD, puntero o array pasa a estar declarado dentro de la lógica interna de Blockly.

En ambos casos, si se declara múltiples veces un mismo TAD, puntero o array solo será válido el primero que se encuentre en el tablero, siendo este orden dependiente de la librería de Blockly, por lo que no garantizamos ningún orden concreto.

5.2.4 Generator

Los generadores de Blockly están situados en '*blockly/generators*'. Para implementar el de C, los ficheros utilizados son '*blockly/c.js*', que define aquellas características propias del lenguaje, como las prioridades de los elementos, la necesidad de paréntesis o las funciones de inicio y fin de generación de código. También usamos este fichero para almacenar todas las funciones auxiliares usadas para facilitar la obtención de datos almacenados en Blockly, como el tipo de una determinada variable, o los TADs declarados.

Las funciones principales para la generación de código C son las siguientes:

Nombre de función	Descripción
addReservedWords	Declara una lista de nombre de variables ilegales.
init	Inicializa la base de datos de variables
finish	Esta función se llama una vez el código ha sido generado. Es usado para anteceder el código con todo lo necesario.
scrubNakedValue	Se usa para añadir el ';\n' necesarios en aquellos bloques con output que no están conectados a nada, necesario para la corrección del código.
quote_	Se usa para escapar los '\n', ' \ ' y ' ' ' introducidos en los campos de texto de los bloques.
scrub_	Esta función maneja la generación de código C, conjuntamente con los comentarios del bloque, si existiesen.

Tabla 5.2-1: Funciones principales del generador

Las funciones auxiliares que hemos desarrollado para agilizar el desarrollo son las siguientes:

Nombre de función	Descripción
getTypes	Obtiene todos los tipos de las variables definidas de forma que sean compatibles para rellenar los *dropdown* de Blockly
getFunctions	Obtiene todas las funciones definidas en el *workspace* de Blockly, formateándolas de forma que sean compatibles para rellenar los *dropdown* de Blockly.
getTads	Obtiene todos los TADs definidos en el *workspace* de Blockly, formateándolas de forma que sean compatibles para rellenar los *dropdown* de Blockly.
getTadPointers	Obtiene todas las variables de punteros a TADs definidas en el *workspace* de Blockly, formateándolas de forma que sean compatibles para rellenar los *dropdown* de Blockly.
getTadVariables	Obtiene todas las variables de TADs definidas en el *workspace* de Blockly, formateándolas de forma que sean compatibles para rellenar los *dropdown* de Blockly.
getTad	Obtiene el bloque de un TAD en concreto, buscándolo por su nombre.
getAllTypes	Obtiene todos los tipos disponibles, tanto los declarados en variables como los TADs, punteros o arrays definidos en el *workspace*.
getAllTypesOptions	Formatea la lista de tipos de forma que sean compatibles para rellenar los *dropdown* de Blockly.
getVariablesOfFunction	Obtiene todas las variables que se han usado en una función en concreto, devolviendo el string de código C con sus declaraciones.
getVariableType	Obtiene el tipo de una variable pasada por parámetro.
getTypeVarTuple	Dado el nombre de una variable, busca su tipo y devuelve una tupla de manera que, en primer lugar, está el tipo de la variable, y en el segundo su nombre. Esta función es necesaria porque los punteros o arrays están siendo declarados en el tipo internamente, y de esta manera devuelve un tipo y nombre de variable formateados de manera correcta en el lenguaje C.
formatTypeVar	Dado el nombre de una variable y su tipo, devuelve una tupla de manera que, en primer lugar, está el tipo de la variable, y en el segundo su nombre. Esta función es necesaria porque los punteros o arrays están siendo declarados en el tipo internamente, y de esta manera devuelve un tipo y nombre de variable formateados de manera correcta en el lenguaje C.

Tabla 5.2-2: Funciones auxiliares del generador

En la carpeta '*blockly/generators/c*' están situados todos los ficheros encargados de traducir un bloque a código C. Para ello, usamos las siguientes funciones que pone a nuestra disposición Blockly:

---

block.getFieldValue('field_name');

Obtiene el valor de un campo, cuyo nombre pasamos como parámetro.

Blockly.C.valueToCode(block, 'value_block_name', Blockly.C.ORDER_ATOMIC);

Genera el código del bloque unido al actual en el 'value input' pasado por parámetro. Su tercer parámetro indica la fuerza máxima de unión del bloque 'value input'. Esto último se usa para calcular si es necesario o no usar paréntesis. Si el valor localizado en esta llamada es mayor que aquel que devuelve el bloque, se utilizarán paréntesis.

Blockly.C.statementToCode(block, 'c_bucle_for_statement');

Genera el código del bloque unido al actual en el 'statement input' pasado por parámetro.

5.2.4.1 *Prioridades*

Como ya vimos existen bloques de distinto tipo, y dependiendo de su tipo es necesario devolver el código o una tupla [código, prioridad].

Bloque tipo output

Estos bloques devuelven una tupla de la siguiente forma:

return [code, Blockly.C.ORDER_ASSIGNMENT];

El segundo parámetro de la tupla indica la fuerza mínima de unión del bloque, lo cual se usa para calcular si es necesario envolver el código en paréntesis o no. Si el valor localizado en esta tupla es menor o igual que aquel con el que ha sido llamado, se utilizarán paréntesis.

Resto de tipos de bloque

Estos bloques devuelven exclusivamente el código generado.

return code;

5.2.4.2 *Bloques dinámicos*

En los bloques que contienen *mutators* o tienen eventos no sabemos cuántos parámetros tenemos, así que vamos obteniendo los inputs y viendo si existen mediante la función:

block.getInput('input_name');

Una vez sabemos si existen o no, usamos el resto de funciones normalmente, concatenando el código generado. En la siguiente figura podemos ver algunos ejemplos de bloques dinámicos, cuyo número de parámetros no es conocido a la hora de generar el código, ya que son variables.

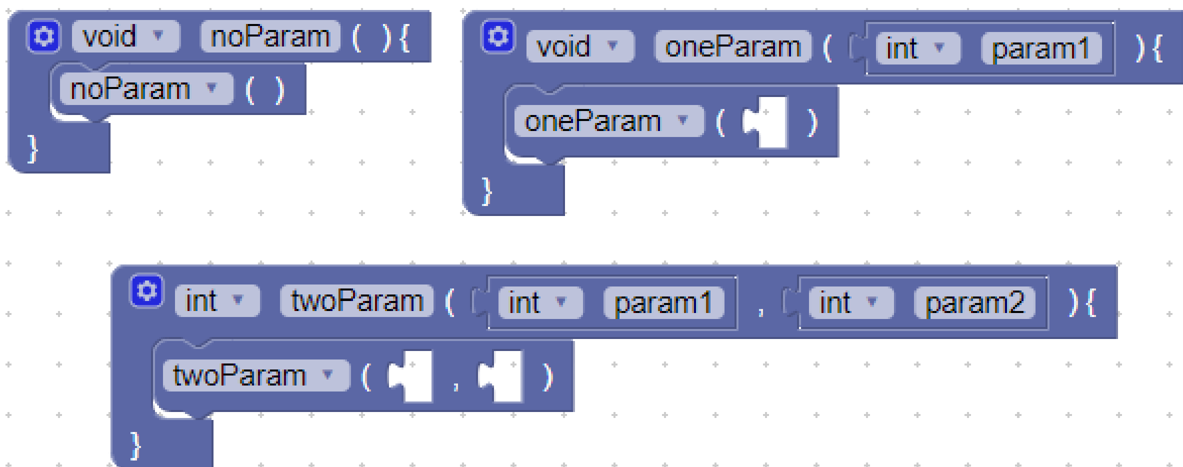


Figura 5-5: Bloques con diferente número de inputs

5.2.4.3 *Compatibilidad con el intérprete*

Todo el desarrollo de estas funciones se ha realizado con las condiciones necesarias para que su salida sea compatible con la entrada del intérprete. Esto ha implicado una serie de limitaciones, llegando en algunos casos a tener que deshacer alguna funcionalidad ya implementada. Debido a la necesidad de paréntesis, una funcionalidad que se ha tenido que deshabilitar son las prioridades del lenguaje C, de forma que aun cuando no serían necesarios estos paréntesis, son colocados. Dado que esto supuso dar un paso atrás en el desarrollo, se guardó una copia del estado del proyecto para no perder esta información y facilitar una futura reimplantación de las prioridades.

Otro de los requisitos es la necesidad de tener exclusivamente un espacio antes y después de cada paréntesis, excepto en aquellos de último nivel, es decir, los que engloban toda la sentencia. Para asegurar esta condición, se ajustan los espacios de toda cadena generada, de forma que la salida de toda función cumple estos requisitos.

6 Integración, pruebas y resultados

Se describe a continuación este aspecto en relación al producto completo.

6.1 Integración

Para empezar el desarrollo de la aplicación web, empezamos a desarrollar los módulos de manera paralela, haciendo en cada uno de ellos las pruebas correspondientes y usando una web provisional. Una vez todos los módulos estuvieron en un estado de maduración suficiente, desarrollamos la plataforma final donde ubicar cada uno de ellos usando *vue.js* y comprobando el correcto cambio entre pestañas sin contenido. Una vez lo tuvimos terminado, rellenamos los huecos con los módulos, y durante este proceso nos encontramos con algunos problemas de visualización al cambiar de pestañas, así como entre el código generado por Blockly con el intérprete. Antes de continuar con el desarrollo nos centramos en conseguir una integración completa entre los diferentes módulos, haciendo en el proceso una demo lo más completa posible.

6.2 Pruebas y resultados

6.2.1 Web

Debido a que la web es algo plenamente visual, todas las pruebas realizadas han sido manuales. Estas pruebas han sido guardadas en forma de demos, guardando el estado del *workspace* que genera el código necesario para realizarlas.

6.2.1.1 *Visualización*

Dado que la navegación dentro del propio *workspace* nunca ha dado problemas, ya que viene bien comprobada por el equipo de Blockly, nos hemos centrado en la corrección de las transacciones entre pestañas y su buen aspecto gráfico. Para ello, hemos probado diferentes combinaciones de pestañas iniciales de la App. En este punto comprobamos que, si la pestaña principal era la de código, no se generaba correctamente el *workspace* de Blockly ya que las dimensiones de los bloques estaban mal escaladas, así como que hacía falta hacer doble click en la pestaña para que funcionase correctamente. Fue aquí donde nos dimos cuenta del problema venía de *Bootstrap* y lo modificamos. Comprobamos que el guardado de Blockly funciona correctamente en cualquier pestaña, sin embargo, si la carga de un fichero Blockly xml se realiza mientras estás en la pestaña de código, los bloques cargados estarán mal dimensionados. En cualquier otro caso, la carga funciona correctamente.

6.2.2 Blockly

Durante la fase de desarrollo individual del módulo, hemos comprobado la corrección del código generado por Blockly comprobando que el código es sintácticamente correcto, sumado con los mecanismos que el propio Blockly pone a nuestra disposición, como la comprobación de que todo bloque usado en el workspace contenga un generador de código. Dado que ninguno de los bugs existentes en el proyecto original de Blockly [20] nos afectaba, hemos supuesto el correcto funcionamiento de los mecanismos internos de Blockly. Mientras elaborábamos la demo completa, encontramos un bug visual en el menú contextual (botón derecho) de los bloques de variables, en concreto la opción que permite generar su opuesto mediante un click, ya que muestra la id de la variable en vez de su nombre.

Una vez obtuvimos la demo completa, comprobamos la correcta integración entre el código generado por Blockly y el intérprete, arreglando todos los fallos que encontramos, todos ellos sintácticos, ya que el intérprete espera una sintaxis concreta de C. Ya que esta demo usa todos los bloques posibles, así como diferentes combinaciones, también ha servido para comprobar el correcto funcionamiento de las funciones de carga y guardado del *workspace* de Blockly.

6.2.3 Producto final

Con todos los módulos probados y una vez aprobado el producto para su uso fuera del entorno de desarrollo, se han comenzado con las pruebas de aceptación, buscando gente de diferentes ámbitos profesionales y se les ha pedido que prueben el producto. La gran mayoría ha aprendido a usarlo rápidamente y han mostrado su conformidad con el estado final del mismo.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

Las aplicaciones web son un método útil y ágil de trabajo que nos permiten desarrollar casi cualquier cosa. El uso de *frameworks* y de librerías externas facilitan el desarrollo de múltiples herramientas para lograr fines más o menos complejos.

En nuestro caso haber desarrollado una aplicación educativa en el entorno web ofrece la posibilidad de que cualquier usuario con acceso a internet mediante un navegador pueda hacer uso de la misma y beneficiarse de las herramientas que hemos desarrollado y enlazado. Estas herramientas proporcionan una ayuda de especial interés a la hora de aprender y afianzar los conceptos básicos de la programación secuencial y con punteros. Permite al usuario un modo de crear código complejo y bien formado sin necesidad de conocimientos avanzados del lenguaje y observar la evolución del programa gracias a la visualización paso a paso de la línea en ejecución y del estado de las memorias del programa. De este modo el usuario será consciente del funcionamiento del lenguaje y será capaz de crear sus propios programas. El propósito de las herramientas desarrolladas está centrado en los casos de uso necesarios en la asignatura de Programación 2.

El uso de Blockly para generar el código mediante cajas ha supuesto una ayuda a nuestro desarrollo, pero también nos ha obligado a indagar acerca de su funcionamiento interno para cambiar cosas como los tipos y las variables. Debido a la poca documentación proporcionada por los desarrolladores de la herramienta se ha requerido un proceso de ingeniería inversa para identificar los cambios necesarios para que sirviese con un lenguaje tipado como C.

Por último, creemos que el proyecto tiene grandes posibilidades para afianzarse como tutorial básico de los principios de la programación en lenguaje C, uso de punteros y gestión de la memoria dinámica pudiendo ser utilizada en la asignatura indicada de la titulación del grado de ingeniería informática y de telecomunicación. El proyecto es ampliable tanto a otros lenguajes como a la integración de librerías externas para aumentar su funcionalidad.

7.2 Trabajo futuro

Como opciones para trabajo en el futuro se plantea, realizar los cambios necesarios en Blockly para poder trabajar con distintos módulos C, como por ejemplo los que aparecen cuando se diseña una aplicación compleja en base a un grupo de TADs, a la vez. Mejorar los bloques de Blockly para controlar dinámicamente los tipos, así como añadir más dinamismo y avisar de un mayor número de errores lógicos en los bloques al usuario.

Asimismo, habría que incorporar algunas limitaciones que se han tenido que imponer en el proceso de desarrollo del proyecto por las características de Blockly y que no han comprometido la finalización del mismo, por ejemplo, la posibilidad de definir variables globales y las otras características de C (fundamentalmente la cuestión de las librerías) que desde el inicio se excluyeron del proyecto.

Incluir más opciones de depuración como por ejemplo permitir entrar o no en los métodos, o retroceder en la ejecución. La posibilidad de avanzar hasta el final directamente o hasta algún breakpoint. Esto facilitaría el uso de la herramienta, ya que ahora mismo si tenemos un programa grande tenemos que recorrer la ejecución paso a paso entrando en todas las sentencias del código las veces necesarias hasta llegar a donde nos interese.

También podría estar bien la opción de cambiar algún valor directamente en el código o en la memoria para realizar mejor las pruebas, no ha sido un requisito para nuestro proyecto, pero sería interesante permitir el cambio de los valores en medio de la ejecución. Esto permitiría, junto con la posibilidad de avanzar hacia atrás y hacia adelante en el paso a paso, la posibilidad de mejorar el proceso de desarrollo de un algoritmo, ya que serían herramientas muy útiles y sencillas.

Se podría incorporar la posibilidad de hacer el proceso bidireccional. En la actualidad sólo se puede generar C desde Blockly, pero sería muy interesante que a partir de un programa C se generara su equivalente Blockly.

La posibilidad de ampliar este proyecto a otros lenguajes, de admitir la inclusión de librerías externas y de añadir lo comentado anteriormente podrían dar pie a la presentación de nuevas opciones de proyectos futuros.

El software del producto se ha desarrollado pensando en el soporte y en la escalabilidad, se ha modularizado y comentado con el fin de facilitar los futuros desarrollos sobre el mismo.

Debido a la ausencia de un proyecto similar en código abierto, podría ser interesante abrirlo a la comunidad, como pudiese ser en *GitHub*, para dar un empuje a su desarrollo y facilitar su mantenimiento, todo ello tras un proceso de documentación y conversión al inglés. Así mismo, se podría extender este producto con un TFG que se encargase de incluir los objetivo omitidos, así como de modularizar completamente el producto, permitiendo la creación de varios módulos de Blockly independientes que actuasen como los diferentes ficheros propios de un proyecto C.

Referencias

[1] Google, «Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/.
[2] Massachusetts Institute of Technology, «MIT App Inventor» [En línea]. Disponible: http://appinventor.mit.edu/explore/index-2.html.
[3] Google, «Blockly Demo: Code» [En línea]. Disponible: https://blockly-demo.appspot.com/static/demos/code/index.html
[4] Computer Research Association, «cake-core» [GitHub]. Disponible: https://github.com/cra16/cake-core
[5] Valgrind, «Valgrind» [En línea]. Disponible: http://valgrind.org/
[6] Josef Weidendorfer, «KCachegrind» [En línea]. Disponible: https://kcachegrind.github.io/html/Home.html
[7] Free Software Foundation, «DDD – Data Display Debugger - GNU Project - Free Software Foundation» [En línea]. Disponible: https://www.gnu.org/software/ddd/
[8] Free Software Foundation, «GDB: The GNU Project Debugger» [En línea]. Disponible: https://www.gnu.org/software/gdb/
[9] SoftIntegration, «Ch -- an embeddable C/C++ interpreter, C and C++ scripting language» [En línea]. Disponible: http://www.softintegration.com/
[10] Fabrice Bellard, «TCC: Tiny C Compiler» [En línea]. Disponible: https://bellard.org/tcc/
[11] Google, «Toolbox | Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/guides/configure/web/toolbox
[12] Google, «Grid | Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/guides/configure/web/grid
[13] Google, «Overview | Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/reference/overview
[14] Evan You, «Vue.js» [En línea]. Disponible: https://vuejs.org/
[15] Mike Bostock, «D3.js - Data-Drivent Documents» [En línea]. Disponible: https://d3js.org/
[16] BootstrapVue, «Bootstrap Vue» [En línea]. Disponible: https://bootstrap-vue.js.org/
[17] PanJiaChen, «vue-splitpane» [GitHub]. Disponible: https://github.com/PanJiaChen/vue-split-pane
[18] Google, «Get Started | Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/guides/get-started/web
[19] Marijn Haverbeke, «CodeMirror: C-like mode» [En línea]. Disponible: https://codemirror.net/mode/clike/
[20] Google, «Blockly issues» [GitHub]. Disponible: https://github.com/google/blockly/issues
[21] Eli Grey, «FileSaver.js» [GitHub]. Disponible: https://github.com/eligrey/FileSaver.js
[22] Kevin David, «Modelos de ciclo de vida del Software» [En línea]. Disponible: https://www.mindomo.com/es/mindmap/fase-comunes-entre-mocelos-de-ciclo-de-vida-del-software-f29d5494c5d146d5a5636f7b3e4a9091
[23] Lifelong Kindergarten, «Scratch - Developers» [En línea]. Disponible: https://scratch.mit.edu/developers
[24] Facebook, «React - A JavaScript library for building user interfaces» [En línea]. Disponible: https://reactjs.org/

[25] Google, «AngularJS — Superheroic JavaScript MVW Framework» [En línea]. Disponible: https://angularjs.org/
[26] Microsoft, «TypeScript - JavaScript that scales» [En línea]. Disponible: https://www.typescriptlang.org/
[27] Óscar Martín Sanz, «Aplicación web para ayuda en el aprendizaje de la gestión de memoria dinámica en programación con el lenguaje C (continuación)». Depositado en la secretaría del departamento de Ingeniaría Informática de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid.

Glosario

API	Application Programming Interface
DDD	Data Display Debugger
GDB	GNU Debugger
MIT	Massachusetts Institute of Technology
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
XML	Extensible Markup Language
DOM	Document Object Model
JSON	JavaScript Object Notation
TAD	Tipo Abstracto de Datos

Anexos

A Manual de uso

Se necesitan las siguientes versiones mínimas de los navegadores. No se garantiza un funcionamiento completo con otras versiones.

Requisitos mínimos:

- Chrome 13
- Firefox 7
- Internet Explorer 10
- Opera 12.02
- Safari 6

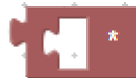
Para aprender el manejo general de Blockly, referirse a su documentación [1]. Las características especiales de nuestra implementación son las siguientes:

Declaración de tipos de datos propio

Antes de poder crear un tipo abstracto de datos como variable, es necesario definir su estructura mediante los bloques de la categoría de TADs. Una vez creados, es posible acceder a su estructura interna mediante los getters y setters en esa misma categoría.

Declaración de punteros y arrays

Siguiendo con el diseño anterior, es necesario declarar con anterioridad el puntero o array que quieres definir mediante los bloques de la categoría '*Punteros*', pudiendo tener un nivel de profundidad ilimitado.



Declaración de funciones

Siguiendo un proceso natural, una función solo aparecerá en el selector de funciones si ha sido declarada con anterioridad, adaptándose la forma del bloque a la función seleccionada.

Paso a paso

Para poder ejecutar correctamente el paso a paso es necesario que el código generado sea correcto. Para tal efecto, existen algunas ayudas en los bloques que avisarán de la incorrección del tablero.

B Manual del programador

7.2.1 Cómo compilar Blockly

Para la compilación de Blockly es necesario tener instalado **Python 2.7** Una vez se tiene la versión correcta de Python, solo es necesario ejecutar la siguiente instrucción en la carpeta de Blockly:

python2.7 build.py

Dicho comando acepta una compilación parcial mediante los siguientes parámetros:

- **core**: Compila el core y los bloques de Blockly
- **accesible**: Genera las versiones accesibles de Blockly
- **generator**: Compila los generadores definidos en /generators
- **langfiles**: Compila los ficheros definidos en /msg/js

Hay que tener en cuenta que este script utiliza servidores de Google para realizar la compresión, por lo que el número de compilaciones que se pueden realizar en un corto periodo de tiempo es limitado. Se recomienda esperar un tiempo entre compilaciones para no ser bloqueado.

En las siguientes páginas explicaremos de forma técnica la configuración de los bloques desarrollados.

Manual de bloques

Este manual técnico está enfocado a facilitar el desarrollo a un futuro programador, proporcionando datos concretos en cada uno de los bloques.

7.2.2 c_bucles.js

En este fichero se define todo aquel bloque relacionado con la declaración de bucles.

Tablas B-1: Bucles

c_bucle_for	
Descripción	Este bloque representa el bucle '*for*', conteniendo inicialización, condición, asignación y cuerpo.
Inputs	• 3 value inputs de tipo '*variables_set_dynamic*'. • 1 statement input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_bucle_while	
Descripción	Este bloque representa el bucle '*while*', condición y cuerpo.
Inputs	• 1 value input de cualquier tipo. • 1 statement input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_bucle_variable_setter	
Descripción	Este bloque es idéntico al bloque original de Blockly para asignación de variables, pero en vez de ser de tipo statement, es de tipo value para permitir introducirlo en los bucles.
Inputs	• 1 dropdown de variables.
Outputs	Salida de tipo '*variables_set_dynamic*'.
Funcionamiento dinámico	Ninguno.

7.2.3 c_functions.js

En este fichero se define todo aquel bloque relacionado con el manejo de funciones.

Tablas B-2: Funciones

c_function_mutator_statement	
Descripción	Este bloque es usado dentro del mutator de declaración de funciones para definir el número de parámetros que esta contiene.
Inputs	• 1 statement input de tipo '*c_function_mutator_param*'.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_function_mutator_param	
Descripción	Este bloque es usado en el mutator anterior y representa el número de parámetros que tiene la función.
Inputs	Ninguna entrada.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_function_declaration	
Descripción	Este bloque es usado para la declaración de funciones.
Inputs	• 1 dropdown de opciones, rellenadas con todos los tipos disponibles. • 1 campo de texto indicando el nombre de la función. • 1 value input por cada parámetro declarado de tipo '*c_function_param*'. • 1 statement input.
Outputs	Ninguna salida.
Funcionamiento dinámico	Este bloque contiene un mutator, mediante el cual se indica el número de parámetros que tendrá la función.

c_function_param	
Descripción	Este bloque representa un parámetro de una función.
Inputs	• 1 dropdown de opciones, rellenado con todos los tipos disponibles. • 1 campo de texto indicando el nombre del parámetro.
Outputs	Salida de tipo '*c_function_param'*
Funcionamiento dinámico	Ninguno.

c_function_call y c_function_call_output	
Descripción	Estos bloques representan las llamadas a funciones. Son bloques idénticos salvo en su forma, ya que uno tiene salida y el otro no.
Inputs	• 1 dropdown de opciones, rellenado con todas las funciones disponibles. • 1 value input por cada parámetro de la función, del tipo declarado en su definición o un literal.
Outputs	**c_function_call**: Ninguna. **c_function_call_output**: Salida de tipo '*c_function_call'*.
Funcionamiento dinámico	Dependiendo de que función se seleccione en el dropdown de funciones, el bloque adapta su forma para contener los parámetros necesarios.

c_function_return	
Descripción	Este bloque representa el retorno de una función.
Inputs	• 1 value input si el retorno no es void.
Outputs	Ninguna salida.
Funcionamiento dinámico	Dependiendo de si la función que lo contiene tiene un tipo de retorno distinto de void o no, el bloque adapta su forma para aceptar un parámetro o no.

c_function_var_get	
Descripción	Este bloque representa un parámetro de la función que lo contiene, de forma que puedan ser tratados como variables en el interior de su cuerpo.
Inputs	• 1 dropdown de opciones, rellenado con todos los parámetros disponibles.
Outputs	Salida de tipo '*variables_get_dynamic*'
Funcionamiento dinámico	Dependiendo de que función lo contenga, el bloque adapta el contenido de su dropdown.

c_function_var_set	
Descripción	Este bloque representa un parámetro de la función que lo contiene, de forma que puedan ser tratados como variables en el interior de su cuerpo.
Inputs	• 1 dropdown de opciones, rellenado con todos los parámetros disponibles. • 1 value input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Dependiendo de que función lo contenga, el bloque adapta el contenido de su dropdown.

7.2.4 c_logica.js

En este fichero se define todo aquel bloque relacionado con la lógica y condiciones.

Tablas B-3: Lógica

c_logic_if y c_logic_if_else	
Descripción	Estos bloques representan las condiciones '*if*' e '*if else*'.
Inputs	• 1 value input de tipo '*c_op_logica*' o '*c_op_literal*'. • 1 statement input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_logic_else	
Descripción	Este bloque representa la condición '*else*'.
Inputs	• 1 statement input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

7.2.5 c_operaciones.js

En este fichero se define todo aquel bloque relacionado con las operaciones aritméticas y lógicas.

Tablas B-4: Operaciones

c_op_emplacement	
Descripción	Este bloque se ha creado para actuar como bloque auxiliar para usar operaciones como si no tuviesen output.
Inputs	• 1 value input de tipo '*c_op_aritmetica*' o '*c_op_logica*'.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_op_aritmetica	
Descripción	Este bloque representa una operación binaria aritmética.
Inputs	• 2 value input de tipo '*variables_get_dynamic*', '*c_op_literal*' o '*c_op_aritmetica*'. • 1 dropdown de opciones, rellenado con las operaciones aritméticas binarias disponibles: '+', '-', '*', '/' y '%'.
Outputs	Salida de tipo '*c_op_aritmetica*'.
Funcionamiento dinámico	Ninguno.

c_op_logica	
Descripción	Este bloque representa una operación binaria lógica.
Inputs	• 2 value input de tipo ‘*variables_get_dynamic*’, ‘*c_op_literal*’, ‘*c_op_logica*’ o ‘*c_op_aritmetica*’. • 1 dropdown de opciones, rellenado con las operaciones lógicas binarias disponibles: ‘&&’ y ‘||’.
Outputs	Salida de tipo ‘*c_op_logica*’.
Funcionamiento dinámico	Ninguno.

c_op_unitaria	
Descripción	Este bloque representa una operación unitaria aritmética.
Inputs	• 1 value input de tipo ‘*variables_get_dynamic*’. • 1 dropdown de opciones, rellenado con las operaciones lógicas unitarias disponibles: ‘++’ y ‘--’.
Outputs	Salida de tipos ‘*c_op_unitaria*’ y ‘*c_op_aritmetica’*
Funcionamiento dinámico	Ninguno.

c_op_literal	
Descripción	Este bloque representa un literal numérico.
Inputs	• 1 introducción por teclado de número.
Outputs	Salida de tipo ‘*c_op_literal*’.
Funcionamiento dinámico	Ninguno.

c_op_literal_string	
Descripción	Este bloque representa un literal de texto.
Inputs	• 1 introducción por teclado de texto.
Outputs	Salida de tipo ‘*c_op_literal*’.
Funcionamiento dinámico	Ninguno.

c_op_comparacion	
Descripción	Este bloque representa una comparación lógica.
Inputs	• 2 value input de tipo ‘*variables_get_dynamic*’, ‘*c_op_literal*’, ‘*c_op_logica*’ o ‘*c_op_aritmetica*’. • 1 dropdown de opciones, rellenado con las operaciones lógicas binarias disponibles: ‘<’, ‘>’, ‘==’, ‘!=’, ‘<=’ y ‘>=’.
Outputs	Salida de tipo ‘*c_op_logica*’.
Funcionamiento dinámico	Ninguno.

c_op_negacion	
Descripción	Este bloque representa una negación lógica.
Inputs	• 1 value input de tipo '*variables_get_dynamic*', '*c_op_literal*', '*c_op_logica*' o '*c_op_aritmetica*'.
Outputs	Salida de tipo '*c_op_logica*'.
Funcionamiento dinámico	Ninguno.

c_op_bitwise	
Descripción	Este bloque representa una operación bitwise. Este bloque no está siendo usado.
Inputs	• 1 value input de cualquier tipo. • 1 dropdown de opciones, rellenado con las operaciones bitwise disponibles: '&' y '|'.
Outputs	Salida sin tipo.
Funcionamiento dinámico	Ninguno.

7.2.6 c_punteros.js

En este fichero se define todo aquel bloque relacionado con las operaciones de manejo de memoria.

Tablas B-5: Punteros

c_pointer_indirection	
Descripción	Este bloque representa el setter de las indirecciones '&'.
Inputs	• 1 value input de tipo '*variables_get_dynamic*', 1 value input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_pointer_indirection_output	
Descripción	Este bloque representa el getter de las indirecciones '&'.
Inputs	• 1 value input de tipo '*variables_get_dynamic*'.
Outputs	Salida sin tipo.
Funcionamiento dinámico	Ninguno.

c_pointer_address	
Descripción	Este bloque representa el setter de las direcciones '*'.
Inputs	• 1 value input de tipo '*variables_get_dynamic*', 1 value input de cualquier tipo.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_pointer_address_output	
Descripción	Este bloque representa el getter de las direcciones '*'.
Inputs	• 1 value input de tipo '*variables_get_dynamic*'.
Outputs	Salida sin tipo.
Funcionamiento dinámico	Ninguno.

c_pointer_sizeof	
Descripción	Este bloque representa la función sizeof.
Inputs	• 1 dropdown de opciones, rellenado con todos los tipos disponibles.
Outputs	Salida de tipo '*size_t*'.
Funcionamiento dinámico	Ninguno.

c_pointer_cast	
Descripción	Este bloque representa un casting de tipos. *Mejora posible: Implementar un funcionamiento dinámico que cambie el tipo del return al del casting,
Inputs	• 1 dropdown de opciones, rellenado con todos los tipos disponibles. • 1 value input de cualquier tipo.
Outputs	Salida sin tipo. *Mejora posible
Funcionamiento dinámico	Ninguno. *Mejora posible

c_pointer_malloc	
Descripción	Este bloque representa la función malloc.
Inputs	• 1 value input de cualquier tipo.
Outputs	Salida sin tipo.
Funcionamiento dinámico	Ninguno.

c_pointer_type	
Descripción	Este bloque es usado para la creación de punteros y arrays, representando su tipo.
Inputs	• 1 dropdown de opciones, rellenado con todos los tipos disponibles.
Outputs	Salida de tipo '*c_type_constructor*'.
Funcionamiento dinámico	Ninguno.

c_pointer_declaration	
Descripción	Este bloque es usado para la creación de punteros y arrays, representando un puntero.
Inputs	• 1 value input de tipo '*c_type_constructor*'.
Outputs	Salida de tipo '*c_type_constructor*'.
Funcionamiento dinámico	Ninguno.

c_array_declaration	
Descripción	Este bloque es usado para la creación de punteros y arrays, representando un array.
Inputs	• 1 value input de tipo '*c_type_constructor*'. • 1 campo numérico, aceptando números mayores o iguales que 1.
Outputs	Salida de tipo '*c_type_constructor*'.
Funcionamiento dinámico	Ninguno.

c_array_access_get	
Descripción	Este bloque es usado como getter de un array.
Inputs	• 1 dropdown de variables. • 1 campo numérico, aceptando números mayores o iguales que 1.
Outputs	Salida de tipo '*variables_get_dynamic*'.
Funcionamiento dinámico	Ninguno.

c_array_access_set	
Descripción	Este bloque es usado como setter de un array.
Inputs	• 1 dropdown de variables. • 1 campo numérico, aceptando números mayores o iguales que 1. • 1 value input de cualquier tipo
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_pointer_null	
Descripción	Este bloque representa el valor null.
Inputs	Ninguna entrada.
Outputs	Salida sin tipo.
Funcionamiento dinámico	Ninguno.

7.2.7 c_salida.js

En este fichero se define todo aquel bloque relacionado con las operaciones de salida.

Tablas B-6: Salida

c_salida_printf	
Descripción	Este bloque representa la función printf.
Inputs	• 1 campo de texto donde introducir el contenido del printf. • 1 value input de tipo '*variables_get_dynamic*' por cada parámetro indicado en el mutator.
Outputs	Ninguna salida.
Funcionamiento dinámico	Este bloque contiene un mutator, mediante el cual se indica el número de parámetros que tendrá la función.

7.2.8 c_tads.js

En este fichero se define todo aquel bloque relacionado con los TADs.

Tablas B-7: TADs

c_tad	
Descripción	Este bloque representa la definición de un TAD.
Inputs	• 1 campo de texto donde introducir el nombre del TAD. • 1 statement input de cualquier tipo donde introducir los elementos del tad.
Outputs	Ninguna salida.
Funcionamiento dinámico	Ninguno.

c_tad_element	
Descripción	Este bloque representa un campo de la definición de un TAD.
Inputs	• 1 dropdown de opciones rellenado con todos los tipos disponibles. • 1 campo de texto donde introducir el nombre del elemento.
Outputs	Ninguna salida. Sólo colocable dentro de un bloque de tipo '*c_tad*'
Funcionamiento dinámico	Ninguno.

c_tad_call y c_tad_access_call	
Descripción	Estos bloques representan los getters de un elemento de un TAD. Ambos bloques son idénticos, salvo en su representación gráfica, ya que uno accede mediante punto ('.'), y otro mediante flecha ('->').
Inputs	• 1 dropdown de opciones rellenado con todas las variables disponibles de tipo TAD, diferenciando si deben ser puntero ('->') o no ('.'). • 1 dropdown de opciones rellenado con todas las variables disponibles dentro del TAD seleccionado.
Outputs	Salida de tipo '*c_tad_call*' y '*variables_get_dynamic*'.
Funcionamiento dinámico	Este bloque rellena el segundo dropdown dinámicamente según el TAD seleccionado, o mostrando mensajes de error en caso de que algo esté mal diseñado.

c_tad_set y c_tad_access_set	
Descripción	Estos bloques representan los setters de un elemento de un TAD. Ambos bloques son idénticos, salvo en su representación gráfica, ya que uno accede mediante punto (‘.’), y otro mediante flecha (‘->’).
Inputs	• 1 dropdown de opciones rellenado con todas las variables disponibles de tipo TAD, diferenciando si deben ser puntero (‘->’) o no (‘.’). • 1 dropdown de opciones rellenado con todas las variables disponibles dentro del TAD seleccionado. • 1 value input de cualquier tipo.
Outputs	Sin salida.
Funcionamiento dinámico	Este bloque rellena el segundo dropdown dinámicamente según el TAD seleccionado, o mostrando mensajes de error en caso de que algo esté mal diseñado.