Trabajo de inserción profesional

Qsim Web: Simulador de arquitectura Q para el browser

Alumnos

Pablo Pissi
pablopissi@gmail.com

Francisco Perez Ramos
franperez21896@gmail.com

Director

Lic. Federico Martinez
federicoemartinez@gmail.com

Diciembre de 2020

DEPARTAMENTO DE CIENCIA Y TECNOLOGÍA
TECNICATURA UNIVERSITARIA EN
PROGRAMACION INFORMÁTICA

Resumen

La arquitectura Q fue creada en la Universidad Nacional de Quilmes con el propósito de facilitar la enseñanza y el aprendizaje de lenguajes de bajo nivel, como assembler, en la materia Organización de Computadoras. Esta es una arquitectura conceptual, es decir, no existe una máquina física que funcione sobre dicha arquitectura. Al ser Organización de Computadoras una materia inicial, los estudiantes encuentran una dificultad en el aprendizaje, marcado principalmente por la poca abstracción con la que cuentan estos lenguajes, sumando en nuestro caso, la imposibilidad de ejecutar sus programas. En este sentido, QSim, un trabajo de inserción profesional presentado en la misma universidad, aborda el problema brindando un entorno de ejecución para el lenguaje Q. Dicho proyecto fue encarado con una aplicación de escritorio escrita en Java que permitía ver los componentes de la arquitectura Q en funcionamiento. Algunos de los problemas que se encontraron a esta implementación fueron la baja performance de la herramienta y la dificultad que encuentran los estudiantes para configurar el entorno que requiere la aplicación para funcionar.

Es por eso que en este trabajo se busca brindar una solución a este problema: un simulador de la arquitectura Q que permita visualizar de manera didáctica el proceso de ejecución de un lenguaje de bajo nivel pero realizada en un entorno web, lo cual evita las complejidades accidentales encontradas y nombradas anteriormente y agrega portabilidad para diversos dispositivos como celulares o tablets. Además, en contexto de la Pandemia por COVID-19, donde la educación en la Universidad tuvo un viraje a la educación virtual, es fundamental contar con una herramienta que pueda ser accedida mediante Internet. El trabajo brinda una librería escrita en javascript que implementa el lenguaje Q y una interfaz visual que utiliza esa librería para mostrar de manera didáctica el proceso de ejecución. Esta separación permite realizar cambios en la interfaz de usuario sin afectar a la lógica del lenguaje.

Abstract

The Q architecture was created at the Universidad Nacional de Quilmes with the purpose of helping to teach and learn low-level languages, such as Assembly, in the context of the course of study Computer Organization. It's a conceptual architecture, meaning that there is not a physical computer that works with it. Students often face a learning difficulty caused mainly by the lack of abstraction that these languages have, adding in our case the fact that Computer Organization is an initial subject and the impossibility of executing their programs. Given this, QSim, a TIP presented at the same university, addresses the problem by providing an execution environment for the Q language. The project was developed as a Java desktop application, which made it resource hungry and hard to set up.

Therefore this project tries to provide a solution to this problem: a tool that allows to didactically visualize the process of executing a low-level language while running on a web environment, which avoids the accidental complexities found in QSim and that also adds portability for various devices such as cell phones or tablets. Moreover, in the context of a pandemic due to COVID-19, and universities embracing virtual learning, an online tool becomes even more significant. This project provides a library written in javascript that implements the Q language and a visual interface that uses said library to display the execution process in a more didactic way. This separation allows changes to be made to the user interface without altering the language's logic.

Índice

Agradecimientos

- A Federico Martinez por sus grandes aportes conceptuales, por su buena onda y predisposición para ayudar.
- Al equipo docente de Organización de Computadoras por brindarnos su perspectiva y apoyo sobre el diseño de la herramienta.

Agradecimientos de Francisco

- A Pablo, por confiar en mí y ofrecerme trabajar con él en este hermoso proyecto.
- A Fede, por acompañarnos siempre, aguantarnos y compartir de su humor y sabiduría.
- A mis compañeros, amigos y familia, por siempre estar ahí apoyándome y escuchándome.
- A la Universidad Nacional de Quilmes, por el maravilloso clima que siempre genera, haciendo que sin importar quien esté en ella, se sienta cómodo y bienvenido.

Agradecimientos de Pablo

- A Fede por invitarme a colaborar con la materia y darme libertad para crecer en la docencia.
- A Tati por escucharme hablar de mis cursadas como si fuese algo novedoso para ella.
- A la dirección de la carrera, en especial Gabi, que siempre estuvo para darme una mano.
- A mis compañeros y compañeras, que me ayudaron a llegar hasta dónde estoy, muchas veces en sentido literal.
- A mi familia y amigos por escuchar mis explicaciones con más amor que entendimiento.

Capítulo 1

Introducción

La materia Organización de Computadoras es dictada en el ciclo básico de la Tecnicatura Universitaria en Programación Informática de la Universidad Nacional de Quilmes y brinda contenidos que aportan un panorama general sobre arquitectura de computadoras. Nuestras computadoras utilizan una arquitectura llamada Arquitectura de Von Neumann, la cual consta de una Unidad Central de Procesamiento (CPU por sus siglas en inglés) y una memoria donde se almacenarán tanto los datos como los programas. Para interactuar con la arquitectura se utilizan lenguajes de programación de bajo nivel, como puede ser el lenguaje ensamblador. Una de las implementaciones de la Arquitectura de Von Neumann es x86, la arquitectura creada por Intel que sirve como modelo para los procesadores actuales.

Para poder ejecutar un programa este debe ser ensamblado en memoria. El ensamblado consiste en convertir el código de lenguaje ensamblador en código binario. El proceso de ejecución en la arquitectura de Von Neumann consta de 3 etapas: fetch, decode, execute. La etapa fetch se encarga de buscar las instrucciones previamente ensambladas en memoria. Una vez que se encontraron las instrucciones en memoria, estas son decodificadas en la etapa de decode, donde se resuelven los valores de sus operandos, si hubiese alguno. Como paso final, se ejecuta la instrucción en la etapa de execute. Dicha ejecución puede llevar a cabo cambios en el estado de la computadora.

En la materia Organización de Computadoras se creó una arquitectura conceptual llamada Q, utilizando el modelo de la Arquitectura de Von Neumann. Dicha arquitectura se define como conceptual ya que no existe una máquina física que funcione en ella. A su vez, esta arquitectura cuenta con el lenguaje homónimo para su utilización. Está pensada para la enseñanza de arquitecturas de computadoras en un nivel inicial.

Como la materia es dictada en el ciclo básico, muchos/as estudiantes encuentran la complejidad de no poder ver el resultado de sus programas, ya que deben comprender los conceptos para poder simularlos manualmente. En este sentido, QSim, un simulador de la arquitectura Q que fue desarrolado hace algunos años en el contexto de otro trabajo de inserción profesional realiza un aporte significativo, brindando una interfaz visual para la ejecución de programas Q. Sin embargo, dadas las tecnologías y el paradigma de diseño de aplicaciones utilizado, resulta dificultoso para los y las estudiantes su uso actual. Algunos de los inconvenientes encontrados son:

- Quienes quieren usarlo, deben tener instalado un conjunto de aplicaciones, lo cual aporta complejidades accidentales que podrían ser evitadas.
- No puede ser utilizado en dispositivos móviles, ya que las tecnologías utilizadas y la solución propuesta no contemplaban el uso masivo de dispositivos móviles que vemos actualmente.
- Falta de accesibilidad para personas que utilicen lectores de pantalla.
- Tiene problemas de rendimiento, como la baja velocidad.

Es por eso que se decidió realizar una implementación del lenguaje Q utilizando las tecnologías y paradigmas actuales, evitando las dificultades encontradas en las herramientas mencionadas. El

propósito de este trabajo es crear un simulador que permita ejecutar programas en la arquitectura Q. De esta manera las y los estudiantes de la materia tendrán la posibilidad de escribir programas en dicho lenguaje y ver el impacto de los mismos en las partes que componen a la arquitectura, como son registros, memoria, flags y registros especiales. Esta herramienta debe cumplir las siguientes propiedades:

- Poder ejecutar cualquier programa de la arquitectura Q.
- Permitir la visualización de los resultados o los errores de ejecución.
- No requerir instalación.
- Poder usarse desde dispositivos móviles.
- Ser accesible para estudiantes con disminución visual o daltonismo.

A su vez, este trabajo tiene como objetivo la creación de una librería escrita en javascript, que pueda ser utilizada por otras herramientas que busquen un objetivo similar, separando la lógica del lenguaje de su visualización, permitiendo así, mantener actualizadas las herramientas visuales sin tener que actualizar la lógica del lenguaje.

1.1. Resumen del trabajo

Esta trabajo se organiza de la siguiente manera:

- El capítulo 1 es esta introducción, que pretende establecer el concepto de Arquitectura de computadoras de manera básica, la arquitectura Q planteada en la universidad y las motivaciones que llevan al desarrollo de QSim Web, incluyendo un análisis de las herramientas existentes.
- El capítulo 2 explicará más en detalle la estructura de la arquitectura Q, definiendo sus capas y los conceptos que se introducen en cada una de ellas.
- El capítulo 3 realiza una descripción de la totalidad del trabajo, diferenciando la interfaz de usuario de la librería QLib.
- El capítulo 4 cuenta las devoluciones de los y las estudiantes luego de haber utilziado la herramienta durante un cuatrimestre.
- El capítulo 5 describe las conclusiones desarrolladas a partir de la realización del trabajo.

Capítulo 2

Arquitectura Q

En la materia Organización de Computadoras se creó una arquitectura de computadoras teórica llamada Q, utilizando el modelo de la Arquitectura de Von Neumann. A su vez, esta arquitectura

cuenta con el lenguaje homónimo para su utilización. Está pensada para la enseñanza de conceptos de organización y arquitecturas de computadoras en un nivel inicial.

Su diseño está dividido en 5 capas llamadas Q1, Q2, Q3, Q4, Q5. Cada capa contiene a las instrucciones y modos de direccionamiento de la capa anterior, agregando nuevas funciones a la misma. Esto permite que un programa escrito en Q1 pueda ser ejecutado en Q5, obteniendo el mismo resultado.

2.1. Q1

El objetivo de esta capa es introducir algunos conceptos básicos de la programación de lenguajes de bajo nivel, como son las instrucciones, registros e inmediatos.

Sus caracteristicas son: 8 registros de uso general, llamados desde R0 a R7; un modo de direccionamiento para trabajar con constantes llamado Inmediato y 5 instrucciones de dos operandos cada una, MOV, ADD, SUB, DIV, MUL.

Los registros tienen 16 bits de almacenamiento. Los inmediatos tienen 16 bits de longitud y se escriben en hexadecimal con la sintaxis: 0xAAAA.

Las instrucciones de esta capa son:

Operación	**Cod Op**	**Efecto**	**Ejemplo de uso**
ADD	0010	destino $\leftarrow$ $destino + origen$	ADD R7, 0x23FA
DIV	0111	destino $\leftarrow$ $destino \% origen$	DIV R4, 0x23FA
MOV	0001	destino $\leftarrow$ $origen$	MOV R4, R2
MUL	0000	destino $\leftarrow$ $destino \times origen$	MUL R5, 0x0ABC
SUB	0011	destino $\leftarrow$ $destino - origen$	SUB R2, [0x0ABC]

Todas estas instrucciones comparten el siguiente formato:

Cod Op	**Modo Destino**	**Modo Origen**	**Destino**	**Origen**
4 bits	6 bits	6 bits	16 bits	16 bits

Los modos de direccionamiento son:

Modo	**Codificación**
Inmediato	000000
Registro	100rrr [1]

Un ejemplo de instrucción invalida es: ADD 0x23FA, R7 porque una constante no puede almacenar el resultado de una operación.

[1] La codificación de un registro varía según el número del mismo. R2 se codifica 100010 ya que 2 se representa como 010 en BSS(3).

Un ejemplo de ensamblado de la instrucción ADD R6, 0x2323 en la celda 0x00F0

Celda	Valor
...	...
0x00F0	0010 100110 000000
0x00F1	0010 0011 0010 0011
...	...

2.2. Q2

El objetivo de esta capa es brindar la posibilidad de interactuar con la memoria, algo que se omite en Q1 por simplicidad.

Para ello, se agrega un modo de direccionamiento llamado Directo el cual especifica la dirección de memoria donde se encuentra el valor del operando. La dirección del operando se escribe entre corchetes: [0x23AB].

Los modos de direccionamiento son:

Modo	Codificación
Inmediato	000000
Registro	100rrr
Directo	**001000**

Un ejemplo de ensamblado de la instrucción ADD R6, [0x2323] en la celda 0x00F0

Celda	Valor
...	...
0x00F0	0010 100110 001000
0x00F1	0010 0011 0010 0011
...	...

2.3. Q3

El objetivo de esta capa es brindar la posibilidad de escribir rutinas, que permiten explicar los conceptos de reutilización, modularización y documentación.

En esta capa toman relevancia los registros PC, SP, IR, MAR, MBR y los conceptos de pila y etiqueta.

- El PC es un registro especial que sirve para indicar a la CPU cuál es la siguiente instrucción a ejecutar. Al leer una instrucción su valor se incrementa en 1.

- La pila es una sección reservada de la memoria principal para almacenar las direcciones de retorno de las rutinas invocadas.
- El SP es un registro especial que funciona como puntero a la pila. Es utilizado por las instrucciones CALL y RET descriptas a continuación.
- El IR es un registro especial que contiene la instrucción que está siendo ejecutada (o parte de ella). Tiene 48 bits de almacenamiento.
- El MAR es un registro especial que contiene la dirección de memoria que se está leyendo o escribiendo.
- El MBR es un registro especial que contiene el valor de la dirección de memoria que indica el MAR.
- Las etiquetas son cadenas de texto que se utilizan para abstraer el valor de una constante. Se pueden utilizar como operando de la instrucción CALL. Sus nombres no pueden repetirse ya que funcionan como identificadores.

Además de las caracteristicas de las capas anteriores, en esta capa se agregan dos instrucciones: CALL y RET que permiten interactuar con rutinas.

CALL: Invoca a una rutina, para ello guarda el PC en la pila, decrementa el SP en 1 y copia el PC a la dirección de memoria donde está ensamblada la primera instrucción de dicha rutina.

Su formato de instrucción es:

Cod Op	**Relleno**	**Modo Origen**	**Origen**
4 bits	6 bits	6 bits	16 bits

Un ejemplo de uso es:

```
CALL calcularPromedio
```

RET: Finaliza la ejecución de una rutina, para ello incrementa el SP y pone en el PC el valor de la pila.

Su formato de instrucción es:

Cod Op	**Relleno**
4 bits	12 bits

Su uso es:

```
RET
```

Las instrucciones de esta capa son:

Operación	Cod Op	Efecto	Ejemplo de uso
ADD	0010	destino $\leftarrow destino + origen$	ADD R7, 0x23FA
DIV	0111	destino $\leftarrow destino \% origen$	DIV R4, 0x23FA
MOV	0001	destino $\leftarrow origen$	MOV R4, R2
MUL	0000	destino $\leftarrow destino \times origen$	MUL R5, 0x0ABC
SUB	0011	destino $\leftarrow destino - origen$	SUB R2, [0x0ABC]
CALL	1011	$[SP] \leftarrow PC; SP \leftarrow SP - 1; PC \leftarrow origen$	CALL unaRutina
RET	1100	$SP \leftarrow SP + 1; PC \leftarrow [SP]$	RET

2.4. Q4

El objetivo de esta capa es introducir los conceptos de ejecución condicional y repeticiones. Para ello se introducen los conceptos de flags y saltos. Los flags son registros de 1 bit que se calculan al ejecutarse instrucciones aritméticas.

- Z: es 1 cuando el resultado de la operación es 0.
- N: es 1 cuando el resultado de la operación es negativo.
- C: es 1 cuando el resultado de la operación tiene carry o borrow en el bit más significativo.
- V: es 1 cuando el resultado de la operación tiene overflow en complemento a 2.

Las caracteristicas de esta capa son, además de las mencionadas anteriormente, saltos condicionales relativos y saltos incondicionales absolutos.

La instrucción salto incondicional absoluto es JMP, que copia el PC a la dirección de memoria indicada por su etiqueta.

Su formato de instrucción es:

Cod Op	Relleno	Modo Origen	Origen
4 bits	6 bits	6 bits	16 bits

Un ejemplo de uso es:

```
JMP esNumeroPositivo
```

Las instrucciones de saltos condicionales relativos son las nombradas a continuación, cuya ejecución tiene como efecto incrementar o decrementar el PC dado un desplazamiento, si su condición de salto se cumple.

Operación	Cod Op	Descripción	Condición de salto	Ejemplo de uso
JE	0001	Igual / Cero	Z	JE etiqueta
JNE	1001	No igual	not Z	JNE etiqueta
JLE	0010	Menor o igual	Z or (N xor V)	JLE etiqueta
JG	1010	Mayor	not (Z or (N xor V))	JG etiqueta
JL	0011	Menor	N xor V	JL etiqueta
JGE	1011	Mayor o igual	not (N xor V)	JGE etiqueta
JLEU	0100	Menor o igual sin signo	C or Z	JLEU etiqueta
JGU	1100	Mayor sin signo	not (C or Z)	JGU etiqueta
JCS	0101	Carry / Menor sin signo	C	JCS etiqueta
JNEG	0110	Negativo	N	JNEG etiqueta
JVS	0111	Overflow	V	JVS etiqueta

Su formato de instrucción es:

Prefijo	Cod Op	Desplazamiento
4 bits	4 bits	8 bits

2.5. Q5

En esta última capa se definen dos modos de direccionamiento indirectos, que permiten el trabajo con recorridos, arreglos e iteraciones. Si bien es técnicamente posible realizar lo mencionado sin estos modos, agregaría una complejidad no buscada en la materia. Estos modos de direccionamiento son Indirecto e Indirecto Registro.

El modo de direccionamiento Indirecto especifica la dirección de memoria donde se encuentra la dirección de memoria que contiene el valor del operando. La dirección del operando se escribe entre doble corchetes: [[0x23AB]].

El modo de direccionamiento Indirecto Registro especifica el registro donde se encuentra la dirección de memoria que contiene el valor del operando. El registro se escribe entre corchetes: [R5].

Los modos de direccionamiento y sus codificiones son:

Modo	Codificación
Inmediato	000000
Registro	100rrr
Directo	001000
Indirecto	**011000**
Indirecto Registro	**110rrr**

Un ejemplo de ensamblado de la instrucción ADD [R6], [[0x2323]] en la celda 0x00F0

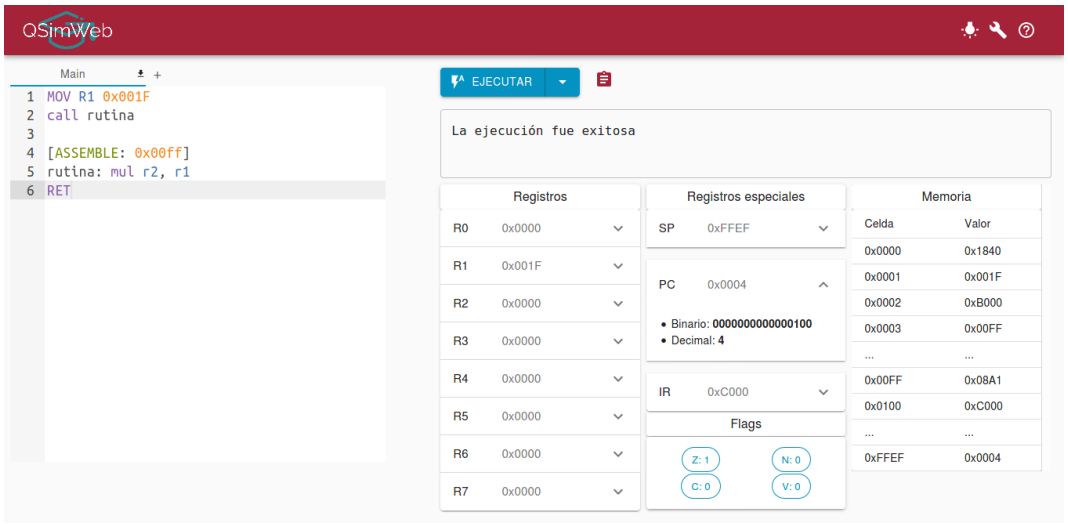
Celda	**Valor**
...	...
0x00F0	0010 110110 011000
0x00F1	0010 0011 0010 0011
...	...

Capítulo 3

Qsim web

La entrega de este trabajo cuenta con una librería que implementa la especificación de la arquitectura Q y una interfaz de usuario que permite hacer uso de la librería de una manera más didáctica.

La construcción de la librería fue realizada utilizando javascript. El parser utiliza la librería nearley. El testing unitario y de integración fue realizado con mocha y cypress. La construcción de la interfaz de usuario fue realizada en React con la librería Material-UI y react-ace para el editor. Esta implementación esta subida a un servidor utilizando Heroku, en el siguiente link: https://qweb-unq.herokuapp.com/. Su código fuente está subido a Gitlab y puede verse en el siguiente link: https://gitlab.com/qsim-web-project/qsim-web.



3.1. Proceso de ejecución

Una de las necesidades principales es poder ejecutar código Q. La ejecución de un programa se compone de los siguientes pasos:

Se escribe un código Q en el editor de texto y se presiona ejecutar. Luego se analiza sintacticamente el texto que representa al programa, esta tarea es realizada por el Parser. Si ocurre un error se mostrará en pantalla informando lo ocurrido. Si el parseo pudo realizarse correctamente, se procede a traducir el resultado del parseo a instrucciones de la librería, esta tarea es realizada por el Translator. Luego se procede a ensamblar y ejecutar el programa, ambas tareas son realizadas por Computer. Si ocurre un error se mostrará en pantalla informando lo ocurrido. Si el programa finaliza, se mostrarán los resultados en pantalla.

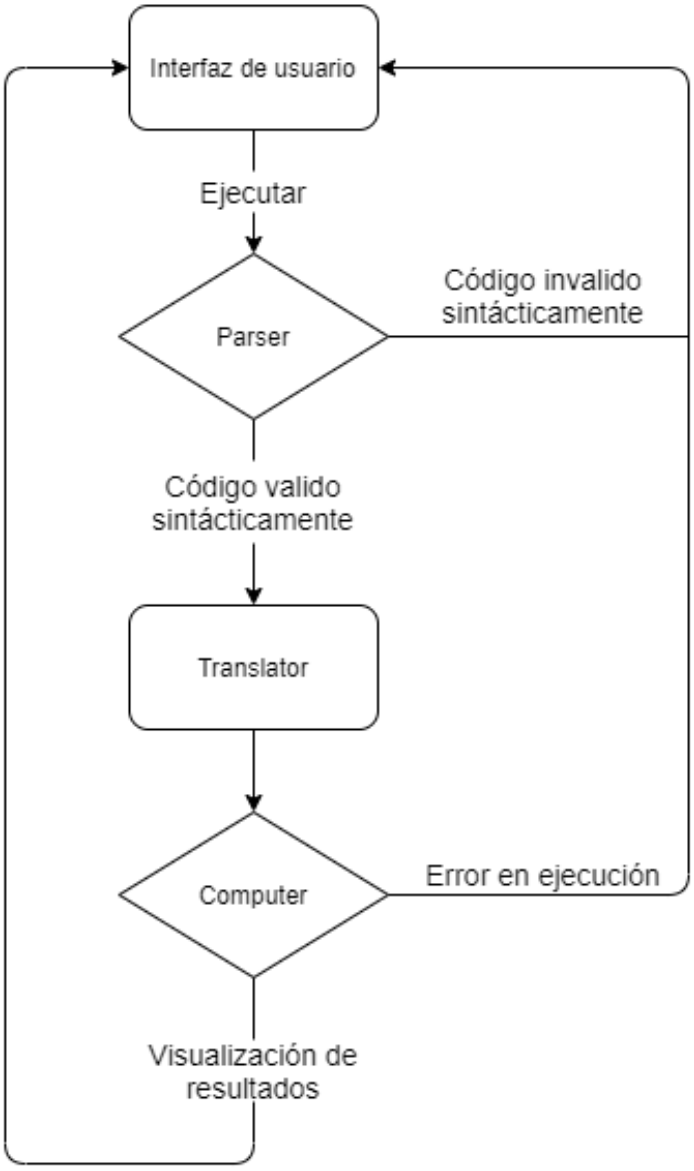


Figura 3.1: Proceso de ejecución separado por componentes

Los detalles relacionados con la interfaz de usuario serán explicados en la sección que sigue a continuación.

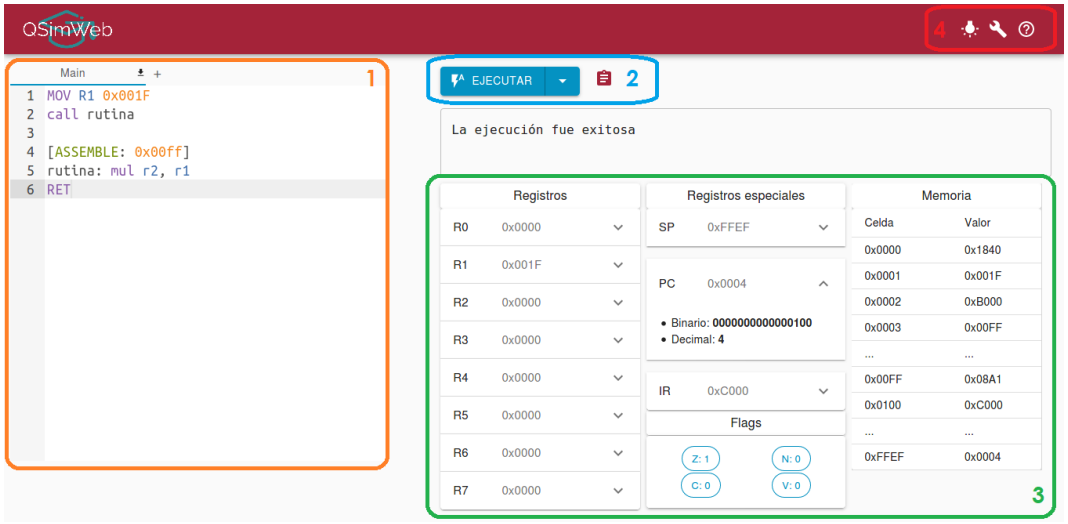
Los detalles relacionados con la librería serán explicados en la sección 3.3

3.2. Interfaz de usuario

La interfaz cuenta con una pantalla inicial donde se pueden escribir y ejecutar programas Q, permitiendo visualizar los resultados de la ejecución. Además cuenta con una sección que permite configurar algunos aspectos de uso de la misma.

Se encarga de coordinar los pasos requeridos para ejecutar un programa utilizando la librería, es decir, parseo, traducción, ensamblado y ejecución del código.

Las secciones nombradas anteriormente se describen a continuación:



1. Editor

Los programas se escriben sobre un editor de texto que cuenta con resaltado de sintaxis y errores. Estos programas pueden ser escritos tanto en mayúsculas como en minúsculas. Cuando las instrucciones tienen más de un operando, estos pueden ser separados mediante uno o mas espacios o una coma. El editor fue desarrollado utilizando la librería react-ace.

Figura 3.2: Editor de texto con resaltado de sintaxis.

Se permiten crear rutinas, ensamblandolas en celdas específicas utilizando [assemble: X], donde X es un inmediato representando la celda de memoria donde se ensamblará la rutina.

```
1 MOV R1 0x0002
2 CALL sumar1
3
4 [ASSEMBLE: 0x00F0]
5 sumar1: ADD R1, 0x0001
6 RET
```

La meta instrucción ASSEMBLE permite indicar que la rutina sumar1 se ensamblará a partir de 0x00F0.

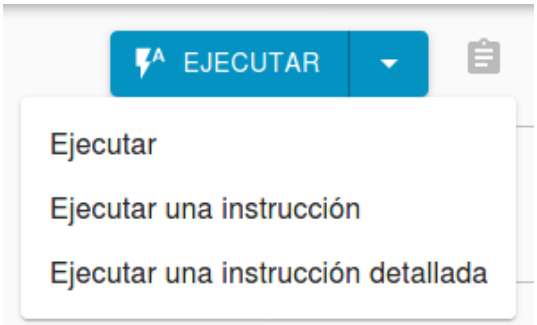
La herramienta además permite la realización de comentarios que no afectarán la ejecución. Un comentario comienza con # y termina al final de la línea.

```
1 MOV R1 0x0002
2 CALL sumar1
3
4 #Assemble sirve para definir dónde se
5 #ensambla la rutina.
6 [ASSEMBLE: 0x00F0]
7 sumar1: ADD R1, 0x0001 #Sumo uno a R1
8 RET
```

Cuenta también con la posibilidad de dividir el código en distintas pestañas, importación de archivos .txt y descarga de los programas.

2. Botón de ejecución

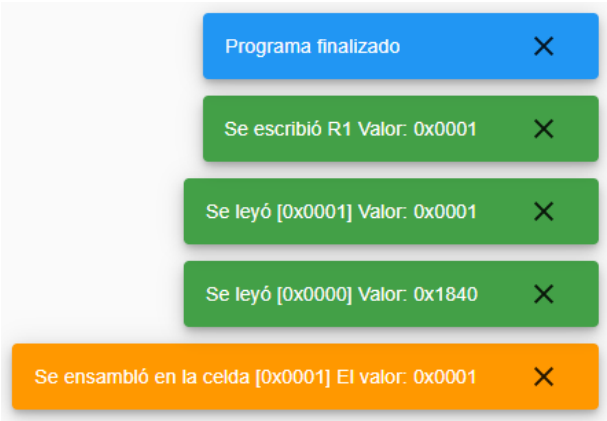
Cuando se quiere probar el código escrito, se debe ejecutar, haciendo click en el botón de ejecución. La interfaz da la opción de ejecutar un programa de 3 modos distintos: Ejecución completa, Ejecución por instrucción y Ejecución por instrucción detallada.



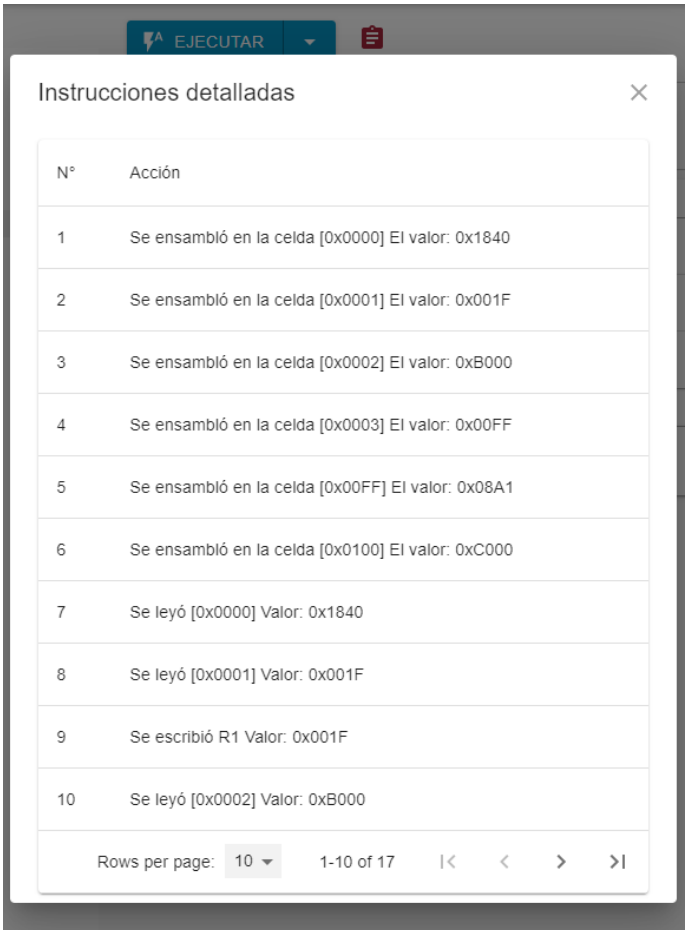
El modo Ejecución completa realiza la ejecución de la totalidad de las instrucciones del programa mostrando, al final, los cambios en la sección de resultados.

Ejecución por instrucción realiza la ejecución de una instrucción y muestra los cambios en la sección de resultados, al hacer click nuevamente, ejecutará la siguiente instrucción si existe. Si no existe se comenzará a ejecutar el programa nuevamente.

Para el modo de Ejecución por instrucción detallada, mostrará mensajes con las acciones producidas en cada paso de la ejecución.



En todos los modos de ejecución, dichas acciones quedan registradas y disponibles para su revisión:



3. Resultados

Una vez finalizada la ejecucion, con o sin errores, se actualizará la sección de resultados.

Los resultados se componen de los valores de registros, registros especiales, flags y memoria. Los valores de registros y registros especiales pueden ser visualizados en hexadecimal, binario y decimal.

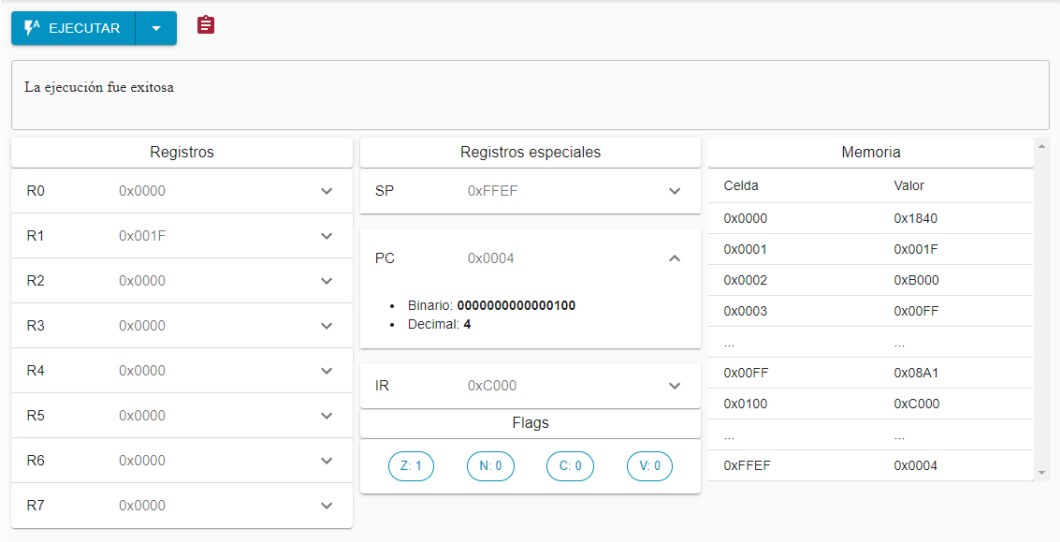


Figura 3.3: Resultados de la ejecución, separados en registros, registros especiales y memoria.

Cuando la memoria contiene celdas inicializadas que no son contiguas se resumen las celdas intermedias colocando "...".

Memoria	
Celda	Valor
0x0000	0x1840
0x0001	0x0002
0x0002	0xB000
0x0003	0x00F0
...	...
0x00F0	0x2840
0x00F1	0x0001
0x00F2	0xC000
...	...
0xFFEF	0x0004

Si durante la ejecución se produjo un error, este será mostrado en pantalla.

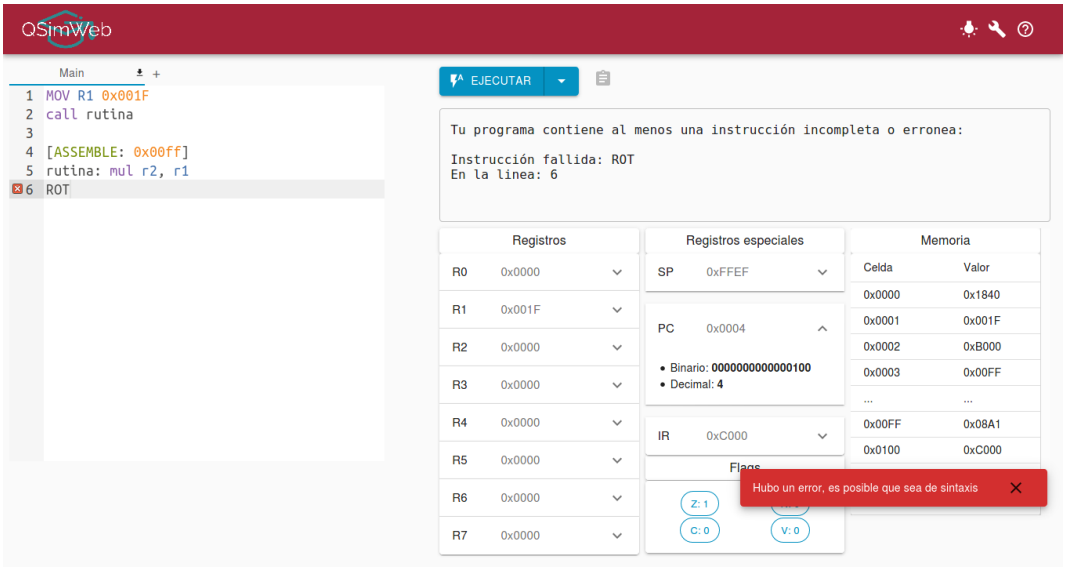


Figura 3.4: Error de sintaxis ocurrido al presionar ejecutar.

4. Configuraciones y ayuda

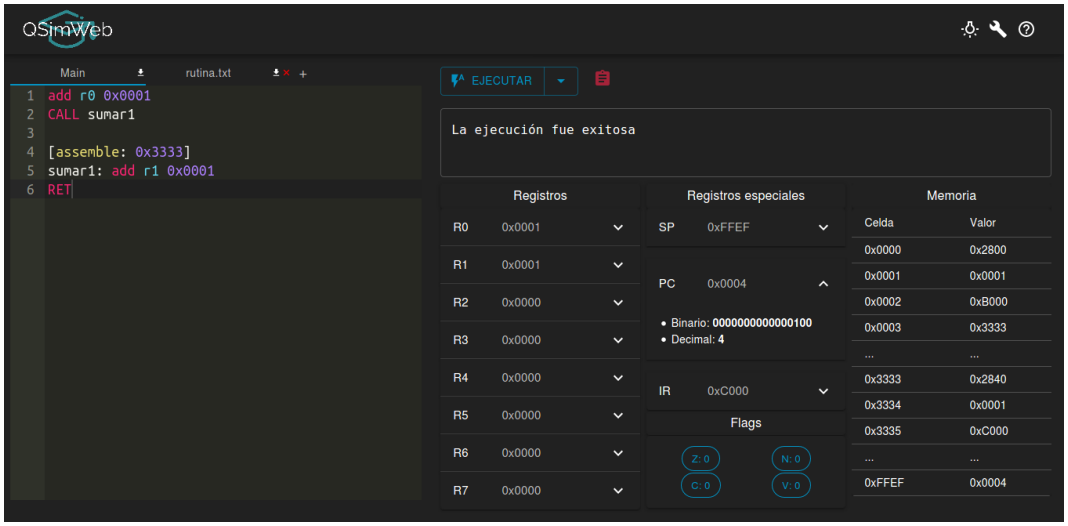
La ventana principal cuenta además con un menú que permite configurar parametros de la ejecución.

En la configuración se permite cambiar la cantidad de registros, el comportamiento de MUL respecto al registro R7, los valores por defecto de las celdas de memoria y registros, los modos de direccionamiento e instrucciones habilitadas. Es decir, todas las configuraciones que se especificarán en QConfig (ver 3.3.3). Además permite cargar un archivo de configuración para simplificar el armado de la misma.

Estas prefencias mencionadas se guardan en el navegador del usuario, permitiendo que no se pierdan al finalizar su uso. Además se permite importar un archivo de configuración, donde se pueden especificar las configuraciones nombradas anteriormente.

Se permite además ver un manual de uso.

Existen los modos oscuro y claro, que facilitan el uso por parte de personas con daltonismo.



3.2.1. Accesibilidad

Una de las funcionalidades esperadas era brindar la posibilidad de la herramienta de ser usada con lectores de pantalla. En este sentido se hicieron evaluaciones de accesibilidad utilizando herramientas estandarizadas y una prueba de campo con un estudiante no vidente de la tecnicatura. Dichas evaluaciones permitieron hacer un diagnóstico de accesibilidad y posterior corrección de las fallas encontradas. Actualmente la herramienta contiene código para facilitar su uso, permitiendo que se pueda acceder a todas las funcionalidades de la misma.

Las mejoras realizadas hasta el momento fueron:

- Nombrar correctamente el lenguaje de la aplicación ya que react pone por defecto el lenguaje en inglés.
- Agregar etiquetas a todos los botones.
- Agregar encabezados a la sección de resultados, para permitir su navegación y lectura.
- Corrección de autofoco en los modales.
- El contenido de las ventanas emergentes se lee automáticamente por el lector de pantalla.

3.3. Librería

La librería creada por este trabajo recibe el nombre de QLib. El objetivo de la creación de la librería es brindar una base de código que permita la creación o modificación de herramientas para la enseñanza del lenguaje en el ecosistema de javascript, brindando así la posibilidad de ser utilizada en navegadores web. Además la existencia de una librería así, permite la creación de interfaces de usuario que extiendan a la entregada en este trabajo.

La misma esta compuesta por tres capas principales: Parser, Translator y Computer. Las funcionalidades que ofrece son:

- Implementación del lenguaje Q: todas las instrucciones de Q, modos de direccionamiento, registros, flags, PC, SP, IR, MAR, MBR, etiquetas, pila están implementadas.
- Manejo de etiquetas: se proveen mecanismos para definir etiquetas, asociarlas a una dirección y utilizarlas en el ensamblado de programas.
- Manejo del ensamblado: métodos para pasar de un programa Q a la representación binaria de las instrucciones y viceversa.
- Ejecución completa, paso a paso y paso a paso detallada.
- Parseo del código Q.
- Capa de traducción entre el parseo y las instrucciones de la librería (ver 3.3.2).
- Opciones de configuración de la librería: modificar la cantidad de registros, valores por defecto de celdas y registros, modos de direccionamiento e instrucciones habilitadas en la ejecución.
- Exposición de valores del estado a lo largo de la ejecución.
- Exposición de acciones realizadas a lo largo de la ejecución.

Está diseñada utilizando el paradigma de programación orientada a objetos el cual define la existencia de clases y sus interacciones.

3.3.1. Parser

El parser es la unidad encargada del analisis sintáctico para determinar si el programa es correcto de acuerdo a la gramática del lenguaje Q. Recibe como entrada una cadena de texto y devolverá objetos javascript en caso de ser la entrada sintacticamente válida o un error en caso contrario. Dicha salida será la entrada del Translator (ver 3.3.2).

Para la definición de la gramática y su análsis sintáctico se utilizó la librería nearley.js, la cual provee una sintaxis para la definición de gramáticas basada en *Extended Backus-Naur Form*. Los detalles de EBNF escapan al desarrollo de este trabajo por lo que es suficiente con pensar que este funciona como un metalenguaje, es decir, un lenguaje que permite la construcción de otros lenguajes.

Para utilizar nearley.js se genera un archivo de gramática con la extensión .ne, donde se define la idea de instrucción, operando, tipos de instrucción y etiquetas. Luego, dicho archivo se compila a javascript utilizando la librería y esto permite el uso de la gramática en navegadores.

Una vez que la gramática está compilada, se utilizará la clase Parser provista por la librería para ingresar las instrucciones en forma de texto. Si son sintacticamente correctas, se devolverá un objeto preparado para traducirse a las instrucciones del lenguaje Q.

Este diseño que separa la etapa de parseo de la etapa de traducción a las clases del lenguaje permite la utilización de un parser distinto, siempre y cuando su salida respete el formato esperado por el Translator.

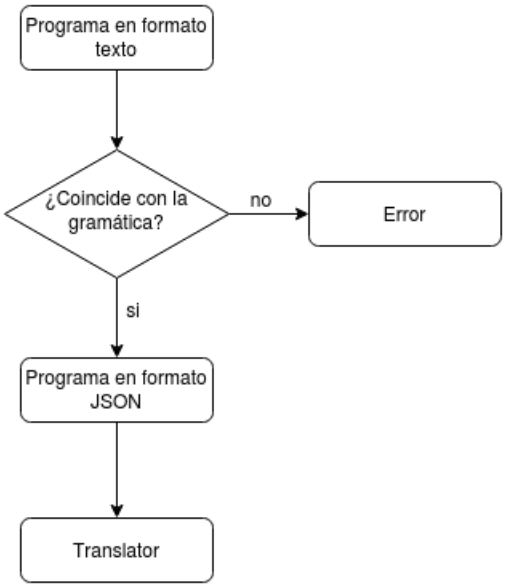


Figura 3.5: Proceso de parseo del código.

3.3.2. Translator

Como uno de los objetivos de QLib es permitir su utilización con distintos parsers, se provee una capa que actúe de interfaz entre los resultados brindados por el parser y las rutinas que utilizará la clase Computer para ejecutar el programa.

En él se definen un conjunto de reglas que determinan cómo mapear las posibles entradas de formato JSON a instrucciones soportadas por QLib. Estas reglas se dividen en 3 grupos: *reglas para instrucciones*, para *tipos de instrucciones* y *para operandos*.

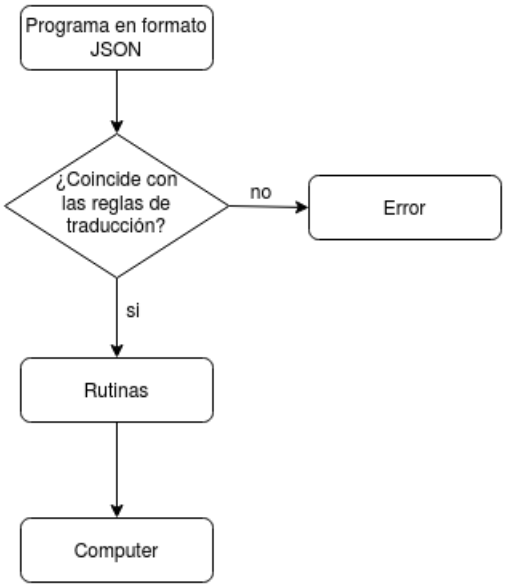


Figura 3.6: Proceso de traducción del código.

Cuando se quiere traducir un programa, se realiza el siguiente proceso línea por línea:

En base a la línea que se está traduciendo, se busca una instrucción que coincida con el nombre de la misma en las *reglas para instrucciones*. Una vez encontrada la instrucción, se busca el tipo de la instrucción en las *reglas de tipos*. El tipo de la instrucción sirve para determinar cómo se traducirán los operandos. Luego se obtienen los operandos de acuerdo al tipo de instrucción y el nombre del operando en las *reglas para operandos*, obteniendo así, una instancia de operando de QLib. Con estas búsquedas realizadas se puede instanciar la instrucción de QLib y así armar las rutinas.

3.3.3. Computer

Esta capa tiene como funcionalidades principales el manejo del ensamblado y la ejecución.

Ensamblado

El ensamblado permite, utilizando las rutinas que genera el translator, obtener el código binario de las instrucciones para luego poder ejecutarlo. En el proceso de ensamblado intervienen principalmente 4 clases: Computer, Instruction, State y Memory. En esta etapa, la clase Computer se encarga de recibir las rutinas a ensamblar y delega en las instrucciones de cada rutina la obtención del código binario para entregarlo a la clase State. State se encarga de guardar en memoria el código binario de cada una de las instrucciones utilizando la clase Memory.

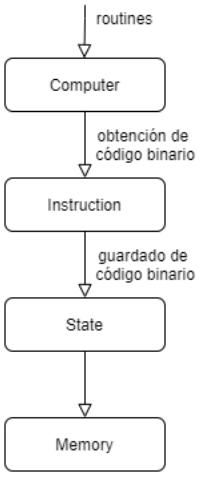


Figura 3.7: Proceso de ensamblado.

Como las instrucciones de una rutina pueden tener etiquetas o hacer uso de las mismas como operando, se definieron las clases Label y LabelReference para su manejo. Una rutina entonces, puede contener en su lista de instrucciones tanto elementos de tipo Instruction como de tipo Label. El objetivo de la etapa de ensamblado es convertir las etiquetas en direcciones de memoria ya que las etiquetas son texto y este no puede ser ensamblado en memoria. Para esto, el primer paso es calcular las direcciones de memoria donde se encuentren las definiciones de las etiquetas, es decir instancias de la clase Label. Luego, se deberán reemplazar los usos de esas etiquetas por operandos de tipo inmediato.

Para calcular las direcciones de memoria de las etiquetas se ven involucradas las clases Instruction y Label ya que son los elementos que puede tener una rutina y el PC, que permite conocer la dirección de memoria actual. La clase Instruction, al solo representar instrucciones y no etiquetas, tiene la responsabilidad de avanzar el PC. Label en cambio, tiene la responsabilidad de guardar el PC actual y su identificador en una lista de etiquetas y delegar en su instrucción el avance del PC.

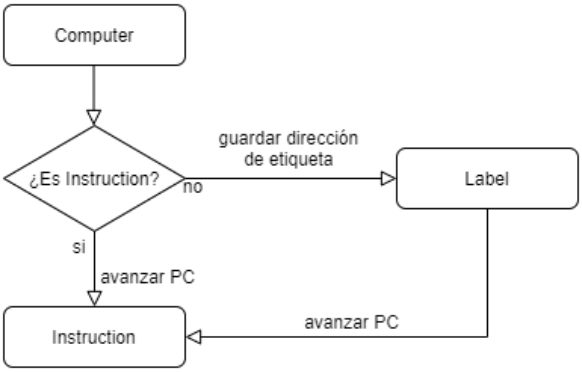


Figura 3.8: Cálculo de etiquetas.

Ejecución

La etapa de ejecución se encarga de decodificar el código previamente ensamblado, traducirlo a instrucciones Q y ejecutarlas en orden. De esta manera se da la noción de ejecución de un programa o rutina. Las instrucciones son decodificadas y ejecutadas de a una a la vez y esto permite que el programa o rutina pueda mutar a lo largo de su ejecución.

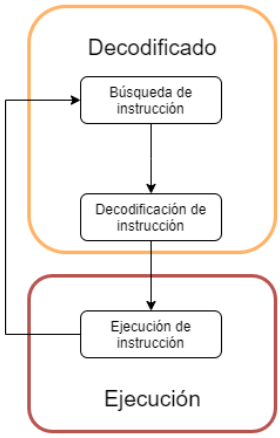


Figura 3.9: Diagrama de ejecución.

Para ello, Computer lee la dirección de memoria indicada por el PC, cuyo contenido será decodificado. El decodificado del código máquina queda delegado en Instruction, ya que cada instrucción puede determinar si un código binario le corresponde y sabe cómo decodificar sus operandos.

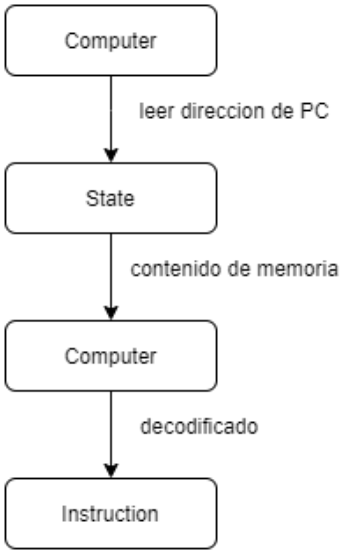


Figura 3.10: Proceso de decodificado.

Una vez que la instrucción fue decodificada junto con sus operandos, se procede a ejecutarla. La ejecución de una instrucción recibe el estado actual y lo modifica acorde a su efecto. Cada instrucción fue modelada con una clase, la cual conoce su efecto y lo aplica al ejecutarse. Este enfoque nos permite agregar nuevas instrucciones facilmente siguiendo la interfaz brindada.

Estas modificaciones pueden incluir cambios en los flags, en el operando destino, en el PC o en la pila según la instrucción que se esté ejecutando.

Durante esta estapa pueden darse lecturas o escrituras de los operandos de origen y/o destino. Estos operandos pueden estar en modo inmediato, registro, directo, indirecto o registro indirecto. Para conocer o modificar sus valores, los operandos dependen de los valores almacenados en el estado, quien guarda los valores de registros y conoce a la memoria. Es por ello que se definieron 5 clases para representar cada una de estas posibilidades, las cuales conocen la forma de obtener valores del estado y guardarlos en él.

Como algunas instrucciones modifican los flags mientras que otras no lo hacen, esta tarea es delegada a cada instrucción, la cual conoce qué flags modifica y cómo lo hace.

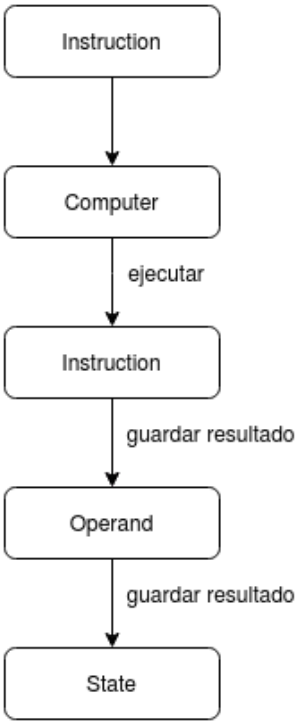


Figura 3.11: Proceso de ejecución.

La ejecución desde una mirada educativa

En la definición de Q no existe la diferenciación de programa y rutina, como así tampoco la idea de finalización de ejecución. La computadora está constantemente ejecucutando alguna instrucción, ya sea una instrucción con efecto o sin efecto.

Dado que QLib es una librería que simula una arquitectura de computadoras enfocada en la enseñanza, permite la creación de programas y rutinas. Estos se diferencian entre sí, porque los programas no finalizan con una instrucción RET mientras que las rutinas si.

En la arquitectura Q no existe la idea de finalización de ejecución. Esto puede ser confuso para los/las estudiantes, ya que se esperaría que el programa tenga un inicio y un resultado final. Es por eso que consideramos que la librería debía proveer un método para identificar que la ejecución finalizó.

Para ello, analizamos las siguientes condiciones:

Al iniciar una ejecución las celdas que no hayan sido escritas, no tendrán valor. Por lo tanto el PC eventualmente llegará a una celda no incializada, esto puede darse por dos razones:

- Un salto o una llamada a rutina en una celda vacía.

- Incremento orgánico del PC. Se considera que el PC se incrementó organicamente cuando su incremento no es causado por un offset o por una asignación directa.

Analizando dichas razones:

- Si se llega a una celda no inicializada mediante un salto o una llamada a rutina, asumimos que es un error de programación. Por ejemplo:

```
MOV R1 R2
JMP 0x0fff
```

En este caso se salta a la celda 0x0fff que no fue previamente inicializada.

```
MOV R3 R0
CALL 0xAFFF
```

En este caso se llama a la rutina ensamblada en la celda 0xAFFF que no fue previamente inicializada.

- Si se llega a una celda no inicializada mediante un incremento orgánico del PC pueden darse dos casos:

  La pila no está vacía, por lo tanto asumimos un error de programación. Posiblemente por el olvido de un RET.

```
MOV R3 R0
CALL rutina
MOV R0 R1

[ASSEMBLE: 0x00ff]
rutina: MOV R2 R3
```

  La pila está vacía, puede haber dos razones:

  La etapa de ejecución no es *FETCH* [2].

```
MOV R3 [R0]
```

En este caso, se lee el indirecto registro R0, pero la celda a la que apunta R0 no fue inicializada. Si bien esto no es un error, el resultado puede ser inesperado.

  La etapa de ejecución es FETCH, por lo tanto el programa finalizó.

```
MOV R3 R0
CALL rutina
MOV R0 R1

[ASSEMBLE: 0x00ff]
rutina: MOV R2 R3
RET
```

[2] FETCH es la etapa de la ejecución donde se busca en memoria la próxima instrucción a ejecutar.

Concluyendo, se asume la finalización de la ejecución cuando:

- La celda leída está no inicializada.
- El PC no fue modificado mediante un salto o una llamada a rutina.
- La pila se encuentra vacía.
- La etapa de ejecución es FETCH.

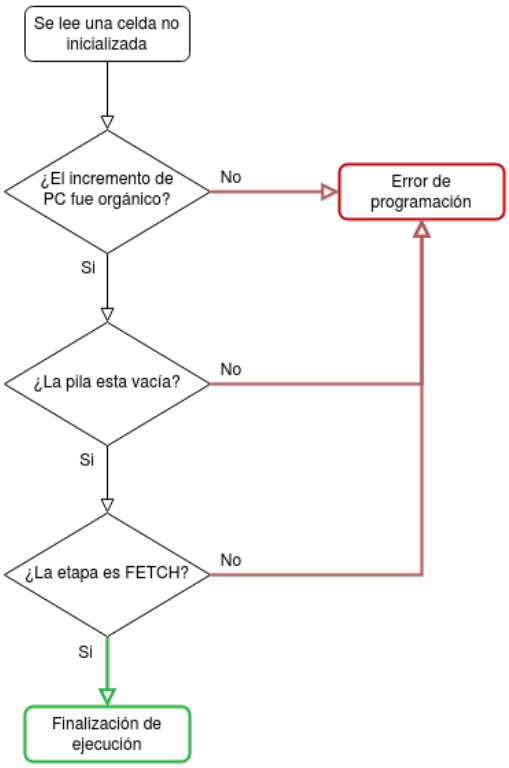


Figura 3.12: Condiciones de finalización.

Antes de definir una condición de finalización, tuvimos en cuenta la posibilidad de agregar una instrucción de fin, donde por ejemplo, el usuario podría hacer uso de esta para definir cuándo o dónde el programa terminaría su ejecución.

Ejemplo de programa con instrucción de fin:

```
MOV R3 R0
ADD R0 R1
FIN
```

Con esta instrucción ya no sería necesario identificar una condición de finalización.

Optamos por definir una condición de finalización ya que esta facilita el entendimiento de un alumno o alumna de cuál es el resultado del programa que esté ejecutando abstrayendo la idea de que la computadora nunca deja de ejecutar. Por otro lado se evita agregar una instrucción que no se encuentra en Q.

QConfig

Existen distintos aspectos en QSim Web que pueden ser configurados con el objetivo de lograr que la herramienta tenga un enfoque más educativo y a su vez, para poder crear subconjuntos del lenguaje. QConfig es la clase encargada de guardar y modificar la configuración elegida. Existen las siguientes configuraciones:

- Cantidad de registros: Permite modificar la cantidad de registros entre 1 y 8 (R0-R7). Si se intenta usar un registro deshabilitado, la herramienta arrojará un error especificando lo sucedido.
- Comportamiento de la instrucción MUL: Permite elegir si se quiere que los 16 bits más significativos de una multiplicación se guarden en R7 o no. Esto es interesante ya que al principio resulta difícil de entender por los alumnos y que esté habilitado agregaría una complejidad esencial no buscada en los comienzos de la materia.
- Valor por defecto en celdas: Permite elegir si el valor por defecto de una celda será cero; un error, si se intenta acceder al valor de una celda no inicializada la herramienta lanzará una excepción; aleatorio, todas las celdas de memoria comenzarán con un valor aleatorio, ayudando así a simular que la memoria siempre puede contener valores inesperados.
- Valor por defecto en registros: Permite elegir si el valor por defecto de un registro será cero o aleatorio.
- Modos de direccionamiento: Permite elegir qué modos de direccionamiento estarán habilitados en la ejecución. Si se intenta usar un modo de direccionamiento deshabilitado, la herramienta lanzará una excepción indicando lo sucedido.
- Instrucciones: Permite elegir qué instrucciones estarán habilitadas en la ejecución. Si se intenta usar una instrucción deshabilitada, la herramienta lanzará una excepción indicando lo sucedido.

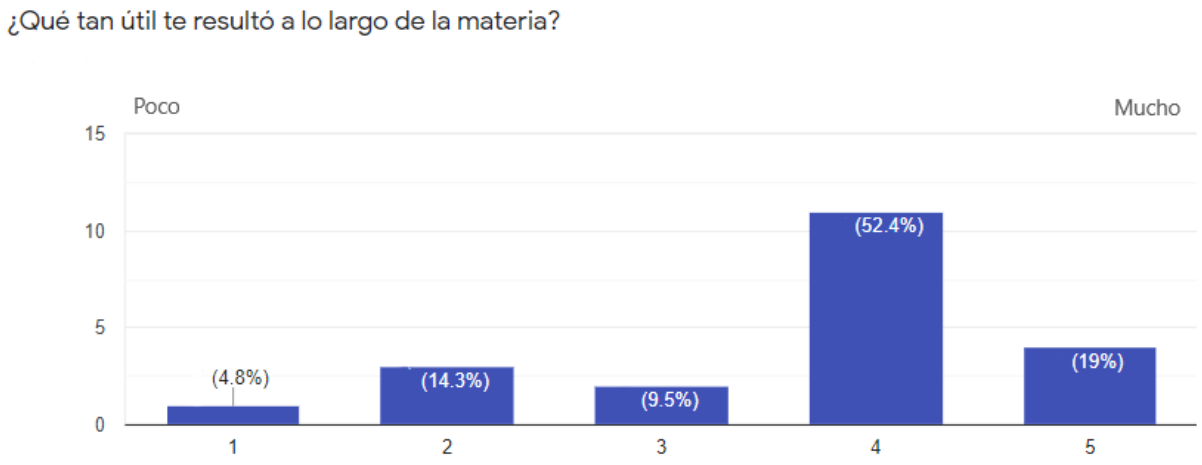
Por ejemplo, si solo se requiere de Q1, basta con habilitar únicamente las instrucciones MOV, MUL, ADD, SUB y DIV y los modos de direccionamiento Inmediato y Registro.

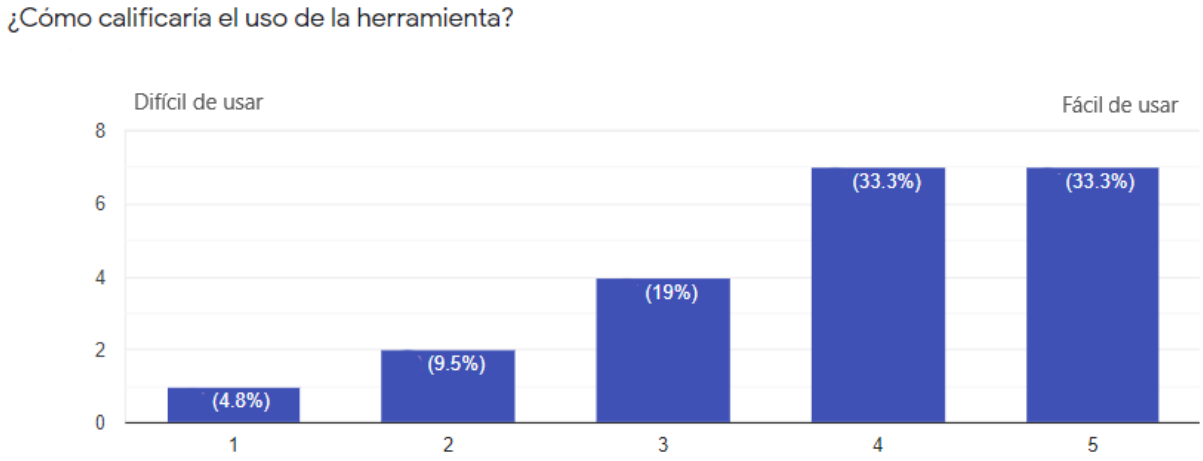
Capítulo 4

Devoluciones de estudiantes

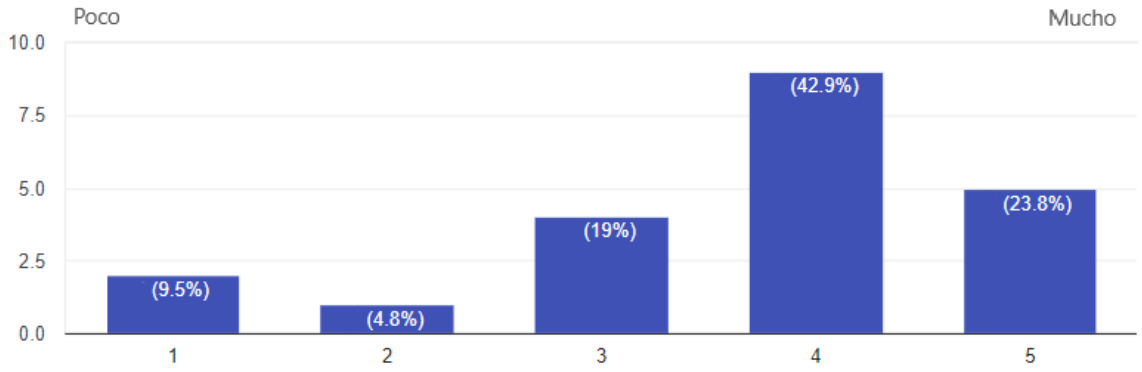
Como parte del trabajo incluía una prueba de uso por estudiantes, se decidió compartir la herramienta en las comisiones de Organización de Computadoras del segundo cuatrimestre del 2020. Al finalizar la cursada fue enviada una encuesta para conocer las opiniones de los y las estudiantes acerca de la herramienta para poder saber qué tan útil resultó la herramienta y conocer ideas de posibles mejoras.

Los gráficos de los resultados obtenidos en la encuesta se pueden ver a continuación.

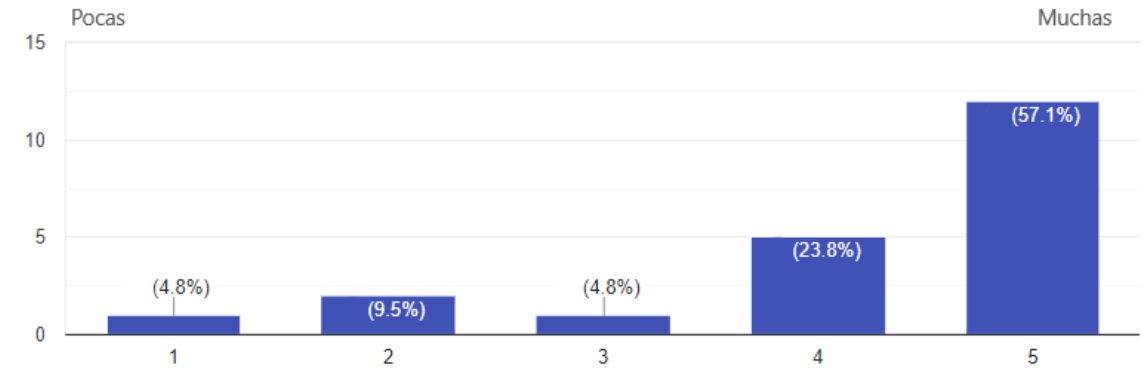




¿Su uso fue intuitivo?

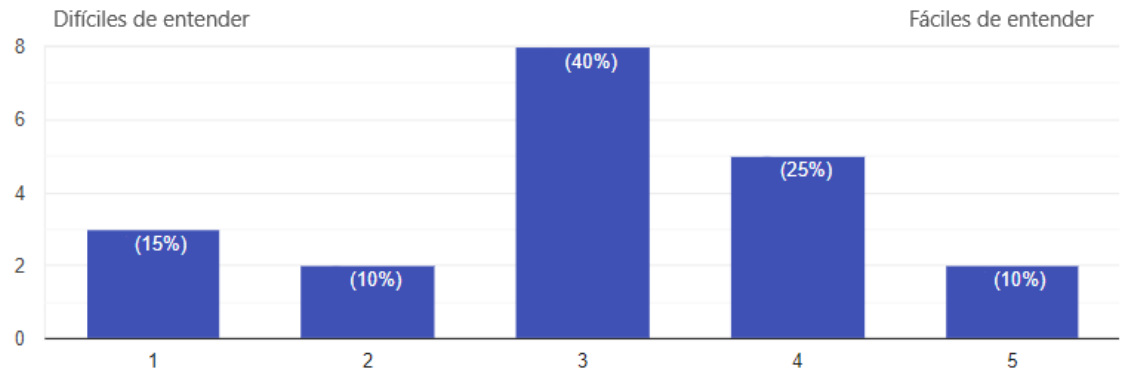


¿Cuáles son las posibilidades de que la recomiendes a compañeros que vayan a cursar en un futuro?



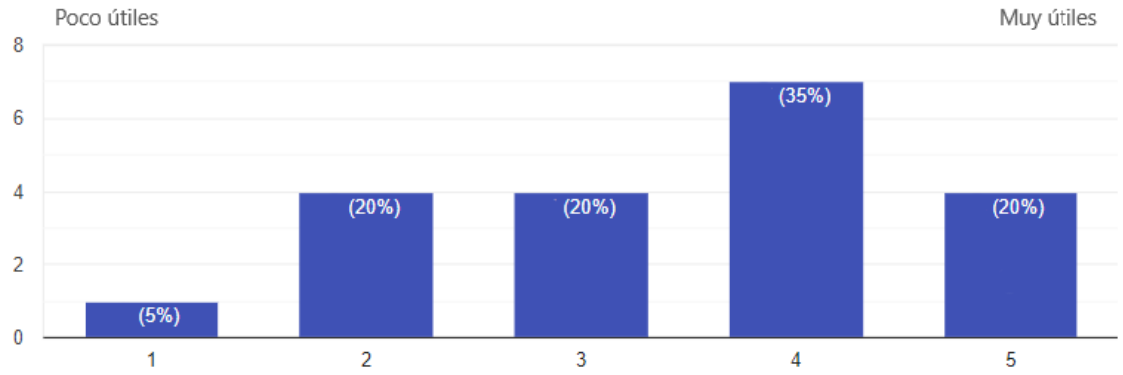
A pesar de que al comienzo del semestre los alumnos usaron una versión incompleta de QSim Web se puede observar, dado los resultados obtenidos, que la herramienta cumplió con su cometido, les fue útil, fácil de usar e intuitiva.

Los mensajes de error fueron:

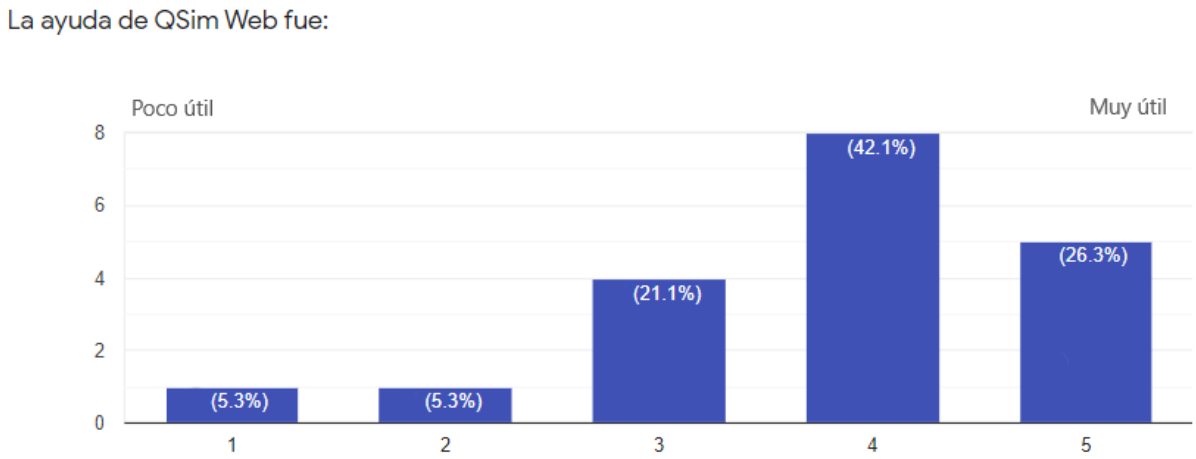


Aquí podemos observar, más allá de que el resultado no fue negativo, que los mensajes de error no fueron suficientemente declarativos.

Las configuraciones de QSim Web fueron:



La configurabilidad de la herramienta nos parece un punto fuerte, pero al estar la mayoría de las funcionalidades habilitadas por defecto, esperábamos que aquí el gráfico tienda a “Poco útiles”. Nos sorprende positivamente que haya sido de utilidad.



Además les hemos hecho preguntas donde se podían expresar de forma escrita. Estas preguntas apuntaban a conocer su opinión acerca de posibles mejoras de la herramienta, nuevas funcionalidades, funcionalidades que más gustaron y mejoras en la visualización de resultados. En general las respuestas fueron positivas y pudimos sacar varias conclusiones e ideas de nuevas funcionalidades para realizar a futuro, como se listarán a continuación. Muchos comentarios no fueron realizados sobre la última versión de la herramienta, ya que la misma crecía con el pasar del cuatrimestre, y algunas de las funcionalidades pedidas ya se encontraban implementadas.

Las nuevas funcionalidades son:

- Mejorar la sección de ayuda.
- Mejorar descripción de errores de sintaxis.
- Mejorar resaltado de sintaxis.
- Visualización de resultados en Ca2.
- Dar la posibilidad de ejecutar instrucciones paso a paso automaticamente con un tiempo de diferencia entre cada una.
- Poder detener la ejecución.
- Resaltado de la linea de código que se está ejecutando.
- Cuando se detecta un error de sintaxis, además de marcarlo dar una posible solución.
- Poder modificar valores de registros, memoria y flags mientras se está ejecutando un programa.

Capítulo 5

Conclusiones

El resultado de este trabajo es un simulador de la arquitectura Q que corre en un navegador y que puede ser usado por los y las estudiantes en el contexto de la materia Organización de Computadoras; permitiéndoles probar los programas que escriben, sin tener que simularlos en papel. Además, fomentó la curiosidad, ya que realizaron programas por fuera de las prácticas y pusieron a prueba sus conocimientos.

Desde el desarrollo de la herramienta notamos un aprendizaje en términos didácticos ya que habitualmente el software que realizamos en industria no es apuntado a la enseñanza. Sabemos que queda mucho por mejorar en este aspecto pero quienes hicimos este trabajo tenemos la intención de continuarlo para poder dar mejores funcionalidades tanto a estudiantes como a docentes.

Uno de los desafíos era la construcción de la idea de lenguaje, mediante el uso de gramáticas y parseo, aprendimos lo necesario para poder completar el trabajo y seguramente sigamos aprendiendo a lo largo de la licenciatura.

Construimos una librería pensando en evitar siempre el acoplamiento a nuestra interfaz de usuario e incluso a nuestro parser, creemos que puede ser realmente el puntapie inicial para que otros/as estudiantes de la carrera puedan intervenir y mejorar el trabajo realizado.

5.1. Trabajos futuros

Encontramos durante el desarrollo algunas ideas que pueden incluirse a futuro:

- Manejo de instrucciones I/O. Actualmente la arquitectura Q no define estas instrucciones pero podría ser una extensión que aporte interactividad a los programas.
- Permitir arquitecturas de computadoras distintas a Q, por ejemplo, poder variar el tamaño del bus de datos o direcciones.
- Incluir algún método de revisión docente, como por ejemplo deep link o envío de mails.
- Instrucción NOOP, para poder terminar programas sin necesariamente realizar una acción como última instrucción.
- Permitir la inicialización de registros y memoria antes de la ejecución.
- Mejorar el manejo de errores para mostrar mensajes más descriptivos.

Referencias

[1] Richard E. Pattis, *EBNF: A Notation to Describe Syntax*, University of California, Irvine.

[2] Nearley, *How to grammar good*, `https://nearley.js.org/docs/how-to-grammar-good`

[3] *Especificación - Q*, `http://orga.blog.unq.edu.ar/wp-content/uploads/sites/5/`215/08/Especificacion-Q.pdf