Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Proyecto de fin de Grado en Ingeniería Informática

Desarrollo de una herramienta software para la visualización de los registros de activación y el estado del cómputo durante la ejecución de un programa

Jaime María Alcalá Galicia

Dirigido por: Fernando López Ostenero

Curso 2022/2023, convocatoria septiembre

Desarrollo de una herramienta software para la visualización de los registros de activación y el estado del cómputo durante la ejecución de un programa

Proyecto de fin de Grado en Ingeniería Informática

de modalidad específica

Realizado por: Jaime María Alcalá Galicia

Dirigido por: Fernando López Ostenero

Fecha de lectura y defensa: 3 de octubre de 2023

Agradecimientos

Quisiera dar las gracias a mi mujer Bea, a mi hija Daniella y a mi hijo Alejandro, por apoyarme en todo momento y ser comprensivos por el tiempo que les he dejado de dedicar para hacer el proyecto.

Al resto de mi familia por todo el apoyo recibido durante todos estos años.

A todos mis compañeros de la UNED, que en mayor o menor medida me han ayudado a no flaquear durante la carrera.

Y a Fernando López Ostenero, por aceptar dirigir el proyecto y convertirlo en una herramienta de ayuda al estudiante.

Resumen

El objetivo del proyecto es el desarrollo de un lenguaje de programación y un compilador como herramienta para el estudio y aprendizaje del funcionamiento de los registros de activación y el estado del cómputo.

Al ejecutar un programa, las llamadas a los procedimientos y funciones, y los valores que devuelven los mismos se manejan en tiempo de ejecución mediante una pila llamada *pila de control*. Cada llamada en vivo a estos procedimientos y funciones produce en la parte superior de esta pila su propio *registro de activación*, que incluye todos los datos necesarios para el correcto funcionamiento del programa.

Asimismo, durante la ejecución de un programa, los valores que contienen las variables en un momento determinado de la ejecución se denomina *estado del cómputo*.

Estos dos conceptos de *registro de activación* y *estado del cómputo* se estudian en profundidad en la UNED a lo largo del Grado en Ingeniería Informática dentro de las asignaturas de Teoría de los Lenguajes de Programación, Procesadores del Lenguaje I y Procesadores del Lenguaje II.

El proyecto muestra el desarrollo de un entrono web donde el alumno puede escribir y compilar sus propios programas en un lenguaje muy sencillo diseñado adhoc para al proyecto.

Posteriormente a esta compilación, el usuario puede de una manera fácil y sencilla, ir viendo como se crean y destruyen los diferentes *registros de activación* dentro de la *pila de control*, los valores que se van almacenando en las diferentes partes que compone cada *registro de activación* y el *estado del cómputo* que es la evolución de las variables que va teniendo a lo largo de la ejecución del programa en un momento dado del mismo.

Abstract

The objective of the project is the development of a programming language and a compiler such as tool for the study and learning of the operation of activation register and the status of computation.

When a program runs, the procedures and functions calls, and the values they return are handled at run time by a stack called *stack control*. Every live call to these procedures and functions produces on then top of this stack its own *activation register*, which includes all the data necessary for the proper functioning of the program.

Likewise, during the execution of a program, the values contained in the variables in a specific moment of execution is called the *status of computation*.

These two concepts of *activation register* and *status of computation* are studied in depth at UNED throughout the Degree in Computer Engineering within the subjects of Theory of Programming Languages, Language Processors I and Language Processors II.

The project shows the development of a web environment where the student can write and compile its owns programs in a very simple language designed ad hoc for the project.

After this compilation, the user can, in an easy and simple way, see how the different *activation register* are created and destroyed within the *control stack*, the values that are stored in the different parts that make up each *activation register* and the *status of computation*, which is the evolution of the variables that it has throughout the execution of the program at a given moment in the program.

Palabras clave

Compilador

Lenguaje

Pila

Registro

Activación

Variable

Enlace

Control

Acceso

Estado

Cómputo

UNED

Teoría

Lenguaje

Programación

Web

HTML

CSS

JavaScript

Jison

Keywords

Compiler

Language

Stack

Register

Activation

Variable

Link

Control

Access

Status

Computation

UNED

Theory

Language

Programming

Web

HTML

CSS

JavaScript

Jison

Índice

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

El presente documento constituye la memoria del Proyecto de Fin de Grado con título *"Desarrollo de una herramienta software para la visualización de los registros de activación y el estado del cómputo durante la ejecución de un programa"*.

La intención principal del proyecto es intentar proporcionar a los alumnos de la asignatura *"Teoría de los lenguajes de programación"* impartida en el segundo curso del Grado de Ingeniería Informática en la **Universidad Nacional de Educación a Distancia** un entorno que les ayude a entender el funcionamiento de los registros de activación y el estado del cómputo.

Mediante este capítulo de introducción vamos a ver desde los motivos que nos llevan a realizar este proyecto, los objetivos que nos marcamos para su realización y cómo está estructurado el documento para poder comprender la realización del mismo.

1.1. Motivación

Hoy en día, el uso de herramientas digitales en las aulas es un recurso cada vez más extendido por parte de profesores y estudiantes en el estudio y refuerzo de las diferentes materias de aprendizaje. Dentro de estas herramientas podemos encontrar entre otras entornos virtuales de aprendizaje o entornos virtualizados donde un estudiante puede simular determinados procesos o flujos de trabajo, y observar paso a paso la evolución del entorno simulado para poder visualizar cómo van cambiando a lo largo de la simulación las diferentes situaciones y conceptos que se dan en el mismo, lo que es muy útil para una correcta comprensión del problema.

La **Universidad Nacional de Educación a Distancia**, dentro de su continuo proceso de renovación y mejora, se está apoyando en estas tecnologías para ayudar al estudiante dentro de su proceso de aprendizaje desarrollando diferentes contenidos audiovisuales y herramientas, incidiendo en puntos en los que los estudiantes encuentran más dificultades.

En el *Departamento de Lenguajes y Sistemas Informáticos* y más concretamente dentro de la asignatura de *Teoría de los Lenguajes de Programación* uno de los puntos claves es conocer el funcionamiento interno durante la ejecución de un programa. Durante esta ejecución se hacen llamadas a diferentes funciones y procedimientos, para lo cual el compilador tiene que apoyarse en una serie de estructuras para poder seguir el flujo tanto del programa como de los valores que va obteniendo. Todo esto está controlado por lo que se denomina el ambiente.

Entre estas estructuras hay dos en particular que son los registros de activación y el estado del cómputo. En ellas el ambiente va guardando las posiciones de memoria, de retorno y los valores que va obteniendo durante la ejecución de un programa para la correcta ejecución de mismo.

1.2. Alcance y objetivos

El proyecto se fundamenta en la elaboración de un entorno web a modo de herramienta, la cual permita a los alumnos desarrollar sus propios programas en un lenguaje creado ad-hoc para el entorno y puedan ver durante la ejecución del programa cómo evoluciona el estado del cómputo y los registros de activación.

Para ello los objetivos marcados son:

- Diseñar un lenguaje de programación sencillo capaz de admitir declaración de variables, subprogramas anidados y recursión.
- Desarrollar una herramienta software con un interfaz gráfico que sea capaz recoger programas escritos en el lenguaje diseñado.
- Diseñar un compilador dentro de esta herramienta que acepte el lenguaje que hemos diseñado y que compile a una máquina virtual capaz de mostrar cómo evolucionan los registros de activación y el estado del cómputo según se van ejecutando las diferentes instrucciones del programa introducido.

1.3. Estructura de la memoria

La memoria está estructurada en los siguientes capítulos.

- **1. Introducción**

  En este capítulo inicial hago una introducción al proyecto que voy a desarrollar, exponiendo las motivaciones que me han llevado a la realización del mismo, los objetivos que voy a llegar a realizar, y la estructura que va a presentar la memoria para la correcta realización y entendimiento del proyecto.

- **2. Estado del arte**

  En el capítulo 2 hago un análisis de las herramientas que existen en la actualidad y que contienen algún tipo de similitud al proyecto que voy a realizar, de tal manera que pueda comprobar que no existe una herramienta que haga lo mismo, y a la vez me permita tomar ideas de características que quiera incorporar al proyecto.

- **3. Propuesta**

  En el capítulo 3 desarrollo la propuesta del proyecto, incluyendo los pasos necesarios para llegar a realizar la aplicación y una descripción general de los elementos que van a componer el proyecto según lo analizado en en estado del arte 2.

- **4. Metodología**

  En el capítulo 4 se explica la metodología a usar en el proyecto, la razón de la elección de la misma, y se explica cómo se aplica la metodología al proyecto.

- **5. Historias de usuario**

  En el capitulo 5 se recogen las historias de usuario, donde se recogen todas las historias de usuario necesarias para la realización de proyecto.

- **6. Tecnología utilizada**

  En el capítulo 6 se explica toda la tecnología necesaria para la realización el proyecto en todas y cada una de sus fases.

- **7. Diseño y desarrollo del proyecto**

  En el capitulo 7 se aborda todo lo referente al diseño del proyecto, actividades necesarias para la elaboración de las diferentes partes que componen el mismo y la construcción de estas diferentes partes partes para que, una vez unidas, hagan cumplir los objetivos marcados para el proyecto, el uso de las herramientas empleadas para la correcta consecución del mismo, el diseño del lenguaje de programación que voy a crear, los diseños del código intermedio y código final que voy a necesitar para lograr los objetivos que me he marcado y por último los retoques finales que se dan al Front-End para dar por finalizado el proyecto.

- **8. Estudio económico**

  En el capítulo 8 describo todo lo referente a la estimación de costes.

- **9. Pruebas**

  En el capítulo 9 se detallan todas las pruebas que he ido haciendo sobre las diferentes partes que componen el programa para comprobar el correcto funcionamiento de cada una de ellas y en su conjunto.

- **10. Conclusiones y trabajos futuros**

  En el capítulo 10 muestro mis conclusiones sobre el proyecto y expongo las mejoras que se le pueden realizar en un futuro para aumentar la funcionalidad y usabilidad de cara al usuario final.

- **Bibliografía**

  En este capítulo muestro la recopilación de libros, textos y enlaces que he usado durante la realización del proyecto.

- **Anexos**

  Por último se pueden encontrar una serie de manuales y anexos donde se detallan en profundidad las diferentes partes de la aplicación o del desarrollo y funcionamiento del sistema para que sirvan a modo de consulta en caso de necesidad de ampliar la información sobre alguna de las partes del proyectos. Entre ellos podemos encontrar desde definiciones de las diferentes partes necesarias para crear el lenguaje y el compilador a manuales para consultar detalles de la aplicación.

Capítulo 2

Estado del arte

En este capítulo se analizan las características de otras herramientas y plataformas que incluyen compiladores, intérpretes y análisis de algoritmos con fines didácticos, para poder analizar cuanto se acercan a nuestros objetivos y ver si es posible incorporar alguna de sus características al proyecto.

2.1. Análisis

En la web, existen múltiples herramientas online que permiten compilar o interpretar algoritmos y muestran los resultados obtenidos. La gran mayoría tienen un marcado enfoque didáctico.

A continuación vamos a presentar el análisis de las características de alguna de estas plataformas.

2.1.1. TLP-Compiler

TLP-Compiler[1] es una herramienta creada para un *Proyecto de fin de grado* en la **Universidad Nacional de Educación a Distancia** orientada al aprendizaje de la tabla de símbolos.

Como se puede ver en la figura 2.1 , la herramienta permite escribir código de forma sencilla en un lenguaje denominado TLPUNED o cargar una serie de programas que vienen predefinidos para proceder al estudio de la tabla de símbolos. Una vez se tiene el programa fuente sin errores se pasa a la fase de ejecución donde se puede avanzar por las diferentes líneas de código para ver como va evolucionando el estado de la tabla de símbolos por cada una de las líneas de código por dónde se va pasando.

Como añadido, se permite mediante la sentencia '*' la posibilidad de establecer un punto donde el usuario tiene que resolver un ejercicio, el cual tiene que introducir los diferentes símbolos y las características de los mismos que existen en el punto que ha marcado con la sentencia '*'. Una vez hecho esto, si pinchamos en el botón 'Corregir', el sistema evalúa los datos que se han introducido y comprueba si el resultado es correcto, dándonos ligeras pistas en el caso de que no se haya llegado a una solución correcta.

También se puede mediante botones cambiar la colocación de los datos que se muestran por pantalla, permitiendo al usuario personalizar la vista de los resultados que se van produciendo.

[1] http://docencia.lsi.uned.es/TLP-Compiler

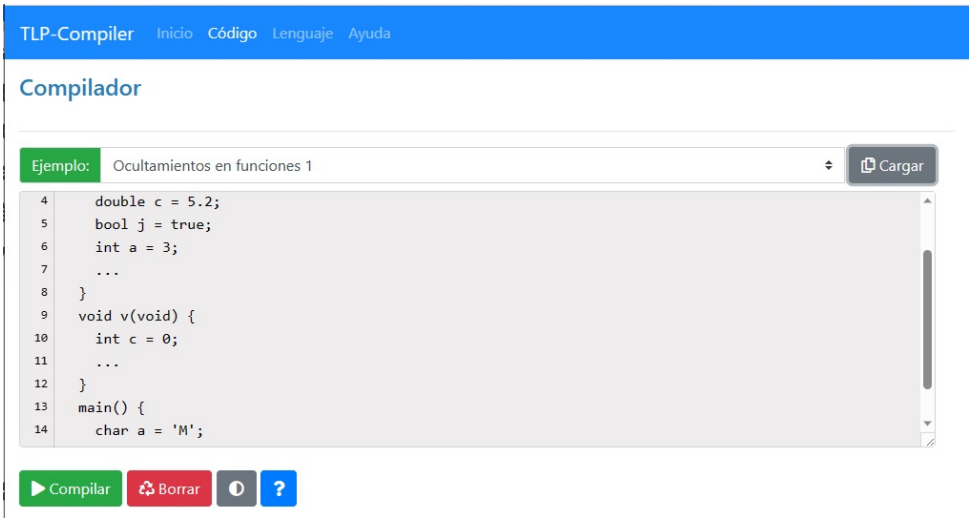

Figura 2.1: Captura de pantalla de TLP-Compiler.

Analizando el sistema en profundidad, vemos que internamente, el parser está implementado mediante un programa denominado Jison[2], el cual recibiendo como entrada una gramática nos genera genera un archivo JavaScript capaz de analizar el lenguaje descrito por esa gramática. Este punto es muy interesante ya que para la diseño del compilador voy a necesitar una herramienta que realice esa tarea.

Otro punto interesante es la manera de gestionar el interface de usuario, ya que no muestra nada de la fase de ejecución hasta que no se ha completado la fase de compilación.

Por contra, la herramienta solo se centra en la tabla de símbolos, por lo que no permite ejecutar el código fuente que se ha introducido para conocer el valor que va tomando cada unos de los símbolos durante la ejecución del programa, ni sus registros de activación.

2.1.2. VGA

VGA[3], como podemos ver en la figura 2.2, es también una herramienta creada para un *Proyecto de fin de grado* en la **Universidad Nacional de Educación a Distancia**[4] orientada al estudio del funcionamiento de una serie de algoritmos predefinidos como son los algoritmos de las familias de algoritmos voraces, divide y vencerás, y programación dinámica mediante la visualización del funcionamientos de los mismos.

Dentro de esta herramienta nos encontramos con una pantalla dónde podemos elegir para estudiar una de las tres familias de algoritmos descritas anteriormente . Una vez elegida la familia nos encontramos con la descripción tenemos la descripción de los algoritmos que la componen y la posibilidad de ver en funcionamiento uno de ellos.

Al elegir la opción 'Verlo en acción' nos da una descripción pormenorizada del enunciado del problema, la descripción del algoritmo que se va a usar, el código del algoritmo y las estructuras

[2] https://gerhobbelt.github.io/jison/docs/
[3] http://atlas.uned.es
[4] http://e-spacio.uned.es/fez/view/bibliuned:grado-ETSIInformatica-II-Ccaride

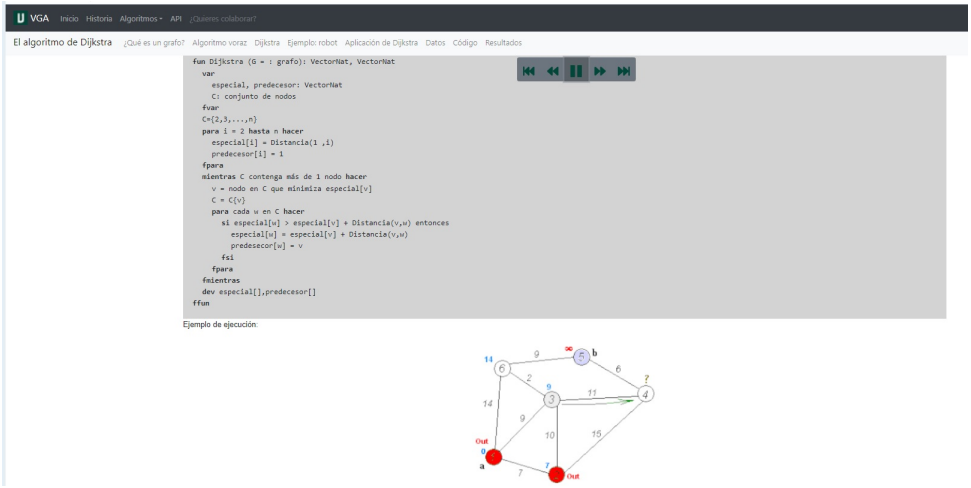

Figura 2.2: Captura de pantalla de VGA.

de datos necesarias para resolver el problema. En este punto podemos ejecutar paso a paso o realizar una ejecución completa para ver y comprender cómo funciona el algoritmo.

La herramienta está muy enfocada al aprendizaje, pero sólo aborda unos determinados tipos de algoritmos muy concretos, y si bien es cierto que podemos ir viendo paso a paso los valores que van tomando ciertas estructuras de datos en tiempo de ejecución, no muestra de manera completa el estado del cómputo ni los registros de activación.

2.1.3. OnlineGDB

Online GDB[5] es un compilador online que nos permite escribir y ejecutar programas en múltiples lenguajes. Además, en algunos de estos lenguajes se nos permite abrir un depurador para ir viendo la evolución de las variables que se declaran en el programa.

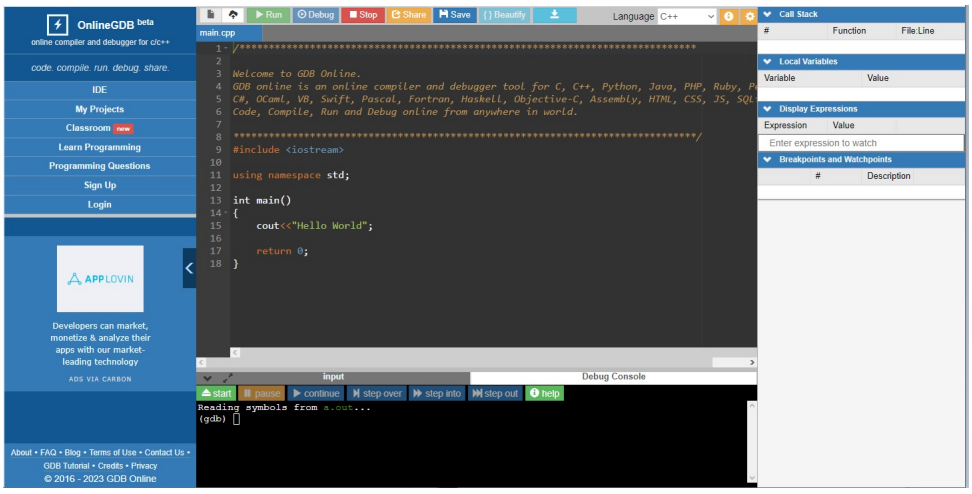

Figura 2.3: Captura de pantalla de OnlineGDB.

La forma de uso es bastante sencilla e intuitiva. Como podemos ver en la figura 2.3, dentro de

[5] https://www.onlinegdb.com/

la pantalla principal de la aplicación elegimos mediante un desplegable el lenguaje en el que queremos escribir nuestro programa y se nos carga un programa sencillo 'Hola mundo' a modo de ejemplo en el lenguaje elegido.

Una vez escrito el programa tenemos varias opciones, como son ejecutarlo directamente, y depurarlo. En caso de que elijamos depurarlo se nos abre un depurador en el que podemos ver toda la información necesaria con las variables, pila de llamadas, punto de interrupción y demás elementos que nos pueden ser útiles para la depuración del programa.

Existen también la posibilidad de crearse un usuario dentro de la plataforma, lo que nos permite guardar programadas y compartirlos con otras personas.

Como punto fuerte del sistema a intentar reproducir en la aplicación que voy a desarrollar está en que al ser un entorno online no tenemos nada que instalar para comenzar a usarla, que la parte del depurador que es muy sencilla e intuitiva para el usuario, y por último, la colocación mediante desplegables que permiten mostrar y ocultar datos de la información que puede ver el usuario durante la depuración.

Por el contrario, no hay manera de mostrar los registros de activación dentro de la aplicación, y no todos en todos los lenguajes se puede habilitar el modo de depuración, aunque parece que este modo lo está activando progresivamente según van creando esa funcionalidad para el lenguaje en cuestión.

2.1.4. Codepen

Codepen[6] es una herramienta online orientada al aprendizaje de entornos web.

Al igual que las anteriores plataformas, tiene la ventaja de que al ser online podemos comenzar a introducir inmediatamente el código sin requerimientos previos.

Como podemos ver en la figura 2.4, al entrar en el entorno nos encontramos con un interfaz sencillo y amigable que nos permite insertar código HTML, CSS y JavaScript en áreas de trabajo delimitadas para cada lenguaje, mostrando el resultado obtenido en un cuarto área dedicado a la visualización.

El usuario tiene la posibilidad de colocar las diferentes áreas en varias posiciones predefinidas para facilitar al usuario un mejor manejo de la herramienta.

Como ventaja tienes que al ser un entorno online orientado al aprendizaje, es muy rápido e intuitivo generar código e ir comprobando los resultados.

No obstante, al ser una herramienta centrada en el uso y aprendizaje de los lenguajes HTML, CSS y JavaScript para la generación de páginas web, no contiene un depurador donde podamos ir viendo al ejecutar el código JavaScript los valores que van tomando las variables ni ver los diferentes registros de activación que se pudiesen crear al ejecutar el código.

[6] https://codepen.io/

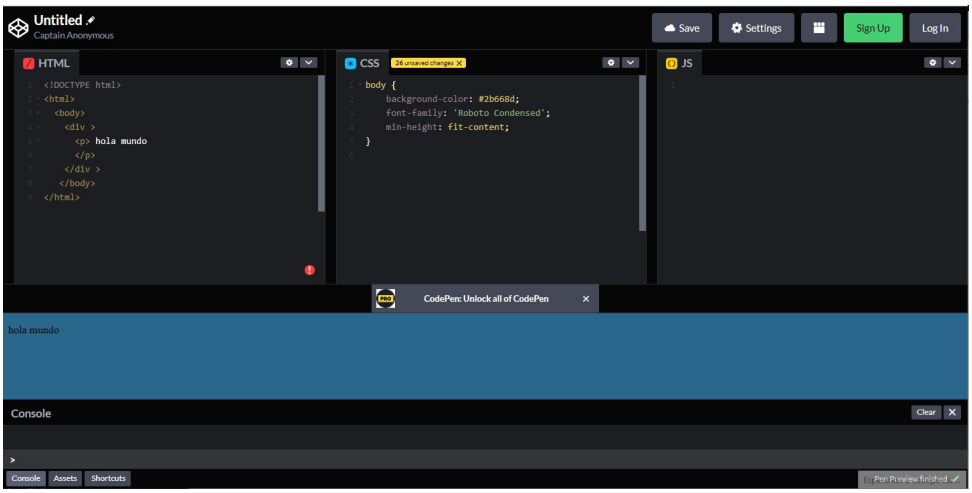

Figura 2.4: Captura de pantalla de Codepen.

2.1.5. ENS2001

ENS2001[7] es un ensamblador y simulador del estándar IEEE 694 [1] creado en un *Trabajo fin de carrera* en la **Universidad Politécnica de Madrid**.

Es una aplicación capaz de ejecutar un subconjunto de instrucciones del estándar IEEE 694 [1].

Tiene dos entornos diferenciados, el primero es en modo consola y el segundo, como podemos ver en la figura 2.5, es en modo gráfico. Ambos nos permiten introducir un programa mediante subconjunto de instrucciones del estándar IEEE 694 [1] y seguir instrucción a instrucción la evolución del mismo mediante la visualización de una serie de estructuras como son la pila de control, la memoria y el banco de registros.

Como ventaja respecto a los programas ya analizados destacaría y punto a reproducir es que es un entorno en el que se pueden ver cómo se van creando en la pila de control los diferentes registros de activación que genera el programa durante su ejecución, pero al no estar específicamente enfocado al estudio de los mismos, es complicado poder seguir su evolución.

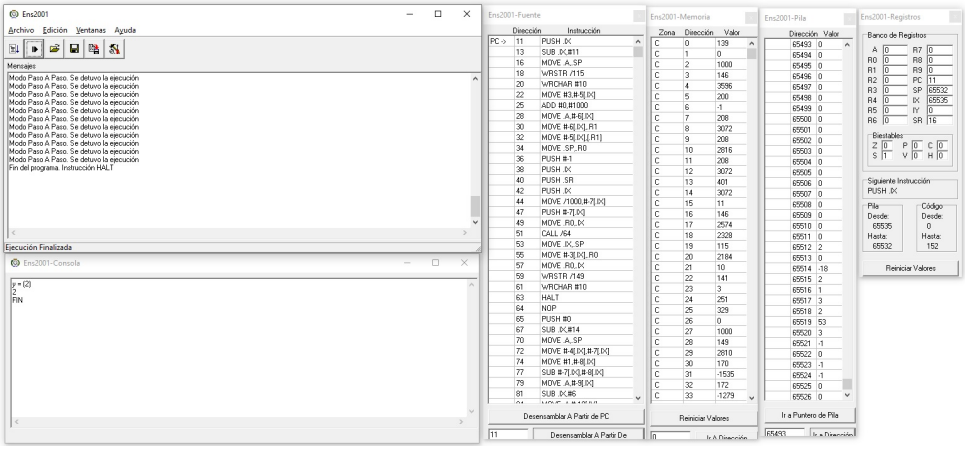

Figura 2.5: Captura de pantalla de ENS2001 en modo gráfico.

[7] https://ens2001.falvarez.es/

Como aspecto negativo respecto al fin que quiero conseguir, destacaría que hay que introducir los programa directamente en lenguaje ensamblador, lo que le aleja del objetivo propuesto que es poder escribir los programas que se quieran ejecutar en un lenguaje de programación sencillo.

2.2. Conclusiones

Aunque las herramientas mostradas cumplen con algunas características de las buscadas para el nuevo entorno, no hay ninguna que cumpla con la totalidad de las características requeridas, que son, un entorno web online que nos permita introducir código fuente de manera sencilla, y que nos permita hacer un seguimiento de la ejecución del programa para ver cómo va cambiando el estado del cómputo y cómo se van creando los diferentes registros de activación cuando se hacen llamadas a procedimientos y funciones.

Hay aspectos muy interesantes que he puesto como ventajas en bastantes de las herramientas analizadas que voy a usar o intentar reproducir, lo cual me sirve de base y guía para muchos de los puntos a desarrollar y que mencionaré en cada fase del diseño y desarrollo del proyecto, en el capítulo 7.

A continuación hacemos una recopilación de todos los puntos interesantes encontrados que vamos a usar o intentar reproducir en nuestro proyecto.

- El parser se va a implementar con Jison, ya que tomando como entrada una gramática libre de contexto podemos generar un fichero JavaScript capaz de parsear el lenguaje que describe la gramática.
- Sería interesante gestionar el interface de usuario, de modo que no se muestre la operativa de visualización de los registros de activación y estado del cómputo hasta que no se ha completado la fase de compilación.
- Posibilidad de ejecutar paso a paso hacer una ejecución completa.
- No debe tener dependencias con otros programas. Debe funcionar con un simple explorador.
- El interface de usuario debe de ser muy sencillo.
- La colocación mediante desplegables que permitan mostrar y ocultar datos de la información que puede ver el usuario durante la depuración.
- La información debe verse de forma fácil e intuitiva al ir comprobando resultados.
- La simulación y gestión de la pila de control puede basarse en cómo lo hace ENS2001 aunque hay que que intentar que sea mas fácil su seguimiento mediante algún tipo de información o ayuda visual.

Capítulo 3

Propuesta

El objetivo principal de este proyecto es crear un programa donde los alumnos puedan visualizar los registros de activación y el estado del cómputo durante la ejecución de un programa.

Como es un entorno orientado al aprendizaje, y tras el estudio del estado del arte, la plataforma donde voy a crear el programa va a ser una página web ya que es un entorno al cual puede acceder el alumno desde cualquier navegador. Esta página debe de ser lo mas sencilla y amigable posible para que el alumno se centre en el objeto de aprendizaje y no en aprender el funcionamiento del programa.

A grandes rasgos el programa se va a dividir en dos fases como se puede ver en la figura 3.1.

1. **Fase 1.** Los alumnos podrán escribir un programa en un lenguaje de programación y lo compilarán para ver si cumple con todas las características requeridas por el lenguaje.
2. **Fase 2.** Los alumnos podrán ejecutar el programa que han escrito como si fuese una especie de depurador donde podrán ver en cada instrucción que consumen cómo evolucionan los registros de activación y el estado del cómputo.

Parte del desarrollo de este proyecto viene ligado a las asignaturas de *Procesadores del lenguaje I* y *Procesadores del lenguaje II* que se estudian en el Grado de Ingeniería Informática en la **Universidad Nacional de Educación a Distancia**.

Debido a que el desarrollo del proyecto lo voy a hacer a la vez que curso estas asignaturas, el desarrollo de este va a estar supeditado al avance del curso en las mismas, ya que, en el trascurso del año según se va avanzando en el temario de estas, se van estudiando las diferentes fases necesarias para construir un compilador, las cuales se pueden ver en la figura 3.2, y que a su vez es la primera parte del proyecto que voy a desarrollar.

Con estas premisas, dentro de la fase 1, lo primero voy a hacer es diseñar un nuevo lenguaje de programación muy reducido que llamaré PFGUnedTLP el cual admita declaración de variables, subprogramas anidados y recursión. Este lenguaje va a servir al usuario de base para escribir programas en la aplicación a desarrollar de modo que pueda ser compilado. Como es un entorno orientado al aprendizaje, habrá una serie de ejemplos de programas para que el usuario pueda cargarlos sin necesidad de escribirlos.

Cuando el usuario compile su programa, se tienen que llevar a cabo una serie de tareas para comprobar que el programa es correcto, que como como se puede leer en [2], las fases que tanto un intérprete como un compilador deben llevar a cabo son:

1. Primero, un analizador léxico debe identificar los tokens del programa (palabras clave, constantes, identificadores, etc.), ya que inicialmente el programa se entiende como una

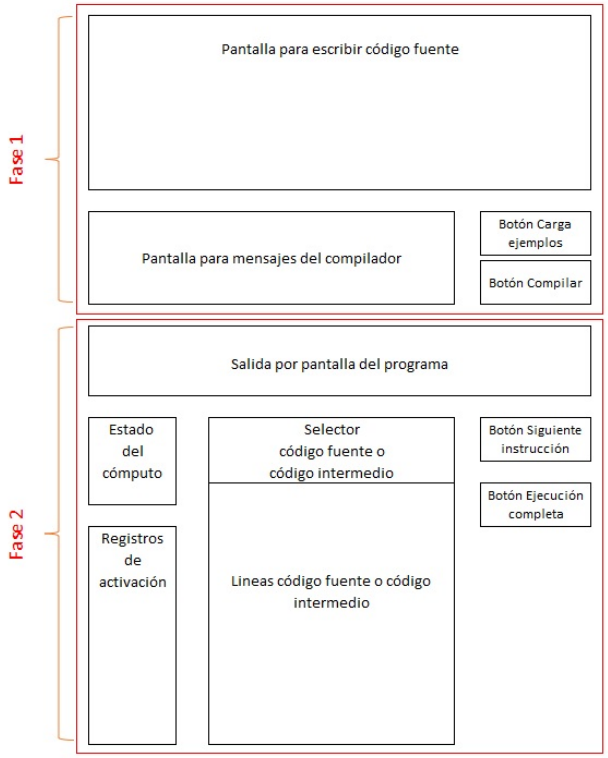

Figura 3.1: Diseño Borrador inicial.

secuencia de caracteres. En ocasiones, hay un preprocesamiento previo, para transformar el programa en una entrada correcta del analizador léxico.

2. A continuación, un analizador sintáctico o gramatical identifica las estructuras correctas que definen las secuencias de tokens.

3. Finalmente, un analizador semántico asigna el significado de forma suficiente para su ejecución o la obtención del programa objetivo.

En caso de que haya algo incorrecto en el programa fuente se le informará al usuario en forma de pantalla de error con la información necesaria para que corrija el error y vuelva a compilar. Este paso se repetirá hasta que el programa esté libre de errores.

Ciñéndome a planning de estudio de las asignaturas, las fases donde se crea el analizador léxico y el analizador sintáctico o gramatical se estudian a fondo en la asignatura de *Procesadores del lenguaje I*, mientras que en analizador semántico se ya se trata en *Procesadores del lenguaje II*.

Una vez terminado el del analizador semántico, también dentro de la asignatura de *Procesadores del lenguaje II*, y cómo se explica en [3] capítulo 6, se estudia el siguiente paso a realizar que es la generación del código intermedio.

Este código intermedio es una representación lineal de la secuencia de pasos que tiene que realizar el programa para ejecutar cada instrucción.

Llegado a este punto en el que el usuario ha escrito su programa en lenguaje PFGUnedTLP y lo ha compilado sin que haya tenido ningún tipo de errores léxicos, sintácticos ni semánticos,

nos adentraremos en la fase 2, donde se nos habilitará una pantalla que no nos permitirá ver el código intermedio que ha generado el compilador. Este será mostrado dentro de una lista en forma de cuádruplas.

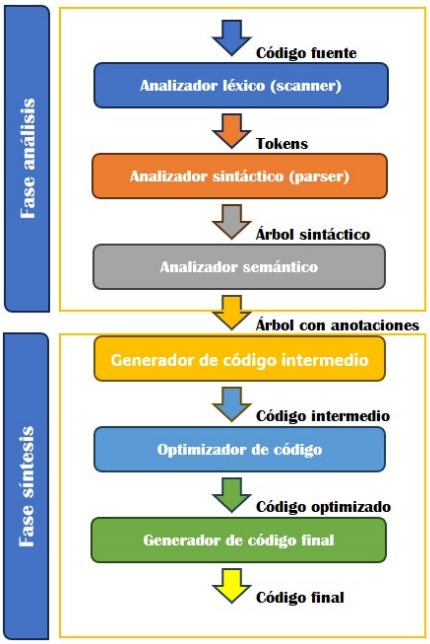

Figura 3.2: Fases de un compilador

También se habilitarán nuevas las opciones que permitan al usuario ir leyendo estas instrucciones de código intermedio, y a su vez generar una representación dentro de la web de la evolución de los registros de activación y del estado del cómputo. Para ello voy a crear una máquina virtual que interprete el código intermedio generado y donde pueda ir paso a paso, basándome en la máquina que se estudia y utiliza para la práctica de *Procesadores del lenguaje II* [4]ENS2001[1] que es a su vez un *Trabajo de fin de carrera* realizado en la **Universidad Politécnica de Madrid** por el alumno Federico J. Álvarez Valero para ir visualizando esta información según se vaya pasando por las instrucciones de código intermedio.

Del mismo modo que con el código intermedio, se habilitará también la posibilidad para que el usuario pueda ir avanzando paso a paso por las líneas de código fuente del programa.

Por último se habilitarán ayudas para poder relacionar las posiciones de los diferentes registros de activación y el contenido de los mismos para que el usuario pueda visualizar la relación entre los mismos.

[1] https://ens2001.falvarez.es/

Capítulo 4

Metodología

Habiendo desplegado a grandes rasgos las lineas maestras en las que se basa la realización del proyecto y una vez conocidas las características del mismo, hay que tener en cuenta que el desarrollo estará condicionado por lo que se vaya aprendiendo en las asignaturas de *Procesadores del lenguaje I* y *Procesadores del lenguaje II* ya que, a medida que avanzo en el curso voy definiendo las partes de la solución a implementar aplicando los conocimientos adquiridos.

Para el desarrollo del proyecto he escogido un marco de trabajo con metodología Agile, en concreto Scrum. Esta metodología ágil permite dar al proyecto un enfoque iterativo en cuanto a la planificación y el desarrollo del mismo. De igual forma, en caso de ser necesario, me permite la suficiente flexibilidad como para poder ir modificando las partes del programa que considere no estén bien definidas o quizá, aun contando con una definición completa sea necesario reestructurarlas.

Otra de las claves para la utilización de esta metodología, es la necesidad de contar con mecanismos que me permitan cambiar la prioridad de ejecución de las tareas de tal manera que se minimicen los tiempos muertos dentro del proyecto.

Bajo el marco teórico de la metodología presentada, debemos tener en cuenta que por las características que engloban el *Proyecto de fin de grado*, se deben realizar una serie de ajustes que afectan principalmente a los roles asignados. En este contexto, el total de las funciones serán desarrolladas tanto por el director del proyecto, Fernando López Ostenero, como por mi.

4.1. ¿Qué es Agile?

Agile es una metodología que se usa principalmente en desarrollo de software que hace hincapié en la flexilibidad e implementación eficiente para planificar el workflow, lo cual nos da la capacidad de elegir en cada situación la opción mas correcta sin comprometer al proyecto.

Se basa en 4 valores y 12 principios recogidos en el llamado **Manifiesto Ágil** donde se postula la base de la metodología y que mencionamos a continuación.

Los 4 valores sobre los que se basa la metodología Ágil son:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.

- Respuesta ante el cambio sobre seguir un plan.

Bajo esos cuatro valores se redactaron los siguientes principios:

- Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al período de tiempo más corto posible.
- Los responsables del negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos ágiles promueven el desarrollo sostenido. Los promotores, desarrolladores y usuarios debemos mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de los equipos auto-organizados.
- A intervalos regulares, el equipo reflexiona sobre cómo volverse más efectivo para, a continuación, ajustar y perfeccionar su comportamiento en consecuencia.

El trabajo se organiza en una serie de iteraciones cortas también llamadas sprints en las que el equipo de trabajo debe completar unas serie de trabajo establecido y prepararlo para su presentación y revisión. Para proyectos de larga duración tradicionalmente los sprints suelen ser cada 30 días, pero el periodo de tiempo al igual que casi todo en esta metodología no es fijo, sino que se va adaptando al proyecto.

4.2. ¿Qué es Scrum?

Scrum es un sistema o proceso en el que regularmente hay que aplicar un conjunto de buenas prácticas para obtener el mejor resultado dentro de un proyecto trabajando colaborativamente en equipo. Estas prácticas interactúan entre ellas y tienen su origen en el estudio de la forma de trabajar de equipos altamente productivos.

4.2.1. Marco teórico

El principio básico de Scrum es la realización de entregas parciales y periódicas del producto final, dando prioridad a las que mas beneficio aportan al cliente. Por ello, Scrum es altamente efectivo para para proyectos en entornos complejos, donde los requisitos varían a lo largo del desarrollo del proyecto, donde no están completamente definidos y donde la flexibilidad es fundamental, lo que lo hace un marco ideal para el desarrollo del proyecto ya que parte de los requisitos los iré extrayendo a lo largo del curso de las asignaturas de Procesadores del lenguaje I y Procesadores del lenguaje II.

Al no ser un proceso propiamente dicho sino un marco de trabajo, se pueden usar un grupo de técnicas y procesos diferentes.

4.2.2. Ciclo de Scrum

El ciclo de Scrum comienza con el Sprint 0, en el cual se hace una reunión donde se aborda de forma global el problema y a partir de ese paso inicial se priorizan y detallan las funcionalidades que se necesitan tener en primer lugar. Esta reunión inicial se hace al principio de cada ciclo para revisar y modificar los siguientes incremento si es necesario.

Al finalizar el sprint se realiza lo que se denomina el incremento, donde se hace una entrega de parte de la funcionalidad de del producto. Este ciclo o iteración es continuo hasta la finalización del proyecto, y entre la planificación de cada sprint que es lo que marca el inicio del ciclo y la entrega del incremento se hacen breves reuniones diarias donde se hace una revisión del trabajo de cada miembro del día anterior y del día actual. Es lo que se denomina "daily scrum". Al finalizar el sprint también se hace la reunión de revisión donde se revisa y prueba la funcionalidad del incremento y la reunión retrospectiva, que se centra en cómo se está construyendo el proyecto.

Estos ciclos son continuos hasta la finalización de todos los sprints que componen el proyecto, en cuyo caso, se ha debido llegar a la finalización del proyecto y puede darse por cerrado con el cliente.

En el siguiente esquema se pueden ver los componentes del ciclo estandar de Scrum:

- **Roles**

  Forman el equipo de desarrollo.

  - Equipo de desarrollo o "Development Team".
  - Propietario del producto o "Product Owner".
  - Scrum Master.

- **Artefactos**

  - Pila de producto o "Product Backlog".
  - Pila de sprint o "Sprint Backlog".
  - Incremento.

- **Eventos**
  - Sprint, también denominado ïteración".
  - Reunión de planificación del sprint o "Sprint Planning".
  - Scrum diario o "Daily Scrum".
  - Revisión del sprint o "Sprint Review".
  - Retrospectiva del sprint o "Sprint Retrospective".

4.3. Aplicación de la metodología elegida al proyecto

Al comienzo del capítulo se han puesto sobre la mesa los motivos de la elección de la metodología para el desarrollo del proyecto y a continuación se explica la aplicación de la metodología sobre el ciclo de Scrum.

Eliminando la parte contractual que entraría dentro de la lógica de negocios de cualquier empresa de servicios, este paso se sustituye por el proceso de propuesta, validación y firma del anteproyecto al tratarse de un proyecto de fin grado.

El siguiente paso es dotar al proyecto de una visión global de los pasos a realizar durante toda la ejecución del proyecto para poder extraer los diferentes trabajos a realizar, y bajo esa linea temporal, dar comienzo al ciclo Scrum y sus reglas.

Figura 4.1: Etapas del desarrollo del proyecto.

En la figura 4.1 podemos ver la línea temporal a seguir para la realización del proyecto.

4.3.1. Stakeholders

Una vez establecida la linea de trabajo, el siguiente paso es recopilar todos los interesados en el proyecto y los roles que van a cumplir.

Por las características especiales de un proyecto de fin de grado, el desarrollo completo se tiene que realizar realizar por una sola persona que es el alumno que realiza el proyecto, por lo cual la mayoría de las figuras están representadas por mi y el director de proyecto.

Registro de interesados(Stakeholders)						
Nombre	**Posición**	**Organización**	**Rol dentro del PFG**	**Información de contacto**	**Requisitos**	**Expectativas**
Jaime María Alcalá Galicia	Alumno PFG	UNED	Scrum Master y Development Team	@alumno.uned.es	Aplicación web Memoria Defensa PFG	Superar PFG
Fernando López Ostenero	Director PFG	UNED	Product Owner	@lsi.uned.es	Aplicación web Memoria	Dirección del proyecto
Tribunal	Tribunal PFG	UNED	Interesados	-------	Evaluación	PFG apto

Figura 4.2: Stakeholders (Registro de interesados).

En la figura 4.2 se presenta el registro de interesados o Stakeholders del proyecto.

4.3.2. Artefactos

El siguiente paso en la metodología Scrum es la presentación de los artefactos que la componen.

Los primeros que aparecen son la pila de producto inicial o Initial Product Backlog y Initial sprint backlog o Sprint Backlog inicial del proyecto. En ellos se escriben de forma global las funcionalidades y requisitos que va a tener el proyecto para su correcta consecución y una planificación de las mismas. Son creados tras la reunión inicial del Product Owner y el Scrum Master según los requerimientos del proyecto, que en nuestro caso es el proyecto de fin de grado.

INITIAL SPRINT BACKLOG PFG JAIME

ID	Descripción\Requerimiento	Status	Pte. Reparto	Esfuerzo	Prioridad	Sprint 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
PFG-00	Planificación del proyecto. Líneas maestras.	Planificada	0	5	1	5																						
PFG-01	Investigar asignatura Procesadores del lenguaje I.	Planificada	0	25	2	5	8	12																				
PFG-02	Aprendizaje tecnologías HTML, CSS y JavaScript	Planificada	0	40	3	6	6	6	1	3	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		
PFG-03	Revisar otras aplicaciones similares (Estado del arte).	Planificada	0	8	4		6	2																				
PFG-04	Crear lenguaje programacion para el compilador.	Planificada	0	4	5				4																			
PFG-05	Crear analizador léxico y sintáctico.	Planificada	0	25	6				12	13																		
PFG-06	Investigar asignatura Procesadores del lenguaje II.	Planificada	0	31	7						7	16	8															
PFG-07	Crear analizador semántico.	Planificada	0	38	8								8	15	15													
PFG-08	Crear código intermedio sin llamada a subprogramas.	Planificada	0	10	9											10												
PFG-09	Crear cuadruplas sin llamada a subprogramas.	Planificada	0	50	10												16	16	18									
PFG-10	Código intermedio con llamada a subprogramas.	Planificada	0	5	11															5								
PFG-11	Crear cuadruplas con llamada a subprogramas.	Planificada	0	30	12															12	18							
PFG-12	Funcionalidad para leer e interpretar las cuádruplas.	Planificada	0	50	13																	12	15	15	8			
PFG-13	Front-End final. Crear Debugger.	Planificada	0	10	14																				7	3		
PFG-14	Comprobaciones finales y banco de pruebas.	Planificada	0	10	15																					10		
PFG-15	Aprender Latex.	Planificada	0	6	16						3															3		
PFG-16	Documentacion, manuales, memoria PFG y anexos.	Planificada	0	80	17				3	3	2	2	2	2	2	3	2	2	1	2	2	4	4	5	5	4	20	10
PFG-17	Preparacion y defensa PFG.	Planificada	0	10	18																							10
PFG-18	Costes indirectos administración Scrum.	Planificada	0	13	19	2				1	2					2,5				1		2,5					1	1
						Horas	13	20	20	20	20	17	19	19	18	18	16,5	19	19	20	21	21	19,5	20	21	21	21	21	21
						Media	18,3					18,2					19,4						20,5					21,0	

Figura 4.3: Sprint Backlog inicial del proyecto.

En las figuras 4.3 y 4.4 podemos ver el Sprint Backlog inicial y el Product Backlog inicial del proyecto.

Como he comentado con anterioridad Scrum no necesita una planificación total y exaustiva previa, sino una planificación de partida con la que comenzar el proyecto, ya que según van avanzando los sprints esta planificación varia considerablemente, descartando alguna tareas, o replanificando y añadiendo otras según los requerimientos del proyecto en cada sprint.

INITIAL PRODUCT BACKLOG PFG JAIME

ID	Descripción\Requerimiento\Funcionalidad	Esfuerzo	Prioridad	Sprint	Status
PFG-00	Planificación del proyecto. Líneas maestras.	5	1	0 al 2	Planificada
PFG-01	Investigar asignatura Procesadores del lenguaje I.	30	1	0 al 2	Planificada
PFG-02	Aprendizaje tecnologías HTML, CSS y JavaScript	40	2	0 al 10	Planificada
PFG-03	Revisar otras aplicaciones similares (Estado del arte).	8	3	0, 1	Planificada
PFG-04	Crear lenguaje programacion para el compilador.	4	4	1	Planificada
PFG-05	Crear analizador léxico y sintáctico.	25	5	1, 2	Planificada
PFG-06	Investigar asignatura Procesadores del lenguaje II.	31	6	3 al 5	Planificada
PFG-07	Crear analizador semántico.	38	7	3 al 5	Planificada
PFG-08	Crear código intermedio sin llamada a subprogramas.	10	8	5, 6	Planificada
PFG-09	Crear cuadruplas sin llamada a subprogramas.	50	9	5 al 7	Planificada
PFG-10	Código intermedio con llamada a subprogramas.	5	10	7	Planificada
PFG-11	Crear cuadruplas con llamada a subprogramas.	30	11	7, 8	Planificada
PFG-12	Funcionalidad para leer e interpretar las cuádruplas.	50	12	8 al 10	Planificada
PFG-13	Front-End final. Crear Debugger.	10	13	10	Planificada
PFG-14	Comprobaciones finales y banco de pruebas.	10	14	10	Planificada
PFG-15	Aprender Latex.	6	15	0 al 2, 11	Planificada
PFG-16	Documentacion, manuales, memoria PFG y anexos.	80	16	0 al 12	Planificada
PFG-17	Preparacion y defensa PFG.	10	17	12	Planificada
PFG-18	Costes indirectos administración Scrum.	13	18	0 al 12	Planificada

Figura 4.4: Product Backlog inicial del proyecto.

4.3.3. Historias de usuario

Otro elemento importante dentro de Scrum son las historias de usuario.

Una historia de usuario es una explicación general e informal de una función requerida por el software que vamos a desarrollar escrita desde la perspectiva del usuario final, ya que el primer valor dentro del desarrollo de software ágil es poner a las personas antes por delante de los desarrollos y los procesos, y las historias de usuarios introducen al usuario real en el centro de la conversación y por consiguiente en el desarrollo.

Una opción recomendable es es escribirlas con la técnica desarrollo guiado por el comportamiento propia de BDD (Behavior Driven Development) y con Gherkin que es un lenguaje específico de dominio BDD.

BDD (Behavior-Driven Development) o desarrollo guiado por comportamiento es un marco colaborativo que permite interactuar al equipo de desarrollo con las personas que conocen el negocio y los usuarios lo cual reduce mucho la distancia que suele haber entre ellos en el desarrollo de proyectos.

BDD va siempre de la mano del lenguaje Gherkin. Gherkin es un lenguaje específico de dominio que sirve para la resolución de problemas muy concretos. Se basa en el patrón básico de de Role-Feaure-Reason (Rol-Característica-Motivo) y el elemento Given-When-Then (Como [x], quiero [y] para que [z]), lo cual nos ayuda a desarrollar toda la funcionalidad dentro del lenguaje Gerkin.

El desarrollo de estas historias de usuario lo podemos ver completo en el capítulo 5 Historias de usuario.

4.3.4. Planificación inicial

En el sprint 0 se crean los artefactos Sprint Backlog inicial 4.3 y el Product Backlog inicial 4.4 de donde se obtiene una línea temporal con los diferentes sprints necesarios para el desarrollo de la metodología.

Se ha dividido cada sprint en unas 20 horas aproximadamente distribuyendo así los esfuerzos de una forma mas o menos uniforme durante todo el desarrollo del proyecto, pasando así de la descripción de las diferentes etapas que se habían creado al inicio del proyecto que veíamos en la figura 4.1 a una planificación desglosada, ordenada y temporal de cada unos de los item para la consecución del proyecto.

El sprint 0 finaliza con la reunión de revisión de este sprint inicial una vez aceptado el anteproyecto por la **Universidad Nacional de Educación a Distancia**, en donde se acuerda proceder a la matriculación en la tutela del *Proyecto de fin de grado*. En la parte técnica se acuerda en que el entorno integrado de desarrollo (IDE) va a ser *Visual Studio Code* y además, el tutor como Product owner, me recomienda para ver el estado del arte una serie de aplicaciones que revisar.

4.3.5. Planificación final del proyecto

Durante el paso de las diferentes iteraciones, y debido a la filosofía de Scrum con el continuo análisis y replanificación del sistema en cada Sprint Review, se puede comprobar en la figura 4.5 como se llega a una planificación final totalmente distinta a lo que teníamos planificado en el Product Backlog inicial del proyecto 4.4.

PRODUCT BACKLOG PFG JAIME				
ID-Fase	Sprint	Descripción\Requerimiento\Objetivo	Prioridad	Status
PFG00 - Contacto	0	Toma de requerimientos.	1	Finalizada
		Redacción Product Backlog.	1	Finalizada
PFG01 - Inicio	1	Descripción del sistema.	1	Finalizada
		Estudio del estado del arte.	1	Finalizada
		Escritura historias de usuario.	1	Finalizada
		Elección de las tecnologías a usar.	1	Finalizada
		Planificación del proyecto.	1	Finalizada
		Estimación de costes.	2	Finalizada
		Redacción del anteproyecto.	2	Finalizada
PFG02 - Lenguaje programación	2	Diseño del lenguaje de programación.	3	Finalizada
		Especificación del lenguaje programación.	3	Finalizada
PFG03 - Analizador léxico	3	Diseño analizador léxico.	4	Finalizada
		Desarrollo analizador léxico.	4	Finalizada
PFG04 - Analizador sintáctico	4	Diseño analizador sintactico.	5	Finalizada
		Desarrollo analizador sintactico.	5	Finalizada
PFG05 - Prototipo Front-End	5	Prototipo interfaz usuario.	6	Finalizada
		Pruebas analizador léxico y sintáctico.	7	Finalizada
PFG06 - Analizador semántico	6	Elaboración modelo de datos.	8	Finalizada
		Diseño del analizador semántico.	9	Finalizada
		Implementación del analizador semántico.	9	Finalizada
		Pruebas analizador semántico.	9	Finalizada
PFG07 - Código intermedio	7	Diseño código intermedio.	10	Finalizada
		Desarrollo código intermedio.	10	Finalizada
		Pruebas código intermedio.	10	Finalizada
PFG08 - Prototipo Front-End simulador	8	Prototipo Iterfaz usuario con máquina virtual simulación.	11	Finalizada
PFG09 - Código final	9	Diseño código final.	12	Finalizada
		Desarrollo código final.	12	Finalizada
PFG10 - Desarrollo final Front-End	10	Desarrollo interface simulacion.	13	Finalizada
		Pruebas código final.	14	Finalizada
PFG11 - Representación gráfica	11	Representación gráfica registro activación.	15	Finalizada
		Representación gráfica estado del cómputo.	15	Finalizada
PFG12 - Refinamiento	12	Refinamiento del proyecto.	16	Finalizada
		Pruebas finales.	17	Finalizada
		Elaboración manuales.	18	Finalizada
PFG13 - Documentación PFG	13	Elaboración de la memoria.	19	En curso
		Elaboración de las presentaciones.	19	Planificada
PFG14 - Presentación PFG	14	Presentación del proyecto.	20	Planificada

Figura 4.5: Product Backlog del proyecto.

En los siguientes capítulos se desglosan los principales trabajos y tareas que se han ido desarrollado a lo largo de todo el proyecto hasta la consecución del mismo.

Capítulo 5

Historias de usuario

Durante el desarrollo de este proyectos de fin de grado se han ido recopilando todas las historias de usuario necesarias para el desarrollo de la aplicación.

Las historias de usuario sirven para definir la interacción de un usuario con la interfaz, lo que nos proporciona toda la información necesaria para realizar un correcto diseño de la misma.

En la la figura 5.1 se puede ver un diagrama que comprende todas las historias de usuario que componen la aplicación. Como se puede observar en la misma todas estas historias las realiza un sólo actor que es el estudiante, ya que en el aplicativo no hay necesidad de ningún tipo de administración o tarea distinta a la ejecución del programa que requiera de otro tipo de usuario.

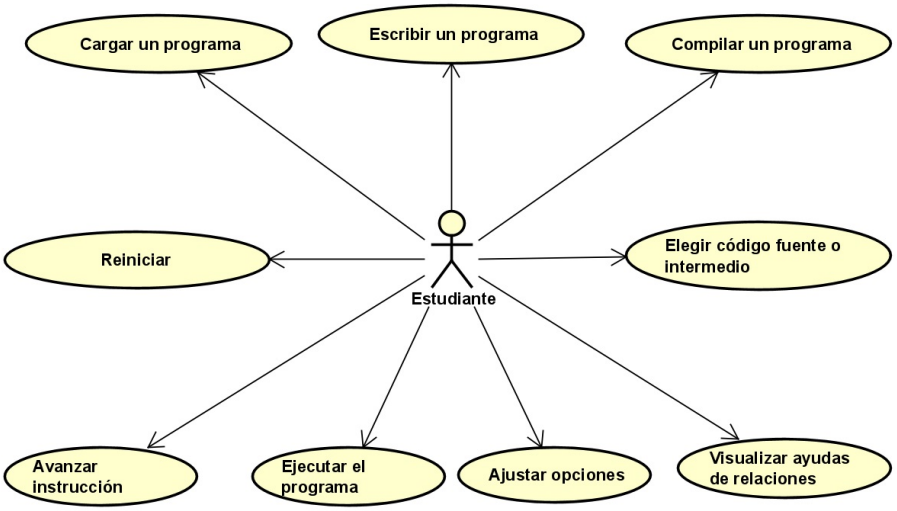

Figura 5.1: Historias de usuario

A continuación se desglosan todas las historias de usuario.

5.1. Historia HU01 - Escribir un programa

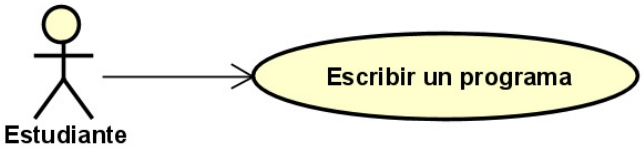

Figura 5.2: Escribir un programa

5.1.1. Descripción

Como: Alumno.

Quiero: Escribir un programa informático en lenguaje PFGUnedTLP.

Para: Poder compilarlo y así poder ver la evolución de los registros de activación y el estado del cómputo.

5.1.2. Criterios de aceptación

- El alumno puede escribir un programa en lenguaje PFGUnedTLP.
- El área de escritura para el código fuente solo está editable si no hay una simulación en curso.

5.2. Historia HU02 - Cargar un programa

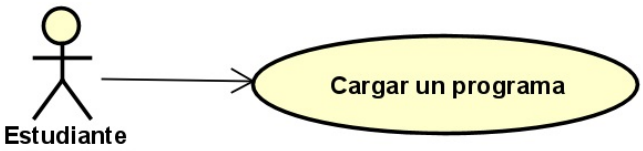

Figura 5.3: Cargar un programa

5.2.1. Descripción

Como: Alumno.

Quiero: Cargar un programa informático de ejemplo escrito en lenguaje PFGUnedTLP.

Para: Poder compilarlo y así poder ver la evolución de los registros de activación y el estado del cómputo.

5.2.2. Criterios de aceptación

- El alumno puede elegir mediante una desplegable un programa de ejemplo precargado.
- Si se pulsa en el botón cargar se produce un reinicio del sistema marcado en la **Historia HU04 - Reiniciar.**
- Se carga el ejemplo elegido en lenguaje PFGUnedTLP en el área de escritura para el código fuente.

5.3. Historia HU03 - Compilar un programa

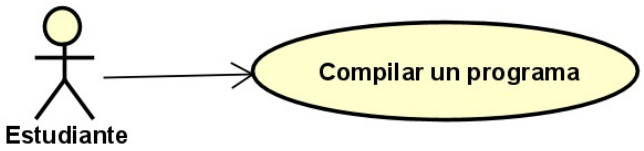

Figura 5.4: Compilar un programa

5.3.1. Descripción

Como: Alumno.

Quiero: Compilar un programa informático escrito en lenguaje PFGUnedTLP.

Para: Comprobar si hay errores en el programa y corregirlos en caso de que se produzca alguno. En caso de que no haya errores generar el código intermedio para así comenzar con la ejecución del programa.

5.3.2. Criterios de aceptación

- El programa debe estar escrito en lenguaje PFGUnedTLP.
- Se ejecuta el análisis léxico, sintáctico y semántico del programa.
- Si se encuentra algún error se muestra al alumno por pantalla.
- Si ejecuta el analizador sin errores se pasa a la fase de simulación.
  - Se muestra mensaje de ausencia de errores.
  - Se reinician e inicializan las estructuras internas de datos.
  - Se genera el código intermedio.
  - Se hace visible la funcionalidad de simulación del programa.

5.4. Historia HU04 - Reiniciar

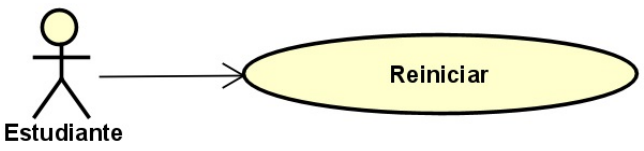

Figura 5.5: Reiniciar

5.4.1. Descripción

Como: Alumno.

Quiero: Reiniciar la aplicación.

Para: Comenzar desde cero y volver a escribir o cargar un programa informático escrito en lenguaje PFGUnedTLP.

5.4.2. Criterios de aceptación

- Se reinician e inicializan las estructuras internas de datos.
- Se hace editable la zona para escribir el código fuente.
- Se oculta la funcionalidad de simulación del programa.

5.5. Historia HU05 - Elegir código fuente o intermedio

Figura 5.6: Elegir código fuente o intermedio

5.5.1. Descripción

Como: Alumno.

Quiero: Elegir código fuente o intermedio.

Para: Poder elegir si quiero ver la evolución de los registros de activación y el estado del cómputo mientras consumo instrucciones de código intermedio o código fuente.

5.5.2. Criterios de aceptación

- Tiene que estar visible la funcionalidad de simulación del programa.
- Si se elige código fuente. dentro del área de simulación se hace visible el código fuente y se le quita la visibilidad al código intermedio.

- Si se elige código intermedio, dentro del área de simulación se hace visible el código intermedio y se le quita la visibilidad al código fuente.
- Si se elige una opción que ya está en uso no se hace nada.

5.6. Historia HU06 - Avanzar instrucción

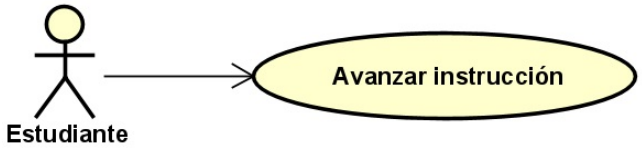

Figura 5.7: Avanzar instrucción

5.6.1. Descripción

Como: Alumno.

Quiero: Avanzar una instrucción.

Para: Consumir una instrucción ya sea de código fuente o código intermedio y ver como varían los registros de activación y el estado del cómputo después de procesar la misma.

5.6.2. Criterios de aceptación

- Tiene que estar habilitada la fase de simulación.
- Tiene que haber instrucciones pendientes de consumir.
- Si ya se ha comenzado la simulación se le quita la marca a la instrucción que se ha ejecutado con anterioridad.
- Se marca en azul la nueva instrucción a ejecutar.
- Se procesa la instrucción y se actualizan las estructuras internas de datos con los cambios que produce.
- Se actualiza la visualización de la pila de llamadas con los nuevos datos.
- Se actualiza la visualización de la pila de control con los nuevos datos.
- Se actualiza la visualización del estado del cómputo con los nuevos datos.
- Si era la última instrucción.
  - Se oculta el botón de ”Siguiente”.
  - Se oculta el botón de ”Ejecución completa”.
  - Se marca todo el código en azul como ejecutado.

5.7. Historia HU07 - Ejecutar el programa

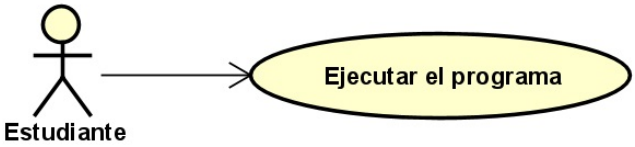

Figura 5.8: Ejecutar el programa

5.7.1. Descripción

Como: Alumno.

Quiero: Hacer una ejecución completa.

Para: Procesar todas las líneas de código ya sea intermedio o código fuente, y ver cómo queda el registro de activación y el estado del computo al finalizar la ejecución del programa.

5.7.2. Criterios de aceptación

- Tiene que estar habilitada la fase de simulación.
- Tiene que haber instrucciones pendientes de consumir.
- Se procesan secuencialmente todas las instrucciones pendientes de consumir, actualizando con cada una de ellas las estructuras internas de datos con los cambios que produce la instrucción.
- Se actualiza la visualización de la pila de llamadas con los nuevos datos.
- Se actualiza la visualización de la pila de control con los nuevos datos.
- Se actualiza la visualización del estado del cómputo con los nuevos datos.
- Se oculta el botón de "Siguiente".
- Se oculta el botón de "Ejecución completa".
- Se marca todo el código en azul como ejecutado.

5.8. Historia HU08 - Ajustar opciones

5.8.1. Descripción

Como: Alumno.

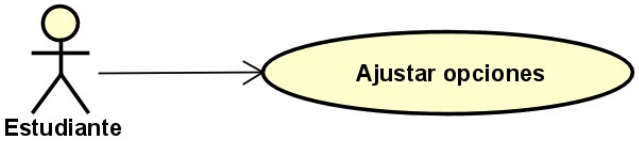

Figura 5.9: Ajustar opciones

Quiero: Ajustar opciones de ejecución.

Para: Poder visualizar los datos del registro de activación y estado del cómputo de la manera mas acorde con lo que necesito según el proceso que esté ejecutando o a lo que pretenda hacer el seguimiento.

5.8.2. Criterios de aceptación

- Tiene que estar habilitada la fase de simulación.
- Si se marca la opción "Mostrar temporales"se muestran en la visualización del estado del cómputo y de la pila de control las variables temporales. Es la pila de control, la visualización está supeditada a que no se esté mostrando el registro de activación reducido.
- Si se desmarca la opción "Mostrar temporales"se eliminan de la visualización del estado del cómputo y de la pila de control las variables temporales.
- Si se marca la opción "Pila Control. Mostrar registro de activación reducido"se quita de la visualización de la pila de control todo menos el enlace de acceso, enlace de control y valor de retorno de los registros de activación que contenga.
- Si se desmarca la opción "Pila Control. Mostrar registro de activación reducido"se visualizan todos los valores que contenga la pila de control.
- Si se marca la opción "Estado cómputo. Mostrar sólo variables visibles"se quitan de la visualización del estado del cómputo las variables marcadas con el campo visible igual a no.
- Si se desmarca la opción "Estado cómputo. Mostrar sólo variables visibles"se visualizan todas las variables que contenga el estado del cómputo.

5.9. Historia HU09 - Visualizar ayudas de relaciones

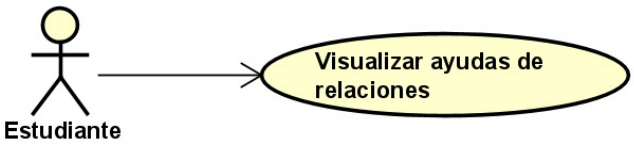

Figura 5.10: Visualizar ayudas de relaciones

5.9.1. Descripción

Como: Alumno.

Quiero: Visualizar ayudas.

Para: Para poder relacionar y comprender los diferentes valores almacenados tanto en la pila de llamadas, como en los registros de activación y el estado del cómputo.

5.9.2. Criterios de aceptación

- Tiene que estar habilitada la fase de simulación.
- Al pinchar en un registro de la pila de llamadas, pila de control o estado del cómputo se elimina de ellos las marcas previas que pudieran tener.
- Si se pincha en un registro de la pila de llamadas se marca en amarillo el registro seleccionado, y dentro de la pila de control y estado del cómputo también se marca en amarillo los registros pertenecientes a la llamada seleccionada.
- Si se pincha en un registro de la pila de control se marca en amarillo el registro seleccionado y dentro de la pila de llamadas la llamada a la que pertenece.

  Si además el registro seleccionado es un enlace de control o enlace de acceso se marca en rojo dentro de la pila de llamadas la llamada al registro de activación que enlazan y en la pila de control la posición donde comienza en registro de activación enlazado.

  También puede ocurrir que el registro seleccionado sea una variable o un parámetro, en cuyo caso se marca dentro del estado del cómputo en amarillo su valor.
- Si se pincha un registro del estado del cómputo se marca en amarillo dentro de la pila de llamadas la llamada a la que pertenece, y dentro de la pila de control, el registro de la pila donde se almacena siempre que este no esté oculto por opciones.

Capítulo 6

Tecnología utilizada

En este capítulo vamos a ver la tecnología utilizada en este proyecto. Esta tecnología viene condicionada por el requisito de que la aplicación sea un entorno web, lo que nos permite descartar directamente otras tecnologías (como pudiera ser Java o C++) sin ni siquiera considerarlas.

6.1. Lenguajes programación

HTML 5

HTML 5 (figura 6.1) (HyperText Markup Language, versión 5) es la quinta revisión del lenguaje HTML que sirve para construir estructuras dentro de una página web. En el proyecto se usa para construir la estructura de la web que va a servir al alumno de entorno de aprendizaje.

Figura 6.1: Logotipo de HTML 5.

CSS 3

CSS 3 (figura 6.2 es un lenguaje que permite definir y cambiar las presentación estética dentro de un documento HTML. En nuestro proyecto se usa para definir la estética de las páginas web y cada uno de sus elementos en cuanto a disposición, color, forma, tamaño, etc..

Figura 6.2: Logotipo de CSS 3.

JavaScript

JavaScript (JS) (figura 6.3) es un lenguaje de programación interpretado que se usa para proporcionar interactividad a las páginas web. En nuestro proyecto se usa tanto para el desarrollo del todo lo referente al desarrollo del compilador y manejo del mismo por el alumno.

Figura 6.3: Logotipo de JavaScript.

Una posible alternativa a este lenguaje, sería PHP[1], que presenta un inconveniente fundamental, ya que se ejecuta en el servidor y no en el navegador del cliente. Esto haría más complejo el desplegado del entorno, por lo que se descartó.

6.2. Librerías y frameworks

jQuery

jQuery[2] (figura 6.4) es una librería JavaScript que simplifica la manipulación de elementos HTML que, unido a su facilidad de usar, es ideal para la manipulación dentro de objetos del DOM(Document Object Model). Durante el proyecto se usa para manejar elementos de la web a la que se va a conectar el alumno para desarrollar ejercicios.

Figura 6.4: Logotipo de jQuery.

JQuery numberedtextarea

jQuery linedtextarea plugin [3] es un plugin de jQuery con licencia MIT[4] que permite visualizar números en áreas de texto dentro de documentos HTML.

Dentro del proyecto, como podemos ver en la figura 6.5 se emplea en el área donde el alumno introduce su código fuente con el fin de simular el número de línea que aparece en los editores de texto y que sirven de ayuda cuando el compilador arroja algún error referenciando un número de línea en el código fuente.

[1] https://www.php.net/
[2] https://jquery.com/
[3] https://www.jqueryscript.net/form/lined-textarea.html
[4] https://www.licen.cc/es/licencias/mit/

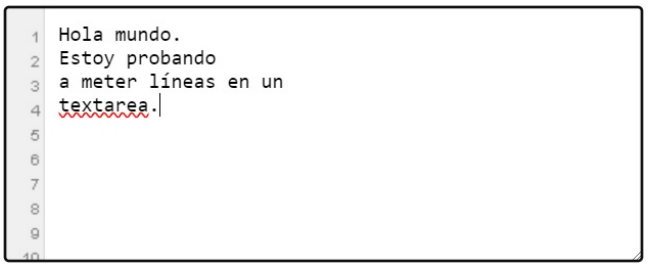

Figura 6.5: Recorte del editor de código fuente de la aplicación.

Node.js

Node.js [5] es un entorno en tiempo de ejecución para JavaScript construido con V8, motor de JavaScript de Chrome, es multiplataforma y que contiene todo lo necesario para ejecutar JavaScript fuera de un explorador.

En este entorno, mediante su gestor de paquetes por defecto npm se desplegará el entorno para usar Jison y sus scripts.

En la figura 6.6 podemos ver la imagen de la página principal de Node.js.

Figura 6.6: Página web Node.js.

Jison

Jison[6] (figura 6.7) es un entorno de ejecución con licencia MIT[7] que toma una gramática libre de contexto como entrada y genera un archivo JavaScript de salida capaz de analizar el lenguaje descrito por esa gramática.

El analizador que usa es LALR(1) y es muy similar a Bison, lo que permite incluir tanto reglas gramaticales como la propagación de atributos.

En el proyecto se usa para realizar en él el analizador léxico, sintáctico, semántico y creación de código intermedio que se produce al compilar el programa fuente que introduce el alumno.

[5] https://nodejs.org/es
[6] https://gerhobbelt.github.io/jison/docs/
[7] https://www.licen.cc/es/licencias/mit/

Figura 6.7: Página web de Jison.

6.3. Entornos de desarrollo

Notepad++

Notepad++[8] (figura 6.8) es un editor de código fuente gratuito que se rige por la licencia pública general GNU[9] al usarse en entornos Windows.

En el proyecto se usa para escribir la gramática que va a procesar Jison para generar el archivo JavaScript que usaremos como analizador.

Figura 6.8: Logotipo de Notepad++.

Visual Studio Code

Visual Studio Code, como podemos ver en la figura 6.9, es un editor de código fuente desarrollado por Microsoft que es multiplataforma. Permite la depuración de código, resalte de sintaxis, uso de innumerables plugins y está integrado con Github.

En el proyecto se usa para el desarrollo de la parte web tanto en lo relativo al interface con el alumno como el manejo del compilador.

[8] https://notepad-plus-plus.org/
[9] https://www.gnu.org/licenses/licenses.es.html

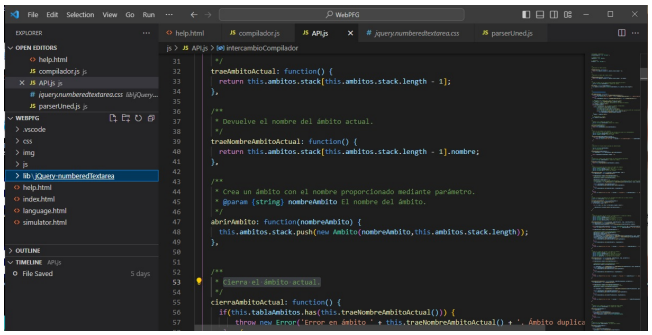

Figura 6.9: Captura de pantalla entorno desarrollo Visual Studio Code.

GitHub

GitHub[10] (figura 6.10) es un software que permite alojar proyectos utilizando el sistema de control de versiones Git.

En el proyecto se usa para el control de cambios y poder recuperar una situación anterior en caso de ser necesario.

Figura 6.10: Logotipo de GitHub.

6.4. Documentación

Overleaf

Overleaf[11] es un editor colaborativo de LaTeX que permite en su versión gratuita crear documentos con una alta calidad tipográfica.

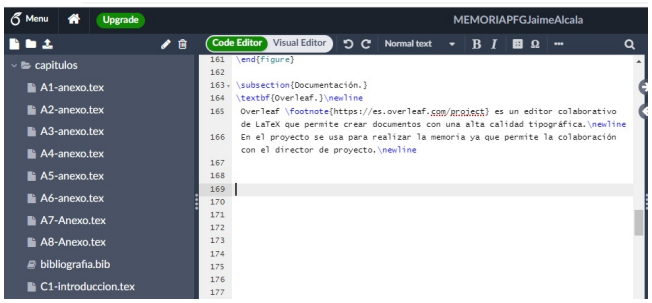

Figura 6.11: Captura de pantalla entorno Overleaf.

[10] https://github.com/
[11] https://es.overleaf.com/project

Como se puede observar en la figura 6.11 en el proyecto se usa para realizar la memoria ya que permite la colaboración con el director de proyecto.

Astah UML

Es una herramienta de modelado UML que permite a los diseñadores de software visualizar, especificar, construir y documentar un sistema.

En el proyecto se usa bajo la licencia student y como se puede ver en la figura 6.12 se usa para el diseño partes del sistema como son las historias de usuario o el modelo de datos.

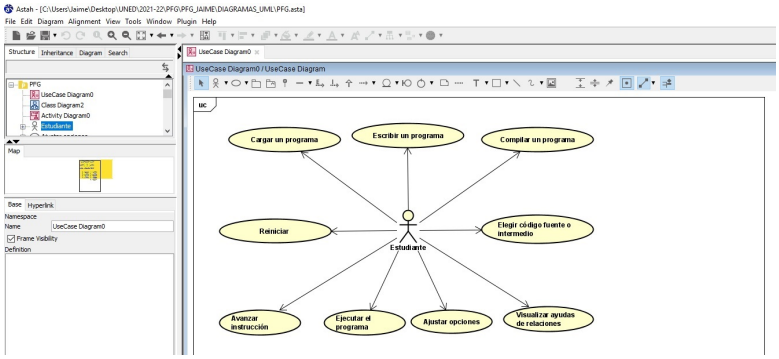

Figura 6.12: Captura entorno AstahUML

Excel

Excel es un programa multiplataforma desarrollado por Microsoft que permite editar hojas de cálculo.

En el proyecto bajo la licencia de estudiante UNED se usa para la creación de algunos artefactos usados por la metodología elegida como podemos ver en la figura 6.13.

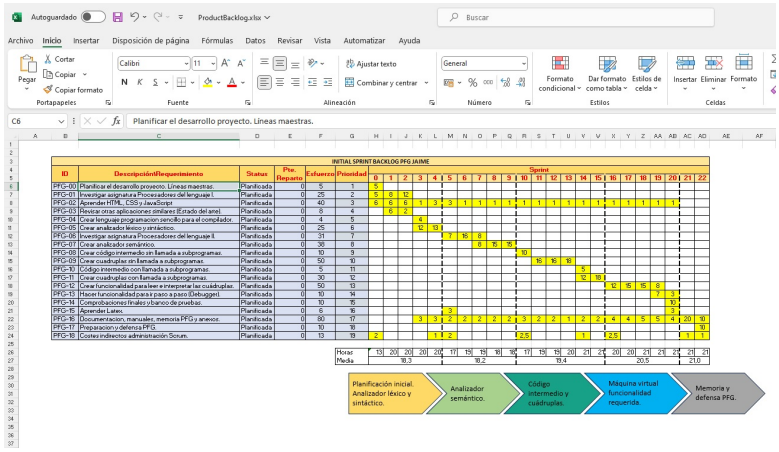

Figura 6.13: Captura desarrollo en excel del initial sprint backlog.

Paint

Paint es un editor de imágenes desarrollado por Microsoft que viene incorporado en el sistema operativo Windows.

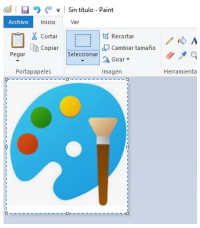

Figura 6.14: Captura uso de Microsoft Paint.

En el proyecto, como se puede ver en la figura 6.14, se utiliza para tratamiento de imágenes de la memoria.

Capítulo 7

Diseño y desarrollo del proyecto

Como se vio anteriormente, algunas de las historias de usuario interaccionan con el compilador, por lo que en este capítulo procederemos a diseñar el lenguaje y el compilador, así como a indicar su interacción con la interfaz.

En un primer momento se hace un análisis rápido de los diferentes lenguajes de programación que se usan como ejemplo en la asignatura de *Teoría de los lenguajes de programación* [5], así como de los ejemplos usados en las actividades de la misma asignatura para realizar ejercicios referentes a subprogramas y ambientes, llegando a la conclusión de que el lenguaje mas indicado para la realización del compilador sería Pascal, ya que su sintaxis es usada de forma recurrente en muchas de las asignaturas que se estudian en la carrera y a los alumnos les facilitaría mucho partir de un lenguaje conocido. Por otra parte, el lenguaje Pascal excede bastante la funcionalidad requerida para la realización el proyecto, por lo que finalmente decido, partiendo del una base fundamentada en el lenguaje Pascal, hacer mi propio lenguaje de programación que llamaré **PFGUnedTLP** de tal manera que sea bastante reducido pero que a su vez sea capaz de abordar todas las instrucciones necesarias para la consecución de los objetivos iniciales del proyecto respecto a los requerimientos del lenguaje, que son, un lenguaje de programación mínimo que admita declaración de variables, subprogramas anidados y recursión.

Para realizar nuestro lenguaje de programación tenemos que crear una gramática libre de contexto que nos defina el lenguaje.

Según [3], capítulo 2.2.1 una gramática libre de contexto tiene cuatro componentes:

1. Un conjunto de símbolos terminales, a los que algunas veces se les conoce como “tokens”. Los terminales son los símbolos elementales del lenguaje definido por la gramática.
2. Un conjunto de no terminales, a las que algunas veces se les conoce como “variables sintácticas”. Cada no terminal representa un conjunto de cadenas o terminales, de una forma que describiremos más adelante.
3. Un conjunto de producciones, en donde cada producción consiste en un no terminal, llamada encabezado o lado izquierdo de la producción, una flecha y una secuencia de terminales y no terminales, llamada cuerpo o lado derecho de la producción. La intención intuitiva de una producción es especificar una de las formas escritas de una instrucción; si el no terminal del encabezado representa a una instrucción, entonces el cuerpo representa una forma escrita de la instrucción.
4. Una designación de uno de los no terminales como el símbolo inicial.

Todas los aspectos relevantes referentes a la especificación del lenguaje se pueden ver en detalle dentro el anexo A *Definición del lenguaje.*

7.1. Diseño y desarrollo del analizador léxico

El desarrollo de un analizador léxico también denominado scanner es el primer paso para el desarrollo de un compilador. Como podemos ver en la figura 3.2 el análisis léxico da comienzo dentro de las fases de un compilador a la fase de análisis.

Según [3] capítulo 1.2.1, el analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma: (nombre-token, valor-tributo) que pasa a la fase siguiente, el análisis de la sintaxis.

El funcionamiento esperado para nuestro analizador léxico es el de un programa que recibe un flujo de caracteres, en nuestro caso el código fuente del programa, y lo divide en tokens o símbolos, los cuales los clasifica según una serie de reglas que le asignamos en las diferentes categorías, como pueden ser palabras reservadas, constantes, identificadores, comentarios, etc.

Después de una búsqueda de diferentes opciones, y según lo visto en las conclusiones estado del arte (ver sección 2.2), elegimos Jison para realizar el compilador y, por ende, para realizar nuestro analizador léxico. Este nos va a permitir, a partir de las especificaciones del lenguaje, generar un fichero Javascript que podemos usar desde nuestra web para compilar el programa fuente de nuestro lenguaje.

El código para realizar el analizador en Jison tiene muchas similitudes con CUP y Flex usados en las prácticas de Procesadores del Lenguaje I y Procesadores del Lenguaje II, por lo que la escritura de la gramática es muy parecida.

La forma de convertir nuestra gramática a código JavaScript capaz de analizar nuestro lenguaje una vez escrita la gramática, como podemos ver en la figura 7.1 , desde la línea de comandos ejecutamos el comando jison junto con el archivo donde hemos escrito la gramática, y si no arroja ningú error nos genera un fichero con el mismo nombre que el que le hemos dado al fichero que le hemos pasado con la gramática pero con extensión JavaScript.

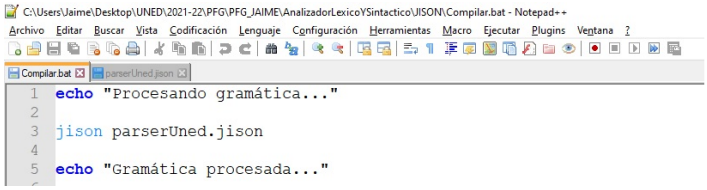

Figura 7.1: Ejecución Jison

Podemos encontrar todas las especificaciones léxicas del lenguaje dentro del anexo E Reglas léxicas.

Al finalizar el desarrollo del analizador léxico, podemos a través de este, descomponer el código fuente que introduce el alumno en lenguaje PFGUnedTLP en un flujo de tokens que son categorizados según su composición en palabras reservadas, identificadores y operadores. Además, podemos tener en cuenta la precedencia de los operadores mediante una reglas de precedencia que les marcamos y podemos excluir del flujo de tokens los comentarios que se han introducido en el código. Asimismo, al terminar la realización de este analizador léxico, el compilador es

capaz de mostrar errores si no es capaz de descomponer alguna parte del flujo de tokens de tal manera de nos muestre un error por pantalla con información necesaria para poder identificar y corregir el error.

En la figura 7.2 se muestra un ejemplo de un error léxico, ya que por diseño, un identificador no puede contener un guión bajo y no le encaja el flujo de tokens en ningún patrón de los que tiene en las especificaciones.

Figura 7.2: Error léxico. Los identificadores no pueden contener un guión bajo.

A continuación podemos ver una lista con la declaración de terminales que componen la gramática del lenguaje PFGUnedTLP, que son, precisamente, los que va a reconocer el analizador léxico.

Declaración de terminales

- ASIGNACION
- BEGIN
- CADENATEXTO
- COMA

- DOSPUNTOS
- ELSE
- END
- EOF
- epsilon
- EXIT
- FUNCTION
- IDENTIFICADOR
- IF
- IGUAL
- INTEGER
- LITERALENTERO
- MAS
- MENOS
- PARDCHO
- PARIZDO
- PROCEDURE
- PROGRAM
- PTOYCOMA
- PUNTO
- THEN
- VAR
- WRITELN

7.2. Diseño y desarrollo del analizador sintáctico

El siguiente paso en la realización de nuestro compilador es el analizador sintáctico.

Según [3], capítulo 1.2.2, la segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens.

En definitiva, un analizador sintáctico es un programa que recibe este flujo de tokens en el que se han dividido en las diferentes categorías de elementos que componen nuestro lenguaje PFGUnedTLP, y es capaz de comprobar que cumplen entre ellos una serie de reglas sintácticas que le hemos definido.

Jison, además de las reglas léxicas que hemos visto en el apartado anterior, permite añadir reglas sintácticas, de tal manera, que se puede construir con él todas las reglas sintácticas que componen el lenguaje PFGUnedTLP.

A continuación podemos ver una lista con la declaración de no terminales que componen la gramática del lenguaje PFGUnedTLP.

Declaración de no terminales

- cuerpoBloque
- decFuncion
- decParametros
- decProcedimiento
- decVariable
- decVariables
- expLlamadaFuncion
- expLlamadaProcedimiento
- expresionAritmetica
- expresionLogica
- expresionParametros
- inic
- inicioFuncion
- inicioProcedimiento
- inicioPrograma

- instrucciones
- listaFuncionesProcedimientos
- listaVariables
- llamadaParametros
- procedimientoOFuncion
- retornoFuncion
- seccionVariables
- sentenciaAsignacion
- sentenciaSalida
- sentenciaSi
- sentenciaSinoOpcional
- unaSentencia

En el anexo F podemos encontrar todas las reglas de producción que hacen posible la construcción del lenguaje PFGUnedTLP.

En el siguiente esquema podemos ver la estructura general de un programa en lenguaje PFGUnedTLP. Como se puede observar en el mismo es un esquema muy sencillo en el que cada parte del programa tiene una zona bien definida para su ubicación.

```
1
2  /*
3  Esquema general lenguaje PFGUnedTLP
4  */
5
6  Program NombreDelPrograma;
7      //declaracion de variables
8      //declaracion de funciones y procedimientos que permiten ser anidados.
9  begin //del programa principal
10    //sentencias
11 end.
```

También podemos ver a continuación, el esquema general de los procedimientos y funciones, los cuales a su vez pueden tener declarados internamente otros procedimientos o funciones ya que según el esquema definido en en análisis sintáctico se permite su anidación.

```
1
2  /*
3  Esquema general procedimiento y funciones.
4  Se permite funciones anidadas.
5  */
6
7  procedure/function NombreProcedimientoOFuncion(parametros):
8                                                  valorRetornoSiFuncion;
9      //declaracion de variables
```

```
10        //declaracion de funciones y procedimientos que permiten ser anidados.
11    begin /del procedimiento o funcion
12        //sentencias
13        exit(variableRetorno);//solo en funciones
14    end;
```

Una vez finalizada la fase de construcción de este analizador sintáctico, el programa es capaz de distinguir todas las estructuras sintácticas que tiene que tener nuestro lenguaje PFGUnedTLP y mostrar por pantalla cualquier información referente al no cumplimiento de estas reglas a modo de error, con información suficiente para poder identificar y corregir el error.

En la siguiente figura 7.3 podemos ver un ejemplo de la identificación de un error sintáctico en el cual el compilador se encuentra la palabra reservada *var* en un lugar del código fuente donde no puede estar según las reglas sintácticas que tiene cargado el lenguaje.

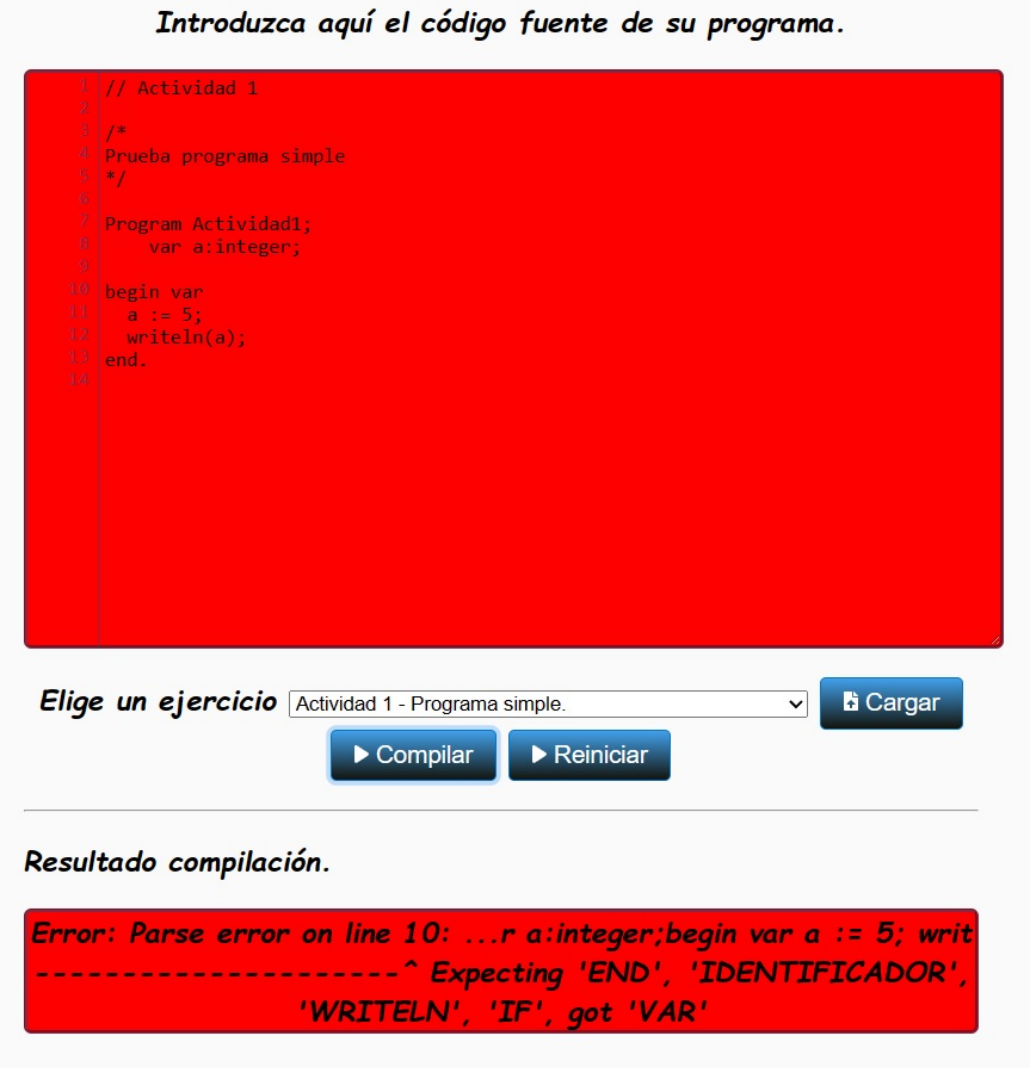

Figura 7.3: Error sintáctico.

7.3. Diseño web. Fase compilación

Con el analizador sintáctico finalizado, ya podemos generar a través de Jison un fichero en JavaScript que hace la función de compilador, el cual, pasándole fichero de texto, ya nos identifique si corresponde a la estructura de un programa fuente en lenguaje PFGUnedTLP, y a su vez nos devuelva los posibles errores léxicos y sintácticos que contiene el programa para corregirlos.

Es en este momento dónde es necesario comenzar a dar los primeros trazos al interfaz web de usuario donde el alumno pueda, una vez introducido su programa, ver mediante un mensaje que nos devuelva el parser la información referente a los errores que contiene nuestro código fuente o una confirmación de que el programa está generado correctamente respecto a las especificaciones léxicas o sintácticas.

El primer prototipo como se puede ver en la figura 7.4, es un diseño muy simple que contiene sólo lo necesario para realizar su cometido, que en estos momentos es un área donde el usuario pueda escribir un programa con la estructura del lenguaje PFGUnedTLP, un label donde el compilador nos va a devolver el resultado de la compilación, y un botón compilar.

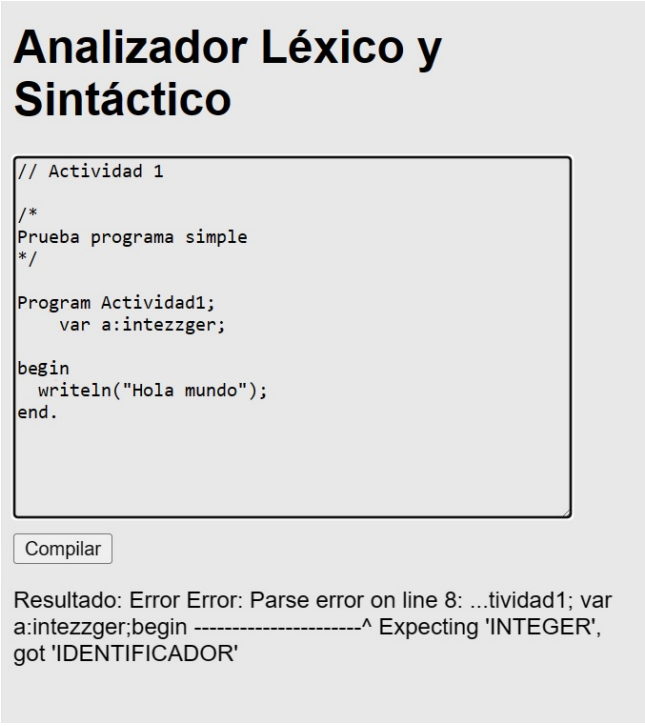

Figura 7.4: Prototipo de fase de compilación.

Cuando el botón *Compilar* sea pulsado, se va a enviar el código fuente al fichero JavaScript generado por Jison, el cual, una vez procesado de acuerdo a las reglas léxicas y sintácticas, nos va a devolver un mensaje si ha encontrado algún error.

7.4. Diseño y desarrollo de las acciones semánticas

En la siguiente fase comenzamos a tratar todo lo relacionado con las reglas semánticas.

Las reglas semánticas son una serie de acciones y comprobaciones que le tenemos que asignar al compilador para tratar todo lo relacionado con la creación de los diferentes ámbitos del programa, declaración de tipos, declaración de símbolos, construcción de las reglas semánticas, propagación de la información y manejo de errores semánticos que necesitamos controlar para el correcto funcionamiento del programa.

El desarrollo de esta fase se ha hecho usando Jison como en las fases anteriores, ya también nos permite realizar desde él el manejo de toda la parte semántica.

- Ámbitos del programa.

  El primer paso que tenemos que dar dentro del analizador semántico es definir los ámbitos del programa. Un ámbito de una variable, función o procedimiento es la región donde estos existen o desde donde se pueden acceder o ser llamados.

  En el lenguaje PFGUnedTLP, según lo hemos diseñado, sólo necesitamos crear ámbitos nuevos al llamar al programa principal, y al hacer llamadas a procedimientos y funciones, ya que no está permitido crear ámbitos diferentes dentro de un mismo programa principal, procedimiento o función.

  Dentro de cada ámbito del lenguaje PFGUnedTLP, tenemos la información necesaria sobre él, que son el nombre del ámbito y los símbolo que contiene. Además, dentro de este ámbito del lenguaje, podemos encontrar mas información que, aunque redundante, hace que mas fácil el manejo de la estructura.

  Estos ámbitos se almacenan a su vez dentro de una estructura de tipo pila donde se van apilando y desapilando según se van creando ámbitos o cerrándolos en nuestro código fuente. Las funciones que usamos dentro de nuestro compilador para crear y destruir estos ámbitos son:

  - **abrirAmbito: function(nombreAmbito)**
    Crea un ámbito con el nombre proporcionado mediante parámetro.
  - **cierraAmbitoActual: function()**
    Función que cierra el ámbito actual.

  En la figura 7.5 podemos observar el detalle de la composición de un ámbito.

  El desarrollo completo de estas funciones las podemos encontrar en el código fuente de la aplicación dentro del fichero API.js.

- Declaración de tipos.

  Otra de las tareas fundamentales dentro de un compilar es la declaración de tipos. Debido a la simplicidad de nuestro lenguaje PFGUnedTLP, esta declaración de tipos se ha reducido drásticamente hasta el punto de que sólo es necesario un único tipo que es el tipo entero.

Figura 7.5: Estructura de un ámbito.

- Declaración de símbolos.

  Otro de los puntos fundamentales en la construcción de un compilador es la llamada tabla de símbolos. Un símbolo contiene información relevante de cada variable, procedimiento o función respecto a su ubicación, ámbito al que pertenece y tipo de dato al que pertenece.

  Las funciones principales para la creación de estos símbolos son:

  - **nuevoSimboloVariable: function(noLinea, nombreVariable, tipo)**
  - **nuevoSimboloParametro: function(noLinea, nombreParametro, tipo, nombreFuncOProc)**
  - **nuevoSimboloFuncion: function(noLinea, nombreFuncion, tipo, parametros)**
  - **nuevoSimboloProcedimiento: function(noLinea, nombreProcedimiento, parametros)**

  De esta manera, una vez creamos el ámbito, se le van añadiendo mediante estas funciones los distintos símbolos que están contenidos dentro del él.

  Podemos encontrar la definición completa de estas funciones dentro del fichero API.js incluido en el código fuente del programa .

- Comprobaciones semánticas.

  Una vez solucionada la gestión de tipos y símbolos, se necesita añadir una serie de comprobaciones semánticas para adecuar el uso del programa a los requerimientos del programa. Dentro del lenguaje PFGUnedTLP las comprobaciones semánticas que se realizan son:

  - Existencia/No existencia de identificadores en ámbitos.

  - Existencia y visibilidad de funciones y procedimientos a los que se llaman.

  - Correlación en número de parámetros y tipo[1] entre las funciones y procedimientos con sus respectivas llamadas.

  - Comprobación de existencia de sentencia return en funciones.

  - No se hacen comprobaciones de tipo ya que no son necesarias la haber un sólo tipo y declararse al abrir el programa principal.

  La definición completa de todas estas acciones también las podemos encontrar en el código fuente del programa dentro del fichero API.js.

[1] En el sentido de que una llamada a función debe ser considerada una expresión, mientras que una llamada a un procedimiento se trataría como una sentencia.

7.5. Diseño del registro de activación

Llegados a este punto, y antes de comenzar con la fase de realización de código intermedio hay que especificar el diseño de la pila de control, y para ello hay que diseñar el registro de activación donde voy a almacenar las diferentes activaciones que se van a producir durante la realización de un programa.

En base a las especificaciones iniciales, el sistema debe soportar recursión y anidamiento de funciones, por lo que tenemos que ir claramente a un diseño de registro de activación en los que se puedan almacenar entre otros en enlace de control y el enlace de acceso. Aparte necesitamos almacenar el valor de retorno, para conocer lo que devuelven las funciones, estado de la máquina para poder restaurar el estado al cerrar un registro de activación, dirección de retorno a la que volver una vez eliminemos un registro de activación, y posiciones para almacenar los parámetros, variables y temporales que se usen en cada registro de activación.

Como vimos en las conclusiones del estado del arte 2.2, voy a basar la pila de control en el modelo que se usa en ENS2001[4].

La pila comienza en la posición 65535 y evoluciona de forma decreciente, por lo que usamos el registro índice .IX como puntero marco.

Por simplicidad elegimos el mismo diseño del registro de activación tanto para la llamada principal como para las diferentes llamadas a procedimientos y funciones.

En la tabla 7.1 podemos ver un esquema del diseño del registro de activación usado para el proyecto.

	DESCRIPCIÓN
# 0[.IX]	Valor de retorno.
# - 1 [.IX]	Estado de la máquina.
# - 2 [.IX]	Enlace de control.
# - 3 [.IX]	Enlace de acceso.
# - 4 [.IX]	Parámetros actuales.
# - 4 - Num. Parámetros[.IX]	Dirección de retorno.
# - 5 - Num. Parámetros[.IX]	Variables locales.
# - 5 - Num. Parámetros - Num. variables[.IX]	Temporales.

Tabla 7.1: Esquema de diseño del registro de activación.

La definición de los elementos que la componen es la siguiente:

- **Valor de retorno.**

  Almacena el valor de retorno de una función.

- **Estado de la máquina.**

  Copia de los registros antes de la llamada para restaurarlos después.

- **Enlace de control.**

  Puntero al registro de activación del subprograma llamante.

- **Enlace de acceso.**

  Puntero al registro de activación del subprograma padre.

- **Parámetros actuales.**

  Parámetros de la función o procedimiento.

- **Dirección de retorno.**

  Dirección de código intermedio a la que se debe de saltar cuando se acaba la ejecución de la función o procedimiento llamado.

- **Variables locales.**

  Variables de la función o procedimiento.

- **Temporales.**

  Variables temporales de la función o procedimiento.

7.6. Diseño y desarrollo del código intermedio

Finalizada la etapa semántica continuamos con la elaboración del código intermedio.

Como podemos ver en la figura 3.2 el comienzo del código intermedio da a su vez paso, dentro de las fases de un compilador a la fase de síntesis.

QUADRUPLA	PARÁMETROS			DESCRIPCIÓN
STARTGLOBAL	null	null	null	Crea R.A del procedimiento principal.
STARTSUBPROGRAMAP	null	null	null	Crea R.A del procedimiento.
STARTSUBPROGRAMAF	null	null	null	Crea R.A de la función.
VAR	param1	param2	null	Inicializa param1 con el valor param2.
PUNTEROGLOBAL	param1	param2	null	Marca la posición final del R.A global.
PUNTEROLOCAL	param1	param2	null	Marca la posición final del R.A local.
MV	param1	param2	null	param1 = param2
MVA	param1	param2	null	param1 = &param2
MVP	param1	param2	null	param1 = *param2
STP	param1	param2	null	*param1 = param2
ADD	param1	param2	param3	param1 = param 2 + param3
SUB	param1	param2	param3	param1 = param 2 - param3
EQ	param1	param2	param3	param1 = (param2 = param3) ? 1:0
BRF	param1	param2	null	Si param1 salto a param2
BR	param1	null	null	Salto a param1
INL	param1	null	null	Inserta etiqueta param1.
WRITEINT	param1	null	null	Escribe param1.
WRITETXT	param1	param2	null	Escribe param2 y lo almacena en param1.
CALL	param1	null	null	Llama a la función o proc. param1.
FINSUBPROGRAMA	param1	null	null	Fin de procedimiento o función.
EXIT	param1	null	null	Devuelve param1. Retorno de función.
DEVCALL	param1	null	null	Elimina el R.A. de param1.
HALT	null	null	null	Parada. Fin de la ejecución.
Leyenda: R.A. = Registro de activación				

Tabla 7.2: Código intermedio. Juego de cuádruplas.

El código intermedio, como su propio nombre indica, es una representación intermedia entre el lenguaje de programación y el código máquina, de modo que genere una representación lineal de las instrucciones que tiene que realizar el compilador y a su vez sea mas fácil de traducir.

El desarrollo completo de las instrucciones completas de código intermedio que se ha usado para el proyecto lo podemos encontrar dentro del anexo B.

La segunda tarea importante dentro de la generación de código intermedio es la propagación del mismo, ya que el fin último de este es tener una representación secuencial de las instrucciones que tiene que ejecutar el compilador para llevar a cabo la ejecución del programa.

Esta propagación se realiza como en los apartados anteriores mediante Jison y la propagación de atributos que nos permite su operativa, lo que nos lleva al llevar a cabo la realización completa del análisis semántico a tener una lista ordenada de todas las cuádruplas que son necesarias para la correcta ejecución del programa, junto con su número de línea, ya que es un dato que nos va a ser muy útil a la hora de hacer la ejecución del programa.

7.7. Diseño web. Fase depuración

Una vez finalizada la fase de compilación y creado el código intermedio del programa que hemos introducido al sistema podemos entrar en la fase de simulación. Esto implica que nuestra web nos tiene que mostrar una funcionalidad capaz de albergar el código intermedio que acabamos de generar, las estructuras que contengan la pila de control y el estado del cómputo, y algún tipo de botón con el que ir consumiendo instrucciones para posteriormente generar el código final.

Según las conclusiones del estudio del arte (ver sección 2.2), el compilador Online GDB[2] tiene una interface gráfica de usuario con una disposición que cuadra bastante con los objetivos del proyecto, por lo que decido intentar realizar una disposición parecida de los elementos ajustándolo a lo que necesito mostrar por pantalla.

Bajo estos objetivos, genero una estructura que sólo se hace visible al realizar una compilación sin errores del programa fuente, la cual muestra en el centro de la pantlla el código intermedio o código fuente del programa que se está simulando. Al igual que Online GDB desarrollo en la parte izquierda una estructura de desplegables donde colgar las estructuras de datos que soportan el estado del computo y la pilla de control, una zona que emula las salidas por pantalla que nos puede ir dando nuestro código fuente. Por último un botón que pulsándolo, consume la siguiente instrucción que le toque de código.

Una vez conseguido todos estos objetivos, se realizan algunas mejoras para que el interface sea mas amigable, creando el botón para poder cambiar la simulación de código intermedio a código fuente y viceversa, el botón para finalizar la ejecución y se añade en los despegables también la pila de llamadas para que el alumno tenga mas información del proceso cuando haga la simulación y se dan los primeros toques de diseño para facilitar la visualización como podemos ver en el prototipo mostrado en la figura 7.6.

7.8. Diseño y desarrollo del código final

El proceso de generación del código final, como vimos en la figura 3.2, es el último paso en la fase de síntesis para la finalización de nuestro compilador.

[2] https://www.onlinegdb.com/

Figura 7.6: Prototipo diseño fase de simulación.

Consiste en traducir las instrucciones que hemos generado de código intermedio en instrucciones a más bajo nivel capaces de realizar todos los cambios necesario para que al completarse la ejecución tengamos el resultado correcto del programa sobre el que estamos haciendo la simulación. Esto implica acciones como la apertura y cierre de ámbitos, creación de una pila de llamadas, creación de la tabla de símbolos, creación de la pila de control con sus registros de activación, y toda la funcionalidad para que, una vez leída cada instrucción de código intermedio, sea capaz de traducir y realizar los cambios pertinentes sobre las estructuras que hemos mencionado.

7.8.1. Estructuras de datos

Pila de control

Según [3], las llamadas a los procedimientos y los retornos de los mismos se manejan mediante una pila en tiempo de ejecución, conocida como *pila de control*. Cada activación en vivo tiene un registro de activación que también es conocido como marco dentro esta pila de control, con la raíz del árbol de activación en la parte inferior, y toda la secuencia de registros de activación en la pila que corresponde a la ruta en el árbol de activación que va hacia la activación en la que reside el control en ese momento. Esta última activación tiene su registro en la parte superior de la pila.

Siguiendo el diseño del registro de activación que habíamos visto con anterioridad 7.5, nuestra pila de control contendrá tantos registros de activación como activaciones vaya teniendo el programa, es decir, una activación del programa principal, y una activación por cada llamada a procedimiento o función que esté activa en cada momento.

La estructura que voy a seguir, como se ha desarrollado en [4], comenzará en la posición 65535 e irá creciendo hasta posiciones inferiores.

Dentro de JavaScript, se opta desarrollarla con una estructura de tipo map, que nos permite simular una pila y a su vez nos permite acceder de forma sencilla a todos sus elementos de forma rápida. Como clave se usa la dirección de memoria y como valor un campo de tipo entero

que contendrá el valor de la posición de memoria. De este modo podemos acceder q todos los métodos y atributos que os ofrece una estructura tipo map en JavaScript.

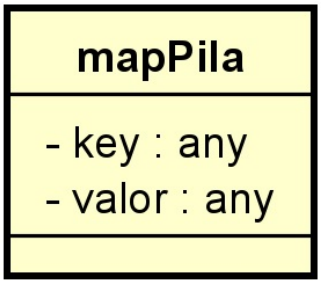

Figura 7.7: Estructura tipo map que simula la pila de control.

En la figura 7.7 podemos ver su estructura , donde usamos el campo key como dirección ya que no se puede repetir y el campo valor para guardar el contenido de la dirección de memoria de key.

Estado del cómputo

Por definición, según [2], el estado del computo es el valor de las variables de un programa en un momento dado y su función es asignar localizaciones de almacenamiento a valores.

Bajo estas premisas, para hacer la simulación del estado del cómputo se necesita una estructura de datos que nos permita almacenar tanto la localización de almacenamiento como el valor.

Se descarta desde un primer momento la opción de no usar ninguna estructura mas allá que la propia pila de control para mostrar en tiempo de ejecución solo las direcciones que pertenezcan a variables y parámetros. Se estudia en segundo lugar la posibilidad de hacerla con un map, al igual que la pila de control. Finalmente se opta por una estructura de tipo array de dos dimensiones de tal manera que en la misma posición i, el valor [i,1] corresponde a la posición de la variable en la pila de memoria y el valor [i,2] corresponde al valor almacenado en la posición de memoria indicada. Las posiciones de memoria se calculan por direccionamiento indirecto desde el inicio de la pila.

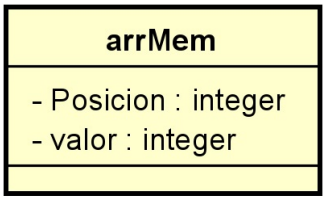

Figura 7.8: Estructura tipo array de dos dimensiones para almacenar el estado del cómputo.

A esta estructura de tipo array, que podemos ver en la figura 7.8, siempre se la alimenta mediante el método disponible en JavaScript para los arrays unshift,el cual mete el nuevo valor que está insertando siempre en la primera posición del array. Esto nos permite simular la inserción como si fuese una pila, y a su vez nos permite recorrer la estructura iterativamente tanto para mostrar el estado del cómputo al usuario, como para modificar y eliminar valores del array o usar cualquiera de los métodos creados en JavaScript para el manejo de arrays si fuese necesario.

Pila de llamadas

La pila de llamadas o call stack es una estructura de datos que almacena información sobre las llamadas activas que se han hecho durante la ejecución de un programa.

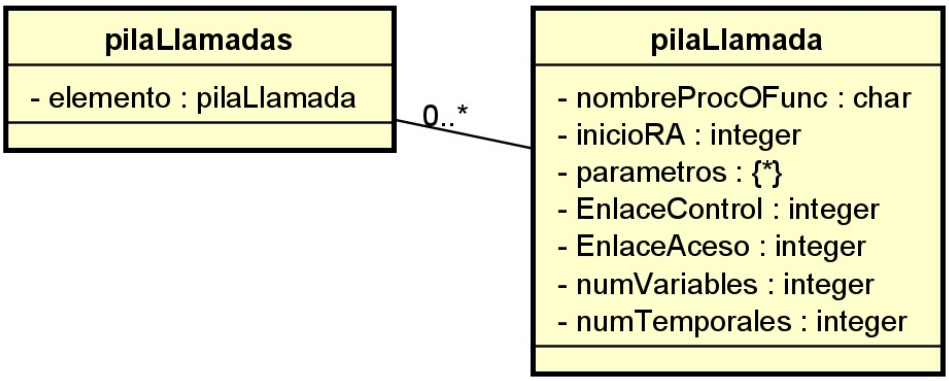

Figura 7.9: Diseño de la estructura de la pila de llamadas

Con motivo de facilitar la comprensión del alumno en el estudio de los registros de activación y estado del cómputo, se opta generar también la pila de llamadas ya que facilita ver de un modo mas fácil como cada llamada en la pila de llamadas genera un registro de activación.

Como en el anterior apartado se opta por una estructura de tipo array en la cual también se crean nuevos registros insertándolos en la primera posición, de tal manera simulamos la estructura LIFO de una pila.

Como se puede ver en la figura 7.9 cada posición del array está compuesta a su vez por una estructura pilaLlamadas que contiene una serie de campos necesarios o útiles para el desarrollo del proyecto.

7.8.2. Operaciones requeridas por el juego de cuádruplas

Una vez generadas las estructuras que van a soportar internamente los datos, y partiendo del código intermedio generado, vamos realizando las operaciones requeridas por el juego de cuádruplas 7.2

Para realizar todas las operaciones requeridas para la transformación del código intermedio en código al código final sobre las estructuras de datos descritas, se generan una serie de funciones que facilitan el desarrollo de las mismas. A continuación se describen las funciones realizadas.

- **function posPila()**

  Calcula la posición de la en la pila respecto al conador de programa.

- **function posPilaDI(pos)**

  Calcula la posición de la en la pila respecto a pos.

- **function posMem(pos)**

  Se le pasa un direccionamiento indirecto y devuelve la posición de memoria.

- **function posParametro(pos)**

  Calcula la posición del parámetro en la pila respecto al inicio de su registro de activación.

- **function recuperaValor(variable)**

  Recupera de arrMem el valor de la variable que se le pasa como parámetro.

- **function recuperaPosicionMemoria(variable)**

  Recupera de arrMem la posición de memoria que ocupa la variable que se le pasa como parámetro.

- **function recuperaValorCadena(cadena)**

  Recupera el valor de una cadena de texto.

- **function traeEnlaceDeControl()**

  Recupera el valor del enlace de Control de la ultima llamada activa.

- **function traeEnlaceDeAcceso(nombreProcOFunc)**

  Recupera el valor del enlace de acceso del procedimiento o funcion que se le pasa por parámetro.

- **function traeDireccionRetornoRA(nombreProcOFunc)**

  Recupera el valor de la dirección de retorno del registro de activación. Es el valor de la línea de código desde la cual se había hecho la llamada.

- **function traePosicionEtiqueta(etiq)**

  Recupera el valor de la variable que se le pasa como parámetro.

Por último, se realizan las tareas de paso de código intermedio a código final para hacer la simulación traduciendo las instrucciones de la siguiente manera:

- **STARTGLOBAL, STARTSUBPROGRAMAP y STARTSUBPROGRAMAF**

  Crea la pila de control e inicializa las posiciones de valor de retorno, estado de la máquina, enlace de control y enlace de acceso.

- **VAR**

  Crea e inicializa las posiciones de la variable en la estructura mapPila que simula la pila de control y en la estructura arrMem que simula es estado del cómputo.

- **PARAM**

  Crea en mapPila la posición del parámetro y le asigna el valor que lleva.

- **PUNTEROGLOBAL, PUNTEROLOCAL**

  Para el ámbito con el que se le llama guarda en arrMem los parámetros con la posición que les toca. Posteriormente garda lo temporales en mapPila y arrMem, se genera una nueva llamada en la pila de llamadas, y rellena el enlace de acceso y el de control del registro de activación.

- **MV**

  Asigna valor a una variable dentro de mapPila.

- **MVA**

  Asigna el contenido de una dirección de memoria a una variable dentro de mapPila.

- **STP**

  Al contenido de la dirección de memoria se le asigna un valor.

- **MVP**

  En mapPila recoge el valor de la dirección almacenada en el segundo parámetro y se lo asina a la posición de memoria del primer parámetro.

- **ADD**

  Guarda dentro de mapPila en la posición que ocupa el primer parámetro la suma de los contenido del los dos siguiente parámetros.

- **SUB**

  Guarda dentro de mapPila en la posición que ocupa el primer parámetro la resta de los contenido del los dos siguiente parámetros.

- **EQ**

  Comprara los valores del 2^{o} y 3^{o} parámetro. Si son iguales le asigno a mapPila un 1 al primer parámetro y sino un 0.

- **BRF**

  Si el valor del primer parámetro es cero salto a la posición de la etiqueta que viene en el segundo parámetro.

- **BR**

  Salto a la linea donde esté la etiqueta del primer parámetro.

- **INL**

  No hace nada en código final, solo para indicar etiquetas a donde saltar.

- **WRITEINT**

  Recupera el valor del primer carácter y muestra el resultado por la consola de salida mas un retorno de carro.

- **WRITETXT**

  Recupera el valor de la cadena y el contenido lo concatena a lo que ya había en la salida por pantalla. Para finalizar rellena un retorno de carro.

- **CALL**

  Rellena en mapPila la dirección de retorno del registro de activación y salto a la línea marcada por la etiqueta.

- **FINSUBPROGRAMA**

  Salto a la linea de la dirección de retorno del registro de activación.

- **EXIT**

  Rellena dentro de mapPila con el valor del primer parámetro la posición de inicio de registro de activación.

- **DEVCALL**

  Guardamos en temporal el valor de la salida del registro de activación. Eliminamos de arrMem los valores del registro de activación que cerramos. Eliminamos de mapPila los valores del registro de activación que cerramos. Eliminamos la llamada de la pila de llamadas.

- **HALT**

  Finalizamos la ejecución.

7.9. Diseño web. Fase final

Una vez llegados a este punto entramos en la fase final de nuestro proyecto, en la cual se introducen una serie de mejoras en el interfaz gráfico con motivo de ofrecer una mayor usabilidad por parte del usuario.

- **Opciones de filtrado**

  Se introducen en la parte de simulación una serie de opciones que permiten al alumno filtrar la vista de los resultados que va obteniendo durante el trascurso de la simulación. Esto permite centrar el foco en en sólo los datos necesarios según el objetivo de aprendizaje que esté buscando en la simulación.

  Las posibilidades de filtrado de los datos son 3:

  - Mostrar temporales.
  - Pila Control. Mostrar registro de activación reducido.
  - Estado cómputo. Mostrar sólo variables visibles.

  Se puede encontrar la descripción completa de estas opciones de filtrado dentro del manual de usuario C.

- **Ayudas visuales**

  Se introducen una serie de ayudas visuales que permiten al alumno relacionar de modo visual entre si los registros de la pila de llamadas, pila de control y estado del computo.

  Estas ayudas se ofrecen señalando con colores los diferentes tipos de relación que existen entre los datos obtenidos. Se puede encontrar la descripción completa de estas ayudas visuales dentro del manual de usuario C.

Capítulo 8

Estudio económico y viabilidad.

8.1. Coste estimado del desarrollos

Para comenzar el proyecto voy a hacer una estimación del coste para analizar la viabilidad del mismo y ver si se puede adaptar a las horas asignadas por el proyecto de fin de grado.

Esta estimación de coste se realizará empleando la ecuación del software de Putnam [6], desarrollada por Putnam y Myers en 1992, y que asume una distribución especifica del esfuerzo durante la vida de un proyecto. Esta ecuación se extrajo a partir del estudio de los datos de productividad de alrededor de 4.000 proyectos y trata proyectos destinados a fines comerciales.

Debido a las características del proyecto puede hacerse esta estimación como si el producto fuera destinado a fines comerciales.

La ecuación del software es un modelo dinámico multivariable que supone una distribución específica del esfuerzo a lo largo de la vida de un proyecto desarrollo de software.

Como se puede observar en la figura 8.1, para realizar esta estimación, se usa el método del cálculo de los puntos de función. En este método consiste en construir una tabla con el conteo estimado de valores de las entradas externas (EE), salidas externas (SE), consultas externas (CE), número de archivos lógicos internos (ALI) y número de archivos de interfaz externos (AIE).

La determinación de este conjunto de medidas es la siguiente.

- EE = 10. Botones, entradas de datos por pantalla y desplegable carga ejercicios.
- SE = 7. Salida mensajes compilación, 3 desplegables de pilas, salida por pantalla y 2 pantallas que muestran código.
- CE = 3. Ayudas al pinchar en los desplegables
- ALI = 3. Compilación, parser y API.
- AIE = 0. No afecta.

Este conteo de los valores del dominio de información se pondera en base a su complejidad, para obtener un valor final con del número de puntos de función obtenidos en la figura 8.1.

Esta determinación de la complejidad puede que sea un tanto subjetiva, pero se pretende realizar de la manera más objetiva posible para afinar lomas posible la estimación a la realidad.

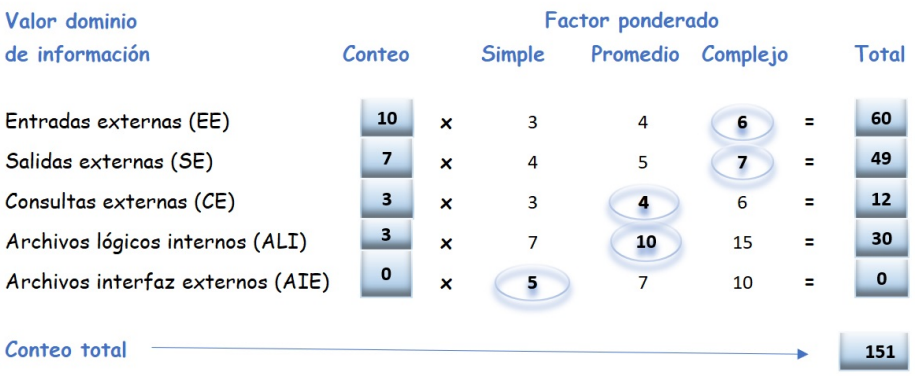

Figura 8.1: Estimación por cálculo de puntos de función

Este número de puntos de función calculado se pondera mas adelante en base a valores de ajuste de valor, que vienen dados por la valoración de una serie de características diferentes del proyecto.

Por último, para calcular los puntos de función se emplea la siguiente ecuación:

$$PF = conteoTotal \times [0,65 + 0,01 \times \sum_{i=1}^{14} F_i] \tag{8.1}$$

En esta ecuación, $\sum_{i=1}^{14} F_i$ se corresponde a los ajustes de valor, que se calculan haciendo una valoración en una escala que va desde de 0 (sin importancia) hasta a 5 (esencial):

- 0 = Sin importancia.
- 1 = Accidental.
- 2 = Moderado.
- 3 = Medio.
- 4 = Significativo.
- 5 = Esencial.

según la respuesta a cuestiones relacionadas con el desarrollo.

Las cuestiones y su valoración son:

1. ¿El sistema requiere respaldo y recuperación confiables? **0 puntos.**

   No es necesario.

2. ¿Se requieren comunicaciones de datos especializadas para transferir información hacia o desde la aplicación? **0 puntos.**

   No es necesario.

3. ¿Existen funciones de procesamiento distribuidas? **1 punto.**

   El software debe de ser accesible desde Internet, pero toda la funcionalidad se ejecutará en el cliente.

4. ¿El desempeño es crucial? **3 puntos.**

   Sí bastante relevante pero no es crucial que tenga un buen rendimiento. Se le asigna tres puntos.

5. ¿El sistema correrá en un entorno operativo existente enormemente utilizado? **5 puntos.**

   El sistema se podrá ejecutar sobre diferentes dispositivos y plataformas.

6. ¿El sistema requiere entrada de datos en línea? **3 puntos.**

   Sí, la entrada de datos la realizará el alumno en forma de programa.

7. ¿La entrada de datos en línea requiere que la transacción de entrada se construya sobre múltiples pantallas u operaciones? **0 puntos.**

   La entrada debería ser muy sencilla, se hace sobre un único formulario de entrada de código fuente.

8. ¿Los ALI se actualizan en línea? **0 puntos.**

   No es necesario.

9. ¿Las entradas, salidas, archivos o consultas son complejos? **5 puntos**.

   La entrada es complejas ya que se trata de escribir un programa.Las salidas y consultan también lo son ya que requieren de múltiples cálculos.

10. ¿El procesamiento interno es complejo? **5 puntos.**

    Sí, el proceso de compilación es complejo, y la máquina que interpreta el programa compilado también lo es.

11. ¿El código se diseña para ser reutilizable? **4 puntos.**

    Sí, el código a realizar se diseñará de modo que sea fácil de interpretar y facilite su mantenimiento.

12. ¿La conversión y la instalación se incluyen en el diseño? **0 puntos.**

    No, ya que se pretende que la instalación sea casi trivial.

13. ¿El sistema se diseña para instalaciones múltiples en diferentes organizaciones?**0 puntos.**

    No, se diseña exclusivamente para la UNED.

14. ¿La aplicación se diseña para facilitar el cambio y su uso por parte del usuario? **3 puntos.**

    Sí, se pretende que la aplicación sea fácil de usar, y fácil de mantener.

Sumando el total, obtenemos **29 puntos** de los factores de ajuste de valor (FAV).

Estimado el número de puntos de ajuste de valor ya podemos realizar el cálculo de los puntos de función.

$$PF = 151 \times [0,65 + 0,01 \times \sum_{i=1}^{14} F_i] \tag{8.2}$$

$$PF = 151 \times [0,65 + 0,01 \times 29] \tag{8.3}$$

$$PF = 151 \times [0,94] = 141,94 \approx 142 \tag{8.4}$$

Una vez estimado el número de puntos de función, necesitamos estimar el número de LOC (Lines Of Code) que tendrá el proyecto, ya que es una medida que se usa en la ecuación del software de Putnam.

El número de líneas de código a partir del número de puntos de función depende del lenguaje empleado para realizar la aplicación. Se estima que se compondrá de un 90 % de código JavaScript y un 10 % de código HTML y CSS.

En base a los resultados de "Function Point Languages Table" aportados por Quantitative Software Management, en su versión 5 [7], se obtienen los siguientes valores, tomando la media de líneas de código por cada uno de estos lenguajes:

- 1. JavaScript. 47 líneas de código / punto de función.
- 2. HTML. 34 líneas de código / punto de función.

Sobre CSS no existen datos en la tabla , pero dada su similitud con el lenguaje HTML, asimilamos que el valor de líneas de código / punto de función sea similar al HTML, por lo que tomamos el mismo número de líneas de código por punto de función que para el lenguaje HTML.

Ponderando en base al porcentaje esperado de cada lenguaje, se obtiene la estimación de las líneas de código del proyecto:

$$LOC = \text{ puntosDeFunción} \times (0,9 \times Avg(JavaScript) + 0,1 \times Avg(HTML, CSS)) \tag{8.5}$$

$$LOC = 142 \times (0,9 \times 47 + 0,1 \times 34) \tag{8.6}$$

$$LOC = 142 \times 45,7 = 6489{,}4 \tag{8.7}$$

El siguiente paso es calcular el tiempo mínimo de desarrollo y el esfuerzo en persona-mes.

Las ecuaciones para realizar estas estimaciones son:

$$t_{min} = 8,14 \times (\frac{LOC}{P})^{0,43} \tag{8.8}$$

$$E = (\frac{LOC \times B^{0,333}}{P})^3 \times \frac{1}{t^4} \tag{8.9}$$

Para obtener el resultado de ambas ecuaciones es necesario calcular un valor para los parámetros P y B por lo que ahora vamos a explicar el significado de cada uno de estos parámetros y el valor elegido.

- P. Parámetro de productividad, que refleja: madurez global del proceso y prácticas administrativas, la medida en la que se usan buenas prácticas de ingeniería de software, el nivel de lenguajes de programación utilizado, el estado del entorno de software, las habilidades y experiencia del equipo de software y la complejidad de la aplicación.

  Los valores que se usan son 2.000 para un software de tiempo real, 10.000 para software de comunicaciones y sistemas, 12.000 para software científico y 28.000 para aplicaciones comerciales de sistemas. Se decide asignar un valor de 20.000 teniendo presente que es un software que reúne características que podrían estar caballo entre software científico y un software de sistemas comerciales.

- B. Factor especial de destrezas. Para programas pequeños B vale 0.16, para programas intermedios vale 0.28, para programas grandes vale 0.39. Al tratarse de un proyecto pequeño, se le asigna el valor de 0,16.

Con estos datos, si procedemos al cálculo del tiempo mínimo de desarrollo:

$$t_{min} = 8,14 \times (\frac{LOC}{P})^{0,43} \tag{8.10}$$

$$t_{min} = 8,14 \times (\frac{6489}{20000})^{0,43} = 5,01 \text{ meses} \tag{8.11}$$

y al cálculo del esfuerzo en persona-año, teniendo en cuenta que 5,01 meses $\approx$ 0,42 años.

$$E = (\frac{LOC \times B^{0,333}}{P})^3 \times \frac{1}{t^4} \tag{8.12}$$

$$E = (\frac{6489 \times 0,16^{0,333}}{20000})^3 \times \frac{1}{0,42^4} \tag{8.13}$$

$$E = 0,176^3 \times \frac{1}{0,42^4} = 0,176 \text{ persona-año} \tag{8.14}$$

Si multiplicamos el valor por los meses que dura un año, se puede obtener el cálculo en persona – mes redondeando al valor entero más cercano.

$$E = 0,176 \times 12 = 2,112 \text{ persona-mes} \tag{8.15}$$

El esfuerzo obtenido tras este redondeo da como resultado 2,2 persona-mes.

Suponiendo una tarifa de mano de obra de un trabajador especializado en costes propios y de empresa son unos de 6.000 euros-mes, el coste de personal para el desarrollo de la aplicación resulta de multiplicar el esfuerzo en persona-mes por la tarifa de mano de obra:

$$C = 2,2 \text{ persona-mes} \times 6000 \text{ euros-mes} = 13200 \text{ euros} \tag{8.16}$$

Es esta estimación debe añadirse el coste de licencias de software empleadas, pero dado que todo lo que se pretende usar es gratuito, el coste del proyecto final se estima en 13.200 euros.

8.2. Estudio de coste de mantenimiento y explotación.

Para que la aplicación funcione de modo que los alumnos de la UNED puedan usarla durante el estudio de alguna de las asignaturas mencionadas a lo largo de esta memoria, es necesario que la aplicación se hospede en un servidor de alojamiento web.

Debido a la simplicidad del sistema en cuando a requerimientos, el alojamiento que se ajusta a nuestra funcionalidad es básico, ya que no tiene ningún requerimiento especial que pudiera hacer que necesitásemos un hosting mas avanzado.

De acuerdo a unos de los principales proveedores especializados en hosting que hay en España como es Arsys [8] [1] ofrece un plan de hosting profesional básico con las siguientes características:

- Espacio web 10GB.
- 1 Base de Datos MySQL de 1GB.
- Dominio 1 año Gratis.
- Tráfico ilimitado.
- Programación: PHP, Perl, Python.
- Compatible con WordPress, Joomla!, Prestashop...
- Un año de Basic SSL (ver otros SSL).
- 5 Cuentas de correo de 10 GB.
- Migrador de correo gratis.

Este hosting se ajusta perfectamente a nuestras necesidades y tiene un precio mensual de 6,90 euros al mes con un descuento del 56 % por lo que el primer año se nos quedaría una cuota de 3 euros al mes lo que nos da una cuota final total de 36 euros anuales el primer año incluyendo el registro del dominio.

A partir del segundo año habría que sopesar una serie de factores como la subida de precio del hosting y la acogida de la aplicación entre los estudiantes para ver si se continua con el alojamiento o por el contrario se cancela por desestimar la utilidad del aplicativo para el estudio y aprendizaje de la evolución de los registros de activación y el estado del cómputo durante la ejecución de un programa.

[1] https://www.arsys.es/hosting

Capítulo 9

Pruebas

En este capítulo se muestran las pruebas que se ha realizado durante el desarrollo del proyectos para comprobar que el resultado es el esperado.

La pruebas se dividen en 2 tipos pruebas de usuario 9.1 donde se validan las historias de usuario con sus criterios de aceptación y pruebas de algoritmos 9.2, donde se comprueban que los resultados mostrados por la aplicación concuerdan con el resultado esperado.

9.1. Pruebas de usuario

En esta sección se muestran las pruebas que se han ido realizando para validar que las historias de usuario cumplen con sus sus criterios de aceptación.

9.1.1. Prueba PHU01 - Escribir un programa

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU01 - Escribir un programa.

9.1.1.1. Pasos a realizar

- **Precondiciones**

  1. El área de escritura para el código fuente solo está editable si no hay una simulación en curso.

- **Pasos**

  1. Escribir líneas de código fuente hasta completar un programa en lenguaje PFGUnedTLP.

9.1.1.2. Resultado esperado

El área destinada a la escritura de código fuente contiene un lenguaje escrito en lenguaje PFGUnedTLP.

9.1.1.3. Resultado obtenido

Una vez el alumno ha escrito las líneas necesarias para completar su programa en el área destinada a contener el código fuente está escrito el programa en lenguaje PFGUnedTLP.

9.1.1.4. Evidencias

Situamos el cursor en el área destinada a escribir el programa en lenguaje PFGUnedTLP. El área está editable porque no hay ninguna simulación en curso.

Comenzamos a escribir el programa y una vez finalizado el programa aparece completo en el área destinada al efecto. Se efectúa una segunda prueba de para intentar escribir un programa con una simulación en curso pero no es posible debido a que el área destinada a la escritura del código fuente no está editable.

Figura 9.1: Programa escrito en lenguaje PFGUnedTLP.

9.1.2. Prueba PHU02 - Cargar un programa

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU02 - Cargar un programa.

9.1.2.1. Pasos a realizar

- **Precondiciones**

  No existen.

- **Pasos**

  1. Elegimos un ejemplo de programa de entre los que dispone precargado el sistema mediante el desplegable habilitado al efecto. Las descripciones de cada programa que aparecen en el desplegable ofrecen información necesaria para saber de que trata el ejemplo a cargar.
  2. Una vez elegido el ejemplo pinchamos en el botón cargar.

Figura 9.2: Carga Actividad 6 - Recursividad Funciones. Prog. Tetraédrico

9.1.2.2. Resultado esperado

Se carga el programa elegido en el área destinada a la escritura de código fuente listo para compilar y la apariencia del sistema es como si no hubiese ninguna simulación en curso.

9.1.2.3. Resultado obtenido

Se obtiene la carga el programa elegido en el área destinada a la escritura de código fuente listo para compilar y la apariencia del sistema es como si no hubiese ninguna simulación en curso.

9.1.2.4. Evidencias

Estando en una simulación en curso elegimos dentro del desplegable la *Actividad 6 - Recursividad Funciones. Prog. Tetraédrico* y al pinchar en el botón cargar se reinicia el sistema con el resultado final de que aparece se limpian los datos de todas las estructuras internas de la simulación que había en curso, se deshabilitan la funcionalidad de la fase de simulación, el área de trabajo se vuelve editable y con fondo blanco, y por último aparece cargado en el área destinada al código fuente el programa que hemos elegido,

9.1.3. Prueba PHU03 - Compilar un programa

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU03 - Compilar un programa.

9.1.3.1. Pasos a realizar

- **Precondiciones**

  1. Tiene que haber un programa escrito en el área destinada a escribir programas en lenguaje PFGUnedTLP.

- **Pasos**

  1. Comprobamos que el programa que quiero compilar está escrito en el área habilitada al efecto.
  2. Pinchamos en el botón compilar.

9.1.3.2. Resultado esperado

El sistema tiene que comprobar que el programa escrito tiene las características léxicas, sintácticas y semánticas. Si no pasa alguna de estas tres comprobaciones el área destinada al código fuente se vuelve de color rojo y el sistema debe mostrar un error en el área destinada a los mensajes del compilador con información suficiente para que el alumno sepa que ocurre y pueda corregir el problema y vuelva compilar para hacer de nuevo estas tres comprobaciones.

Si por el contrario pasa los tres tipos de análisis mencionados en el punto anterior, el sistema mostrará un mensaje que ponga que el resultado de la compilación es correcta, el área destinada a la escritura de código fuente se pone de color verde y no editable, se habilitará los controles

necesarios para iniciar la simulación e internamente se generará el código intermedio asociado al código fuente compilado.

9.1.3.3. Resultado obtenido

Al compilar un programa si este es incorrecto nos muestra el error por pantalla y nos pone el área de código fuente en rojo. Al compilar un programa si no hay errores el área de código fuente se vuelve no editable y de color verde, además nos muestra un mensaje que indica que la compilación es correcta, y se habilita la zona de simulación con el código intermedio generado.

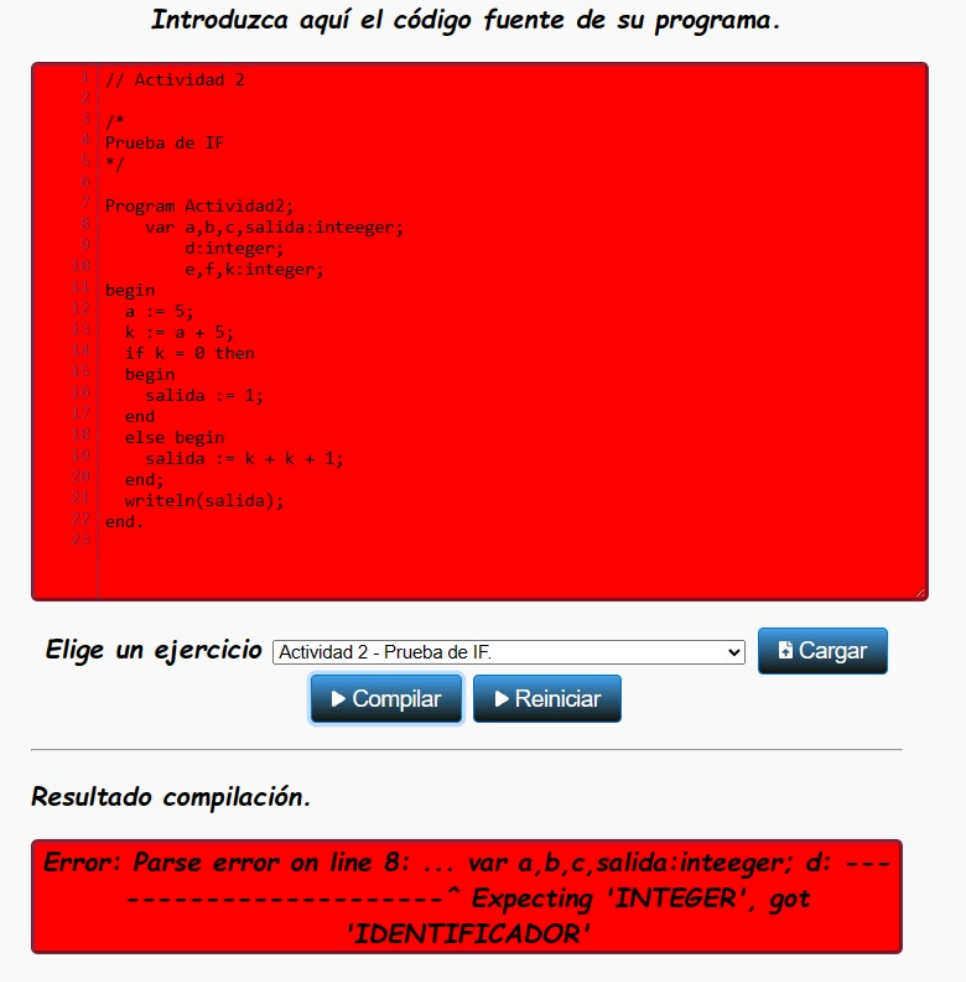

Figura 9.3: Resultado de compilación con error.

9.1.3.4. Evidencias

Se procede a compilar un programa con un error léxico que introducimos de modo voluntario en el programa y el área destinada a escribir código fuente se vuelve de color rojo y nos detecta el error que habíamos introducido de forma voluntaria.

Acto seguido corregimos el error y al compilar de nuevo nos aparece el mensaje de compilación

correcta, la zona donde se encuentra el código fuente se vuelve de color verde y no editable, y además se hace visible la funcionalidad para comenzar con la simulación. Podemos ver además que se ha generado el código intermedio que corresponde a lo esperado por el programa que acabamos de compilar y que el resto de estructuras internas que se usan para la simulación no contienen datos de posibles simulaciones anteriores.

Para comprobar que hace bien el reinicio antes de generar el código intermedio hacemos una segunda prueba avanzando un par de pasos en la simulación, donde ya hay datos en las estructuras que contiene el estado del cómputo, pila de llamadas y pula de control, y al pinchar en compilar vemos que está de nuevo todo como antes de avanzar ese par de pasos en la simulación.

9.1.4. Prueba PHU04 - Reiniciar

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario PHU04 - Reiniciar.

9.1.4.1. Pasos a realizar

- **Precondiciones**

  No existen.

- **Pasos**

  1. Pinchamos en el botón reiniciar.

9.1.4.2. Resultado esperado

El sistema tiene que inicializar todos las estructuras internas de datos eliminando así datos de una posible simulación en curso. Además tiene que ocultar la zona donde se muestra la funcionalidad de simulación y por último dejar el área destinada a la introducción del código fuente lista para que el alumno pueda introducir o cargar un nuevo programa.

9.1.4.3. Resultado obtenido

Una vez pulsado el botón reiniciar durante una simulación en curso, se oculta la zona habilitada para ejecutar la simulación, y el área donde se introduce el código fuente se queda lista para volver a cargar o escribir un nuevo programa.

9.1.4.4. Evidencias

Se comienza una simulación cargando la *Actividad 3- Invocación de un subprograma* y se avanza hasta el punto donde se hace la llamada al subprograma 9.4, estando las estructuras internas de

datos con datos referentes a la simulación. En ese momento pinchamos en reiniciar y el sistema hace un reinicio, con lo que se eliminan todos los datos internos de la simulación, se oculta toda la funcionalidad de la fase de simulación o sólo está visible y editable el área para introducir código fuente. Visualmente queda todo como al arrancar el programa.

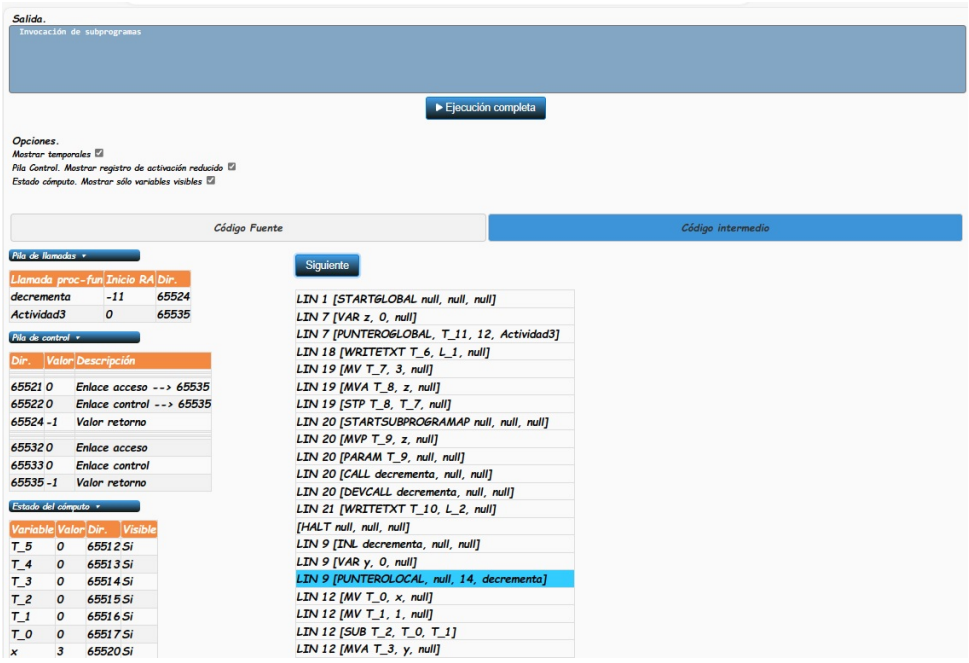

Figura 9.4: Simulación en curso. Llamada a un subprograma

Al comprobar por diseño las estructuras internas se puede ver que todas han sido inicializadas y como se puede ver en la figura 9.5 no queda ningún dato de la simulación que estaba en curso.

Figura 9.5: Reinicio realizado. Situación de las estructuras de datos.

9.1.5. Prueba PHU05 - Elegir código fuente o intermedio

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU05 - Elegir código fuente o intermedio.

9.1.5.1. Pasos a realizar

- **Precondiciones**
  1. Tiene que estar activa una simulación en curso ya que es un control específico de la fase de simulación.
- **Pasos**
  1. Si se pincha en el botón que activa el código fuente o intermedio.

9.1.5.2. Resultado esperado

Estando dentro de una simulación en curso al pinchar en el botón que cambia a código fuente o intermedio el sistema tiene que mostrar el código elegido dentro del área de simulación. También tiene que marcar estas líneas de código según el estado en el que se encuentre la simulación en ese momento.

9.1.5.3. Resultado obtenido

Encontrándome en una simulación en curso, al pinchar el botón de código intermedio muestra el código intermedio del programa y al volver a pinchar en código fuente volvemos a visualizar el código fuente. En ambos casos marca en azul la línea de código por donde va la simulación

9.1.5.4. Evidencias

Estando dentro de una simulación en curso con la simulación iniciada pincho en el botón para cambiar de código intermedio a código fuente, obteniendo como resultado que desaparece de la vista el código intermedio del programa y aparece el código fuente como se puede ver en la figura 9.6. La línea que marca en azul el sistema tiene el mismo número de línea que marcaba el código intermedio por lo que comprobamos que va por el mismo sitio. Al volver a pinchar de nuevo en código intermedio desaparece el código fuente y aparece el código intermedio marcando en azul la misma línea que tenía marcada la simulación antes del cambio a código fuente.

Hacemos una segunda comprobación de cambio de código intermedio a código fuente sin haber consumido ninguna instrucción y vemos que hace el cambio correcto en ambos sentidos pero sin poner ninguna marca en las líneas para marcar el lugar donde se encuentra la simulación en curso, lo cual es correcto ya que no se ha iniciado la simulación.

Como tercera comprobación hacemos el mismo cambio de código intermedio a código fuente y viceversa pero una vez consumidas todas las instrucciones y observamos que hace bien el cambio en ambos sentidos marcando en el mismo todas las instrucciones en azul, que es el indicativo de que se han consumido todas las instrucciones.

Por último se hace la prueba de elegir el mismo tipo de código que está visible y se comprueba que no varía nada en la visualización.

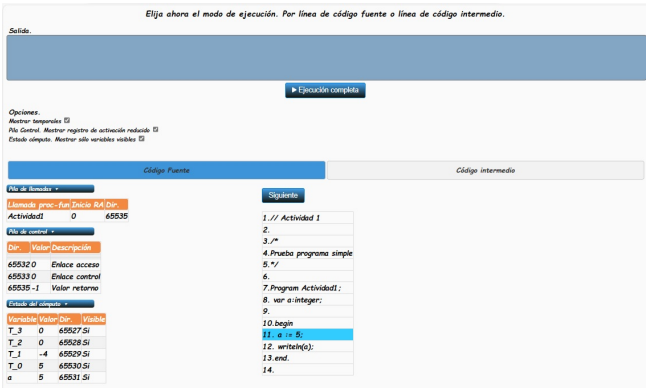

Figura 9.6: Cambio de código intermedio a código fuente en medio de una ejecución.

También se comprueba que el estado del resto de controles de la aplicación no varía al hacer este cambio entre los dos tipos de código.

9.1.6. Prueba PHU06 - Avanzar instrucción

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU06 - Avanzar instrucción.

9.1.6.1. Pasos a realizar

- **Precondiciones**
  1. Tiene que estar activa una simulación en curso ya que es un control específico de la fase de simulación.
  2. Tiene que haber al menos una instrucción pendiente de consumir en la simulación.
- **Pasos**
  1. Se pulsa en el botón siguiente para consumir la siguiente instrucción de código fuente o código intermedio.

9.1.6.2. Resultado esperado

Al consumir una instrucción el programa tiene que realizar internamente los procesos asignados a la misma de tal manera que se actualicen de acuerdo a ella todas estructuras de datos internas del sistema.

En cuanto a la visualización, se tiene que quedar actualizada la pila de llamadas, pila de control y estado del cómputo de acuerdo a la nueva situación, y además, marcada en azul la nueva instrucción consumida, salvo que no haya mas instrucciones por consumir en cuyo caso se queda todo el código marcado en azul, y dado que se ha llegado al fin de programa, se ocultan los botones de Siguiente y ejecución completa puesto que no hay nada que continuar.

9.1.6.3. Resultado obtenido

Al pinchar siguiente se consume la siguiente instrucción y se actualizan los datos y controles visibles por pantalla de acuerdo a la nueva situación obtenida.

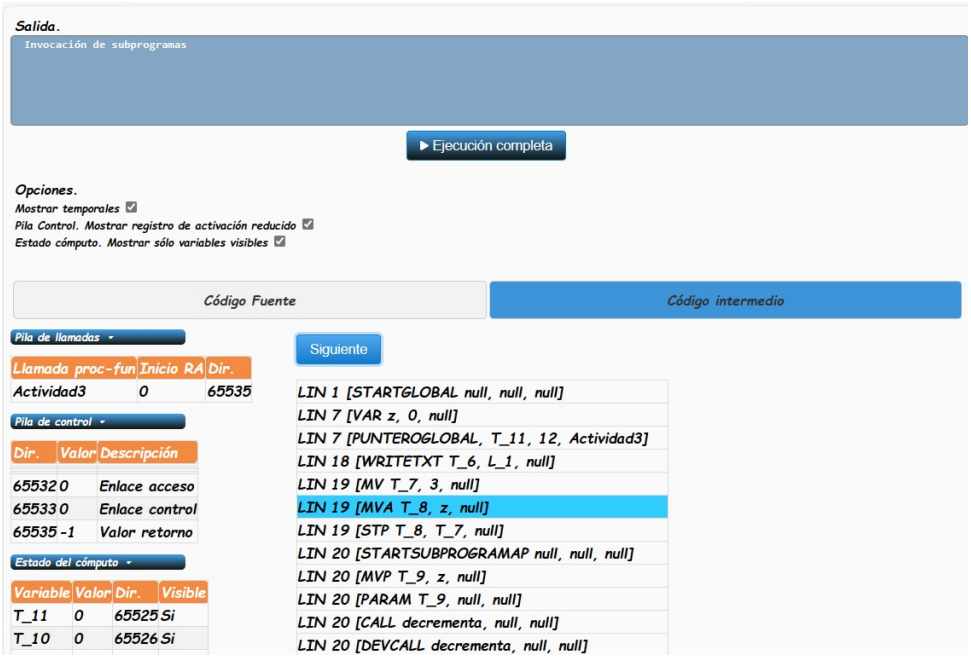

Figura 9.7: Simulación mediante el botón de siguiente instrucción.

9.1.6.4. Evidencias

Se comprueba que si no se ha entrado en la fase de simulación no se visualiza por pantalla el botón siguiente.

Se hace una segunda comprobación que, habiendo terminado la simulación, no se visualiza por pantalla el botón siguiente.

Por último se comprueba que siempre que hay una simulación en curso con instrucciones pendientes de consumir el botón siguiente está visible para su uso.

Después de esas comprobaciones nos situamos dentro de una simulación en curso y al pinchar en el botón siguiente, el proceso coge la siguiente instrucción que le toca procesar y realiza las tareas correspondientes a la instrucción. Visualmente vemos que la nueva instrucción que se ha marcado es la instrucción que tocaba como siguiente en el proceso. Los controles de datos se han actualizado con la información relativa a los cambios que ha producido esta instrucción consumida.

Al seguir consumiendo instrucciones y procesar la última instrucción, con esta vemos como se marcan todas las líneas de código en azul para indicar que están procesadas y se ocultan los botones siguiente y ejecución completa.

9.1.7. Prueba PHU07 - Ejecutar el programa

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU07 - Ejecutar el programa.

9.1.7.1. Pasos a realizar

- **Precondiciones**

  1. Tiene que estar activa una simulación en curso ya que es un control específico de la fase de simulación.
  2. Tiene que haber al menos una instrucción pendiente de consumir en la simulación.

- **Pasos**

  1. Se pulsa en el botón ejecución completa para consumir secuencialmente todas las instrucciones que queden pendientes de código fuente o intermedio.

9.1.7.2. Resultado esperado

El programa llega a su fin actualizando las estructuras de datos de acuerdo a las instrucciones procesadas, se marcan todas las instrucciones en azul para que se vean que están procesadas y se ocultan los botones que permiten consumir mas instrucciones como son el botón de siguiente y botón de ejecución completa.

9.1.7.3. Resultado obtenido

Una vez pulsando el botón de ejecución completa están marcadas todas las instrucciones en azul, están ocultos los botones de siguiente y botón de ejecución completa, y se han actualizado los valores de las estructuras de datos internas con el resultado de consumir secuencialmente todas estas instrucciones.

9.1.7.4. Evidencias

Se comprueba que si no se ha entrado en la fase de simulación no se visualiza por pantalla el botón ejecución completa.

Se hace una segunda comprobación que, habiendo terminado la simulación, no se visualiza por pantalla el botón ejecución completa.

Por último se comprueba que siempre que hay una simulación en curso con instrucciones pendientes de consumir el botón ejecución completa está visible para su uso.

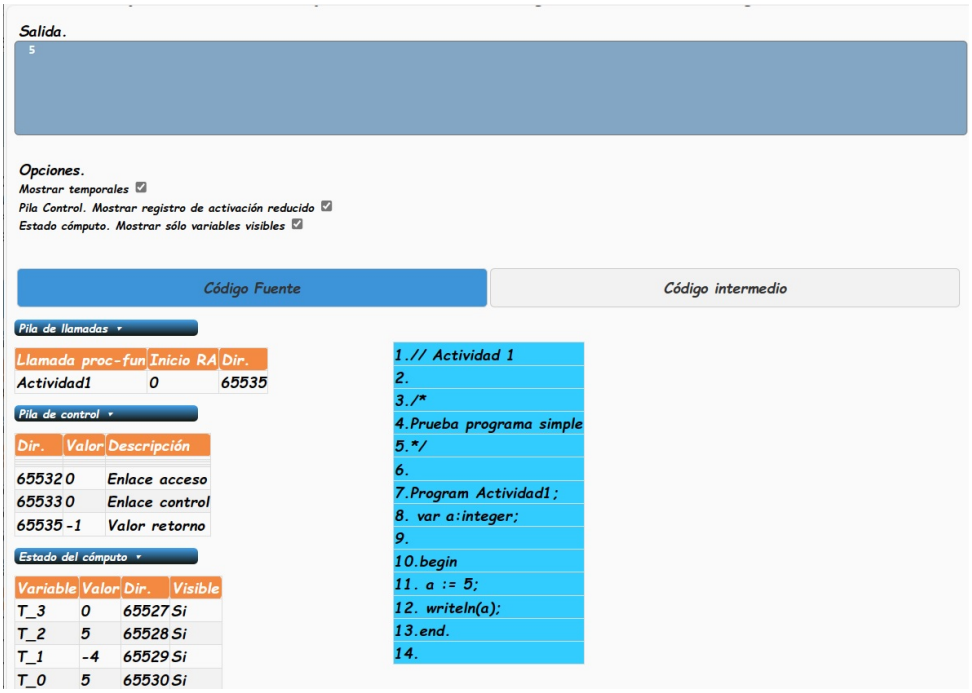

Figura 9.8: Resultado final de una simulación.

Después de esas comprobaciones nos situamos dentro de una simulación en curso y al pinchar en el botón ejecución completa, el proceso consumes instrucciones secuencialmente realizando las tareas correspondientes a la instrucción hasta que ya no quedan mas instrucciones por consumir. Visualmente vemos que al acabar se marcan todas las líneas de código en azul para indicar que están procesadas y se ocultan los botones siguiente y ejecución completa como se puede ver en la figura 9.8.

9.1.8. Prueba PHU08 - Ajustar opciones

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario PHU08 - Ajustar opciones.

9.1.8.1. Pasos a realizar

- **Precondiciones**

  1. Tiene que estar activa una simulación en curso ya que es un control específico de la fase de simulación.

- **Pasos**

  1. Se marca/desmarca alguna de las casillas de opciones de visualización y se tienen que filtrar los datos relacionados con la opción de filtrado de acuerdo con las las condiciones que marca.

9.1.8.2. Resultado esperado

Después de marcar o desmarcar alguna de las opciones, se tiene que actualizar la vista de los elementos que componen la opción de acuerdo a lo marcado.

9.1.8.3. Resultado obtenido

Se hace una prueba de todas las opciones posibles.

Al marcar la opción *Mostrar temporales* se muestran en la visualización del estado del cómputo y de la pila de control las variables temporales. Dentro de la pila de control, la visualización está supeditada a que no se esté mostrando el registro de activación reducido.

Al marcar la opción *Mostrar temporales* se eliminan de la visualización del estado del cómputo y de la pila de control las variables temporales.

Al marcar la opción *Pila Control. Mostrar registro de activación reducido* se quita de la visualización de la pila de control todo menos el enlace de acceso, enlace de control y valor de retorno de los registros de activación que contenga.

Al marcar la opción *Pila Control. Mostrar registro de activación reducido* se visualizan todos los valores que contenga la pila de control.

Al marcar la opción *Estado cómputo. Mostrar sólo variables visibles* se quitan de la visualización del estado del cómputo las variables marcadas con el campo visible igual a no.

Por último, al marcar la opción *Estado cómputo. Mostrar sólo variables visibles* se visualizan todas las variables que contenga el estado del cómputo.

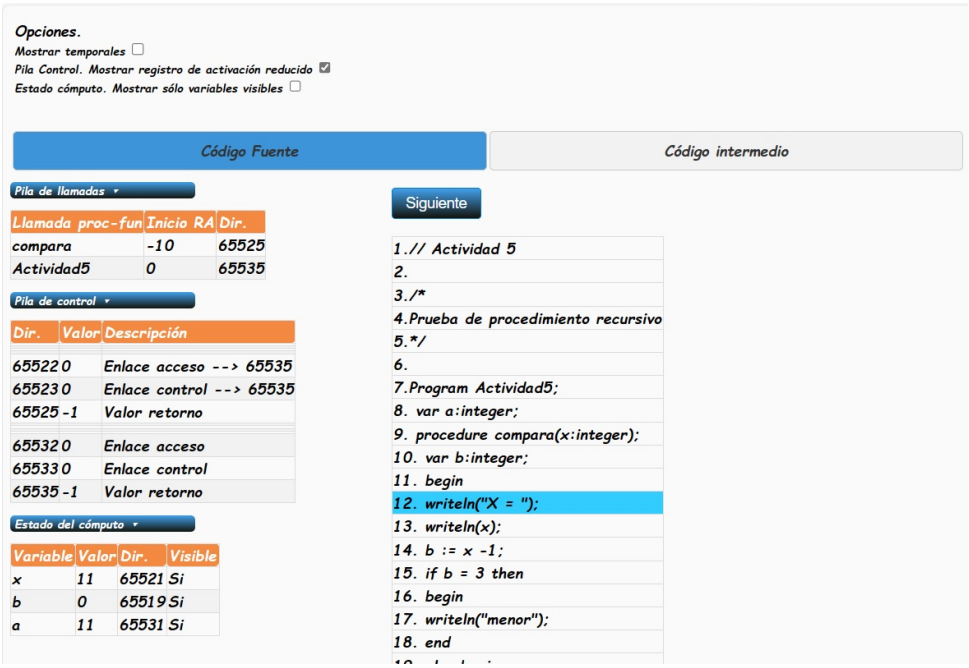

Figura 9.9: Ajuste de opciones de visualización en una simulación.

9.1.8.4. Evidencias

En medio de una simulación en curso, que ha activado varios registros de activación jugamos con las opciones de visualización para ir probando todas las opciones posibles para comprobar resultados.

Empezando con la *Mostrar temporales* marcada, vemos que hay una serie de registros en la pila de control y el estado del cómputo marcados como temporales, que, al desmarcar la opción, se oculta de la vista.

Hacemos una segunda prueba con *Mostrar temporales* activando la marca con la opción *Pila Control. Mostrar registro de activación reducido* y se ven los temporales en el estado del cómputo pero no en la pila de control.

Procedemos ahora con la opción *Pila Control. Mostrar registro de activación reducido*, al marcarlo nos aparecen varios registros de activación en los que sólo se ven dentro de la pila de control los valores del valor de retorno, enlace de acceso y enlace de control. Cuando le quitamos la marca a *Pila Control. Mostrar registro de activación reducido* se ven todos valores del registro de activación.

Por último probamos la opción *Estado cómputo. Mostrar sólo variables visibles*. Cuando la tenemos marcada dentro del estado de cómputo sólo se ven variables marcadas como visibles, pero si le desmarcamos el valor aparecen variables marcadas con el campo visible marcado con el valor no.

En la siguiente figura 9.9 se muestra el resultado en la visualización de una combinación de diferentes opciones.

9.1.9. Prueba PHU09 - Visualizar ayudas de relaciones

Con esta prueba vamos a comprobar la correcta realización de la historia de usuario HU09 - Visualizar ayudas de relaciones.

9.1.9.1. Pasos a realizar

- **Precondiciones**

  1. Tiene que estar activa una simulación en curso ya que es un control específico de la fase de simulación.

- **Pasos**

  1. Marcar algún registro de la pila de llamadas, pila de control o estado del cómputo.

9.1.9.2. Resultado esperado

Al pinchar en un registro de la pila de llamadas, pila de control o estado del cómputo, se eliminan todas las marcas que pudieran tener todos los registro de alguna visualización de relaciones anterior y se iluminan y/o colorean los registros relacionados con el registro marcado.

9.1.9.3. Resultado obtenido

Al marcar un registro de la pila de llamadas se marca en amarillo el registro seleccionado os relacionados con la pila de llamada que están dentro dela pila de control y estado del cómputo. Al pinchar en en un registro de la pila de control se marca en amarillo el registro seleccionado y dentro de la pila de llamadas la llamada a la que pertenece.

Al pinchar en un registro marcado como enlace de control o enlace de acceso se marca en rojo dentro de la pila de llamadas la llamada al registro de activación que enlazan y en la pila de control la posición donde comienza en registro de activación enlazado. Si el registro seleccionado es una variable o parámetro se marca dentro del estado del cómputo en amarillo su valor.

Por último, si se pincha un registro del estado del computo se marca en amarillo dentro de la pila de llamadas la llamada a la que pertenece, y dentro de la pila de control, el registro de la pila donde se almacena siempre que este no esté oculto por opciones.

9.1.9.4. Evidencias

Estando dentro de una simulación en curso se marca un registro de la pila de llamadas el cual se pone en amarillo, al mismo tiempo que este, se quedan marcados dentro de la pila de control y también en amarillo los registros relacionadas con la pila de llamadas que habíamos marcado. También se han marcado en amarillo dentro del estado del computo los registros pertenecientes a la llamada seleccionada.

Seguidamente pinchamos registros en la pila de control con el resultado de que siempre los registros marcados anteriormente con colores se quedan sin sus marcas y se marca en amarillo el nuevo registro registro seleccionado a la par que se marca dentro de la pila de llamadas la llamada a la que pertenece.

A continuación pincho en un registro marcado como enlace de control y se marca en rojo dentro de la pila de llamadas la llamada al registro de activación que enlazan y en la pila de control la posición donde comienza en registro de activación enlazado. El mismo resultado lo obtengo al pinchar en un registro marcado como enlace de acceso como se puede ver en la figura 9.10.

Como última prueba dentro de la pila de control pincho en un registro de una variable y se marca dentro del estado del cómputo en amarillo su valor. El mismo resultado obtengo al pinchar un registro de tipo parámetro.

Para finalizar hacemos pruebas con registros del estado del cómputo , y al pincharlos se marcan en amarillo dentro de la pila de llamadas la llamada a la que pertenece, y dentro de la pila de

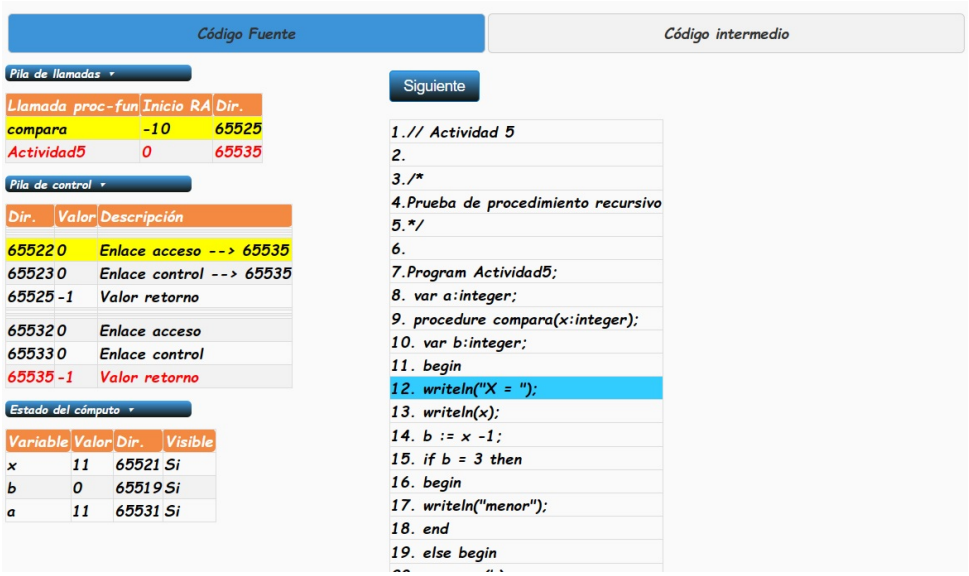

Figura 9.10: Muestra de relaciones sobre un enlace de acceso en la pila de control.

control, el registro de la pila donde se almacena siempre que este no esté oculto por opciones.

9.2. Pruebas de software

Al finalizar el desarrollo del código final se hace una batería de pruebas a base de ejemplos para comprobar que la traducción del mismo es correcta para llegar de forma correcta al resultado de cada algoritmo. En la comprobación de todos los ejemplos que muestro a continuación se ha comprobado no solo el resultado, sino todo el proceso de creación y destrucción de registros de activación, correcta composición del mismo, visibilidad de variables, estado del cómputo, salidas por pantalla y correcto funcionamiento de todo el sistema.

- Actividad 1 - Programa simple.

  Comprobación de simulación de un programa simple.

- Actividad 2 - Prueba de IF.

  Comprobación de simulación de un programa que contiene estructuras condicionales.

- Actividad 3 - Invocación a subprograma.

  Comprobación de simulación de un programa que llama a un procedimiento.

- Actividad 4 - Llamada a función.

  Comprobación de simulación de un programa que llama a una función y del retorno de forma correcta del valor de salida de la función.

- Actividad 5 - Recursividad Procedimientos.

  Comprobación de simulación de un programa que tiene llamadas recursivas de procedimientos.

- Actividad 6 - Recursividad Funciones. Prog. Tetraédrico.

  Comprobación de simulación de un programa que tiene llamadas recursivas de funciones. En concreto se hace con el programa Tetraédrico.

- Actividad 7 - Recursividad Funciones. Prog. Fibonacci.

  Comprobación de simulación de un programa que tiene llamadas recursivas de funciones. En concreto se hace con el programa Fibonacci.

- Actividad 8 - Anidamientos variables no locales.

  Comprobación de simulación de un programa que tiene funciones anidadas para comprobar si funciona el anidamiento en variables no locales.

También se hacen diversas pruebas de interfaz comprobando que la aplicación funciona perfectamente con los navegadores mas comunes, como son Edge, Crome, Opera y Firefox.

Capítulo 10

Conclusiones y trabajos futuros

El desarrollo de este capítulo muestra las conclusiones a las que se han llegado tras la ejecución el proyecto y sobre las posibles mejoras o ampliaciones que se pueden hacer a la herramienta de cara a mejorar su usabilidad por parte del alumno o ampliar su funcionalidad.

10.1. Conclusiones

El desarrollo del proyecto, por su complejidad, ha resultado un proceso con bastantes altibajos que, en su conjunto, ha resultado ser bastante grato en cuanto al resultado y enriquecedor en cuanto a lo aprendido.

Por las peculiaridades del proceso de construcción, en el cual conocía las líneas básicas del camino a seguir para la realización de un compilador pero no su fondo, ha sido un proceso constante de aprendizaje que me ha ayudado a su vez a la comprensión de las asignaturas de que he cursado mientras hacía el proyecto que son *Procesadores del lenguaje I* y *Procesadores del lenguaje II*.

La elección de la metodología ha hecho que, precisamente al no tener que tener un modelo de diseño cerrado para comenzar la implementación, me haya ayudado bastante en poder acompasar el proyecto con estas asignaturas.

En relación al compilador, que objetivamente ha sido la parte con mas dificultad de todo el proceso, el hecho de su construcción sea un proceso incremental en cuanto a sus fases (mostradas en la figura 3.2), ha hecho que la finalización de cada una de ella fuese una inyección de moral para abordar la siguiente fase.

En cuanto a los objetivos, podemos afirmar que se han cumplido todos los marcados inicialmente, e incluso, en la parte partes como el Frond-End, se han ampliado respecto a lo inicialmente previsto, incluyendo funcionalidad que hace mas fácil al alumno el manejo de la herramienta y el seguimiento de la simulación como son la inclusión de ejercicios de ejemplo predefinidos o las ayudas en cuanto a la visualización y seguimientos de los registros de activación y el estado del cómputo.

En general, gracias a la realización del proyecto, he aprendido en profundidad las fases del desarrollo de un compilador y he comprendido su funcionamiento que, en determinados aspectos, como los registros de activación, pienso que es mas sencillo de aprenderlos con la práctica.

La conclusión final, y creo que la mas importante, es que creo que esta herramienta puede servir de gran ayuda a los estudiantes del Grado de Ingeniería Informática en la **Universidad Nacional de Educación a Distancia**.

10.2. Trabajos futuros

Aunque el proyecto cumple todos los objetivos marcados al inicio del mismo, este permite múltiples opciones de mejora para su ampliación o usabilidad de cara al alumno.

A continuación se presentan una serie posibles de trabajos futuros que aportan valor al proyecto.

- **Relativos al lenguaje**
  1. Ampliar los tipos de datos permitidos por el lenguaje.
  2. Ampliar el lenguaje con mas estructuras tipo while, for, etc...
  3. Permitir el paso de parámetros por referencia.
- **A nivel de Front-End**
  1. Permitir autentificación de usuarios UNED.
  2. Poder cargar nuevos ejemplos mediante una importación de un fichero XML, csv, Json o similar.
  3. Permitir a los alumnos guardar sus propios programas.
  4. Poder grabar simulaciones en un determinado punto para continuar con la simulación en otro momento.
  5. Poner puntos de interrupción en la simulación como si fuera un depurador de código.
  6. Mejorar las ayudas visuales que ofrece el sistema con flechas que indiquen las relaciones.

Bibliografía

[1] P. De Miguel, *Fundamentos de los Computadores. Quinta Edición Revisada.* Ed. Paraninfo, 1996.

[2] F. López Ostenero, A. M. García Serrano., *Teoría de los lenguajes de programación.* Editorial Universitaria Ramon Areces., 2014.

[3] A. Aho, R. Sethi, J. D. Ullman, *Compiladores: Principios, técnicas y herramientas.* PEARSON Educación, 2008.

[4] F. Álvarez Valero, *ENS2001. Manual de usuario.* https://ens2001.falvarez.es/files/Memoria-TFC-ENS2001.pdf, 2003.

[5] Departamento de Lenguajes y Sistemas Informáticos. ETS de Ingeniería Informática (UNED), *Guía de estudio. Teoría de los lenguajes de programación.* http://portal.uned.es/portal/page?_pageid=93,61703783&_dad=portal&_schema=PORTAL&idAsignatura=71012024, 2022.

[6] J. M. T. Roger S Pressman, *Ingeniería del software.* McGraw Hill, 1988.

[7] Quantitative Software Management, *Function Point Languages Table.* https://www.qsm.com/resources/function-point-languages-table, 22 de Junio de 2023, 2023.

[8] ARSYS, *Planes de hosting.* https://www.arsys.es/hosting, 22 de Junio de 2023, 2023.

[9] Manuel Alfonseca Moreno, Marina de la Cruz Echeandía, Alfonso Ortega de la Puente, Estrella Pulido Cañabate, *Compiladores E Interpretes: Teoría Y Practica.* PEARSON Educación, 2006.

[10] C. Larman., *UML y Patrones.* Pearson Educación, 2003.

[11] K. Schwaber, J. Sutherland, *La Guía de Scrum ™. La Guía Definitiva de Scrum: Las Reglas del Juego.* https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Spanish-European.pdf, 2020.

[12] Palacio, Marta, *Scrum Master. Temario Troncal 1. Versión 3.0. Scrum Manager.* https://www.scrummanager.com/files/scrum_master.pdf, 2022.

[13] D. de Lenguajes y Sistemas Informáticos. ETS de Ingeniería Informática (UNED), *Guía de estudio. Procesadores del Lenguaje I.* http://portal.uned.es/portal/page?_pageid=93,61703783&_dad=portal&_schema=PORTAL&idAsignatura=71013130&idContenido=5&idTitulacion=7101, 2022.

[14] ——, *Guía de estudio. Procesadores del Lenguaje II.* http://portal.uned.es/portal/page?_pageid=93,61703783&_dad=portal&_schema=PORTAL&idAsignatura=71013118, 2022.

[15] J. Team, *Manual JFlex.* https://jflex.de/manual.html, 15 de mayo de 2022, 2023.

[16] S. Hudson, *CUP User's Manual.* https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html, 15 de mayo de 2022, 1999.

[17] Z. Carter, *Documentación Jison.* https://gerhobbelt.github.io/jison/docs/, 2009.

[18] Free Software Foundation, Inc. Manual de bison https://www.gnu.org/software/bison/manual/bison.html, 22 de Febrero de 2023.

Apéndice A

Definición del lenguaje

A.1. Descripción del lenguaje

Este apartado es una de descripción técnica del lenguaje PFGUnedTLP.

El lenguaje PFGUnedTLP surge como lenguaje inventado para fines didácticos resultado de la combinación de varios lenguajes con el objetivo de poder estudiar el registro de activación y el estado del cómputo durante la ejecución de un programa.

A.2. Aspectos Léxicos

Desde un enfoque léxico, un programa en PFGUnedTLP está compuesto por una secuencia de tokens ordenados. Un token en una entidad léxica indivisible que tiene un sentido único dentro del lenguaje PFGUnedTLP.

Los tokens se pueden agrupar en varias categorías según su uso, como son los identificadores, palabras reservadas, operadores o los delimitadores. Estos se irán describiendo en sus respectivos apartados a lo largo del documento.

A.2.1. Comentarios

Un comentario es una secuencia de caracteres sobre la cual el compilador no actúa pasándolos al analizador léxico, de tal modo que no interfieren en el funcionamiento del programa. Estos comentarios son creados por el programador para dar información o un valor añadido de lo que va haciendo el programa a lo largo de todo su código fuente.

En PFGUnedTLP hay comentarios de dos tipos:

- Comentarios de una línea.

  Empiezan por // y consideran como comentario todo lo que haya desde su comienzo hasta el final de la línea.

  Ej: Ejemplo de comentario de una línea.

  //Comentario en una línea.

- Comentarios multilínea.

  Comienza por /* y consideran como comentario todo lo que haya después hasta que aparece el fin de comentario multilínea */.

  Ej: Ejemplo de comentario multilínea.

```
/* Esto es un
comentario
que abarca
varias líneas */
```

A.2.2. Constantes literales

En PFGUnedTLP existe la posibilidad del uso de constantes literales para escribir el programa. Estas se dividen principalmente en 2 tipos:

- Enteras.

  Las constantes literales enteras sirven para representar valores enteros positivos, por lo que no se permite el uso del operados unario -. Tampoco es posible que una de estas constantes que contenga más de un dígito lleve ceros por la izquierda.

  Ej: Ejemplo de constante literales enteras.

```
0, 123, 54
```

- Cadenas de caracteres.

  Son una secuencia de caracteres delimitadas por comillas dobles. Se usan para sacar mensajes por pantalla. No se admiten secuencias de escape como /t /n /r (tabulación, salto de línea y retorno de carro) dentro de las mismas.

  Ej: Ejemplo de cadenas de texto.

```
"Esto es una cadena"
"MENSAJE"
```

A.2.3. Identificadores

Un identificador o símbolo una secuencia de letras y números que sirve para asignar un nombre a las variables, parámetros, funciones y procedimientos que componen el programa. En PFGUnedTLP un literal debe de comenzar obligatoriamente por una letra seguida de un número indeterminado de letras o dígitos. No se admite ningún otro tipo de carácter que no sean letras o números. El lenguaje es (case-sensitive), por lo que discrimina mayúsculas y minúsculas. No se pueden usar palabras clave como identificador. No pueden ser declarados dos identificadores con el mismo nombre dentro del mismo ámbito. Los identificadores de las funciones y procedimientos deben de ser únicos por lo que no pueden repetirse el nombre de un procedimiento o función aunque estén en diferentes ámbitos.

A.2.4. Palabras reservadas

Las palabras reservadas son un componente del lenguaje con un significado específico dentro del mismo por lo que no pueden ser usadas como identificadores.

PALABRA RESERVADA	DESCRIPCIÓN
Program	Inicio del programa.
integer	Tipo de dato entero.
function	Inicio de una función.
procedure	Inicio de un procedimiento.
var	Inicio de la declaración de variables.
if	Inicio condicional si.
else	Inicio alternativa condicional si.
then	Inicio del cuerpo del bloque condicional si.
begin	Inicio del cuerpo del bloque de sentencias.
end	Fin del cuerpo del bloque de sentencias.
writeln	Procedimiento para escribir por pantalla.
exit	Retorno de función.

Tabla A.1: Lista de identificadores.

A.2.5. Delimitadores

Los delimitadores se usan como su nombre indica para determinar determinadas partes del lenguaje. Los delimitadores utilizados en PFGUnedTLP son los siguientes.

DELIMITADOR	DESCRIPCIÓN
(	Delimitador de inicio de expresiones y parámetros.
)	Delimitador de fin de expresiones y parámetros.
.	Delimitador de final de programa.
,	Delimitador de identificadores.
;	Delimitador de sentencias.
:	Delimitador de tipo en la declaración de variables y parámetros.
//	Delimitador de comentario.
/*	Delimitador de inicio de comentario multilínea.
*/	Delimitador de fin de comentario multilínea.

Tabla A.2: Lista de delimitadores.

A.2.6. Operadores

En PFGUnedTLP también existen diferentes tipos de operadores para poder construir el lenguaje. Los operadores utilizados en PFGUnedTLP son los siguientes.

OPERADOR	DESCRIPCIÓN
+	Operador aritmético suma.
-	Operador aritmético resta.
=	Operador relacional igual.
:=	Operador asignación.

Tabla A.3: Lista de operadores.

A.3. Aspectos Sintácticos

Un programa escrito en PFGUnedTLP tiene que ceñirse a una estructura preestablecida. A continuación mostramos los detalles de esta estructura desde la estructura general hasta el detalle de cada uno de sus componentes.

A.3.1. Estructura y ámbitos de un programa

El esquema general de un programa escrito en PFGUnedTLP es el siguiente:

```
Program nombreDelPrograma;
    //Sección para la declaración de variables.
    //Sección para la declaración de funciones
    //y/o procedimientos.
begin //cuerpo del programa principal.
    //Lista de sentencias.
end.
```

Un programa escrito en PFGUnedTLP comienza siempre por la palabra reservada Programa seguida del identificador que da nombre al programa y un punto y coma ";".

A continuación comienza la sección opcional de declaración de variables, luego la sección opcional de declaración de funciones o procedimientos y por último el bloque del cuerpo del programa principal que comienza con un "begin" seguido de cero o varias sentencias hasta que finaliza el programa principal con la palabra reservada "end" seguida de un punto ".".

Los comentarios se pueden insertar en cualquier parte del programa.

Dentro de la sección de declaración de funciones o procedimientos, la estructura es similar a la del programa principal con unas pequeñas variaciones que se verán en la sección correspondiente, permitiendo así la anidación de subprogramas.

La visibilidad de cada identificador viene dada por la sección de código donde se ha declarado, siendo solo accesibles aquellos identificadores que hayan sido declaradas previamente dentro de un ámbito que esté abierto, de tal modo que un programa de PFGUnedTLP nos podemos encontrar referencias globales que son las que están declaradas en el programa principal, referencias locales que son los símbolos declarados dentro de una función procedimiento y un tercer tipo de referencias no locales que son símbolos que son accesibles desde un determinado procedimiento o función y que han sido declarados en algún ámbito previo dentro de la estructura de anidamiento de subprogramas.

A.3.2. Tipos primitivos

En PFGUnedTLP solo existe un tipo primitivo identificado con la palabra reservada integer que identifica valores numéricos enteros.

A.3.3. Declaraciones de Variables

En PFGUnedTLP el uso de variables está supeditado a la declaración previa y de forma correcta de las mismas. Esta declaración comienza por la palabra reservada "var" seguido de una o varias líneas que contengan una lista de uno o más identificadores seguida cada una de ellas del

delimitador dos puntos ":", el tipo al que pertenece que en nuestro lenguaje solo puede ser el identificado por la palabra reservada "integer" y un punto y coma ";".

Ej: Un ejemplo de declaración puede ser el siguiente.

```
var
    salida :integer;
    a, b, c: integer;
```

A.3.4. Declaración de subprogramas

Con el fin de organizar el programa modularmente, encapsular código, y analizar el comportamiento del registro de activación en llamadas a subprogramas y el estado del cómputo según los diferentes ámbitos, PFGUnedTLP permite la declaración de subprogramas.

Estos subprogramas dentro del lenguaje PFGUnedTLP pueden ser de dos tipos, los procedimientos y las funciones. La estructura de ambos es muy similar existiendo pequeñas diferencias entre ambos que los distinguen. A continuación vemos la estructura de cada una de ellos.

A.3.4.1. Declaración de procedimientos

Procedimientos son rutinas que realizan una determinada operación. Los procedimientos en PFGUnedTLP deben de seguir la siguiente estructura sintáctica:

```
procedure NBDelProcedimiento(param1: integer , ,paramN :integer);
    //Sección para la declaración de variables locales.
    //Sección para la declaración de funciones
    //y/o procedimientos
    //locales al procedimiento.
begin //cuerpo del procedimiento
    //Lista de sentencias.
end;
```

Comienzan declarando la cabecera del procedimiento con la palabra reservada procedure seguida por el identificador que va a dar nombre al procedimiento, a continuación se usa el delimitador de inicio de parámetros "(" seguido de una lista de parámetros opcional, el delimitador de fin de parámetros ")" y un punto y coma ";".

La lista de parámetros que como hemos indicado es opcional, está compuesta por la declaración de los distintos parámetros que tiene el procedimiento separados por coma ",". Estos parámetros se declaran indicando primero el identificador del parámetro, seguido de dos puntos ":" y el tipo primitivo del parámetro que en PFGUnedTLP es siempre "integer".

Ej: Ejemplo de declaración de dos parámetros.

```
x:integer , p:integer;
```

A partir de aquí nos encontramos una estructura casi calcada a la del programa principal exceptuando que en símbolo de cierre de procedimiento es un punto y coma ";" en vez de un punto ".". es decir, podemos encontrar la parte opcional de declaración de variables y subprogramas, que tiene la misma estructura que en el programa principal. Como hemos comentado anteriormente el lenguaje permite anidamiento de subprogramas, por lo que un procedimiento puede tener declarados otros procedimientos y/o funciones locales.

Seguidamente, comienza la parte obligatoria del cuerpo del subprograma, encerrada entre los delimitadores de inicio "begin" y fin "end" y un punto y coma ";". Dentro de estos delimitadores, al igual que en el programa principal podemos encontrar una lista de sentencias que componen el subprograma.

Ej: Ejemplo de procedimiento.

```
procedure decrementa(x:integer);
var
    y:integer;
begin
    y := x - 1;
    writeln("y(2) = ");
    writeln(y);
end;
```

A.3.4.2. Declaración de funciones

Funciones son rutinas que realizan una determinada operación y que al final devuelven un valor. Las funciones en PFGUnedTLP deben de seguir la siguiente estructura sintáctica:

```
function NBDeLaFuncion(param1:integer, ,paramN :integer):integer;
    //Sección para la declaración
    //de variables locales.
    //Sección para la declaración de funciones
    //y/o procedimientos locales a la función.
begin //cuerpo de la función.
    //Lista de sentencias.
    exit(identificador de retorno);
end;
```

Comienzan declarando la cabecera de la función con la palabra reservada function seguida por el identificador que va a dar nombre a la función, a continuación se usa el delimitador de inicio de parámetros "(" seguido de una lista de parámetros opcional, el delimitador de fin de parámetros ")" . A continuación se usa el delimitador de tipo ":" y el tipo de retorno de la función que en PFGUnedTLP es siempre "integer" y un punto y coma ";".

La lista de parámetros que como hemos indicado es opcional, está compuesta por la declaración de los distintos parámetros que tiene la función separados por coma ",". Estos parámetros se declaran indicando primero el identificador del parámetro, seguido de dos puntos ":" y el tipo primitivo del parámetro que en PFGUnedTLP es siempre "integer".

Ej: Ejemplo de declaración de dos parámetros.

```
x:integer , p:integer
```

A partir de aquí nos encontramos una estructura casi calcada a la del programa principal exceptuando que en símbolo de cierre de función es un punto y coma ";" en vez de un punto ".". Es decir, podemos encontrar la parte opcional de declaración de variables y subprogramas, que tiene la misma estructura que en el programa principal. Como hemos comentado anteriormente el lenguaje permite anidamiento de subprogramas, por lo que una función puede tener declarados otras funciones y/o procedimientos locales.

Seguidamente, comienza la parte obligatoria del cuerpo del subprograma, encerrada entre los delimitadores de inicio "begin" y fin "end" y un punto y coma ";". Dentro de estos delimitadores, al igual que en el programa principal podemos encontrar una lista de sentencias que componen el subprograma. La última sentencia del cuerpo del subprograma que además es obligatoria, es la sentencia de retorno de la función que se identifica con la palabra reservada "exit", la cual devuelve el valor calculado en la función. El valor que devuelve se tiene que adecuar al tipo declarado para la función que en este lenguaje es siempre "integer".

Ej: Ejemplo de declaración de una función el PFGUnedTLP.

```
function triangular(k:integer):integer;
var
    salida:integer;
begin
    if k = 0 then
    begin
        salida := 0;
    end
    else begin
        salida := k + triangular(k-1);
    end;
    exit(salida);
 end;
```

A.3.4.3. Paso de parámetros a funciones y procedimientos

Los parámetros siempre se pasan por valor, y pueden ser Identificadores, literales enteros, o una combinación de sumas y/o restas de los mismos.

A.3.5. Sentencias y Expresiones

Los bloques del cuerpo de programa principal, cuerpo del procedimiento y cuerpo de función están compuestos por sentencias que a su vez contienen expresiones.

Una expresión es una combinación de elementos del lenguaje que se evalúan y dan como resultado un valor. En PFGUnedTLP las expresiones se engloban en dos tipos.

A.3.5.1. Expresiones Aritméticas

Son aquellas que al ser evaluadas devuelven un valor entero.

Para trabajar con estas expresiones asociamos una precedencia y asociatividad entre sus operadores.

Precedencia Asociatividad

() izquierdas

+ - izquierdas

Ej: Ejemplo de expresiones aritméticas

```
0 + a
k + funcionX(k-1)  //Llamada a función.
x - 1
```

A.3.5.2. Expresiones lógicas

Son aquellas que devuelven un valor de tipo lógico al programa. En lenguaje PFGUnedTLP solo se dan al usar el operador relacional "=" y solo pueden usarse en la evaluación de la sentencia if, para entrar dentro del cuerpo del condicional si cumple la expresión o mandarlo a la alternativa del condicional en caso de que esta exista y no cumpla la expresión.

Ej: Ejemplo de uso de expresión lógica asociada al if.

```
if k = 0 then
begin
    //sentencias si cumple el condicional.
end
else begin
   //sentencias si no cumple el condicional.
end;
```

A.3.5.3. Sentencias

Las sentencias en PFGUnedTLP se usan para realizar determinadas operaciones dentro del lenguaje. Hay de varios tipos y todas tienen que acabar por el delimitador de fin de sentencias punto y coma ";".

- Sentencia de asignación. Sirven para asignar valores a una variable. La sintaxis es la siguiente. Primero se escribe el nombre del identificador al cual se le quiere asignar el valor seguido del operador de asignación dos puntos igual ":=", una expresión y el delimitador de fin de sentencias punto y coma ";".

  Ej: Ejemplo del uso de sentencias de asignación.

```
salida := 0;
salida := 4 + k + 1;
salida := triangular(n) + tetraedrico(n-1); //Llamada a fun-
```
ción.

- Sentencia de flujo condicional if. Sirve para condicionar el flujo de la ejecución de un programa de acuerdo al cumplimiento o no de una expresión lógica. La sintaxis tiene que cumplir la siguiente estructura. Comienza por la palabra reservada if seguido de una expresión lógica y a continuación las palabras reservadas then begin y el bloque sentencias si cumple la expresión. En este punto pueden ocurrir dos cosas , que no exista el bloque opcional de sentencias cuando no cumple la expresión en cuyo caso terminaría la expresión if con la palabra reservada end seguida de un punto y coma ";", o que exista la parte opcional de sentencias si no cumple la expresión en cuyo caso vendrían las palabras reservadas end else begin, el bloque opcional de sentencias si no culpe la condición lógica y terminaríamos con la palabra reservada end seguida de un punto y coma ";".

  Ej: Ejemplo de uso de sentencia de flujo condicional if.

```
if k = 0 then
begin
    a = a  + 3;
    salida :=  a- 1;
end
else begin
    salida := k + k + 1;
end;
if a = 8 then
begin
    a = 0;
end;
```

- Sentencia de salida. Para mostrar mensajes por pantalla o resultados, el lenguaje PFGUnedTLP tiene una función predefinida que realiza esta salida por pantalla. La sintaxis de la sentencia comienza por la palabra reservada writeln seguida de la apertura de parentesis "(", a continuación el mensaje o resultado que se va a mostrar, que puede ser una cadena de texto entrecomillada o una expresión aritmética, y finalizamos con un cierre de paréntesis ")" y un punto y coma ";".

  Ej: Ejemplo de uso de setencias de salida.

```
writeln(a);
writeln("Llamada␣a␣subprogramas");
writeln("Fin.");
```

- Llamadas a procedimientos.

  Para llamar a un procedimiento en PFGUnedTLP ha de escribirse el identificador con el que se ha declarado el procedimiento y una apertura de paréntesis "(", dentro de este paréntesis hay que poner los parámetros que vamos a pasarle al procedimiento separados

por comas. El número de parámetros tiene que ser el mismo con el que se ha declarado el mismo. Una vez finalizada esa lista de parámetros se introduce el delimitador de fin de sentencias “)” y un punto y coma “;”.

Ej: Ejemplo de uso de llamadas a procedimientos.

```
resta(a,b);
decrementa(z);
calculaTotales();
```

- Llamadas a funciones.

  Se podrían englobar en la parte de las expresiones ya que devuelven un resultado.

  Para llamar a una función en PFGUnedTLP ha de escribirse el identificador con el que se ha declarado la función y una apertura de paréntesis ”(”, dentro de este paréntesis hay que poner los parámetros que vamos a pasarle a la función separados por comas. El número de parámetros tiene que ser el mismo con el que se ha declarado el mismo. Una vez finalizada esa lista de parámetros se introduce el delimitador de fin de sentencias “)” y un punto y coma “;”. Como no es una sentencia propiamente dicha ya que devuelven un valor no pueden ir solas, sino que van incluidas dentro de otras sentencias.

  Ej: Ejemplo de uso de llamadas a funciones.

```
resultado := suma(a,b);
salida := triangular(n)+tetraedrico(n-1);
```

Apéndice B

Código intermedio

Este apartado es una descripción técnica del código intermedio generado por el sistema al compilar un programa escrito en lenguaje PFGUnedTLP.

B.1. Especificación de las instrucciones.

El código intermedio generado por el compilador al compilar un programa escrito en lenguaje PFGUnedTLP es una representación lineal de las instrucciones que tiene que seguir el compilador para producir el resultado requerido en el código fuente.

Como se puede observar en la figura B.1 está formado por dos partes, la primera parte consta de la línea del código fuente desde la que se genera la línea de código intermedio y la segunda parte es una cuádrupla o cuarteto formada por una instrucción seguida de 3 parámetros.

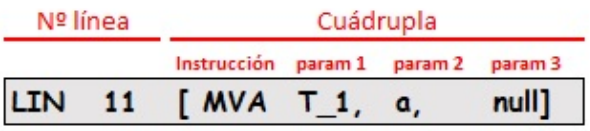

Figura B.1: Estructura línea código intermedio.

A continuación se describe el juego de instrucciones de código intermedio que usa el compilador:

- **STARTGLOBAL**

  Instrucción que marca el inicio de la ejecución de nuestro programa.

  Es siempre la primera instrucción que lleva el código intermedio.

  Crea el registro de activación del programa principal e inicializa el valor de retorno, estado de la máquina, enlace de control y enlace de acceso.

  Formato instrucción:

  LIN N [STARTGLOBAL null , null , null]

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción. El valor es siempre 1.
  2. Instrucción: STARTGLOBAL.
  3. Parámetro 1: null.

4. Parámetro 2: null.
5. Parámetro 3: null.

- **STARTSUBPROGRAMAP**

  Crea el registro de activación del procedimiento e inicializa el valor de retorno, estado de la máquina, enlace de control y enlace de acceso.

  Formato instrucción:

  ```
  LIN N [STARTSUBPROGRAMAP null, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: STARTSUBPROGRAMAP.
  3. Parámetro 1: null.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **STARTSUBPROGRAMAF**

  Crea el registro de activación de la función e inicializa el valor de retorno, estado de la máquina, enlace de control y enlace de acceso.

  Formato instrucción:

  ```
  LIN N [STARTSUBPROGRAMAF null, null, null ]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: STARTSUBPROGRAMAF.
  3. Parámetro 1: null.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **VAR**

  Instrucción que sirve para declarar e inicializar una variable.

  Crea la entrada en la pila de control con la posición de la variable.

  Genera en la estructura que compone el estado del cómputo un registro con la posición que ocupa en la pila y el nombre de la variable.

  Formato instrucción:

  ```
  LIN N [VAR a, 0, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: VAR.
  3. Parámetro 1: Nombre de la variable.

4. Parámetro 2: Al usar solo el tipo de datos entero siempre se inicializa cero.
5. Parámetro 3: null.

- **PUNTEROGLOBAL**

  Para el ámbito global genera las variables temporales en el registro de activación y el estado del cómputo.

  Genera la llamada del programa principal en la pila de llamadas.

  Rellena dentro de su registro de activación las posiciones del enlace de control y en enlace de acceso.

  Formato instrucción:

  LIN N [PUNTEROGLOBAL, vartemporal, TamaRA , AmbitoActual]

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: PUNTEROGLOBAL.
  3. Parámetro 1: Último temporal creado en tiempo de compilación.
  4. Parámetro 2: Tamaño del registro de activación del programa principal.
  5. Parámetro 3: Nombre del programa principal.

- **PUNTEROLOCAL**

  Para el ámbito del procedimiento o función genera los parámetro y las variables temporales en el registro de activación y el estado del cómputo.

  Genera la llamada del procedimiento o función en la pila de llamadas.

  Rellena dentro de su registro de activación las posiciones del enlace de control y en enlace de acceso.

  Formato instrucción:

  LIN N [PUNTEROLOCAL, null, TamaRA ,AmbitoActual]

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: PUNTEROLOCAL.
  3. Parámetro 1: null.
  4. Parámetro 2: Tamaño del registro de activación del procedimiento o función.
  5. Parámetro 3: Nombre del procedimiento o función.

- **MV**

  Instrucción que mueve el valor de y a x.

  x := y

  Formato instrucción:

  LIN N [MV param 1, param 2, null]

Descripción de los parámetros:

1. Línea: Línea que genera la instrucción.
2. Instrucción: MV.
3. Parámetro 1: Destino.
4. Parámetro 2: Origen.
5. Parámetro 3: null.

- **MVA**

  Instrucción que mueve la dirección de memoria de y a x.

  x := &y

  Formato instrucción:

  ```
  LIN N [MVA param 1, param 2, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: MVA.
  3. Parámetro 1: Destino.
  4. Parámetro 2: Origen.
  5. Parámetro 3: null.

- **MVP**

  Instrucción que mueve el contenido de dirección de memoria de y a x.

  x := *y

  Formato instrucción:

  ```
  LIN N [MVP param 1, param 2, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: MVP.
  3. Parámetro 1: Destino.
  4. Parámetro 2: Origen.
  5. Parámetro 3: null.

- **STP**

  Instrucción que guarda en el contenido de la dirección de memoria de x el valor de y.

  *x := y

  Formato instrucción:

  ```
  LIN N [STP param 1, param 2, null]
  ```

  Descripción de los parámetros:

1. Línea: Línea que genera la instrucción.
2. Instrucción: STP.
3. Parámetro 1: Destino.
4. Parámetro 2: Origen.
5. Parámetro 3: null.

- **ADD**

  Instrucción de suma. Suma y + z y el resultado lo coloca en x.

  x := y + z

  Formato instrucción:

  ```
  LIN N [ADD param 1, param 2, param3]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: ADD.
  3. Parámetro 1: Destino.
  4. Parámetro 2: Operando 1.
  5. Parámetro 3: Operando 2.

- **SUB**

  Instrucción de resta. Resta y - z y el resultado lo coloca en x.

  x := y - z

  Formato instrucción:

  ```
  LIN N [SUB param 1, param 2, param3]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: SUB.
  3. Parámetro 1: Destino.
  4. Parámetro 2: Operando 1.
  5. Parámetro 3: Operando 2.

- **EQ**

  Instrucción que comprara los valores de los dos últimos parámetros. Si son iguales le asigno un 1 al primer parámetro x y sino se le asigna un 0 a x.

  x := (y == z) ?1:0

  Formato instrucción:

  ```
  LIN N [EQ param 1, param 2, param3]
  ```

  Descripción de los parámetros:

1. Línea: Línea que genera la instrucción.
2. Instrucción: EQ.
3. Parámetro 1: Resultado.
4. Parámetro 2: Operando 1.
5. Parámetro 3: Operando 2.

- **BRF**

  Instrucción de salto. Si el valor del primer parámetro es cero (falso) salto a la posición de la etiqueta que viene en el segundo parámetro.

  Formato instrucción:

  ```
  LIN N [BRF param 1, param 2, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: BRF.
  3. Parámetro 1: Valor.
  4. Parámetro 2: Etiqueta.
  5. Parámetro 3: null.

- **BR**

  Instrucción de salto. Salta a a línea donde esté la etiqueta del primer parámetro.

  Formato instrucción:

  ```
  LIN N [BR param 1, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: BR.
  3. Parámetro 1: Etiqueta.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **INL**

  Instrucción que inserta una una etiqueta L.Se usa para marcar posiciones de salto.

  Formato instrucción:

  ```
  LIN N [INL param 1, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: INL.
  3. Parámetro 1: Etiqueta.

4. Parámetro 2: null.
5. Parámetro 3: null.

- **WRITEINT**

  Instrucción de escritura. Escribe el contenido del entero x y después mete un retorno de carro.

  Formato instrucción:

  ```
  LIN N [WRITEINT param 1, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: WRITEINT.
  3. Parámetro 1: Valor entero.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **WRITETXT**

  Instrucción de escritura. Escribe el contenido de la cadena de texto x quitándole la comilla inicial y final, y después mete un retorno de carro.

  Formato instrucción:

  ```
  LIN N [WRITETXT param 1, param 2, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: WRITETXT.
  3. Parámetro 1: Variable temporal que almacena la cadena.
  4. Parámetro 2: Cadena de texto.
  5. Parámetro 3: null.

- **CALL**

  Llamada a una función o procedimiento. Salta a la línea de código intermedio donde está la etiqueta de ese procedimiento o función.

  Formato instrucción:

  ```
  LIN N [CALL param 1, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: CALL.
  3. Parámetro 1: Nombre de la etiqueta.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **FINSUBPROGRAMA**

  Instrucción de fin de función o procedimiento. Salta a la línea de código intermedio que tiene marcada la dirección de retorno de registro de activación en el que estamos.

  Formato instrucción:

  ```
  LIN N [FINSUBPROGRAMA param 1, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: FINSUBPROGRAMA.
  3. Parámetro 1: Nombre procedimiento o función.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **EXIT**

  Valor de salida de una función. Pone en valor de x en la posición del valor de retorno del registro de activación.

  Formato instrucción:

  ```
  LIN N [EXIT param 1, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: Línea que genera la instrucción.
  2. Instrucción: EXIT.
  3. Parámetro 1: Valor salida.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

- **DEVCALL**

  Es la ultima instrucción que genera la llamada a un procedimiento o una función. Sirve devolver el valor por una función después de la ejecución.

  Internamente al ejecutarse elimina todo lo relativo al registro de activación del procedimiento o función que acabamos de terminar de ejecutar.

  Guarda en en un temporal el valor de la salida del registro de activación.

  Elimina del estado del cómputo los valores del registro de activación que estamos eliminando.

  Elimina de la pila de control los valores del registro de activación que estamos eliminando.

  Elimina la entrada del procedimiento o función de la pila de llamadas.

  Formato instrucción:

  ```
  LIN N [DEVCALL param 1, null, null]
  ```

  Descripción de los parámetros:

1. Línea: Número de línea del código fuente que genera la instrucción.
2. Instrucción: DEVCALL.
3. Parámetro 1: Nombre del procedimiento o función que devuelve la llamada.
4. Parámetro 2: Si se trata de una función contiene la variable temporal donde se almacena el valor devuelto por la función. En caso de que sea un procedimiento el que genera la instrucción el valor almacenado en este parámetro sería un null.
5. Parámetro 3: null.

- **HALT**

  Instrucción que marca el fin de la ejecución.

  Cuando el compilador alcanza la instrucción *HALT* se da por finalizada la ejecución del programa compilado.

  Es única instrucción que no lleva asociado numero de línea por no ser necesaria para la ejecución del programa.

  Formato instrucción:

  ```
  [HALT null, null, null]
  ```

  Descripción de los parámetros:

  1. Línea: No es necesaria.
  2. Instrucción: HALT.
  3. Parámetro 1: null.
  4. Parámetro 2: null.
  5. Parámetro 3: null.

Apéndice C

Manual de usuario

El manual de usuario es una guía de apoyo al usuario para conocer las diferentes funcionalidades que ofrece la aplicación.

C.1. Visión general

La aplicación es una web que pretende ser de ayuda al usuario para el el estudio de los registros de activación y el estado del cómputo.

Las posibilidades que esta web ofrece son las siguientes:

- Crear sus propios programas en lenguaje PFGUnedTLP o usar alguno de los programas que trae predefinidos y compilarlos.
- Ver la evolución de los registros de activación y el estado del cómputo del programa compilado tanto desde el código fuente cómo desde el código intermedio que genera el programa la ser compilado.
- Visualizar la especificación del lenguaje PFGUnedTLP, lo cual permite al usuario aprender el lenguaje y solucionar posibles errores que se puedan dar en la construcción del programa.
- Consultar la ayuda para orientar al usuario sobre las posibilidades que le frece el programa.

C.2. Estructura

La estructura de la web está compuesta por cuatro páginas.

- **Inicio**

  Es la página de presentación de la web la cual permite el acceso al resto de páginas que componen la misma.

- **Simulador**

  Es la página donde el usuario puede generar sus propios programas o usar unos ejemplos predefinidos para compilarlos y poder ver el seguimiento a la evolución de los registros de activación y el estado del cómputo tanto desde el código fuente del programa compilado como desde el código intermedio que genera ese mismo programa al ser compilado.

- **Lenguaje**

  Esta página contiene las especificaciones del lenguaje PFGUnedTLP. A través de ella el usuario puede aprender y consultar las especificaciones del lenguaje para la correcta construcción del los programas.

- **Ayuda**

  Esta última página contiene una guía de toda la funcionalidad que contiene la página *Simulador* la cual el usuario puede consultar a modo de manual de usuario como de ayuda para conocer o usar toda la funcionalidad que ofrece el programa durante la ejecución.

C.3. Inicio

La página de inicio es la página representada en la figura C.1 que da la bienvenida y acceso al resto de páginas que componen la web.

Está compuesta por un acceso directo a la simulación y una barra de navegación donde puede acceder a las cuatro páginas que componen la web.

Figura C.1: Página de inicio.

C.4. Simulador

La entrada a la sección de *Simulador* es la página representada en la figura C.2 donde el usuario puede escribir un cargar un programa en lenguaje PFGUnedTLP y compilarlo para el posterior estudio y seguimiento de los registros de activación y el estado del cómputo.

A grandes rasgos la funcionalidad de la página está dividida en 2 fases, la primera donde el usuario escribe un programa en lenguaje PFGUnedTLP para su posterior compilación, y una vez que el programa está compilado sin errores, da acceso a la segunda fase donde se muestra

en la propia página la funcionalidad para hacer el seguimiento de los registros de activación y el estado del cómputo.

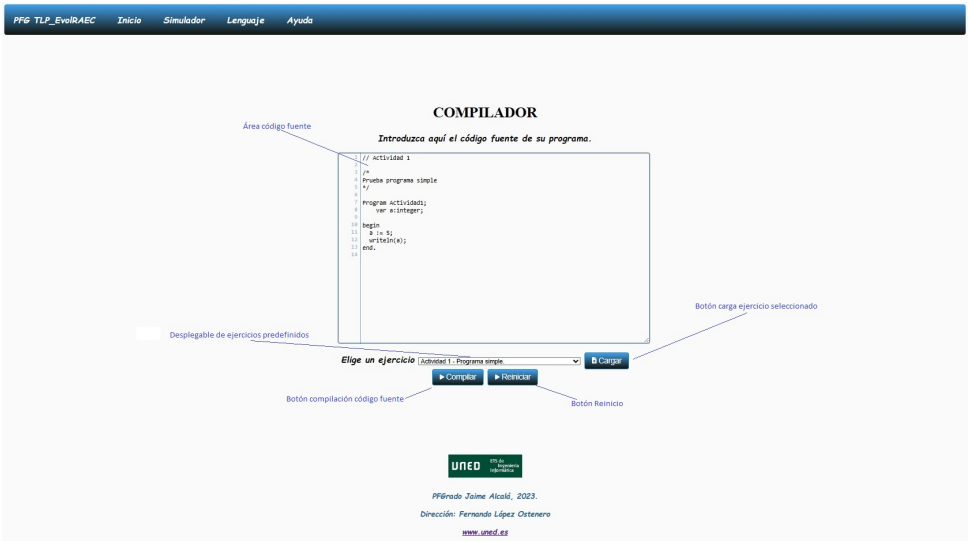

Figura C.2: Página Simulador. Código fuente.

C.4.1. Generar un programa

Lo primero que hay que hacer es escribir un programa en lenguaje PFGUnedTLP.

Para ello se ha habilitado un área marcada en la figura C.2 como *"Área código fuente"* donde se puede escribir el código fuente del programa requerido o cargar una serie de ejemplo de programas ya predefinidos que vienen por defecto en la aplicación.

En caso de querer cargar uno de estos programas predefinidos habrá que elegir mediante un desplegable marcado en la figura C.2 como *"Desplegable de ejercicios predefinidos"* las diferentes actividades que tiene cargadas el sistema, las cuales tienen como título una breve descripción del contenido de las mismas.

Una vez elegida al opción requerida, mediante el botón marcado en la figura C.2 como *"Botón carga ejercicio seleccionado"* se elimina todo el contenido que pudiera haber en el área designado para escribir el código fuente y se sustituye por el del programa que hemos elegido.

C.4.2. Compilar un programa

Una vez tenemos escrito el código fuente hay que proceder a compilar el programa.

Para ello lo único que tenemos que hacer es pinchar en el botón marcado en la figura C.2 como *"Botón compilación código fuente"* y el sistema realiza el análisis léxico, semántico y sintáctico del programa para comprobar que cumple con los requerimientos del lenguaje PFGUnedTLP.

Si el compilador encuentra algún problema al compilar el programa nos avisa del mismo dando indicaciones de la línea dónde se ha producido el error y el motivo del mismo para proceder a

su corrección.

Este error se muestra dentro de un nuevo área que aparece dentro de la figura C.3 designado como *"Área resultado compilación"*, el cual da información a cerca del error y cómo solucionarlo.

Figura C.3: Página Simulador. Error fase-1

Una vez subsanado el error se tiene que volver a compilar el programa hasta que el sistema no encuentre mas errores y dé el programa como válido, en cuyo caso saldrá un mensaje de que el programa ha sido compilado sin errores y se habilitará la parte de ejecución del programa.

Este mensaje se muestra dentro de un nuevo área que aparece dentro de la figura C.4 designado como *"Área resultado compilación"* y da paso a la funcionalidad para ver la evolución de los registros de activación y el estado del cómputo que pasamos a ver a continuación en el siguiente punto del manual denominado C.4.3 "Ejecutar un programa".

Figura C.4: Página Simulador. OK fase-1

C.4.3. Ejecutar un programa

Una vez compilado el programa fuente en lenguaje PFGUnedTLP se habilitan las funciones necesarias para ejecutar el programa como podemos ver en la figura C.5 en la cual podemos distinguir una serie de zonas bien diferenciadas para adaptar la ejecución del programa a nuestras necesidades para el seguimiento de la evolución de los registros de activación y el estado del cómputo.

- **Salida por pantalla**

  Es un área destinada a simular la salida por pantalla de las instrucciones del código fuente que lo requieran.

- **Opciones visualización de datos**

  Son una serie de opciones que nos ofrece la aplicación para facilitar la visualización de los datos de acuerdo al uso que necesitemos de ellos durante la ejecución del programa.

- **Botón ejecución completa**

  Botón que nos permite ejecutar el programa hasta su finalización.

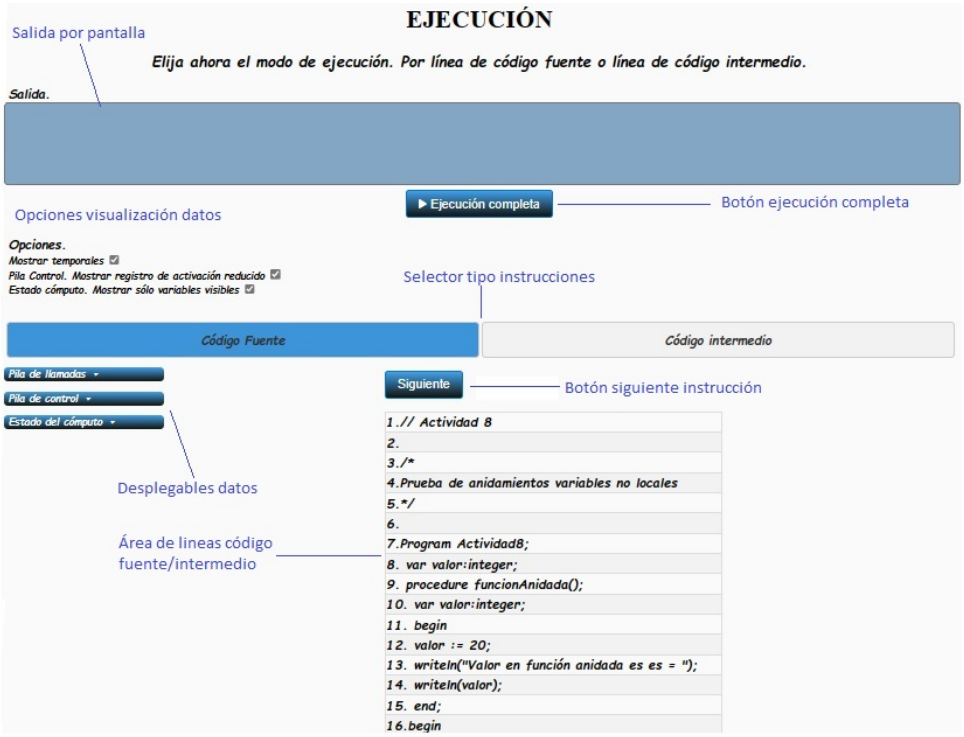

Figura C.5: Página Simulador. Fase 2.

- **Selector tipo instrucciones**

  Selector que nos permite elegir si queremos ver las instrucciones de código fuente o código intermedio del programa compilado.

- **Desplegables de datos**

  Desplegables donde se pueden la evolución de los diferentes tipos de datos que se van generando durante la ejecución del programa.

- **Botón siguiente instrucción**

  Botón que nos permite avanzar una instrucción durante la ejecución del programa. El tipo de instrucción que se consume depende de si en el selector hemos seleccionado líneas de código fuente o de código intermedio.

- **Área de líneas código fuente/intermedio**

  Área donde se muestran las líneas de código fuente o de código intermedio que tiene nuestro programa compilado.

Como vemos en la figura C.5 el sistema nos ofrece la posibilidad de ejecutarlo de dos maneras dependiendo de lo que elijamos en en área de "Selector tipo instrucciones", mediante instrucciones de código fuente o acercándonos un poco a código máquina, ejecutando líneas de código intermedio.

Al ir consumiendo instrucciones de código fuente o instrucciones de código intermedio según el modo elegido mediante el "Botón siguiente instrucción" o el "Botón ejecución completa", se va marcando en azul dentro de el "Área de líneas de código fuente/intermedio"la línea o instrucción que se acaba de procesar, y a la vez podemos ver cómo en el área "Desplegable de

datos"se van creando y eliminando los registros de activación en la "Pila de control", cómo van cambiado las variables dentro del "Estado del cómputo" y cómo se van creando y eliminando llamadas dentro de la "Pila de llamadas" como podemos ver en la figura C.6.

Asimismo, en área de "Salida por pantalla" se puede ir viendo lo que va escribiendo el programa por pantalla.

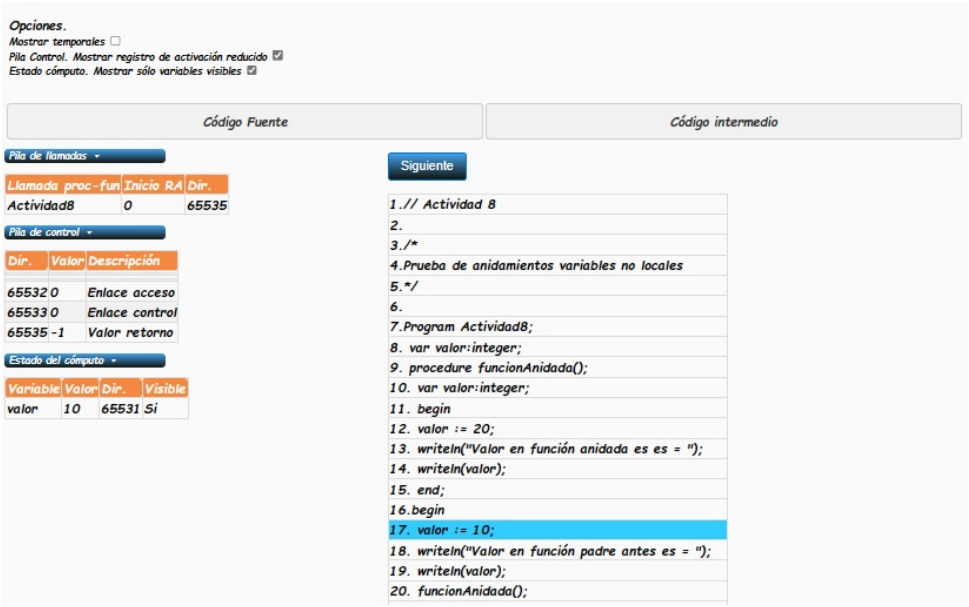

Figura C.6: Página Simulador. Ejecución Fase 2.

Opciones

Como vemos en la figura C.5 se han habilitado en el área "Opciones visualización de datos" tres opciones para facilitar el tratamiento de los datos en el uso de la aplicación.

- **Mostrar temporales**

  Si se marca esta opción, en las zonas de "Estado del cómputo" y "Pila de control" se muestran los temporales que usa el programa internamente para poder llegar al resultado. Estos temporales comienzan todos por T_ y su uso se ve claramente al ir avanzando instrucciones de código intermedio.

- **Pila Control. Mostrar registro de activación reducido**

  Permite la posibilidad de sólo mostrar tres elementos de cada registro de activación que son "Valor de retorno", "Enlace de control" y "Enlace de acceso". Esta opción tiene preferencia sobre la de "Mostrar temporales", por lo que si está marcada no se mostrarán los temporales aunque esté activa la opción para mostrarlos.

  Al activar este modo se crea una pequeña separación visual entre cada registro de activación con motivo de facilitar la distinción de cada registro de activación.

- **Estado cómputo. Mostrar sólo variables visibles**

  Al marcar esta opción el "Estado del cómputo" solo muestra las variables que son visibles en el momento de ejecución.

Como podemos observar en la figura C.7 el uso de opciones hace en un momento determinado de la ejecución de nuestro programa compilado sólo se muestren los datos que hemos elegido dentro del área “Opciones visualización de datos“.

El uso de estas opciones o veremos mas adelante en profundidad dentro de la sección de ayudas C.4.3

En el caso de que durante la ejecución de nuestro programa compilado alguna de las instrucciones genere una salida por pantalla, podemos ir viendo esas salidas que se van generando en el área ”Salida por pantalla”habilitada para ello como podemos ver en la figura C.8.

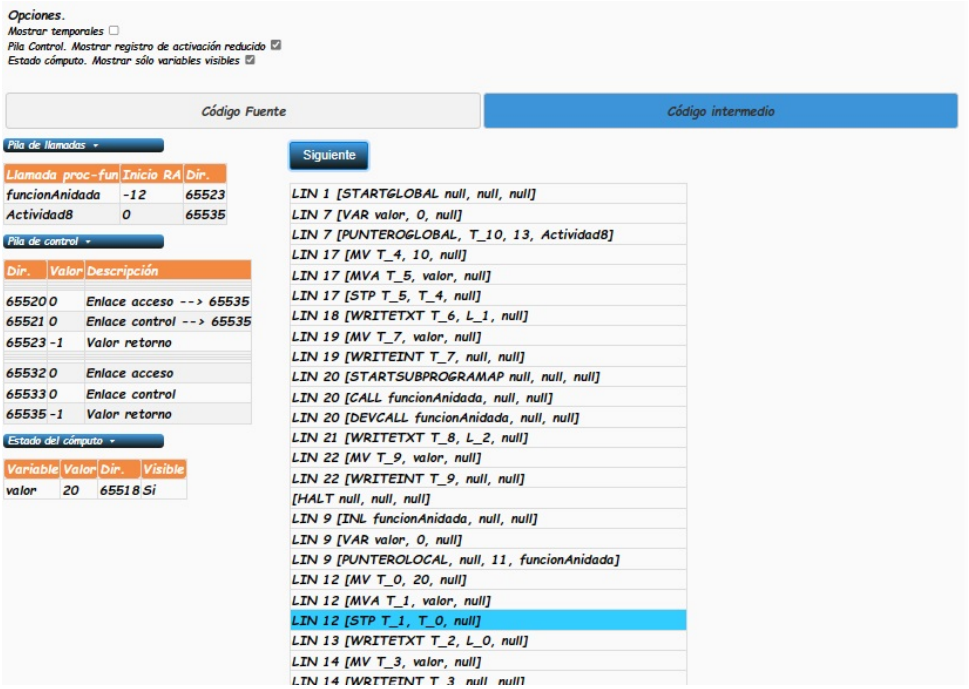

Figura C.7: Página Simulador. Ejecución en curso Fase 2.

Al continuar con la ejecución, como podemos ver en la figura C.8 y una vez se ejecutan todas las instrucciones, el programa llega a su fin, con lo que se elimina la posibilidad de seguir consumiendo instrucciones mediante la ocultación de los botones que nos permiten consumir instrucciones y a su vez, se queda todo el código marcado en azul para indicar que no hay instrucciones por ejecutar.

Además, perdemos otra serie de opciones como son el ”Botón ejecución completa”, el cual, si se pincha en cualquier momento de la ejecución de nuestro programa compilado, consume todas las instrucciones disponibles hasta llegar al final de la ejecución.

Por su parte, el ”Botón reinicio”, puede usarse en cualquier momento tanto en fase de compilación como en la de ejecución, el cual, elimina todos los datos que aparecen por pantalla y deja el compilador en su estado inicial, es decir, sin datos y listo para comenzar otra vez a introducir un nuevo programa.

Figura C.8: Página Simulador. Fin Fase 2.

Ayudas

Como hemos visto en la figura C.5, con motivo de facilitar el estudio, se han habilitado en el área "Opciones visualización de datos" una serie de ayudas visuales dentro de la "Pila de llamadas", "Pila de control" y "Estado del cómputo".

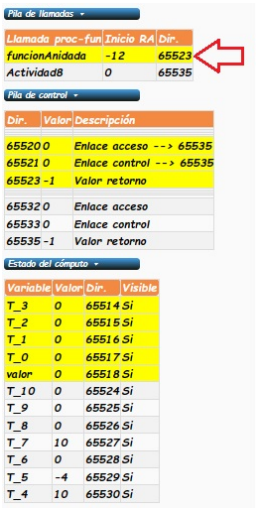

Figura C.9: Página Simulador. Click Pila LLamadas.

Dentro de la "Pila de llamadas", como podemos ver en la figura C.9, si hacemos click en cualquiera de sus registros, nos iluminará en color amarillo los registros de "Pila de control" y "Estado del cómputo" asociados al registro de la "Pila de llamadas" que hemos marcado.

Del mismo modo, como podemos ver en la figura C.10, si hacemos click en un registro de la "Pila de control", nos iluminará en amarillo el registro de la "Pila de llamadas" al que pertenece.

Si a su vez el registro marcado en un "Enlace de acceso" o "Enlace de control", nos marcará en rojo el registro de la "Pila de llamadas" y de la "Pila de control" a la que apunta.

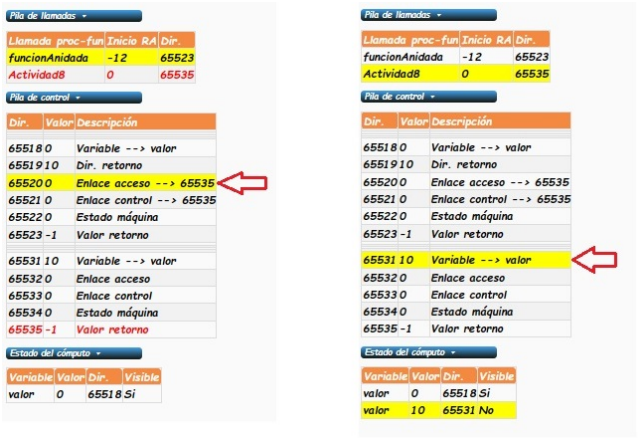

Figura C.10: Página Simulador. A la izquierda Click registro Pila Control de tipo enlace, a la derecha Click registro Pila Control de tipo variable.

Si el registro que marcamos en la "Pila de control" pertenece a una variable, parámetro o temporal, además de marcar en la "Pila de llamadas" el registro al que pertenece, ilumina en color amarillo el registro correspondiente del "Estado del cómputo".

Por último, como podemos ver en la figura C.11,si hacemos click sobre un registro del "Estado del cómputo", nos ilumina en color amarillo el registro de la "Pila de llamadas" al que pertenece y la posición de la "Pila de control" donde está almacenado.

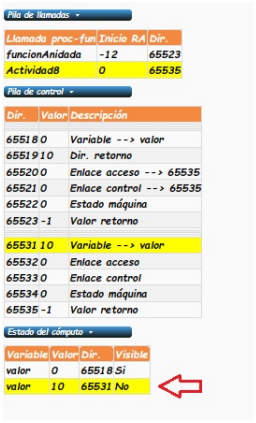

Figura C.11: Página Simulador. Click registro del Estado Computo.

C.5. Lenguaje

Desde dentro de la aplicación, como podemos ver en la figura C.12 podemos acceder a la página "Lenguaje", donde el usuario puede consultar en línea las especificaciones del lenguaje que podemos ver en el apéndice A .

Figura C.12: Página Lenguaje.

C.6. Ayuda

De la misma manera que con la especificación del lenguaje, como podemos ver en la figura C.13, podemos acceder desde la web a una ayuda el línea para consultar el funcionamiento del programa. Esta ayuda es similar a la que podemos ver dentro del apéndice C en el apartado Simulador C.4.

Figura C.13: Página Ayuda.

Apéndice D

Manual de despliegue

Con motivo de facilitar al máximo el despliegue del proyecto en un entorno de producción, el proyecto se ha desarrollado de modo que no contenga ningún tipo de dependencias con otros programas externos a la aplicación, con lo cual no hace falta disponer nada mas que un servidor web para alojar la estructura de ficheros que componen el programa y un simple navegador web.

El proceso a realizar es el siguiente:

- Realizar una copia del directorio WEBPFG que contiene el proyecto en un directorio accesible por el servidor web que queremos que aloje el sitio web.

  Es fundamental que que se conserve la estructura de los ficheros que contiene el directorio WEBPFG, ya que si se varía o elimina alguno de los elementos que lo componen dará lugar a que el proyecto no se ejecute correctamente.

- Enlazar el proyecto a la dirección que desee apuntando al fichero *index.html* que se encuentra en el directorio WEBPFG o enlazar este fichero *index.html* desde alguna otra página ya publicada en el servidor web para el acceso a la aplicación.

  Dentro del manual de configuración e instalación del entorno de desarrollo que podemos ver en el apéndice G se puede encontrar información básica sobre el servidor web que se ha usado para el desarrollo del proyecto.

  Se puede probar el proyecto en modo local sin necesidad de publicarlo en un servidor web ejecutando directamente el fichero *index.html* desde un navegador web, siempre y cuando no se modifique la estructura del proyecto que se encuentra en el directorio WEBPFG.

Apéndice E

Reglas léxicas.

```
1
2   /*
3      PFG UNED. Jison utilizando Node en Windows
4      Fichero : parserUned.jison
5      Autor   : Jaime Alcalá Galicia
6      Version : 1.0
7      Fecha   : 01/01/2023
8      Descripción: Análisis léxico.
9   */
10
11
12  %lex
13  %options case-sensitive
14  %%
15
16
17  /* OPERADORES Y PALABRAS RESERVADAS */
18
19  "Program"           return 'PROGRAM';
20  ";"                 return 'PTOYCOMA';
21  "function"          return 'FUNCTION';
22  "procedure"         return 'PROCEDURE';
23  "("                 return 'PARIZDO';
24  ")"                 return 'PARDCHO';
25  "integer"           return 'INTEGER';
26  ":="                return 'ASIGNACION';
27  ":"                 return 'DOSPUNTOS';
28  "var"               return 'VAR';
29  "+"                 return 'MAS';
30  "-"                 return 'MENOS';
31  "."                 return 'PUNTO';
32  "if"                return 'IF';
33  "else"              return 'ELSE';
34  "then"              return 'THEN';
35  "begin"             return 'BEGIN';
36  "end"               return 'END';
37  ","                 return 'COMA';
38  "="                 return 'IGUAL';
39  "writeln"           return 'WRITELN';
40  "exit"              return 'EXIT';
41
42
43  /* ESPACIOS EN BLANCO */
44
45  \s+                 // Elimina los espacios en blanco
46  [ \t\n\r]+          {/*Ignoramos estos caracteres. Tabulacion, salto de
47                        línea y retorno de carro*/}
48
49
```

```
50  /* MACROS */
51
52  [1-9][0-9]*|0                       return 'LITERALENTERO'
53  [0-9]+\b                            return 'ENTERO';
54  [0-9]                               return 'DIGITO';
55  [1-9]                               return 'DIGITOPOS';
56  [a-zA-Z][a-zA-Z0-9]*                return 'IDENTIFICADOR';
57  [a-zA-Z]                            return 'LETRA';
58  \"([^\"])*\"                        return 'CADENATEXTO'
59
60
61  /* COMENTARIOS */
62
63  "//".*                                          // Comentario una línea
64  [/][*][^*]*[*]+([^/*][^*]*[*]+)*[/]             // Comentario múltiple
65
66
67  <<EOF>>                 return 'EOF';
68
69  .                       {this.parseError("Línea " + yylloc.first_line +
70                           ". Carácter no permitido : " + yytext ,
71                            { text : yytext , token : null ,
72                            line : yylloc.first_line -1}) ;}
73
74  /lex
75
76  %{
77  %}
78
79  /* ASOCIACION DE OPERADORES Y PRECEDENCIA */
80
81  %left   'IGUAL'
82  %left   'MAS' 'MENOS'
83  %left   'PARIZDO' 'PARDCHO'
```

Apéndice F

Reglas gramaticales.

A continuación se muestra el código Jison empleado para construir las reglas gramaticales junto con las acciones semánticas y la generación de código intermedio.

```
1
2   /*
3      PFG UNED. Jison utilizando Node en Windows
4      Fichero : parserUned.jison
5      Autor   : Jaime Alcalä Galicia
6      Version : 1.0
7      Fecha   : 01/01/2023
8      Descripciön: Anälisis lëxico, sintäctico, semänto y cödigo intermedio.
9   */
10
11  %% /* Definimos la gramätica */
12
13  inic
14  : instrucciones EOF {
15  ;
16
17  instrucciones
18  : inicioPrograma PTOYCOMA
19      seccionVariables
20      listaFuncionesProcedimientos
21    BEGIN
22      cuerpoBloque
23    END PUNTO
24    {
25      $$=     {intCode : [],TamaRA:0};
26      $$.intCode = $1.intCode;
27      TamaRA = yy.tamanoRA(yy.traeNombreAmbitoActual())+1;
28      if (typeof $3 !== "undefined") {
29          for (let step = 0; step < $3.intCode.length; step++) {
30              $$.intCode = $$.intCode.concat("LIN " + this._$.first_line +
31                          $3.intCode[step]);
32      }
33      }
34      $$.intCode = $$.intCode.concat("LIN " + this._$.first_line +
35          "  [PUNTEROGLOBAL, " + yy.creaTemporal() + ", " + TamaRA + ", " +
36          yy.traeNombreAmbitoActual() + "]");
37      if (typeof $6 !== "undefined") {
38          $$.intCode = $$.intCode.concat($6.intCode);
39      }
40      $$.intCode = $$.intCode.concat("[HALT null, null, null]");
41      if (typeof $4 !== "undefined") {
42      $$.intCode = $$.intCode.concat($4.intCode);
43      }
44  }
45  ;
```

```
46
47  inicioPrograma : PROGRAM IDENTIFICADOR
48  {
49      $$=     {intCode : []};
50      yy.abrirAmbito($2);
51      yy.tablaTipos.set("integer", yy.traeNombreAmbitoActual());
52      yy.tablaTipos.set("boolean", yy.traeNombreAmbitoActual());
53  }
54  ;
55
56  seccionVariables : VAR listaVariables
57  {
58      $$=     {intCode : [], lVar:[]};
59      if (typeof $2 !== "undefined") {
60          $$.intCode = $$.intCode.concat($2.intCode);
61          $$.lVar = $$.lVar.concat($2.lVar);
62          yy.insertaVariablesEnAmbito($$.lVar);
63      }
64  }
65  |;
66
67  listaVariables : decVariable listaVariables
68  {
69      $$=     {intCode : [], lVar:[]};
70      $$.intCode = $1.intCode;
71      $$.lVar = $1.lVar;
72      if (typeof $2 !== "undefined") {
73          $$.intCode = $$.intCode.concat($2.intCode);
74          $$.lVar = $$.lVar.concat($2.lVar);
75      }
76  }
77  |;
78
79  decVariable : decVariables DOSPUNTOS INTEGER PTOYCOMA
80  {
81      $$=     {intCode : [], lVar:[]};
82      $$.intCode = $$.intCode.concat($1.intCode);
83      $$.lVar = $$.lVar.concat($1.lVar);
84  }
85  ;
86
87  decVariables : IDENTIFICADOR COMA decVariables
88  {
89      $$=     {intCode : [], lVar:[]};
90      $$.intCode.push(" [VAR " + $1 + ", 0, null]");
91      $$.lVar.push($1);
92      $$.intCode = $$.intCode.concat($3.intCode);
93      $$.lVar = $$.lVar.concat($3.lVar);
94      yy.nuevoSimboloVariable(this._$.last_line, $1, "integer");
95      }
96  |IDENTIFICADOR
97  {
98      $$=     {intCode : [], lVar:[]};
99      yy.nuevoSimboloVariable(this._$.last_line, $1, "integer");
100     $$.intCode.push(" [VAR " + $1 + ", 0, null]");
101     $$.lVar.push($1);
102 }
103 ;
```

```
104
105 listaFuncionesProcedimientos : procedimientoOFuncion
106                               listaFuncionesProcedimientos
107 {
108     $$=     {intCode : []};
109     $$.intCode = $1.intCode;
110     if (typeof $2 !== "undefined") {
111         $$.intCode = $$.intCode.concat($2.intCode);
112     }
113 }
114 |;
115
116 procedimientoOFuncion: decProcedimiento
117 {
118     $$=     {intCode : []};
119     $$.intCode = $1.intCode;}
120 |decFuncion{
121     $$=     {intCode : []};
122     $$.intCode = $1.intCode;}
123 ;
124
125 decFuncion : inicioFuncion PARIZDO decParametros PARDCHO
126             DOSPUNTOS INTEGER PTOYCOMA
127                 seccionVariables
128                 listaFuncionesProcedimientos
129             BEGIN
130                 cuerpoBloque
131                 retornoFuncion
132             END PTOYCOMA
133 {
134     $$=     {intCode : [],TamaRA:0};
135     $$.intCode = $1.intCode;
136     if (typeof $3 !== "undefined") {
137         $$.intCode = $$.intCode.concat($3.intCode);
138     }
139     if (typeof $8 !== "undefined") {
140         for (let step = 0; step < $8.intCode.length; step++) {
141             $$.intCode = $$.intCode.concat("LIN " + this._$.first_line +
142             $8.intCode[step]);
143         }
144     }
145     TamaRA = yy.tamanoRA(yy.traeNombreAmbitoActual())+1;
146     $$.intCode = $$.intCode.concat("LIN " + this._$.first_line +
147     " [PUNTEROLOCAL, null, " + TamaRA + ", " + yy.traeNombreAmbitoActual() +
148     "]");
149     if (typeof $11 !== "undefined") {
150         $$.intCode = $$.intCode.concat($11.intCode);
151     }
152     if (typeof $12 !== "undefined") {
153         $$.intCode = $$.intCode.concat($12.intCode);
154     }
155     $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
156     " [FINSUBPROGRAMA, " + $1.nombre + ", null, null]");
157     if (typeof $9 !== "undefined") {
158         $$.intCode = $$.intCode.concat($9.intCode);
159     }
160     yy.cierraAmbitoActual();
161     }
```

```
162 ;
163
164 decProcedimiento : inicioProcedimiento PARIZDO decParametros
165                                                 PARDCHO PTOYCOMA
166                 seccionVariables
167                 listaFuncionesProcedimientos
168             BEGIN
169                 cuerpoBloque
170             END PTOYCOMA
171 {
172     $$=     {intCode : [],TamaRA:0};
173     $$.intCode = $1.intCode;
174     if (typeof $3 !== "undefined") {
175         $$.intCode = $$.intCode.concat($3.intCode);
176     }
177     if (typeof $6 !== "undefined") {
178         for (let step = 0; step < $6.intCode.length; step++) {
179             $$.intCode = $$.intCode.concat("LIN " + this._$.first_line +
180                                             $6.intCode[step]);
181         }
182     }
183     TamaRA = yy.tamanoRA(yy.traeNombreAmbitoActual())+1;
184     $$.intCode = $$.intCode.concat("LIN " + this._$.first_line +
185                 " [PUNTEROLOCAL, null, " + TamaRA + ", " +
186                 yy.traeNombreAmbitoActual() + "]");
187     if (typeof $9 !== "undefined") {
188         $$.intCode = $$.intCode.concat($9.intCode);
189     }
190     if (typeof $10 !== "undefined") {
191         $$.intCode = $$.intCode.concat($10.intCode);
192     }
193     $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
194                 " [FINSUBPROGRAMA, " + $1.nombre + ", null, null]");
195     if (typeof $7 !== "undefined") {
196         $$.intCode = $$.intCode.concat($7.intCode);
197     }
198     yy.cierraAmbitoActual();
199     }
200 ;
201
202 inicioFuncion : FUNCTION IDENTIFICADOR
203 {
204     $$=     {intCode : [],nombre:""};
205     $$.nombre = $2;
206     yy.nuevoSimboloFuncion(this._$.last_line,$2,"integer",[]);
207     $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
208                 " [INL " + $2 +  ", null, null]");
209     yy.abrirAmbito($2);
210     }
211 ;
212
213 inicioProcedimiento : PROCEDURE IDENTIFICADOR
214 {
215     $$=     {intCode : [],nombre:""};
216     $$.nombre = $2;
217     yy.nuevoSimboloProcedimiento(this._$.last_line,$2,[]);
218     $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
219                 " [INL " + $2 +  ", null, null]");
```

```
220         yy.abrirAmbito($2);
221         }
222     ;
223
224     decParametros : IDENTIFICADOR DOSPUNTOS INTEGER
225     {
226         $$=     {parametros : []};
227         yy.nuevoSimboloParametro(this._$.last_line, $1, "integer",
228                                  yy.traeNombreAmbitoActual());
229         $$.parametros.push($1);
230     }
231     |IDENTIFICADOR DOSPUNTOS INTEGER COMA decParametros
232     {
233         $$=     {parametros : []};
234         yy.nuevoSimboloParametro(this._$.last_line, $1, "integer",
235                                  yy.traeNombreAmbitoActual());
236         $$.parametros.push($1);
237         $$.parametros = $$.parametros.concat($5.parametros);
238     }
239     |;
240
241     cuerpoBloque :  cuerpoBloque unaSentencia
242     {
243         $$=     {intCode : []};
244         if (typeof $1 !== "undefined") {
245             $$.intCode = $$.intCode.concat($1.intCode);
246         }
247         if (typeof $2 !== "undefined") {
248             $$.intCode = $$.intCode.concat($2.intCode);
249         }
250         }
251     |;
252
253     unaSentencia : sentenciaAsignacion
254     {
255         {intCode : []};
256         $$.intCode = $1.intCode;
257     }
258     |sentenciaSi
259     {
260         {intCode : []};
261         $$.intCode = $1.intCode;
262         }
263     |sentenciaSalida
264     {
265         {intCode : []};
266         $$.intCode = $1.intCode;
267     }
268     |expLlamadaProcedimiento
269     {
270         {intCode : []};
271         $$.intCode = $1.intCode;
272         }
273     ;
274
275     sentenciaAsignacion : IDENTIFICADOR ASIGNACION expresionAritmetica PTOYCOMA{
276         $$=     {intCode : [],Ta:"",Tb:""};
277         yy.errorVisibilidad($1,this._$.last_line);
```

```
278         yy.errorNoEsVariableoParametro($1,this._$.last_line);
279         Ta = yy.creaTemporal();
280         Tb=$3.varTemp;
281         $$.intCode = $$.intCode.concat($3.intCode);//Parte expresion aritmetica
282         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
283                 " [MVA " + Ta + ", " +  $1 + ", null]");//parte izda asignacion
284         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
285                 " [STP " + Ta + ", " + Tb + ", null]");
286         }
287     ;
288
289     sentenciaSalida : WRITELN PARIZDO CADENATEXTO PARDCHO PTOYCOMA{
290         $$=     {intCode : [],La:"",Ta:""};
291         La = yy.creaCadenaTxt($3);
292         Ta = yy.creaTemporal();
293         $$.intCode = this.$.intCode.concat($3.intCode );
294         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
295                     " [WRITETXT " + Ta + ", " + La + ", null]");
296         }
297     |WRITELN PARIZDO expresionAritmetica PARDCHO PTOYCOMA{
298         $$=     {intCode : []};
299         $$.intCode = this.$.intCode.concat($3.intCode );
300         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
301                 " [WRITEINT " + $3.varTemp + ", null, null]");
302     }
303     ;
304
305     sentenciaSi : IF expresionLogica THEN
306                 BEGIN
307                     cuerpoBloque
308                 END
309                     sentenciaSinoOpcional
310     {
311         $$=     {intCode : [], La:"", Lb:""};
312         La = yy.creaEtiqueta();
313         Lb = yy.creaEtiqueta();
314         $$.intCode = $$.intCode.concat( $2.intCode );
315         if (typeof $7 !== "undefined") {
316             $$.intCode = $$.intCode.concat("LIN " + $7.qLin +
317                         " [BRF " + $2.varTemp+ ", " + La + ", null]");
318         }
319         else
320         {
321             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
322                         " [BRF " + $2.varTemp+ ", " + La + ", null]");
323         }
324         if (typeof $5 !== "undefined") {
325             $$.intCode = $$.intCode.concat($5.intCode);
326         }
327         if (typeof $7 !== "undefined") {
328             $$.intCode = $$.intCode.concat("LIN " + $7.qLin +
329                             " [BR " + Lb +  ", null, null]");
330             $$.intCode = $$.intCode.concat("LIN " + $7.qLin +
331                             " [INL " + La +  ", null, null]");
332             $$.intCode = $$.intCode.concat($7.intCode);
333         }
334         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
335                             " [INL " + Lb +  ", null, null]");
```

```
336 }
337 ;
338
339 sentenciaSinoOpcional : ELSE
340                       BEGIN
341                           cuerpoBloque
342                       END PTOYCOMA
343 {
344     $$={intCode : [], qLin:""};
345     if (typeof $3 !== "undefined"){
346         $$.qLin = this._$.first_line;
347         $$.intCode = $$.intCode.concat($3.intCode);
348     }
349 }
350 |PTOYCOMA
351 ;
352
353 expresionAritmetica : LITERALENTERO
354 {
355     $$=     {intCode : [], varTemp: ""};
356     $$.varTemp =yy.creaTemporal();
357     $$.intCode.push("LIN " + this._$.last_line + " [MV " + $$.varTemp +
358                     ", " + $1 + ", null]" );
359 }
360 |PARIZDO expresionAritmetica PARDCHO
361 {
362     $$=     {intCode : [], varTemp: ""};
363     $$.varTemp = $2.varTemp;
364     $$.intCode.push( $2.intCode);
365 }
366 |expresionAritmetica MAS expresionAritmetica
367 {
368     $$=     {intCode : [], La: "", varTemp: ""};
369     La = yy.creaTemporal();
370     $$.intCode = $$.intCode.concat($1.intCode);
371     $$.intCode = $$.intCode.concat($3.intCode);
372     $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
373         " [ADD " + La + ", " + $1.varTemp+ ", " + $3.varTemp + "]");
374     $$.varTemp = La;
375 }
376 |expresionAritmetica MENOS expresionAritmetica
377 {
378     $$=     {intCode : [], La: "", varTemp: ""};
379     La = yy.creaTemporal();
380     $$.intCode = $$.intCode.concat($1.intCode);
381     $$.intCode = $$.intCode.concat($3.intCode);
382     $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
383         " [SUB " + La + ", " + $1.varTemp+ ", " + $3.varTemp + "]");
384     $$.varTemp = La;
385 }
386 |expLlamadaFuncion
387 {
388     $$=     {intCode : [], varTemp: ""};
389     $$.varTemp = $1.varTemp;
390     $$.intCode = $$.intCode.concat($1.intCode);
391 }
392 |IDENTIFICADOR
393 {
```

```
394         $$=     {intCode : [], varTemp: ""};
395         yy.errorVisibilidad($1,this._$.last_line);
396         yy.errorNoEsVariableoParametro($1,this._$.last_line);
397         $$.varTemp =yy.creaTemporal();
398         $$.intCode.push("LIN " + this._$.last_line + " [MV " + $$.varTemp +
399                         ", " + $1  + ", null]");
400     }
401     ;
402
403     retornoFuncion : EXIT PARIZDO IDENTIFICADOR PARDCHO PTOYCOMA
404     {
405         $$= {intCode : []};
406         yy.errorVisibilidad($3,this._$.last_line);
407         yy.errorNoEsVariableoParametro($3,this._$.last_line);
408         $$.intCode.push("LIN " + this._$.last_line + " [EXIT " + $3 +
409                         ", null, null]");
410     };
411
412     expresionLogica : PARIZDO expresionLogica PARDCHO
413     {
414         $$=     {intCode : [],varTemp: ""};
415         $$.varTemp =$1.varTemp;
416         $$.intCode.push($2.intCode);
417     }
418     |LITERALENTERO IGUAL LITERALENTERO
419     {
420         $$=     {intCode : [], Ta: "", Tb: "",varTemp: ""};
421         Ta = yy.creaTemporal();
422         Tb= yy.creaTemporal();
423         $$.varTemp = yy.creaTemporal();
424         $$.intCode.push("LIN " + this._$.last_line +
425                             " [MV " + Ta + ", " +  $1 + ", null]");
426         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
427                             " [MV " + Tb + ", " +  $3 + ", null]");
428         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
429                             " [EQ " + $$.varTemp + ", " + Ta + ", " + Tb +"]");
430     }
431     |IDENTIFICADOR IGUAL IDENTIFICADOR
432     {
433         $$=     {intCode : [], Ta: "", Tb: "",varTemp: ""};
434         yy.errorVisibilidad($1,this._$.last_line);
435         yy.errorVisibilidad($3,this._$.last_line);
436         yy.errorNoEsVariableoParametro($1,this._$.last_line);
437         yy.errorNoEsVariableoParametro($3,this._$.last_line);
438         Ta = yy.creaTemporal();
439         Tb = yy.creaTemporal();
440         $$.varTemp = yy.creaTemporal();
441         $$.intCode.push("LIN " + this._$.last_line + " [MV " + Ta + ", " +
442                         $1 + ", null]");
443         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
444                         " [MV " + Tb + ", " +  $3 + ", null]");
445         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
446                         " [EQ " + $$.varTemp + ", " + Ta + ", " + Tb +"]");
447     }
448     |LITERALENTERO IGUAL IDENTIFICADOR
449     {
450         $$=     {intCode : [], Ta: "", Tb: "",varTemp: ""};
451         yy.errorVisibilidad($3,this._$.last_line);
```

```
452         yy.errorNoEsVariableoParametro($3,this._$.last_line);
453         Ta = yy.creaTemporal();
454         Tb = yy.creaTemporal();
455         $$.varTemp = yy.creaTemporal();
456         $$.intCode.push("LIN " + this._$.last_line + " [MV " + Ta + ",
457                         " +  $1 + ",null]");
458         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
459                         " [MV " + Tb + ", " +  $3 + ",null]");
460         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
461                         " [EQ " + $$.varTemp + ", " + Ta + ", " + Tb+"]");
462     }
463     |IDENTIFICADOR IGUAL LITERALENTERO
464     {
465         $$=     {intCode : [], Ta: "", Tb: "",varTemp: ""};
466         yy.errorVisibilidad($1,this._$.last_line);
467         yy.errorNoEsVariableoParametro($1,this._$.last_line);
468         Ta = yy.creaTemporal();
469         Tb = yy.creaTemporal();
470         $$.varTemp = yy.creaTemporal();
471         $$.intCode.push("LIN " + this._$.last_line +
472                             " [MV " + Ta + ", " +  $1 + ", null]");
473         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
474                             " [MV " + Tb + ", " +  $3 + ", null]");
475         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
476                             " [EQ " + $$.varTemp + ", " + Ta + ", " + Tb+"]");
477     }
478     ;
479
480     expLlamadaFuncion : IDENTIFICADOR PARIZDO llamadaParametros PARDCHO
481     {
482         $$=     {intCode : [],NoParams:0,varTemp: ""};
483         yy.errorVisibilidad($1,this._$.last_line);
484         yy.errorNoEsFuncion($1,this._$.last_line);
485         if (typeof $3 !== "undefined") {
486             $$.varTemp = yy.creaTemporal();
487             $$.NoParams = $3.NoParams;}
488             yy.compruebaNumeroParametros($1,$$.NoParams,this._$.last_line);
489             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
490                             " [STARTSUBPROGRAMAF null, null, null ]");
491             $$.intCode = $$.intCode.concat($3.intCode);
492             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
493                             " [CALL " + $1 +  ", null, null]");
494             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
495                             " [DEVCALL " + $1 + ", " + $$.varTemp + ", null]");
496     }
497     ;
498     expLlamadaProcedimiento : IDENTIFICADOR PARIZDO
499                                                 llamadaParametros
500                                             PARDCHO PTOYCOMA
501     {
502         $$=     {intCode : [],NoParams:0};
503         yy.errorVisibilidad($1,this._$.last_line);
504         yy.errorNoEsProcedimiento($1,this._$.last_line);
505         if (typeof $3 !== "undefined") {
506             $$.NoParams = $3.NoParams;}
507             yy.compruebaNumeroParametros($1,$$.NoParams,this._$.last_line);
508             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
509                                 " [STARTSUBPROGRAMAP null, null, null]");
```

```
510             if (typeof $3 !== "undefined") {
511                 $$.intCode = $$.intCode.concat($3.intCode);
512             }
513             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
514                                     " [CALL " + $1 +  ", null, null]");
515             $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
516                                     " [DEVCALL " + $1 +  ", null, null]");
517     }
518     ;
519
520     llamadaParametros : expresionParametros
521     {
522         $$=     {intCode : [],NoParams:0};
523         $$.NoParams = $$.NoParams + 1;
524         $$.intCode = $1.intCode;
525         $$.intCode.push("LIN " + this._$.last_line + " [PARAM " + $1.varTemp +
526                         ", null, null]");
527     }
528     |expresionParametros COMA llamadaParametros
529     {
530         $$=     {intCode : [],NoParams:0};
531         $$.NoParams = $$.NoParams + 1;
532         $$.intCode = $1.intCode;
533         $$.intCode.push("LIN " + this._$.last_line +
534                         " [PARAM " + $1.varTemp + ", null, null]");
535         if (typeof $3 !== "undefined") {
536             $$.intCode = $$.intCode.concat($3.intCode);
537             $$.NoParams = $$.NoParams  + $3.NoParams;
538         }
539     }
540     |;
541
542     expresionParametros : LITERALENTERO
543     {
544         $$=     {intCode : [], varTemp: ""};
545         $$.varTemp =yy.creaTemporal();
546         $$.intCode.push("LIN " + this._$.last_line +
547                         " [MV " + $$.varTemp +", " +  $1 + ", null]" );
548     }
549     |PARIZDO expresionParametros PARDCHO
550     {
551         $$=     {intCode : [], varTemp: ""};
552         $$.varTemp = $2.varTemp;
553         $$.intCode.push( $2.intCode);
554     }
555     |expresionParametros MAS expresionParametros
556     {
557         $$=     {intCode : [], La: "", varTemp: ""};
558         La = yy.creaTemporal();
559         $$.intCode = $$.intCode.concat($1.intCode);
560         $$.intCode = $$.intCode.concat($3.intCode);
561         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
562                     " [ADD " + La + ", " + $1.varTemp+ ", " + $3.varTemp + "]");
563         $$.varTemp = La;
564     }
565     |expresionParametros MENOS expresionParametros
566     {
567         $$=     {intCode : [], La: "", varTemp: ""};
```

```
568         La = yy.creaTemporal();
569         $$.intCode = $$.intCode.concat($1.intCode);
570         $$.intCode = $$.intCode.concat($3.intCode);
571         $$.intCode = $$.intCode.concat("LIN " + this._$.last_line +
572                     " [SUB " + La + ", " + $1.varTemp+ ", " + $3.varTemp + "]");
573         $$.varTemp = La;
574     }
575     |IDENTIFICADOR
576     {
577         $$=     {intCode : [], varTemp: ""};
578         yy.errorVisibilidad($1,this._$.last_line);
579         yy.errorNoEsVariableoParametro($1,this._$.last_line);
580         $$.varTemp =yy.creaTemporal();
581         $$.intCode.push("LIN " + this._$.last_line +
582                         " [MVP " + $$.varTemp + ", " + $1  + ", null]");
583     }
584     ;
```

Apéndice G

Instalación y configuración del entorno de desarrollo.

En este apartado de describen todas las tareas que se han realizado para la instalación y configuración de los principales componentes usados en el entorno de desarrollo, necesarios para la realización del proyecto.

G.1. Node

Node.js es un entorno en tiempo de ejecución para JavaScript que contiene todo lo necesario para ejecutar JavaScript fuera de un explorador. Su gestor de paquetes por defecto npm se desplegará el entorno para usar Jison y sus scripts.

Podemos descargar el instalador directamente desde la página de node.js [1]

En la siguiente figura G.1 podemos ver un fragmento con las opciones de instalación por defecto de Node.js.

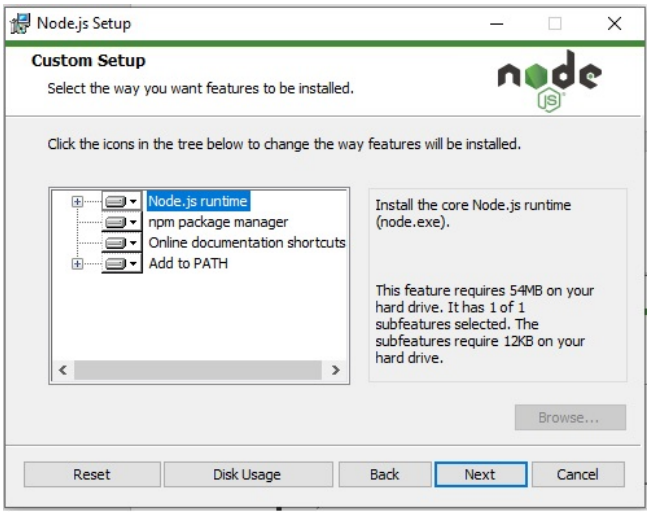

Figura G.1: Instalación de Node.js

[1] https://nodejs.org/es

G.2. Jison

Jison es el entorno donde vamos a generar nuestro analizador.

Como podemos ver en la figura G.2 se instala directamente a través de npm desde la línea de comandos .

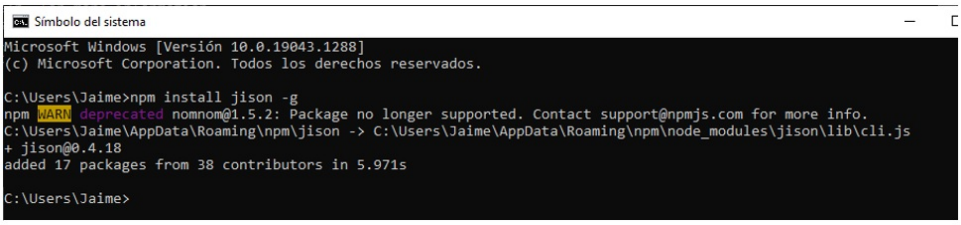

Figura G.2: Instalación de Jison

G.3. Visual Studio Code

Visual Studio Code es el IDE que hemos elegido para el desarrollo del entorno web.

Su instalación es muy sencilla a través de su asistente, y puede descargarse directamente de la página de Visual Studio Code [2].

G.4. GitHub

GitHub es el programa usado durante el desarrollo para el control de versiones.

Podemos encontrar una extensa y completa documentación sobre la instalación y uso de Github junto a Visual Studio Code en la propia web de microsoft [3].

G.5. Astah UML

Astah UML es la herramienta de modelado UML usada durante el desarrollo de la aplicación.

Su instalación se hace directamente bajando el programa desde su página web [4]. Una vez descargada a través de un asistente se eligen las opciones de instalación. Para el proyecto se han elegido las opciones por defecto. Una vez instalado, he solicitado una licencia de uso de estudiante a través del propio programa y la he cargado en la aplicación.

[2] https://code.visualstudio.com/download
[3] https://visualstudio.microsoft.com/es/vs/github/
[4] https://astah.net/downloads/