**Repositorio**

<https://github.com/hiphoox/c321-turingmachine>

**Roles asignados**

* Gómez Bolaños Luis Aldo - **System Integrator**
* Bastida Vargas Karen Mariel - **Project Manager**
* Corona Nava Pedro Jair - **Tester**
* Ugalde Velasco Armando - **System Architect**

**Requerimientos**

* Mejora de mensajes de error en el compilador
* Implementación de tests.
* Implementación de banderas.

**Forma de trabajo**

En el repositorio, se contará con cinco ramas:

1. **main:** Aquí se colocará la versión final de la entrega actual. No se realizarán cambios aquí. El código debe ser completamente funcional.
2. **staging:** Aquí se combinará la mayor parte del código. Estará delante de la rama main y fungirá como preparación para la entrega a realizar.
3. **feature\_flags:** En esta rama se implementará lo relacionado a la funcionalidad de banderas.
4. **feature\_error\_messages:** En esta rama se implementará lo relacionado a los mensajes de error.
5. **feature\_testing:** En esta rama se implementarán los tests necesarios.

Cada miembro del equipo trabajará en una rama feature, en la que pueden trabajar uno o más integrantes. Una vez que la funcionalidad haya sido terminada, se hará un pull request a la rama staging, y todos los integrantes colaborarán para hacer una revisión del código. Una vez que esto suceda, se aprobará el pull request.

Cuando todas las funcionalidades hayan sido combinadas en la rama staging, y se haya asegurado que el código es completamente funcional, auxiliándose en las pruebas, se hará un pull request a la rama main y se combinará, señalando la finalización de esta etapa del proyecto.

Todos los miembros colaborarán para la implementación de las funcionalidades. Sin embargo, para agilizar el trabajo y permitir contribuciones en paralelo, se propone la siguiente división del trabajo para esta entrega:

* **Karen Bastida y Aldo Gómez** - Implementación de mejora de mensajes de error
* **Pedro Corona** - Implementación de tests
* **Armando Ugalde** - Implementación de banderas

Respecto al tiempo de trabajo, se propone un tiempo total de dos horas para la realización de todas las tareas asignadas a cada persona. Esto puede incluir tiempo de investigación, resolución de bugs, desarrollo, entre otras actividades.

La comunicación entre los miembros del equipo se llevará a cabo a través de WhatsApp. Si los miembros lo consideran necesario, será posible agendar reuniones breves para temas como discusión de código, pair programming, entre otras.

**Milestones planeados**

**15 de septiembre:** Para este punto, todos los miembros del equipo estaremos familiarizados con el lenguaje Elixir.

**18 de septiembre:** Implementación de banderas concluida.

**19 de septiembre:** Implementación de mensajes de error concluida.

**20 de septiembre:** Implementación de testing concluida.

**25 de septiembre:** Versión final concluida. Código completamente funcional y se aseguró que se cumplen los requerimientos.

**26 de septiembre:** Presentación con el profesor.

**Bitácora**

*23 de agosto - 15 de septiembre*

* Cada integrante estudió el lenguaje Elixir de forma independiente.

*15 de septiembre*

* Se agregaron los archivos iniciales al repositorio y se estudió el código.

*16 de septiembre*

* Implementación inicial del requerimiento de las banderas.
* Implementación de mensaje de error en lexer (token inválido)

*17 de septiembre*

* Mejora: Detener la generación de código cuando el lexer o parser encuentran un error.

*18 de septiembre*

* Implementación de mensajes de error en parser.

*19 de septiembre*

* Se arregló bug en el generador de código.
* Se agregó el script de testing propuesto por Nora Sandler

*25 de septiembre*

* Se arreglaron algunos bugs en el lexer y parser.
* Se refactorizaron algunas partes del código.
* Se implementaron tests automatizados.
* Se agregó integración continua para ejecutar los tests uniformemente.

**Resultados**

* Mejora de mensajes de error en el compilador
  + Cuando la compilación del código fallaba, el programa mostraba errores difíciles de leer. Lo anterior se debía a que el generador de código se ejecutaba sin importar que las etapas anteriores fallaran. Para resolverlo, se previno que éste se ejecutara de encontrarse algún error.
  + Sin embargo, si se encuentra un error durante el lexing, se permite que el parser se ejecute para encontrar más errores y desplegarlos.
  + Cuando hay un error en el lexer, se imprime el token inválido encontrado.
  + Cuando hay un error en el parser, se imprime el error gramatical encontrado.
* Implementación de tests.
  + Se implementaron todos los tests correspondientes a la etapa 1 del compilador. Para lograrlo, se utilizó la biblioteca ExUnit. Se definió una serie de tests cuya compilación debe ser válida y otros inválida.  
      
    Para los válidos, cada test unitario se encarga de compilar el código especificado y comprobar que el proceso haya tenido éxito. Una vez que se tiene el código generado, se intenta ligar y generar el ejecutable. De ser exitosa la acción anterior, finalmente se ejecuta el programa compilado y se comprueba que el valor de retorno sea el esperado.  
      
    En cambio, para los inválidos, en cada test unitario únicamente se intenta compilar el código especificado, y se comprueba que el proceso haya fallado.
  + También se implementaron tests para comprobar el correcto comportamiento de la funcionalidad de banderas: si alguna se especifica al ejecutar el programa, no se debe generar el ejecutable del archivo .c a compilar. En cambio, si se trata de la bandera A, se espera que el ejecutable sí se genere.
* Implementación de banderas.
  + Se implementaron 5 banderas:
    - **A:** Imprime la salida de todas las etapas de compilación, y además genera el ejecutable.
    - **sn:** Imprime la salida del sanitizer.
    - **l:** Imprime la salida del scanner.
    - **p:** Imprime la salida del parser.
    - **s:** Imprime la salida del generador de código.

**Diagrama de Gantt**



