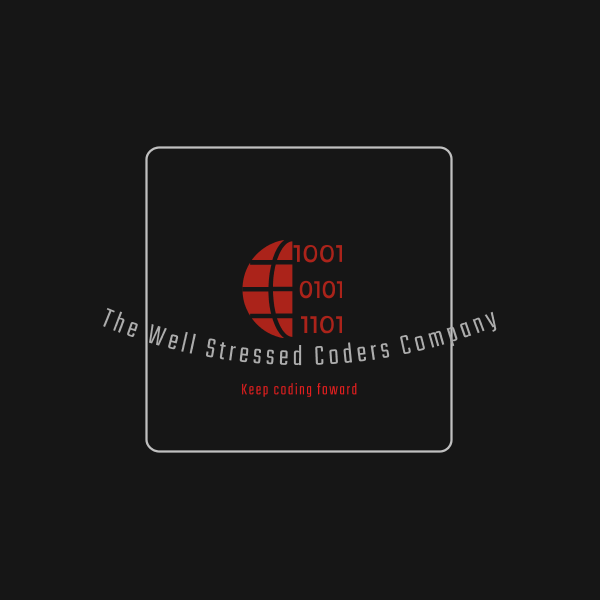


**Compilador de un subconjunto de funciones de C en Elixir**



“The Well Stressed Coders Company”

**Profesor :**

Norberto Jesús Ortigoza Márquez

**Miembros del equipo de desarrolladores:**

Barrón Pérez Marian Andrea

Monterrubio López Charlie Brian

Ortiz Martínez Brenda

Grupo: 04

Semestre 2019-2

Índice

|  |  |
| --- | --- |
| **1**. Introducción | 2 |
| **1.1** Alcance del proyecto. Road Map | 3 |
| **1.2** Plan de trabajo | 4 |
| **1.3** Información sobre el equipo de trabajo | 6 |
| **2**. Requerimientos | 7 |
| **2.1** Restricciones | 9 |
| **2.2** Características de los usuarios | 9 |
| **3.** Análisis del proyecto | 9 |
| **3.1** Análisis léxico (Lexer) | 10 |
| **3.2** Análisis sintáctico (parser) | 11 |
| **3.3** Generador de código | 12 |
| **4.** Diseño del software | 12 |
| **5**. Construcción | 13 |
| **6.** Pruebas | 17 |

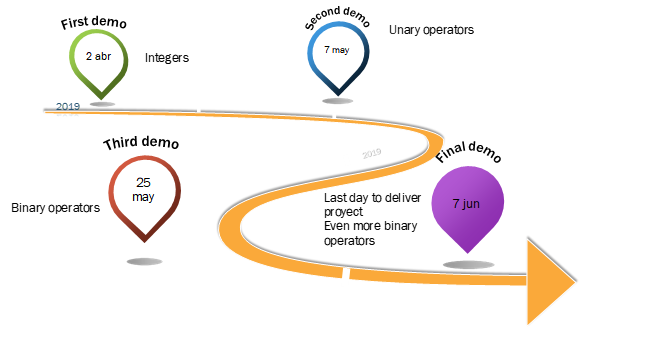
1. Introducción

El cliente Norberto Jesus Ortigoza Márquez requiere de un programa en línea de comandos que compile códigos de lenguaje C. Los códigos que necesita compilar los proporciona el mismo cliente con la finalidad de que todos ellos sean compilados de manera correcta generando los ejecutables correspondientes.

Norberto solicita además que el programa sea creado bajo ciertas especificaciones la cuales se detallan en los requerimientos. Pide además un manual de instalación y uso para que el programa pueda ser utilizado por cualquier usuario con conocimientos básicos de línea de comandos.

Por último, el cliente ofrece un repositorio personal en GitHub con el fin de dar seguimiento al proyecto y sus versiones generadas. Además, brinda un calendario de entregas de avances del proyecto. Dichas entregas en el calendario deben cumplirse totalmente en tiempo y forma. Al realizar las entregas, Norberto solicitará que se explique la arquitectura del programa y la funcionalidad del código fuente además mostrar una suite de pruebas anexas a los códigos proporcionados por el cliente.

* 1. Alcance del proyecto (Roadmap)

El proyecto se desarrollará en aproximadamente 4 meses. Los objetivos del proyecto obedecen al siguiente Roadmap:****

**Entregable 1**

Para el primer demo se tienen 6 semanas para el desarrollo del programa y compilará un número entero y soportará los elementos típicos que conforman a la función main y la primer entrega será para el día 2 de abril de 2019.

El código objetivo a compilar del entregable 1, proporcionado por el cliente, es el siguiente:

int main (){

return 2;

}

El ejecutable devolverá como resultado el número entero 2.

**Entregable 2**

A partir de la primera entrega se tendra 5 semanas disponibles para la segunda demo, el cual, realizará operaciones unarias y las compilará generando un ejecutable. Se entregará el día 7 de mayo de 2019

El código objetivo a compilar del entregable 2 es el siguiente:

*(pendiente)*

**Entregable 3**

Se tendrá 3 semanas disponibles para la tercera demo. Realizará operaciones binarias (suma, resta, multiplicación y división) y las compilará generando un ejecutable. Se entregará el día 28 de mayo de 2019.

**Entregable final del proyecto**

10 días disponibles

El proyecto ya finalizado realizará operaciones binarias y lógicas (AND &&, OR ||, ==, !=, <, >, =>, <=) y las compilará generando el ejecutable. Se entregará el día 7 de Junio.

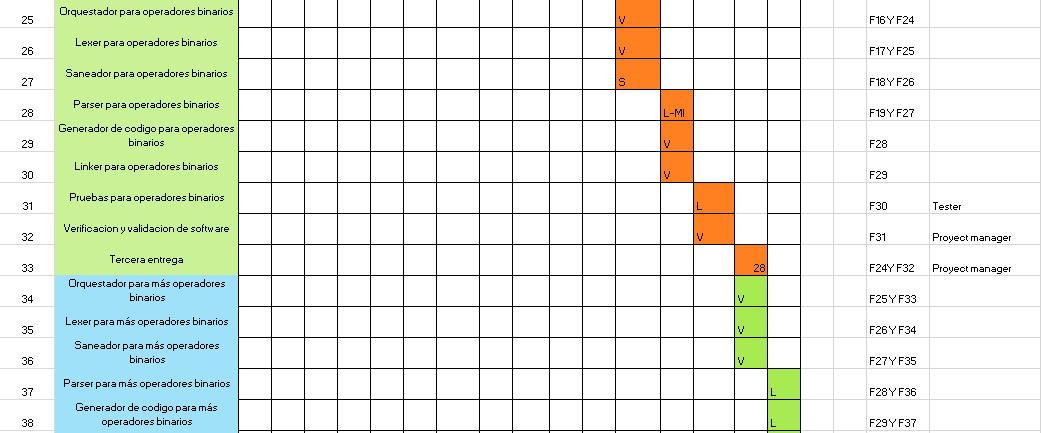
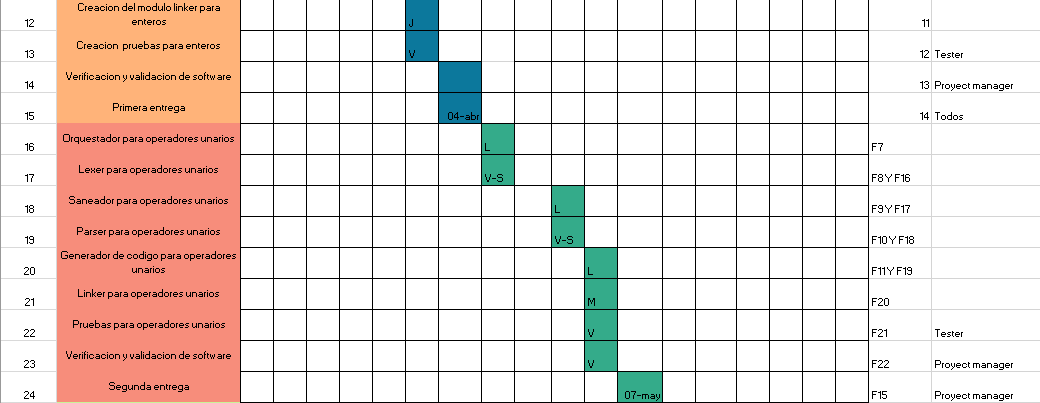
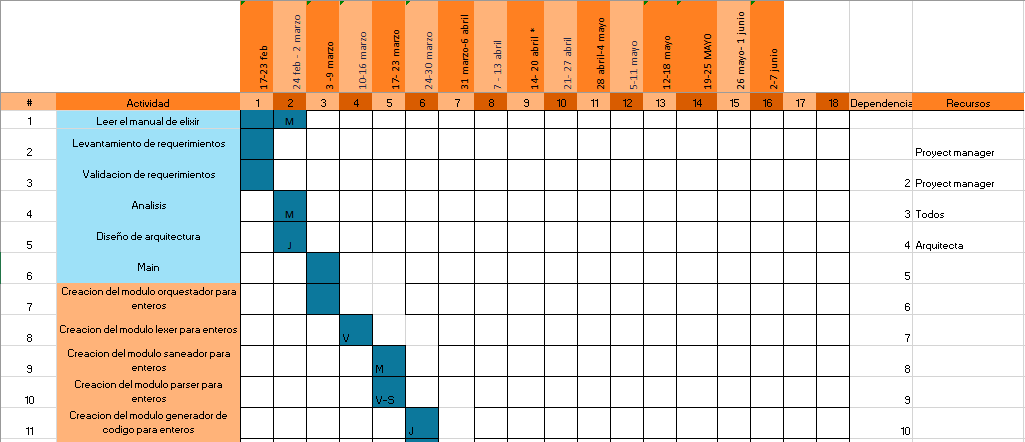
El código objetivo a compilar del proyecto final es el siguiente:

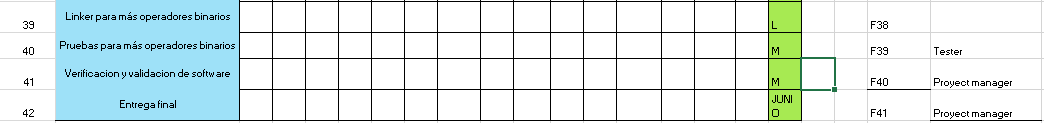
int main(){

return (3+4 <= 4 || 1&&2 != 3 > -6);

}

1.2 Plan de trabajo

****

****

1.3 Información del equipo de desarrollo

El equipo asignado para el desarrollo del proyecto se divide en los siguientes roles:

1. La administradora del proyecto es la encargada de organizar las reuniones del equipo de trabajo. Su responsabilidad es mantener un registro del proyecto y asegurar de que se realicen todos los entregables a tiempo además de generar un plan de trabajo.
2. El arquitecto del sistema será quien defina la arquitectura del compilador, los módulos e interfaces.
3. El integrador del sistema definirá la plataforma de desarrollo y las herramientas (coordinación del equipo, administrar el GIT (commits correctos) y que se use de la manera adecuada), el entorno de integración y el archivo compilado en su totalidad para asegurar que los componentes del compilador funcionen en conjunto
4. El tester define las pruebas, escribirá código, el cómo va a correr el proyecto. Cada miembro del equipo se espera que ejecute las pruebas unitarias y dicho tester asegure que el compilador cumpla con las especificaciones del lenguaje

|  |  |
| --- | --- |
| Nombre del integrante | Rol en el equipo |
| Barrón Pérez Marian Andrea | Administradora de software |
| Ortiz Martinez Brenda | Arquitecta de software |
| Monterrubio Lopez Charlie Brian | Integrador |
| Valdez Mondragon Erik | Tester |

2. Requerimientos

El cliente Norberto Jesus Ortigoza Márquez requiere de un compilador.El compilador está dirigido para cualquier persona que tenga conocimientos sobre línea de comandos.

Para la primera entrega el dia 2 abril del 2019, el compilador debe soportar enteros, así como también la siguiente línea de código:

int main(){(return “entero”)}

En dónde entero podría ser cualquier número entero.

Se requiere que el compilador reciba un código fuente en c que contenga solo una línea de código como anteriormente se mostro. Después de recibir el código deberá de leer el código y revisar que los elementos sean válidos de lo contrario que devuelva un error de sintaxis o lógico e indique la línea en donde encontró el error. Si los elementos son válidos, el usuario podrá elegir entre las diferentes banderas lo que desea visualizar, si el usuario no ingresa una bandera deberá generar el ejecutable.

El programa deberá de correr en la línea de comandos, de tal manera que podrá recibir 1 o 2 parámetros, uno para poder leer la bandera y otro para poder ingresar la dirección del archivo. El usuario podrá ingresar una bandera junto a la dirección del archivo indicando la opción que desee ver en la línea de comandos, también existirá una bandera que despliegue una ayuda de las banderas que son aceptadas.

El archivo ejecutable deberá de estar en la misma carpeta donde se encuentra el archivo en .c, el usuario podrá ingresar escoger el nombre del ejecutable, esto con la ayuda de una bandera, por lo que se tendrá las siguientes banderas:

|  |  |
| --- | --- |
| bandera corta | elemento a mostrar |
| -t | mostrará la lista de tokens |
| -a | mostrará el árbol |
| -s | generar el código ensamblador |
| -h | Despliega ayuda sobre los tipos de bandera |
| -o | Para poder ingresar un nombre al ejecutable |

Si el usuario ingresa alguna de las banderas, no generará el archivo ejecutable, si se desea generar el ejecutable, el usuario no deberá ingresar ninguna bandera y este no mostrará ningún proceso.

Se necesitará un manual de configuración, compilacion y ejecucion. Las pruebas automatizadas se deberán de poder correr de forma automática.

|  |  |  |  |
| --- | --- | --- | --- |
| Número | Requerimiento | Descripción | Prioridad |
| F1 | Recibir código fuente | Podrá recibir un código fuente en c con la siguiente linea int main(){return “entero”} en donde entero podra ser cualquier numero entero positivo | alta |
| F2 | Revisión de los elementos | Revisará que los elementos ingresados sean válidos | Alta |
| F3 | Línea de error de sintaxis | Devolverá un error de sintaxis o lógico e indicará la línea en donde encontró el error. | Baja |
| F4 | Parámetros | Deberá recibir 1 o 2 parámetros en la línea de comandos | Media |
| NF1 | Código fuente de otro lenguaje | El compilador no puede recibir código fuente de otro lenguaje | Alta |
| NF3 | Manual | Se necesitará un manual de configuración de compilación y ejecución | Baja |
| NF4 | Calidad | El programa deberá ser inteligente y analizar entradas de teclado; esto al ingresar las banderas. | Media |
| NF5 | Pruebas automatizadas | El compilador será capaz de compilar pruebas automatizadas | Alta |

2.1 Restricciones

* El compilador no puede recibir código fuente de otro lenguaje que no sea en C.
* El compilador solo esta implementado para que funcione con sistemas operativos Unix.
* Solo es capaz de soportar 1 bandera corta.
* El generador de código devolverá el código en ensamblador x86

2.2 Características de los usuarios

El compilador esta realizado para aquellas personas que manejen lo básico de línea de comandos en UNIX.

**3. Análisis**

Un compilador se compone internamente de varias etapas, o fases, que realizan operaciones lógicas.

* Análisis Léxico
* Análisis Sintáctico
* Generador de código

3.1 Análisis léxico (Lexer)

Una gramática es el conjunto de reglas que definen las palabras que reconoce, acepta y/o genera un lenguaje a partir de un determinado alfabeto.

Es la primera fase del compilador y es conocido con el nombre de Scanner. Transforma un conjunto de caracteres que de entrada que se leen del programa fuente uno a uno y salen componentes léxicos o también conocidos como tokens que después utilizará el analizador sintáctico. Y se encarga de reconocer identificadores palabras clave, constantes, operadores, etc.

Los componentes léxicos están formados por símbolos terminales de la gramática y pueden estar formados por:

❖ Palabras reservadas

❖ Identificadores

❖ Símbolos especiales

❖ Constantes de caracteres

El analizador léxico trabaja a petición del analizador sintáctico dándole un componente léxico cada que lo necesita el sintáctico. Los componentes léxicos se especifican haciendo uso de expresiones regulares.

En este caso se mostrarán los tokens que el lexer debe reconocer y la definición de la expresión regular de cada uno de ellos:

* Open brace {
* Close brace }
* Open parenthesis \(
* Close parenthesis \)
* Semicolon ;
* Int keyword int
* Return keyword return
* Identifier [a-zA-Z]\w\*
* Integer literal [0-9]+

3.2 Análisis sintáctico (parser)

A la segunda fase del compilador se le conoce como análisis sintáctico o *parsing.* El analizador sintáctico usa los *tokens* producidospor el analizador léxico para crear una estructura de árbol que represente la estructura gramatical de éstos mismos.

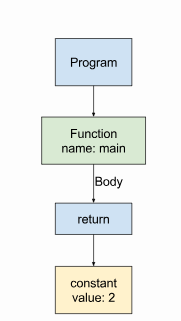
Este análisis es necesario para determinar si una serie de tokens suministrados por el análisis léxico son válidos en un lenguaje determinado, es decir, si la oración tiene la estructura o forma correcta. Pero no todas las oraciones sintácticamente correctas son válidas, para determinar esto con precisión se necesitará el análisis semántico.

Un analizador sintáctico toma la salida del analizador léxico en la forma de una cadena de tokens y lo analiza apoyándose en las reglas de producción para detectar errores en el código. La salida de este es un árbol de sintaxis.

De esta manera el análisis sintáctico logra dos tareas, analizar gramaticalmente el código, encontrar errores y generar un árbol de sintaxis como salida.

El analizador sintáctico debe de analizar todo el código aunque existan errores, para eso se ocupan estrategias de recuperación de errores.

A continuación se muestra un diagrama de Nora Sandler del AST para return\_2



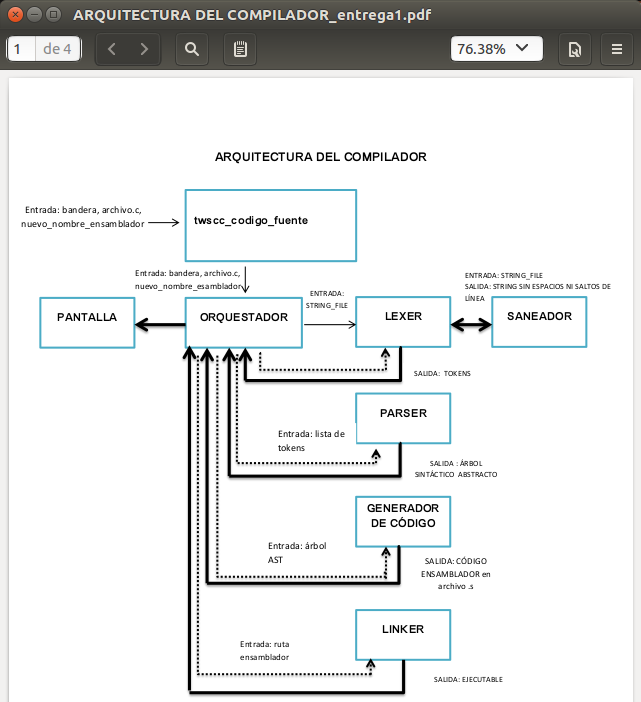
3.3 Generador de código

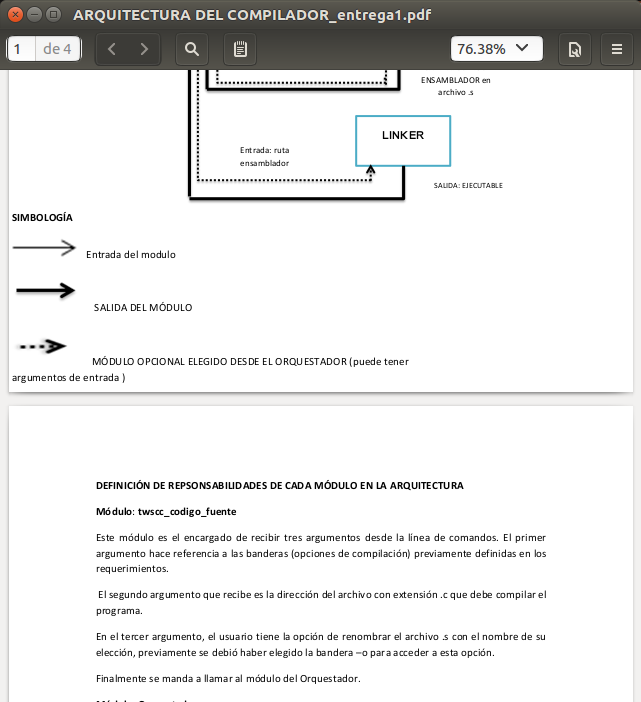
El generador de código es la fase mediante se convierte un programa sintácticamente correcto en una serie de instrucciones para ser interpretadas por una máquina. La entrada para esta fase es representada por un árbol sintáctico,

las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. De tal manera que genera código ensamblador.

4. Diseño de Software

Este es la arquitectura en la cual nos basamos para construir el compilador de manera adecuada.

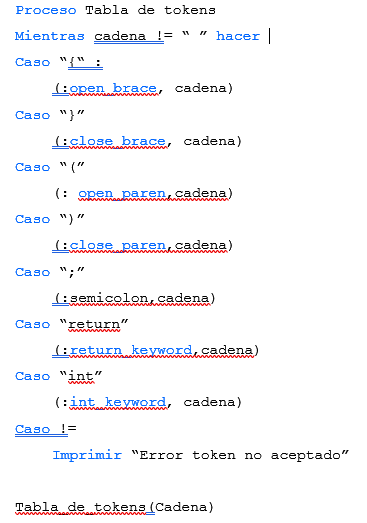
****

****

5. Construcción

**Lexer**

Para el lexer se tiene el siguiente pseudocódigo, el cual etiqueta a los tokens recibidos.

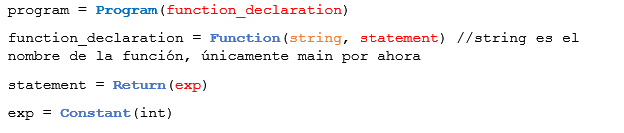
****

**Parser**

Después de etiquetar los tokens recibidos, se realiza el parser.

Para la creación del árbol se recomienda utilizar el algoritmo descendente recursivo.

Para el algoritmo descendente recursivo se define una función para analizar cada símbolo no terminal en la gramática y devuelve un nodo AST correspondiente. La función para analizar símbolo *S* debería eliminar fichas desde el principio de la lista hasta alcanzar una derivación válida de *S* . Si, antes de que termine el análisis, golpea un token que no está en la regla de producción para *S* , debería fallar. Si la regla para *S* contiene otros no terminales, debe llamar a otras funciones para analizarlos.



Mostrando el árbol de la siguiente manera:

{:node\_name, data, {:child 1, data}}

donde

node\_name: es el nombre asignado al nodo

data: Contiene el dato de dicho nodo, puede estar vacio

child 1: Contendrá a los posible hijos del nodo (NT) o bien,puede estar vacia indicando que el nodo es final(T).

**Generador de código**

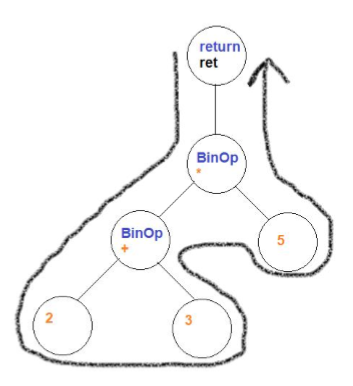
Como puede observarse, nuestro árbol tiene la estructura correcta para poder enviárselo al generador de código y pueda extraer los elementos terminales en el segundo elemento de la tupla (data) que es la información realmente útil para generar el ensamblador.

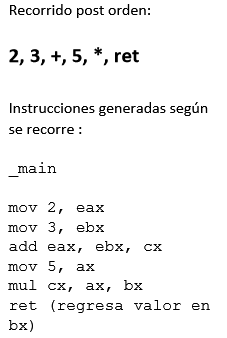
Nora Sandler indica que el árbol se visitará en el siguiente orden:

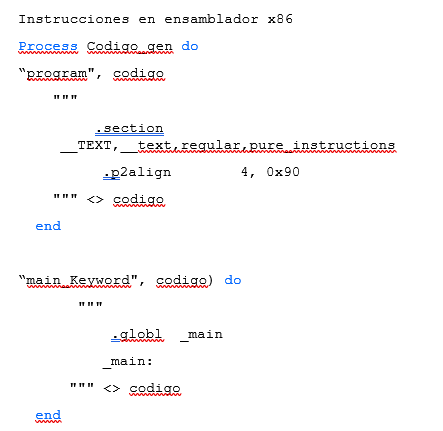
1. Nombre de la función (no es un nodo en realidad pero será lo primero a generar) 2. Valor a retornar

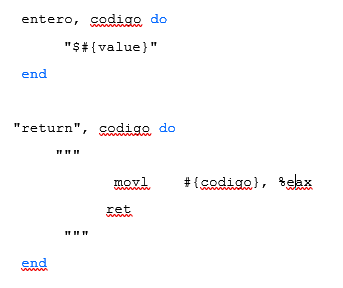
3. Instrucción de retorno ret

A continuación, el árbol puramente AST que recorrerá en post-orden el generador de código ya que nos garantiza que primero generamos el número entero y luego lo devolveremos.









6. Pruebas

Se configura un escenario de pruebas utilizando el comando mix test del proyecto de Elixir en la carpeta /test del proyecto.

El código fuente de las pruebas se encuentra en el archivo *test/proyecto\_compilador\_test.exs*. Dicho archivo contiene las siguientes pruebas a ejecutar con el fin de validar el correcto funcionamiento del programa.

**Pruebas para el módulo Lexer:**

Test 1:

Prueba elementos separados únicamente por espacios

int main( ) { return 2;}

Test 2:

Prueba elementos con saltos de línea y tabuladores

\n\tint main(\n ) { \nreturn \t2; }

Test 3:

Prueba lista con elementos totalmente separados

["int", "main", "(", ")", "{", "return", "2", ";", "}"]

Test 4:

Lista con algunos elementos separados

["int", "main()", "{", "return", "2", ";}"]

Test 5:

Prueba de código fuente sin espacios

Lexer.scan\_word(File.read!("test/spaces.c")

Test 6:

Prueba valor de retorno 2

File.read!("test/return\_2.c")

int main() {

return 2;}

Test 7:

Prueba valor de retorno 0

File.read!("test/return\_0.c")

int main() {

return 0;}

Test 8:

Sin saltos de línea

File.read!("test/no\_newlines.c")

int main(){return 0;}

Test 9:

Valor de retorno 100 (múltiples dígitos)

File.read!("test/multi\_digit.c")

int main() {

return 100;

}

Test 10:

Con saltos de línea

File.read!("test/new\_line.c")

int

main

(

)

{

return

0

;

}

Test 11:

Prueba de return en mayúsculas

File.read!("test/wrong\_case.c")

int main() {

RETURN 0;}

**Prueba en el módulo saneador:**

Test 12:

Prueba código fuente vacío

“”

**Pruebas para el módulo Parser:**

Test 13:

Prueba árbol ast de un código que devuelve 2

File.read!("test/return\_2.c")

int main() {

return 2;

}

Test 14:

árbol ast de un código que devuelve 100

File.read!("test/multi\_digit.c")

int main() {

return 100;

}

Test 15:

código al cual le falta un paréntesis que cierra

File.read!("test/missing\_paren.c")

int main( {

return 0;

}

Test 16:

Prueba código al cual le falta la expresión return

\n\tint main(\n ) { \n \t2; }

Test 17:

Prueba de código vacío proveniente desde el Lexer

Lexer.scan\_word("", :no\_output);