UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERÍA

Ciudad Universitaria

Proyecto: Compilador de un subconjunto de funciones de C en Elixir

Equipo de trabajo
“The Well Stressed Coders Company”

Miembros del equipo de desarrolladores:

Barrón Pérez Marian Andrea
Monterrubio Lopez Charlie Brian
Ortiz Martinez Brenda
Valdez Mondragón Erik

Profesor : Norberto Jesus Ortigoza Marquez
Grupo: 04

Semestre 2019-2

Índice

1. Introducción

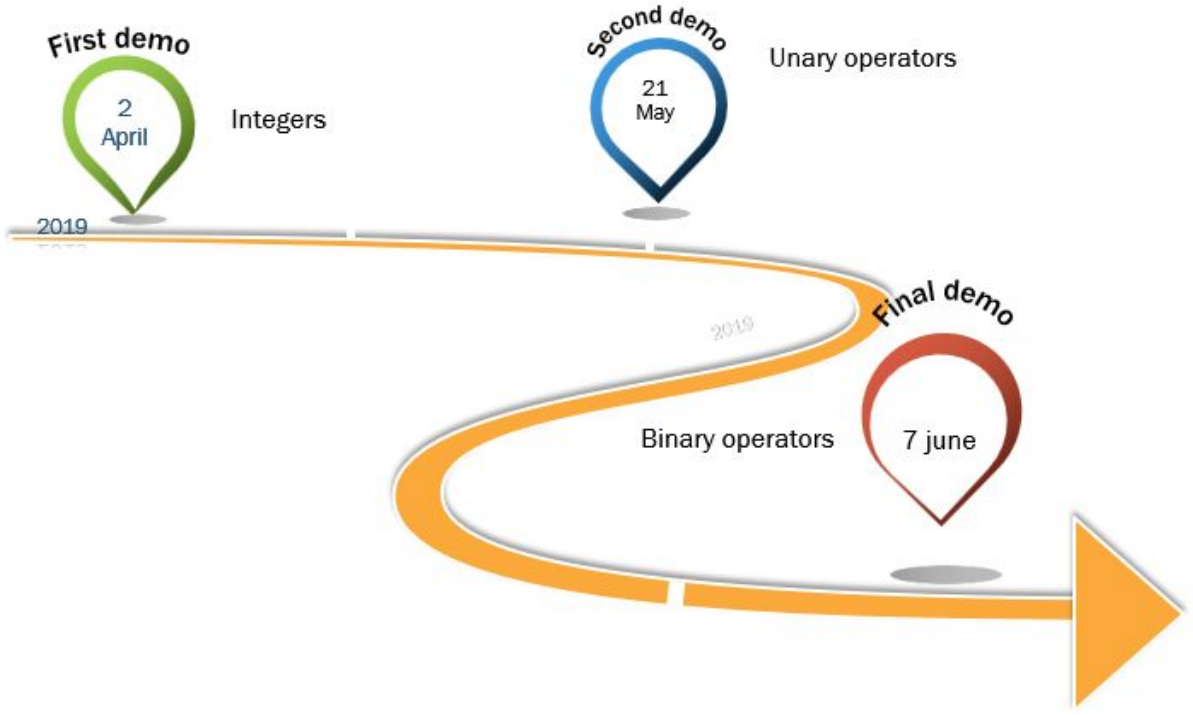
El cliente Norberto Jesus Ortigoza Márquez requiere de un programa en línea de comandos que compile códigos de lenguaje C. Los códigos que necesita compilar los proporciona el mismo cliente con la finalidad de que todos ellos sean compilados de manera correcta generando los ejecutables correspondientes.

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.

1.1 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 21 de mayo de 2019

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

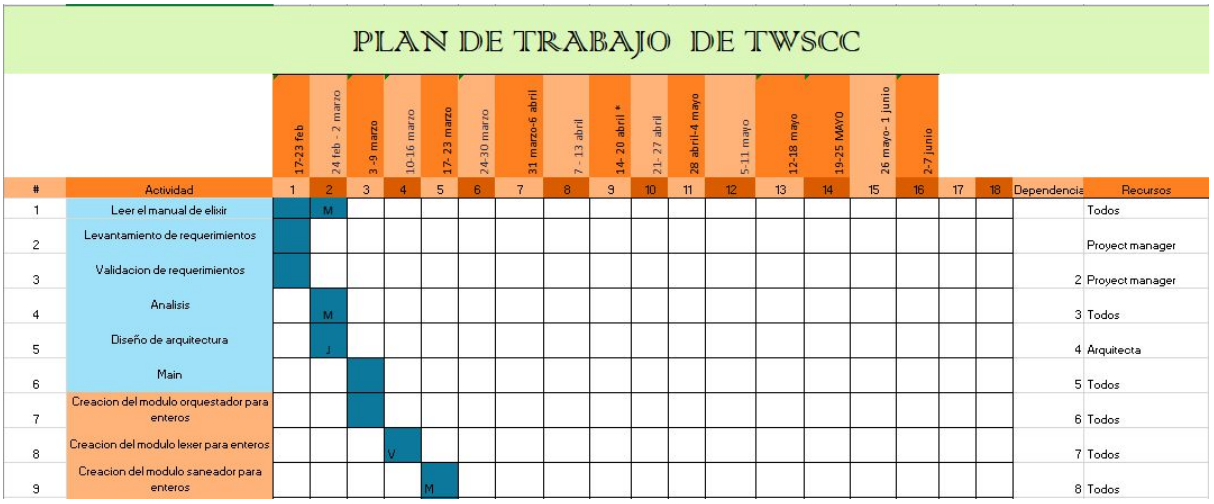
```
int main (){
     return ----3;
}
```

Entregable final del proyecto

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 7 de Junio de 2019.
El código objetivo a compilar del entregable 3 es el siguiente:

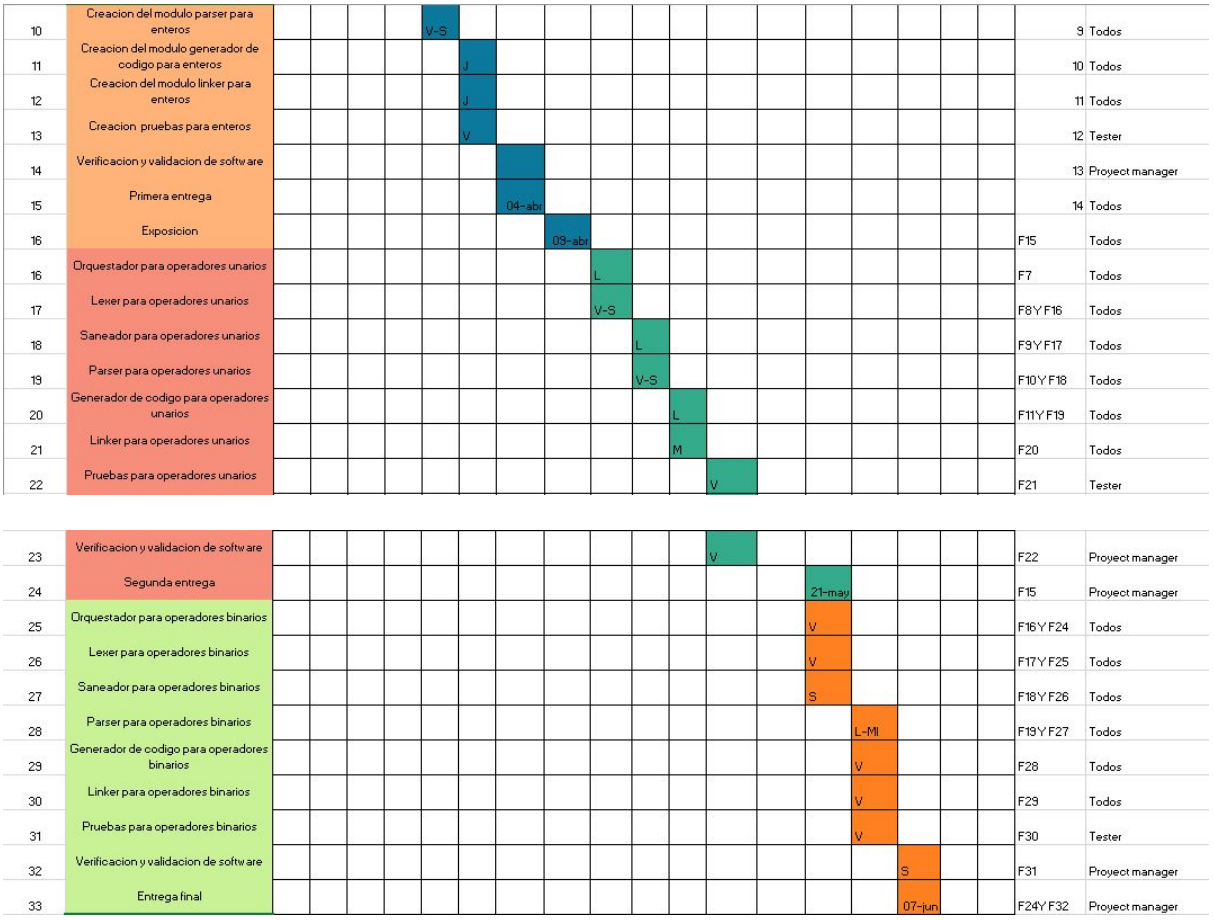
```
int main () {
return 2+3*4;
}
```

1.2 Plan de trabajo



PLAN DE TRABAJO DE TWSCC

#	Actividad	17-23 feb	24 feb - 2 marzo	3 -9 marzo	10-16 marzo	17- 23 marzo	24-30 marzo	31 marzo-6 abril	7 - 13 abril	14- 20 abril *	21 - 27 abril	28 abril-4 mayo	5-11 mayo	12-18 mayo	19-25 MAYO	26 mayo- 1 junio	2-7 junio	Dependencia	Recursos
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18		
1	Leer el manual de elixir		M																		Todos
2	Levantamiento de requerimientos																				Proyect manager
3	Validacion de requerimientos																			2	Proyect manager
4	Analisis		M																	3	Todos
5	Diseño de arquitectura		J																	4	Arquitecta
6	Main																			5	Todos
7	Creacion del modulo orquestador para enteros																			6	Todos
8	Creacion del modulo lexer para enteros				V															7	Todos
9	Creacion del modulo saneador para enteros					M														8	Todos



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 del Proyecto
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.

Se requiere que el compilador reciba un código fuente en c que contenga una línea de código como anteriormente se mostró.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 unitarias se deberán de poder correr de forma automática.

Número	Requerimiento	Descripción	Prioridad
F1	Recibir código fuente	Recibirá un código fuente en c con la siguiente linea `int main(){return numero}` en donde número podría ser cualquier número entero o unario	alta

F2	Revisión de los elementos	Revisará que los elementos ingresados sean válidos	Alta
F3	Linea 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 compilacion y ejecucion	Baja
NF4	Calidad	El programa deberá ser inteligente y analizar entradas de teclado; esto al ingresar las banderas.	Media

2.3 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
- ➢ Está realizado para personas que tengan conocimientos sobre línea de comandos.

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 Analisis lexico (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]+`

- Negation -
- Bitwise complement ~
- Logical negation !
- Addition +
- Multiplication *
- Division /

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* producidos por el analizador léxico para crear una estructura de árbol que represente la estructura gramatical de éstos mismos.

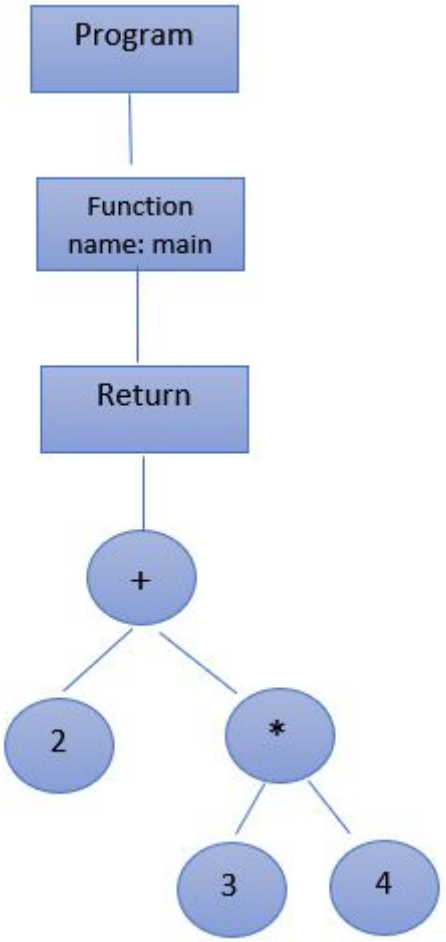
Este análisis es necesario para determinar si una serie de tokens suministrados por el análisis léxico son válidos en un lenguaje determinado, es decir, si la oración tiene la estructura o forma correcta. Pero no todas las oraciones sintácticamente correctas son válidas, para determinar esto con precisión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

```
int main (){ return(2+3*4)}
```

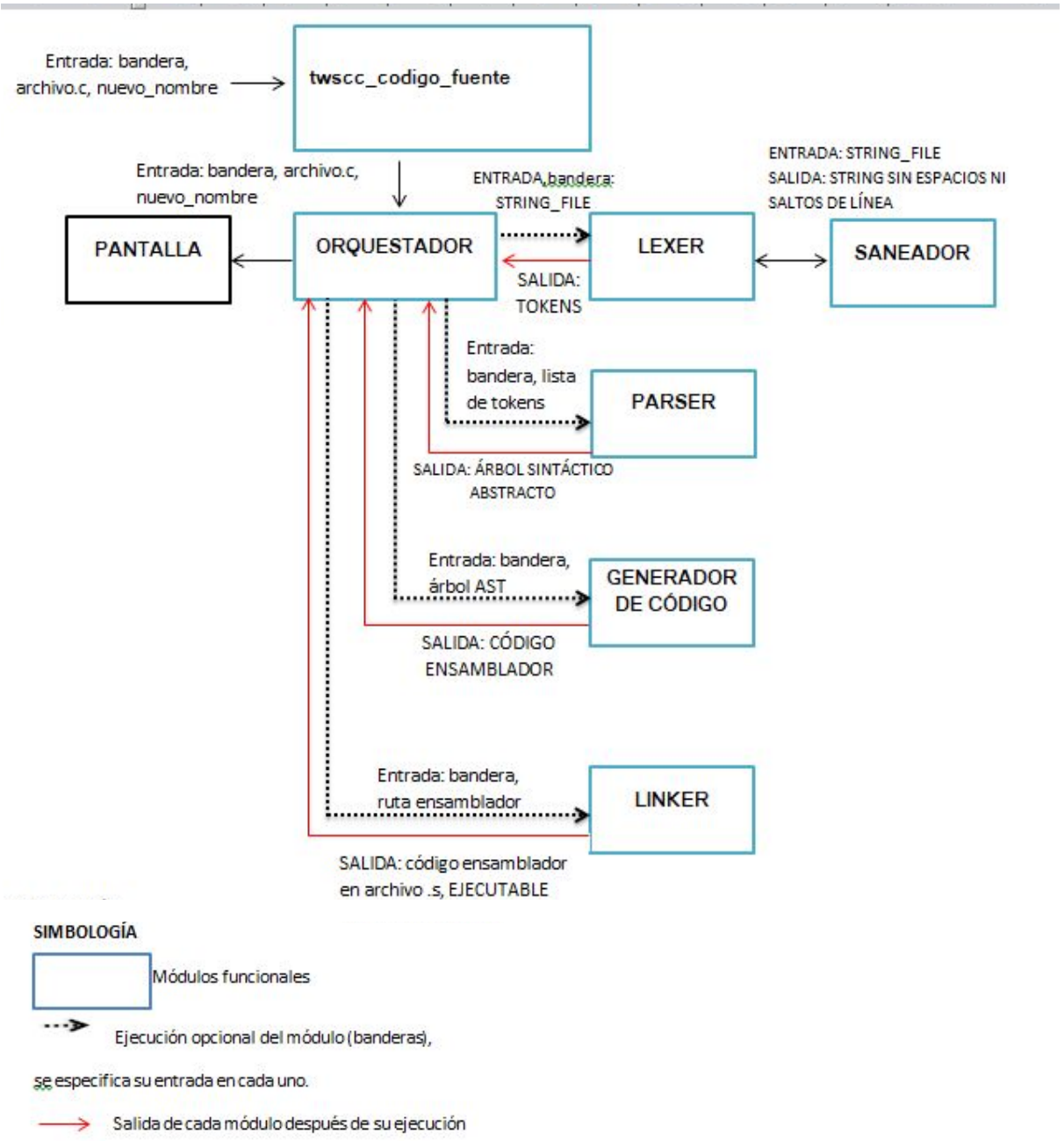


3.3 Generador de código

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 codigo 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.

```
Proceso Tabla de tokens
Mientras cadena != " " hacer
Caso "{" :
     (:open brace, cadena)
Caso "}"
     (:close brace, cadena)
Caso "("
     (: open paren,cadena)
Caso ")"
     (:close paren,cadena)
Caso ";"
     (:semicolon,cadena)
Caso "return"
     (:return_keyword,cadena)

Caso "int"
     (:int keyword, cadena)
Caso "-"
     (:negation keyword,cadena)
Caso "!"
     (:logicalNeg keyword,cadena)
Caso "~"
     (:bitewise keyword,cadena)

Caso "+"
     (:addition keyword,cadena)
Caso "*"
     (:multiplication keyword,cadena)
Caso "/"
     (:division keyword,cadena)
Caso !=
     Imprimir "Error token no aceptado"
```

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.

De manera que se tiene el siguiente pseudocódigo:

```
program ::= <function>
function::= "int" <id> "(" ")" "{" <statement> "}"
statement ::= "return" <exp> ";"
exp ::= <term> { ("+" | "-") <term> }
term ::= <factor> { ("*" | "/") <factor> }
factor ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

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).

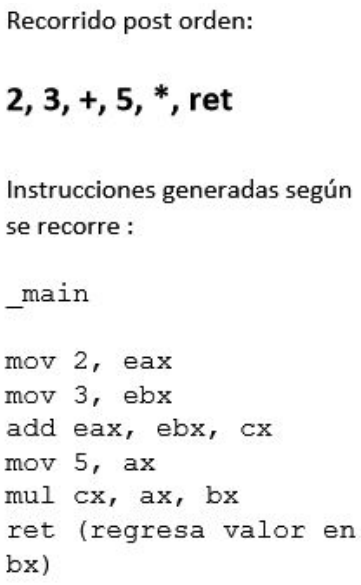
Generador de código

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.




```
def codigo_gen(:program, _, codigo, _) do
  """
  .p2align        4, 0x90
  """ <> codigo #concatena esto antes del codigo
end

def codigo_gen(:function, _, codigo, _) do
  """
      .globl  main         ## -- Begin function main
  main:                   ## @main
  """  <> codigo #concatena esto antes del codigo
end

def codigo_gen(:constant, value, codigo, post_stack) do
   #IO.puts("OP bin detected")
```

```
      IO.inspect(post_stack)
    if List.first(post_stack) == "+" or List.first(post_stack) == "*" or List.first(post_stack)
== "/" or List.first(post_stack) == "-" or List.first(post_stack) == "~"  or
List.first(post_stack) == "!" do
      #IO.puts("OP bin detected")
      codigo <> """
         mov     $#{value}, %rax
      """
    else
      #IO.inspect(value)
      codigo <> """
         mov     $#{value}, %rax
         push    %rax
      """
    end

  end

  ##pega el valor de la constante y añade una instruccion return
  def codigo_gen(:return_Keyword, _, codigo, _) do
    codigo <> """
       ret
    """
  end

  def codigo_gen(:negation_Keyword, _, codigo, _) do
    codigo <> """
       neg     %rax
    """
  end

  def codigo_gen(:bitewise_Keyword, _, codigo,post_stack) do
     if List.first(post_stack) == "return" do
       codigo <> """
```

```
        not     %rax
      """
    else
      codigo <> """
        not     %rax
        push    %rax
      """
    end

  end

  def codigo_gen(:logicalNeg_Keyword, _, codigo, _) do
    codigo <> """
      cmp     $0, %rax
      mov     $0, %rax
      sete    %al
    """
  end

  def codigo_gen(:addition_Keyword, _, codigo, _) do
    #IO.puts(codigo)
    #almacenar el primer operando usando un push a %eax
    codigo <> """
      pop     %rcx
      add     %rcx, %rax
    """
  end

  def codigo_gen(:multiplication_Keyword, _, codigo, _) do
    #IO.puts(codigo)
    #almacenar el primer operando usando un push a %eax
    codigo <> """
      pop     %rcx
      imul    %rcx, %rax
```

```
  """
end

def codigo_gen(:division_Keyword, _, codigo, _) do
  #IO.puts(codigo)
  #almacenar el primer operando usando un push a %eax
  codigo <> """
      pop     %rcx
      div     %rcx
  """
end

def codigo_gen(:minus_Keyword, _, codigo, _) do
  #IO.puts(codigo)
  #almacenar el primer operando usando un push a %eax
  codigo <> """
      pop     %rcx
      sub     %rax, %rcx
      mov     %rcx, %rax
  """
end
```

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 twssc_main:

1) Test 1:

   "Archivo inexistente en el directorio"

Proyecto_compilador.compile("/codigo_inexistente.c", :no_output) == "Archivo inválido o no existe en el directorio."

2) Argumentos inválidos del programa (--token -o codigo.c)

Proyecto_compilador.main(["--token", "-o", "codigo.c"]) == "Compilador de C de twscc. Escriba -h para la ayuda."

Pruebas para el módulo Lexer:

1) Test 1:

   Prueba del módulo Lexer al procesar el código fuente del entregable 1 sin saltos de línea.

```
int main( ) { return 2; }
```

2) Test 2:

   Prueba del módulo Lexer al procesar el código fuente del entregable 1 con saltos de línea.

```
int\n main(\n )\n { \nreturn \n2;\n }\n
```

3) Test 3:

   Prueba del módulo Lexer al procesar el código fuente del entregable 1 con saltos de línea y tabuladores.

```
int main(\n\t )\n\n { return \n2; }
```

4) Test 4: árbol ast de un código que devuelve 2

```
{:ok,
   ast: {:ok, {:program, "program",
            {:function, "main",
             {:return_Keyword, "return", {:constant, 2,
{}, {}}, {}}, {}}, {}}
           }}
```

5) Test 5: árbol ast de un código que devuelve 100

```
{:ok, {:program, "program",
             {:function, "main",
              {:return_Keyword, "return", {:constant,
100, {}, {}}, {}}, {}}, {}}}
```

6) Test 6: código al cual le falta un paréntesis que cierra

```
  {:ok, {:program, "program",
              {:function, "main",
               {:return_Keyword, "return",
                {:bitewise_Keyword, "~", {:constant, 0,
{}, {}}, {}}, {}}, {}}, {}}}
```

7) Test 7: Sin valor de retorno

```
{:ok, {:program, "program",
              {:function, "main",
               {:return_Keyword, "return",
                {:negation_Keyword, "-", {:constant, 5,
{}, {}}, {}}, {}}, {}}, {}}}
```

8) Test 8 Código al cual le falta la expresión return

```
{:error, "Error de sintáxis. Se esperaba return y se
encontró: (vacío)"}
```

9) Test 9:Unario, negacion lógica

```
  {:ok, {:program, "program",
              {:function, "main",
               {:return_Keyword, "return",
                {:logicalNeg_Keyword, "!", {:constant, 12,
{}, {}}, {}}, {}}, {}}, {}}}
```

10)Test 10: Unario, complemento del 0

```
 {:ok, {:program, "program",
              {:function, "main",
               {:return_Keyword, "return",
                {:bitewise_Keyword, "~", {:constant, 0,
{}, {}}, {}}, {}}, {}}, {}}}
```

11)Test 11: Unario, número negativo

```
{:ok, {:program, "program",
              {:function, "main",
               {:return_Keyword, "return",
```

```
                {:negation_Keyword, "-", {:constant, 5,
{}, {}}, {}}, {}}, {}}, {}}}
```

12)Test 12: Operadores unarios anidados

```
{:ok, {:program, "program",
             {:function, "main",
              {:return_Keyword, "return",
               {:logicalNeg_Keyword, "!",
                {:negation_Keyword, "-", {:constant, 3,
{}, {}}, {}}, {}}, {}}, {}}, {}}}
```

13)Test 13: Operadores unarios anidados 2

```
 {:ok, {:program, "program",
             {:function, "main",
              {:return_Keyword, "return",
               {:negation_Keyword, "-",
                {:bitewise_Keyword, "~", {:constant, 0,
{}, {}}, {}}, {}}, {}}, {}}, {}}}
```

14)Test 14: Cero negado

```
{:ok, {:program, "program",
             {:function, "main",
              {:return_Keyword, "return",
               {:logicalNeg_Keyword, "!", {:constant, 0,
{}, {}}, {}}, {}}, {}}, {}}}
```

15)Test 15: Sin punto y coma

```
{:error, "Error de sintáxis. Se esperaba ; y se encontró:
}"}
```

16)Test 16: Operadores anidados sin constante

```
{:error, "Error de sintaxis: Se esperaba una constante u
operador y se encontró ;."}
```

17)Test 17: Operadores en desorden

```
{:error, "Error de sintaxis: Falta el segundo operando
después de -."}
```

18)Test 18:

Prueba del módulo Orquestador, Lexer y Parser al procesar un código fuente con un número entero de retorno de tres dígitos.

```
int main() {
    return 100;
}
```

19) Test 19:

Prueba del módulo Orquestador, Lexer y Parser al procesar un código fuente con un saltos de línea en cada elemento que lo conforma.

```
int
main
(
)
{
return
0
;
}
```

20) Test 20:

Prueba del módulo Orquestador, Lexer y Parser al procesar un código fuente sin saltos de línea.

```
int main(){return 0;}
```

21) Test 21:

Prueba del módulo Orquestador, Lexer y Parser al procesar un código fuente que tiene un cero como valor de retorno.

```
int main() {
    return 0;
}
```

22) Test 22::

Prueba del módulo Orquestador, Lexer y Parser al procesar un código fuente que tiene un dos como valor de retorno.

```
int main() {
    return 2;
}
```

23)Test 23::

Prueba del módulo Orquestador, Lexer y Parser al procesar un código fuente con multiespacios entre los elementos.

```
   int   main    (  )  {   return  0 ; }
```

24)Test 24:

Prueba del módulo Orquestador, Lexer y Parser utilizando un código fuente de C el cual le falta un paréntesis.

```
int main( {
    return 0;
}
```

25)Test 25:

Prueba del módulo Saneador al procesar un código fuente que tiene un error al escribir el nombre de la función main.

```
 int min( ) { return 2; }
```

26)Test 26:

Prueba del módulo Linker al compilar un archivo con instrucciones de ensamblador generado previamente.

```
.p2align        4, 0x90
    .globl  main            ## -- Begin function main
main:                       ## @main
    movl    $0, %eax
    ret
```

27)Test 27:

Prueba del módulo Generador de código al generar el código ensamblador desde un árbol abstracto sintáctico definido para la prueba.

```
{:program, "program", {:function, "main",
{:return_Keyword, "return", {:constant, 5, {}, {}}, {}},
{}}, {}}
```

Para realizar la suite de pruebas se debe ejecutar el siguiente comando: `mix test`