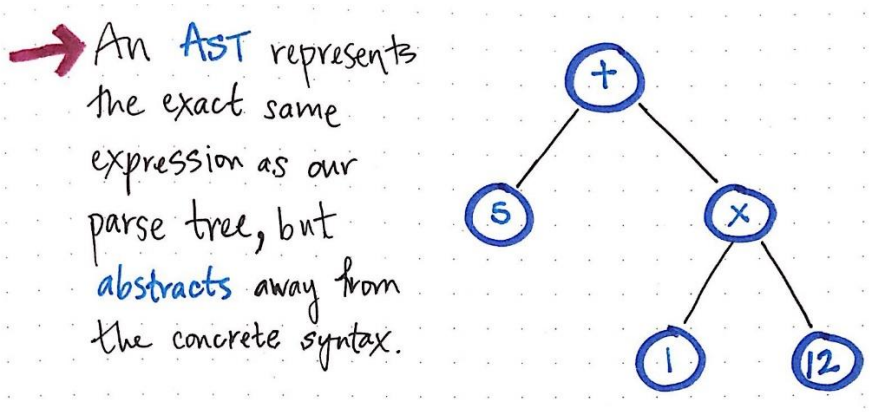
Árbol de Sintaxis Abstracta AST

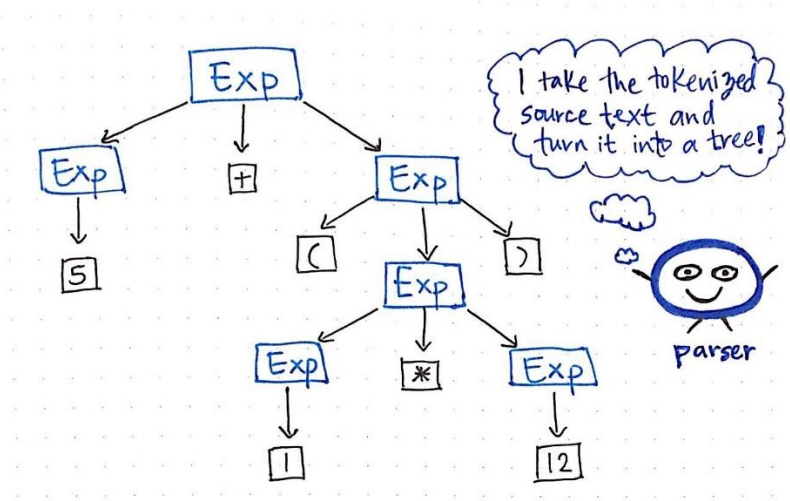
¿Qué es?

Es la representación de una estructura sintáctica **simplificada** del código fuente. Cada **nodo** del árbol denota una **construcción que ocurre en el código fuente**.

El AST representa la misma expresión que nuestro árbol de parser eliminando los elementos concretos de la sintaxis del código fuente.



Lo contrario a este es del Árbol de Sintaxis Correcta (ASC) generado a partir de los tokens una vez finalizado el análisis léxico.



¿Qué no se incluye de nuestra lista de tokens?

Código de prueba para el entregable 1:

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

Lista de tokens:
[~~:int_Keyword~~, **:main_Keyword**, ~~:open_paren~~, ~~:close_paren~~, ~~:open_brace~~, **:return_Keyword**, **{:constant, 2}**, ~~:semicolon~~, ~~:close_brace~~]

Siguiendo dicha definición, lo único que se representará en el árbol, y generará instrucciones en ensamblador x86 serán los *keyword* main, *return* y *constant*. También podría contender información adicional como la posición del elemento en el código fuente.

Estructura de datos utilizada para representar el AST

Se solicita mostrar en la consola el árbol mediante alguna estructura de datos fácil de entender. El equipo de desarrollo seguirá la estructura propuesta por *Nora Sandler* para representar el primer árbol.

Nodo del árbol, definición:

{:node_name, data, [child 1, child 2…]}

Tipo: *Átomo de Elixir.*	Tipo: *String, int,* operador (+, -, *, /) u operador lógico (&&, ||).	Tipo Únicamente nodos de la forma *{:node_name, data, [child 1, child 2…]}.*
Es el nombre asignado al nodo, por ejemplo: **:program, :function, :return, :bin_op, :constant**	Contendrá el dato de dicho nodo. Puede estar vacío.	Contendrá a los posibles hijos del nodo (NT) o bien, puede estar vacía indicando que el nodo es final (T)

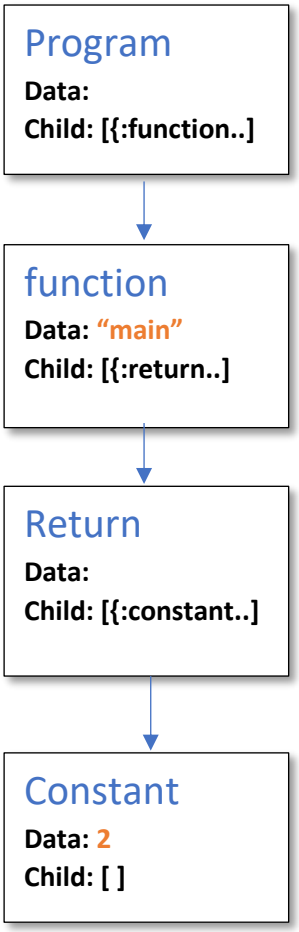
Se propone este elemento para la estructura completa del árbol a partir de la definición de los nodos del AST propuesta por Nora Sandler a continuación.

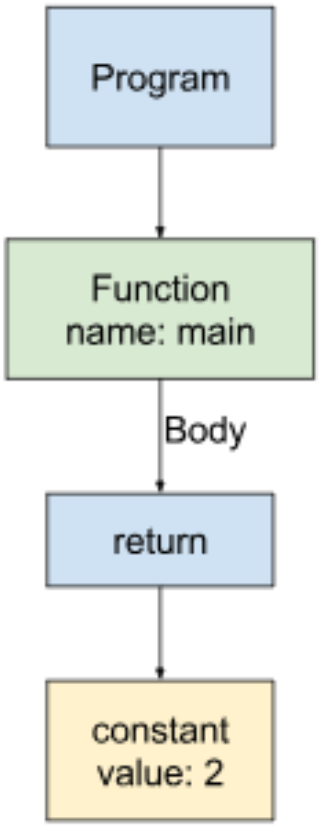
```
program = Program(function_declaration)

function_declaration = Function(string, statement) //string es el
nombre de la función, únicamente main por ahora

statement = Return(exp)

exp = Constant(int)
```

Trasladando lo anterior a nuestra estructura del árbol AST, existirán únicamente 4 nodos.

```
{:program, null, [{:function, "main", [{:return,
null, [{:constant, 2, []}]]}
```

A continuación, el diagrama del AST generado por nosotros y el de Nora Sandler para el código del entregable 1 con la estructura de datos anterior:



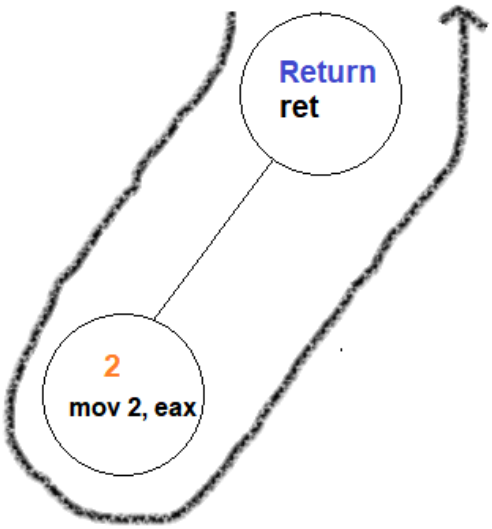


Como puede observarse, nuestro árbol tiene la estructura correcta para poder enviárselo al generador de código y pueda extraer los elementos terminales en el segundo elemento de la tupla (data) que es la información realmente útil para generar el ensamblador.

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 **2**
3. Instrucción de retorno **ret**

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



Código ensamblador final generado para este programa extrayendo en orden los elementos que generan código de la pila:

Recorrido postórden:

2, return

Código generado para x86:

```
_main:
mov 2, eax
ret
```

Se muestra otro ejemplo de AST con etiquetas y AST puro con operaciones binarias las cuales también requerimos de un recorrido post orden.

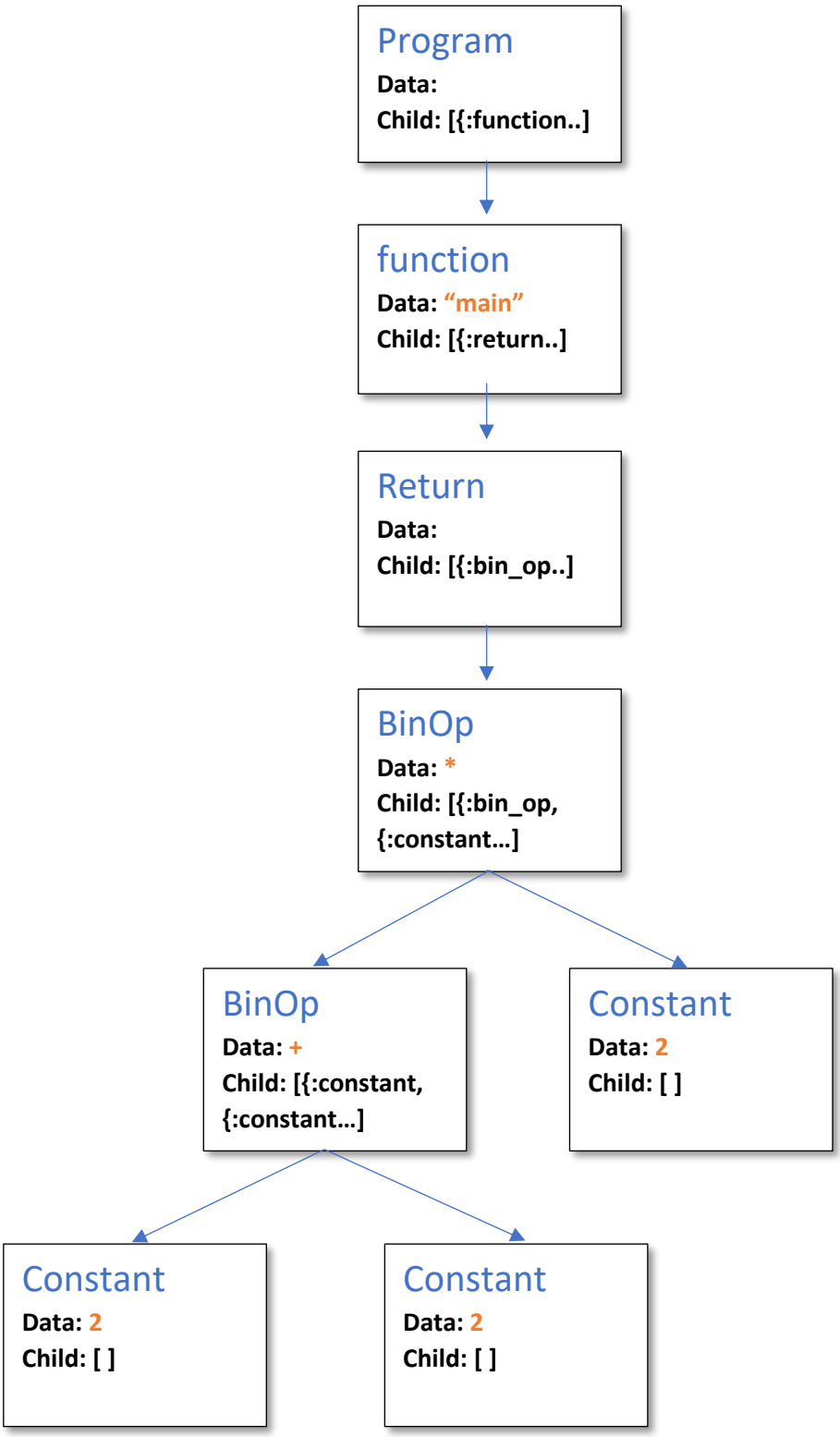
Código de prueba:

```
int main () {
     return 2 + 3 * 5;
}
```

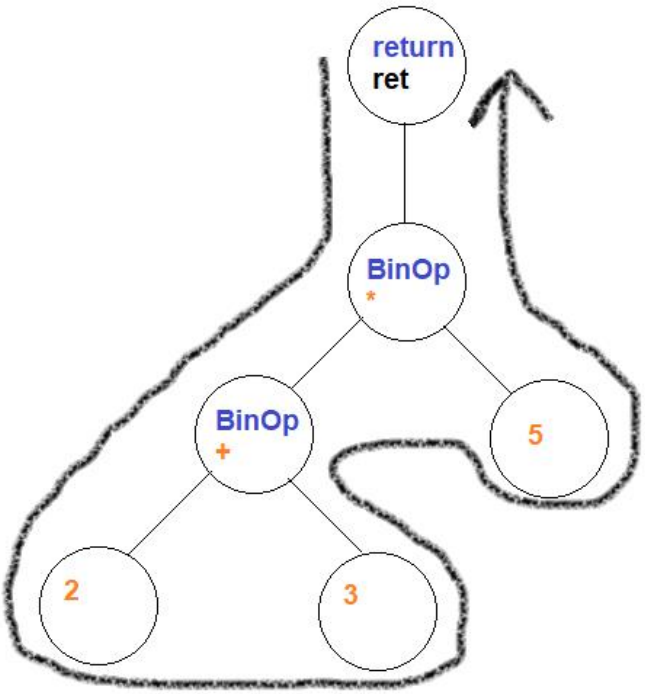
Árbol generado por el parser, estructura:

```
{:program, null, [{:function, "main", [{:return,
null, [{:bin_op, *, [{:bin_op, +, [{:constant, 2,
[]} ,{:constant, 3, []}]},{:constant, 5, []}]}]}]}]}
```

Diagrama del árbol AST



Árbol AST y el recorrido en post orden a realizar para generar el ensamblador.



Recorrido post orden:

2, 3, +, 5, *, ret

Instrucciones generadas según se recorre :

```
_main

mov 2, eax
mov 3, ebx
add eax, ebx, cx
mov 5, ax
mul cx, ax, bx
ret (regresa valor en
bx)
```

Como de demuestra, la estructura de datos para el árbol propuesta resultó ser la correcta tanto para devolver un simple entero como para realizar una o dos operaciones binarias.

El generador de código podrá extraer la información **realmente valiosa para generar código** a partir de la tupla de su elemento data y, en algunos casos, del nombre del nodo (como es el caso de la instrucción ret)

{:node_name, data, [child 1, child 2...]}