**INSTITUTO TECNOLÓGICO SUPERIOR DE ZAPOPAN**



**Alumnos**

Pérez García Cecilia

Carriola Monroy Ángel Fernando

**Materia**

Lenguajes y Autómatas II

**Maestro**

Armando Ramos

**Práctica**

Analizador Semántico

**Fecha de Entrega**

15 Abril de 2016

Índice

[Introducción 3](#_Toc449025702)

[Objetivos 4](#_Toc449025703)

[ Objetivo General 4](#_Toc449025704)

[ Objetivo Particular 4](#_Toc449025705)

[Desarrollo 5](#_Toc449025706)

[ Gramática 5](#_Toc449025707)

[ Pruebas del programa terminado 9](#_Toc449025708)

[Conclusiones 17](#_Toc449025709)

[Bibliografía 17](#_Toc449025710)

[Apéndice 18](#_Toc449025711)

[ Árbol de Derivación 18](#_Toc449025712)

[Acrónimos 21](#_Toc449025713)

# Introducción

Hay determinadas características de los lenguajes de programación que no pueden ser modeladas mediante gramáticas incontextuales y que es necesario comprobar en una fase posterior al análisis sintáctico. Por otro lado, las fases posteriores de la compilación o interpretación necesitan una representación de la entrada que les permita llevar a cabo sus funciones de manera adecuada. Estas dos vertientes —detección de errores y representación de la información— están muy relacionadas y se solapan en la práctica.

La semántica corresponde al significado asociado a las estructuras formales (sintaxis) del lenguaje.

Con el analizador semántico nosotros podremos identificar problemas con un código que será analizado mediante este analizador, tales como:

* Desperdicio de memoria
* Incompatibilidad de tipos

En esta práctica se tratará el desarrollo del analizador léxico, sintáctico, así como la gramática utilizada y poner a prueba el mismo para la verificación de su funcionalidad.

# Objetivos

# Objetivo General

Elaborar un analizador semántico que identifique problemas en cierto código mediante una lectura código escrito por el usuario el cual será analizado por el programa e indicará los problemas encontrados tales como desperdicio de memoria y/o incompatibilidad de tipos.

# Objetivo Particular

1. Crear la gramática correspondiente.
2. Crear el analizador sintáctico.
3. Crear el léxico.
4. El analizador semántico podrá indicarnos la falta de declaración de alguna variable.
5. El analizador semántico podrá indicarnos la falta de declaración de algún método.
6. El analizador semántico podrá indicarnos la incompatibilidad de tipos.
7. El analizador semántico podrá indicarnos cuando exista un desperdicio de memoria, es decir, cuando se ha declarado una variable o un método pero no se utiliza durante el programa.
8. Poner a prueba el analizador semántico.

# Desarrollo

# Gramática

A continuación se muestra la gramática utilizada para la creación del analizador sintáctico, léxico y semántico.

|  |  |  |
| --- | --- | --- |
| Token | Código/Atributo | Observaciones |
| <Tipo> | 0 | Tipos de datos:  int|float|double|string|char|bool |
| <Identificador> | 1 | Identificador de variable o función:  [a-zA-Z\_][a-zA-Z0-9\_]\* |
| <Constante Entera> | 2 | Número constante entero:  [1-9][0-9]\*|0 |
| <Constante Flotante> | 3 | Número constante con decimales:  ([1-9][0-9]\*|0)\.[0-9]+ |
| <Operador Aditivo> | 4 | Operadores de suma y resta:  +|- |
| <Operador Multiplicativo> | 5 | Operadores de multiplicación, división y módulo:  \*|/|% |
| <Operador Relacional> | 6 | Operadores relacionales:  ==|<=|>=|!=|<|> |
| <Operador Lógico> | 7 | Operadores lógicos AND y OR:  &&|\|\| |
| ¡ | 8 | Operador de negación. |
| = | 9 | Operador de asignación. |
| ; | 10 | Separador de sentencia. |
| { | 11 | Apertura de bloque de código. |
| } | 12 | Cierre de bloque de código. |
| ( | 13 | Paréntesis de apertura. |
| ) | 14 | Paréntesis de cierre. |
| , | 15 | Separador de variables y parámetros. |
| if | 16 | Condicional SI. |
| else | 17 | Condicional SI NO. |
| for | 18 | Bucle for. |
| while | 19 | Bucle while. |
| continue | 20 | continue de un bucle. |
| break | 21 | break de un bucle o un switch. |
| switch | 22 | Condicional de casos switch. |
| case | 23 | Prefijo para un caso switch. |
| : | 24 | Sufijo para un caso switch. |
| return | 25 | Retorno de una función. |
| main | 26 | Identificador de la función de arranque inicial. |
| default | 27 | Default de un switch |
| <Cadena> | 28 | Cadena de caracteres entre comillas dobles. |
| <Carácter Constante> | 29 | Carácter entre comillas simples. |
| <Operador Incremento> | 30 | Operadores de incremento ++ y – |
| <Booleano> | 31 | Palabras reservadas true y false. |
| <SASM> | 32 | Funciones para usarse en SASM:  print\_int y newline |

**Reglas base**

1. <Programa> 🡪 <Contenido Global> <Más Contenido Global>
2. <Contenido Global> 🡪 <Declaración> ";" | <Función>
3. <Más Contenido Global> 🡪 <Contenido Global> <Más Contenido Global> | vacío
4. <Bloque> 🡪 "{" <Contenido Bloque> "}"
5. <Contenido Bloque> 🡪 <Sentencia> <Contenido Bloque> | <If> <Contenido Bloque> | <For> <Contenido Bloque> | <While> <Contenido Bloque> | <Switch> <Contenido Bloque> | <Retorno Función> <Contenido Bloque> | vacío
6. <Cuerpo Estructura> 🡪 <Bloque> | <Sentencia> | ";"
7. <Sentencia> 🡪 <Declaración> ";" | <Asignación> ";" | <Llamada Función> ";"
8. <Declaración> 🡪 <Tipo> <Identificador> <Asignación Declaración> <Declaración Múltiple>
9. <Declaración Múltiple> 🡪 "," <Identificador> <Asignación Declaración> <Declaración Múltiple> | vacío
10. <Asignación Declaración> 🡪 "=" <Expresión> | vacío
11. <Asignación> 🡪 <Identificador> "=" <Expresión>

**Reglas para condicional if**

1. <If> 🡪 "if (" <Expresión> ")" <Cuerpo Estructura> <Else>
2. <Else> 🡪 "else if (" <Expresión> ")" <CuerpoEstructura> <Else> | "else" <Cuerpo Estructura> | <vacío>

**Reglas para bucle for**

1. <For> 🡪 "For (" <Control Variable For> ";" <Control Condicional For> ";" <Incremento For> ")<Cue Erpostructura>
2. <Control Variable For> 🡪 <Declaración> | <Asignación> | vacío
3. <Control Condicional For> 🡪 <Expresión> | vacío
4. <Incremento For> 🡪 <Expresión> | <Asignación> | vacío

**Reglas para bucle while**

1. <While> 🡪 "while (" <Expresión> ")" <Cuerpo Estructura>

**Reglas para condicional switch**

1. <Switch> 🡪 "switch (" <Expresión> ")" <Cuerpo Switch>
2. <Cuerpo Switch> 🡪 "{" <Caso Switch> <Más Casos Switch> <Caso Default Switch> "}"
3. <Más Casos Switch> 🡪 <Caso Switch> <Más Casos Switch> | vacío
4. <Caso Switch> 🡪 "case" <Constante Entera> ":" <Contenido Bloque> <Break Switch>
5. <Break Switch> 🡪 "break;" | vacío
6. <Caso Default Switch> 🡪 "default :" <Contenido Bloque> <Break Switch>

**Reglas para funciones**

1. <Función> 🡪 <Tipo> <Identificador> "(" <Parámetro Función> ")" <Bloque>
2. <Parámetro Función> 🡪 <Tipo> <Identificador> <Más Parámetros Función> | vacío
3. <Más Parámetros Función> 🡪 "," <Tipo> <Identificador> <Más Parámetros Función> | vacío
4. <Retorno Función> 🡪 "return" <Expresión> ";" | vacío
5. <Llamada Función> 🡪 <Identificador> "(" <Argumentos Función > ")"
6. <Argumentos Función> 🡪 <Expresión> <Más Argumentos> | vacío
7. <Más Argumentos> 🡪 "," <Expresión> <Más Argumentos> | vacío

**Reglas para expresiones**

1. <Expresión> 🡪 <Expresión Relacional> <Más Expresiones Relacionales>
2. <Expresión Relacional> 🡪 <Expresión Algebraica> <Más Expresiones Algebraicas>
3. <Más Expresiones Relacionales> 🡪 <Operador Lógico> <Expresión Relacional> <Más Expresiones Relacionales> | vacío
4. <Expresión Algebraica> 🡪 <Término> <Más Términos>
5. <Más Expresiones Algebraicas> 🡪 <Operador Relacional> <Expresión Algebraica> <Más Expresiones Algebraicas> | vacío
6. <Término> 🡪 <Factor> <Más Factores>
7. <Más Términos> 🡪 <Operador Aditivo> <Término> <Más Términos> | vacío
8. <Factor> 🡪 <Identificador> | <Constante Entera> | <Constante Flotante> | <Cadena> | <Carácter> | "!" <Factor> | <Llamada Función> | <Incremento> | "(" <Expresión> ")" | true | false
9. <Más Factores> 🡪 <Operador Multiplicativo> <Factor> <Más Factores> | Vacío
10. <Incremento> 🡪 <Identificador> <Operador Incremento> | <Operador Incremento> <Identificador>

**Reglas base para tokens**

1. <Número> 🡪 "0" | … | "9"
2. <Tipo> 🡪 "int" | "double" | "float" | "char" | "string" | "bool" | "void"
3. <Letra> 🡪 "a" | … | "z"
4. <Cadena> 🡪 "\"" <Caracteres> "\""
5. <Caracteres> 🡪 <Carácter> <Caracteres> | vacío
6. <Carácter Constante> 🡪 "'" <Caracteres> "'" | vacío
7. <Carácter> 🡪 <\_\_ASCII\_\_>

**Reglas para token identificador**

1. <Identificador> 🡪 <Prefijo Identificador> <Sufijo Identificador>
2. <Prefijo Identificador> 🡪 <Letra> | \_
3. <Sufijo Identificador> 🡪 <Símbolos Sufijo Identificador> <Sufijo Identificador> | vacío
4. <Símbolos Sufijo Identificador> 🡪 <Letra> | <Número> | \_

**Reglas para token constante entera**

1. <Constante Entera> 🡪 <Prefijo Constante Entera> <Sufijo Constante Entera> | 0
2. <Prefijo Constante Entera> 🡪 "1" | … | "9"
3. <Sufijo Constante Entera> 🡪 <Número> <Sufijo Constante Entera> | vacío

**Reglas para token constante flotante**

1. <Constante Flotante> 🡪 <Constante Entera> "." <Decimales>
2. <Decimales> 🡪 <Número> <Decimales> | vacío

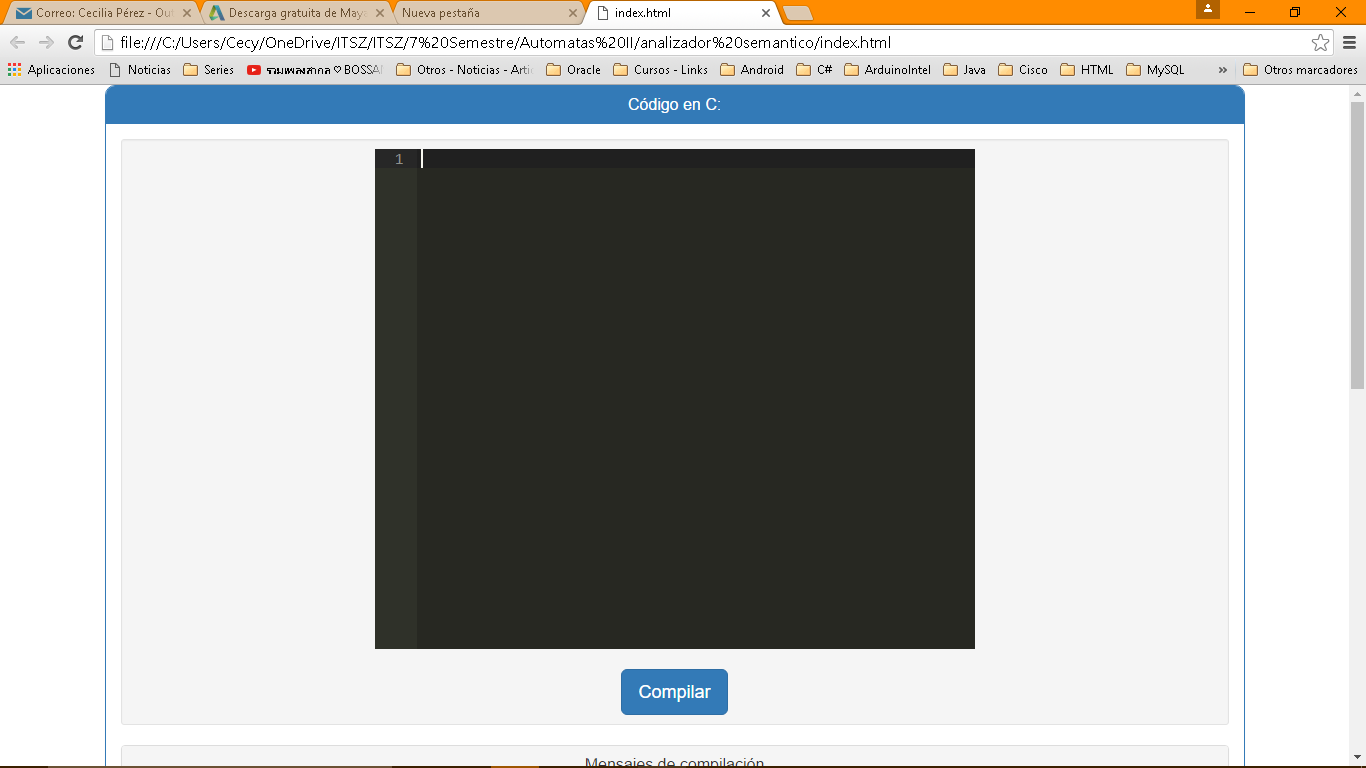
**Reglas para tokens de operadores**

1. <Operador Aditivo> 🡪 "+" | "-"
2. <Operador Multiplicativo> 🡪 "\*" | "/" | "%"
3. <Operador Relacional> 🡪 "==" | "<=" | ">=" | "!=" | "<" | ">"
4. <Operador Lógico> 🡪 "&&" | "||"

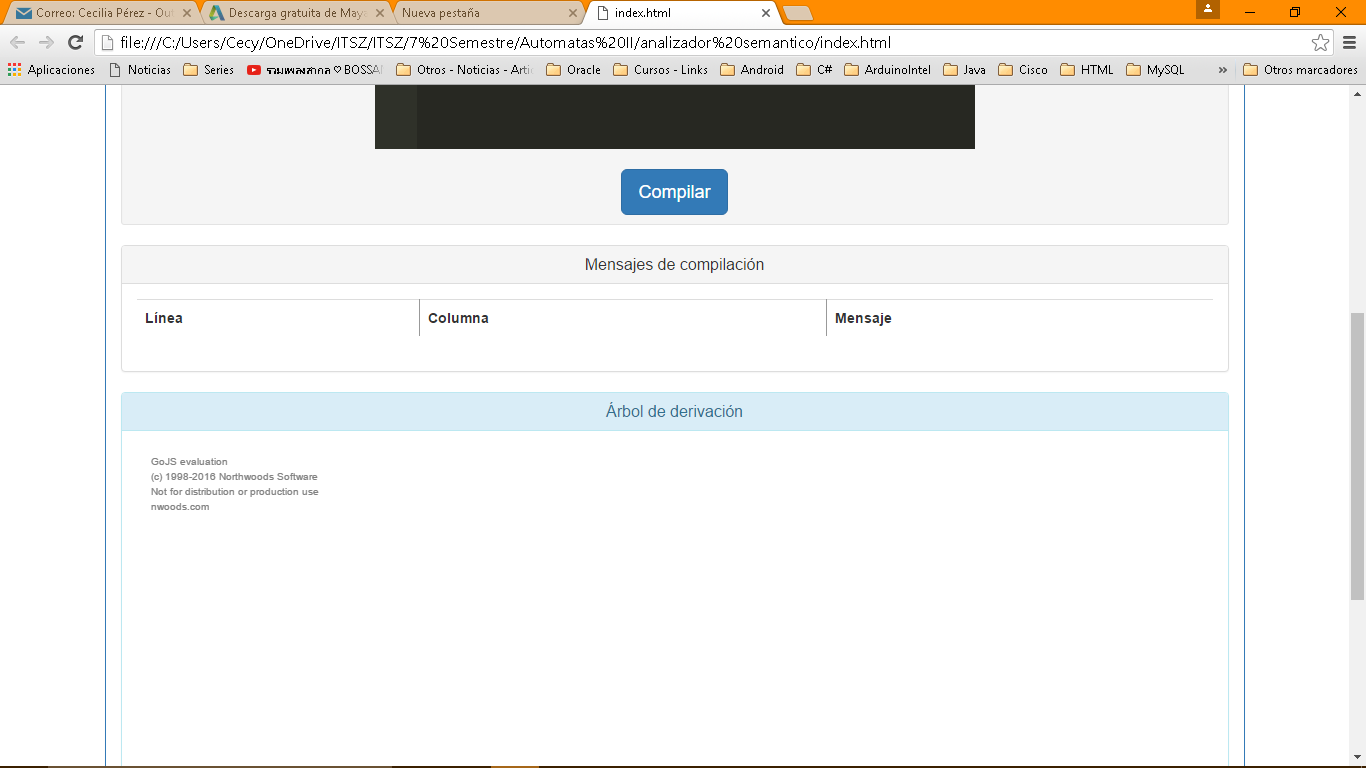
# Pruebas del programa terminado

El programa se divide en varias secciones:

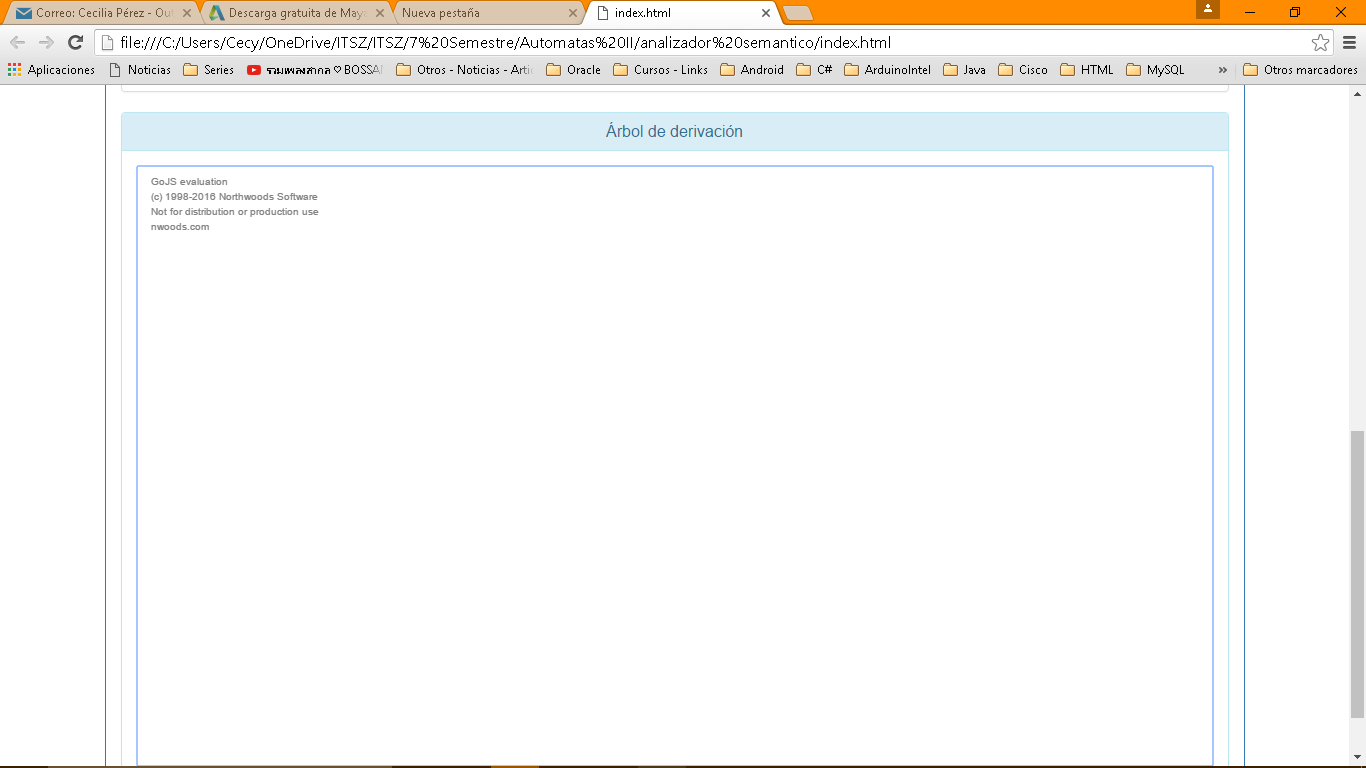
-Pantalla donde irá el usuario escribiendo el código.



-Pantalla donde irá visualizando los errores y warnings.



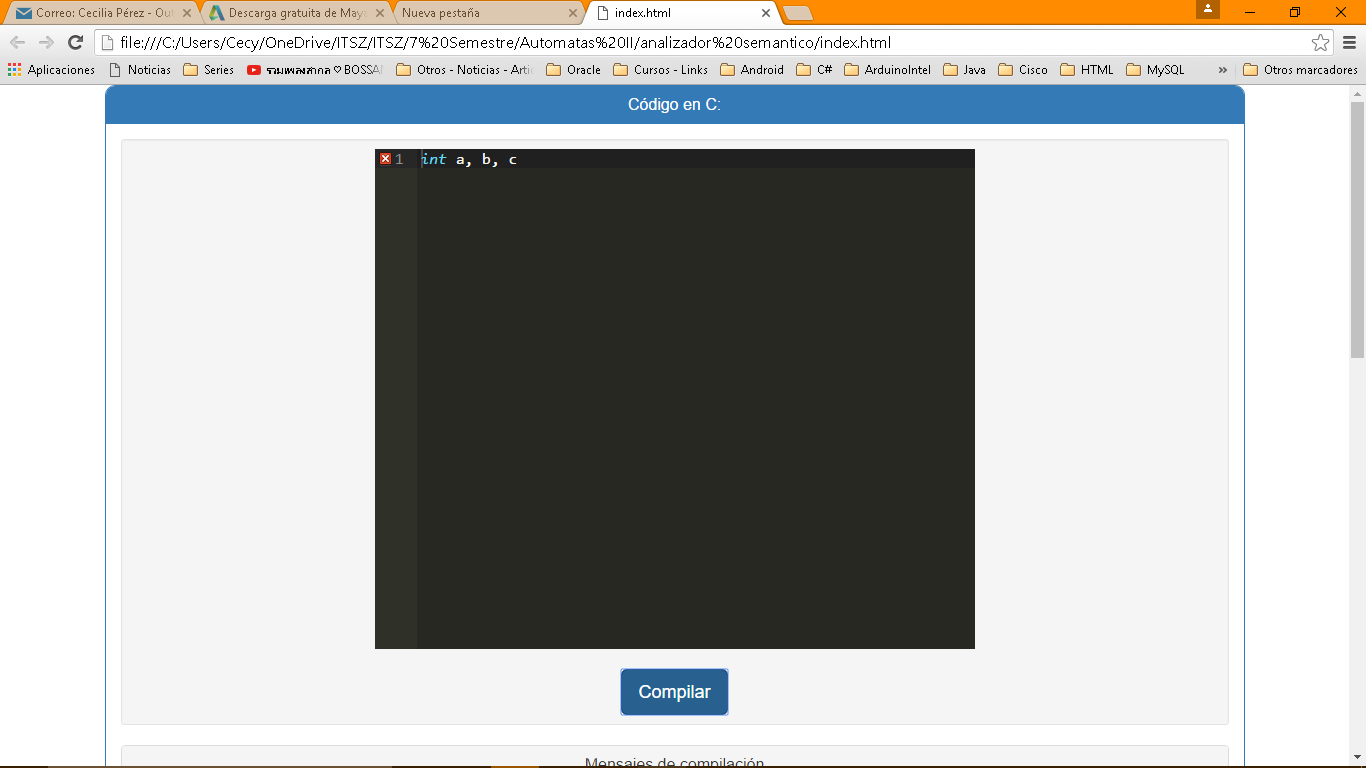
-Pantalla donde se irá visualizando los árboles de derivación.



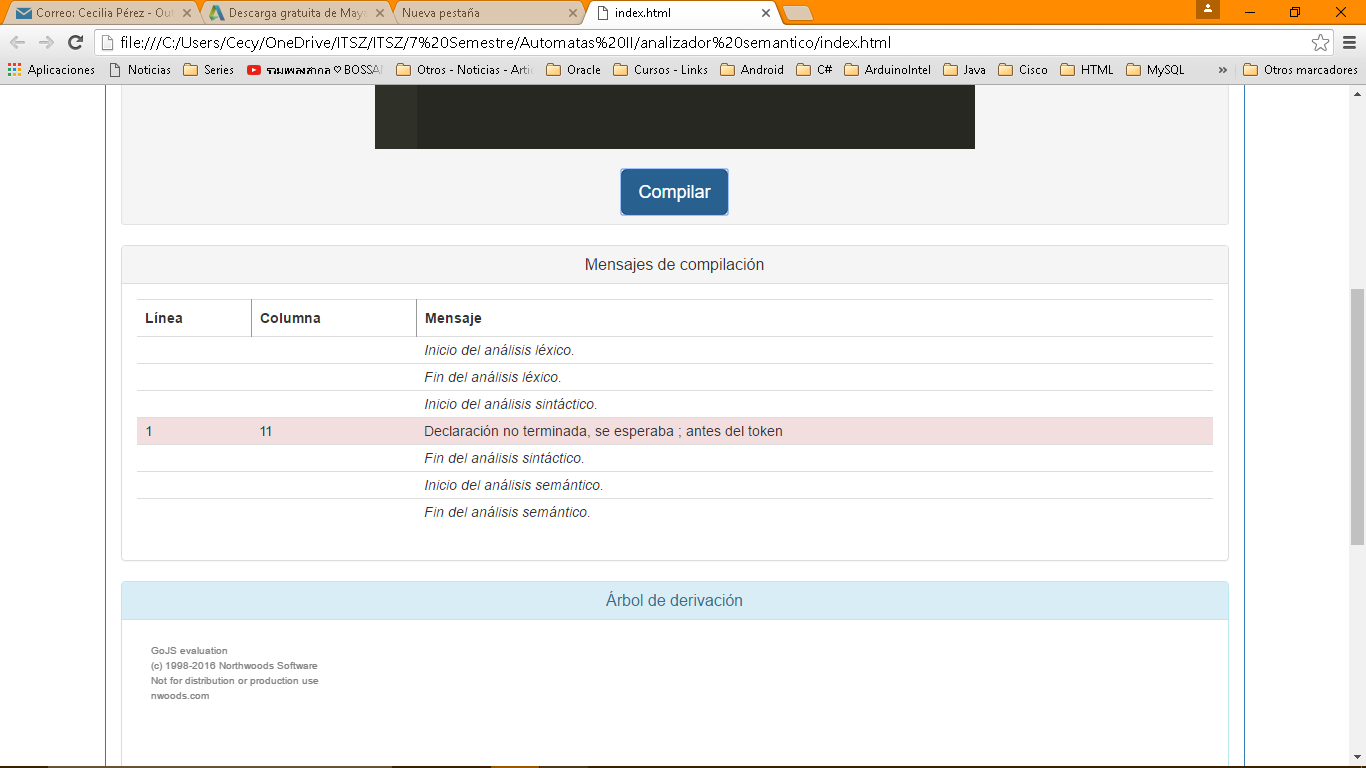
En la pantalla donde se verán los errores de sintaxis y warnings se divide en 3 partes, la parte de analizador sintáctico, léxico y semántico, además de que nos informará de qué tipo de error es.

A continuación se mostrarán varias pruebas donde se comprueba el funcionamiento del programa.

* **Ejemplo de error de sintaxis.**

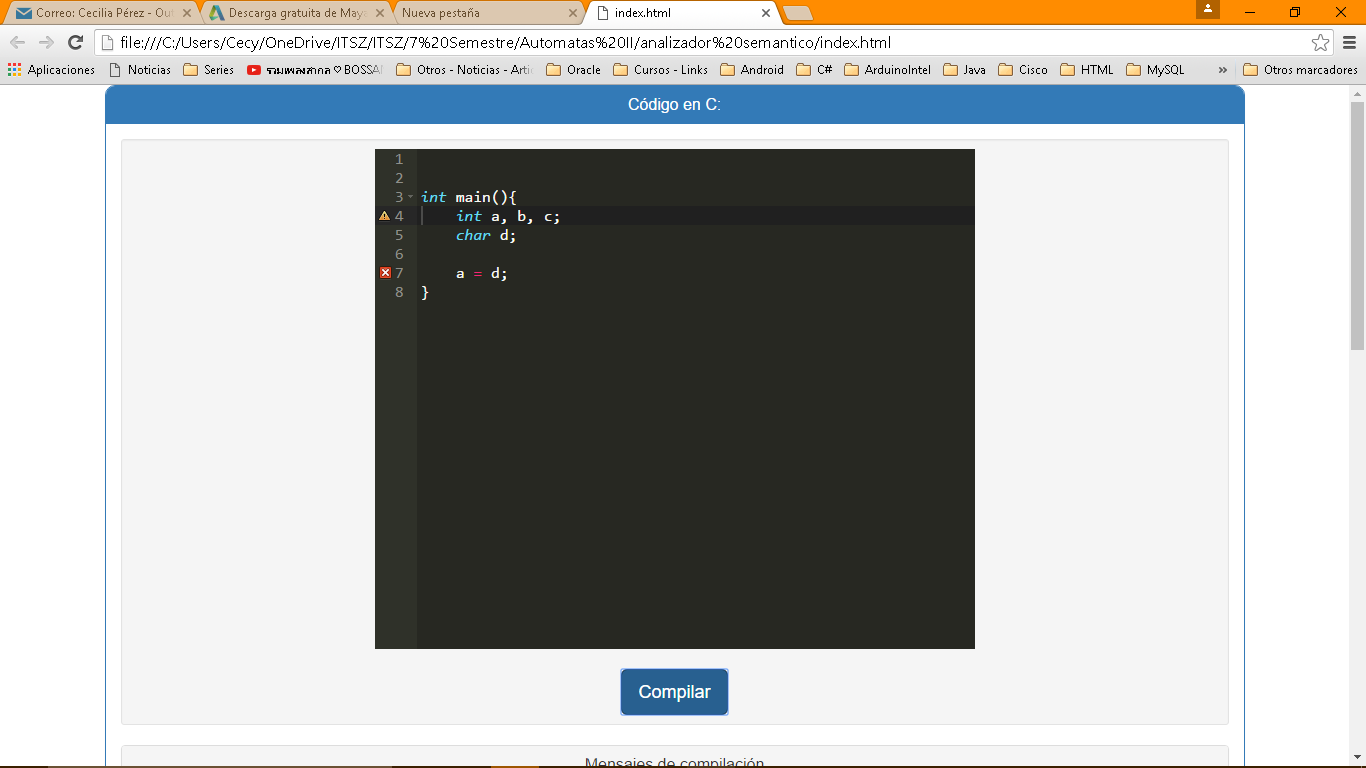


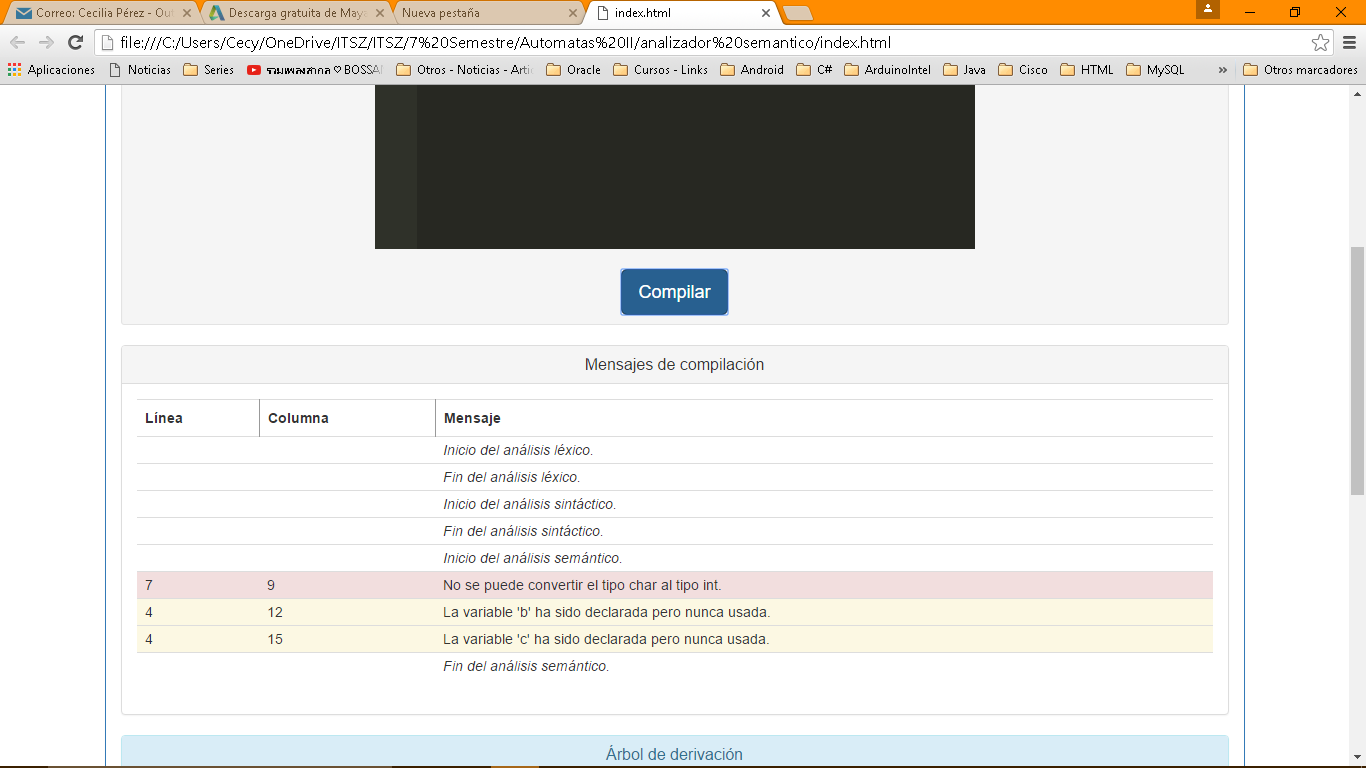
Nos aparecerá el siguiente error, que es un error de sintaxis porque se esperaba un ; al final de la sentencia.



* **Ejemplo de incompatibilidad de tipos**

Viendo otro ejemplo de incompatibilidad de tipos, tenemos int y char, en el ejemplo asignaremos un char a int pero saldrá el error de incompatibilidad.

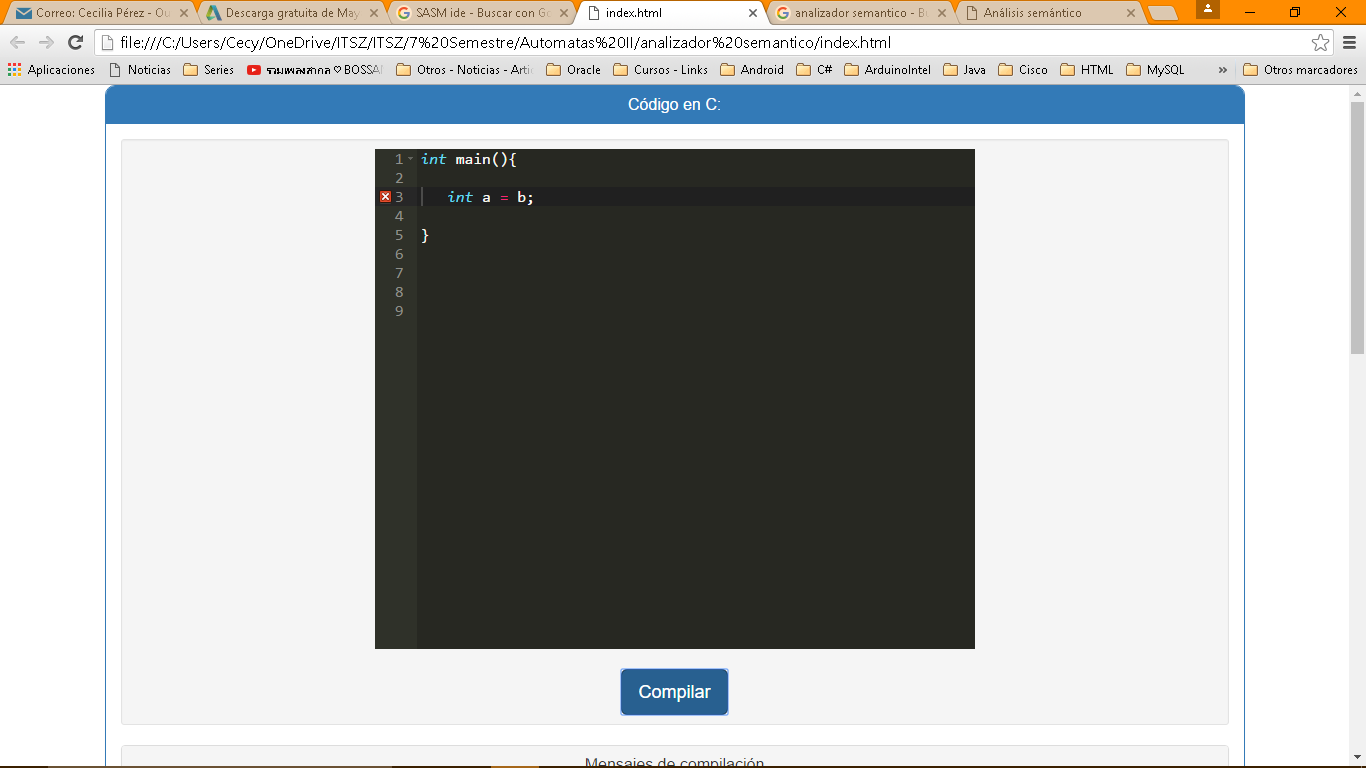


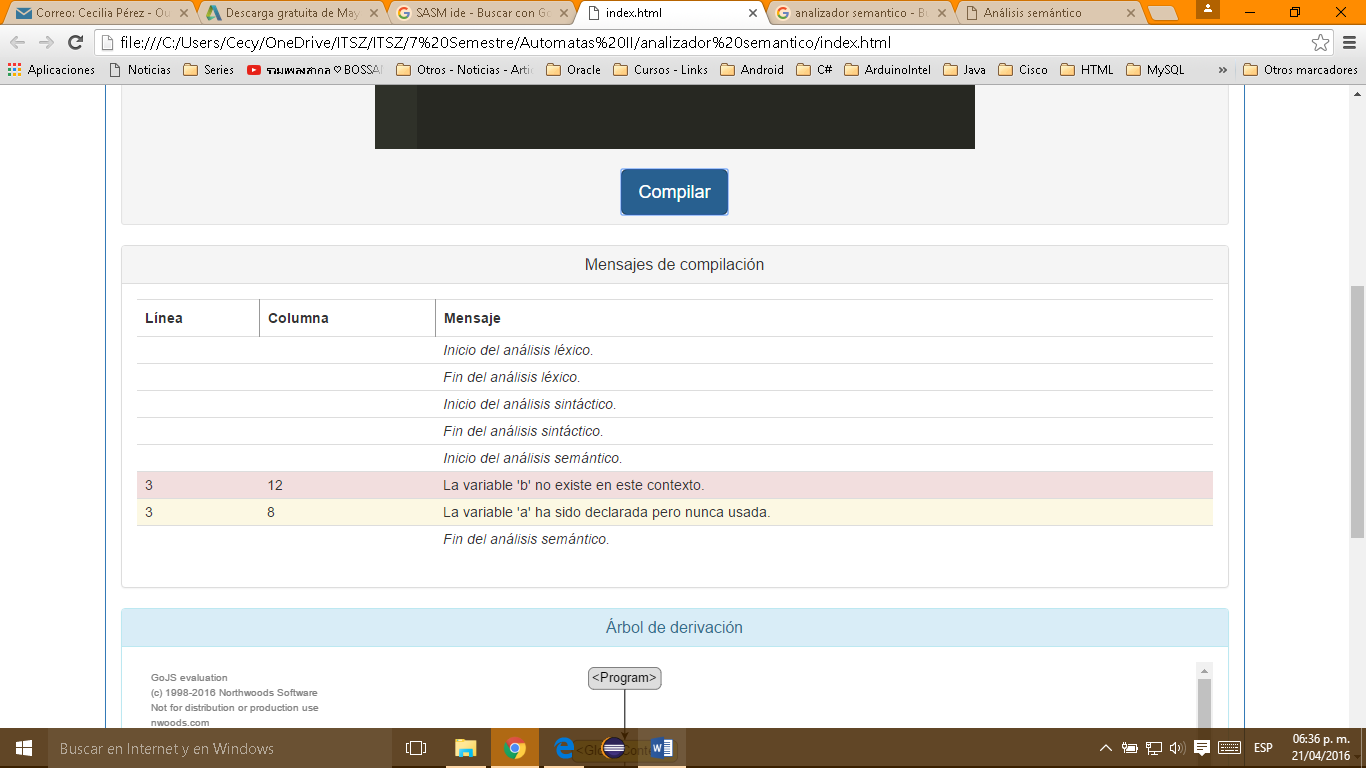


Además nos aparecen los warning de desperdicio de memoria ya que hemos declarado algunas variables int pero nunca las usamos. Ahora veremos un ejemplo de incompatibilidad de tipos pero en funciones.

* **Ejemplo de utilización de variables sin declarar.**

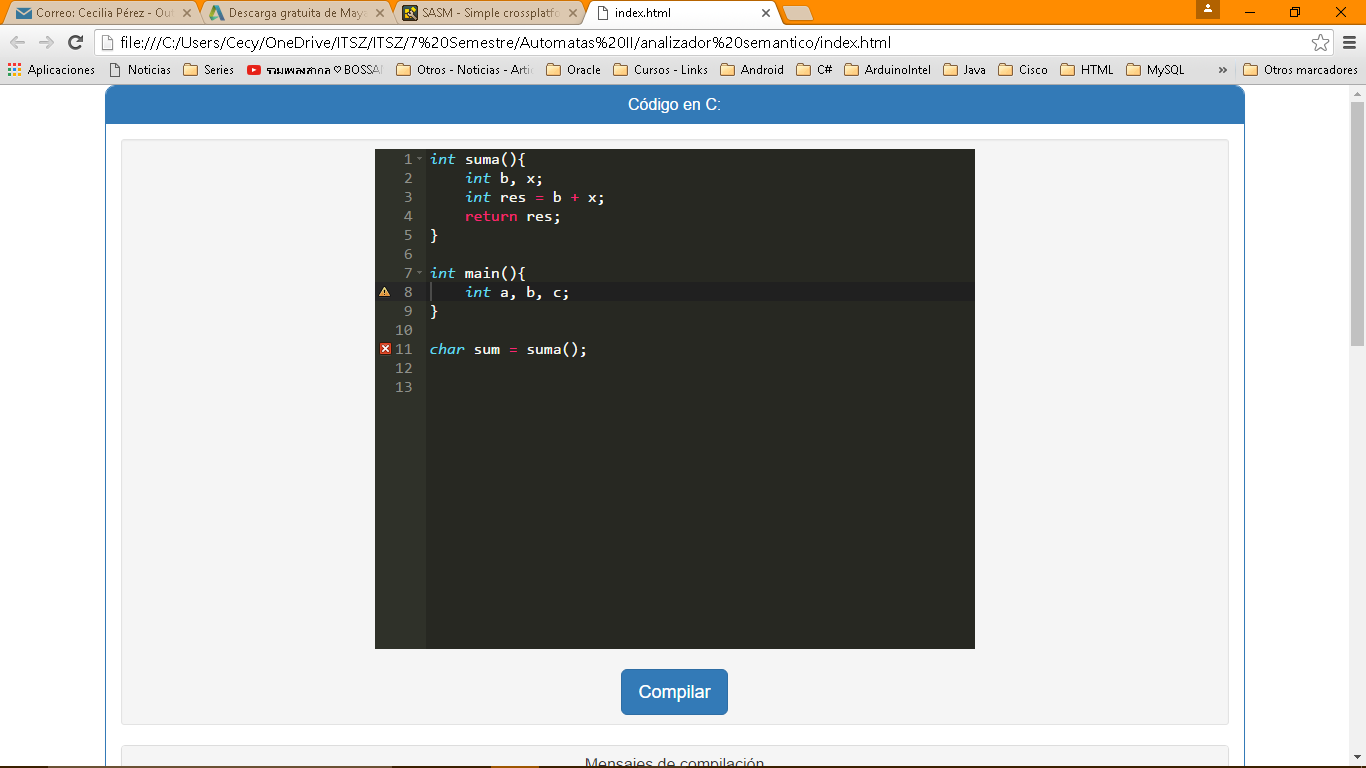
En este ejemplo, utilizaremos una variable donde se le asignarán variables sin asignar, por tanto nos aparecerá un error de esa variable sin declarar no existe.

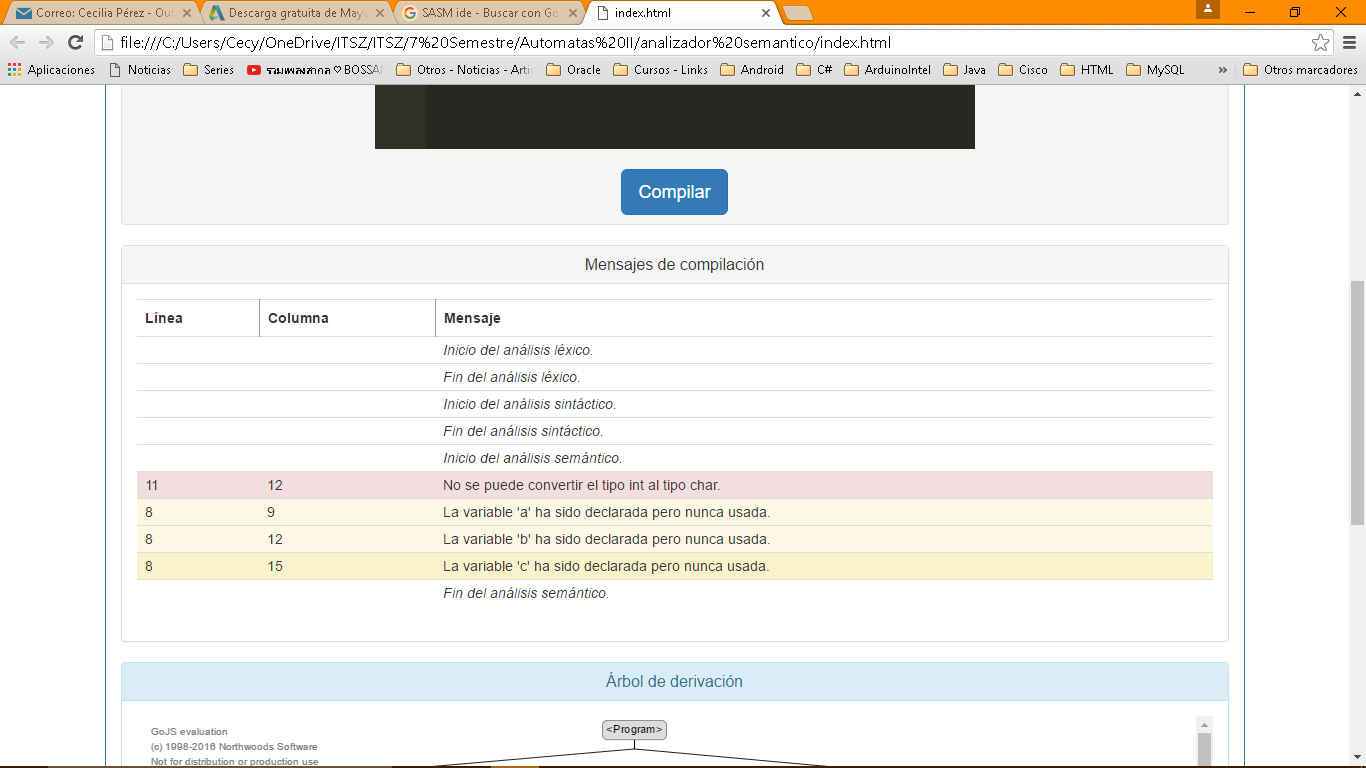




* **Ejemplo de incompatibilidad de tipos en funciones.**

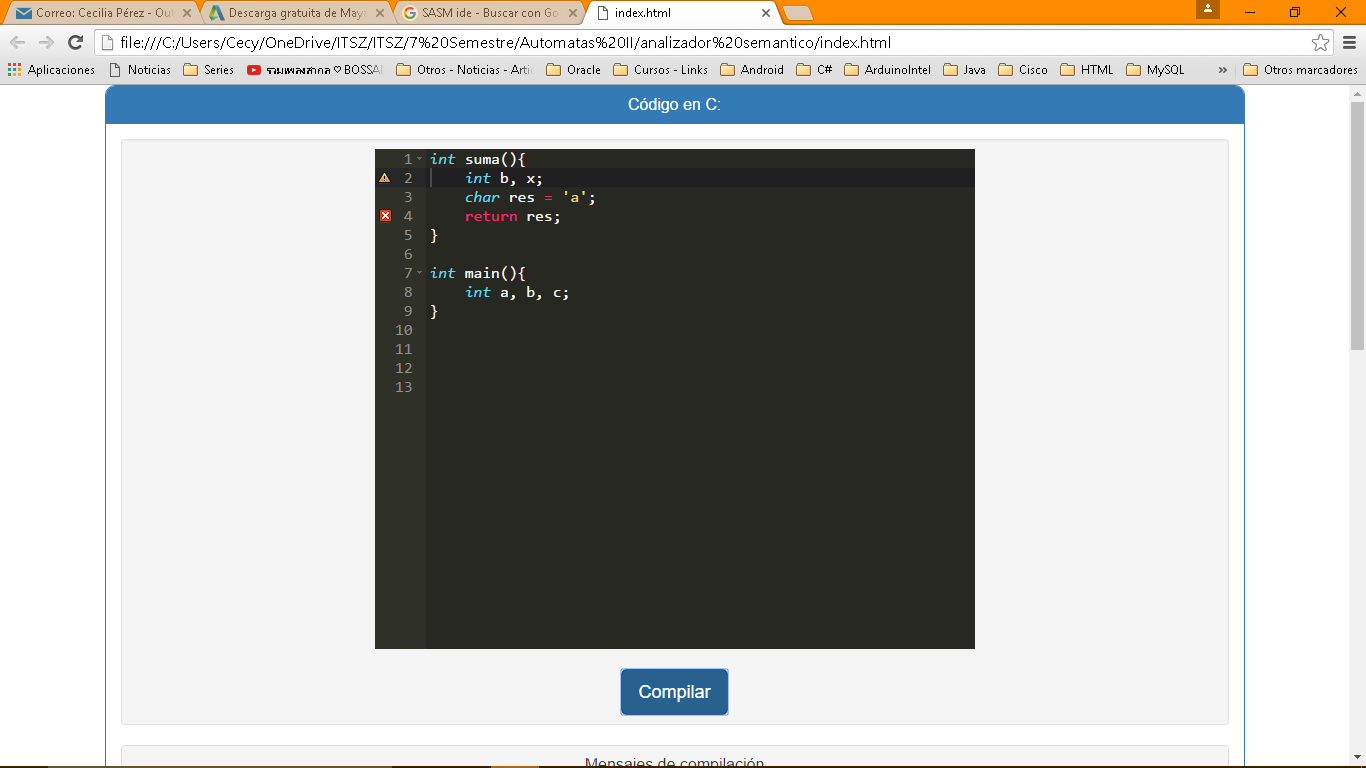
En este ejemplo, tenemos una función llamada suma, la cual retorna un entero, pero cuando lo asignamos, lo asignamos a un tipo char y nos sale un error.

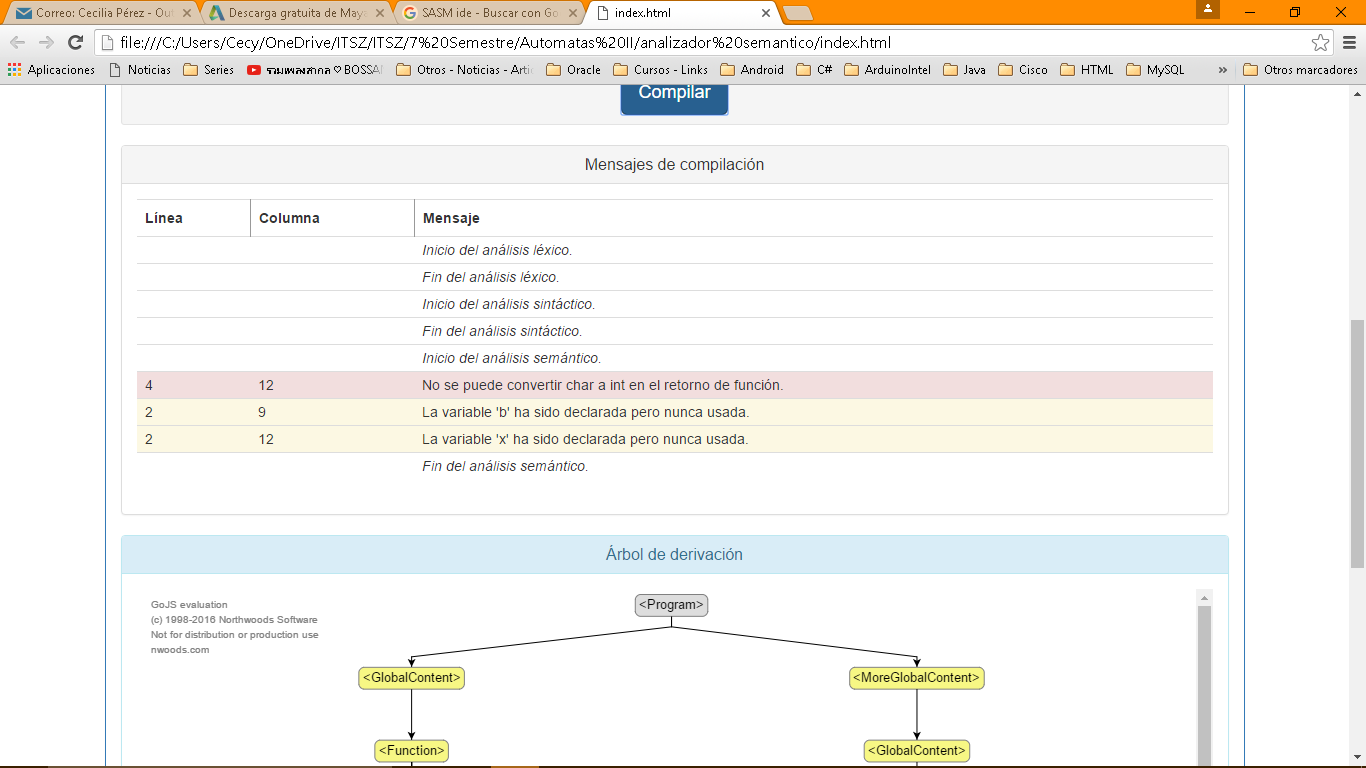




* **Ejemplo de incompatibilidad de retorno.**

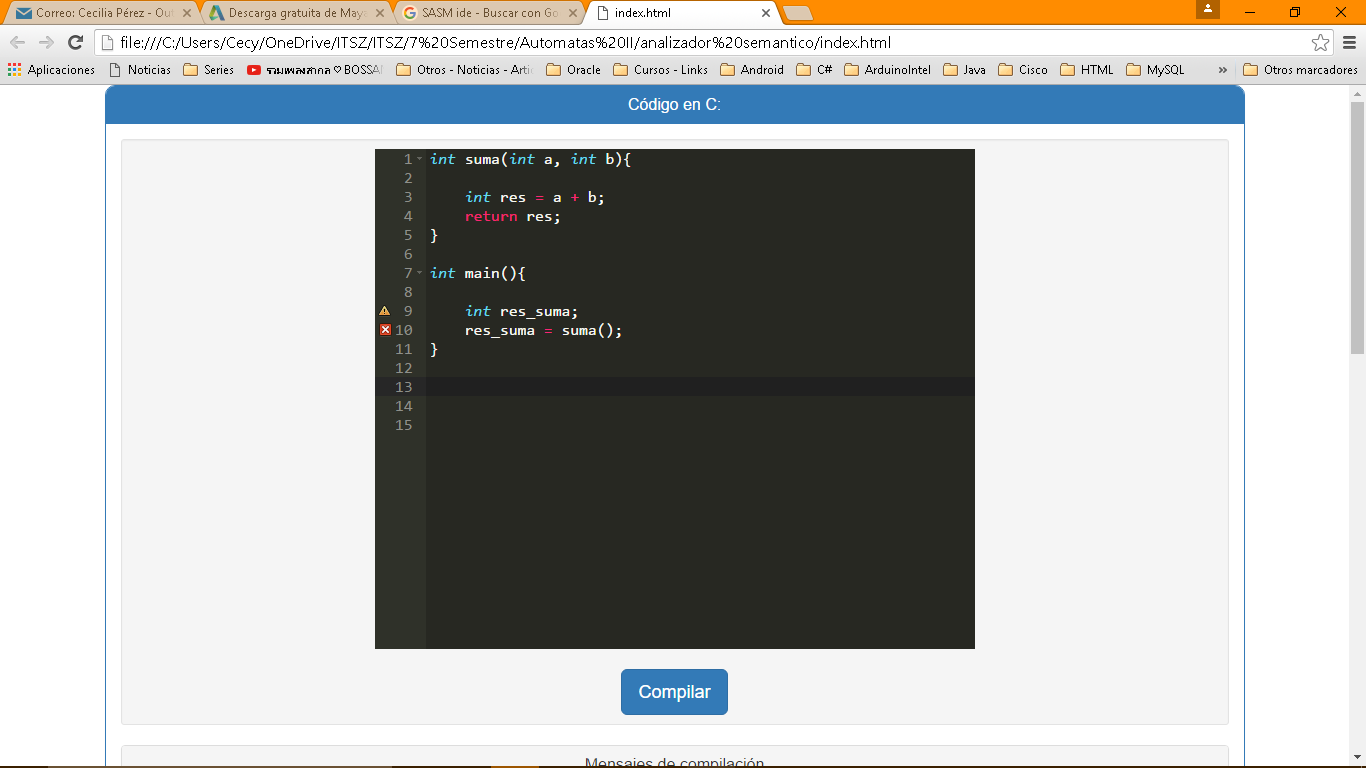
Este tipo de error ocurre cuando declaramos la función de un tipo y retornamos otro tipo, en este caso la función fue declarada de tipo int y se retorna un tipo char.





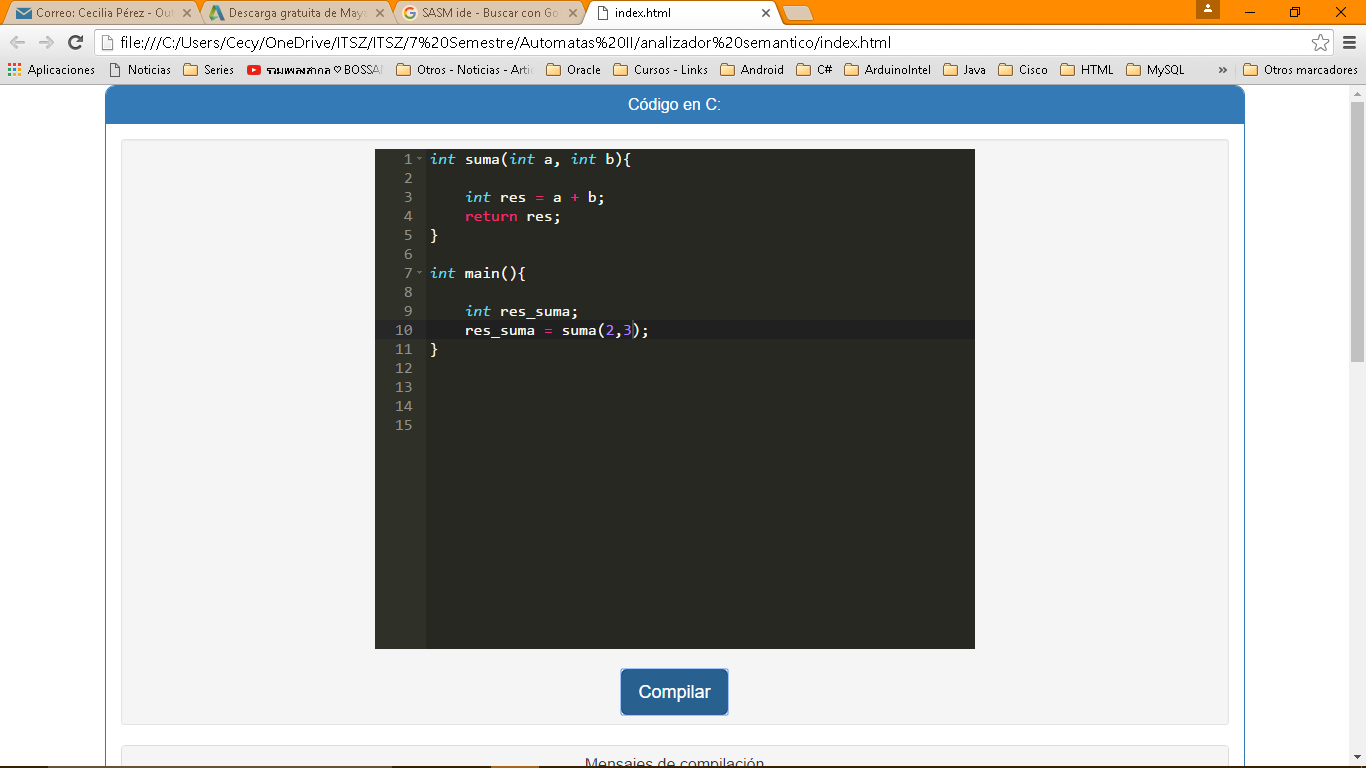
* **Ejemplo de error de argumentos en funciones.**

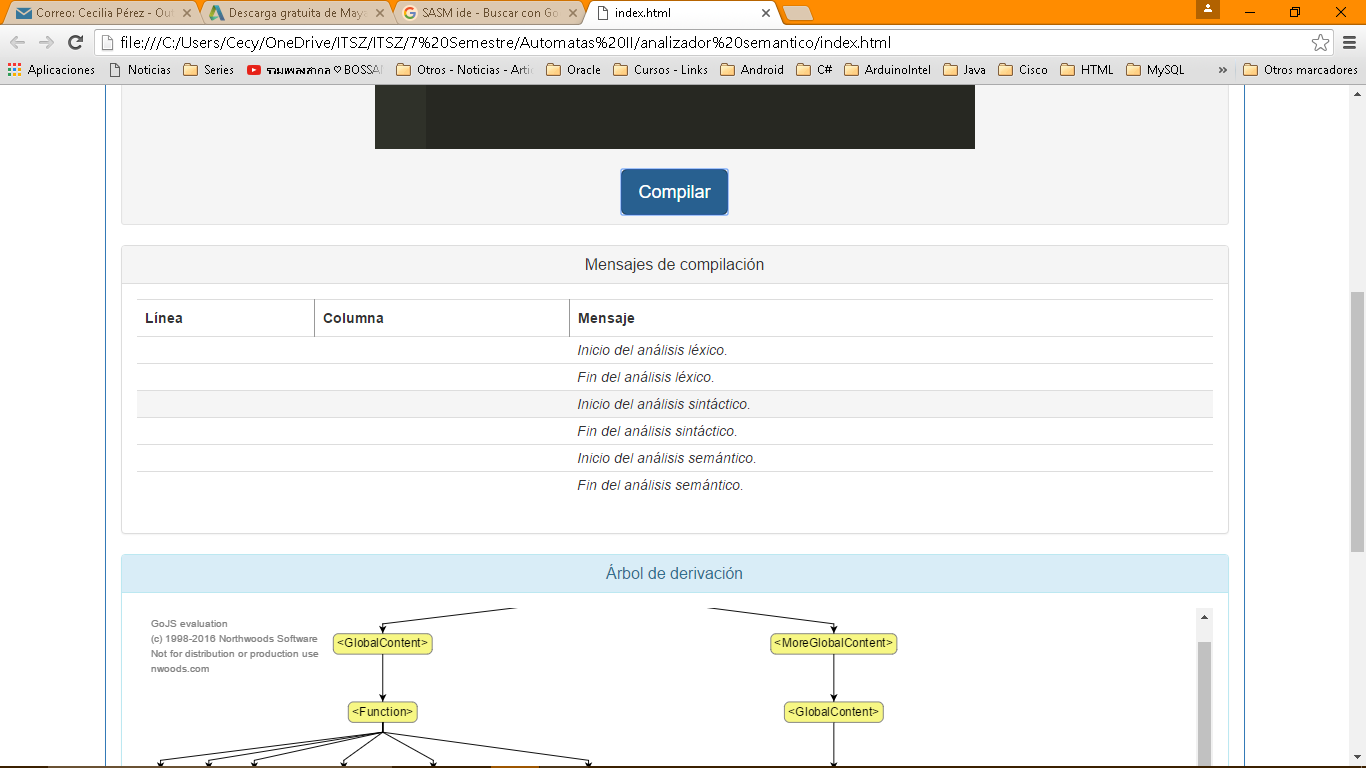
Esto ocurre cuando las funciones tienen parámetros, pero cuando las mandamos llamar, no la utilizamos con los parámetros que se piden.





* **Ejemplo del programa sin errores**





# Conclusiones

Esta práctica fue interesante pero compleja pues el objetivo desde un principio fue el desarrollo de un compilador para el lenguaje C desde cero, realizando cada una de las fases o componentes que lleva un compilador, es decir, primeramente debimos realizar la gramática del lenguaje C tomando solo las reglas principales que se utilizan en cualquier programa escrito en C pero fue complicado ya que aun simplificando la gramática resultaron casi 60 reglas gramaticales. Una vez realizada la gramática, se prosiguió a elaborar la tabla de símbolos o tokens de nuestro lenguaje para realizar el analizador léxico el cual evalúa si el código de entrada contiene símbolos válidos y genera los tokens que fueron elaborados en la tabla de símbolos o tokens de nuestro lenguaje y a su vez son enviados al analizador sintáctico que se encarga de evaluar si el código de entrada está correctamente escrito donde los tokens deben tener una forma estructurada y ordenada para generar una árbol de derivación el cuál es recibido por el analizador semántico para validar si el código de entrada tiene un sentido de ser, comprobando la compatibilidad de tipos, declaración de variables, desperdicio de memoria, entre otras. Y con esto finalizar el análisis mostrando mensajes de error o advertencias si hubo errores de compilación ya sea en el análisis léxico, sintáctico o semántico.

# Bibliografía

Anónimo (SIN FECHA). Capítulo 5, Análisis Semántico. URL: <http://arantxa.ii.uam.es/~alfonsec/docs/compila5.htm>

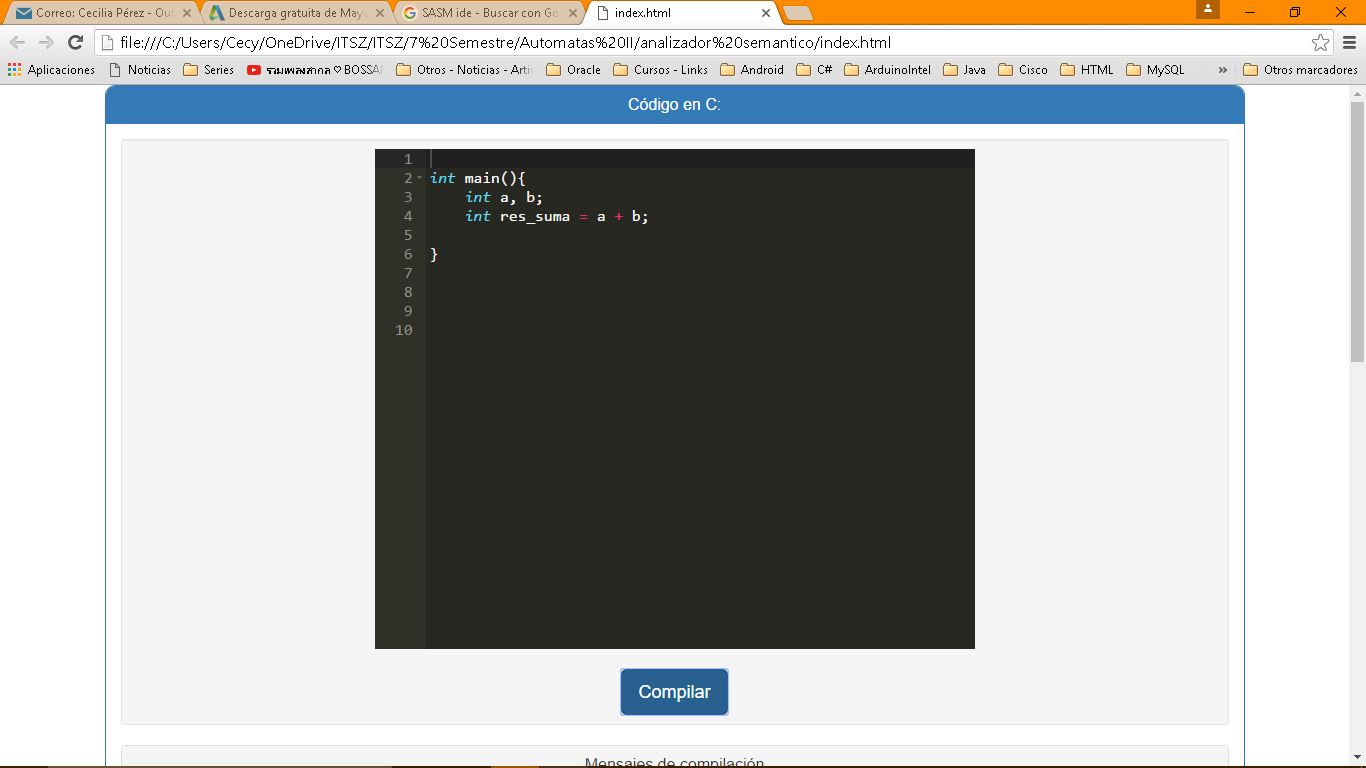
Universidad de Oviedo (2004). Análisis Semántico en Procesadores de Lenguajes. Editoral Servitec. ISBN: 84-688-6208-8

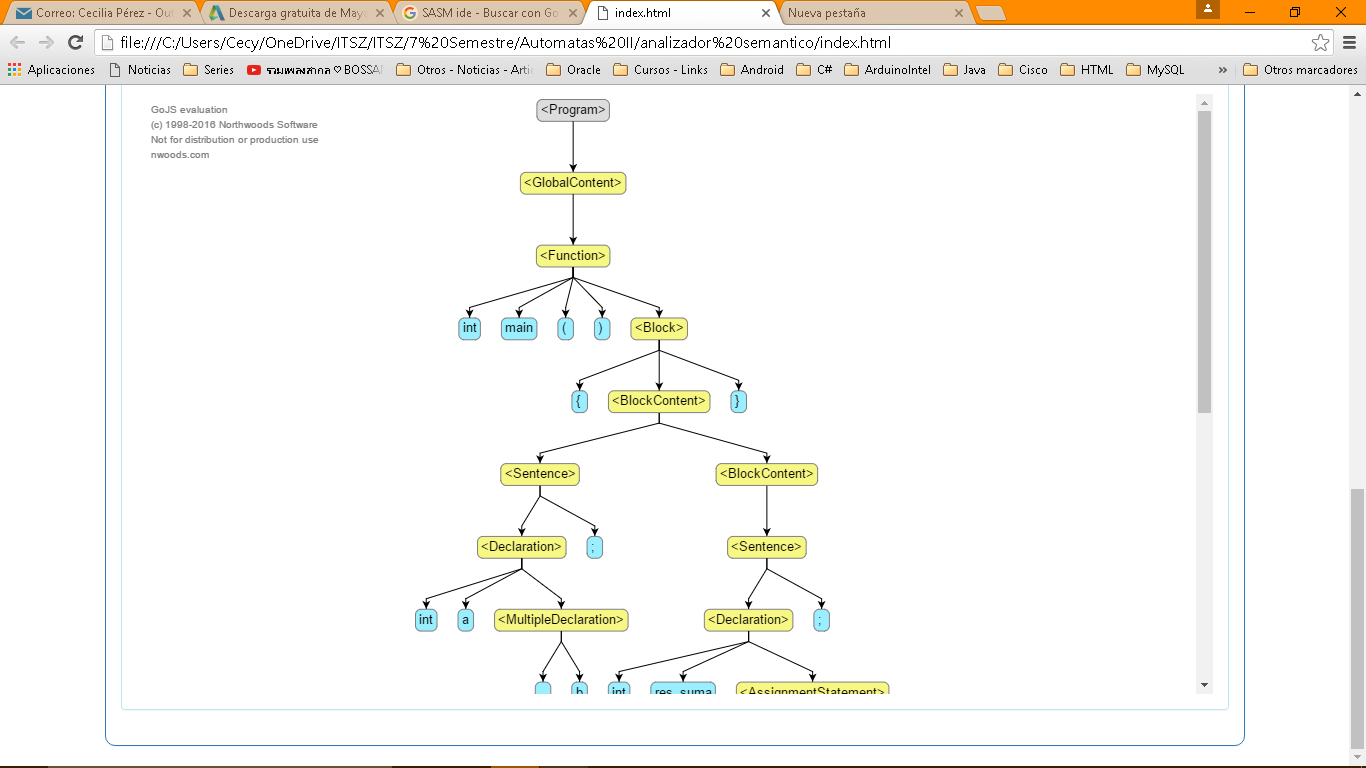
Universitat Jaume. (SIN FECHA). Procesadores de Lenguajes, Análisis Semántico.

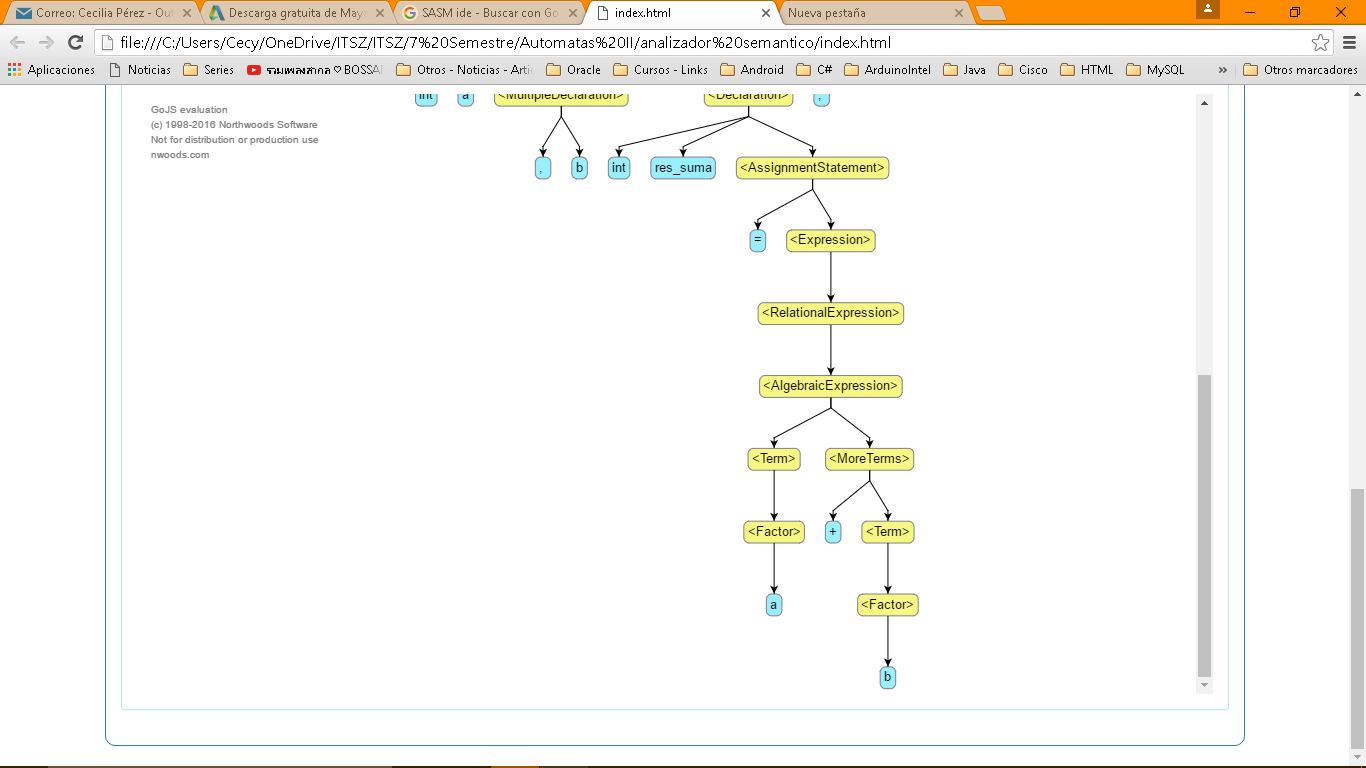
# Apéndice

# Árbol de Derivación

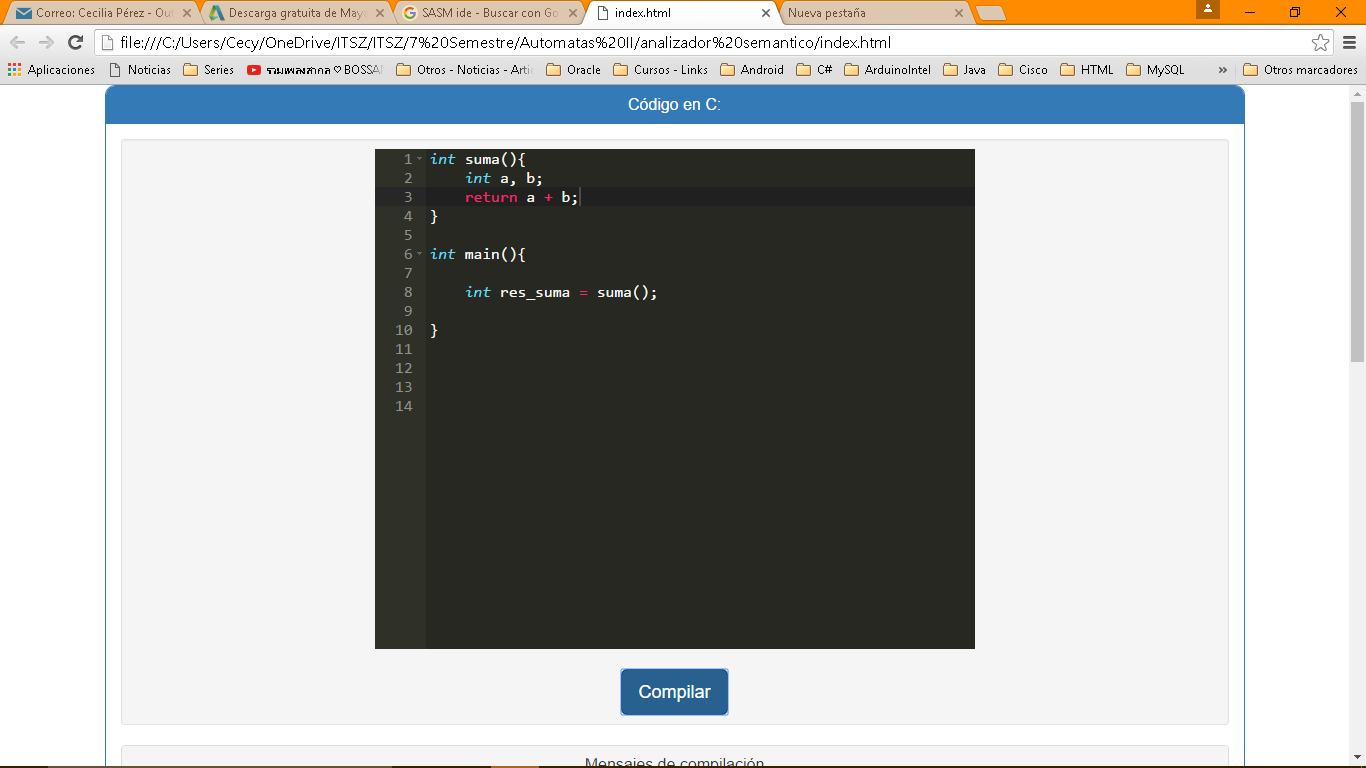
Los arboles de derivación se van generando con cada programa compilado, se podrán algunos ejemplos de sumas, restas y utilización de funciones.



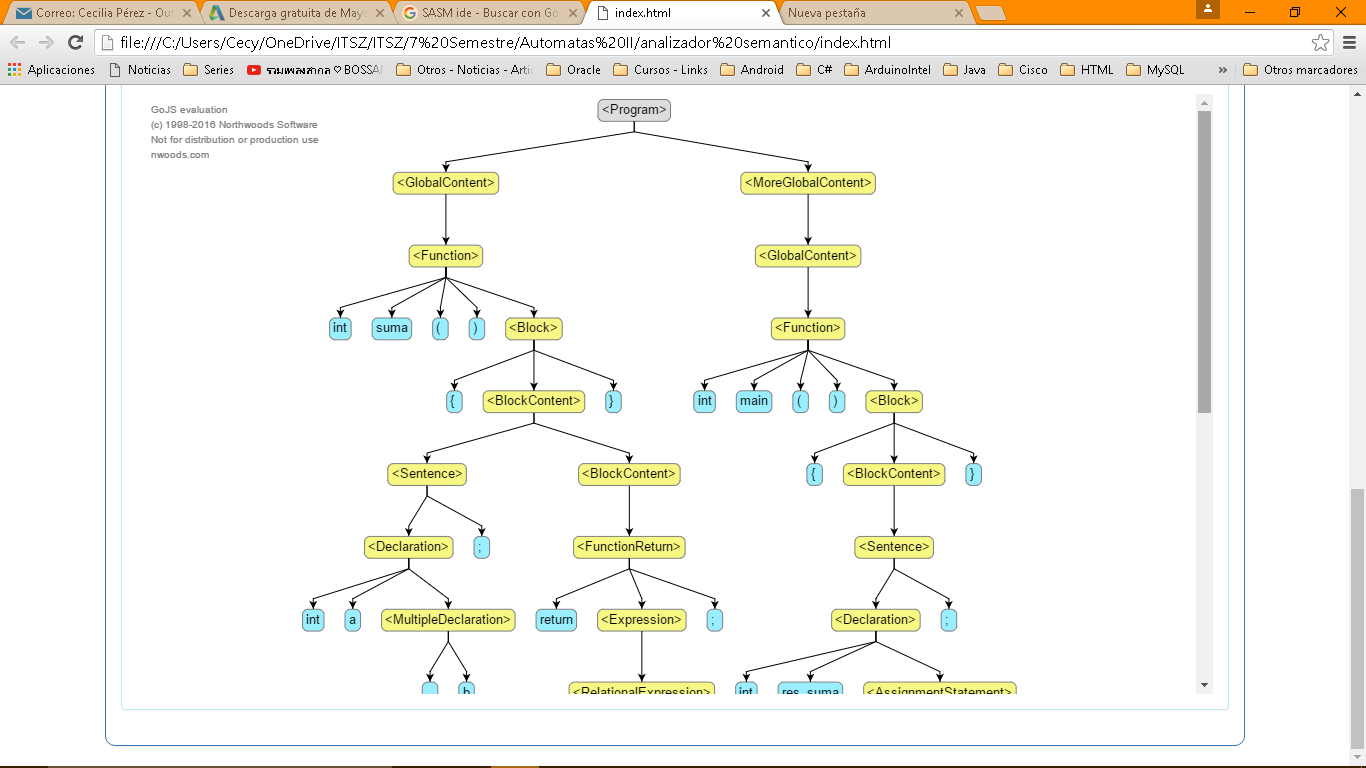


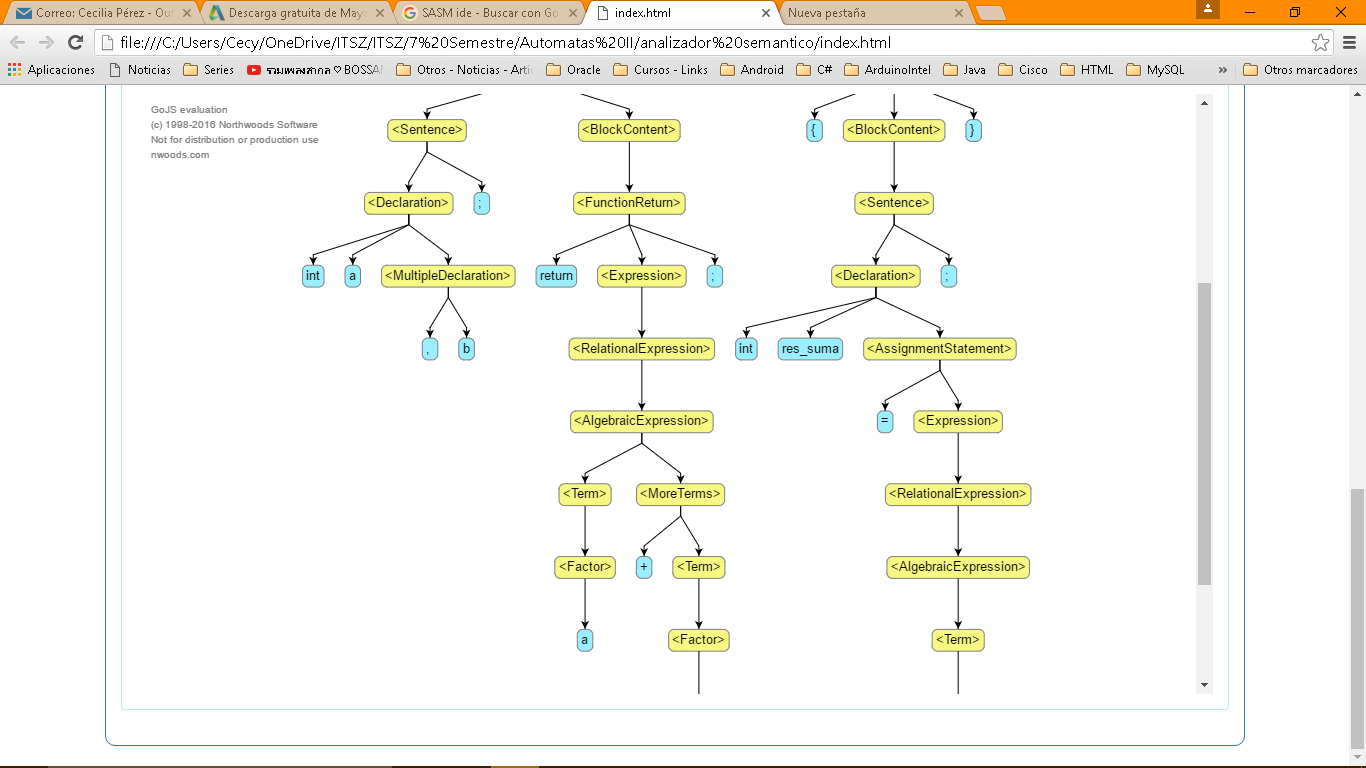


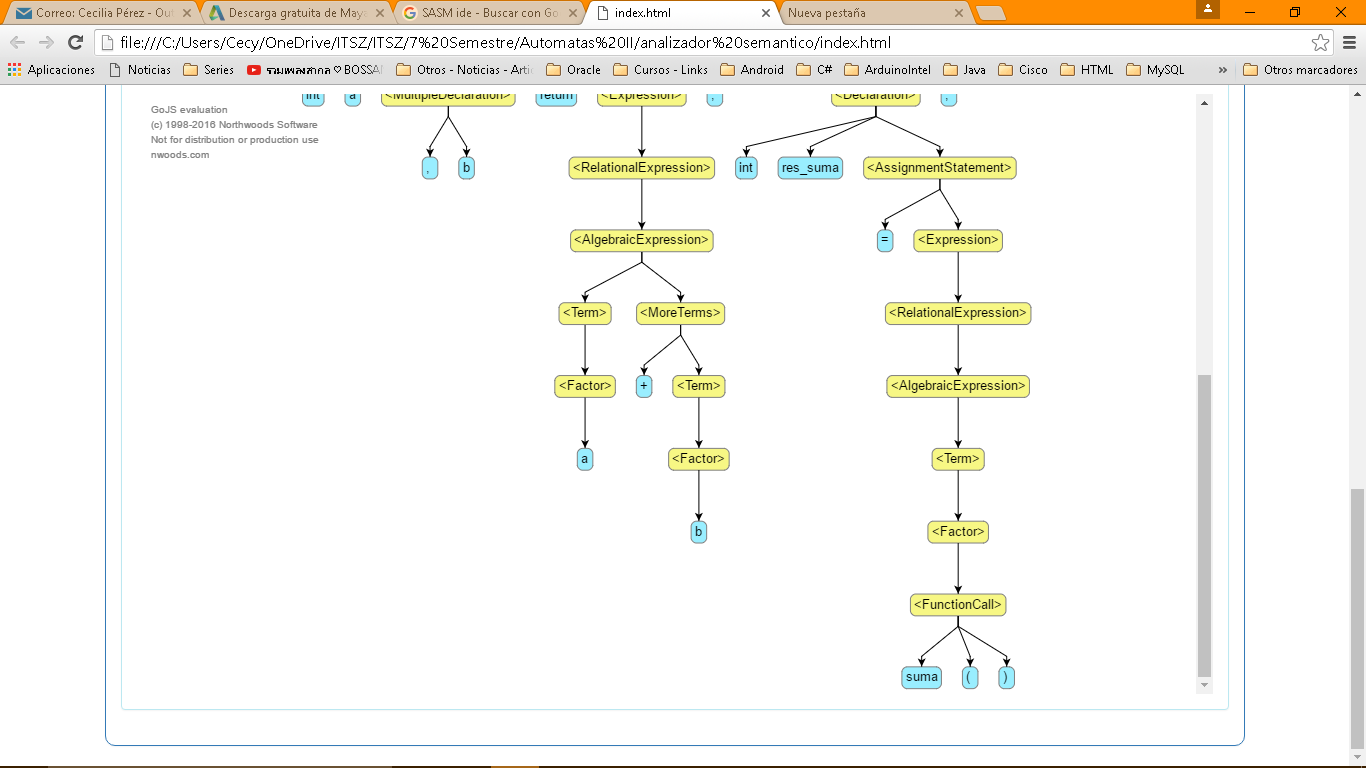
Árbol de derivación utilizando funciones. Primero tenemos nuestro código a compilar.



A continuación se muestra el árbol de derivación de este pequeño código que se compiló.







# Acrónimos

Ninguno se utilizó para esta práctica.