



UNIVERSIDAD DE  
GUADALAJARA  
Red Universitaria e Institución Benemérita de Jalisco

# Centro Universitario de Ciencias Exactas e Ingenierías

---

## Seminario de solución de problemas de traductores II

Profesor: José Juan Meza  
Espinoza

Sección: D01

Practica 2: Analizador  
semántico

### Alumnos:

- Botello Martínez Nadia Noemi
- Ángel Gabriel Mercado Hernández
- Ivo Alberto Ramirez Gaeta
- Uziel Reyes Becerra

09 de noviembre 2024

## 1. Introducción

El **analizador semántico** es una fase crucial en el proceso de compilación de un lenguaje de programación, que se lleva a cabo después del análisis léxico y sintáctico. Su principal objetivo es garantizar que el programa cumpla con las reglas semánticas del lenguaje, verificando que las construcciones del código tengan sentido lógico y coherente. A diferencia del análisis sintáctico, que se enfoca en la estructura gramatical, el análisis semántico valida aspectos como la correcta declaración y uso de variables, la compatibilidad de tipos de datos en las operaciones y la correcta invocación de funciones. Esta etapa ayuda a detectar errores que, aunque no impiden que el programa sea aceptado sintácticamente, pueden generar comportamientos erróneos en tiempo de ejecución.

## 2. Contenido

En esta actividad se detalla el código desarrollado en lenguaje C para implementar un analizador semántico, incluyendo algunos ejemplos de aplicación.

Este programa se basa en el anterior, que contiene un analizador léxico y sintáctico. Debido a que esos fragmentos de código ya fueron explicados en la actividad previa, no se volverán a describir aquí. Así que, comencemos.

Para este código se utilizó una gramática que describe la sintaxis de un programa **sencillo** en lenguaje C. Esta es la siguiente:

```
E' -> E
E -> int M
M -> main(){ A
A -> a=S
S -> n+n;P
P -> }
```

Estado	
0	E' -> • E E -> • int M
1	E' -> E •
2	E -> int • M M -> • main() { A
3	M -> main() { • A A -> • a = S

4	A -> a = • S S -> • n + n ; P
5	S -> n • + n ; P
6	S -> n + • n ; P
7	S -> n + n • ; P
8	S -> n + n ; • P P -> • }
9	P -> } •

Tabla LR(0):

Estado	Int	main()	{	a	=	n	+	;	}	E	M	A	S	P
0	S2									1				
1									Acc					
2		S3									4			
3			S5											
4				S&								7		
5						S8								
6					S9									
7						S10								
8							S11							
9									R5					

## Código

Aquí definimos vectores, tipos de datos, operadores, números, paréntesis, corchetes y en general caracteres especiales que utilizaremos en el analizador para identificar y clasificar los diferentes tipos de elementos:

```
vector<string> dataType = {"float","int","signed","unsigned"};

vector<string> keywords = {"auto","break","case","char","const","continue","default",
    "do","double","else","enum","extern","for","goto",
    "if","signed","sizeof","static","struct","switch","typedef","union",
    "void","volatile","while"
    };

vector<string> operators = {"+", "-", "*", "/", ">", "<", ">=", "<=", "||", "&&", "!=", "="};
vector<string> numbers = {"0","1","2","3","4","5","6","7","8","9"};
vector<string> parenthesis = {"(", ")", "("};
vector<string> brackets = {"[", "]", "["};
vector<string> brace = {"{", "}", "{"};
vector<string> specialChar = {";", " ", ".", "%", "#", "!"};
```

En esta función se recibe una cadena, y verifica si corresponde a alguno de los tipos de datos, utilizamos find para buscar la cadena de los vectores definidos y clasificarlos y si o encuentra el token en los vectores entonces divide la cadena en partes y las procesa.

```
void lexer(string myString)
{
    myString.erase(std::remove(myString.begin(), myString.end(), '\n'), myString.end());
    /// Eliminamos posibles saltos de linea
    char chars[] = {'=', '+', '-', '*', '/', '(', ')', ';', ',', ' '};
    size_t position = myString.find_first_of(chars);

    if(find(keywords.begin(), keywords.end(), myString) != keywords.end())
        cout<<myString<<" \t Palabra reservada\n";

    if(find(dataType.begin(), dataType.end(), myString) != dataType.end())
        cout<<myString<<" \t Tipo de dato\n";
}
```

En esta función en caso de encontrar operadores o caracteres especiales se divide la cadena en tres partes y la llama recursivamente para procesarla

```
void others(string myString, size_t position)
{
    if (position == 1)    /// Si solo hay un caracter antes del signo o caracter (parte
    izq)
    {
        string newString = string(1, myString[0]);
        lexer(newString);
    }
    else /// Si hay mas de un caracter antes del signo o caracter (parte izq)
        lexer(myString.substr(0, position));

    string newString2 = string(1, myString[position]); /// Tomamos en cuenta el signo o
    caracter (mid)
    lexer(newString2);

    if (myString.length() > position + 1)
        lexer(myString.substr(position + 1)); /// Tomamos en cuenta el lado derecho del
    signo o caracter (der)
}
```

Aquí verificamos si una subcadena (sub) existe dentro de una cadena (str). Usa find para buscar la subcadena y devuelve true si la encuentra, o false si no.

```
bool contains(const std::string &str, const std::string &sub) {  
    // Busca la subcadena en la cadena  
    return str.find(sub) != std::string::npos;  
}
```

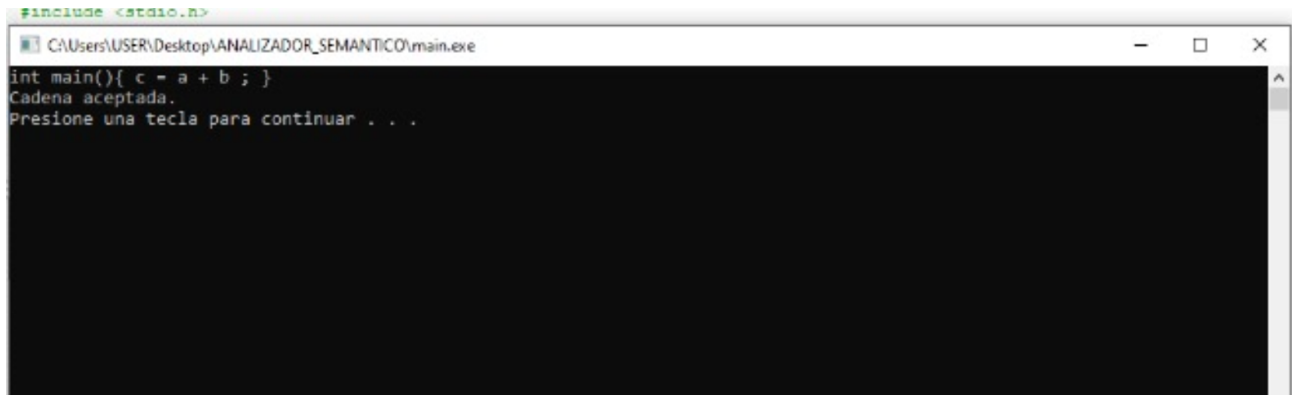
Los valores positivos indican un "shift", los valores negativos indican una "reducción", y los valores especiales como 500 indican una aceptación.

```
int tablaLR[estados][simbolos] =  
{  
    // E M A S P i m a n + ; = } $  
    /* 1 */ { 1,0,0,0,0,2,4,6,9,0,0,0,14,0 },  
    /* 2 */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,500 },  
    /* 3 */ { 0,3,0,0,0,0,4,0,0,0,0,0,0,0 },  
    /* 4 */ { 0,0,0,0,0,-2,-2,-2,-2,-2,-2,-2,-2,-2 },  
    /* 5 */ { 0,0,5,0,0,0,0,6,0,0,0,0,0,0 },  
    /* 6 */ { 0,0,0,0,0,-3,-3,-3,-3,-3,-3,-3,-3,-3 },  
    /* 7 */ { 0,0,0,0,0,0,0,0,0,0,0,0,7,0 },  
    /* 8 */ { 0,0,0,8,0,0,0,0,9,0,0,0,0,0 },  
    /* 9 */ { 0,0,0,0,0,-4,-4,-4,-4,-4,-4,-4,-4,-4 },  
    /* 10 */ { 0,0,0,0,0,0,0,0,0,10,0,0,0,0 },  
    /* 11 */ { 0,0,0,0,0,0,0,0,0,11,0,0,0,0 },  
    /* 12 */ { 0,0,0,0,0,0,0,0,0,12,0,0,0,0 },  
    /* 13 */ { 0,0,0,0,13,0,0,0,0,0,0,0,14,0 },  
    /* 14 */ { 0,0,0,0,0,-5,-5,-5,-5,-5,-5,-5,-5,-5 },  
    /* 15 */ { 0,0,0,0,0,-6,-6,-6,-6,-6,-6,-6,-6,-6 }  
};
```

Esta función convierte un símbolo en su valor entero corre. Este valor es necesario para consultar la tabla LR.

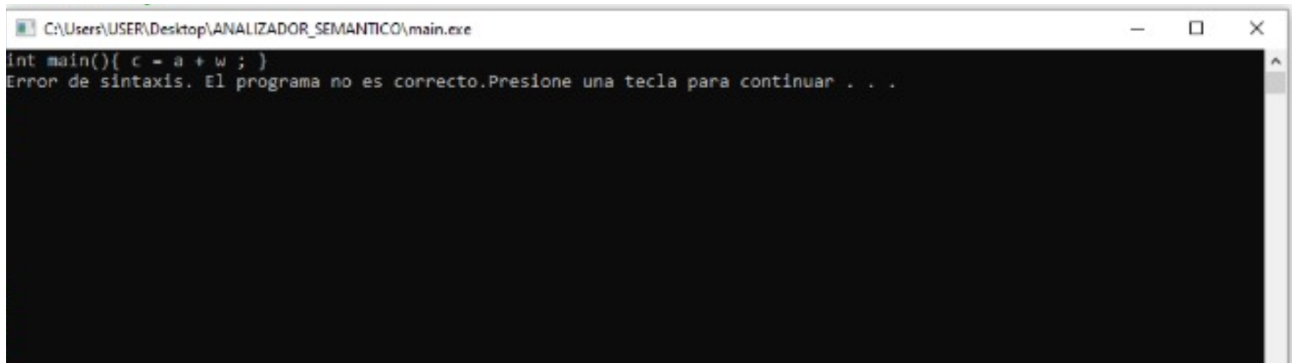
```
int simboloAEntero(string palabra)  
{  
    /* E M A S P i m a n + ; = } $ */  
    // int main(){ c=a+b; }  
    if (palabra == "E")  
    {  
        return 0;  
    }  
}
```

### 3. Resultados



```
#include <stdio.h>

int main(){ c = a + b ; }
Cadena aceptada.
Presione una tecla para continuar . . .
```



```
C:\Users\USER\Desktop\ANALIZADOR_SEMANTICO\main.exe
int main(){ c = a + w ; }
Error de sintaxis. El programa no es correcto.Presione una tecla para continuar . . .
```

### 4. Conclusión

El analizador semántico es una parte fundamental de los compiladores y traductores de lenguajes de programación, que se encarga de verificar la corrección semántica de un programa, es decir, que el código no solo sea sintácticamente válido, sino también lógico y coherente según las reglas del lenguaje. A diferencia del analizador léxico y sintáctico, que se enfocan en la estructura y los símbolos, el analizador semántico se centra en aspectos como la declaración y el uso de variables, los tipos de datos, las asignaciones y las operaciones.

## 5. Bibliografía

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compiladores: Principios, técnicas y herramientas* (2da ed.). Pearson Education.