



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 5: Producto integrador

Alumnos:

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

14 noviembre 2024

Introducción

Esta actividad es el proyecto se creo un compilador en lenguaje C que incluye las tres etapas clave del proceso de compilación: análisis léxico, sintáctico y semántico, desarrollando un sistema capaz de leer, interpretar y verificar el código de un programa en C.

Contenido

Analizador Léxico: En esta etapa, el compilador revisa el código para encontrar las partes fundamentales del lenguaje, como palabras reservadas, operadores y símbolos especiales. Usando vectores y la función `lexer`, el compilador identifica y clasifica cada elemento según su tipo (por ejemplo, palabra reservada, tipo de dato, operador, etc.), lo que facilita su análisis en las siguientes fases.

Comenzamos definiendo nuestros vectores con algunos de los lexemas que se pueden implementaren el lenguaje del siguiente modo:

```
10  vector<string> dataType = {"float","int","signed","unsigned"};
11
12  vector<string> keywords = {"auto","break","case","char","const","continue","default",
13  |                           "do","double","else","enum","extern","for","goto",
14  |                           "if","signed","sizeof","static","struct","switch","typedef","union",
15  |                           "void","volatile","while"
16  |                           };
17
18  vector<string> operators = {"+","-","*","/",">","<",">=","<=","||","&&","!=","="};
19  vector<string> numbers = {"0","1","2","3","4","5","6","7","8","9"};
20  vector<string> parenthesis = {"(",")","{","}"};
21  vector<string> brackets = {"[","]"};
22  vector<string> brace = {"{","}"};
23  vector<string> specialChar = {";",",",".", "%", "#", "!"};
24
25  string line; /// Linea a leer, para analisis sintactico
```

Código en el cual se utilizan vectores para, como se verá posteriormente, utilizar funciones de búsqueda dentro de ellos. Así, al iniciar el programa, se ejecuta la siguiente función `main()`: Se encarga de leer un conjunto de líneas del declado, separándolas en palabras distintas para después llamar a la función `lexer` que es nuestro analizador lexico.

Código en el cual se utilizan vectores para, como se verá posteriormente, utilizar funciones de búsqueda dentro de ellos. Así, al iniciar el programa, se ejecuta la siguiente función `main()`:

Se encarga de leer un conjunto de líneas del declado, separándolas en palabras distintas para después llamar a la función `lexer` que es nuestro analizador lexico.

```
31 void lexer(string myString)
32 {
33     myString.erase(std::remove(myString.begin(), myString.end(), '\n'), myString.end());
34     char chars[] = {'=', '+', '-', '*', '/', '(', ')', ';', ',', ' '};
35     size_t position = myString.find_first_of(chars);
36
37     if(find(keywords.begin(), keywords.end(), myString) != keywords.end())
38         cout<<myString<<" \t Palabra reservada\n";
39
40     if(find(dataType.begin(), dataType.end(), myString) != dataType.end())
41         cout<<myString<<" \t Tipo de dato\n";
42
43     else if((isalpha(myString[0]) || myString[0] == '_') && position == string::npos) //
44     {
45         for (int i = 0; i < myString.length(); i++)
46         {
47             if (!isalnum(myString[i]) && myString[i] != '_')
48             {
49                 return;
50             }
51         }
52         cout<<myString<<" \t Identificador\n";
53     }
```

La cual recibe palabras (recordemos que eso es lo que se pasa como argumento en la función `main`) y las analiza, intentando encontrar una coincidencia. Por ejemplo, si la palabra recibida es `"int"`, entonces buscará la coincidencia, entre otros vectores, en el vector `"dataType"` o, si la palabra recibida es `"void"`, entonces la buscará y encontrará en el vector `"keywords"`, y así sucesivamente. Además, como se ve en el código, se utiliza la función `"find"` para hallar la coincidencia de la palabra en el vector.

Finalmente, lo más complejo se deja para la parte `"else"` de la estructura selectiva múltiple, en donde se validan palabras del tipo `"ixd"`, en donde existen más de una palabra, pero sin espacios entre ellas y existe una parte izquierda `"i"`, luego una parte intermedia `"x"` y una parte derecha `"d"`, por ejemplo: `c=a+b;`, en donde `i = c`, `x = =`, y `d = a+b;`, es decir, todo lo previo al primer operador carácter especial es `i`, el operador o carácter especial es `x` y todo a su derecha es `d`. Entonces, lo que hace la función `"others"`, llamada en esta parte, es precisamente dividir la palabra en la forma `"ixd"`, y lo hace del siguiente modo:

```

81 void others(string myString, size_t position)
82 {
83     if (position == 1)    /// Si solo hay un caracter antes del signo o caracter (parte izq)
84     {
85         string newString = string(1, myString[0]);
86         lexer(newString);
87     }
88     else /// Si hay mas de un caracter antes del signo o caracter (parte izq)
89     {
90         lexer(myString.substr(0, position));
91     }
92     string newString2 = string(1, myString[position]); /// Tomamos en cuenta el signo o caracter (mid)
93     lexer(newString2);
94     if (myString.length() > position + 1)
95     {
96         lexer(myString.substr(position + 1)); /// Tomamos en cuenta el lado derecho del signo o caracter (der)
97     }
98 }

```

En donde se logra esta separación para, posteriormente, llamar al analizador léxico con cada palabra obtenida.

Así, de momento se tienen los siguientes resultados:

Programa a analizar:

```

int main()
{
int a,b,c; a = 1;
b = 2;
c = a + b; print("%d", c);
}

```

```

int main()
{
    int a,b,c;
    a = 1;
    b = 2;
    c = a + b;
    print("%d", c);
}
int      Tipo de dato
main     Identificador
(        Parentesis
)        Parentesis
{        Llave
int      Tipo de dato
a        Identificador
,        Caracter especial
b        Identificador
,        Caracter especial
c        Identificador
;        Caracter especial
a        Identificador
=        Operador
1        Constante
;        Caracter especial
b        Identificador
=        Operador
2        Constante
;        Caracter especial
c        Identificador
=        Operador
a        Identificador
+        Operador
b        Identificador
;        Caracter especial
print    Identificador
(        Parentesis
,        Caracter especial
c        Identificador
)        Parentesis
;        Caracter especial
}        Llave

Process returned 0 (0x0)   execution time : 1.474 s
Press any key to continue.

```

Analizador Sintáctico: Aquí, el compilador revisa la estructura del código según reglas gramaticales definidas previamente. Los estudiantes implementaron un analizador LR(0) descendente, que sigue una gramática específica para detectar errores de sintaxis y asegurar que el código tenga la estructura correcta según el lenguaje.

Este programa es una continuación del programa previo, que corresponde a un analizador léxico. La principal diferencia es que, para desarrollar el analizador sintáctico en esta ocasión, se requieren dos elementos:

- 1) Elegir el tipo de analizador a utilizar (ascendente o descendente).
- 2) Definir la gramática adecuada.

En este caso, se optó por desarrollar un analizador sintáctico descendente (LR(0)), con las siguientes reglas gramaticales para las distintas operaciones permitidas en el programa:

E' -> E

E -> int M

M -> main(){ A

A -> a=S

S -> n+n;P

P -> }

Como establece el método, se debe construir un autómata, el cual se muestra a continuación:

Tabla LR(0):

E	E2	M	A	S	P	Int S3	Main() S5	A S1	n S10	+	;	=	} S15	\$
1														
2							S2							
3		4						S1						
4			6			r2						S8		
5									S10					
6						r3				S11				
7									S12					
8				9		r4					S13			
9													S15	
10														
11														
12														
13					14									
14						r5								
15						r6								

Analizador Semántico: En esta última fase, el compilador valida que el significado del código sea correcto. Aunque un programa pueda estar sintácticamente bien, el analizador semántico revisa que las operaciones y declaraciones sean lógicas y tengan sentido, por ejemplo, comprobando que las variables se utilicen correctamente y que las operaciones entre ellas sean válidas de acuerdo con sus tipos.

```

218     /// Leemos programa de entrada
219     vector<string> palabras;
220     string programa, palabra;
221     getline(cin, programa);
222
223     // Dividimos la cadena en palabras
224     stringstream ss(programa); // Utilizamos stringstream para dividir la cadena en palabras
225

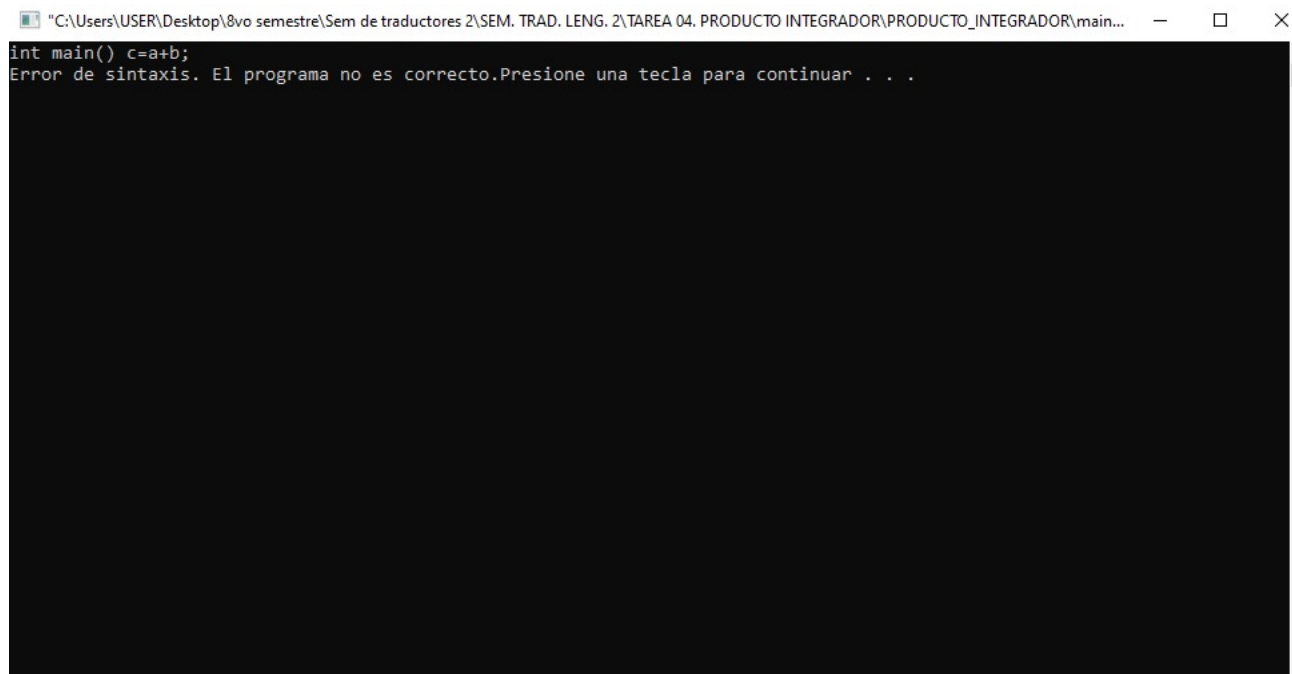
```

En estas fases se asegura, mediante el análisis léxico y sintáctico, que la semántica del programa (o sea, su significado) sea correcta.

A continuación, probaremos el funcionamiento de nuestro analizador semántico con el siguiente fragmento de código en C:

```
int main() { c = a + b; }
```

Después de ejecutar el programa, obtenemos el siguiente resultado:

A screenshot of a Windows command prompt window. The title bar shows the file path: "C:\Users\USER\Desktop\8vo semestre\Sem de traductores 2\SEM. TRAD. LENG. 2\TAREA 04. PRODUCTO INTEGRADOR\PRODUCTO_INTEGRADOR\main...". The command prompt shows the code `int main() c=a+b;` on the first line. On the second line, it displays the error message: `Error de sintaxis. El programa no es correcto.Presione una tecla para continuar . . .`. The rest of the window is black.

El programa ingresado es semánticamente válido. No obstante, cualquier cambio en el código que no siga la sintaxis definida anteriormente resultará en un error sintáctico y, por ende, en un error semántico también.

Esto demuestra, de manera resumida, el correcto funcionamiento del análisis sintáctico del programa.

Conclusión

Este proyecto nos permitió poder implementar el conjunto de las practicas que realizamos anteriormente, los conocimientos adquiridos sobre el proceso de compilación al desarrollar un compilador en lenguaje C que abarca las tres etapas clave: análisis léxico, sintáctico y semántico. A través de este trabajo, logramos implementar un sistema que puede leer, interpretar y validar código en C, abordando aspectos fundamentales de la construcción de un compilador.

El análisis léxico nos permitió reconocer los elementos básicos del lenguaje, clasificándolos para que el código sea procesado en etapas posteriores. En el análisis sintáctico, validamos la estructura del código de acuerdo con reglas gramaticales, asegurándonos de que siga la estructura del lenguaje. Finalmente, el análisis semántico fue esencial para verificar la coherencia y lógica del programa, comprobando que las operaciones y el uso de variables tengan sentido en el contexto del lenguaje C.

Bibliografía

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