



Sintaxis y semántica de los lenguajes

Curso: K2054

PROFESOR: Ing. Pablo Mendez
ASISTE LOS DÍAS: Jueves
EN EL TURNO: Noche
TRABAJO PRÁCTICO N°: 2
TÍTULO: Intérprete de lenguaje micro

INTEGRANTES		
Dorr Pedro	152.845-2	dorrpei@gmail.com
Gioria Gonzalo Luis	149.850-2	gonzalolgioria@gmail.com
Gola Pablo Andres	119.878-6	pabloandresgola@gmail.com
Villavicencio Sergio	157.157-6	sergionicolasvillavicencio@gmail.com

	FECHAS	FIRMA Y ACLARACIÓN DEL DOCENTE
Fecha de entrega N°1	08/11/18	
Fecha de entrega N°2	22/11/18	

INDICACIONES PARA LAS CORRECCIONES:

Índice

Objetivo.....	3
Lenguaje Micro.....	3
Sintaxis de micro.....	3
Gramática léxica:.....	3
Gramática sintáctica:.....	4
Estructura del compilador.....	4
Análisis léxico.....	4
Formas de implementar un scanner.....	4
Análisis sintáctico.....	4
Formas de realizar un análisis sintáctico.....	4
Análisis semántico.....	4
Esquema de trabajo en el equipo.....	5
ANEXO I: LEX (micro.l).....	6
ANEXO II: BISON (micro.y).....	8
ANEXO III: Testing.....	13

Objetivo

Realizar un intérprete del lenguaje MICRO.

Para ello se debe utilizar FLEX y BISON (cualquier producto que lo implemente) y lenguaje C. La entrada de las sentencias debe ser por entrada estándar o desde un archivo que el usuario pueda seleccionar. El programa debe compilar, es decir deben reducirse las expresiones. Además, se deben leer y sacar por pantalla los valores correspondientes cuando las sentencias lo requieran.

Lenguaje Micro

Es un lenguaje teórico desarrollado por Charles Fischer. Es un lenguaje muy simple diseñado específicamente para poseer un LP concreto en el cual se pueda analizar la construcción de un compilador básico. Este lenguaje cuenta con:

- Un único tipo de dato entero.
- Todos sus identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres.
- Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos.
- Las constantes son secuencias de dígitos.
- Las sentencias pueden ser:
 - Asignación:
 - **ID:=Expresión;**
 - Expresión es infija y se construye con identificadores, constantes y los operadores + y -; los paréntesis están permitidos.
 - Entrada/Salida:
 - **leer (Lista de IDs);**
 - **escribir(lista de Expresiones);**
- Cada sentencia termina con un “punto y coma” (;). El cuerpo de un programada está delimitado por inicio y fin.
- **inicio, fin, leer y escribir** son palabras reservadas y deben escribirse en minúscula.

Sintaxis de micro

Gramática léxica:

```
<token> -> uno de <identificador> <constante> <palabraReservada>  
                <operadorAditivo> <asignación> <carácterPuntuación>  
<identificador> -> <letra> {<letra o dígito>}  
<constante> -> <dígito> {<dígito>}  
<letra o dígito> -> uno de <letra> <dígito>  
<letra> -> una de a-z A-Z  
<dígito> -> uno de 0-9  
<palabraReservada> -> una de inicio fin leer escribir  
<operadorAditivo> -> uno de + -  
<asignación> -> :=  
<carácterPuntuación> -> uno de ( ) , ;
```

Gramática sintáctica:

```
<programa> -> inicio <listaSentencias> fin
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> <identificador> := <expresión> ; |
               leer ( <listaIdentificadores> ) ; |
               escribir ( <listaExpresiones> ) ;
<listaIdentificadores> -> <identificador> {, <identificador>}
<listaExpresiones> -> <expresión> {, <expresión>}
<expresión> -> <primaria> {<operadorAditivo> <primaria>}
<primaria> -> <identificador> | <constante> | ( <expresión> )
```

Estructura del compilador

Análisis léxico

El análisis léxico lo realizaremos con un módulo Sanner (Por recomendación del profesor utilizamos win_flex).

El analizador lee, uno a uno, los caracteres que forman un programa fuente, y produce una secuencia de representaciones de tokens. El scanner es una rutina que produce y retorna la representación del correspondiente token, una por vez, en la medida que es invocada por el parser.

Formas de implementar un scanner

- A través de la utilización de un programa auxiliar tipo lex, en el que los datos son tokens representados mediante expresiones regulares.
- Mediante la construcción de una rutina basada en el diseño de un apropiado AFD.

Para este trabajo, utilizaremos la segunda forma.

Análisis sintáctico

El análisis sintáctico lo realizaremos con un módulo Parser(Por recomendación del profesor utilizamos win_bison).

Este analizador procesa los tokens que le entrega el scanner hasta que reconoce una construcción sintáctica que requiere un procesamiento semántico. Entonces, invoca directamente a la rutina semántica que corresponde.

Formas de realizar un análisis sintáctico

- Descendente (Top-Down) que permite ser construido por un programador.
- Ascendente (Bottom-Up) que requiere auxilio de un programa especializado tipo yacc.

Para este trabajo, utilizaremos la forma descendente.

Análisis semántico

El análisis semántico consiste en la realización de tareas de verificación y de conversión que no se pueden realizar en el análisis sintáctico.

Esquema de trabajo en el equipo

Basados en la experiencia y habilidades de cada uno de los integrantes del equipo, y sin perder de vista el plazo de entrega del presente trabajo práctico, la división de tareas inicial y consensuada entre los miembros del equipo fue la siguiente:

Los miembros del equipo que tuvieran más facilidad para la programación en C, se dividirían las tareas de programación, mientras que los restantes integrantes avanzarían con el armado del informe.

Previo al inicio de la codificación, fueron necesarias varias reuniones (por medios electrónicos) para poner en común conceptos y visiones acerca de cómo llevar a cabo el trabajo. Una vez aclarados los puntos necesarios para comenzar con las actividades, dimos inicio a las mismas.

Se creó un repositorio en github para que todos los integrantes pudieran realizar colaboraciones de forma remota, y tener un adecuado control de versiones del código.

Todos los integrantes participaron al momento de tomar decisiones, aportando sus conocimientos e impresiones. De la misma manera, todos participaron de la etapa de testing del software terminado, reportando incidencias y resultados positivos.

ANEXO I: LEX (micro.l)

En este archivo se describen un conjunto de expresiones regulares. Con esto flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones.

Los ficheros de entrada de flex tienen el siguiente formato:

```
%%
```

```
patrón1 {acción1}
```

```
patrón2 {acción2}
```

Donde un patrón es una expresión regular y la acción es un código en c que permita ejecutar la acción necesaria.

Flex se encarga de recorrer la entrada hasta encontrar una concordancia y ahí ejecuta el código asociado.

El fichero de entrada de Flex está compuesto de tres secciones, separadas por una línea donde aparece únicamente un '%%' en esta:

definiciones

```
%%
```

reglas

```
%%
```

código de usuario

Código fuente micro.l:

```
%{
```

```
    #include "micro.tab.h"
```

```
    #include <stdlib.h>
```

```
    #include <stdio.h>
```

```
%}
```

```
%option noyywrap
```

```
espacio      [ \t]+
```

```
digito       [0-9]
```

```
constante    {digito}+
```

```
identificador [a-zA-Z][0-9a-zA-Z]*
```

```
/* Estas son las definiciones de los tokens con expresiones regulares donde + es la clausura positiva y * es la clausura de kleene. */
```

```
%%
/*A partir de acá arrancan las reglas donde según la ER leída, se ejecutará la acción asociada.*/
{espacio}      { /* Ignoramos espacios en blanco */ }

"leer"         return(LEER);
"escribir"     return(ESCRIBIR);
"inicio"       return(INICIO);
"fin"          return(FIN);
{identificador} {
                    strcpy(yylval.valorString,yytext);
                    return(ID);
                }
{constante}    {
                    yylval.valor = atoi(yytext);
                    return(CONSTANTE);
                }
":="          return(ASIGNACION);
"+"           return(SUMA);
"-"           return(RESTA);
","           return(COMA);
"("           return(PARENIZQUIERDO);
")"           return(PARENDERECHO);
";"           return(PUNTOYCOMA);

%%
```

ANEXO II: BISON (micro.y)

Este es el archivo fuente de bison que describe la gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esta gramática.

La forma general de una gramática de bison es la siguiente:

```
%{  
  declaraciones en C  
}%  
Declaraciones de Bison  
%%  
Reglas gramaticales  
%%  
Código C adicional
```

Código fuente micro.y:

```
%{  
    #include <stdio.h>  
    #include <math.h>  
    #include <stdlib.h>  
}%  
/* Se incluyen estas tres librerías ya que stdio.h es la biblioteca estándar del lenguaje C para  
operaciones, math.h para operaciones matemáticas y stdlib.h para la gestión de memoria  
dinámica. */  
  
%union{  
    int valor;  
    char valorString[32];  
}  
/* Union, contiene los atributos de los tipos de dato que nuestros tokens van a reconocer. */  
  
%token <valor> CONSTANTE  
%token <valorString> SENTENCIA  
%token COMA  
%token LEER  
%token ESCRIBIR  
%token INICIO  
%token <valorString> ID  
%token SUMA  RESTA  
%token PARENIZQUIERDO PARENDERECHO  
%token PUNTOYCOMA  
%token FIN  
%token FDT
```



```
/* Acá definimos todos los tokens del lenguaje micro con su tipo de dato en caso de ser
necesario. */

%left SUMA RESTA
%left ASIGNACION
/* Estas son las precedencias de cada uno de los tokens, en el lenguaje micro solo debemos
especificar los de suma, resta y asignación. */

%type <valor> expresion
%type <valor> primaria
/* Acá especificamos los types y sus tipos de dato. */

%start programa // Programa será nuestro axioma.
%%
/* A partir de acá hacemos la definición en BNF de todos los elementos terminales. */

programa: INICIO listaSentencias FIN { imprimirVariables(); }

listaSentencias: sentencia
                | sentencia listaSentencias
                ;

sentencia: ID ASIGNACION expresion PUNTOYCOMA { escribirVariable($1,$3); }
          | LEER PARENIZQUIERDO listaIdentificadores PARENDERECHO PUNTOYCOMA
          | ESCRIBIR PARENIZQUIERDO listaExpresiones PARENDERECHO PUNTOYCOMA
          { printf("\n"); }
          ;

listaIdentificadores: ID { escanearVariable($1); }
                    | listaIdentificadores COMA ID { escanearVariable($3); }
                    ;

listaExpresiones: expresion { printf("%d", $1); }
                | listaExpresiones COMA expresion { printf(" %d ", $3); }
                ;

expresion: primaria { $$=$1; }
          | primaria SUMA expresion { $$=$1+$3; }
          | expresion RESTA primaria { $$=$1-$3; }
          ;

primaria: ID { $$ = leerVariable($1); /* Aca lee los id */ }
          | CONSTANTE { $$=$1; }
          | PARENIZQUIERDO expresion PARENDERECHO { $$=$2; }
          ;
*/
%%
```

```
#define MAX_VARIABLES 30
#define MAX_NOMBRE 32
#define NO_ENCONTRADO 55
struct{
    char nombre[MAX_NOMBRE];
    int valor;
} variables[MAX_VARIABLES];
/* Se define la estructura que almacenará el nombre de las variables que se incorporen en el
programa en lenguaje micro. Se establecen los siguientes máximos:
- cantidad de variables distintas: 32 variables
- largo máximo del identificador: 32 caracteres
*/

extern FILE *yyin; // Se agrega para que pueda enviarse un puntero a un archivo y ser
parseado. En caso de no especificarse, esperará ingreso por entrada estándar (teclado).

int yyerror(char *s) {
    printf("Error: no se reconoce el programa.\n");
} // Acá, especificamos la salida ante un error.

void escanearVariable(char *id) {
    int valorVariable;
    printf("\nIngrese un valor para %s: ", id);
    scanf("%d", &valorVariable);
    escribirVariable(id, valorVariable);
} // Función auxiliar para ingresar una variable por entrada estándar.

void modVariable(char *Name, int Val){
    int x;
    x = buscarVariable(Name);
    if (x != NO_ENCONTRADO) {
        variables[x].valor = Val;
    }
} // Función auxiliar para modificar el valor de una variable.

int buscarVariable(char *Name){
    for(int i=0; i<MAX_VARIABLES; i++){
        if (strcmp(Name, variables[i].nombre) == 0) {
            return i;
        }
    }
    return NO_ENCONTRADO;
} // Función auxiliar para buscar una variable. Si la encuentra, devuelve la posición en el struct.
Si no la encuentra, un valor centinela definido como constante. */

void escribirVariable(char *Name, int Val)
```

```
{
    int x;
    int i;
    i = buscarVariable(Name);
    if (i==NO_ENCONTRADO) {
        x = buscarVariable("");
        if (x != NO_ENCONTRADO) {
            strcpy(variables[x].nombre,Name);
            variables[x].valor=Val;
        }
    }
    if(i!=NO_ENCONTRADO) {
        modVariable(Name,Val);
    }
}
//Función auxiliar para escribir una variable, si la variable existe, la sobrescribe, sinó busca el
primer espacio vacío y la guarda en el espacio correspondiente.
```

```
int leerVariable (char *Name)
{
    int x;
    x = buscarVariable(Name);
    return variables[x].valor;
} //Función auxiliar para leer una variable. Se le proporciona el nombre, y retorna su valor.
```

```
void imprimirVariables(void)
{
    int i;
    printf("\nVARIABLES\n");
    printf("NOMBRE | Valor\n");

    for(i=0; i < MAX_VARIABLES; i++) {
        if (strlen(variables[i].nombre)!= 0)
            printf("%s | %d\n", variables[i].nombre, variables[i].valor);
    }
}
//Función auxiliar para mostrar por pantalla el valor de todas las variables.
```

```
int main(int argc, char *argv[])
{
    FILE *punteroArchivo;
    if (argc == 2) {
        // viene una ruta por parámetro
        punteroArchivo = fopen(argv[1],"r");
        yyin = fopen(argv[1],"r");
        yyparse();
        fclose(punteroArchivo);
    }
    else {
```

```
        yyparse();  
    }  
} //En el main, llamaremos al yyparse, tanto para una ruta por parámetro, como por defecto.
```

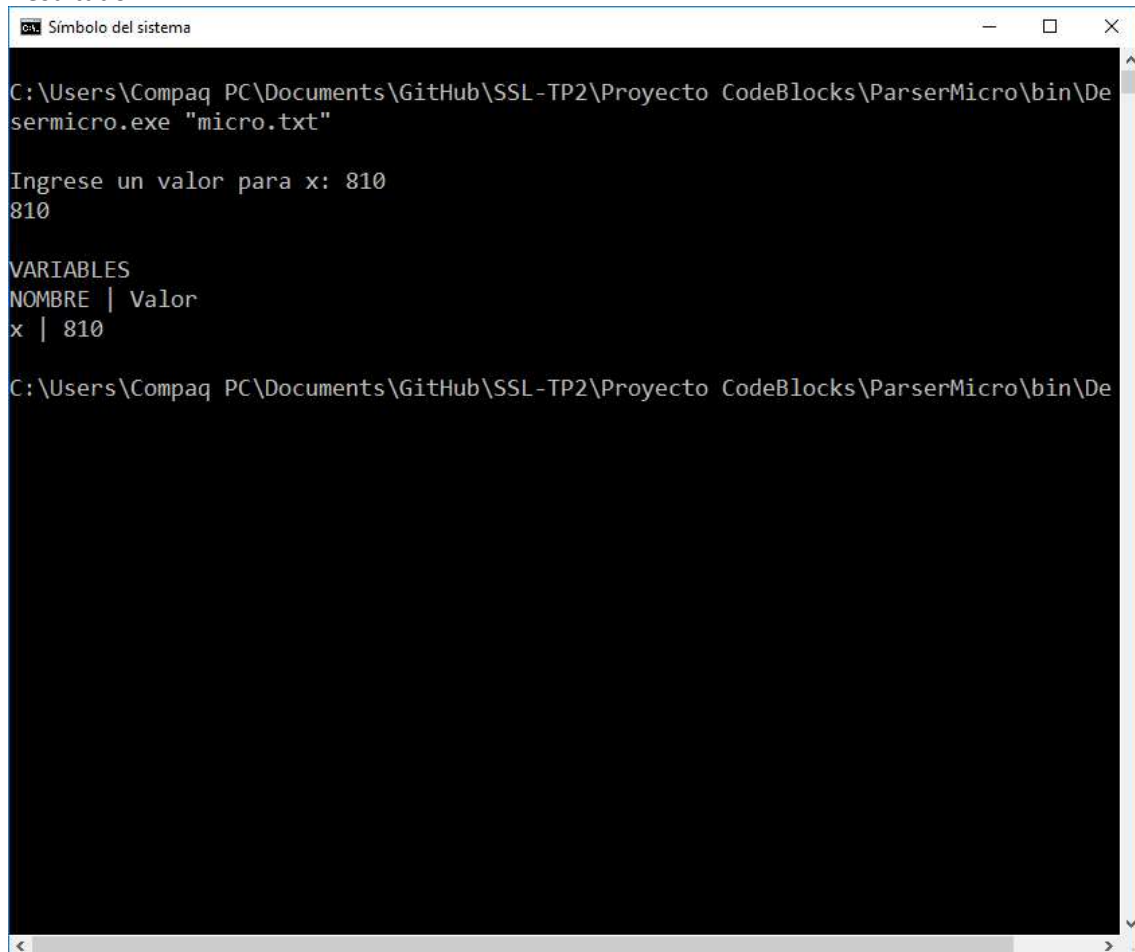
ANEXO III: Testing

A continuación se exponen los casos de prueba de nuestro software, así como el resultado esperado y evidencia del resultado obtenido.

Prueba 1: Ingreso por consola y consulta de un valor

Código	Valor a ingresar	Valor a mostrar
inicio		
leer(x);	x = 810	
escribir(x);		810
fin		

Resultado:



```
Símbolo del sistema

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMicro\bin\Desermicro.exe "micro.txt"

Ingrese un valor para x: 810
810

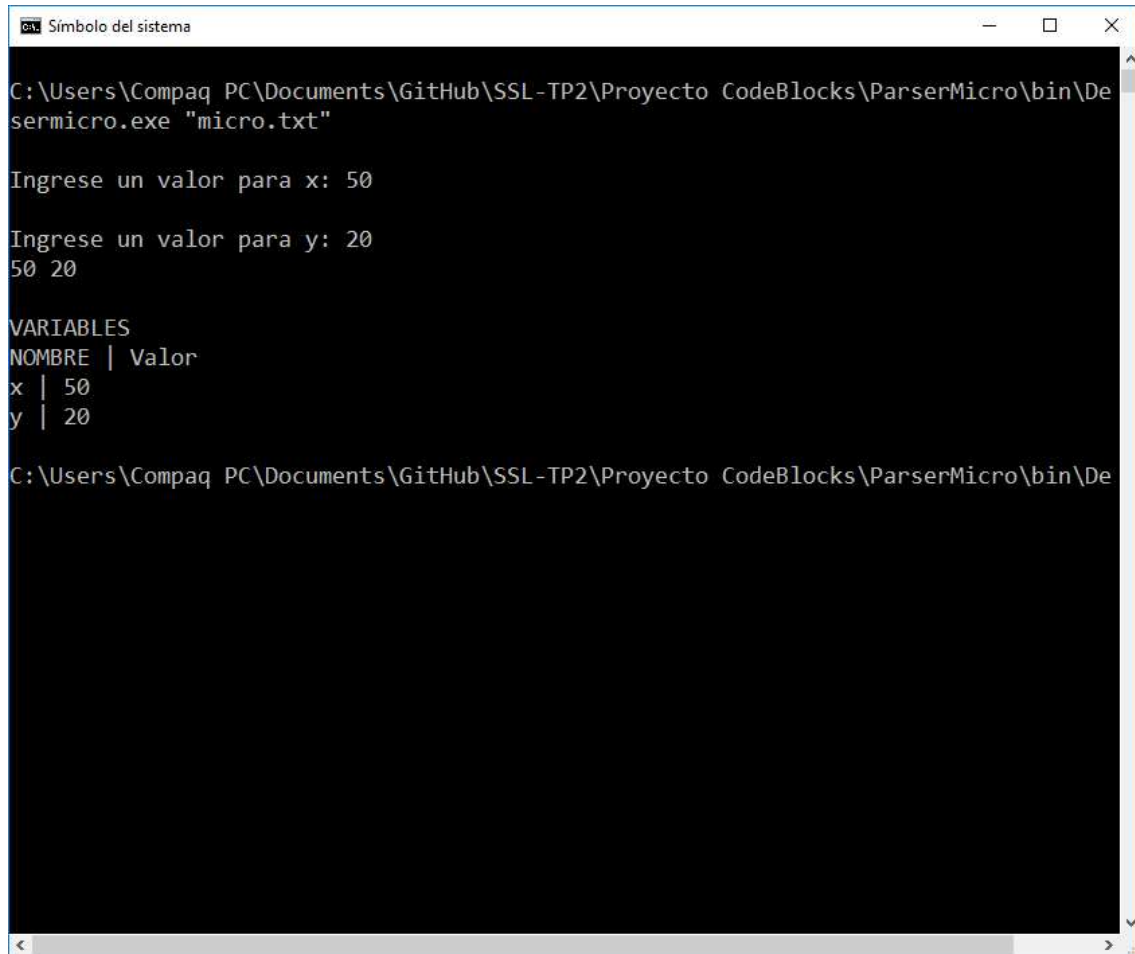
VARIABLES
NOMBRE | Valor
x | 810

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMicro\bin\De
```

Prueba 2: Ingreso por consola y consulta de dos valores

Código	Valor a ingresar	Valor a mostrar
inicio		
leer(x, y);	y = 50, y = 20	
escribir(x, y);		50, 20
fin		

Resultado:



```
Símbolo del sistema
C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMicro\bin\De
sermicro.exe "micro.txt"

Ingrese un valor para x: 50

Ingrese un valor para y: 20
50 20

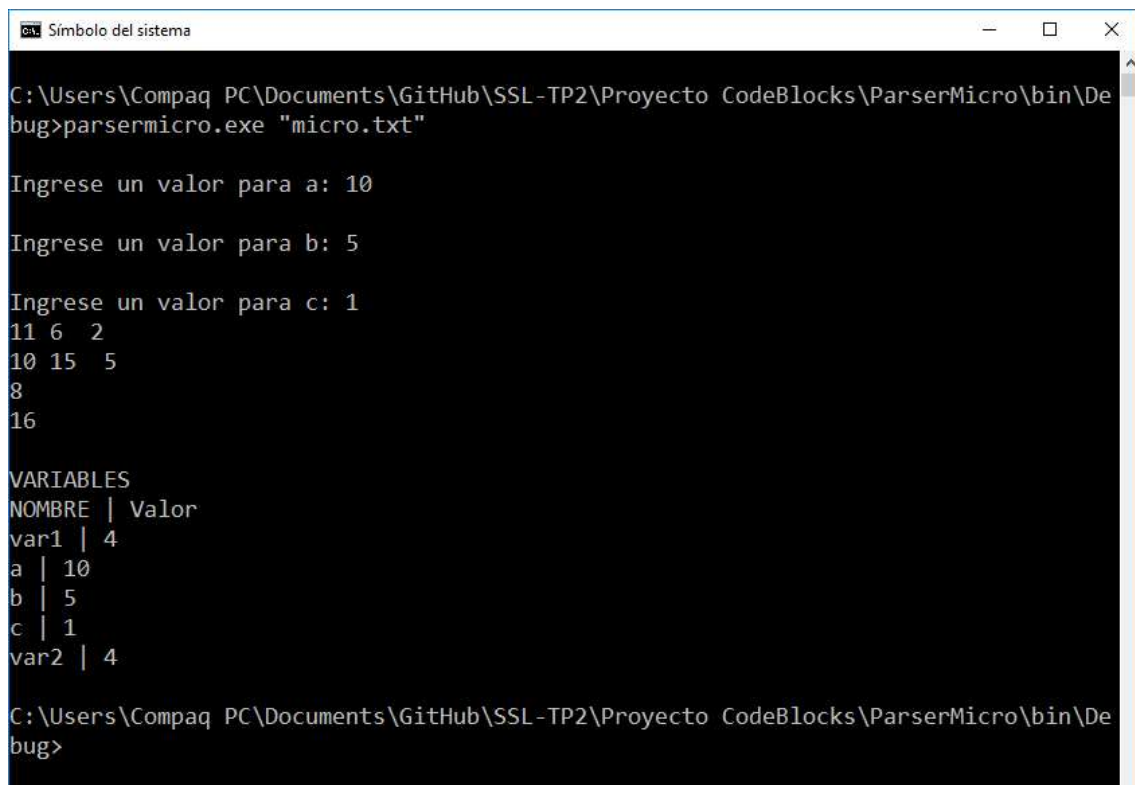
VARIABLES
NOMBRE | Valor
x | 50
y | 20

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMicro\bin\De
```

Prueba 3: Prueba completa

Código	Valor a ingresar	Valor a mostrar
inicio		
var1:=10;		
leer(a,b,c);	a=10, b=5, c=1	
var2:=5;		
escribir(a+1,1+b,c+1);		11, 6, 2
escribir(var1,var1+var2,var1-var2);		10, 15, 5
var1:=4;		
var2:=4;		
escribir(var1+var2);		8
escribir(2+2+1+4+7);		16
fin		

Resultado:



```

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMicro\bin\De
bug>parsermicro.exe "micro.txt"

Ingrese un valor para a: 10

Ingrese un valor para b: 5

Ingrese un valor para c: 1
11 6 2
10 15 5
8
16

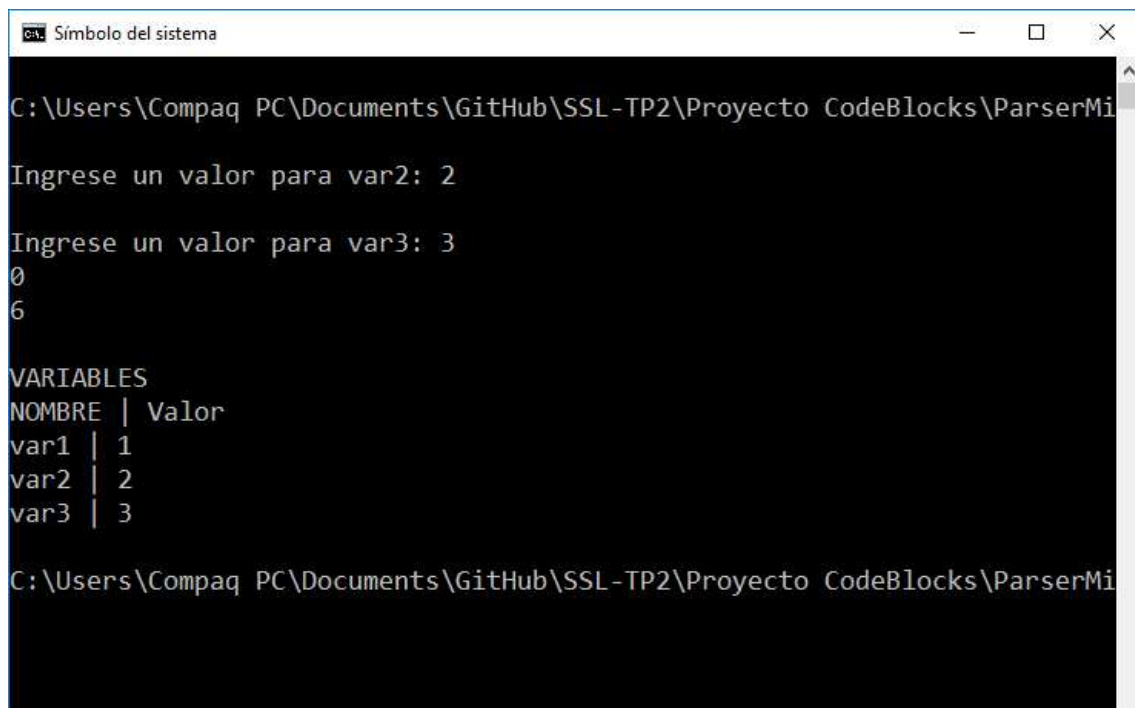
VARIABLES
NOMBRE | Valor
var1 | 4
a | 10
b | 5
c | 1
var2 | 4

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMicro\bin\De
bug>
  
```

Prueba 4: Asignación, lectura y operación con varias variables

Código	Valor a ingresar	Valor a mostrar
inicio		
var1:=5-4;		
leer(var2,var3);	var2 = 2, var3 =3	
escribir(var3-var2-var1);		0
escribir(var3+var2+var1);		6
fin		

Resultado:



```

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMi
Ingrese un valor para var2: 2
Ingrese un valor para var3: 3
0
6

VARIABLES
NOMBRE | Valor
var1   | 1
var2   | 2
var3   | 3

C:\Users\Compaq PC\Documents\GitHub\SSL-TP2\Proyecto CodeBlocks\ParserMi
  
```