

Centro Universitario de Ciencias Exactas e Ingenierías

Compiladores

### **Hands-on 4: Implementación de Analizadores Semánticos**

**Carrera:** Ingeniería en computación.

**Alumno:** Fabián Joheshua Escalante Fernández

**Materia:** Compiladores

**Calendario:** 2025A

**Fecha:** 10/05/2024

## Hands-on 4: Implementación de Analizadores Semánticos

### **Índice:**

#### Introducción

- 1.1. Objetivo del Hands-on
- 1.2. Descripción de los archivos
- 1.3. Flujo de la actividad
- 1.4. Referencias previas (Tech Reading 8)

#### Desarrollo

- 2.1. Construcción de validaciones semánticas
  - 2.1.1. Comprensión del lector
  - 2.1.2. Tabla de símbolos y ámbito
- 2.2. Ejercicios de validaciones semánticas
  - 2.2.1. Ejercicio 1: Declaración y uso correcto de una variable
  - 2.2.2. Ejercicio 2: Uso antes de declaración
  - 2.2.3. Ejercicio 3: Tipos incompatibles en asignación
  - 2.2.4. Ejercicio 4: Retorno incompatible
  - 2.2.5. Ejercicio 5: Ámbitos anidados
  - 2.2.6. Ejercicio 6: Asignación entre arreglos incompatibles
  - 2.2.7. Ejercicio 7: Llamada a función con número incorrecto de argumentos
  - 2.2.8. Ejercicio 8: Redefinición en el mismo ámbito

## Conclusión del lector

### Guía didáctica: Construcción de analizadores semánticos con Flex y Bison

- 4.1. Ejercicio 1: Identificadores no declarados
- 4.2. Ejercicio 2: Verificación de tipos
- 4.3. Ejercicio 3: Verificación de parámetros de funciones
- 4.4. Ejercicio 4: Ámbitos anidados
- 4.5. Ejercicio 5: Validación semántica completa

### Errores y soluciones prácticas

- 5.1. Manejo de syntax error y uso de yyerror
- 5.2. Configuración de %option noyywrap en el scanner
- 5.3. Ajustes en el envío de datos (línea a línea vs. fichero)

## Conclusión

## Bibliografía

## Índice de figuras

Figura 1: Identificadores no declarados: scanner.....	7
Figura 2: Identificadores no declarados: parser.....	8
Figura 3: Identificadores no declarados: ejecución enviando txt.....	9
Figura 4: Identificadores no declarados: ejecución correcta.....	9
Figura 5: Verificación de tipos: scanner correcto.....	10
Figura 6: Verificación de tipos: scanner con fallo.....	10
Figura 7: Verificación de tipos: ejecución scanner con fallo.....	11
Figura 8: Verificación de tipos: parser parte 1.....	11
Figura 9: Verificación de tipos: parser parte 2.....	12
Figura 10: Verificación de tipos: ejecución con txt.....	12
Figura 11: Verificación de tipos: ejecución correcta.....	13
Figura 12: Verificación de tipos: error de salida esperada.....	13
Figura 13: Verificación de tipos: verificación profesor.....	13
Figura 14: Verificación de parámetros de funciones: scanner.....	14
Figura 15: Verificación de parámetros de funciones: parser incorrecto.....	15
Figura 16: Verificación de parámetros de funciones: parser correcto parte 1.....	16
Figura 17: Verificación de parámetros de funciones: parser correcto parte 2.....	17
Figura 18: Verificación de parámetros de funciones: Ejecución correcta.....	18
Figura 19: Ámbitos anidados: scanner.....	19
Figura 20: Ámbitos anidados: error compartido parser.....	19
Figura 21: Ámbitos anidados: parser correcto parte 1.....	20
Figura 22: Ámbitos anidados: parser correcto parte 2.....	21
Figura 23: Ámbitos anidados: ejecución errónea.....	22
Figura 24: Ámbitos anidados: ejecución correcta.....	22
Figura 25: Validación semántica completa en un lenguaje sencillo: scanner.....	23
Figura 26: Validación semántica completa en un lenguaje sencillo: parser correcto parte 1.....	25
Figura 27: Validación semántica completa en un lenguaje sencillo: parser correcto parte 2.....	26
Figura 28: Validación semántica completa en un lenguaje sencillo: parser correcto parte 3.....	27
Figura 29: Validación semántica completa en un lenguaje sencillo: parser correcto parte 4.....	28
Figura 30: Validación semántica completa en un lenguaje sencillo: ejecución correcta.....	29

### **Introducción:**

Vamos a ver en este Hands-on los fundamentos de la construcción de validaciones semánticas mediante ejemplos. Se van a checar dos archivos, analizador-semantico-i y analizador-semantico-ii, primero vamos a comprender el archivo de analizador-semantico-i y luego vamos a ejecutar en IDE el analizador-semantico-ii. Vamos a empezar con la construcción de validaciones semánticas y luego con la construcción de Analizadores Semánticos con Flex y Bison. Se tiene que denotar que algunos ejemplos en los documentos tienen errores, lo voy a comentar debajo de la imagen en el IDE.

Antes de hacer esta actividad cheque el Tech Reading 8 y en resumen se habla de reglas de producción, análisis semántico y tablas de símbolos para que el código fuente no sea solo correcto de forma sintáctica sino lógica también. La gramática y estructura del AST se hacen con las reglas de producción y sus acciones semánticas asociadas se actualizan y consultan en la tabla de símbolos, mientras que el análisis semántico usa ambas estructuras para detectar fallos antes de que se genere el código. El AST es el que transporta la información.

**Desarrollo:**

## **CONSTRUCCIÓN DE VALIDACIONES SEMÁNTICAS**

### **Comprensión del lector**

Las reglas de producción definen como se pueden combinar tokens y no terminales, también pueden tener acciones en C que validan declaraciones y expresiones. La tabla de símbolos guarda metadatos de los identificadores como su ámbito, categoría, tipo, etcétera, esta información se usa durante el análisis. El análisis semántico recorre el AST y usa la tabla de símbolos junto a funciones de consultar para agarrar fallos lógicos. Este documento tiene que ver bastante con el que vimos en el Tech Reading.

### **Ejercicios de la Contrucción de Validaciones Semánticas**

Ahora bien, viendo los ejercicios en código, tienen varios tipos de chequeos y análisis semánticos, los cuales voy a denotar a continuación:

#### **Ejercicio 1: Declaración y uso correcto de una variable**

Aquí ejecutamos la acción semántica de agregar un tipo int que al reconocer la regla para registrar el identificador en la tabla de símbolos usa un buscador para validar su uso. Osea estamos viendo la construcción y consulta de la tabla en el análisis semántico.

#### **Ejercicio 2: Uso antes de declaración**

Aquí el `if (!buscar($1))` detecta referencias a variables con nombres no declarados, así vemos que el análisis semántico comprueba que las variables si existan antes de usarlas.

#### **Ejercicio 3: Tipos incompatibles en asignación**

Cuando comparamos `tipo_de($1)` contra `tipo_de($3)` lo hacemos para validar la compatibilidad de tipos, también sabemos que esto es type checking en el análisis semántico.

#### **Ejercicio 4: Retorno incompatible**

Aquí sacamos el tipo de una expresión y de una función, así aseguramos que el valor retornado coincida con la función y así podemos ver comprobaciones de tipo sobre el AST.

#### **Ejercicio 5: Ámbitos anidados**

Las llamadas a los ambitos, siendo `nuevo_ambito()` y `cerrar_ambito()` gestionan la pila de ámbitos, detectando accesos fuera de la lista durante el análisis semántico.

#### **Ejercicio 6: Asignación entre arreglos incompatibles**

Tenemos una comprobación de arreglos y esto ayuda a que el código siga siendo modular.

#### **Ejercicio 7: Llamada a función con número incorrecto de argumentos**

Sacamos la paridad y numero de argumentos de \$1 y \$3.

#### **Ejercicio 8: Redefinición en el mismo ámbito**

Usamos la condición de buscar en el ambito actual para impedir que se redeclare algo, para evitar bloat en el código y acciones ilegales.

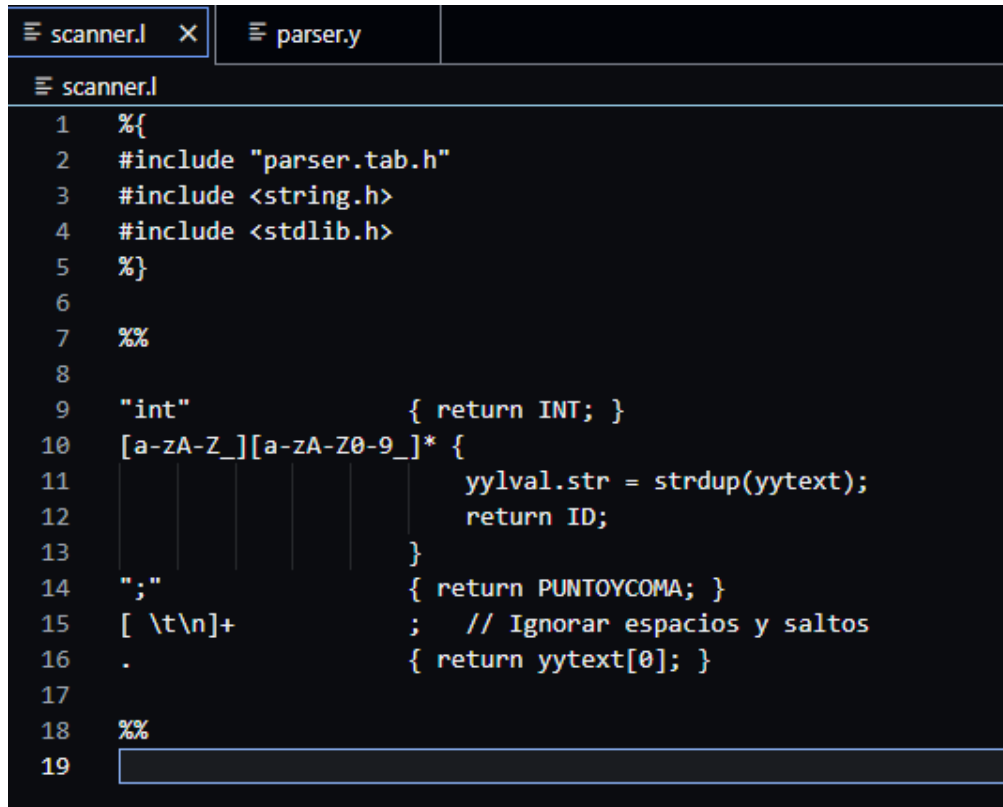
#### **Conclusión del lector**

El lexer al parser produce tokens y aplica reglas de producción para construir el AST, las acciones semánticas interactúan con la tabla de símbolos usando cosas como buscar, tipo\_de, ... mientras que el análisis semántico recorre el AST, usa la tabla y las funciones para validar los tipos, ámbitos, paridad y más cosas de los elementos. Todo esto para tener un código libre de inconsistencias.

## GUÍA DIDÁCTICA: CONSTRUCCIÓN DE ANALIZADORES SEMÁNTICOS CON FLEX Y BISON

### Ejercicio 1: Identificadores no declarados

Este ejercicio valida si los identificadores usados han sido previamente declarados. Si no lo han sido, se reporta un error semántico.



```
1  %{
2  #include "parser.tab.h"
3  #include <string.h>
4  #include <stdlib.h>
5  %}
6
7  %%
8
9  "int" { return INT; }
10 [a-zA-Z][a-zA-Z0-9]* {
11     yylval.str = strdup(yytext);
12     return ID;
13 }
14 ";" { return PUNTOYCOMA; }
15 [ \t\n]+ ; // Ignorar espacios y saltos
16 . { return yytext[0]; }
17
18 %%
19
```

Figura 1: Identificadores no declarados: scanner



```

1  %{
2  #include <stdio.h> // Para entrada/salida estándar
3  #include <stdlib.h> // Para funciones estándar
4  #include <string.h> // Para manejo de cadenas
5  int yylex(void); // Prototipo de función léxica
6  int yyerror(char *s) { printf("Error: %s\n", s); return 0; } // Manejo de erro
7  #define MAX_ID 100 // Tamaño máximo de tabla de símbolos
8  char *tabla[MAX_ID]; // Arreglo para almacenar identificadores
9  int ntabla = 0; // Número actual de identificadores
10 void agregar(char *id) {
11     for (int i = 0; i < ntabla; i++)
12         if (strcmp(tabla[i], id) == 0) return; // No agrega si ya está declara
13     tabla[ntabla++] = strdup(id); // Agrega nuevo identificador
14 }
15 int buscar(char *id) {
16     for (int i = 0; i < ntabla; i++)
17         if (strcmp(tabla[i], id) == 0) return 1; // Retorna 1 si existe
18     return 0; // Retorna 0 si no existe
19 }
20 %}
21 %union { char *str; } // Asociación de tipo para YYSTYPE
22 %token <str> ID // Token ID asociado a cadena
23 %token INT PUNTOYCOMA // Tokens para palabra clave y ;
24 %%
25 programa:
26     declaraciones usos // Un programa son declaraciones
27     ;
28     declaraciones:
29     INT ID PUNTOYCOMA { agregar($2); } // Registra
30     | declaraciones INT ID PUNTOYCOMA { agregar($3); } // Varias decla
31     ;
32     usos:
33     ID PUNTOYCOMA {
34         if (!buscar($1)) // Si el ID no fue declarado
35             printf("Error semántico: '%s' no está declarado\n", $1);
36     }
37     | usos ID PUNTOYCOMA {
38         if (!buscar($2)) // Verifica cada uso posterior
39             printf("Error semántico: '%s' no está declarado\n", $2);
40     }
41     ;
42     %%
43 int main() { return yyparse(); } // Función principal que inicia
44

```

Figura 2: Identificadores no declarados: parser

## Hands-on 4: Implementación de Analizadores Semánticos

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ bison -d parser.y
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ flex scanner.l
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ gcc -o verificador_ids parser.tab.c lex.yy.c -lfl
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_ids < entrada1.txt
Error: syntax error
```

Figura 3: Identificadores no declarados: ejecución enviando txt

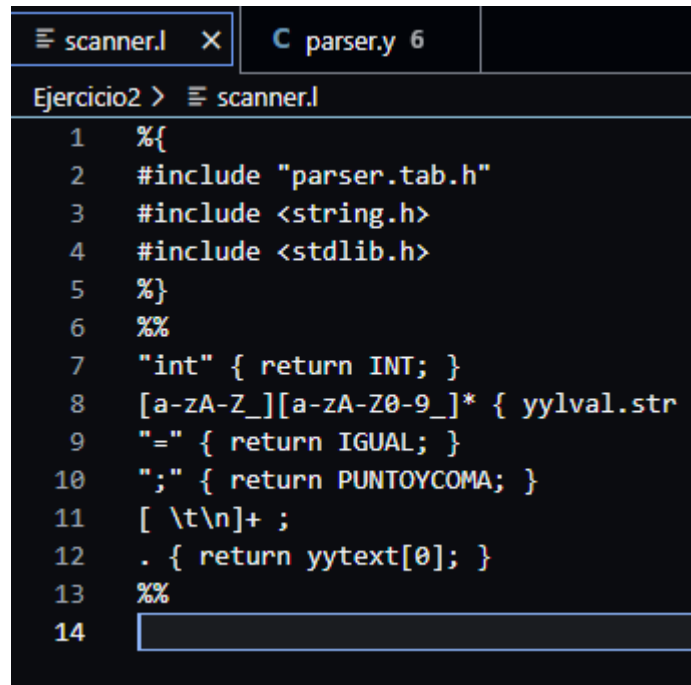
Como vemos cuando envió los datos mediante txt me da un fallo, este lo intente resolver implementando un catch para el parser del error y que entrara bien en el código pero no funciono, también en el scanner le puse al inicio %option noyywrap para ver si así no usaba el origen del error, que pude determinar que era la función yyerror. Busque documentación y es inexistente hasta donde pude buscar, por lo que la solución sencilla que encontré fue introducir los datos uno por uno adentro del verificador.

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_ids
int a;
a;
b;
Error semántico: 'b' no está declarado
```

Figura 4: Identificadores no declarados: ejecución correcta

**Ejercicio 2: Verificación de tipos**

Valida que las asignaciones entre identificadores ocurran entre variables del mismo tipo.



```

1  %{
2  #include "parser.tab.h"
3  #include <string.h>
4  #include <stdlib.h>
5  %}
6  %%
7  "int" { return INT; }
8  [a-zA-Z_][a-zA-Z0-9_]* { yylval.str =
9  "=" { return IGUAL; }
10 ";" { return PUNTOYCOMA; }
11 [ \t\n]+ ;
12 . { return yytext[0]; }
13 %%
14

```

Figura 5: Verificación de tipos: scanner correcto

Encontre un error en el PDF, el scanner tiene mal configurada la línea 11, la resolución se ve aquí es cambiarlo por `[ \t\n]` cuya función es ignorar saltos de línea y tabulación, en el pdf solo sale como `[]`.

Figura 6: Verificación de tipos: scanner con fallo

**scanner.l**

```

%{
#include "parser.tab.h"           // Inclusión d
#include <string.h>               // Para funcio
#include <stdlib.h>               // Para funcio
%}
%%
"int" { return INT; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytex
"=" { return IGUAL; }
";" { return PUNTOYCOMA; }
[
]+ ;
. { return yytext[0]; }
%%

```

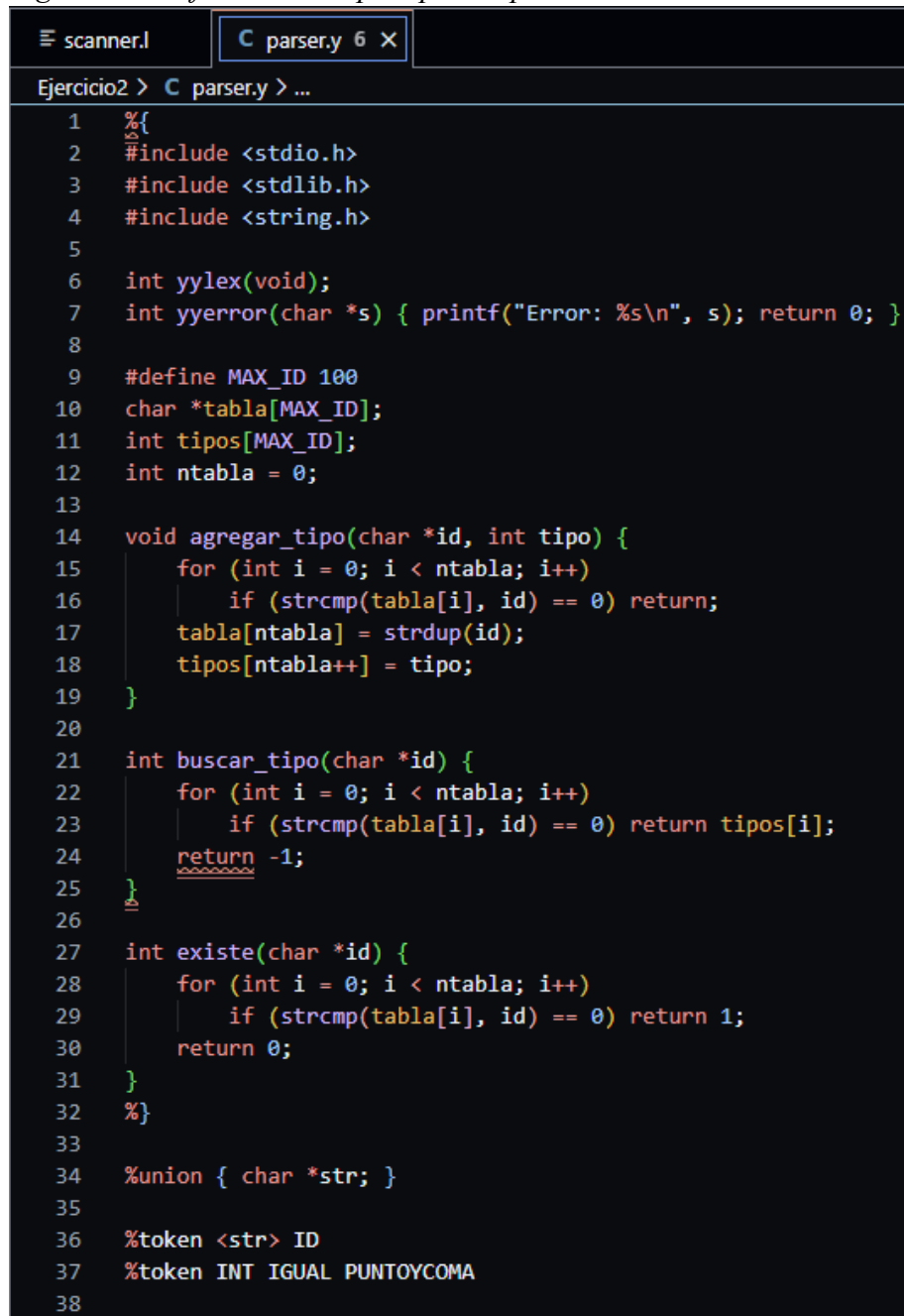
#### Hands-on 4: Implementación de Analizadores Semánticos

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ flex scanner.l
scanner.l:11: bad character class
```

Figura 7: Verificación de tipos: ejecución scanner con fallo

El parser, sin embargo, es correcto.

Figura 8: Verificación de tipos: parser parte 1



```
scanner.l  C parser.y 6 X
Ejercicio2 > C parser.y > ...
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int yylex(void);
7  int yyerror(char *s) { printf("Error: %s\n", s); return 0; }
8
9  #define MAX_ID 100
10 char *tabla[MAX_ID];
11 int tipos[MAX_ID];
12 int ntabla = 0;
13
14 void agregar_tipo(char *id, int tipo) {
15     for (int i = 0; i < ntabla; i++)
16         if (strcmp(tabla[i], id) == 0) return;
17     tabla[ntabla] = strdup(id);
18     tipos[ntabla++] = tipo;
19 }
20
21 int buscar_tipo(char *id) {
22     for (int i = 0; i < ntabla; i++)
23         if (strcmp(tabla[i], id) == 0) return tipos[i];
24     return -1;
25 }
26
27 int existe(char *id) {
28     for (int i = 0; i < ntabla; i++)
29         if (strcmp(tabla[i], id) == 0) return 1;
30     return 0;
31 }
32 %}
33
34 %union { char *str; }
35
36 %token <str> ID
37 %token INT IGUAL PUNTOYCOMA
38
```

```

39  %%
40  programa:
41  |   declaraciones asignaciones
42  ;
43
44  declaraciones:
45  |   INT ID PUNTOYCOMA          { agregar_tipo($2, 0); }
46  |   declaraciones INT ID PUNTOYCOMA { agregar_tipo($3, 0); }
47  ;
48
49  asignaciones:
50  |   ID IGUAL ID PUNTOYCOMA {
51  |       if (!existe($1) || !existe($3))
52  |           printf("Error: identificador no declarado\n");
53  |       else if (buscar_tipo($1) != buscar_tipo($3))
54  |           printf("Error: tipos incompatibles\n");
55  |   }
56  |   asignaciones ID IGUAL ID PUNTOYCOMA {
57  |       if (!existe($2) || !existe($4))
58  |           printf("Error: identificador no declarado\n");
59  |       else if (buscar_tipo($2) != buscar_tipo($4))
60  |           printf("Error: tipos incompatibles\n");
61  |   }
62  ;
63  %%
64  ≡
65  int main() {
66  |   return yyparse();
67  | }
68

```

Figura 9: Verificación de tipos: parser parte 2

```

fabí@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ bison -d parser.y
fabí@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ flex scanner.l
fabí@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ gcc -o verificador_tipos parser.tab.c lex.yy.c -lfl
fabí@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_tipos < entrada2.txt
Error: syntax error

```

Figura 10: Verificación de tipos: ejecución con txt

#### Hands-on 4: Implementación de Analizadores Semánticos

Nuevamente enviar por txt me da error, entonces seguí usando la introducción de datos a mano, se tiene que denotar que el PDF indica la salida de dos resultados, pero también es correcto que salga solo uno.

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_tipos
int a;
int b;
a = b;
a = c;
Error: identificador no declarado
b = a;
```

*Figura 11: Verificación de tipos: ejecución correcta*

Este es el error del PDF

#### Salida esperada:

```
Error: identificador no declarado
Error: identificador no declarado
```

*Figura 12: Verificación de tipos: error de salida esperada*

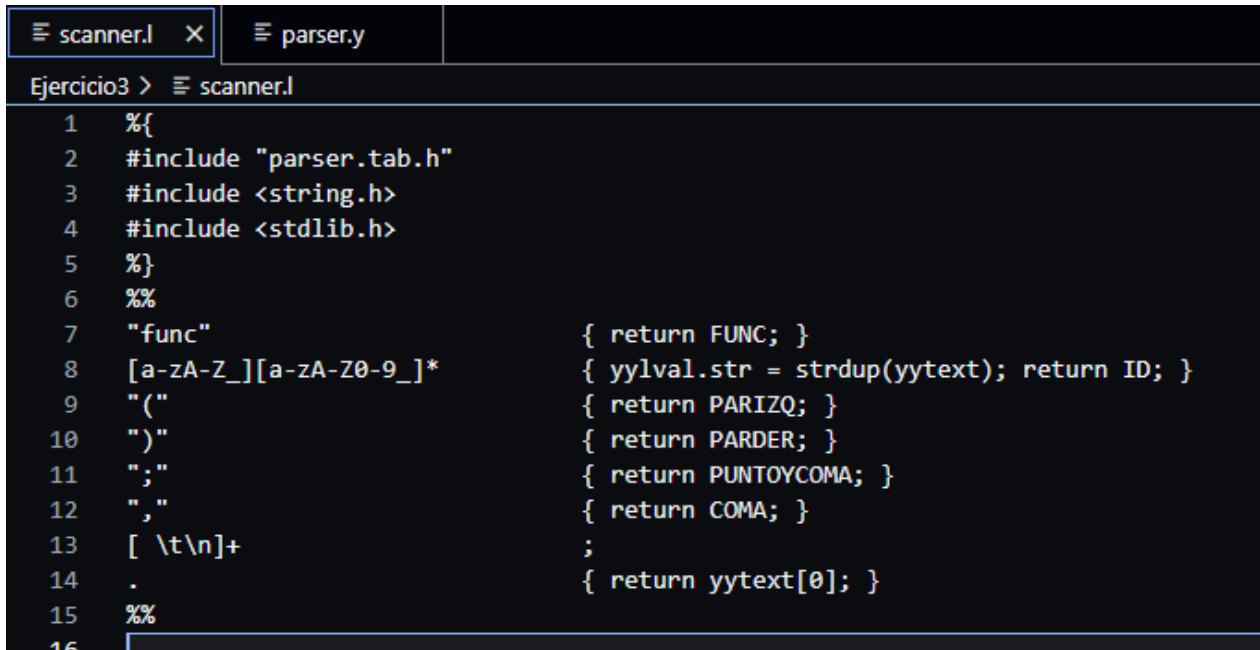
El profesor confirmo en su equipo que solo imprime un fallo

```
analisis-sem — -bash — 114x29
...n/2025-A/udg/compilers/analisis-sem — Vim -S ~/vimrc scanner.l • markdown-preview-macos ~/Documents/Markdo
| Jose Macbook-Air: analisis-sem ipepetron$ ./verificador_tipos < entrada.txt
| Error: identificador no declarado Jose Macbook-Air: analisis-sem ipepetron$
```

*Figura 13: Verificación de tipos: verificación profesor*

**Ejercicio 3: Verificación de parámetros de funciones**

Este ejercicio valida que al invocar una función, el número de argumentos coincida con los parámetros registrados para esa función. Se utiliza una tabla de funciones con nombre y aridad.



```
scanner.l x parser.y
Ejercicio3 > scanner.l
1  %{
2  #include "parser.tab.h"
3  #include <string.h>
4  #include <stdlib.h>
5  %{
6  %%
7  "func"                { return FUNC; }
8  "[a-zA-Z_][a-zA-Z0-9_]*" { yylval.str = strdup(yytext); return ID; }
9  "("                  { return PARIZQ; }
10 ")"                  { return PARDER; }
11 ";"                  { return PUNTOYCOMA; }
12 ","                  { return COMA; }
13 "[ \t\n]+"           ;
14 "."                  { return yytext[0]; }
15 %%
16
```

Figura 14: Verificación de parámetros de funciones: scanner

#### Hands-on 4: Implementación de Analizadores Semánticos

Ahora el parser tiene error en el PDF, esto colaborado con los compañeros de la clase, te da errores en varias líneas, por ejemplo a mi.

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ bison -d parser.y
parser.y:39.31-32: error: $4 of 'declaraciones' has no declared type
 39 |         registrar_funcion($2, $4);
    |                               ^~
parser.y:43.31-32: error: $5 of 'declaraciones' has no declared type
 43 |         registrar_funcion($3, $5);
    |                               ^~
parser.y:48.21-22: error: $$ of 'lista' has no declared type
 48 |         ID { $$ = 1; }
    |           ^~
parser.y:50.21-22: error: $$ of 'lista' has no declared type
 50 |         lista COMA ID { $$ = $1 + 1; }
    |                       ^~
parser.y:50.26-27: error: $1 of 'lista' has no declared type
 50 |         lista COMA ID { $$ = $1 + 1; }
    |                       ^~
parser.y:56.18-19: error: $3 of 'llamadas' has no declared type
 56 |         if (n != $3)
    |                   ^~
parser.y:62.18-19: error: $4 of 'llamadas' has no declared type
 62 |         if (n != $4)
    |                   ^~
parser.y:68.20-21: error: $$ of 'args' has no declared type
 68 |         ID { $$ = 1; }
    |           ^~
parser.y:70.20-21: error: $$ of 'args' has no declared type
 70 |         args COMA ID { $$ = $1 + 1; }
    |                     ^~
parser.y:70.25-26: error: $1 of 'args' has no declared type
```

*Figura 15: Verificación de parámetros de funciones: parser incorrecto*

Entonces para realizar mi parser y solucionar estos errores puse que acepte en cualquier orden y cantidad las declaraciones, eso esta en la línea 36. Incluí para mis errores de syntax error lo que es la llamada a { yyerrok; }, capturando ese error pero continuando. Ya con estos simples cambios puede realizar la ejecución aunque no funcione el parser del PDF. A continuación fotos del parser.



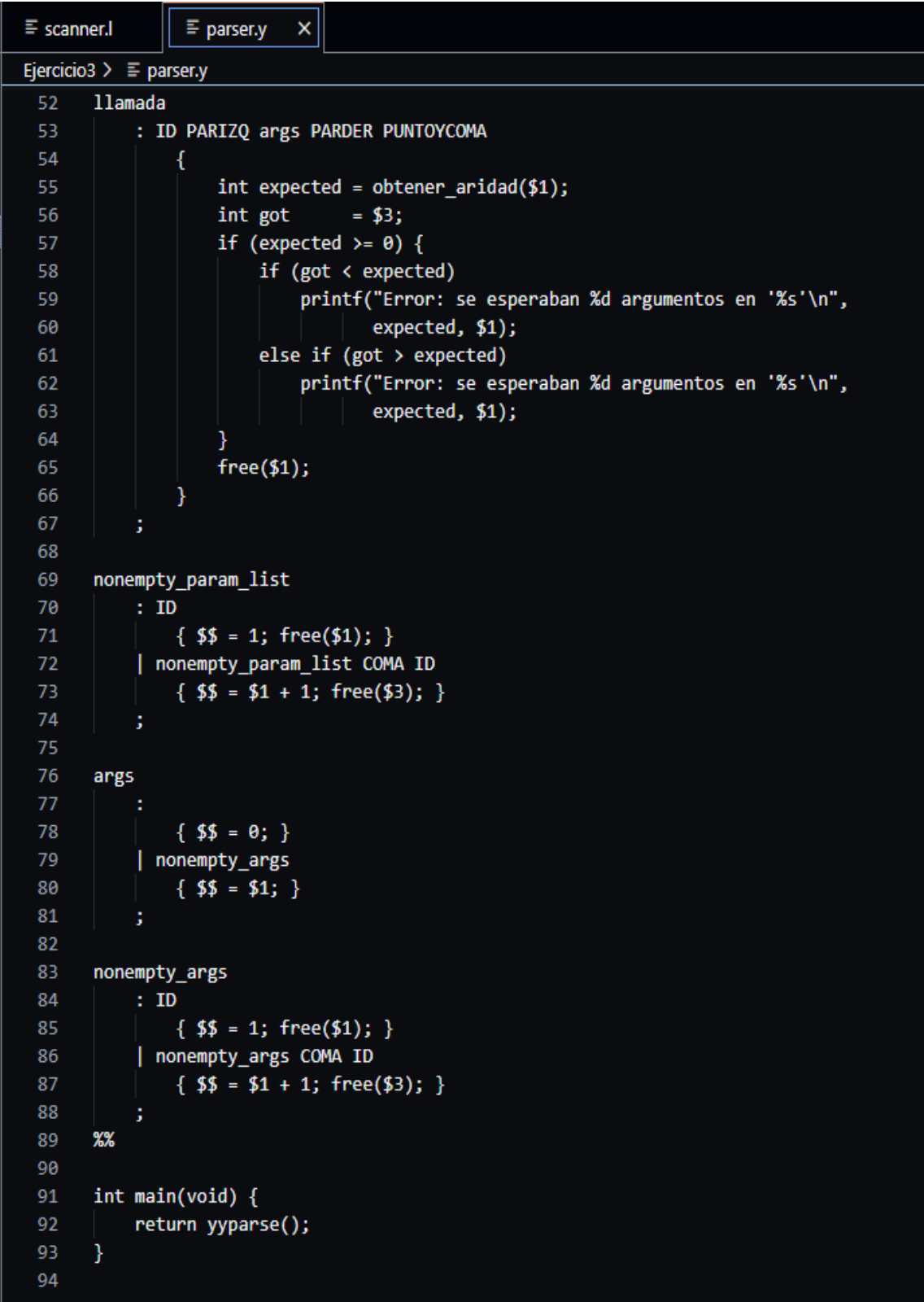
```

1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int yylex(void);
7  int yyerror(char *s) { return 0; }
8
9  #define MAX_FUNC 100
10 static char *funciones[MAX_FUNC];
11 static int  aridades[MAX_FUNC];
12 static int  nfuncs = 0;
13
14 void registrar_funcion(char *id, int n) {
15     funciones[nfuncs] = strdup(id);
16     aridades[nfuncs++] = n;
17 }
18
19 int obtener_aridad(char *id) {
20     for (int i = 0; i < nfuncs; i++)
21         if (strcmp(funciones[i], id) == 0)
22             return aridades[i];
23     return -1;
24 }
25 %}
26
27 %union { char *str; int num; }
28
29 %token <str> ID
30 %token FUNC PARIZQ PARDER PUNTOYCOMA COMA
31
32 %type <num> nonempty_param_list nonempty_args args
33
34 %%
35
36 programa
37 :
38 | programa sentencia
39 ;
40
41 sentencia
42 : declaracion
43 | llamada
44 | error PUNTOYCOMA { yyerrok; }
45 ;
46
47 declaracion
48 : FUNC ID PARIZQ nonempty_param_list PARDER PUNTOYCOMA
49   { registrar_funcion($2, $4); free($2); }
50 ;
51

```

Figura 16: Verificación de parámetros de funciones: parser correcto parte 1

## Hands-on 4: Implementación de Analizadores Semánticos



```
52 llamada
53 : ID PARIZQ args PARDER PUNTOYCOMA
54 {
55     int expected = obtener_aridad($1);
56     int got      = $3;
57     if (expected >= 0) {
58         if (got < expected)
59             printf("Error: se esperaban %d argumentos en '%s'\n",
60                 expected, $1);
61         else if (got > expected)
62             printf("Error: se esperaban %d argumentos en '%s'\n",
63                 expected, $1);
64     }
65     free($1);
66 }
67 ;
68
69 nonempty_param_list
70 : ID
71 { $$ = 1; free($1); }
72 | nonempty_param_list COMA ID
73 { $$ = $1 + 1; free($3); }
74 ;
75
76 args
77 :
78 { $$ = 0; }
79 | nonempty_args
80 { $$ = $1; }
81 ;
82
83 nonempty_args
84 : ID
85 { $$ = 1; free($1); }
86 | nonempty_args COMA ID
87 { $$ = $1 + 1; free($3); }
88 ;
89 %%
90
91 int main(void) {
92     return yyparse();
93 }
94
```

Figura 17: Verificación de parámetros de funciones: parser correcto parte 2

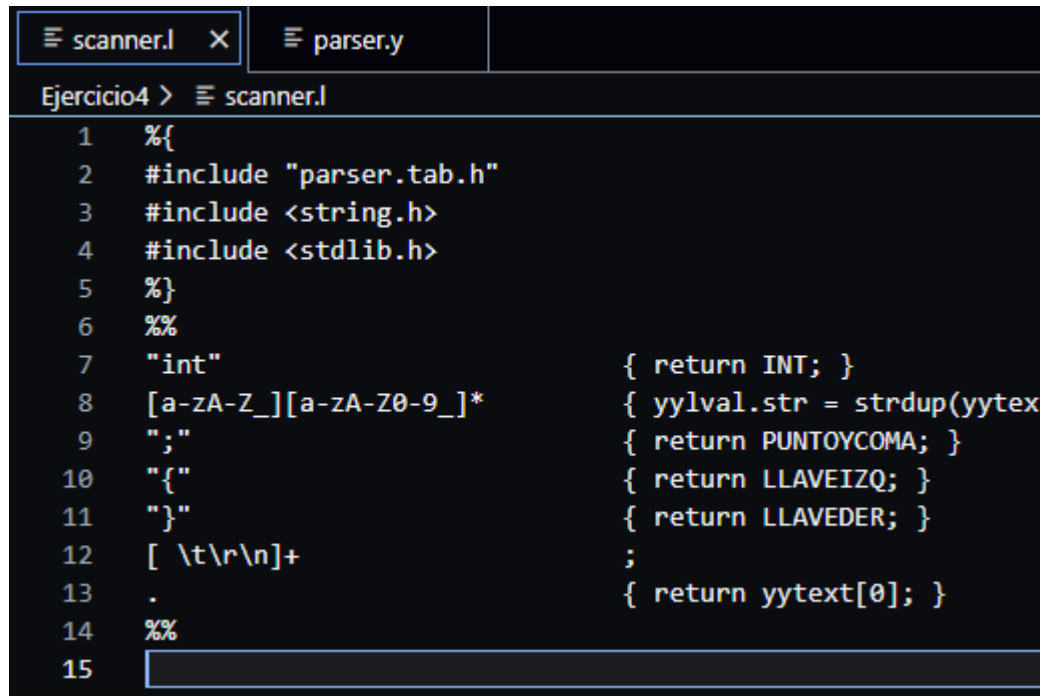
```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_funciones < entrada3.txt  
Error: se esperaban 2 argumentos en 'resta'  
Error: se esperaban 3 argumentos en 'max'  
Error: se esperaban 2 argumentos en 'suma'  
Error: se esperaban 3 argumentos en 'max'
```

*Figura 18: Verificación de parámetros de funciones: Ejecución correcta*

## Hands-on 4: Implementación de Analizadores Semánticos

### Ejercicio 4: Ámbitos anidados

Este ejercicio extiende el análisis semántico para manejar ámbitos anidados con bloques `{}`. Los identificadores se validan en su ámbito local o en los exteriores si es necesario.



```
scanner.l
parser.y

Ejercicio4 > scanner.l
1  %{
2  #include "parser.tab.h"
3  #include <string.h>
4  #include <stdlib.h>
5  %}
6  %%
7  "int"                { return INT; }
8  [a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); }
9  ";"                 { return PUNTOYCOMA; }
10 "{"                 { return LLAVEIZQ; }
11 "}"                 { return LLAVEDER; }
12 [ \t\r\n]+         ;
13 .                   { return yytext[0]; }
14 %%
15
```

Figura 19: Ámbitos anidados: scanner

El parser del PDF tiene un error pequeño, nos genera un syntax error, este error es igualmente porque yyerror esta impidiendo la ejecución correcta del código, simplemente en este caso lo ignoramos, utilizando la herramienta que use en el pasando, siendo yyerrok; para descartar construcciones mal formadas. Adjunto foto de un compañero con el mismo error del parser.

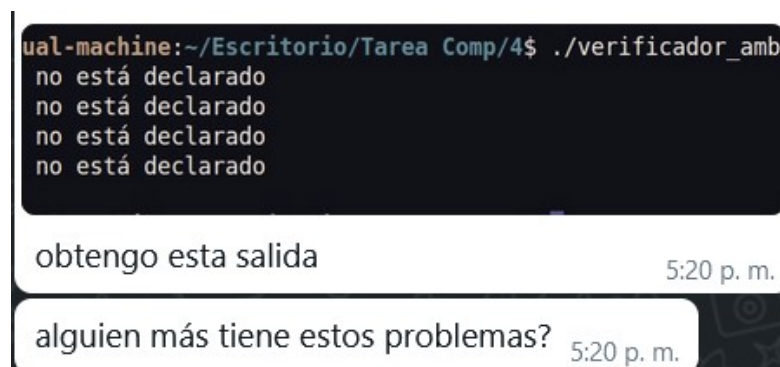


Figura 20: Ámbitos anidados: error compartido parser

```

1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int yylex(void);
7  int yyerror(char *s) { return 0; }
8
9  #define MAX_SCOPE 10
10 #define MAX_ID    100
11
12 static char *ambitos[MAX_SCOPE][MAX_ID];
13 static int  niveles[MAX_SCOPE];
14 static int  tope = 0;
15
16 void entrar_ambito() {
17     tope++;
18     niveles[tope] = 0;
19 }
20 void salir_ambito() {
21     tope--;
22 }
23
24 void agregar_local(char *id) {
25     ambitos[tope][niveles[tope]++] = strdup(id);
26 }
27 int buscar_local(char *id) {
28     for (int i = tope; i >= 0; i--)
29         for (int j = 0; j < niveles[i]; j++)
30             if (strcmp(ambitos[i][j], id) == 0)
31                 return 1;
32     return 0;
33 }
34 %}
35
36 %union { char *str; }
37
38 %token <str> ID
39 %token INT LLAVEIZQ LLAVEDER PUNTOYCOMA
40
41 %%
42
43 programa
44 :
45 | programa sentencia
46 ;
47

```

Figura 21: Ámbitos anidados: parser correcto parte 1

## Hands-on 4: Implementación de Analizadores Semánticos

```
48  sentencia
49      : declaracion
50      | llamada
51      | error PUNTOYCOMA { yyerrok; }
52      | ID
53      {
54          if (!buscar_local($1))
55              printf("Error semántico: '%s' no está declarado\n", $1);
56          free($1);
57          yyerrok;
58      }
59      ;
60
61  declaracion
62      : INT ID PUNTOYCOMA
63      { agregar_local($2); free($2); }
64      | LLAVEIZQ
65      { entrar_ambito(); }
66      instrucciones
67      LLAVEDER
68      { salir_ambito(); }
69      ;
70
71  llamada
72      : ID PUNTOYCOMA
73      {
74          if (!buscar_local($1))
75              printf("Error semántico: '%s' no está declarado\n", $1);
76          free($1);
77      }
78      ;
79
80  instrucciones
81      :
82      | instrucciones instruccion
83      ;
84
85  instruccion
86      : declaracion
87      | llamada
88      ;
89
90  %%
91
92  int main(void) {
93      return yyparse();
94  }
95
```

Figura 22: Ámbitos anidados: parser correcto parte 2

Y aquí la diferencia de ejecución entre los dos parsers. Empezamos con la que usa el parser erróneo y seguimos con la de mi parser.

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_ambitos < entrada4.txt
Error semántico: 'b' no está declarado
Error semántico: 'c' no está declarado
Error semántico: 'c' no está declarado
Error semántico: 'd' no está declarado
Error: syntax error
```

*Figura 23: Ámbitos anidados: ejecución errónea*

La correcta.

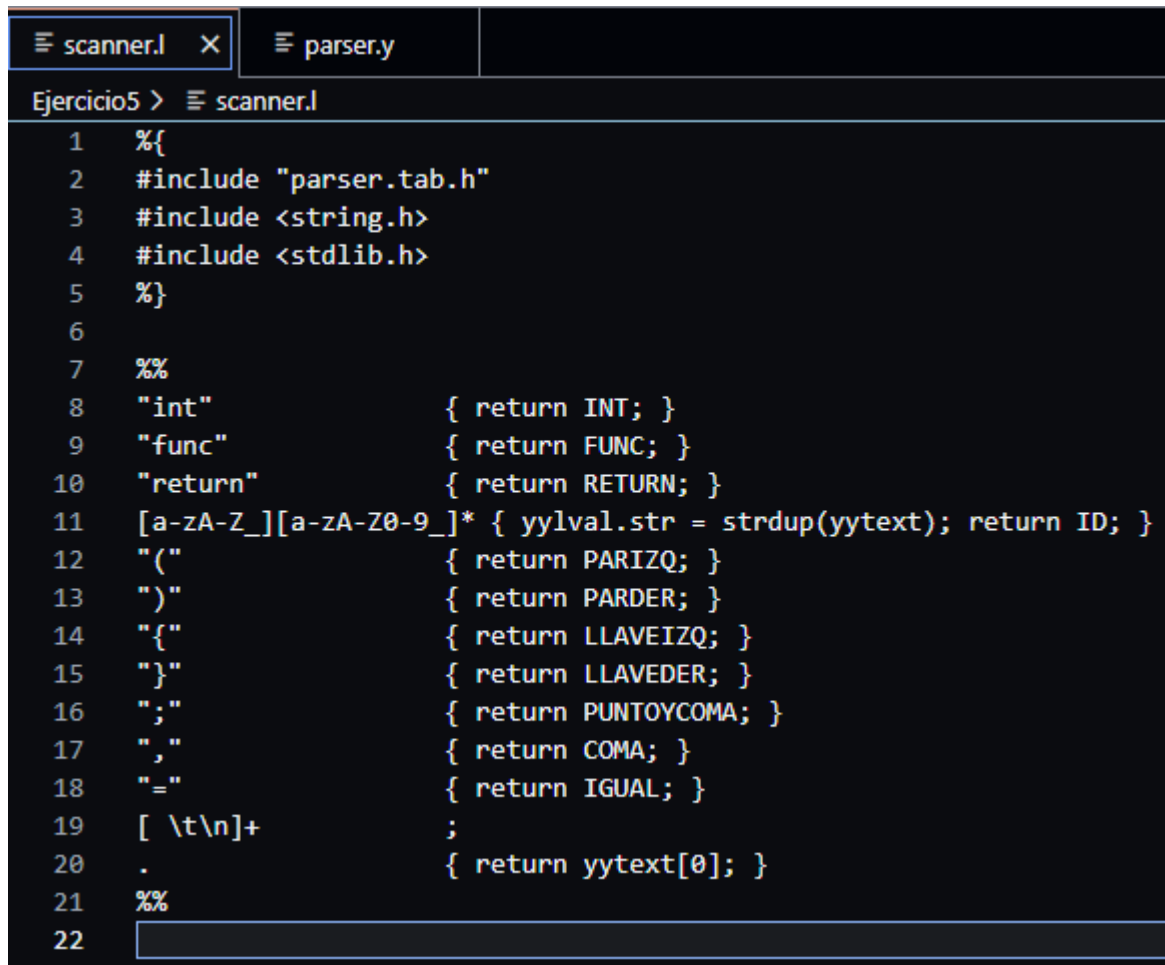
```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_ambitos < entrada4.txt
Error semántico: 'b' no está declarado
Error semántico: 'c' no está declarado
Error semántico: 'c' no está declarado
Error semántico: 'd' no está declarado
Error semántico: 'd' no está declarado
```

*Figura 24: Ámbitos anidados: ejecución correcta*

## Hands-on 4: Implementación de Analizadores Semánticos

### Ejercicio 5: Validación semántica completa en un lenguaje sencillo

Este ejercicio integra validaciones semánticas como: declaración y uso de variables, detección de duplicados, asignaciones con verificación de tipo, declaración y uso de funciones, comprobación de aridad, y manejo de ámbitos anidados, usando una tabla de símbolos extendida con nombre, tipo, ámbito y contexto.



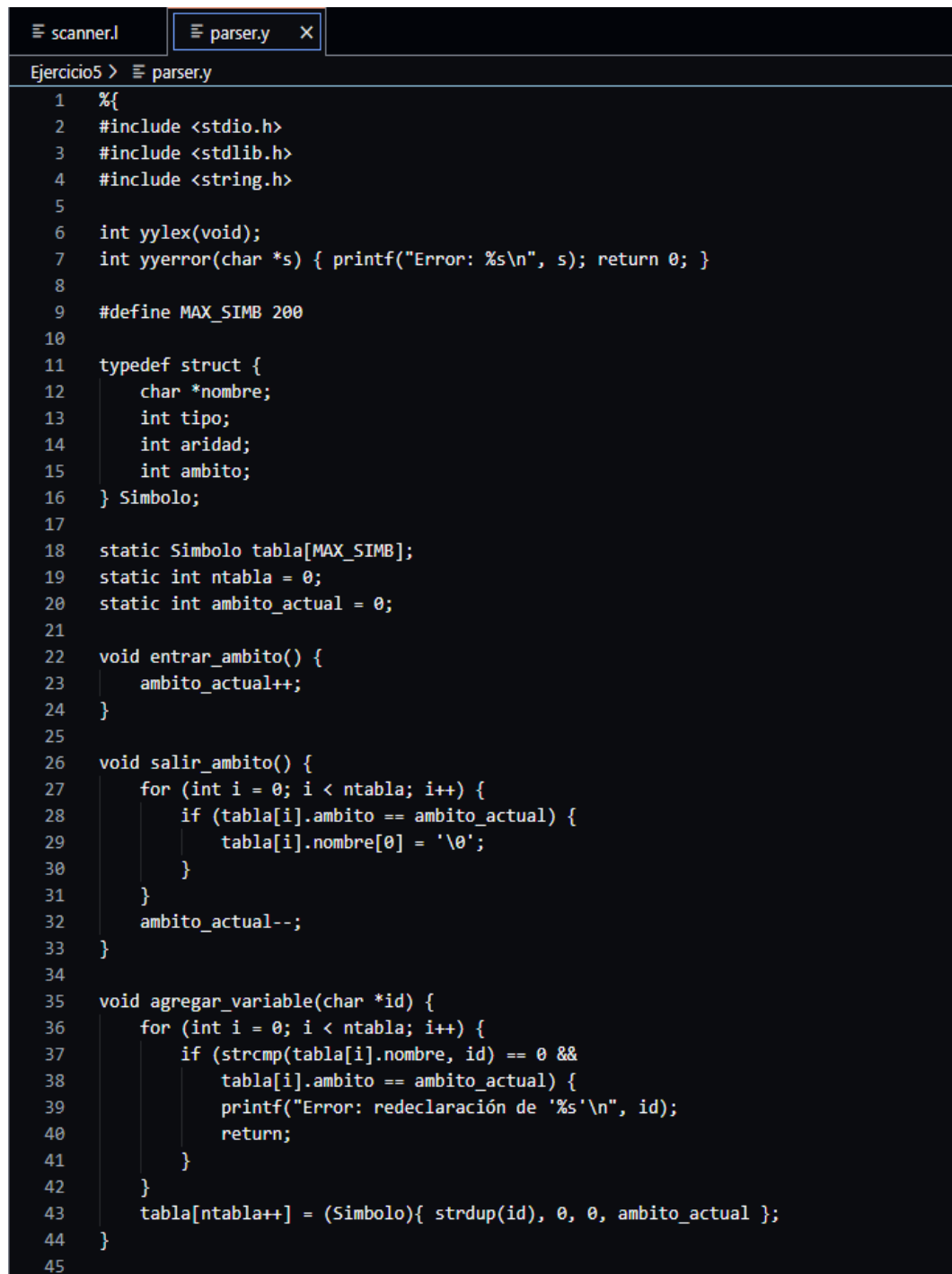
```
1  %{
2  #include "parser.tab.h"
3  #include <string.h>
4  #include <stdlib.h>
5  %}
6
7  %%
8  "int"          { return INT; }
9  "func"         { return FUNC; }
10 "return"       { return RETURN; }
11 "[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
12 "("           { return PARIZQ; }
13 ")"           { return PARDER; }
14 "{"           { return LLAVEIZQ; }
15 "}"           { return LLAVEDER; }
16 ";"           { return PUNTOYCOMA; }
17 ","           { return COMA; }
18 "="           { return IGUAL; }
19 "[ \t\n]+"    ;
20 "."           { return yytext[0]; }
21 %%
22
```

Figura 25: Validación semántica completa en un lenguaje sencillo: scanner



Como los códigos anteriores ahora tuve de nuevo problemas para introducir el txt en el verificador, entonces lo que hice fue aceptar declaraciones de todo tipo en cualquier lugar, algo que ya había hecho anteriormente con el aceptar vacío, solo que ahora también englobo declaraciones e instrucciones.

## Hands-on 4: Implementación de Analizadores Semánticos



```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int yylex(void);
7  int yyerror(char *s) { printf("Error: %s\n", s); return 0; }
8
9  #define MAX_SIMB 200
10
11  typedef struct {
12      char *nombre;
13      int tipo;
14      int aridad;
15      int ambito;
16  } Simbolo;
17
18  static Simbolo tabla[MAX_SIMB];
19  static int ntabla = 0;
20  static int ambito_actual = 0;
21
22  void entrar_ambito() {
23      ambito_actual++;
24  }
25
26  void salir_ambito() {
27      for (int i = 0; i < ntabla; i++) {
28          if (tabla[i].ambito == ambito_actual) {
29              tabla[i].nombre[0] = '\0';
30          }
31      }
32      ambito_actual--;
33  }
34
35  void agregar_variable(char *id) {
36      for (int i = 0; i < ntabla; i++) {
37          if (strcmp(tabla[i].nombre, id) == 0 &&
38              tabla[i].ambito == ambito_actual) {
39              printf("Error: redeclaración de '%s'\n", id);
40              return;
41          }
42      }
43      tabla[ntabla++] = (Simbolo){ strdup(id), 0, 0, ambito_actual };
44  }
45
```

Figura 26: Validación semántica completa en un lenguaje sencillo: parser correcto parte 1

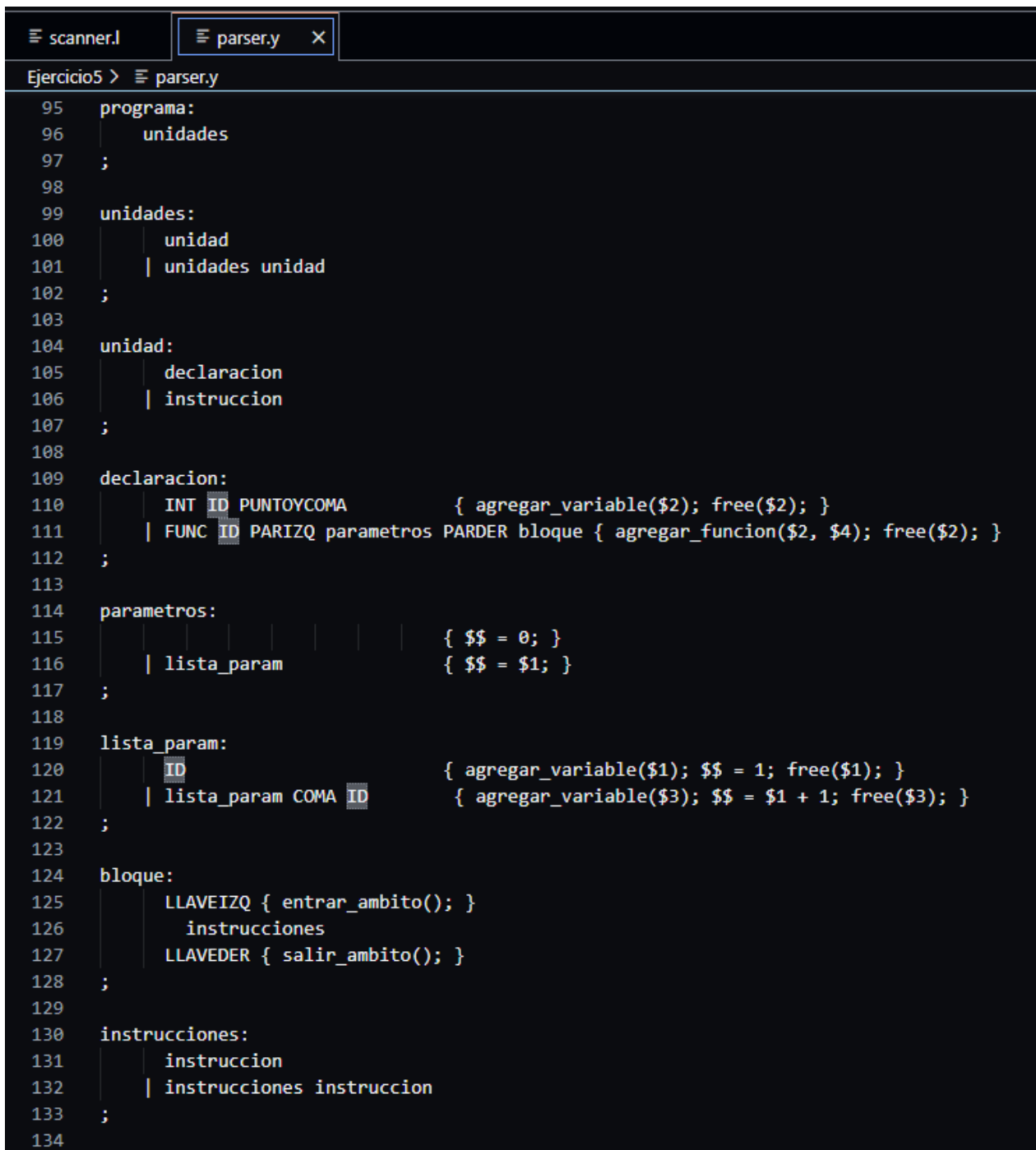
```

46 void agregar_funcion(char *id, int aridad) {
47     for (int i = 0; i < ntabla; i++) {
48         if (strcmp(tabla[i].nombre, id) == 0 &&
49             tabla[i].tipo == 1 &&
50             tabla[i].ambito == 0) {
51             printf("Error: función '%s' ya declarada\n", id);
52             return;
53         }
54     }
55     tabla[ntabla++] = (Simbolo){ strdup(id), 1, aridad, 0 };
56 }
57
58 int buscar_variable(char *id) {
59     for (int nivel = ambito_actual; nivel >= 0; nivel--) {
60         for (int i = 0; i < ntabla; i++) {
61             if (tabla[i].ambito == nivel &&
62                 tabla[i].tipo == 0 &&
63                 strcmp(tabla[i].nombre, id) == 0) {
64                 return 1;
65             }
66         }
67     }
68     return 0;
69 }
70
71 int buscar_funcion(char *id) {
72     for (int i = 0; i < ntabla; i++) {
73         if (tabla[i].tipo == 1 &&
74             strcmp(tabla[i].nombre, id) == 0) {
75             return tabla[i].aridad;
76         }
77     }
78     return -1;
79 }
80 %}
81
82 %union {
83     char *str;
84     int num;
85 }
86
87 %token <str> ID
88 %token INT FUNC RETURN IGUAL
89 %token PARIZQ PARDER LLAVEIZQ LLAVEDER PUNTOYCOMA COMA
90
91 %type <num> parametros lista_param instruccion argumentos lista_args
92
93 %%

```

Figura 27: Validación semántica completa en un lenguaje sencillo: parser correcto parte 2

## Hands-on 4: Implementación de Analizadores Semánticos



```
scanner.l  parser.y x
Ejercicio5 > parser.y
95  programa:
96      unidades
97  ;
98
99  unidades:
100      unidad
101      | unidades unidad
102  ;
103
104  unidad:
105      declaracion
106      | instruccion
107  ;
108
109  declaracion:
110      INT ID PUNTOYCOMA      { agregar_variable($2); free($2); }
111      | FUNC ID PARIZQ parametros PARDER bloque { agregar_funcion($2, $4); free($2); }
112  ;
113
114  parametros:
115      { $$ = 0; }
116      | lista_param          { $$ = $1; }
117  ;
118
119  lista_param:
120      ID                    { agregar_variable($1); $$ = 1; free($1); }
121      | lista_param COMA ID  { agregar_variable($3); $$ = $1 + 1; free($3); }
122  ;
123
124  bloque:
125      LLAVEIZQ { entrar_ambito(); }
126      instrucciones
127      LLAVEDER { salir_ambito(); }
128  ;
129
130  instrucciones:
131      instruccion
132      | instrucciones instruccion
133  ;
134
```

Figura 28: Validación semántica completa en un lenguaje sencillo: parser correcto parte 3

```

135 instrucion:
136     INT ID PUNTOYCOMA      { agregar_variable($2); free($2); }
137     | ID IGUAL ID PUNTOYCOMA {
138         if (!buscar_variable($1) || !buscar_variable($3))
139             printf("Error: identificador no declarado\n");
140         free($1); free($3);
141     }
142     | ID PARIZQ argumentos PARDER PUNTOYCOMA {
143         int esp = buscar_funcion($1);
144         if (esp == -1)
145             printf("Error: función '%s' no declarada\n", $1);
146         else if (esp != $3)
147             printf("Error: función '%s' espera %d argumentos\n", $1, esp);
148         free($1);
149     }
150     | RETURN ID PUNTOYCOMA {
151         if (!buscar_variable($2))
152             printf("Error: identificador no declarado\n");
153         free($2);
154     }
155     | bloque
156 ;
157
158 argumentos:
159     { $$ = 0; }
160     | lista_args { $$ = $1; }
161 ;
162
163 lista_args:
164     ID { if (!buscar_variable($1)) printf("Error: identificador no declarado\n"); $$ = 1; free($1); }
165     | lista_args COMA ID { if (!buscar_variable($3)) printf("Error: identificador no declarado\n"); $$ = $1 + 1; free($3); }
166 ;
167
168 %%
169
170 int main(void) {
171     return yyparse();
172 }
173

```

Figura 29: Validación semántica completa en un lenguaje sencillo: parser correcto parte 4

#### Hands-on 4: Implementación de Analizadores Semánticos

Recordemos que ejecuto línea por línea por el fallo del txt.

```
fabi@Skylon:~/Hands-on 4 Implementación de Analizadores Semánticos$ ./verificador_completo
int x;
x = y;
Error: identificador no declarado
func f(a, b) {
int a;
x = a;
f(a);
Error: función 'f' no declarada
return a;
}
f(x, x);
f(x);
Error: función 'f' espera 2 argumentos
f();
Error: función 'f' espera 2 argumentos
```

*Figura 30: Validación semántica completa en un lenguaje sencillo: ejecución correcta*

### **Conclusión:**

El parser es una línea de defensa para formar un programa y detectar errores lógicos de forma coherente y rápida antes de generar el código o ejecutarlo. Así todo código esta libre de inconsistencias.

### **Bibliografía:**

Avina Méndez, J. A. (2025, 22 de abril; última modificación 7 de mayo). Reglas de Producción en Bison [PDF]. Classroom, Universidad de Guadalajara. Jalisco, México. Recuperado de <https://classroom.google.com/u/1/c/NzQ1NzE3MTA5NzA3/a/NzczMjQxMDk0MjA1/details>

Avina Méndez, J. A. (2025, 22 de abril; última modificación 7 de mayo). Tabla de Símbolos: Fundamentos, Ejemplos Abstractos y Concretos [PDF]. Classroom, Universidad de Guadalajara. Jalisco, México. Recuperado de <https://classroom.google.com/u/1/c/NzQ1NzE3MTA5NzA3/a/NzczMjQxMDk0MjA1/details>

Avina Méndez, J. A. (2025, 22 de abril; última modificación 7 de mayo). Comprender el Análisis Semántico a Través de Preguntas Clave [PDF]. Classroom, Universidad de Guadalajara. Jalisco, México. Recuperado de <https://classroom.google.com/u/1/c/NzQ1NzE3MTA5NzA3/a/NzczMjQxMDk0MjA1/details>

Wikipedia contributors. (2021, 4 diciembre). Semantic analysis (compilers). Recuperado de [https://en.wikipedia.org/wiki/Semantic\\_analysis\\_%28compilers%29](https://en.wikipedia.org/wiki/Semantic_analysis_%28compilers%29)

TutorialsPoint. (2025c, marzo 25). Semantic Analysis in Compiler Design. Recuperado de [https://www.tutorialspoint.com/es/compiler\\_design/compiler\\_design\\_semantic\\_analysis.htm](https://www.tutorialspoint.com/es/compiler_design/compiler_design_semantic_analysis.htm)