UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

COMPILADORES

PROGRAMA NO. 2

PROGRAMA ANÁLISIS SINTÁCTICO

Profesor: Ing. Adrián Ulises Mercado Martínez

Alumnos:

- **Almaguer Rioja Carlos Israel**
- **Ocaña Paredes Kidcia Mireya**
- **Villareal Silva José Antonio**

Grupo No. 2

Semestre 2020 – 2

Fecha De Entrega:
04 – Junio – 2020

a. Análisis del programa

Una vez identificado y desarrollado la parte correspondiente al analizador léxico, se debe trabajar en la implantación y desarrollo del análisis sintáctico. En esta etapa se debe verificar que los atributos o cadenas ingresadas a la entrada corresponden a la gramática dada, para ello se creará un nuevo archivo llamado “parser.y” que contendrá cada una de las reglas gramaticales propuestas para el desarrollo de este proyecto.

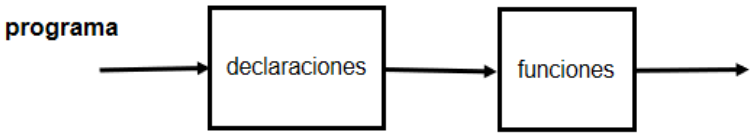
b. Diseño de la solución

Gramática del proyecto:

1. programa → declaraciones funciones

No presenta recursividad o ambigüedad.

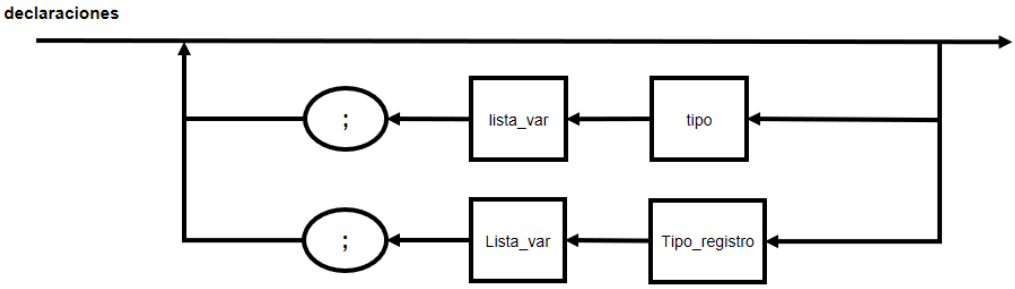
Diagrama de sintaxis:



2. declaraciones → tipo lista_var; declaraciones | tipo_registro lista_va; declaraciones | ε

Presenta recursividad por la derecha. No es ambigua.

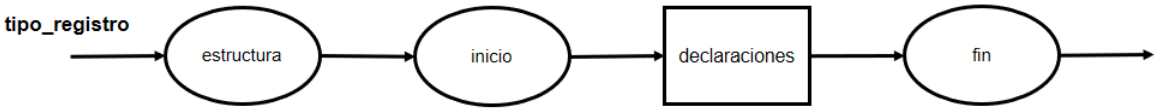
Diagrama de sintaxis:



3. tipo registro → **estructura inicio** declaraciones **fin**

No presenta recursividad o ambigüedad.

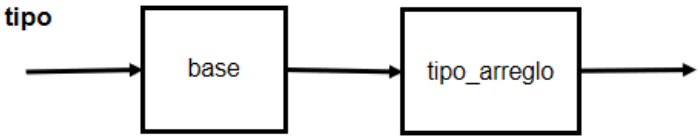
Diagrama de sintaxis:



4. tipo → base tipo arreglo

No presenta recursividad o ambigüedad.

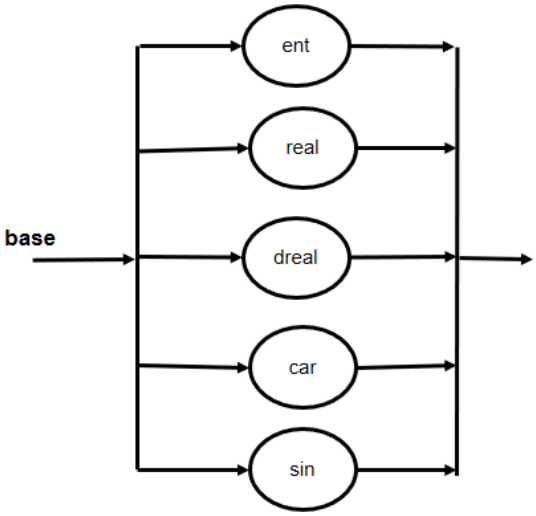
Diagrama de sintaxis:



5. base → **ent** | **real** | **dreal** | **car** | **sin**

No presenta recursividad o ambigüedad.

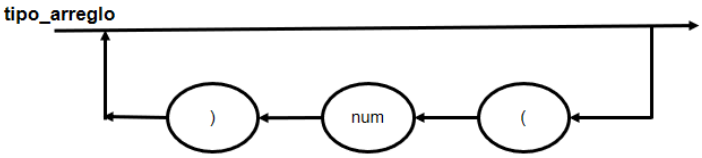
Diagrama de sintaxis:



6. tipo_arreglo → **(num)** tipo arreglo | ε

Presenta recursividad por la derecha. No es ambigua.

Diagrama de sintaxis:



7. lista_var → lista_var**, id** | **id**

Presenta recursividad por la izquierda. No es ambigua.

Eliminando la recursividad por la izquierda:

lista_var → lista_var**, id** | **id**

lista_var → **id** lista_var1

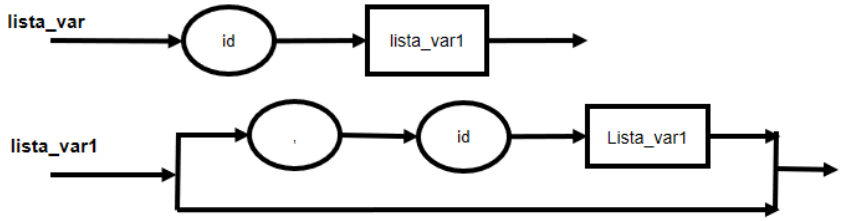
A → A α | β

A → β A'

A' → α A' | ε

lista_var1 → **, id** lista_var1 | ε

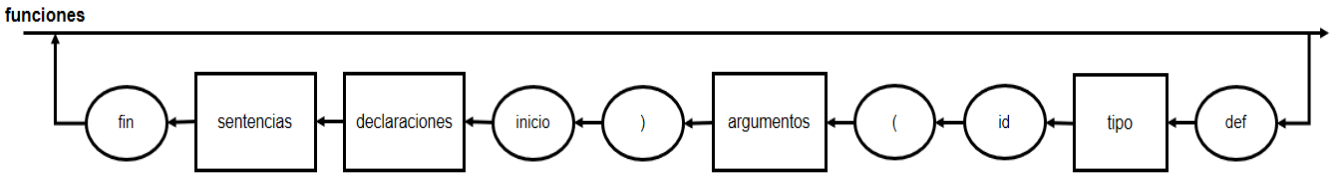
Diagrama de sintaxis:



8. funciones → **def** tipo **id(**argumentos**) inicio** declaraciones sentencias **fin** funciones | ε

Presenta recursividad por la derecha. No es ambigua.

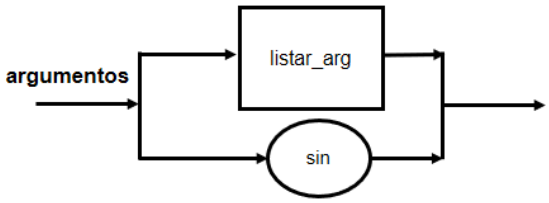
Diagrama de sintaxis:



9. argumentos → listar_arg | **sin**

No presenta recursividad o ambigüedad.

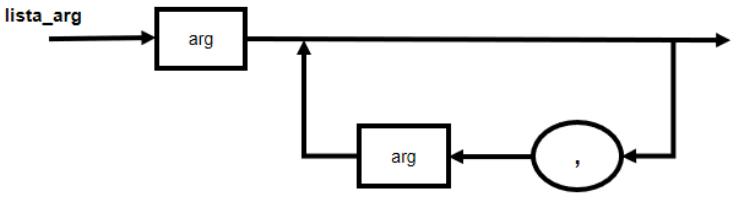
Diagrama de sintaxis:



10. lista_arg → lista_arg, arg | arg

Presenta recursividad por la izquierda. No es ambigua.

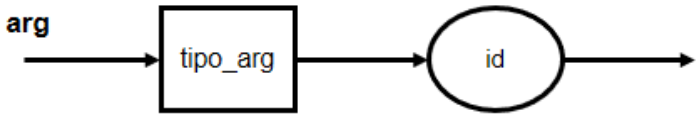
Diagrama de sintaxis:



11. arg → tipo_arg **id**

No presenta recursividad o ambigüedad.

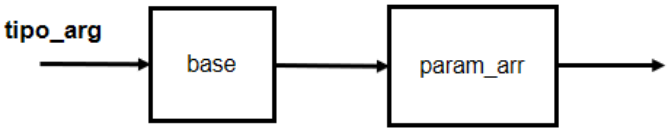
Diagrama de sintaxis:



12. tipo_arg → base param_arr

No presenta recursividad o ambigüedad.

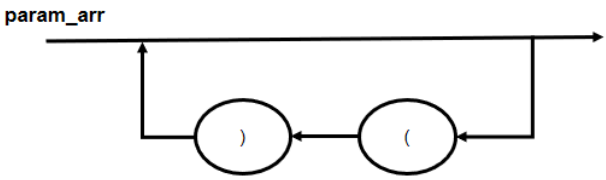
Diagrama de sintaxis:



13. param_arr → **()** param_arr | ε

Presenta recursividad por la derecha. No es ambigua.

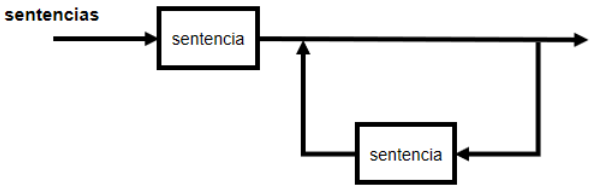
Diagrama de sintaxis:



14. sentencias → sentencias sentencia | sentencia

Presenta recursividad por la izquierda. No es ambigua.

Diagrama de sintaxis:



15. sentencia → **si** e_bool **entonces** sentencia **fin**

| **si** e_bool **entonces** sentencia **sino** sentencia **fin**

| **mientras** e_bool **hacer** sentencia **fin**

| **hacer** sentencia **mientras** e_bool**;**

| **segun (**variable**) hacer** casos predeterminado **fin**

| variable **:=** expresion **;**

| **escribir** expresión **;**

| **leer** variable **;** | **devolver;**

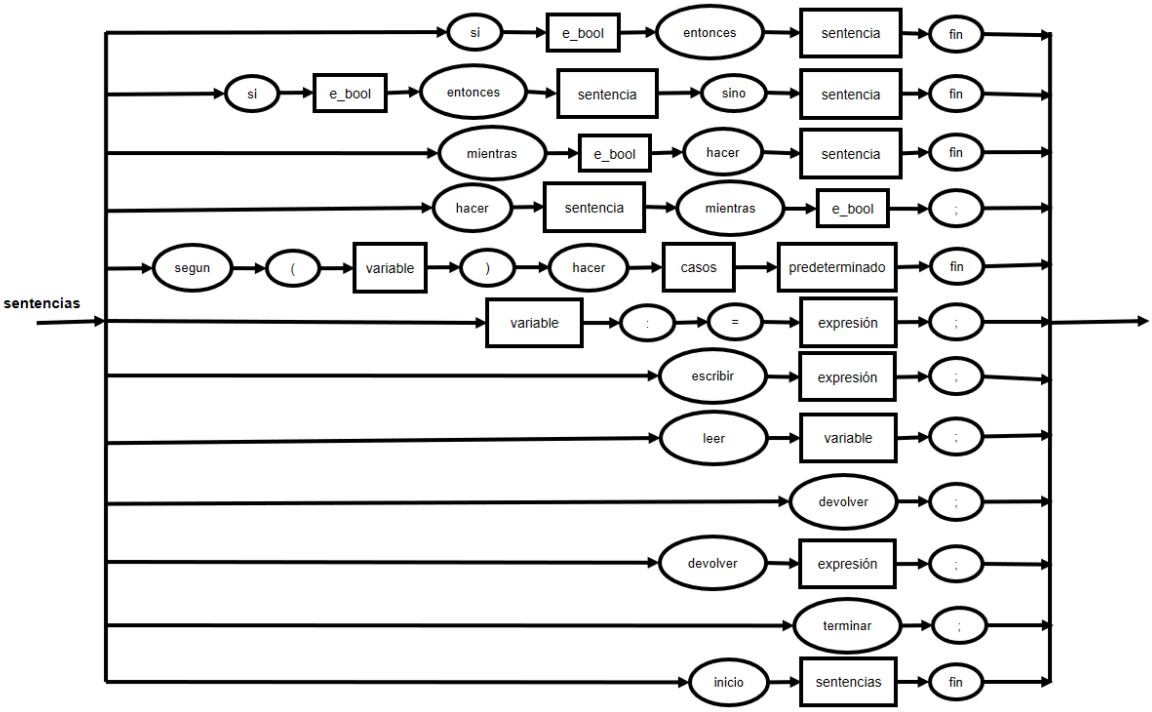
| **devolver** expresion**;**

| **terminar;**

| **inicio** sentencias **fin**

No presenta recursividad o ambigüedad.

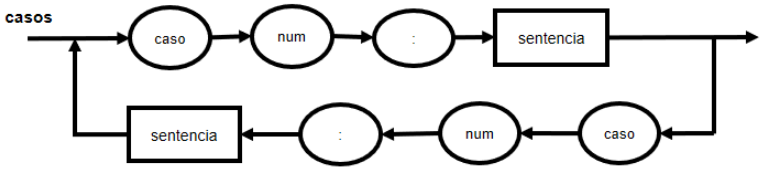
Diagrama de sintaxis:



16. casos → **caso num:** sentencia casos | **caso num:** sentencia

Presenta recursividad por la derecha. No es ambigua.

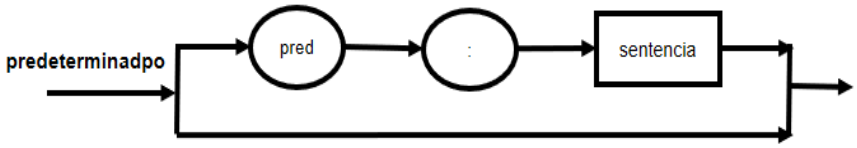
Diagrama de sintaxis:



17. predeterminado → **pred:** sentencia | ε

No presenta recursividad o ambigüedad.

Diagrama de sintaxis:



18. e_bool → e_bool **o** e_bool | e_bool **y** e_bool | **no** e_bool

| relacional | **verdadero** | **falso**

Se presenta ambigüedad.

a. Eliminación de ambigüedad:

e_bool → e_bool **o** e_bool1 | e_bool1

e_bool1 → e_bool1 **y** e_bool2 | e_bool2

e_bool2 → **no** e_bool | relacional | **verdadero** | **falso**

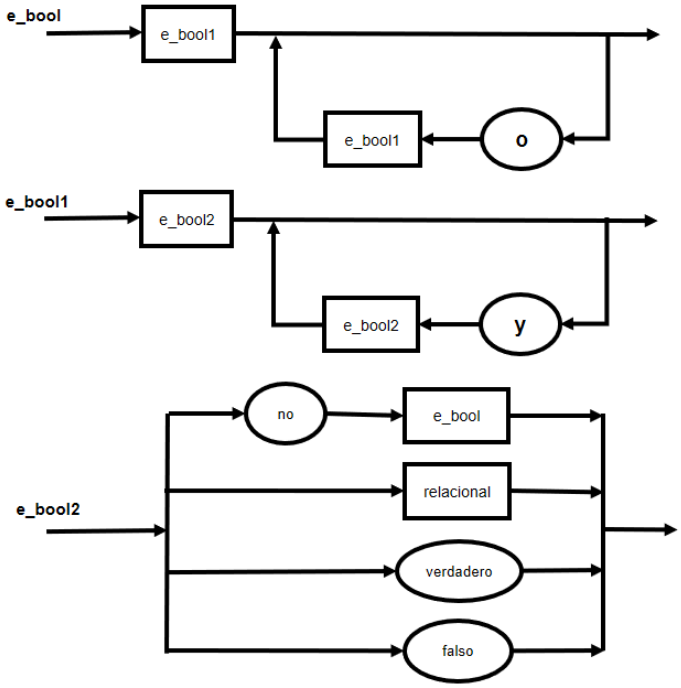
b. Notación EBNF:

e_bool → e_bool1 {**o** e_bool1}

e_bool1 → e_bool2 {**y** e_bool2}

e_bool2 → **no** e_bool | relacional | **verdadero** | **falso**

c. Diagrama de sintaxis:



19. relacional → relacional **>** relacional

| relacional **<** relacional

| relacional **<=** relacional

| relacional **>=** relacional

| relacional **<>** relacional

| relacional **=** relacional

| expresión

Se presenta ambigüedad.

a. Eliminación de ambigüedad:

relacional → relacional **>** relacional1

| relacional **<** relacional1

| relacional **<=** relacional1

| relacional **>=** relacional1

| relacional **<>** relacional1

| relacional **=** relacional1

| relacional1

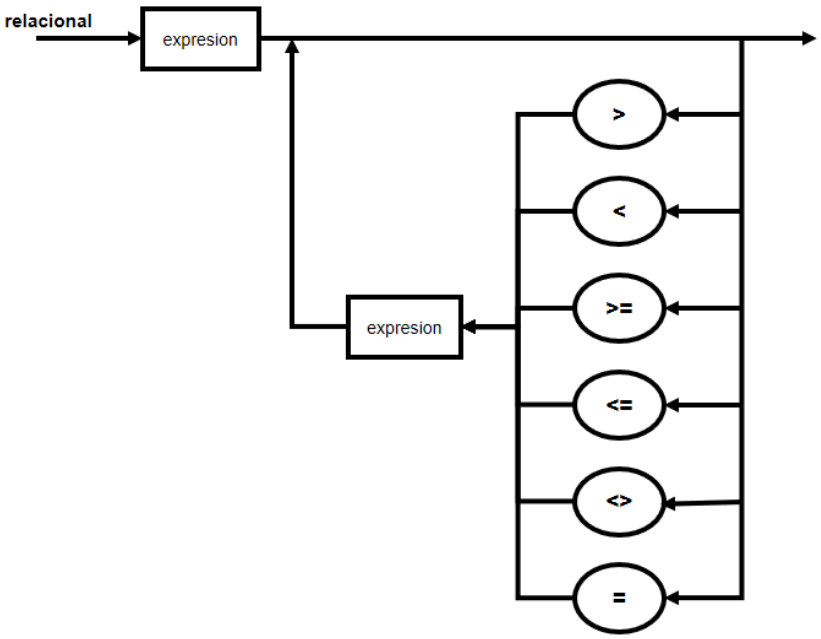
relacional1 → expresion

b. Notación EBNF:

relacional → relacional1 {(**>** | **<** | **<=** | **>=** | **<>** | **=**) relacional1}

relacional 1 → expresion

c. Diagrama de sintaxis:



20. expresion → expresion **+** expresion

| expresion **−** expresion

| expresion **∗** expresion

| expresion ***/*** expresion

| expresion **%** expresion | **(**expresion**)** |

| variable | **num** | **cadena** | **carácter**

Se presenta ambigüedad.

a. Eliminación de ambigüedad:

expresion → expresion **+** expresion1 | expresion **-** expresion1 | expresion1

expresion1 → expresion1 ***** expresion2

| expresion1 ***/*** expresion2

| expresion1 **%** expresion2

| expresion2

expresion2 → **(**expresion**)** | variable | **num** | **cadena** | **carácter**

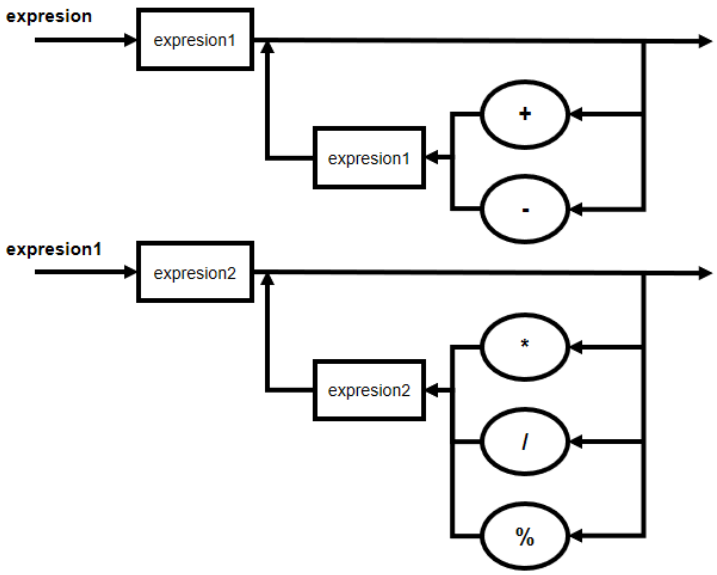
b. Notación EBNF:

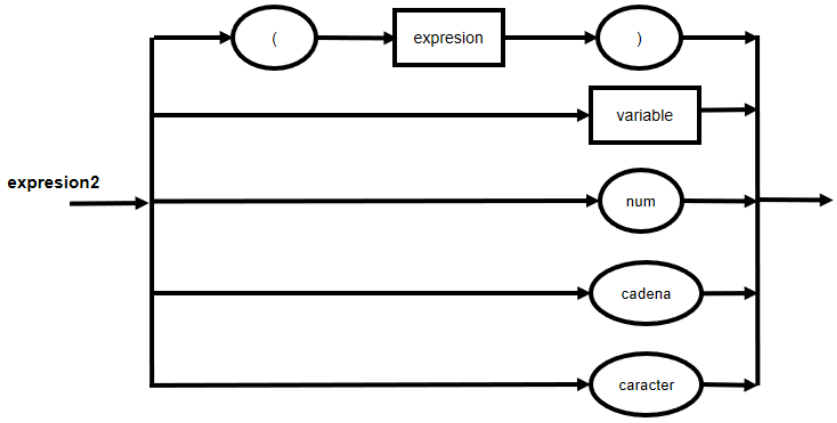
expresion → expresion1 {(**+** | **-**) expresion1}

expresion1 → expresion2 {(***** | ***/*** | **%**) expresion2}

expresion2 → **(**expresion**)** | variable | **num** | **cadena** | **carácter**

c. Diagrama de sintaxis:

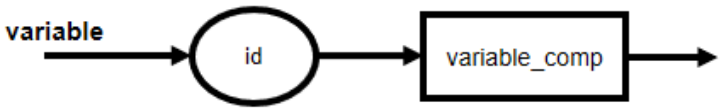




21. variable → **id** variable_comp

No presenta recursividad o ambigüedad.

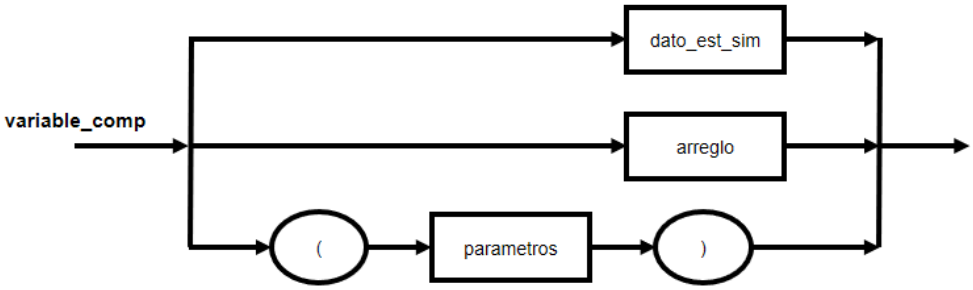
Diagrama de sintaxis:



22. variable_comp → dato_est_sim | arreglo | **(** parametros **)**

No presenta recursividad o ambigüedad.

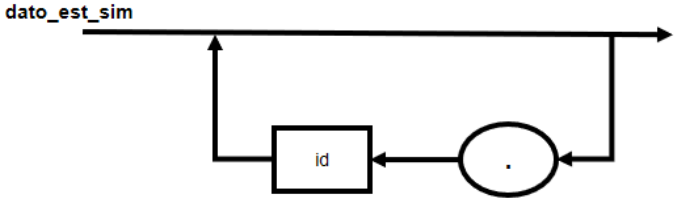
Diagrama de sintaxis:



23. dato_est_sim → dato_est_sim **.id** | ε

Presenta recursividad por la izquierda. No es ambigua.

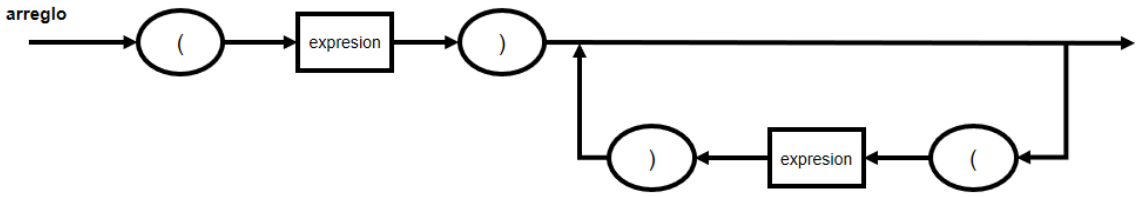
Diagrama de sintaxis:



24. arreglo → arreglo **(** expresion **)** | **(** expresion **)**

Presenta recursividad por la izquierda. No es ambigua.

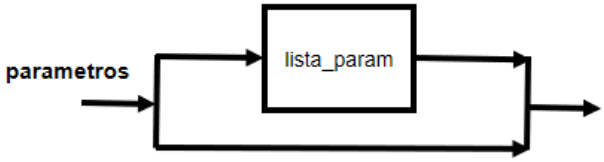
Diagrama de sintaxis:



25. parametros → lista_param | ε

No presenta recursividad o ambigüedad.

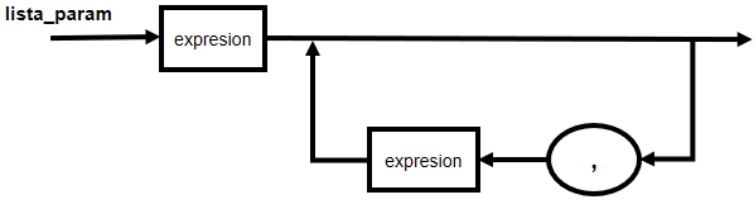
Diagrama de sintaxis:



26. lista_param → lista_param, expresion | expresión

Presenta recursividad por la izquierda. No es ambigua.

Diagrama de sintaxis:



c. Implementación

El analizados sintáctico se desarrollará en Byacc o Bison. Nosotros trabajaremos con Byacc, que conste en 3 secciones: sección de declaraciones, sección del esquema de traducción y sección de código usuario. Cada sección está separada por “%%”.

1. Sección de declaraciones

En esta sección se definen las librerías que se utilizaran para todo el programa, así como también la declaración de los prototipos de algunas funciones, por ejemplo: *extern int yylex().*

Otros aspectos que también se deben tomar en cuenta en esta sección es la declaración de todos los elementos terminales que contiene nuestra gramática. La definición del tipo y características que integran al atributo asociado de los elementos no terminales. La

asignación correspondiente asociatividad de los operadores. Además de señalar el símbolo inicial de la gramática.

2. Sección del esquema de traducción

En esta sección se colocarán cada una de las reglas que conforman la gramática de tal manera que estén representadas con una notación EBNF simplificada.

3. Sección de código usuario

Esta sección se utiliza para rutinas de complemento por ejemplo las que llaman al escáner. La presencia de esta sección es opcional según sea el caso, en nuestro caso únicamente la utilizaremos para declarar el cuerpo de la función yyerror (char *s).

d. Forma de ejecutar el código

Para poder compilar nuestro programa de forma correcta se deben realizar los siguientes pasos:

1. Compilar el programa uno correspondiente al analizador léxico desarrollado en Flex.

Para ello debemos abrir una terminal en nuestro ordenador. Se debe posicionar en la carpeta o dirección correspondiente en donde se encuentra el archivo. Una vez ahí ejecutamos el siguiente comando: lexer.l. Si se presentan errores se deben de corregir sino se debe realizar el paso siguiente.

2. Compilar el programa uno correspondiente al analizador sintáctico desarrollado en Byacc o Bison.

En la misma terminal se ejecutará el programa correspondiente al analizador sintáctico, que debe estar guardado en la misma ubicación que el programa correspondiente al analizador léxico. Una vez ahí ejecutado y sin presentar errores el programa en Flex, se ejecutará el siguiente comando: byacc -d parser.y. Si se presentan errores se deben de corregir sino se debe realizar el paso siguiente.

3. Compilar el archivo main.c.

Si no se presenta algún error se debe ejecutar el siguiente comando: gcc lex.yy.c y.tab.c main.c –o *nombre_ejecutable*, para generar el ejecutable que realiza la parte del analizador léxico y sintáctico de un compilador.

4. Implementación del ejecutable

Para llevar a cabo la implementación del ejecutable se deba aplicar en la terminal el siguiente comando: *./nombre_ejecutable nombre_archivo_prueba.* Donde el *nombre_archivo_prueba* contiene las cadenas a validar por nuestro analizador.