Informe 1

Maximiliano Arévalo, Vicente Brevis, Benjamín Muñoz.

Profesora Consuelo Ramírez.

Tabla de Contenidos

[**Introducción** 1](#_Toc9474081)

[**Marco Teórico** 2](#_Toc9474082)

[**Planteamiento del Problema** 4](#_Toc9474083)

[**Descripción de la Solución** 7](#_Toc9474084)

[**Conclusiones** 14](#_Toc9474085)

[**Referencias** 15](#_Toc9474086)

# **Introducción**

Existen diferentes lenguajes de programación, los cuales se clasifican en lenguajes interpretados y lenguaje compilados dependiendo del proceso que realizan para ejecutar su programa. Entre los lenguajes compilados destacan C y Java, y para el caso de los interpretados Python.

Que un lenguaje sea compilado, quiere decir que necesita de un proceso llamado compilación, en el cual las instrucciones de código fuente pasan a lenguaje de máquina para que el computador las pueda entender y posteriormente llevar a cabo su ejecución (Escobar G.,2017). Para realizar esto, se necesita comprender el funcionamiento del lenguaje de programación, donde uno de sus pasos necesarios corresponde a la etapa de análisis léxico, donde se leen y detectan los componentes léxicos del lenguaje en particular que se está analizando.

Anteriormente, se implementó un analizador léxico para un número finito de elementos del lenguaje de programación C a partir de una serie de diagramas de sintaxis otorgados por la profesora en la plataforma virtual del curso. El objetivo principal de esta experiencia consiste en construir un nuevo analizador léxico, pero utilizando autómatas finitos, en otras palabras, modelar el trabajo realizado anteriormente como un programa en C, pero ahora utilizando autómatas.

El presente documento pretende describir el proceso realizado para construir el autómata que representa el analizador léxico para el lenguaje de programación C, mostrando en otras secciones el marco teórico, el problema planteado y el desarrollo de éste, para dar paso finalmente a las conclusiones del trabajo realizado.

# **Marco Teórico**

Para el presente informe se hará una breve explicación sobre los conceptos claves que tienen mayor relevancia, en cuanto al entendimiento sobre los analizadores léxicos, estos conceptos son:

**Autómatas finitos:** Un autómata finito o máquina de estado finito es un modelo matemático de un sistema que recibe una cadena constituida por símbolos de un alfabeto y determina si esa cadena pertenece al lenguaje que el autómata reconoce. Estos se pueden clasificar en autómata finito determinista (AFD), autómata finito no determinista (AFN), autómata finito no determinista con transiciones ε (AFN-ε). Además de su respectiva definición, donde tenemos:

A = (Q, ∑, δ, q0, F)

Q: conjunto finito de estados.

∑: alfabeto de entrada.

δ: función de transición.

δ:

q0: estado inicial.

q0Q

F: conjunto de estados finales o de aceptación. FQ

Se puede representar a través de tablas de transiciones y diagramas de transiciones. Para el caso de las tablas de transiciones, solo basta desarrollarlas de la siguiente forma:

|  |  |  |
| --- | --- | --- |
| δ | 0 | 1 |
| q0 | q2 | q0 |
| \*q1 | q1 | q1 |
| q2 | q2 | q1 |

**Analizador léxico:** Es la primera fase de un compilador donde recibe un código fuente, y a partir de este genera una salida de elementos léxicos llamados tokens que siguen un patron especificado, los cuales se usan posteriormente para realizar un análisis sintáctico. (Rodríguez C., 2012).

# **Planteamiento del Problema**

El problema por enfrentar corresponde a la construcción de un analizador léxico para el lenguaje de programación C haciendo uso de los autómatas finitos, en este caso un AFN. Es decir, básicamente hay que realizar lo mismo que se realizó para el programa nº1, en donde dados unos diagramas de sintaxis correspondientes a diferentes funcionalidades del lenguaje C, construir un analizador léxico que reconozca como autómata los siguientes elementos terminales:

* Palabras reservadas:
  + main
  + auto
  + extern
  + register
  + static
  + do
  + while
  + for
  + goto
  + if
  + else
  + return
  + sizeof
  + short
  + unsigned
  + long
  + int
  + float
  + char
  + double
  + default
  + continue
  + break
* Símbolos:
  + (
  + )
  + {
  + }
  + ,
  + \*
  + =
  + ;
  + :
  + &
  + -
  + ~
  + |
  + ==
  + !=
  + <
  + >
  + <=
  + >=
  + &&
  + ||
  + !
  + ?
  + +
  + /
  + %
  + ^
  + <<
  + >>
  + ++
  + --

Sin embargo, hay que tener en cuenta que para el diseño del autómata finito se puede utilizar tanto el diagrama de transiciones como la tabla de transiciones, tratando de usar la opción que facilite el diseño de este y también permita comprender el autómata y su funcionamiento de una manera más clara.

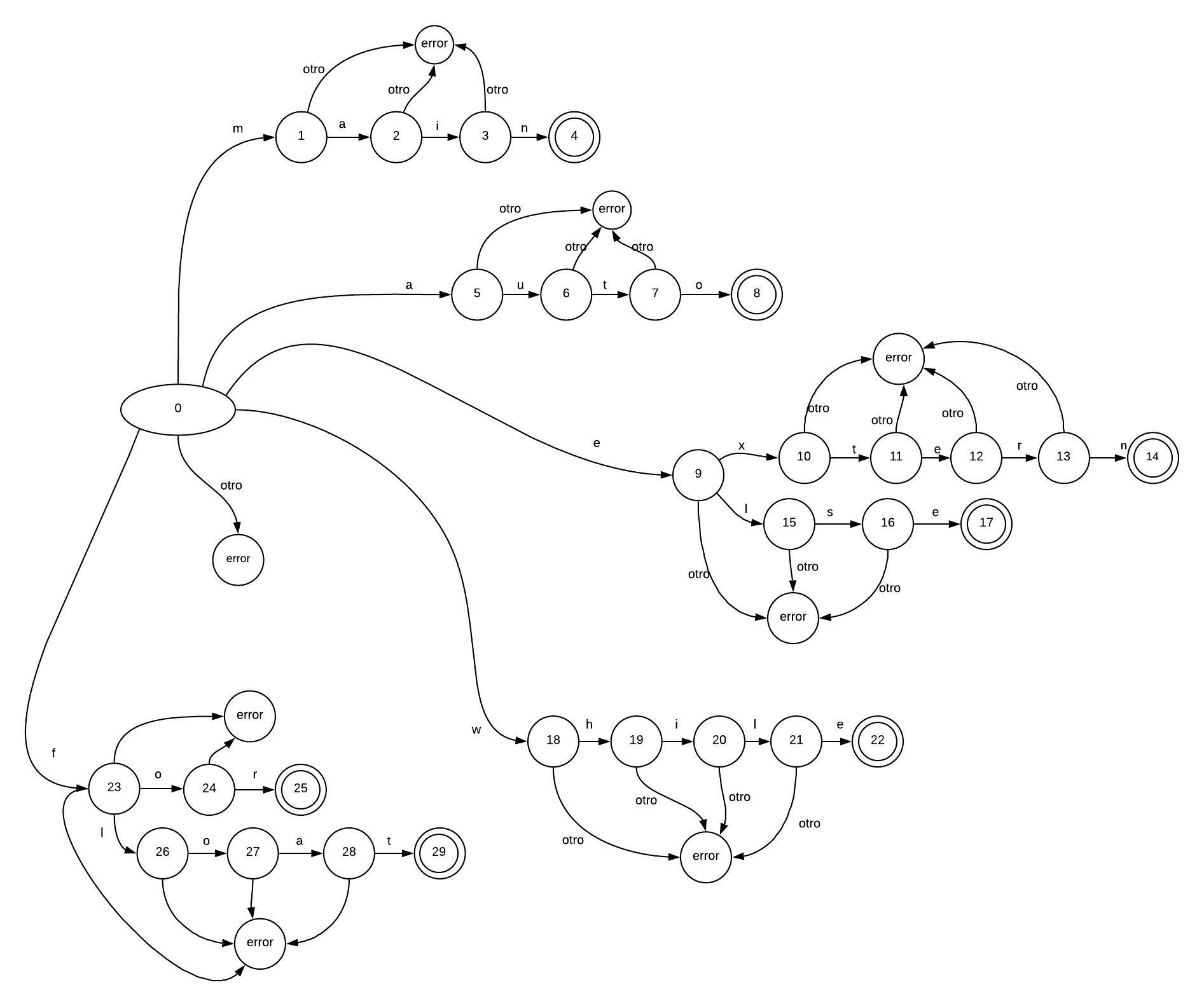
# **Descripción de la Solución**

Para enfrentar el problema, se vuelve a ver la solución ya propuesta una vez en el programa n°1, donde se analiza cada símbolo que se lee, y según este se puedan construir los tokens que deben ser leídos por el autómata. En este caso, se desarrolló un autómata del tipo AFN el cual se caracteriza por no necesariamente tener una salida para cada transición desde un estado. Dicho autómata posee un estado 0, el cual irá a otros estados según el símbolo que se lee, y se envía a otros analizadores más pequeños.

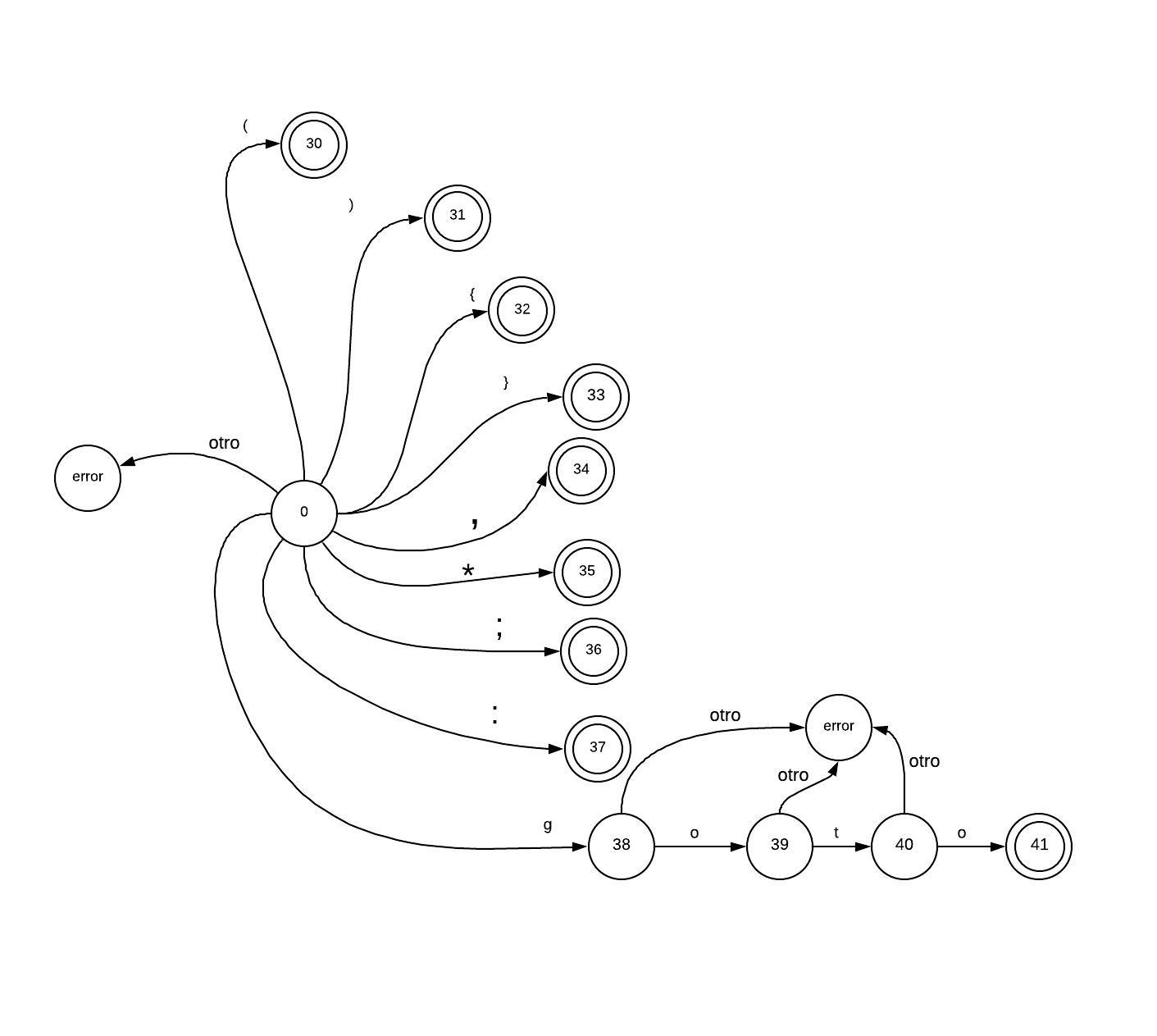
El analizador empleado resultó ser un autómata de 135 estados, en el cual al tener error o detectar un token se reinicia, es decir, comienza a leer de nuevo desde el estado inicial 0 hasta no poder detectar más componentes léxicos.

A continuación, se muestra la solución gráfica de dicho autómata, pero por términos de espacio se divide este en 6 grupos, todos partiendo desde el mismo estado 0.

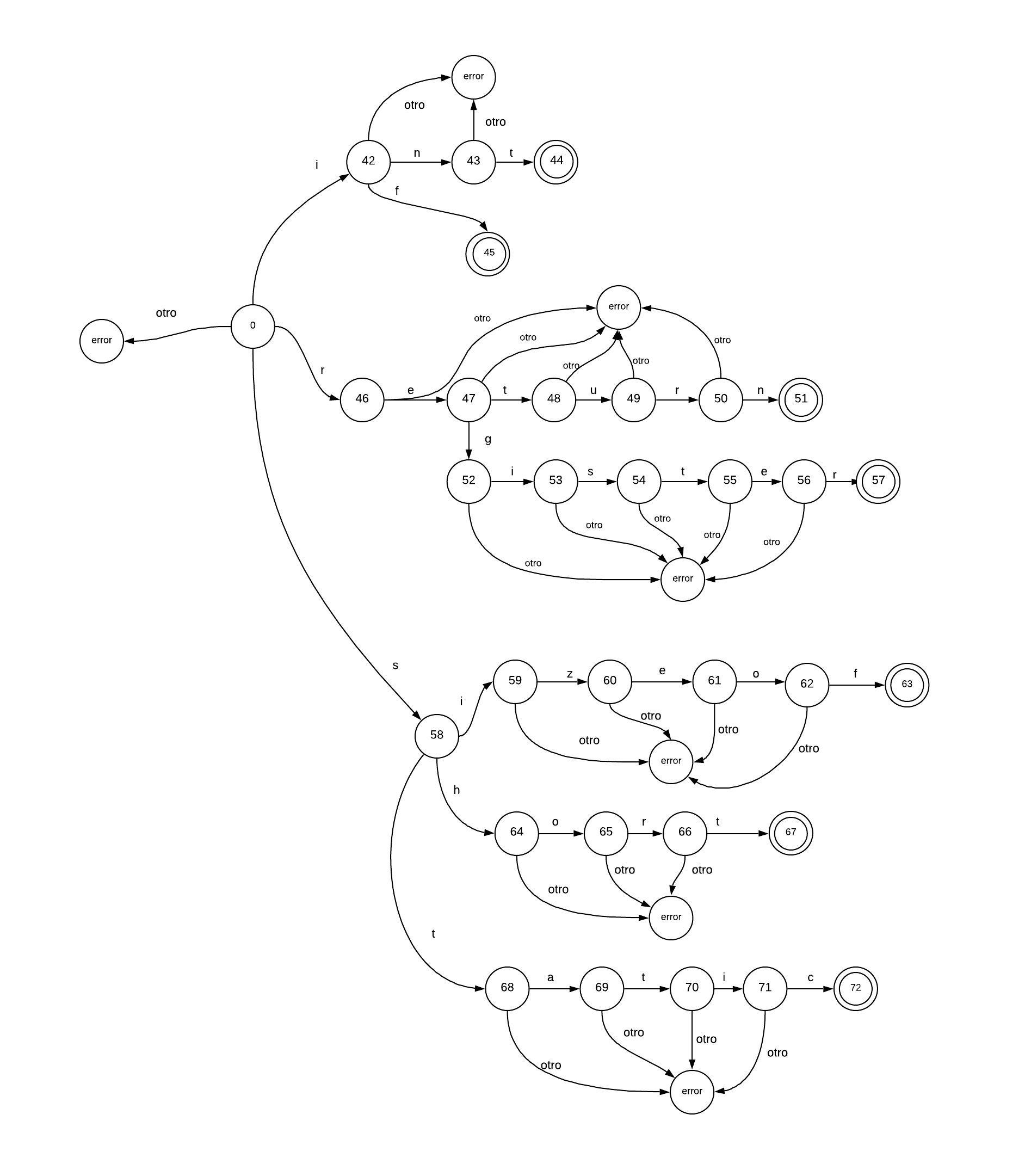
Para el primer grupo se presenta la implementación con las palabras reservadas main, auto, extern, else, while, for y float. En este caso los analizadores que participan no detectan componentes léxicos a no ser que se detecte toda la palabra reservada, ya que si va al estado error el autómata hará reinicio y comenzará a leer nuevamente desde donde quedo.



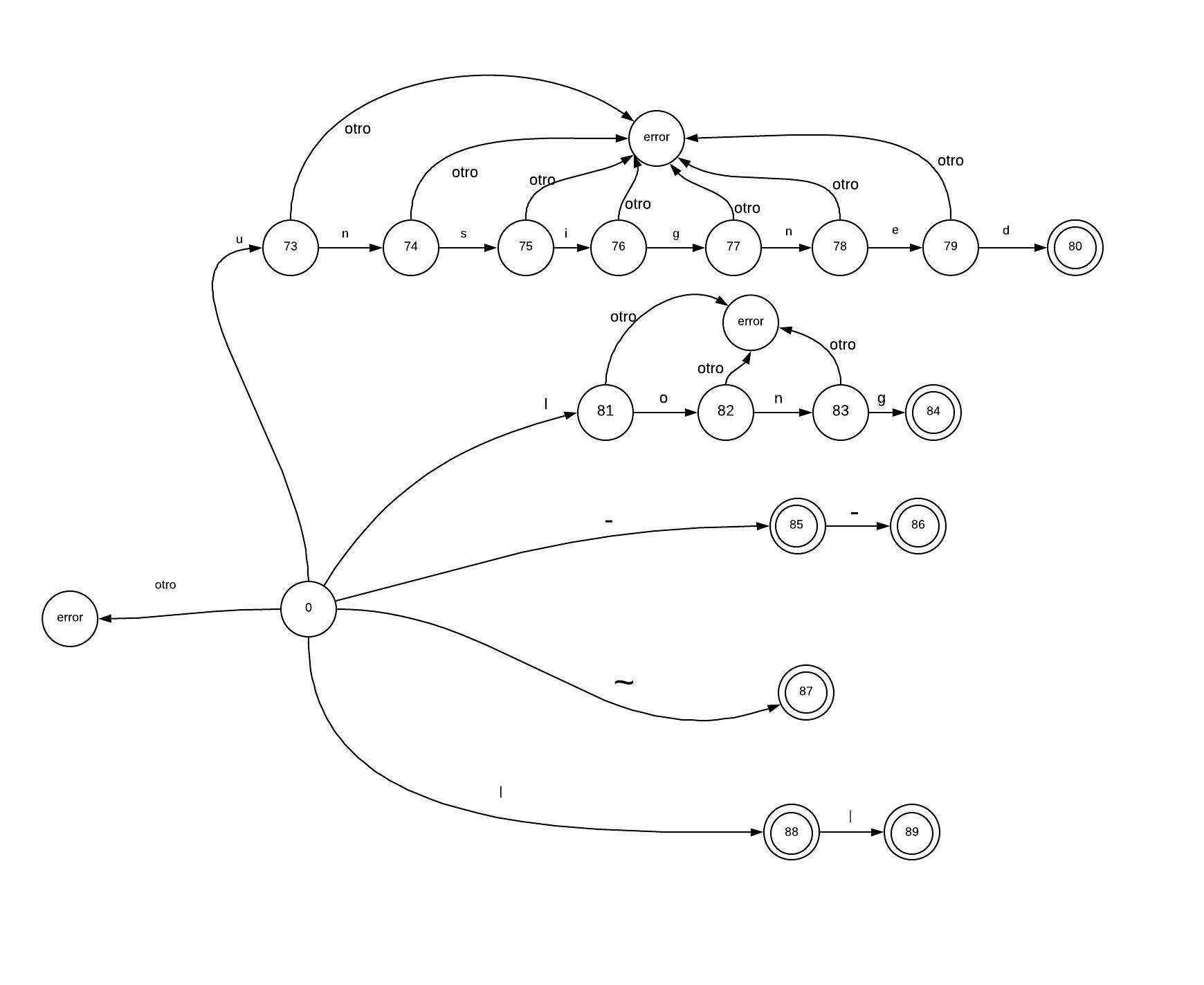
Para el segundo grupo de analizadores, se tiene la implementación para la palabra reservada goto, y los caracteres (, ), {, }, \*, ; , : y ,. En dicho caso se lee el símbolo y el analizador se reiniciará ya que no es necesario juntar más caracteres a excepción de la palabra goto.



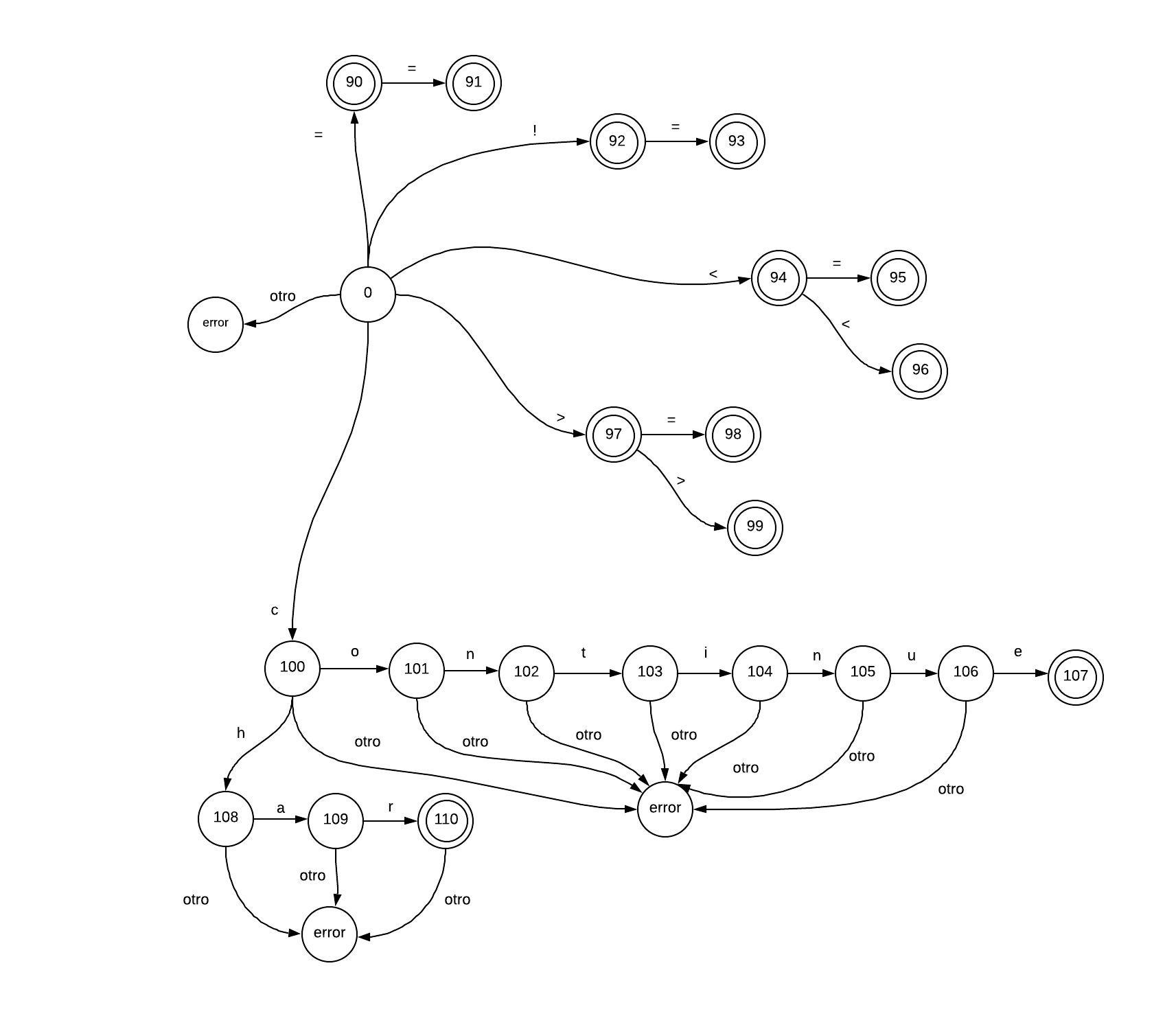
En el tercer grupo se desarrolla el analizador para las palabras int, if, return, register, sizeof, short y static. No muestra diferencias de implementación respecto a los autómatas anteriores.



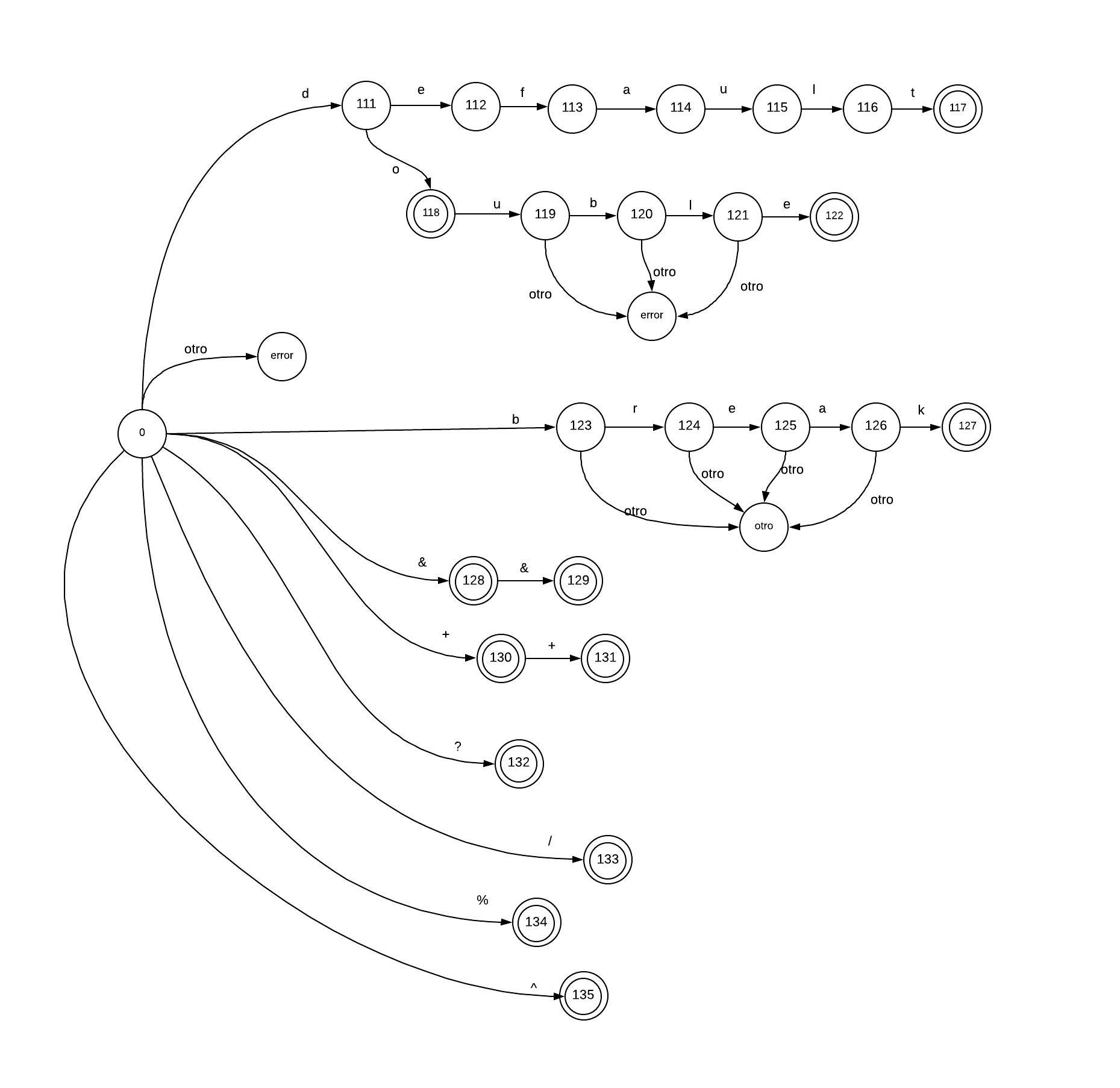
El cuarto grupo se encarga de detectar las palabras unsigned, long, los operadores aritméticos - y --, el operador unario | y el operador lógico ||. En este caso se ve que || y - presentan una repetición de símbolos, por lo que para cada caso un símbolo por sí sólo es terminal (por ejemplo: -) y a nivel de implementación se entiende que si no se presenta un segundo símbolo - o | según sea el caso, el autómata detecta dicho símbolo y procede a reiniciarse.



El quinto grupo presenta la detección de las palabras continue y char, los operadores <=, <<, <, >, >>, >=, == y !=, y los símbolos = y ! por sí solos. La implementación de este no difiere con alguna ya mencionada para los autómatas anteriores.



Finalmente, el último grupo se encarga de leer las palabras reservadas default, do, double, break, los operadores &, &&, +, ++, ?, / %, y ^. Su implementación solo difiere en el caso de do y double, ya que la primera está contenida en la segunda, donde en dicho caso se verifica si después de leer una “o”, se puede seguir para leer double, si no hay nada más que leer el autómata reconoce do y se reinicia.



Finalmente cabe mencionar que este autómata no es la única solución al problema planteado para analizar el lenguaje de programación C, y que tampoco es óptimo ya que no ha pasado por ningún método de minimización, así que puede que en otras implementaciones varíe la cantidad de estados y el orden de estos.

# **Conclusiones**

# **Referencias**

Escobar, Germán (2017). Lenguajes Compilados e Interpretados. Recuperado de <https://blog.makeitreal.camp/lenguajes-compilados-e-interpretados/>

Rodríguez, Casiano (2012). Análisis Léxico. Recuperado de <http://nereida.deioc.ull.es/~pl/perlexamples/node71.html>