Analizador Semántico

Universidad de Guadalajara, Centro Universitario de Ciencias Exactas e Ingenierías.

Autores:

Juan José Salazar Villegas. Código: 215661291

Paola Vanessa Del Rio Gómez. Código: 215480181

Hernández Martínez Mally Samira. Código: 220286113

Juan Emmanuel Fernández de Lara Hernández. Código: 220286571

19/03/2023

Carrera: Ingeniería En Computación (INCO)

Asignatura: Traductores De Lenguajes II

Maestro: Ramos Barajas Armando

Origen del informe: Guadalajara Jalisco México

Ciclo: 2023 A

Sección: D03

Actividad: 3

índice

Introducción

Durante el desarrollo de cada etapa de nuestro compilador podemos observar cómo se abarca el comprender cada elemento para su procesamiento para definir, analizar e identificar cada entrada para posteriormente sumarle más elementos de comprensión.

Ahora desarrollaremos el apartado semántico el cual es una parte importante del proceso de compilación de un programa de computadora. Su función es analizar el código fuente del programa y verificar que las instrucciones escritas en el lenguaje de programación sean coherentes y tengan sentido desde un punto de vista semántico.

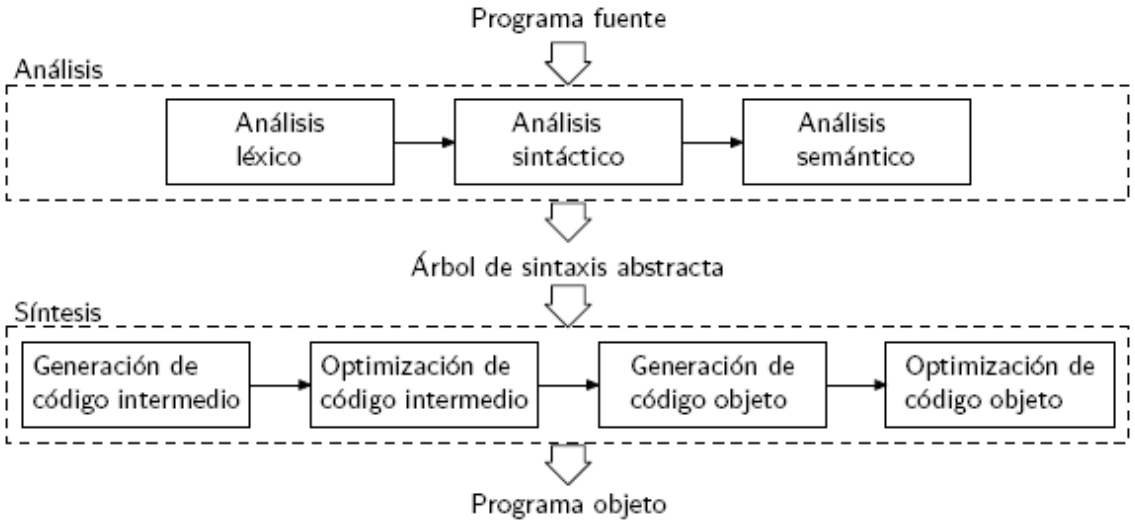
Este analizador se basa en el uso del árbol sintáctico anteriormente realizado y dicha información recopilada en la tabla de símbolos se usa para la comprobación y consistencia semántica de nuestro programa junto con la definición del lenguaje. Este también recopila información sobre el tipo y posteriormente la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla mas tarde al momento de la generación de código intermedio.

Como dijimos lo más destacado del análisis semántico es la comprobación o verificación de los tipos, donde nuestro compilador verifica que cada operador tenga su correspondiente operando que haga coincidencia. Por ejemplo en muchas definiciones de lenguajes de programación es requerido de un índice de un arreglo sea entero, aquí el compilador responde con un error si se utiliza un numero con punto flotante para indexar el arreglo.

Al especificar el lenguaje se puede permitir ciertas conversiones de tipo conocidas como coerciones. Esto puede aplicarse a un operador binario aritmético a un par de entero o un par de números flotantes si este se aplica a un numero de punto flotante y a un entero, el compilador puede convertir u obligar a que se convierta en un numero de punto flotante.

Dentro de esta parte se vera como convertir el texto que se tenía en entrada a una estructura mas elaborada como lo es el árbol, con dicha organización de los

elementos nos dará un sentido mas amplio de la traducción. Donde encontraremos los símbolos correspondientes y un manejador de errores semánticos.



Objetivo General:

Como se planteó en la introducción se busca el demostrar un análisis complementario a nuestras faces ya estructuradas, creando un analizador semántico en C, esto base a los Tokens establecidos como palabras reservadas y continuando con el sesgo de los diversos tipos de datos y caracteres de entrada.

Buscando establecer los puntos principales como lo son la verificación de la compatibilidad de tipos, chequeo de variables, errores de uso de funciones, así como los errores sintácticos para posteriormente recabar dicha información para la comprensión del funcionamiento del compilador.

Objetivo Particular:

Se busca realizar complementar la primera fase realizada sumándole el análisis semántico de las palabras reservadas establecidas esto mismo por los tokens definidos anteriormente así se dará un análisis con metodologías típicas de ingeniería de software. Además de realizar diagramas que permitan la representación gráfica del sistema para un mejor entendimiento.

Desarrollo (Pantallazos)

Desarrollo del programa:

Identificador de Tokens

```
while (compilado.get(caracterToken)) {
  if (caracterToken == '|') {

    if (posicionToken == 0) {
      if (tokenAlmacenado == "identificador"){
        cout << "[Error] Identificador no declarado" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "identificadorString"){
        cout << "[Error] Identificador no declarado" << endl;
      }
      else if (tokenAlmacenado == "numero"){
        cout << "[Error] Numero encontrado antes de tipo de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "cadena") {
        cout << "[Error] Cadena encontrada antes de tipo de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "else"){
        cout << "[Error] else encontrado antes de if" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "+"){
        cout << "[Error] Operador de suma encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "-"){
        cout << "[Error] Operador de resta encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "*"){
        cout << "[Error] Operador de multiplicacion encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "/"){
        cout << "[Error] Operador de division encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "%"){
        cout << "[Error] Operador de modulo encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "="){
        cout << "[Error] Operador de asignacion encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "!"){
        cout << "[Error] Operador de desigualdad encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "=="){
        cout << "[Error] Operador de igualdad encontrado antes de dato" << endl;
        error = true;
      }
      else if (tokenAlmacenado == "!="){
        cout << "[Error] Operador de diferente encontrado antes de dato" << endl;
        error = true;
```

Dentro de esta sección de código podremos evaluar cada uno de los tokens definidos previamente para dar su correspondiente mensaje de error estos van desde palabras reservadas, signos entre otros.

Ahora nuestro objetivo será el definir el tipo de dato además de como nosotros esperamos que se escriba correctamente y mensaje de que no cumpla.

Con este podremos dar evaluaciones al momento de extraer cada token y relacionarlo con cualquier tipo de estructura previamente validada, como lo puede ser un while que no tiene sus respectivos paréntesis u operadores lógicos necesarios para funcionar.

Análisis por estructura:

```
if (tokenCondicion == "while"){
  if (tokenAlmacenado != "while"){
    if (tokenAnterior == "while"){
      if (tokenAlmacenado == "("){
        tokenAnterior = "(";
        goto salto;
      }
      else {
        cout << "[Error] Parentesis de apertura no encontrado despues de while" << endl;
        error = true;
        break;
      }
    }

    if (tokenAnterior == "("){
      if (tokenAlmacenado == "numero" || tokenAlmacenado == "identificador" || tokenAlmacenado == "true" || tokenAlmacenado == "false"){

        tokenAnterior = tokenAlmacenado;
        goto salto;
      }
      else {
        cout << "[Error] Dato no encontrado despues de parentesis de apertura" << endl;
        error = true;
        break;
      }
    }
```

Como previamente lo dijimos al tener dichos tokens definidos para el análisis de los errores debemos darle a cada estructura **(For, While, Do while, etc),** sus respectivas reglas a evaluar asi definimos cuando estas están bien escritas, esto se lleva a cabo leyendo los tokens y segmentándolos en su tipo asi al momento de juntarlos podemos ver que cumplan sus propias reglas. Como lo es este ejemplo con el While

```
174             else if (tokenAlmacenado == "identificador"){
175               tokenAnterior = "identificador";
176             }
177             else if (tokenAlmacenado == "identificadorString"){
178               tokenAnterior = "identificadorString";
179             }
180             else if (tokenAlmacenado == "numero"){
181               tokenAnterior = "numero";
182             }
183             else if (tokenAlmacenado == "repetido"){
184               error = true;
185             }
186             else if (tokenAlmacenado == "error"){
187               error = true;
188             }
189           }
190
```

En esta parte vemos como la semántica nos ayuda a definir desde el tipo de dato, así como el evaluar que la variable este repetida juntando en analizador sintáctico para su evaluación.

```
case 1:
  // cout << "[Identificador] " << token << endl;
  archivo << "identificador|";
  break;

case 51:
  // cout << "[Identificador] " << token << endl;
  archivo << "identificadorString|";
  break;

case 2:
  cout << "[Error] Identificador declarado mas de una vez: " << token << endl;
  archivo << "repetido|";
  error = true;
  break;

case 50:
  cout << "[Error] Identificador no declarado: " << token << endl;
  archivo << "noDeclarado|";
  error = true;
  break;
```

Conclusiones

Juan José Salazar Villegas

Para el desarrollo de esta fase puede comprender la importancia y como cada punto previo desarrollado toma importancia. Así como el analizador semántico nos ayuda a garantizar que el programando tenga errores por su sintaxis y este tenga sentido desde el punto de vista semántico, lo que puede reducir la cantidad de errores de nuestro programa y hacer más fácil la depuración.

Juan Emmanuel Fernández de Lara Hernández

El análisis semántico ayuda a garantizar que el programa funcione correctamente en tiempo de ejecución, evitando posibles errores que puedan causar fallas o interrupciones en su funcionamiento. Por tanto, es esencial dedicar tiempo y recursos a esta etapa para garantizar que el programa esté bien construido y sea de alta calidad. Es importante mencionar que, aunque el análisis semántico puede requerir más tiempo y esfuerzo en su implementación, esto puede resultar en una investigación más profunda y un mejor conocimiento del código fuente.

Mally Samira Hernandez Martinez

En mi opinión, luego de realizar esta actividad puedo decir que el análisis semántico es una etapa crucial del proceso de compilación que se encarga de verificar que el significado de las estructuras sintácticas del código fuente es coherente y válido. En resumen, el análisis semántico ayuda a garantizar que el programa funcione de manera correcta y sin errores en el tiempo de ejecución.

Paola Vanessa Del Rio Gómez

A mí parecer, este programa nos costó un poco más de trabajo ya que presentaba varios errores, pero eso nos hizo que la parte de la investigación fuera mejor, el hacer el análisis semántico nos funciona para su correcta depuración y que el programa no presente errores al momento de nosotros correrlo.

Bibliografía

- Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman. (s. f.). Compiladores principios, técnicas y herramientas (2.a ed.). Pearson.
- Ejemplos de Análisis Semántico. (2014, 2 abril). Gramáticas. Recuperado 25 de octubre de 2021, de https://www.gramaticas.net/2012/05/ejemplos-de- analisis-semantico.html
- Reinhard Wilhelm; Helmut Seidl; Sebastian Hack (13 May 2013). Compiler Design: Syntactic and Semantic Analysis. Springer Science & Business Media.

Apéndices

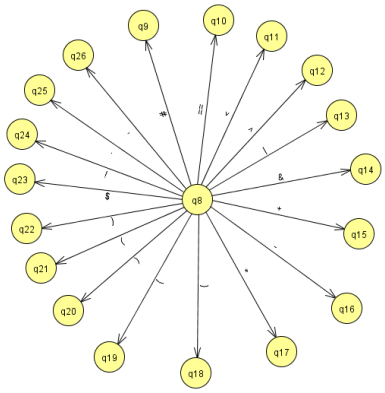
No se cuenta con apéndices para este reporte.

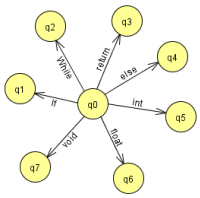
Acrónimos

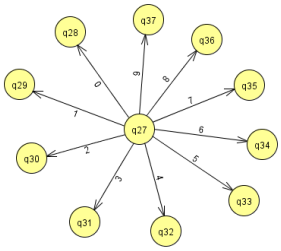
No se cuenta con apéndices para este reporte.

Diagramas

Grafos:







Caso de uso

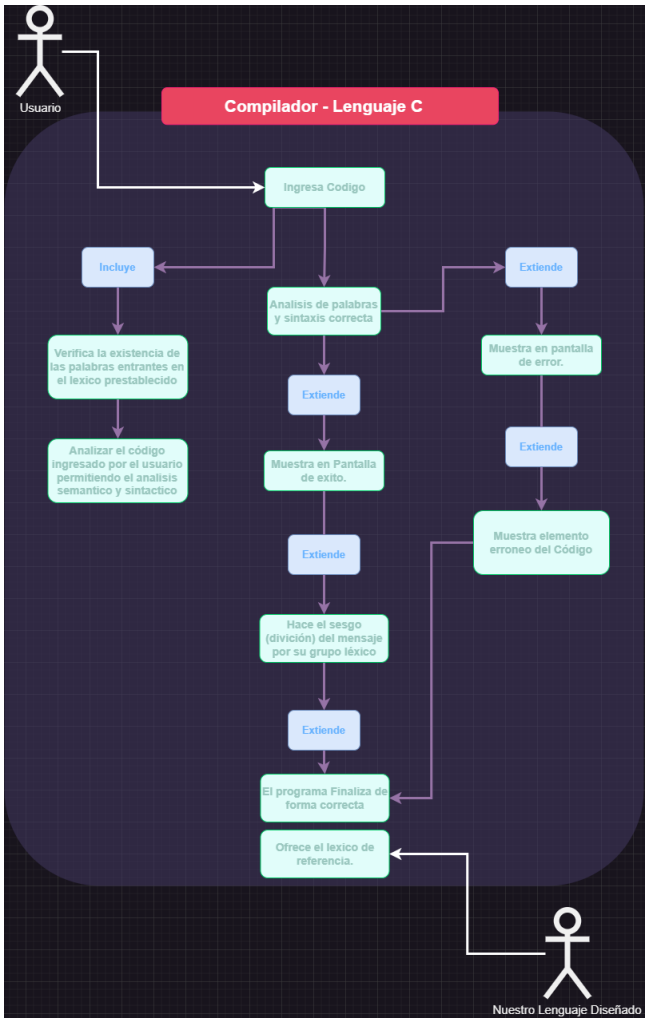


Tabla de transiciones

EDO	if	while	return	else	int	float	void
q0	{q1}	{q2}	{q3}	{q4}	{q5}	{q6}	{q7}
q1	0	0	0	0	0	0	0
q2	0	0	0	0	0	0	0
q3	0	0	0	0	0	0	0
q4	0	0	0	0	0	0	0
q5	0	0	0	0	0	0	0
q6	0	0	0	0	0	0	0
q7	0	0	0	0	0	0	0

EDO	#	==	<	>	|	&	+	-	*	{	}	(	)	[	$	!	.	,
q8	{q9}	{q10}	{q11}	{q12}	{q13}	{q14}	{q15}	{q16}	{q17}	{q18}	{q19}	{q20}	{q21}	{q22}	{q23}	{q24}	{q25}	{q26}
q9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

EDO	0	1	2	3	4	5	6	7	8	9
q27	{q28}	{q29}	{q30}	{q31}	{q32}	{q33}	{q34}	{q35}	{q36}	{q37}
q28	0	0	0	0	0	0	0	0	0	0
q29	0	0	0	0	0	0	0	0	0	0
q30	0	0	0	0	0	0	0	0	0	0
q31	0	0	0	0	0	0	0	0	0	0
q32	0	0	0	0	0	0	0	0	0	0
q33	0	0	0	0	0	0	0	0	0	0
q34	0	0	0	0	0	0	0	0	0	0
q35	0	0	0	0	0	0	0	0	0	0
q36	0	0	0	0	0	0	0	0	0	0
q37	0	0	0	0	0	0	0	0	0	0

Requisitos Funcionales:

Numero de requisito: **RF01**

Nombre de requisito: Actualización de tokens

Tipo: Requisito

Fuente del requisito: Función básica de compilador

Prioridad del requisito: Alto/Esencial

Descripción:

Al iniciar el programa este actualizara los tokens establecidos junto las funciones que clasifican los tipos de datos.

Numero de requisito: **RF02**

Nombre de requisito: Procesamiento de entradas.

Tipo: Requisito

Fuente del requisito: Función básica de compilador

Prioridad del requisito: Alto/Esencial

Descripción:

El usuario ingresa algún código, cadena o palabra la cual se procesará y detectara en que categoría pertenece y asignarla para posteriormente mostrar en pantalla y dar el tipo de dato relacionado a su entrada. De no ser así se mostrará un mensaje de error.

Numero de requisito: **RF03**

Nombre de requisito: Reacción a errores.

Tipo: Requisito

Fuente del requisito: Función básica de compilador

Prioridad del requisito: Alto/Esencial

Descripción:

Cuando se presenta algún tipo de error desde la perspectiva del usuario no generará conflictos ya que no interrumpirá el curso del programa ni la validación de códigos consecuentes.

Requisitos No Funcionales:

Numero de requisito: **RNF01**

Nombre de requisito: GUI / Interfaz Visual

Tipo: Requisito

Fuente del requisito: Interfaz de usuario.

Prioridad del requisito: Media/Deseado

Descripción:

Al pensar en Interfaz hablamos de un menú simple donde se pueda desplazar fácilmente y tenga un orden claro y directa esto para que el usuario pueda ingresar datos fácilmente sin tener que comprender un complejo sistema de interfaz

Numero de requisito: **RNF02**

Nombre de requisito: Eficacia y calidad del software.

Tipo: Requisito

Fuente del requisito: Funcionalidad optima del programa brindando una buena experiencia.

Prioridad del requisito: Alta/Esencial

Descripción:

El programa debe ser capaz de analizar y catalogar correctamente la mayoría de los caracteres de entrada por el usuario, dando una experiencia optima.

Numero de requisito: **RNF03**

Nombre de requisito: Accesibilidad y funcionalidad optima

Tipo: Requisito

Fuente del requisito: Facilidad de uso.

Prioridad del requisito: Alta/Esencial

Descripción:

El acceso y desempeño de las funciones del software deberán ser rápidos y fácil de usar siendo intuitivo y limitando trabas de uso. De ser necesario se buscará optimizar ante el numero de errores y el poco desempeño.

Complejidad Ciclomática:

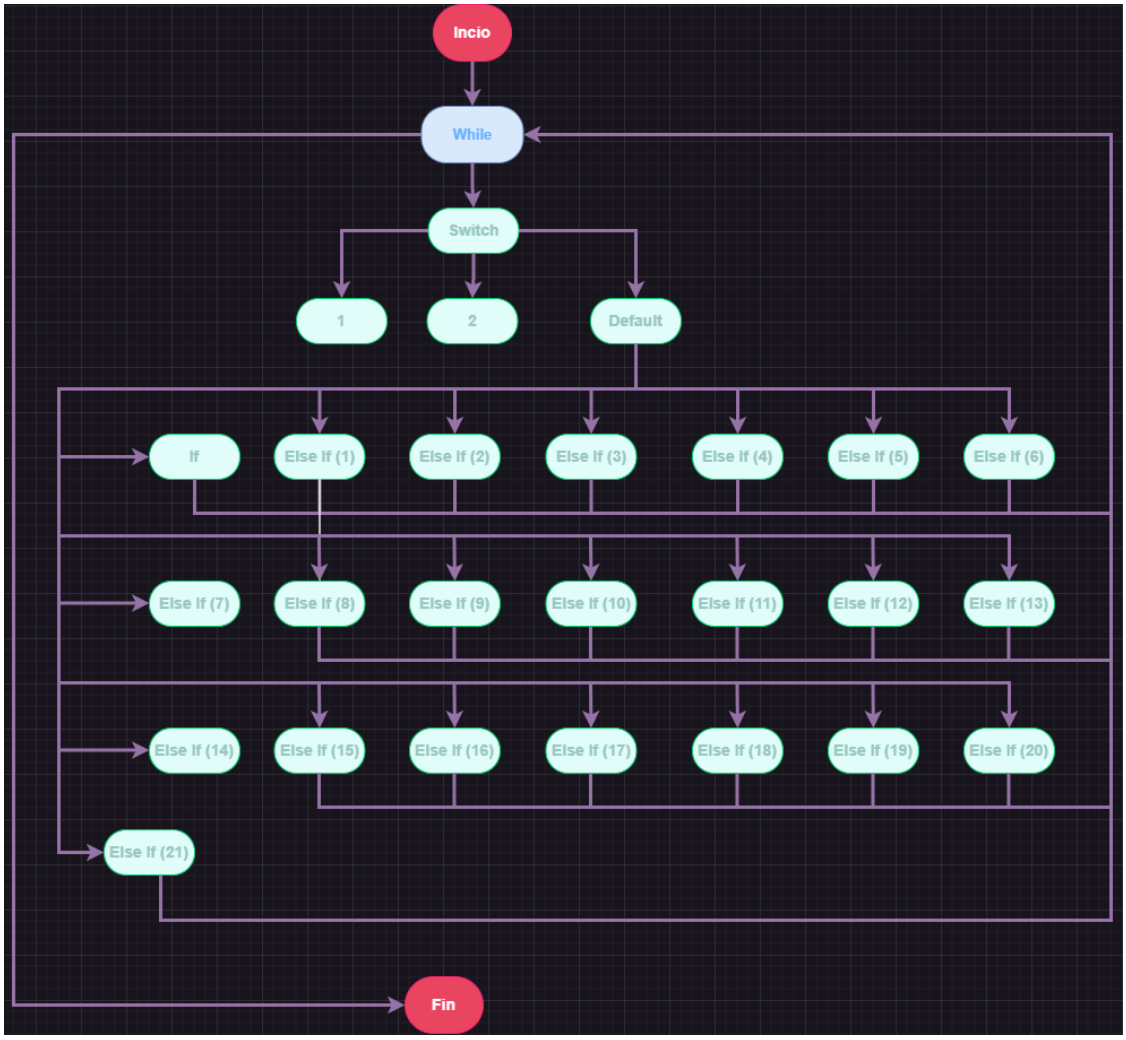
```
switch(res) {
  // case -1:
  //   cout << "[Error] Caracter no valido: " << token << endl;
  //   archivo << "error|";
  //   break;

  case 0:
    cout << "[Error] El identificador no puede ser una palabra reservada: " << token << endl;
    archivo << "error|";
    error = true;
    break;

  case 1:
    // cout << "[Identificador] " << token << endl;
    archivo << "identificador|";
    break;

  case 51:
    // cout << "[Identificador] " << token << endl;
    archivo << "identificadorString|";
    break;

  case 2:
    cout << "[Error] Identificador declarado mas de una vez: " << token << endl;
    archivo << "repetido|";
    error = true;
    break;

  case 50:
    cout << "[Error] Identificador no declarado: " << token << endl;
    archivo << "noDeclarado|";
    error = true;
    break;

  case 3:
    // cout << "[Numero] " << num << endl;
    archivo << "numero|";
    break;

  case 4:
    // cout << "[OperadorSuma] " << token << endl;
    archivo << "+|";
    break;

  case 5:
    // cout << "[OperadorResta] " << token << endl;
    archivo << "-|";
    break;

  case 6:
    // cout << "[OperadorMulti] " << token << endl;
    archivo << "*|";
    break;

  case 7:
    // cout << "[OperadorDiv] " << token << endl;
    archivo << "/|";
    break;
```

```
  case 8:
    // cout << "[OperadorMod] " << token << endl;
    archivo << "%|";
    break;

  case 9:
    // cout << "[OperadorMayorQue] " << token << endl;
    archivo << ">|";
    break;

  case 10:
    // cout << "[OperadorMayorIgual] " << token << endl;
    archivo << ">=|";
    break;

  case 11:
    // cout << "[OperadorMenorQue] " << token << endl;
    archivo << "<|";
    break;

  case 12:
    // cout << "[OperadorMenorIgual] " << token << endl;
    archivo << "<=";
    break;

  case 13:
    // cout << "[OperadorAsignacion] " << token << endl;
    archivo << "=|";
    break;

  case 14:
    // cout << "[OperadorIgualdad] " << token << endl;
    archivo << "==|";
    break;

  case 15:
    // cout << "[OperadorNegacion] " << token << endl;
    archivo << "!|";
    break;

  case 16:
    // cout << "[OperadorDiferente] " << token << endl;
    archivo << "!=|";
    break;

  case 17:
    cout << "[Error] Operador no reconocido " << token << endl;
    archivo << "error|";
    error = true;
    break;

  case 18:
    // cout << "[OperadorY] " << token << endl;
    archivo << "and|";
    break;

  case 19:
    cout << "[Error] Operador no reconocido " << token << endl;
    error = true;
    break;
```

```
  case 20:
    // cout << "[OperadorO] " << token << endl;
    archivo << "or|";
    break;

  case 21:
    // cout << "[Delimitador] " << token << endl;
    archivo << ";|";
    break;

  case 22:
    // cout << "[Delimitador] " << token << endl;
    archivo << "(|";
    break;

  case 23:
    // cout << "[Delimitador] " << token << endl;
    archivo << ")|";
    break;

  case 24:
    // cout << "[Delimitador] " << token << endl;
    archivo << "{|";
    break;

  case 25:
    // cout << "[Delimitador] " << token << endl;
    archivo << "}|";
    break;

  case 26:
    // cout << "[Delimitador] " << token << endl;
    archivo << "[|";
    break;

  case 27:
    // cout << "[Delimitador] " << token << endl;
    archivo << "]|";
    break;

  case 28:
    // cout << "[Cadena] " << token << endl;
    archivo << "cadena|";
    break;

  // Palabras reservadas
  case 30:
    // cout << "[Palabra reservada] " << token << endl;
    archivo << "if|";
    break;

  case 31:
    // cout << "[Palabra reservada] " << token << endl;
    archivo << "else|";
    break;

  case 32:
    // cout << "[Palabra reservada] " << token << endl;
    archivo << "while|";
    break;
```

Formula Complejidad Ciclomática:

C(V) = Aristas – Nodos + 2
C(V) = 50 – 29 + 2
C(V) = 21



COCOMO

Tipo orgánico:

Tamaño	207
Tipo	Organic

$$PM_{nominal} = A_{PM} \cdot (KSLOC)^{B_{PM}}$$

PM	Organic	Semidetached	Embedded
A_{PM}	2.40	3.00	3.60
B_{PM}	1.05	1.12	1.20

$$TDEV = A_{TDEV}(PM)^{B_{TDEV}}$$

TDEV	Organic	Semidetached	Embedded
A_{TDEV}	2.50	2.50	2.50
B_{TDEV}	0.38	0.35	0.32

$$PM = PM_{nominal} \cdot \Pi_{i=1}^{15} EM_i$$

Parametros elegidos			
A_{PM}	2.40	A_{TDEV}	2.50
B_{PM}	1.05	B_{TDEV}	0.38

Esfuerzo	0.46	MM
Duracion	1.86	Meses
Team	0.25	Por persona

	Esfuerzo	Schedule
Planes y requisitos	0.0	0.2
Diseño	0.0	0.1
Desarrollo	0.3	1.2
Integracion y pruebas	0.1	0.5
Total	0.5	2.1

	Esfuerzo	Schedule
Planes y requisitos	6%	12%
Diseño	7%	8%
Desarrollo	62%	65%
Integracion y pruebas	31%	27%
Total	100%	100%

Tipo semi-acoplado:

Tamaño	207
Tipo	Semidetached

$$PM_{nominal} = A_{PM} \cdot (KSLOC)^{B_{PM}}$$

PM	Organic	Semidetached	Embedded
A_{PM}	2.40	3.00	3.60
B_{PM}	1.05	1.12	1.20

$$TDEV = A_{TDEV}(PM)^{B_{TDEV}}$$

TDEV	Organic	Semidetached	Embedded
A_{TDEV}	2.50	2.50	2.50
B_{TDEV}	0.38	0.35	0.32

$$PM = PM_{nominal} \cdot \Pi_{i=i}^{15} EM_i$$

Parametros elegidos			
A_{PM}	3.00	A_{TDEV}	2.50
B_{PM}	1.12	B_{TDEV}	0.35

Esfuerzo	0.51	MM
Duracion	1.98	Meses
Team	0.26	Por persona

	Esfuerzo	Schedule
Planes y requisitos	0.0	0.2
Diseño	0.0	0.2
Desarrollo	0.3	1.3
Integracion y pruebas	0.2	0.5
Total	**0.5**	**2.2**

	Esfuerzo	Schedule
Planes y requisitos	6%	12%
Diseño	7%	8%
Desarrollo	62%	65%
Integracion y pruebas	31%	27%
Total	**100%**	**100%**

Tipo empotrado:

Tamaño	207
Tipo	Embedded

$$PM_{nominal} = A_{PM} \cdot (KSLOC)^{B_{PM}}$$

PM	Organic	Semidetached	Embedded
A_{PM}	2.40	3.00	3.60
B_{PM}	1.05	1.12	1.20

$$TDEV = A_{TDEV}(PM)^{B_{TDEV}}$$

TDEV	Organic	Semidetached	Embedded
A_{TDEV}	2.50	2.50	2.50
B_{TDEV}	0.38	0.35	0.32

$$PM = PM_{nominal} \cdot \Pi_{i=i}^{15} EM_i$$

Parametros elegidos			
A_{PM}	3.60	A_{TDEV}	2.50
B_{PM}	1.20	B_{TDEV}	0.32

Esfuerzo	0.54	MM
Duracion	2.06	Meses
Team	0.26	Por persona

	Esfuerzo	Schedule
Planes y requisitos	0.0	0.2
Diseño	0.0	0.2
Desarrollo	0.3	1.3
Integracion y pruebas	0.2	0.6
Total	**0.6**	**2.3**

	Esfuerzo	Schedule
Planes y requisitos	6%	12%
Diseño	7%	8%
Desarrollo	62%	65%
Integracion y pruebas	31%	27%
Total	**100%**	**100%**

Pruebas Caja Negra y Caja Blanca

Caja Negra:

```
Ingrese el codigo (# para finalizar): int numero = 10;
string texto = "hola mundo";
int resultado = texto + numero;

#

[Error] Cadena detectada en operacion aritmetica

Presione una tecla para continuar . . .
```

Caja Blanca:

```
Ingrese el codigo (# para finalizar): int indice = 20;

if ( indice  {
  print("Hola Mundo");
}

#

[Error] Parentesis de cierre o operador logico no encontrado

Presione una tecla para continuar . . .
```

```
Ingrese el codigo (# para finalizar): int variable = 20;

while( variable2  {
  print("Hasta la proxima");
}
#

[Error] Identificador declarado mas de una vez: variable
[Error] Identificador no declarado: variable2
[Error] Parentesis de apertura no encontrado despues de while

Presione una tecla para continuar . . .
```