

Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería de Sistemas



Trabajos Prácticos 1 y 2

Diseño de Compiladores I

Ayudante a cargo: José A Fernández León

GRUPO 8

Martín Vazquez Arispe martin.vazquez.arispe@gmail.com

Joaquín Benecier joaquinbenecier@hotmail.com

Burckhardt David burck432@gmail.com

11/10/2023

Índice

Índice	2
Introducción	3
Temas particulares asignados	4
Analizador Léxico	5
Decisiones de diseño e implementación	5
Diagrama de transición de estados	6
Matriz de transición de estados	9
Matriz de acciones semánticas	10
Implementación	11
Tabla de Tokens	12
Acciones Semánticas	13
Errores Léxicos y otras consideraciones	14
Warnings	14
Errores	14
Analizador Sintáctico	15
Descripción de proceso de desarrollo	15
Uso de Yacc	15
Conflictos shift-reduce y reduce-reduce	16
Reduce-Reduce	16
Shift-Reduce	17
BNF	18
Lista de No-Terminales	19
Errores Sintácticos y otras consideraciones	20
Warnings	20
Errores	21
Verificaciones de rangos y signos	21
Constantes FLOAT válidos pero fuera de rango	22
Conclusiones	22

Introducción

El presente trabajo se centra en el diseño y desarrollo integral de un compilador en el lenguaje de programación C++. Este proyecto abarca la implementación de los analizadores tanto de la fase léxica como la sintáctica. El lenguaje a compilar se ha definido considerando características especiales y requerimientos específicos, proporcionados por la cátedra Diseño de Compiladores, se trata de un lenguaje inventado para este trabajo en particular. El objetivo principal es lograr un compilador robusto y eficiente que cumpla con los requerimientos y desempeño exigidos en el contexto académico, que logre la traducción de código fuente a código ejecutable.

Se aplicarán las técnicas y conocimientos adquiridos durante la cursada, así como el uso de herramientas y recursos clave, como "YACC" para la generación del analizador sintáctico. Además, se prestará especial atención a la interacción fluida entre el analizador léxico y sintáctico, garantizando un proceso de compilación coherente y preciso.

A lo largo de este informe, se detallarán los pasos, decisiones e implementaciones durante el desarrollo del compilador, proporcionando una visión completa del proceso y los resultados alcanzados. También, se abordarán de manera transparente los conflictos y desafíos surgidos en el camino, presentando soluciones y estrategias aplicadas para superarlos.

Temas particulares asignados

2. Enteros cortos (8 bits): Constantes enteras con valores entre -2^7 y 2^7-1 . Estas constantes llevarán el sufijo “_s”.

7. Punto Flotante de 32 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra “e” (mayúscula o minúscula) y el signo del exponente es obligatorio.

10. Operador especial: Incorporar a la lista de operadores, el operador aritmético --.

13. Palabras reservadas: Incorporar a la lista de palabras reservadas, las palabras WHILE y DO.

18. Herencia por Composición - Uso con nombre

Incorporar, como sentencia declarativa, la declaración de clases con la estructura mostrada en los siguientes ejemplos:

```

CLASS ca {
    INT a;c,    // declaración de atributos
    VOID m() { // declaración de método
        ...
    },
}

CLASS cb{
    FLOAT b,    // declaración de atributo
    FLOAT a,    // declaración de atributo
    VOID n() { // declaración de método
        ...
    },
    ca,          // nombre de clase
}
...

```

Incorporar, como sentencia declarativa, la declaración de objetos de una clase determinada:

```

ca a1; a2,
cb b1; b2; b3,

```

Incorporar, dentro de las sentencias ejecutables, la posibilidad de utilizar referencias a métodos y atributos de objetos indicando explícitamente cuando el atributo o método corresponden a la clase heredada por composición.

19. Declaración de métodos concentrada: La semántica de este tema se explicará en la etapa 3 del Trabajo Práctico

21. Forward declaration: Incorporar la posibilidad de declarar las clases con posterioridad a su uso.

24. Sobreescritura de atributos: La semántica de este tema se explicará en la etapa 3 del Trabajo Práctico.

26. Comprobaciones de uso de variables: Se debe permitir la declaración de funciones locales dentro de funciones locales (método de clase), con un solo nivel de anidamiento. Esto será chequeado en la etapa 3 del TP.

30. Conversiones Implícitas: Se explicará y resolverá en trabajos prácticos 3 y 4.

33. Comentarios multilinea: Comentarios que comiencen con “*{” y terminen con “}*” (estos comentarios pueden ocupar más de una línea).

35. Cadenas multilinea: Cadenas de caracteres que comiencen y terminen con “% ”. Estas cadenas pueden ocupar más de una línea. (En la Tabla de símbolos se guardará la cadena sin los saltos de línea).

Analizador Léxico

Decisiones de diseño e implementación

Para abordar el desarrollo del compilador, se optó, como se mencionó anteriormente, por trabajar en el lenguaje de programación C++. El manejo del archivo del código fuente a compilar se realiza directamente desde el main en el archivo "*Compiler.cpp*" encargado de cargar y verificar que existe el archivo a compilar. A continuación, se describen los demás componentes que se destinaron específicamente para el análisis léxico.

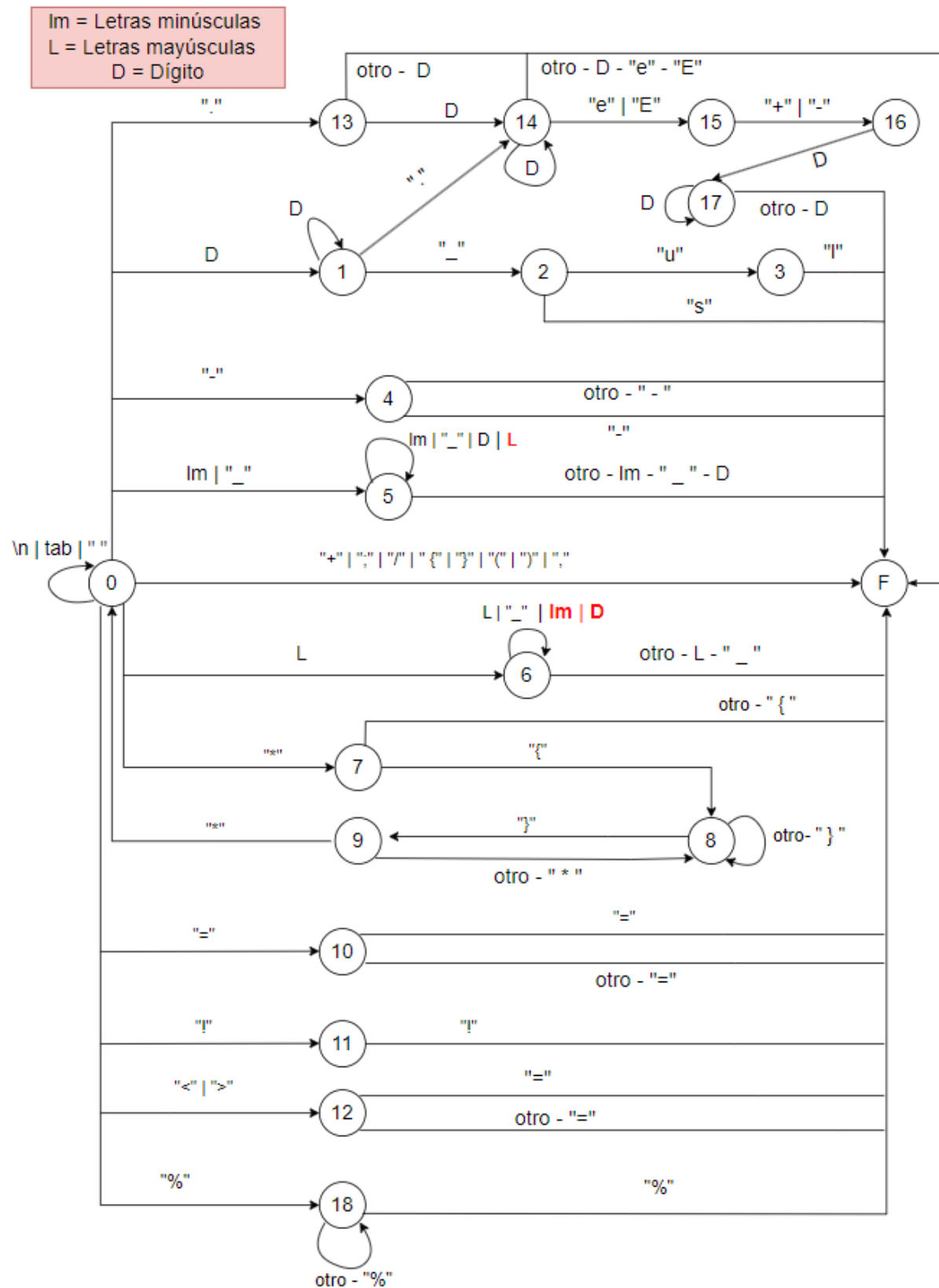
En primer lugar, se creó el archivo "*AnalizadorLexico*" enfocado al desarrollo de la primera etapa de compilación. Este archivo contiene un método denominado "yylex()" que, cada vez que se llama, escanea caracteres de un archivo de texto hasta identificar un token, devolviendo luego el número de token correspondiente. Esta tarea se logra mediante un diagrama de transición de estados implementado con una matriz de NxM, donde N representa la cantidad de estados y M la cantidad de símbolos posibles a leer. La implementación de esta matriz se detalla en una sección más adelante en este informe, pero en resumidas palabras, se trata de un arreglo bidimensional de celdas del tipo Struct denominado "*Celda*", conformadas por un entero referido al estado y un puntero a función para las acciones semánticas,

Para cambiar de estados en el análisis léxico, hemos utilizado una clase "*Automata*" donde se encuentra la matriz. Esta sentencia, al leer el siguiente símbolo en el código, determina el estado al que se debe cambiar y ejecuta la acción semántica correspondiente.

En cuanto a la gestión de la tabla de símbolos y la tabla de palabras reservadas, se optó por construir una clase independiente para cada una de ellas. Esto permite que tanto el analizador léxico como el sintáctico puedan acceder a estas estructuras de manera eficiente. Para ambas clases, la estructura principal es una tabla de hash (unordered_map). En el caso de la tabla de símbolos se ingresa al hash por la clave de tipo String que hace referencia al lexema, mientras que los atributos se encuentran almacenados en un struct "*Datos*" que contiene: el tipo, el valor de token en cuestión y un campo "*consultado*" para que posteriormente el analizador sintáctico verifique si es necesario o no agregar un signo negativo a la constante. Por otro lado, la tabla de palabras reservadas consta de un hash al cual también se accede por la clave de tipo String que se asocia en este caso a la palabra reservada en mayúscula, y el dato obtenido es un entero que indica el número de token.

En cuanto a las acciones semánticas, las hemos implementado a través de una clase llamada "*AccionesSemánticas*". Esta clase define un método "AS#", donde # referencia el número de la acción semántica. Cada uno de estos métodos recibe como parámetro una variable llamada "*carácter*" de tipo char. Cada acción semántica tiene su comportamiento específico y todas ellas se encuentran implementadas en el archivo "*AccionesSemanticas.cpp*". En una sección posterior se explica de que se encarga cada una de ellas.

Diagrama de transición de estados



Las correcciones realizadas al autómata original se resaltaron en rojo. Ahora se permite que al detectar el inicio de una palabra reservada, se puede ciclar y concatenar letras minúsculas y dígitos, al finalizar el token, se emitirá el error de que la palabra reservada no existe. Por otro lado al comenzar el token de un identificador, ahora es posible ciclar también letras mayúsculas, de ser este el caso, al finalizar se emitirá el error correspondiente dado que estos símbolos no se permiten en este tipo de tokens. Previamente a estos cambios el error estaba que al mínimo alternado entre letras mayúsculas o minúsculas el token finaliza y comienza otro, a continuación se muestran algunos ejemplos de la solución:

➤ Token: **SHORTaaa**

- Antes: **Correcto:** Token SHORT por un lado, Token aaa por otro lado
- Ahora: **ERROR:** La palabra reservada SHORTaaa no existe
-

➤ Token: **jcndULONG**

- Antes: **Correcto:** Token jcnd por un lado, Token ULONG por otro lado
- Ahora: **ERROR:** No se permiten identificadores con letras mayúsculas

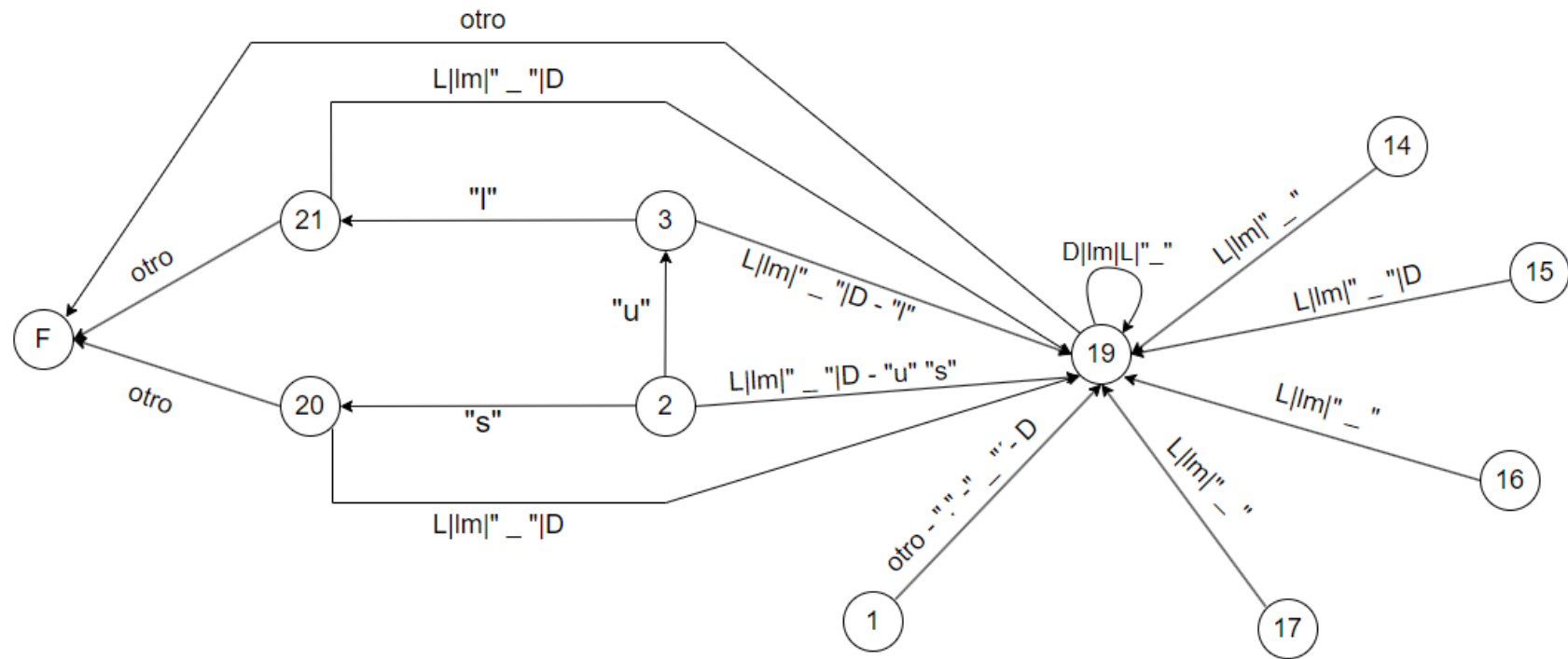
Prosiguiendo con este tipo de errores, se afinó el autómata de tal manera que algunos otros tokens como las constantes incluyendo su sufijo, puedan ser alterados por letras mayúsculas, minúsculas y dígitos sin detectar estas anomalías como tokens distintos, sino que se considera uno solo y se determina el error de acuerdo como haya querido comenzar el token. Por ejemplo

➤ Token: **24_sSHORT**

- Antes: **Correcto:** Token 24_s por un lado, Token SHORT por otro lado
- Ahora: **ERROR:** Constante mal definida

Se adjunta el autómata que refleja todos estos cambios, se añadieron tres nuevos estados, 19, 20 y 21 que permitieron aceptar todas estas combinaciones nuevas en un mismo token. Este nuevo autómata se debe ver como una extensión del anterior. Se dejó separado por cuestiones de legibilidad.

Todo lo referido a las constantes FLOAT corresponde con los tokens 1, 14, 15, 16 y 17. Las constantes SHORT y ULONG dependen de los estados 2 y 3 respectivamente.



En las siguientes secciones se muestran como quedan las nuevas matrices con estos cambios mencionados.

Matriz de transición de estados

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	D	Im	L	"_"	"."	"+"	"-"	"*"	"{"	"}"	"="	"!"	">" "<"	"%"	" " tab \n	";" "/" "(" ")" ","	"u"	"l"	"s"	"e"	"E"	otro	\$
0	1	5	6	5	13	F	4	7	F	F	10	11	12	18	0	F	5	5	5	5	6	X	F
1	1	19	19	2	14	X	X	X	X	X	X	X	X	X	X	X	19	19	19	19	19	X	X
2	19	19	19	19	X	X	X	X	X	X	X	X	X	X	X	X	3	19	20	19	19	X	X
3	19	19	19	19	X	X	X	X	X	X	X	X	X	X	X	X	19	21	19	19	19	X	X
4	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
5	5	5	5	5	F	F	F	F	F	F	F	F	F	F	F	F	5	5	5	5	5	F	F
6	6	6	6	6	F	F	F	F	F	F	F	F	F	F	F	F	6	6	6	6	6	X	F
7	F	F	F	F	F	F	F	F	8	F	F	F	F	F	F	F	F	F	F	F	F	F	F
8	8	8	8	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	8	8	8	8	X
9	8	8	8	8	8	8	8	0	8	8	8	8	8	8	8	8	8	8	8	8	8	8	X
10	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
11	X	X	X	X	X	X	X	X	X	X	X	F	X	X	X	X	X	X	X	X	X	X	X
12	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
13	14	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
14	14	19	19	19	F	F	F	F	F	F	F	F	F	F	F	F	19	19	19	15	15	X	F
15	19	19	19	19	X	16	16	X	X	X	X	X	X	X	X	X	19	19	19	19	19	X	X
16	17	19	19	19	X	X	X	X	X	X	X	X	X	X	X	X	19	19	19	19	19	X	X
17	17	19	19	19	F	F	F	F	F	F	F	F	F	F	F	F	19	19	19	19	19	X	F
18	18	18	18	18	18	18	18	18	18	18	18	18	18	F	18	18	18	18	18	18	18	18	X
19	19	19	19	19	F	F	F	F	F	F	F	F	F	F	F	F	19	19	19	19	19	F	F
20	19	19	19	19	F	F	F	F	F	F	F	F	F	F	F	F	19	19	19	19	19	F	F
21	19	19	19	19	F	F	F	F	F	F	F	F	F	F	F	F	19	19	19	19	19	F	F

Matriz de acciones semánticas

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	D	Im	L	"_"	"."	"+"	"-"	"**"	"{"	"}"	"="	"!"	">" "<"	"%"	" " tab \n	";" "/" "(" ")" ","	"u"	"l"	"s"	"e"	"E"	otro	\$
0	11	11	11	11	11	18	11	11	18	18	11	11	11	11	1	18	11	11	11	11	11	2	NULL
1	4	4	4	4	4	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	4	2	2
2	4	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	4	2	2
3	4	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	4	2	2
4	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
5	22	22	28	22	28	28	28	28	28	28	28	28	28	28	28	28	22	22	22	22	28	28	28
6	19	19	4	4	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	4	2	19
7	30	30	30	30	30	30	30	30	4	30	30	30	30	30	30	30	30	30	30	30	30	30	30
8	1	1	1	1	1	1	1	1	1	NULL	1	1	1	1	1	1	1	1	1	1	1	1	2
9	1	1	1	1	1	1	1	NULL	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
10	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
11	2	2	2	2	2	2	2	2	2	2	2	27	2	2	2	2	2	2	2	2	2	2	2
12	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
13	4	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
14	4	4	4	4	12	12	12	12	12	12	12	12	12	12	12	12	4	4	4	4	4	2	12
15	4	4	4	4	2	4	4	2	2	2	2	2	2	2	2	2	4	4	4	4	4	2	2
16	4	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	4	2	2
17	4	4	4	4	12	12	12	12	12	12	12	12	12	12	12	12	4	4	4	4	4	2	12
18	4	4	4	4	4	4	4	4	4	4	4	4	4	14	4	4	4	4	4	4	4	4	2
19	4	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	4	2	2
20	4	4	4	4	9	9	9	9	9	9	9	9	9	9	9	9	4	4	4	4	4	9	9
21	4	4	4	4	15	15	15	15	15	15	15	15	15	15	15	15	4	4	4	4	4	15	15

Implementación

En esta sección se abordará la implementación de la matriz de transición de estados del analizador léxico. Cada celda de esta matriz se representa mediante una estructura particular de C++, se optó por un struct llamado "*Celda*" con dos campos principales que facilitan la lógica del compilador. En primer lugar, se encuentra un entero denominado "*estadoSiguiente*" que representa el siguiente estado en el autómata al realizar una transición desde el estado actual. En segundo lugar, se emplea un puntero a una función denominado "*AS*" que indica la acción semántica que debe ejecutarse en la transición.

La matriz de transición, denominada "*AutomataMatriz*", se implementó como una matriz estática con una dimensión de 19 filas y 23 columnas. Esta matriz se cargó mediante una estrategia de hardcoding, lo que significa que se especificaron manualmente los valores de cada celda de la matriz en el código fuente. Se descartó la alternativa de utilizar un archivo externo para cargar la matriz, ya que no se consideró que aportará una ventaja significativa en términos de rendimiento o mantenibilidad.

Para mapear una celda específica en la matriz de transición, se requiere un índice que corresponde al estado actual y al símbolo de entrada. Para lograr esta correspondencia, se utilizó una estructura de datos adicional, un "*unordered_map*" de C++. Este mapa permite, dado un carácter de entrada, obtener el índice de columna que referencia la celda en la matriz. Esta estrategia de mapeo eficiente facilita la navegación y búsqueda en la matriz de transición durante el proceso de análisis léxico.

En el contexto del manejo de la matriz de transición de estados, se han implementado tres funciones clave. La primera de ellas, denominada "*pasoAutomata*", que desempeña un papel fundamental al buscar en la estructura alternativa de índices el valor que corresponde al carácter actual. Luego, actualiza la variable "*estadoActual*", lo que permite avanzar al siguiente estado en el procesamiento del próximo carácter. La segunda función, llamada "*reiniciarRecorrido*," tiene una funcionalidad sencilla y su nombre refleja claramente su propósito. Su tarea principal es restablecer la variable "*estadoActual*" a 0, lo que indica el inicio de la lectura de un nuevo token o señala que se ha finalizado la lectura del archivo. Por último, la tercera función, "*indiceCaracter*," es un poco más compleja y se ha diseñado para abordar la necesidad de dividir las columnas de la matriz en conjuntos de símbolos en lugar de tener una columna individual en la matriz para cada símbolo de teclado. Esta función utiliza las funciones de C++ como "*isalpha*," "*isupper*," y "*isdigit*" para filtrar los caracteres, lo que permite hacer referencia a una sola columna de la matriz para un grupo particular de símbolos. Estas funciones pertenecen a la biblioteca estándar de C y C++ y permiten determinar si un carácter dado es una letra del alfabeto, una letra mayúscula o un dígito respectivamente. Esta decisión de implementación logró simplificar la gestión de la matriz y mejorar la eficiencia del análisis léxico al agrupar símbolos relacionados en una única columna de la matriz de transición de estados.

Tabla de Tokens

A continuación se presenta una tabla de tokens que ofrece una visión detallada de los elementos clave del análisis léxico. Se destacan tres columnas esenciales: el nombre del token, su número correspondiente y una indicación que revela si el lexema asociado es único o si no lo es. A través de esta tabla, se proporciona una representación estructurada de los tokens que podrían presentarse en esta primera etapa de compilación, lo que facilita su comprensión y el seguimiento de su ocurrencia a lo largo del código.

Nombre	N°	Lexema Único	Nombre	N°	Lexema Único
IF	257	No	"}"	125	No
ELSE	258	No	"("	40	No
END_IF	259	No	")"	41	No
PRINT	260	No	","	44	No
CLASS	261	No	."	46	No
VOID	262	No	"="	61	No
SHORT	263	No	"=="	269	No
ULONG	264	No	"!!"	270	No
FLOAT	265	No	">"	62	No
WHILE	266	No	"<"	60	No
DO	267	No	">="	271	No
"+"	43	No	"<="	272	No
"_"	45	No	CTE-SHORT	273	Si
"_"	268	No	CTE-ULONG	274	Si
"*"	42	No	CTE-FLOAT	275	Si
"/"	47	No	CTE-CADENA	276	Si
","	59	No	IDENTIFICADOR	277	Si
"{"	123	No	RETURN	278	No

Como se puede observar, se han asignado valores numéricos considerando códigos ASCII para símbolos comunes de teclado, como "=", "+", etc. Esta estrategia facilita la identificación durante el análisis léxico. Por ejemplo, el token "=" se asocia con el valor 61, su correspondiente código ASCII. Para tokens compuestos, palabras reservadas, constantes e identificadores, se inicia la asignación desde 257 para evitar solapamientos con ASCII, garantizando de esta forma, una identificación única. Esta decisión simplifica la gestión de tokens especiales y asegura una adaptabilidad eficiente a las particularidades del lenguaje definido. Un ejemplo de esto podrían ser los tokens "==" ó "!!" a los que se les asignó el código 269 y 270 respectivamente.

Acciones Semánticas

Durante el desarrollo del analizador léxico, surgen las acciones semánticas, operaciones ejecutadas al identificar patrones de caracteres formando tokens válidos. Estas acciones realizan tareas semánticas, como completar la tabla de símbolos, contar líneas, entre otras. En nuestro compilador, las acciones semánticas definidas incluyen:

- **AS1:** Se encarga de llevar el conteo de las líneas del programa.
- **AS2:** Esta acción permite informar acerca de un error léxico ocurrido.
- **AS3:** Destinada a inicializar la variable que almacena el string referido al token.
- **AS4:** Concatena el nuevo carácter a la cadena que se tenga hasta el momento. mientras no sea un salto de línea. Si lo es, invoca AS1 para que realice el conteo.
- **AS5:** Bloquea la lectura del próximo carácter. Permite finalizar la lectura de tokens.
- **AS6:** Agrega un identificador a la tabla de símbolos.
- **AS7:** Chequea rango Float, si es correcto llama a AS25.
- **AS8:** Chequea rango Ulong, si no hay overflow llama a AS23 y AS24.
- **AS9:** Chequea rango Short, si hay overflow llama a AS10 y sino a AS14.
- **AS10:** Reinicia el estado actual a 0 para leer el próximo token.
- **AS11:** Ejecuta acciones semánticas AS3 y AS4, coloca el número de línea que inició el nuevo token.
- **AS12:** Ejecuta acciones semánticas AS10, AS5, AS7.
- **AS13:** Ejecuta acciones semánticas AS5 y AS10.
- **AS14:** Ejecuta acciones semánticas AS4, AS10, AS23 y AS24.
- **AS15:** Ejecuta acciones semánticas AS5, AS8 y AS10.
- **AS16:** Ejecuta AS4 si es que la palabra reservada no se ha concretado aún.
- **AS17:** Ejecuta acciones semánticas AS5, AS6, AS10.
- **AS18:** Transforma el carácter a su valor ASCII correspondiente y ejecuta AS10.
- **AS19:** Busca el token en la tabla de palabras reservadas y ejecuta AS13.
- **AS20:** Verifica si se trata de un comparador “==” y ejecuta AS10 o AS13 si es “=”.
- **AS21:** Verifica si se trata de un operador especial “--” y ejecuta AS10 o AS13 si es “_”.
- **AS22:** Verifica que los identificadores no excedan los 20 caracteres y ejecuta AS4 de ser posible, caso contrario, se emite un warning.
- **AS23:** Agrega una entrada en la tabla de símbolos con su valor y tipo correspondiente.
- **AS24:** Devuelve el número de token, de una constante SHORT, ULONG, o STRING.
- **AS25:** Agrega en la tabla de símbolos un flotante y retorna el token.
- **AS26:** Devuelve los tokens de los símbolos: “>=”, “<=”, “<”, “>” según corresponda y si es “>=” o “<=” ejecuta la AS10, sino la AS13.
- **AS27:** Devuelve el token del símbolo “!” y ejecuta AS10.
- **AS28:** Ejecuta AS17 y devuelve token de identificador.
- **AS29:** Revisa si se trata de un punto o de una constante float y ejecuta la AS12 o las acciones AS5 y AS18 según corresponda.
- **AS30:** Ejecuta AS5 y AS18 .

Errores Léxicos y otras consideraciones

En el análisis léxico pueden surgir errores que representan situaciones en las que el proceso de lectura y reconocimiento de tokens en un programa no cumple con las reglas o formatos definidos por el lenguaje de programación. Estos errores son cruciales en la detección temprana de problemas en el código fuente y permiten garantizar que el programa se ejecute de manera adecuada. En esta sección, se enuncian los errores léxicos tratados y los warnings referidos al compilador en desarrollo:

Warnings

Cada uno de los warnings consta de un mensaje que señala la anomalía detectada y la línea donde se encuentra. En el caso del analizador léxico sólo se incorporó un *warning*, diferente al caso del análisis sintáctico, donde se presentan varios avisos de este tipo.

- Se truncarán los identificadores que superen los 20 caracteres. En este caso, continúa consumiendo los caracteres hasta terminar el identificador pero solo se guardan en la tabla de símbolos los primeros 20 caracteres.

Errores

Para los errores se emite un único mensaje que indica la línea donde se encuentra el problema. En algunos errores específicos se expresa un mensaje más detallado. Además de estos avisos, se ha incorporado una estrategia de **"modo pánico"** para gestionar eficientemente situaciones de error. En este enfoque, luego de la detección de un error léxico en la identificación de un token, el analizador se reinicia desde el estado 0, iniciando así la búsqueda y reconocimiento de un nuevo token. Este reinicio es fundamental para evitar que un error en la interpretación de un símbolo comprometa la validez del análisis en curso. El uso de esta técnica mejora la tolerancia a errores y la capacidad del compilador para recuperarse de situaciones problemáticas. Los siguientes son algunos de los errores identificados:

- Constantes SHORT y ULONG sin el sufijo indicado en el enunciado.
- Constantes FLOAT con un formato invalido o mal definidas.
- Comentarios que no terminan con `"/*"`.
- Identificadores con símbolos inválidos como por ejemplo: `@`.
- Palabras reservadas mal escritas o inexistentes.
- Constantes fuera de rango.
- Uso de carácter no aceptado por el lenguaje.
- Utilizar un único símbolo `"!"` sin acompañarlo de esta forma: `"!!"`.
- Una cadena que no comience y termine con `"%"`.

Analizador Sintáctico

Descripción de proceso de desarrollo

Uso de Yacc

En el proceso de desarrollo del compilador, se utilizó la herramienta YACC (Yet Another Compiler Compiler) para abordar la implementación de la parte sintáctica de manera eficiente y estructurada. YACC es una herramienta de generación de analizadores sintácticos que desempeñó un papel fundamental en la simplificación de esta tarea facilitando la especificación gramatical del lenguaje de programación mediante la definición de reglas gramaticales en un formato específico. Esta herramienta, procesa la gramática proporcionada y genera automáticamente el código del analizador sintáctico correspondiente.

En la estructura generada se pueden destacar dos elementos clave por su relevancia en el proceso de análisis sintáctico: **yyparse()** y **yylval**. Estos componentes desempeñan funciones esenciales en la interpretación y manipulación de la gramática definida dado que el primero de ellos es el motor principal del análisis sintáctico, mientras que el segundo resulta fundamental para la transferencia y manipulación de información.

Cuando se invoca `yyparse()`, el analizador sintáctico comienza a procesar el flujo de tokens proporcionado por el analizador léxico. Durante este proceso, `yyparse()` utiliza la gramática definida en las reglas de YACC para realizar el análisis sintáctico del código fuente. En términos generales, `yyparse()` controla la ejecución del analizador sintáctico y, a medida que avanza en el código fuente, lleva a cabo acciones semánticas según las reglas definidas en el conjunto de instrucciones de YACC. Estas acciones pueden incluir asignaciones de valores a la variable `yylval` y otras operaciones pertinentes. Este método devuelve un valor que indica el resultado del análisis. Comúnmente, se retorna 0 si el análisis sintáctico fue exitoso y 1 para indicar fallos o errores específicos.

En cuanto a `yylval`, es utilizada para almacenar temporalmente el valor asociado a un nodo en el árbol de sintaxis abstracta durante el proceso de análisis. En nuestro caso se utiliza de manera destacada para almacenar información específica de la tabla de símbolos. Esta variable se asigna en las acciones semánticas definidas en las reglas gramaticales.

La forma en que `yylval` se utiliza y asigna depende de las acciones semánticas definidas en el archivo de especificación YACC. Puede contener valores simples, apuntadores a estructuras de datos, o cualquier información necesaria para el análisis.

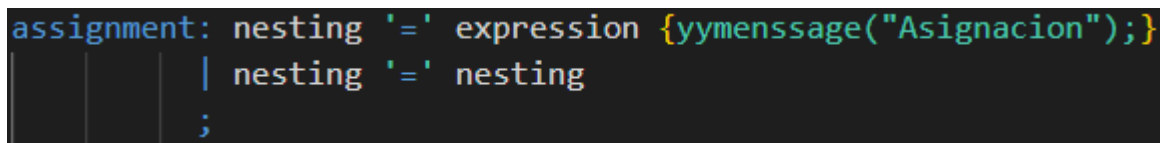
Conflictos shift-reduce y reduce-reduce

En el contexto de los analizadores sintácticos generados por herramientas como YACC, es fundamental comprender dos tipos de conflictos que pueden surgir durante el proceso de análisis: **shift-reduce** y **reduce-reduce**. Estos conflictos se originan en situaciones donde el analizador sintáctico tiene múltiples opciones para realizar acciones, lo que puede generar ambigüedades en la derivación de la gramática. Los conflictos shift-reduce ocurren cuando el analizador tiene la opción de continuar leyendo símbolos (shift) o reducirlos a una producción gramatical. Por otro lado, los conflictos reduce-reduce se presentan cuando hay varias producciones gramaticales que podrían aplicarse en una determinada situación.

En el caso del presente trabajo, cada uno de estos conflictos fue analizado y resuelto para garantizar la coherencia y la precisión en la interpretación gramatical del código fuente. A continuación se presentarán ejemplos representativos de cada tipo de conflicto, detallando las estrategias específicas implementadas para su resolución. Cabe destacar, que estos ejemplos, fueron conflictos reales que surgieron durante el desarrollo.

Reduce-Reduce

En este caso propuesto, el conflicto se da en la asignación. *Ver imagen 1*



```
assignment: nesting '=' expression {yymessage("Asignacion");}
          | nesting '=' nesting
          ;
```

Imagen 1: Regla gramatical asociada a las asignaciones con conflicto reduce-reduce

La producción `assignment` presenta un conflicto *reduce-reduce* debido a la ambigüedad en la interpretación de la secuencia ***nesting* '=' *nesting***. Este conflicto se origina porque el analizador sintáctico tiene varias producciones gramaticales que podrían aplicarse en una determinada situación. Un ejemplo se observa en el siguiente código:

```
{ var_2 = var_1, }
```

En esta situación, el analizador sintáctico se enfrenta a la elección entre dos producciones gramaticales posibles: realizar una reducción de la secuencia ***nesting* '=' *nesting*** a la producción `assignment`, o reducir la secuencia ***nesting* '=' *expression*** a la misma producción.

Para resolver este conflicto, se implementó una estrategia que prioriza la reducción de la secuencia ***nesting* '=' *expression***, considerando la interpretación estándar de una asignación con una expresión a la derecha del operador "=". Esto asegura una interpretación coherente y precisa de la gramática en presencia de expresiones de asignación. La reducción ***nesting* '=' *nesting*** se eliminó y se modificó la gramática de forma tal que *expression* pueda obtenerse a partir de un *nesting*. Esto involucró los respectivos cambios en la regla gramatical referida al *factor* quedando de la siguiente manera. *Ver imágenes 2, 3 y 4.*


```
assignment: nesting '=' expression {yymessage("Asignacion");}
          ;
```

Imagen 2: Resolución de conflicto en regla gramatical referida a la asignación

```
factor: nesting
      | constant
      | nesting LESSLESS
      ;
```

Imagen 3: Regla gramatical asociada al factor

```
nesting: nesting '.' ID
        | ID
        ;
```

Imagen 4: Regla gramatical asociada al anidamiento

Shift-Reduce

Para el siguiente caso elegido, el conflicto se da en regla gramatical referida al retorno de las funciones. Ver imagen 5

```
return: RETURN ','
       | RETURN {yyerror("Falta , al final del RETURN");}
       ;
```

Imagen 5: Regla gramatical referida al retorno con conflicto shift-reduce

La producción return presenta un conflicto shift-reduce debido a la ambigüedad en la interpretación de la presencia de una coma (',') antes del final de la declaración de retorno. Este conflicto se origina porque el analizador sintáctico no puede optar entre realizar un desplazamiento (shift) para leer la coma o reducir la secuencia a la producción *return*. Se muestra un ejemplo del conflicto.

{ RETURN , }

En este caso, el analizador sintáctico debe decidir entre realizar un desplazamiento (shift) para leer la coma o reducir la secuencia RETURN a la producción *return*.

Para resolver este conflicto, se implementó una estrategia que prioriza el desplazamiento (shift) para leer "RETURN ," como parte de una expresión de retorno. Para esto se eliminó la posibilidad de tener RETURN sin coma(,). Se adjunta imagen del resultado. Ver imagen 6

```
return: RETURN ','
       ;
```

Imagen 6: Resolución de conflicto en regla gramatical referida a los retornos

BNF

1. `<program> ::= '{' <sentence_list> '}'`
2. `<sentence_list> ::= <sentence_list> <sentence> | <sentence>`
3. `<sentence> ::= <declarative> ',' | <executable> ','`
4. `<declarative> ::= <function> | <declaration> | <class>`
5. `<executable_list> ::= <executable_list> <executable> ',' | <executable> ','`
6. `<executable> ::= <if> | <while> | <print> | <function_call> | <assignment>`
7. `<declaration> ::= <variable> | <object>`
8. `<variable> ::= <type> <variable_list>`
9. `<object> ::= 'ID' <object_list>`
10. `<object_list> ::= <object_list> ';' 'ID' | 'ID'`
11. `<variable_list> ::= <variable_list> ';' 'ID' | 'ID'`
12. `<assignment> ::= <nesting> '=' <expression>`
13. `<nesting> ::= <nesting> '.' 'ID' | 'ID'`
14. `<function> ::= <function_header> '{' <function_body> '}'`
15. `<function_header> ::= 'VOID' 'ID' '(' <formal_parameter> ')' | 'VOID' 'ID' '(' ')'`
16. `<function_body> ::= <sentence_list> <return> | <return>`
17. `<formal_parameter> ::= <type> 'ID'`
18. `<function_call> ::= <nesting> '(' <real_parameter> ')' | <nesting> '(' ')'`
19. `<real_parameter> ::= <expression>`
20. `<if> ::= 'IF' <condition> <iterative_body> <else> <iterative_body> 'END_IF' | 'IF' <condition> <iterative_body> 'END_IF'`
21. `<else> ::= 'ELSE'`
22. `<while> ::= <while_header> <condition> 'DO' <iterative_body>`
23. `<while_header> ::= 'WHILE'`
24. `<iterative_body> ::= '{' <executable_list> '}' | '{' <executable_list> <return> '}' | <executable> ',' | <return>`
25. `<condition> ::= '(' <comparison> ')'`
26. `<comparison> ::= <expression> <logic_operator> <expression>`
27. `<logic_operator> ::= '==' | '!=' | '>=' | '<=' | '<' | '>'`
28. `<class> ::= <class_header> '{' <sentence_list> '}' | <class_header> '{' <sentence_list> <heredity> '}' | <class_header>`
29. `<class_header> ::= 'CLASS' 'ID'`
30. `<heredity> ::= 'ID' ','`
31. `<expression> ::= <expression> '+' <termino> | <expression> '-' <termino> | <termino>`
32. `<termino> ::= <termino> '*' <factor> | <termino> '/' <factor> | <factor>`
33. `<factor> ::= <nesting> | <constant> | <lessless>`
34. `<lessless> ::= <nesting> '--'`
35. `<constant> ::= 'CTESHORT' | '-' 'CTESHORT' | 'CTEFLOAT' | '-' 'CTEFLOAT' | 'CTEULONG'`
36. `<type> ::= 'SHORT' | 'ULONG' | 'FLOAT'`
37. `<print> ::= 'PRINT' <cadena>`
38. `<cadena> ::= 'CTESTRING'`
39. `<return> ::= 'RETURN' ','`

Lista de No-Terminales

- **program**: Representa la estructura principal de un programa, incluyendo las sentencias que lo conforman encerradas entre llaves.
- **sentenceList**: Una lista de sentencias, tanto declarativas como ejecutables.
- **sentence**: Una sentencia que puede ser declarativa o ejecutable. Finaliza con “;”.
- **declarative**: Puede representar una función, una declaración o una clase.
- **executableList**: Una lista de sentencias ejecutables.
- **executable**: Una sentencia ejecutable, que puede ser una estructura de control (if, while), una impresión, una llamada a función o una asignación.
- **declaration**: Puede ser una declaración de variable o de objeto.
- **variableDeclaration**: Una declaración que incluye el tipo y una lista de variables.
- **objectDeclaration**: Una declaración que incluye el tipo y una lista de objetos.
- **objectList**: Lista de objetos, que puede ser una sola o múltiples separadas por “;”
- **variableList**: Lista de variables, que puede ser una sola o múltiples separadas por “;”
- **assignment**: Una asignación, que implica una expresión y una estructura de anidación (*nesting*).
- **nesting**: Puede ser un identificador o una estructura de anidación, que es utilizada para acceder a atributos y métodos de objetos.
- **function**: La definición de una función, que puede o no tener parámetros formales.
- **functionBody**: El cuerpo de la función, son sentencias seguidas de un return al final.
- **functionHeader**: Definición del encabezado de una función, incluido parámetros.
- **formalParameter**: Los parámetros formales de una función, incluyen el tipo y el ID.
- **functionCall**: Llamada a una función, que puede o no tener parámetros reales.
- **realParameter**: Parámetro real de una función, es una expresión.
- **ifStatement**: Una estructura condicional *if* que puede incluir un bloque *else*.
- **whileStatement**: Una estructura de bucle, que incluye una condición y un cuerpo.
- **while**: Declaración de la palabra reservada WHILE.
- **iterativeBody**: El cuerpo de un *if* o *while*, se compone de sentencias ejecutables.
- **condition**: Una condición para las estructuras de control, involucra una comparación.
- **class**: La definición de una clase.
- **classHeader**: Definición del encabezado de la clase.
- **heredity**: Permite indicar que una clase hereda de otra.
- **comparison**: Representa una comparación lógica entre factores.
- **expression**: Una expresión matemática que puede incluir términos y operadores.
- **termino**: Un término en una expresión matemática, incluye factores y operadores.
- **factor**: Un factor puede ser una constante o una anidación(*nesting*) con o sin “--”.
- **lessless**: Una anidación que tiene el operador especial “--”.
- **operatorsLogics**: Operadores lógicos utilizados en las comparaciones.
- **constant**: Una constante, que puede ser de tipo SHORT, ULONG o FLOAT.
- **type**: El tipo de dato, que puede ser SHORT, ULONG o FLOAT.
- **print**: Una sentencia ejecutable de impresión que incluye una cadena.
- **cadena**: Una cadena de texto utilizada en la sentencia de impresión.
- **return**: Una sentencia de retorno utilizada al final de las funciones.

Errores Sintácticos y otras consideraciones

Durante el proceso de análisis sintáctico, el compilador se enfrenta a una importante tarea, la verificación de la estructura gramatical del código fuente. En este contexto, la detección de errores sintácticos es esencial para garantizar la validez del programa y facilitar su comprensión por parte de los desarrolladores. En este informe, se abordan los distintos tipos de errores detectados por el analizador sintáctico del compilador, así como las estrategias implementadas para informarlos y, cuando sea posible, corregirlos. Se emplean mensajes claros y detallados, acompañados de advertencias para orientar al programador y facilitar la depuración del código. A continuación, se detallan algunas de las funciones clave implementadas para gestionar estos mensajes.

- **yymessage**(*string message*): Esta función se encarga de mostrar mensajes informativos. Utilizada para comunicar la detección de estructuras sintácticas en el código fuente.
- **yyerror**(*string message*): La función *yyerror* se activa al detectar un error sintáctico y muestra un mensaje detallado junto con la línea donde ocurrió el mismo.
- **yywarning**(*string message*): Se utiliza para emitir advertencias durante el análisis sintáctico. Informan al programador sobre posibles problemas o prácticas que podrían generar resultados inesperados.
- **chequearRangoSHORT**(*string value*): Esta función evalúa si una constante está dentro del rango permitido. En caso contrario, emite un mensaje de error indicando que la constante está fuera del rango aceptado.

Avanzando en esta sección, se procede a señalar los errores y warnings que el analizador sintáctico es capaz de detectar durante el análisis del código fuente.

Warnings

Estos mensajes presentan advertencias que alertan sobre situaciones que, aunque no impiden la ejecución, sugieren prácticas o estructuras que podrían requerir atención.

- **yywarning**("Programa vacío;"): Advierte que el programa está vacío al encontrar llaves sin contenido.
- **yywarning**("Función vacía;"): Advierte que una función está definida sin contenido en su cuerpo. Solo se encuentra la sentencia "RETURN;".
- **yywarning**("If vacío;"): Advierte sobre la presencia de un "if" sin sentencias.
- **yywarning**("Else vacío;"): Advierte que el bloque "else" está vacío en una estructura "if-else".
- **yywarning**("Bloque vacío;"): Advierte la presencia de un bloque vacío, como en una estructura condicional o bucle. Destinado al caso de sentencias entre llaves "{}".

Errores

Los siguientes mensajes, identificados como *yyerror*, indican problemas en la estructura del programa que pueden afectar la correcta interpretación del código, proporcionando información crucial para la corrección temprana de posibles errores.

- **yyerror**("Falta llaves delimitadores de programa"): Indica un error al no encontrar las llaves delimitadoras del programa.
- **yyerror**("Sentencia declarativa en lugar de una ejecutable"): Señala un error al detectar una sentencia declarativa en lugar de una ejecutable.
- **yyerror**("Falta segundo paréntesis en la condición"): Indica un error al no encontrar el segundo paréntesis en una condición.
- **yyerror**("Falta primer paréntesis en la condición"): Indica un error al no encontrar el primer paréntesis en una condición.
- **yyerror**("Faltan paréntesis en la condición"): Indica un error al encontrar una condición sin paréntesis adecuados.
- **yyerror**("Una constante ULONG no puede ser negativa"): Indica un error al encontrar una constante ULONG negativa, lo cual no es válido.
- **yyerror**("Constante SHORT fuera de rango"): Indica un error al encontrar una constante SHORT negativa que exceda los límites permitidos. ($x < -128_s$)
- **yyerror**("Función sin RETURN obligatorio al final"): Todas las funciones y métodos deben terminar con una sentencia RETURN.
- **yyerror**("La herencia debe ir al final de la declaración de la clase"): La herencia en una clase se debe detallar en la última línea de la declaración.
- **yyerror**("Expresión no puede ir entre paréntesis"): Las expresiones deben ir sin paréntesis.
- **yyerror**("Se ha detectado una falta de coma"): Al finalizar una sentencia no se ha colocado la coma que las delimita.

Verificaciones de rangos y signos

Por último, la función *chequearRangoSHORT* y llamados a la Tabla de Símbolos son fundamentales para garantizar la coherencia de las constantes en el análisis sintáctico.

En el caso de las constantes SHORT, se utiliza *chequearRangoSHORT* para verificar que estén dentro del rango permitido. En el caso de las constantes FLOAT, no es necesario realizar una verificación en el análisis sintáctico debido a que, por su naturaleza, no presentan cambios en el rango al cambiar de signo. Por otro lado, las constantes ULONG se someten a una verificación de rango en el análisis léxico, lo que hace innecesario repetir esta validación en la segunda etapa de compilación.

Los llamados a la Tabla de Símbolos se efectúan para gestionar la negatividad de las constantes, ya sea por el signo '-' en las constantes SHORT y FLOAT o por la restricción de negatividad en constantes ULONG.

Estas validaciones contribuyen a mantener la integridad semántica del código fuente. Aquí se presentan estas verificaciones, cabe aclarar que aplican notación posicional.

- `'CTESHORT {TablaDeSimbolos::chequearNegativos($2);}`
- `'CTEFLOAT {TablaDeSimbolos::chequearNegativos($2);}`
- `CTEFLOAT {TablaDeSimbolos::chequearPositivos($1);}`
- `CTESHORT {chequearRangoSHORT($1);}`

Constantes FLOAT válidos pero fuera de rango

Inicialmente el rango delimitado para las constantes flotantes en el enunciado del trabajo práctico es el siguiente:

$$1.17549435E-38 < x < 3.40282347E+38 \text{ U } -1.17549435E-38 < x < -3.40282347E+38 \text{ U } 0.0$$

En C++ a pesar de intentar comparar el valor flotante con el límite inferior 1.17549435E-38 Se toman como inválidos los siguientes siete valores que deberían ser válidos:

- 1.17549436E-38
- 1.17549437E-38
- 1.17549438E-38
- 1.17549439E-38
- 1.17549440E-38
- 1.17549441E-38
- 1.17549442E-38

El problema surge debido a lo pequeños que son los números a comparar, el lenguaje hace un recorte en la cantidad de decimales al intentar chequear estos límites. Con los máximos pasa lo mismo. Existen valores flotantes menores al máximo permitido que se toman como inválidos debido a un problema en la comparación con números tan grandes. Existen doce valores menores a 3.40282347E+38 que se toman como inválidos pero deberían ser válidos, estos son:

- 3.40282346E+38
- 3.40282345E+38
- 3.40282344E+38
- 3.40282343E+38
- 3.40282342E+38
- 3.40282341E+38
- 3.40282340E+38
- 3.40282339E+38
- 3.40282338E+38
- 3.40282337E+38
- 3.40282336E+38
- 3.40282335E+38

El rango real finalmente que deben cumplir las constantes flotantes debido a estas limitaciones del lenguaje C++ es:

$$1.17549442\text{E-}38 < x < 3.40282335\text{E+}38$$

Problema de coma faltante

Se solucionó el problema de la coma faltante en las sentencias de tal manera que siga compilando correctamente el código completo. Los cambios de la gramática referidos se muestran a continuación.

```
sentence: declarative', '  
        | executable', '  
        | declarative {yyerror("Se ha detectado una falta de coma");}  
        | executable {yyerror("Se ha detectado una falta de coma");}  
        ;
```

Imagen 7: Resolución de conflicto de coma faltante

Principalmente las sentencias declarativas y ejecutables que no tengan coma derivan en la emisión de un error que indica “*Se ha detectado una falta de coma*”. De esta manera el usuario podrá detectar a qué sentencias les falta este símbolo además del resto de los errores que pueda tener el código fuente.

Las excepciones donde el compilador no aceptará este tipo de faltas es dentro de las sentencias iterativas IF y WHILE ni tampoco en las sentencias de tipo RETURN.

Conclusiones

En el transcurso del desarrollo de este compilador, hemos abordado con éxito dos, de las tres etapas cruciales que se van a trabajar: el análisis léxico y el análisis sintáctico. El lenguaje de programación C++ fue elegido como el marco para la implementación, y representó un interesante desafío para poner a prueba las técnicas y conocimientos adquiridos durante la cursada en Diseño de Compiladores.

En la fase de análisis léxico, se diseñó un robusto analizador basado en autómatas finitos y matrices de transición de estados. La implementación de la matriz de transición, fundamentada en la eficiencia y la claridad lógica, se llevó a cabo con éxito, permitiendo la identificación precisa de tokens y su adecuada clasificación. Además, la gestión de la tabla de símbolos y palabras reservadas se implementaron con estructuras de datos eficientes, como lo son los mapas de C++, garantizando un acceso rápido y preciso.

En la segunda etapa del desarrollo, la herramienta YACC, propuesta por la cátedra, demostró ser esencial para el análisis sintáctico. Facilitó la especificación gramatical del lenguaje y generó automáticamente el código del analizador sintáctico. Destacamos dos elementos cruciales en este proceso: `yyparse()` y `yylval`. El primero, como motor principal del análisis sintáctico, comanda la ejecución del analizador y lleva a cabo acciones semánticas según las reglas definidas. El segundo, `yylval`, desempeña un papel destacado al almacenar temporalmente valores asociados a nodos en la estructura de la tabla de símbolos, enriqueciendo así el análisis semántico.

En conjunto, estas etapas del desarrollo nos han llevado a la creación de un compilador robusto y eficiente al que todavía le queda trabajo por delante. Hasta el momento, este proyecto no solo representa una respuesta satisfactoria a los desafíos académicos planteados, sino que también destaca la aplicación exitosa de conocimientos teóricos en el ámbito práctico del diseño de compiladores.