Compiladores e Intérpretes

Trabajos Prácticos 1 y 2

Analizador Léxico y Analizador Sintactico

Grupo 15

Integrantes:
Arriaga Maria Eugenia marriaga@alumnos.exa.unicen.edu.ar
Lopez Milagros milopez@alumnos.exa.unicen.edu.ar
Rizzalli Bianca brizzalli@alumnos.exa.unicen.edu.ar

Profesora asignada:
Marcela Ridao

Temas asignados:
4 6 7 12 17 23 26 27 30 33

Introducción

El presente informe desarrolla el proceso de construcción de las dos primeras etapas de un compilador: el analizador léxico (Trabajo Práctico Nº1) y el analizador sintáctico (Trabajo Práctico Nº2).

En la primera etapa se implementó un analizador léxico en Java, encargado de leer el código fuente y dividirlo en unidades léxicas o tokens, verificando la validez de identificadores, palabras reservadas, operadores y constantes numéricas, y registrando los símbolos reconocidos en una tabla de símbolos. También se implementaron las acciones semánticas asociadas al autómata que describe las transiciones, las cuales permiten validar rangos, construir lexemas y manejar errores léxicos.

En la segunda etapa se desarrolló un analizador sintáctico utilizando YACC, que recibe los tokens generados por el analizador léxico y comprueba si el código fuente cumple con la sintaxis definida por la gramática del lenguaje. Para ello se diseñó y depuró una gramática libre de conflictos *shift-reduce* y *reduce-reduce*, incorporando además la detección y recuperación de errores sintácticos.

Ambos módulos trabajan de forma integrada, permitiendo analizar un programa fuente y producir como salida los tokens reconocidos, las estructuras sintácticas detectadas, los errores léxicos y sintácticos encontrados, y el contenido final de la tabla de símbolos.

Analizador Léxico

Temas asignados

4. Enteros largos sin signo (32 bits): Constantes enteras con valores entre 0 y 232 – 1 que se escriben como una secuencia de dígitos seguidos del sufijo UL. Se debe incorporar a la lista de palabras reservadas la palabra ulong.

6. Punto flotante de 64 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra "D" (mayúscula) seguido del signo del exponente que es obligatorio, y luego el valor del exponente. El '.' es obligatorio. Las partes entera o decimal pueden estar ausentes. Si está ausente la parte entera, debe estar presente la parte decimal. Si está ausente la parte decimal, debe estar presente la parte entera.
Ejemplos válidos: 1. .6 -1.2 3. D–5 2.0D+34 2.5D-1 15.0 0.0 .2D+10
Considerar el rango: 2.2250738585072014D-308 < x < 1.7976931348623157D+308
-1.7976931348623157D+308 < x < -2.2250738585072014D-308 0.0
Se debe incorporar a la lista de palabras reservadas la palabra dfloat.

12. Cadenas de 1 línea: Cadenas de caracteres delimitadas por comillas dobles. Estas cadenas no pueden ocupar más de una línea). Ejemplo: "¡Hola mundo !"
Incorporar a la lista de palabras reservadas, las palabras while y do.

33. Comentarios multilínea: Comentarios que comiencen con "##" y terminen con "##" (estos comentarios pueden ocupar más de una línea).

Decisiones de diseño

Para realizar este trabajo optamos por seleccionar el lenguaje de programación **Java**, ya que nos resulta más familiar y sencillo de utilizar. Una vez decidido el lenguaje, comenzamos a armar la estructura general del compilador.
Comenzamos por crear la clase **"AnalisisLexico"**, dentro de la cual definimos el método *"yylex()"*. Este método se encarga de obtener caracteres a partir de la clase **"Buffer"**, que es la responsable de leer el texto fuente: Lee el primer carácter, lo envía a *yylex()* y luego lo elimina para avanzar al siguiente.
A medida que lee caracteres, *yylex()* aplica las **acciones semánticas** necesarias y los concatena hasta crear un **TokenLexema** y devolver su token.
Para crear dicho TokenLexema, el algoritmo recorre una matriz de estados, diseñada a partir del autómata que se mostrará más adelante en el informe.
Con respecto a las acciones semánticas, consideramos implementarlas a través de la interfaz **"AccionSem"** con un método *ejecutar()*. Luego, cada acción semántica es una clase que implementa la interfaz y tiene un comportamiento y atributos propios.

Acciones Semánticas

Las acciones semánticas definen el comportamiento que debe ejecutarse cuando el autómata reconoce una determinada secuencia de caracteres o se encuentra en una transición específica.
A continuación, se detalla la función de cada acción implementada:

AS1: Se ejecuta cuando se reconoce un nuevo lexema. Crea un nuevo objeto *TokenLexema* vacío y asigna como lexema el primer carácter leído.

AS2: Concatena el nuevo carácter al lexema ya existente. Mantiene la referencia al mismo objeto TokenLexema para seguir acumulando el contenido del token.

AS3: Chequea que el ulong que no se pase de 32 bits y que no se vaya de rango, si lo hace reporta error. Luego, si el lexema no se encuentra en la tabla de símbolos, lo agrega, indicando su tipo (Ulong). E ingresa el token correspondiente a constante.

AS4: Devuelve a la entrada el último carácter leído y divide el lexema en base y exponente, calcula su valor numérico y verifica que se encuentre dentro del rango representable por un dfloat (64 bits), si lo hace reporta un error. Luego, si el lexema no se encuentra en la tabla de símbolos, lo agrega, indicando su tipo (Dfloat). Y ingresa el token correspondiente a constante.

AS5: Devuelve el último carácter leído al buffer y verifica si el lexema correspondiente a una palabra reservada existe en la tabla de tokens Si existe asigna el token correspondiente a la palabra clave y sino reporta un mensaje indicando que el lexema no corresponde a una palabra reservada válida.

AS6: Devuelve al buffer el último carácter leído y si el identificador no está en la tabla de símbolos lo agrega con su token correspondiente a ID asegurándose que no tenga más de 20 caracteres (en caso de ser necesario lo trunca).

AS7: Se utiliza en casos donde se necesita devolver el último carácter leído al buffer (retroceso controlado). Agrega el carácter actual al buffer y devuelve el lexema sin modificaciones. Se usa en estados intermedios del autómata donde debe conservarse el último símbolo.

AS8: Concatena el último carácter y si la cadena leída no está en la tabla de símbolos la agrega con el token correspondiente.

AS9: Se ejecuta cuando hay una transición invalida y reporta el error léxico correspondiente.

Estas acciones constituyen el vínculo entre el autómata y la generación de tokens, transformando la lectura secuencial de caracteres en una secuencia estructurada de componentes léxicos para el analizador sintáctico.

Entonces estas acciones semánticas permiten:

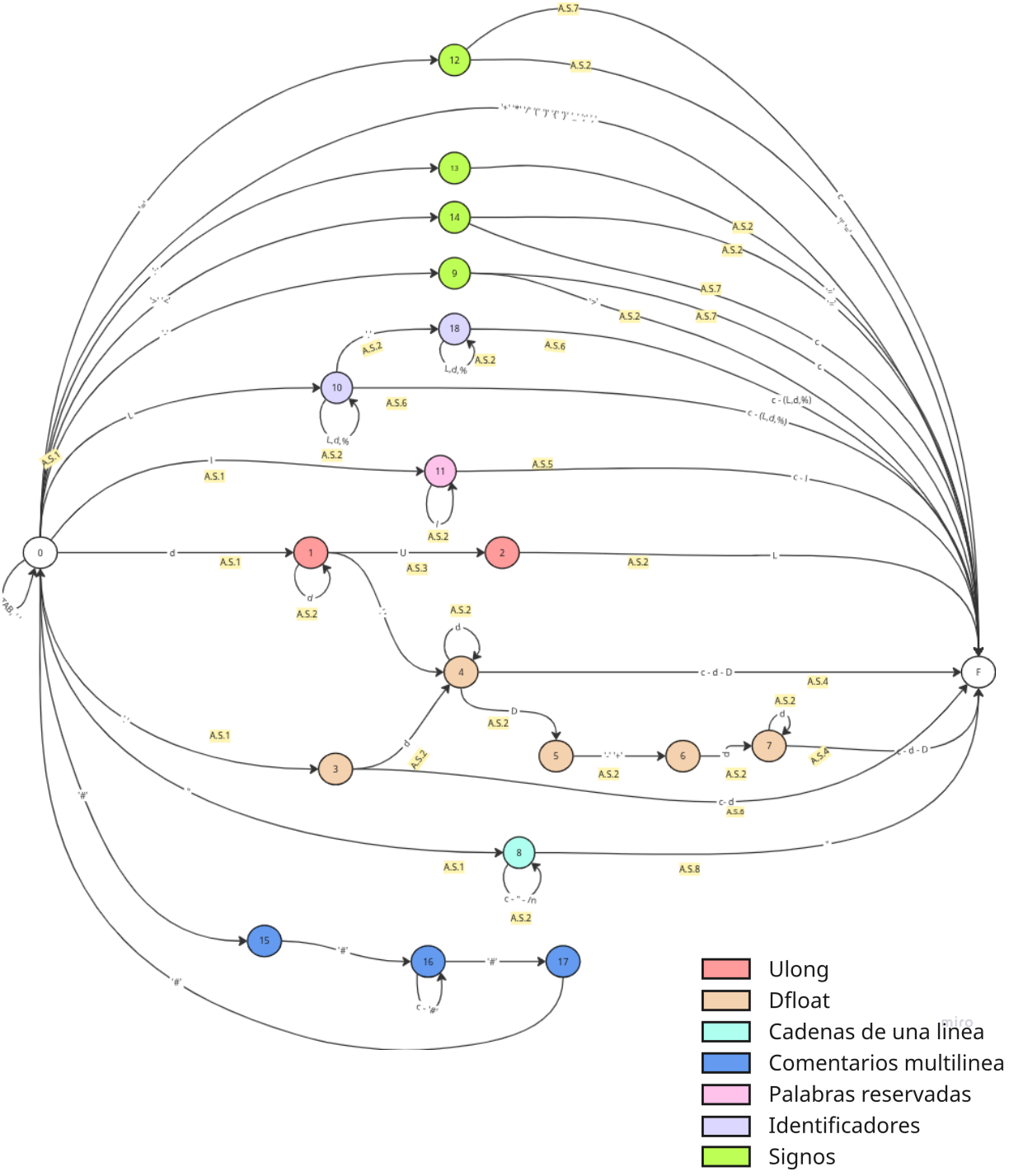
- Construir progresivamente los lexemas.
- Validar rangos numéricos y tipos de datos.
- Registrar tokens e identificadores en la tabla de símbolos.
- Detectar y reportar errores de manera controlada.

Errores léxicos

Durante la construcción del analizador léxico se incorporó la detección de diversos tipos de errores con el objetivo de mejorar la robustez del compilador.

- **Acción semántica 3:** al detectar un ULONG, si el entero largo sin signo excede el rango permitido, el valor se trunca y se lanza un mensaje de error informando la situación.
- **Acción semántica 4:** se aplica un procedimiento similar para los DFLOAT; si el valor sobrepasa el rango admitido, se trunca y se notifica el error.
- **Acción semántica 5:** cuando el lexema leído no se encuentra en la tabla de símbolos, se informa que no corresponde a una palabra reservada.
- **Acción semántica 6:** si un identificador supera los 20 caracteres, el analizador trunca el lexema y genera una advertencia.
- **Acción semántica 9:** se encarga de imprimir todos los errores léxicos relacionados con transiciones inválidas dentro del autómata.

Autómata


Ulong
Dfloat
Cadenas de una linea
Comentarios multilinea
Palabras reservadas
Identificadores
Signos

Matrices

Matriz de transiciones

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	dígito	U	L	D	letra min	Letra mays	.	+	-	*	/	:	>	<	!	=	(	)	{	}	;	_	"	/n	#	/t	' '	,	%
0	1	10	10	10	11	10	3	F2	9	F2	F2	13	14	14	F2	12	F2	F2	F2	F2	F2	F2	8	0	15	0	0	F2	
1	1	2					4																						
2			F																										
3	4	F	F	F	F	F																							
4	4	F	F	5	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
5								6	6																				
6	7																												
7	7	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	F		8	8	8	8	8
9	F	F	F	F	F	F	F	F	F	F	F	F	F2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
10	10	10	10	10	F	10	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	10
11	F	F	F	F	11	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
12	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F2	F2	F	F	F	F	F	F	F	F	F	F	F	F	
13																F2													
14	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F2	F	F	F	F	F	F	F	F	F	F	F	F	
15																									16				
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	17	16	16	16	16
17																									0				

Matriz de acciones semánticas

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	dígito	U	L	D	letra min	Letra mays	.	+	-	*	/	:	>	<	!	=	(	)	{	}	;	_	"	/n	#	/t	' '	,	%
0	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1	A.S.1					A.S.1	
1	A.S.2	A.S.3					A.S.2																						
2			A.S.2																										
3	A.S.2																												
4	A.S.2	A.S.4	A.S.4	A.S.2	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	
5								A.S.2	A.S.2																				
6	A.S.2																												
7	A.S.2	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	A.S.4	
8	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.2	A.S.8		A.S.2	A.S.2	A.S.2	A.S.2	A.S.2
9	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.2	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	
10	A.S.2	A.S.2	A.S.2	A.S.2	A.S.6	A.S.2	A.S.2	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.6	A.S.2
11	A.S.5	A.S.5	A.S.5	A.S.5	A.S.2	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	A.S.5	
12	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.2	A.S.2	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	
13																A.S.2													
14	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.2	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	A.S.7	
15																													
16																													
17																													

Tabla de tokens

Token	Identificador
if	257
else	258
endif	259
print	260
return	261
ulong	262
while	263
do	264
cte	265
cadena	266
id	267
cvr	268
:=	269
+	270
-	271
*	272
/	273
=	274
>=	275
<=	276
>	277
<	278
==	279
!=	280
(	281
)	282
{	283
}	284
_	285
;	286
->	287
.	288
,	289

Analizador Sintactico

Temas asignados

12. while (<condicion>) do <bloque_de_sentencias_ejecutables> ;
<condición> tendrá la misma definición que la condición de las sentencias de selección.
<bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables
delimitadas por { }.

17. Asignaciones que pueden tener menor número de elementos del lado izquierdo
Se deben reconocer asignaciones múltiples, que permitan una lista de variables, separadas por coma (“,”), del
lado izquierdo de la asignación, y una lista de elementos separados por coma (“,”), del lado derecho.
El operador utilizado para la asignación será “=”.
El número de elementos del lado izquierdo puede ser menor que el del lado derecho.
Los elementos del lado derecho sólo pueden ser constantes.
Por ejemplo:
A,B,C = 1l, 2.1D+12, 5l;
X,Y = 1L, 0.25F-2, 2L;
X = 1L, 0.25F-2, 2L;

23. Prefijado Opcional
Incorporar la posibilidad de utilizar, en cualquier lugar donde pueda participar una variable, un prefijado que
indicará la unidad a la que pertenece. Se escribirá como un identificador seguido de un “.” precediendo al nombre
de la variable:
ID.ID
Este prefijado se utilizará para indicar la unidad a la que pertenece la variable, en caso que no esté declarada
localmente.
El uso correcto o incorrecto del prefijado se chequeará en la etapa 3 del Trabajo Práctico.
Por ejemplo:
F1.a := 8l;
F2.X := 1L + F1.X;
X,Y = F2.B, 1l;

26.Copia-Valor-Resultado
// LÉXICO: Agregar a la lista de palabras reservadas, la palabra cvr

En la declaración de la función, antes de cada parámetro formal se podrá indicar, mediante una palabra
reservada, la semántica del pasaje de ese parámetro.
Casos posibles:
<parámetro_formal> es <tipo> ID // semántica por defecto
<parámetro_formal> es cvr <tipo> ID // copia-valor-resultado
Por ejemplo:
int F (cvr int X, cvr int Y, int Z) {
...
}

27. Expresiones Lambda en Línea
Incorporar la definición y uso de expresiones Lambda que pueden ser usadas en línea.
La sintaxis será
<parámetro><cuerpo><argumento>
Estas expresiones permitirán un solo parámetro, que se escribirá entre paréntesis con la estructura <tipo> ID
El cuerpo será un bloque de sentencias ejecutables delimitado por llaves
El argumento podrá ser un identificador o constante entre paréntesis.
Ejemplo:
(int A){if (A>1I) print (“hola”);}(3I) // imprime hola

Errores sintácticos

Falta de nombre de programa.
Falta de delimitador de programa.
Falta de “;” al final de las sentencias.
Falta de nombre en función.
Falta de “,” en declaración de variables.
Falta de nombre de parámetro formal en declaración de función.
Falta de tipo de parámetro formal en declaración de función.
Falta de especificación del parámetro formal al que corresponde el parámetro real.
Falta argumento en sentencia print.
Falta de paréntesis de apertura y/o cierre en condición de selecciones e iteraciones.
Falta de cuerpo en iteraciones..
Falta de endif.
Falta de contenido en bloque then/else.
Falta de operando en expresión.
Falta de operador en expresión.
Falta de comparador en comparación.
Falta do.
Falta de “,” en lista de elementos del lado izquierdo o del lado derecho.
Falta de le después de cv o cr / Falta de cr o cv antes de le.
Falta de delimitadores { y/o } de la función Lambda.

Descripción del proceso de desarrollo

Lo primero a desarrollar en esta parte del trabajo fueron las reglas gramaticales, definidas en el archivo *gramatica.y*. Estas fueron diseñadas para poder identificar la sintaxis del lenguaje a compilar.
Luego usamos la herramienta **yacc** para generar el parser, ejecutando el comando: `yacc -J gramatica.y` en la terminal y se generaron los archivos Parser.java y ParserVal.java. Para integrarlos a nuestro compilador y que funcionen en conjunto creamos la función yylex() en el analizador léxico y la invocamos desde la gramática de la siguiente manera:

```
int yylex (){
    try {
        int token = aLex.yylex();
        yylval = aLex.getYylval();
        return token;
    } catch (IOException e) {
        System.err.println("Error de lectura en el analizador léxico: " + e.getMessage());
        return 0; //Devuelvo 0 como si fuera fin de archivo
    }
}
```

La clase AnalizadorLexico tiene un atributo llamado yylval donde se pasa la referencia de la tabla de símbolos (se le asigna el lexema al atributo sval). Luego en la gramática, en el método yylex, asigna el yylval del AnalizadorLexico al del Parser.

El proyecto cuenta con diferentes archivos de texto dentro de la carpeta “test”, los cuales funcionan como muestra de los tipos de errores contemplados en la gramática.

Problemas surgidos y soluciones adoptadas

Al momento de **controlar los errores**, dos de los requerimientos establecidos por la cátedra se superponían, lo que provocó un **conflicto shift/reduce**.

1. **La falta de operador en expresión:** La regla gramatical sin control de errores era:

```
expresiones     :expresiones operador termino
                |termino
                ;
```

Luego para controlar los errores agregamos la regla `expresiones ERROR termino`.

2. **La falta de “,” en lista de elementos del lado izquierdo o del lado derecho:** La regla gramatical sin control de errores era:

```
lista_id    :tipo_id COMA tipo_id
            |lista_id COMA tipo_id
            ;
```

Luego para contemplar los errores agregamos las reglas `lista_id ERROR tipo_id` y `tipo_id ERROR tipo_id`.

Estas nuevas reglas generaron problemas de shift/reduce, dado que *termino* puede coincidir con *tipo_id*, y el compilador no puede determinar cuál de las reglas aplicar. Como solución

optamos por no agregar la regla de control en **expresiones**, mientras que en **lista_id** incorporamos las reglas *lista_id tipo_id* y *tipo_id tipo_id*, incluyendo un print de error en cada una de ellas.

Luego, agregamos una regla de control general de errores en:

```
prog        :ID bloque
            |error bloque
            |error
            ;
```

Con esta actualización de la regla `error` podemos contemplar el error 1, ya que, al no existir ninguna regla gramatical que coincida, el compilador utiliza esta como default. De esta manera, el error se maneja con un control general y permite continuar con el proceso de compilación sin interrupciones.

Por otro lado, decidimos que las **sentencias condicionales** acepten únicamente **una condicion**, ya que la gramática que definimos no contiene las palabras reservadas "**or**" y "**and**" para llevar a cabo dos o más condiciones lógicas. De esta forma, cada sentencia *if* o *while* evalúa una única expresión.

Además, decidimos no permitir múltiples programas en el mismo archivo de entrada, ya que cuando intentamos permitirlo nos generaba conflictos de shift/reduce.

Por último, al detectar constantes negativas, la gramática las permite; sin embargo, en el caso de los **ULONG**, estos representan enteros largos sin signo. Por lo tanto, cuando aparece una secuencia del tipo '-' *CTE*, el analizador reduce el token sin poder determinar correctamente si se trata de un **DFLOAT** o de un **ULONG**.
Nuestra solución fue **no modificar la tabla de símbolos**, aunque gramaticalmente se permite la aparición de constantes negativas.

Errores

- **Falta de operando:** el control de error por falta de operando no funciona cuando falta el primer operando de la expresión. En caso de faltar el operando izquierdo, no se aplica ninguna regla específica, ya que al intentar contemplarla se generaba un conflicto sintáctico. En esas situaciones, se utiliza la regla general de *error* sintáctico.

- **Falta de punto y coma:** El compilador solo detecta la falta de punto y coma dentro de un bloque entre llaves. Creemos que esto se debe a la sentencia de *error* general, coincide con la sentencia: *sentencia error,* por lo que el compilador imprime *"ERROR SINTÁCTICO".*
  Intentamos solucionarlo cambiando la regla *sentencia error* por *sentencia error PUNTOCOMA*, pensando que esto haría que el compilador frene el error en la próxima sentencia con punto y coma, pero no funcionó. También probamos solucionar esto dejando la regla como *sentencia* pero el Parser nos dió conflicto shift/reduce.

Conclusión

A lo largo del desarrollo del proyecto, se logró implementar tanto el analizador léxico como el analizador sintáctico, cumpliendo con la mayoría de los requerimientos establecidos.

El analizador léxico permitió reconocer los distintos componentes del archivo fuente, clasificándolos según su tipo de token y manteniendo actualizada la tabla de símbolos, donde un mismo identificador puede asociarse a diferentes lexemas según el contexto.

Por otra parte, el analizador sintáctico permitió validar la gramática del lenguaje a través de un conjunto de reglas definidas, contemplando además los posibles errores gramaticales. Estos son notificados sin interrumpir el proceso de compilación.

A pesar de algunos inconvenientes en el manejo de errores, se consiguió desarrollar una gramática funcional, capaz de analizar correctamente la mayoría de las estructuras del lenguaje diseñado.