



INSTITUTO TECNOLÓGICO DE COSTA RICA

Proyecto #2: Parser

Escuela de Ingeniería en Computación
Compiladores e Intérpretes IC-5701

Alonso Navarro Carrillo, c. 2022236435

Carlos Venegas Masis, c. 2022153870

Valeria Gómez Acuña, c. 2022173229

Ing. Ericka Marín Schumann
II Semestre 2024

Tabla de contenidos

Introducción	2
Estrategia de solución	2
Análisis de resultados	4
Lecciones aprendidas	4
Casos de prueba	5
Casos de prueba 1: Comentarios	5
Casos de prueba 2: Operadores	8
Caso de prueba 3: Identificadores	15
Caso de prueba 4: Formato Literales	19
Caso de prueba 5: Palabras reservadas	24
Caso de prueba 6: General	28
Manual de usuario	34
Bitácora	35
Bibliografía	38

Introducción

Este proyecto se ubica en la segunda etapa de la creación de un compilador para el lenguaje de programación C, conocida como el Parsing o Análisis Sintáctico. El propósito principal de esta etapa es diseñar y desarrollar un parser que logre identificar como se relacionan los tokens que se obtienen de la fase de análisis léxico y determinar si estos tokens forman expresiones, sentencias o estructuras válidas para un programa escrito en lenguaje C. Para lograr esto, se utilizaron dos herramientas Java CUP y JFlex, Java CUP permitió definir reglas gramaticales que describen como deben combinarse los tokens para ser reconocidos por el parser como correctos. Mientras que JFlex fue utilizado en la etapa de análisis Léxico, esta etapa se desarrolló en la primera fase del proyecto.

A lo largo de este documento se detalla el proceso de desarrollo del scanner, desde la planificación hasta la implementación y evaluación de los resultados obtenidos. Se describen decisiones técnicas tomadas para garantizar el correcto funcionamiento del proyecto, así como las estrategias empleadas para la solución de los problemas encontrados durante su desarrollo.

Estrategia de solución

Después de leer detenidamente la documentación de JFlex, se comenzó a diseñar las expresiones regulares para los tokens que debía reconocer el escáner. Aquí surgió el primer problema: el escáner reconoce los tokens según el orden de prioridad. Es decir, si la primera expresión regular es un punto, ningún otro token será reconocido, ya que este metacaracter coincide con cualquier carácter, interpretándolo como un error. Por lo tanto, fue crucial definir adecuadamente el orden de las expresiones regulares.

Una vez definido el orden de las expresiones regulares, procedimos a diseñar la estructura de los tokens y sus tipos. Para ello, decidimos crear un mapa que facilitara la búsqueda de tokens repetidos en el documento. De manera similar, cada token guarda las líneas en las que aparece en un mapa, lo que permite incrementar el contador de ocurrencias de un token en una misma línea. Finalmente, los tipos de tokens se almacenan en un enum.

Por último, se implementaron errores definidos, como un número seguido de un identificador o números flotantes que comienzan con un punto. Era necesario definir estos errores como tokens para que pudieran ser reconocidos por el escáner. Los errores no se almacenan en una estructura de datos; en su lugar, se imprimen

directamente y no se agregan al mapa de tokens.

Análisis de resultados

Actividad	Porcentaje realizado	Justificación
Desplegar lista de errores léxicos	100%	
Desplegar lista de errores sintácticos	100%	
Evitar desplegar errores sintácticos en cascada	50%	En situaciones muy específicas el parser se puede caer o dar errores poco precisos. Esto debido a complicaciones en la gramática.
Implementar las funciones de read and write	100%	
Declaración de variables, constantes y lista de variables	100%	
Reconocer la estructura del programa	100%	
Identificar funciones	100%	
Analizar expresiones con operadores aritméticos o booleanos	100%	
Identificar todas las estructuras de control	100%	
Definir buenos mensajes de error	80%	La mayoría de errores están bien definidos, pero si el flujo es erróneo no se obtiene un buen mensaje.

Lecciones aprendidas

Durante el desarrollo del proyecto el grupo pudo observar que en el scanner es muy importante el orden de las expresiones regulares porque, si el orden es incorrecto se pueden capturar tokens no deseados antes que los tokens válidos lo cual provoca errores. Además, identificar y clasificar errores como números o cadenas de caracteres con mal formato e identificadores que no siguen las reglas, permitió al grupo detectar mejor los errores y mostrar mensajes de salida útiles, brindando robustez al scanner gracias al manejo de errores léxicos.

Evitar la separación incorrecta de tokens compuestos fue un reto porque, en algunos casos los errores no se trataban como un único token con mal formato sino

que, se separaban en varios elementos válidos, esto hacía que en ocasiones se interpretara incorrectamente el código. Por lo tanto, para mejorar la precisión del scanner se definieron expresiones regulares más claras.

Casos de prueba

Caso de prueba 1: Comentarios

```
#include <stdio.h> // Comentario después de directiva de preprocesador
#define MAX(a, b) ((a) > (b) ? (a) : (b)) // Macro con comentario en línea

// Este es un comentario en línea
int main() {
    // Comentario con caracteres especiales: !@#$%^&*()
    /* Comentario en bloque con caracteres especiales: !@#$%^&*() */
    // Comentario con código incorrecto: int x = ;
    /* Comentario en bloque con código incorrecto: int y = ; */
    // Comentario con espacios
    /* Comentario en bloque con espacios */
    // Comentario con tabulaciones
    int a = 10; // Comentario al final de una línea de código
    int b = 20; /* Comentario en bloque
                  que se extiende en varias líneas */
    int d = a + b;
    int c = a + b; // Comentario después de una expresión
    char *str = "Este es un string con // comentario en línea";
    char *str2 = "Este es otro string con /* comentario en bloque */";
    b = 20;
    /* Comentario en bloque sin cerrar
    b = 20; // Comentario en línea anidado
    return 0;
}
```

Errores esperados:

- Error en la línea 1: Operador inválido '#'.
- Error en la línea 2: Operador inválido '#'.
- Error en cascada en línea 20: Comentario de bloque sin cerrar.

Es importante resaltar que aunque el proyecto indique que no se pueden tener errores en cascada, al encontrar un comentario de bloque sin cerrar el comportamiento usual será seguir buscando hasta encontrar un "*/".

Resultados:

Errors:

Character unknown: # in 1

Character unknown: # in 2

Block comment without closure: /* Comentario en bloque sin cerrar

 b = 20; // Comentario en linea anidado

 return 0;

} in 20

Token	Tipo de Token	Linea
char	KEYWORD	17, 18
int	KEYWORD	5, 13, 14, 15, 16
MAX	ID	2
a	ID	2(3), 13, 15, 16
b	ID	2(3), 14, 15, 16, 19
c	ID	16
d	ID	15
define	ID	2
h	ID	1
include	ID	1
main	ID	5
stdio	ID	1

str	ID	17	
+-----+	+-----+	+-----+	+-----+
str2	ID	18	
+-----+	+-----+	+-----+	+-----+
(OPERATOR	2(6), 5	
+-----+	+-----+	+-----+	+-----+
)	OPERATOR	2(6), 5	
+-----+	+-----+	+-----+	+-----+
,	OPERATOR	2	
+-----+	+-----+	+-----+	+-----+
.	OPERATOR	1	
+-----+	+-----+	+-----+	+-----+
;	OPERATOR	13, 14, 15, 16, 17, 18, 19	
+-----+	+-----+	+-----+	+-----+
=	OPERATOR	13, 14, 15, 16, 17, 18, 19	
+-----+	+-----+	+-----+	+-----+
{	OPERATOR	5	
+-----+	+-----+	+-----+	+-----+
*	OPERATOR_ARITHM	17, 18	
	ETIC		
+-----+	+-----+	+-----+	+-----+
+	OPERATOR_ARITHM	15, 16	
	ETIC		
+-----+	+-----+	+-----+	+-----+
:	OPERATOR_RELATI	2	
	ONAL		
+-----+	+-----+	+-----+	+-----+
<	OPERATOR_RELATI	1	
	ONAL		
+-----+	+-----+	+-----+	+-----+
>	OPERATOR_RELATI	1, 2	
	ONAL		
+-----+	+-----+	+-----+	+-----+
?	OPERATOR_RELATI	2	
	ONAL		
+-----+	+-----+	+-----+	+-----+
10	LITERAL_INT	13	
+-----+	+-----+	+-----+	+-----+
20	LITERAL_INT	14, 19	

"Este es o	LITERAL_STR	18	
tro string			
con /* co			
mentario e			
n bloque *			
/"			
"Este es u	LITERAL_STR	17	
n string c			
on // come			
ntario en			
linea"			

Caso de prueba 2: Operadores

```
#include <stdio.h>
```

```
int main() {
    int a = 5 + 3;           // Suma
    int b = a - 2;           // Resta
    int c = a * b;           // Multiplicación
    int d = c / 2;           // División
    int e = c % 2;           // Módulo
    int f = a & b;            // AND bit a bit
    int g = a | b;            // OR bit a bit
    int h = a ^ b;            // XOR bit a bit
    int i = ~a;              // NOT bit a bit
    int j = a << 2;           // Desplazamiento a la izquierda
    int k = b >> 1;           // Desplazamiento a la derecha

    if (a == b) {             // Igualdad
        printf("a es igual a b\n");
    }

    if (a != b) {             // Desigualdad
        printf("a no es igual a b\n");
    }
}
```

```

}

if (a < b) {           // Menor que
    printf("a es menor que b\n");
}

if (a > b) {           // Mayor que
    printf("a es mayor que b\n");
}

if (a <= b) {          // Menor o igual que
    printf("a es menor o igual que b\n");
}

if (a >= b) {          // Mayor o igual que
    printf("a es mayor o igual que b\n");
}

if (a && b) {           // AND lógico
    printf("a y b son verdaderos\n");
}

if (a || b) {          // OR lógico
    printf("a o b es verdadero\n");
}

if (!a) {              // NOT lógico
    printf("a es falso\n");
}

// Errores intencionales
int l = a @ b;         // Operador inválido
int n = a // b;        // Comentario mal formado
int p = a +;           // Operador sin operando
int q = ;              // Declaración sin inicialización

return 0;
}

```

Errores esperados:

- Error en la línea 1: Operador inválido '#'.
- Error en la línea 53: Operador inválido '@'.

También se espera que suponga el '/' como un comentario y no como dos operadores de división.

Resultados:

Errors:

Character unknown: # in 1

Character unknown: @ in 53

Token	Tipo de Token	Linea
if	KEYWORD	16, 20, 24, 28, 32, 36, 40, 44, 48
int	KEYWORD	3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 53, 54, 55, 56
return	KEYWORD	58
a	ID	4, 5, 6, 9, 10, 11, 12, 13, 16, 20, 24, 28, 32, 36, 40, 44, 48, 53, 54, 55
b	ID	5, 6, 9, 10, 11, 14, 16, 20, 24, 28, 32, 36, 40, 44, 53
c	ID	6, 7, 8
d	ID	7
e	ID	8
f	ID	9
g	ID	10
h	ID	1, 11

+-----+-----+-----+			
i	ID	12	
+-----+-----+-----+			
include	ID	1	
+-----+-----+-----+			
j	ID	13	
+-----+-----+-----+			
k	ID	14	
+-----+-----+-----+			
l	ID	53	
+-----+-----+-----+			
main	ID	3	
+-----+-----+-----+			
n	ID	54	
+-----+-----+-----+			
p	ID	55	
+-----+-----+-----+			
printf	ID	17, 21, 25, 29, 33, 37, 41, 45, 49	
+-----+-----+-----+			
q	ID	56	
+-----+-----+-----+			
stdio	ID	1	
+-----+-----+-----+			
(OPERATOR	3, 16, 17, 20, 21, 24, 25, 28, 29, 32, 3	
		3, 36, 37, 40, 41, 44, 45, 48, 49	
+-----+-----+-----+			
)	OPERATOR	3, 16, 17, 20, 21, 24, 25, 28, 29, 32, 3	
		3, 36, 37, 40, 41, 44, 45, 48, 49	
+-----+-----+-----+			
.	OPERATOR	1	
+-----+-----+-----+			
;	OPERATOR	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17	
		, 21, 25, 29, 33, 37, 41, 45, 49, 53, 55	
		, 56, 58	
+-----+-----+-----+			
=	OPERATOR	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 53	
		, 54, 55, 56	
+-----+-----+-----+			
{	OPERATOR	3, 16, 20, 24, 28, 32, 36, 40, 44, 48	

}	OPERATOR	18, 22, 26, 30, 34, 38, 42, 46, 50, 59	
%	OPERATOR_ARITHM	8	
	ETIC		
*	OPERATOR_ARITHM	6	
	ETIC		
+	OPERATOR_ARITHM	4, 55	
	ETIC		
-	OPERATOR_ARITHM	5	
	ETIC		
/	OPERATOR_ARITHM	7	
	ETIC		
!	OPERATOR_LOGICA	48	
	L		
&&	OPERATOR_LOGICA	40	
	L		
	OPERATOR_LOGICA	44	
	L		
!=	OPERATOR_RELATI	20	
	ONAL		
<	OPERATOR_RELATI	1, 24	
	ONAL		
<=	OPERATOR_RELATI	32	
	ONAL		
==	OPERATOR_RELATI	16	
	ONAL		

>	OPERATOR_RELATI	1, 28	
	ONAL		
+-----+-----+-----+-----+			
>=	OPERATOR_RELATI	36	
	ONAL		
+-----+-----+-----+-----+			
&	OPERATOR_BITWIS	9	
	E		
+-----+-----+-----+-----+			
<<	OPERATOR_BITWIS	13	
	E		
+-----+-----+-----+-----+			
>>	OPERATOR_BITWIS	14	
	E		
+-----+-----+-----+-----+			
^	OPERATOR_BITWIS	11	
	E		
+-----+-----+-----+-----+			
	OPERATOR_BITWIS	10	
	E		
+-----+-----+-----+-----+			
~	OPERATOR_BITWIS	12	
	E		
+-----+-----+-----+-----+			
0	LITERAL_INT	58	
+-----+-----+-----+-----+			
1	LITERAL_INT	14	
+-----+-----+-----+-----+			
2	LITERAL_INT	5, 7, 8, 13	
+-----+-----+-----+-----+			
3	LITERAL_INT	4	
+-----+-----+-----+-----+			
5	LITERAL_INT	4	
+-----+-----+-----+-----+			
"a es fals	LITERAL_STR	49	
o\n"			
+-----+-----+-----+-----+			
"a es igua	LITERAL_STR	17	
l a b\n"			

"a es mayo	LITERAL_STR	37	
r o igual			
que b\n"			
"a es mayo	LITERAL_STR	29	
r que b\n"			
"a es meno	LITERAL_STR	33	
r o igual			
que b\n"			
"a es meno	LITERAL_STR	25	
r que b\n"			
"a no es i	LITERAL_STR	21	
gual a b\n			
"			
"a o b es	LITERAL_STR	45	
verdadero\			
n"			
"a y b son	LITERAL_STR	41	
verdadero			
s\n"			

Caso de prueba 3: Identificadores

```
#include <stdio.h>

int 1invalidVar = 10; // Identificador comienza con un dígito
int validVar = 20; // Identificador válido
float 2anotherInvalid = 30.5; // Identificador comienza con un dígito
char special#CharVar = 'A'; // Identificador contiene un carácter especial '#'
int valid_var_123 = 50; // Identificador válido

void function() {
    int invalid@Var = 100; // Identificador contiene un carácter especial '@'
    int valid_var = 0; // Identificador válido
    float invalid%percent = 40.2; // Identificador contiene un carácter especial '%'
    double validDoubleVar = 123.456; // Identificador válido
    double 3invalidStart = 987.654; // Identificador comienza con un dígito
}

int main() {
    printf("Valor de validVar: %d\n", validVar); // Identificador válido
    printf("Valor de valid_var_123: %d\n", valid_var_123); // Identificador válido

    return 0;
}
```

Errores esperados:

- Error en la línea 1: Operador inválido '#'.
- Error en la línea 3: Dígito antes de id '1invalidVar'.
- Error en la línea 3: Dígito antes de id '1invalidVar'.
- Error en la línea 5: Dígito antes de id '2anotherInvalid'.
- Error en la línea 6: Operador inválido '#'.
- Error en la línea 10: Operador inválido '@'.
- Error en la línea 14: Dígito antes de id '3invalidStart'.

Resultados:

Errors:

Character unknown: # in 1

Digit before id: 1invalidVar in 3

Digit before id: 2anotherInvalid in 5

Character unknown: # in 6

Character unknown: @ in 10

Digit before id: 3invalidStart in 14

Token	Tipo de Token	Linea
char	KEYWORD	6
double	KEYWORD	13, 14
float	KEYWORD	5, 12
int	KEYWORD	3, 4, 7, 10, 11, 17
return	KEYWORD	21
void	KEYWORD	9
CharVar	ID	6
Var	ID	10
function	ID	9
h	ID	1
include	ID	1
invalid	ID	10, 12
main	ID	17
percent	ID	12

printf	ID	18, 19	
+-----+	+-----+	+-----+	+-----+
special	ID	6	
+-----+	+-----+	+-----+	+-----+
stdio	ID	1	
+-----+	+-----+	+-----+	+-----+
validDoubl	ID	13	
eVar			
+-----+	+-----+	+-----+	+-----+
validVar	ID	4, 18	
+-----+	+-----+	+-----+	+-----+
valid_var	ID	11	
+-----+	+-----+	+-----+	+-----+
valid_var_	ID	7, 19	
123			
+-----+	+-----+	+-----+	+-----+
(OPERATOR	9, 17, 18, 19	
+-----+	+-----+	+-----+	+-----+
)	OPERATOR	9, 17, 18, 19	
+-----+	+-----+	+-----+	+-----+
,	OPERATOR	18, 19	
+-----+	+-----+	+-----+	+-----+
.	OPERATOR	1	
+-----+	+-----+	+-----+	+-----+
;	OPERATOR	3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 18, 1	
		9, 21	
+-----+	+-----+	+-----+	+-----+
=	OPERATOR	3, 4, 5, 6, 7, 10, 11, 12, 13, 14	
+-----+	+-----+	+-----+	+-----+
{	OPERATOR	9, 17	
+-----+	+-----+	+-----+	+-----+
}	OPERATOR	15, 22	
+-----+	+-----+	+-----+	+-----+
%	OPERATOR_ARITHM	12	
	ETIC		
+-----+	+-----+	+-----+	+-----+
<	OPERATOR_RELATI	1	
	ONAL		
+-----+	+-----+	+-----+	+-----+

>	OPERATOR_RELATI	1	
	ONAL		
+-----+-----+-----+			
0	LITERAL_INT	11, 21	
+-----+-----+-----+			
10	LITERAL_INT	3	
+-----+-----+-----+			
100	LITERAL_INT	10	
+-----+-----+-----+			
20	LITERAL_INT	4	
+-----+-----+-----+			
50	LITERAL_INT	7	
+-----+-----+-----+			
"Valor de	LITERAL_STR	18	
validVar:			
%d\n"			
+-----+-----+-----+			
"Valor de	LITERAL_STR	19	
valid_var_			
123: %d\n"			
+-----+-----+-----+			
'A'	LITERAL_CHAR	6	
+-----+-----+-----+			
123.456	LITERAL_DOUBLE	13	
+-----+-----+-----+			
30.5	LITERAL_DOUBLE	5	
+-----+-----+-----+			
40.2	LITERAL_DOUBLE	12	
+-----+-----+-----+			
987.654	LITERAL_DOUBLE	14	
+-----+-----+-----+			

Caso de prueba 4: Formato Literales

```
#include <stdio.h>

int main() {
    // Números decimales
    int decimal = 12345; // Decimal positivo
    int negativeDecimal = -6789; // Decimal negativo

    // Números octales
    int octal = 0123; // Octal (equivalente a 83 en decimal)
    int negativeOctal = -07654; // Octal negativo (equivalente a -4012 en decimal)

    // Números hexadecimales
    int hexadecimal = 0x1A3F; // Hexadecimal (equivalente a 6719 en decimal)
    int negativeHexadecimal = -0xDEAD; // Hexadecimal negativo (equivalente a -57005 en decimal)

    // Números binarios (C11 en adelante)
    int binary = 0b1101; // Binario (equivalente a 13 en decimal)

    // Números con punto flotante
    float floatNum = 3.14159; // Flotante positivo
    float negativeFloat = -0.9876; // Flotante negativo

    // Números en notación exponencial (notación científica)
    double scientificPos = 1.23e4; //  $1.23 \times 10^4 = 12300$ 
    double scientificNeg = -5.67E-3; //  $-5.67 \times 10^{-3} = -0.00567$ 

    // Números flotantes con notación hexadecimal (C99)
    double hexFloat = 0x1.1p3; // Equivalente a 8.5 en decimal

    // Strings-caracteres
    char *f = "Cadena válida"; // Cadena válida de caracteres
    char g = 'a'; // Caracter válido
    char h = #65; // Literal char válido

    // Errores intencionales:
    int invalidHex = 0x1G; // Error: carácter inválido en número hexadecimal
    float invalidExp = 12.34e+; // Error: exponente no válido
```

```

int invalidOctal = 089; // Error: dígitos no válidos en número octal
double invalidFloat = 1.23.45; // Error: punto flotante mal formado
    double invalid = 5..38; // Punto flotante mal formado
int invalidInt = 123abc;          // Número entero inválido
char *s = "Esto es un \n
string inválido porque está en múltiples líneas";

return 0;
}

```

Errores esperados:

- Error en la línea 1: Operador inválido '#'.
- Error en la línea 28: Formato de número no válido '.1'.
- Error en la línea 31: Dígito antes de id '0x1G'.
- Error en la línea 34: Formato de número no válido '.45'.
- Error en la línea 40: Formato de número no válido '1.23.45'.
- Error en la línea 41: Formato de número no válido '5..38'.
- Error en la línea 43: String no valido "Esto es un string inválido porque está en múltiples líneas".

Resultados:

Errors:

Character unknown: # in 1

Invalid number format: .1 in 28

Digit before id: 0x1G in 31

Invalid number format: .45 in 34

Invalid number format: 1.23.45 in 40

Invalid number format: 5..38 in 41

Strings cannot span multiple lines: "Esto es un \n string inválido porque está en múltiples líneas".

+-----+-----+-----+-----+			
Token	Tipo de Token	Linea	
+-----+-----+-----+-----+			
char	KEYWORD	31, 32, 33	
+-----+-----+-----+-----+			
double	KEYWORD	24, 25, 28, 40, 41	

float	KEYWORD	20, 21, 38	
int	KEYWORD	3, 5, 6, 9, 10, 13, 14, 17, 37, 39, 42	
return	KEYWORD	44	
binary	ID	17	
decimal	ID	5	
e	ID	38	
f	ID	31	
floatNum	ID	20	
g	ID	32	
h	ID	1, 33	
hexFloat	ID	28	
hexadecimal	ID	13	
l			
include	ID	1	
invalid	ID	41	
invalidExp	ID	38	
invalidFlo	ID	40	
at			
invalidHex	ID	37	
invalidInt	ID	42	

invalidOct	ID	39	
al			
+-----+			
main	ID	3	
+-----+			
negativeDe	ID	6	
cimal			
+-----+			
negativeFl	ID	21	
oat			
+-----+			
negativeHe	ID	14	
xadecimal			
+-----+			
negativeOc	ID	10	
tal			
+-----+			
octal	ID	9	
+-----+			
scientific	ID	25	
Neg			
+-----+			
scientific	ID	24	
Pos			
+-----+			
stdio	ID	1	
+-----+			
xDEAD	ID	14	
+-----+			
(OPERATOR	3	
+-----+			
)	OPERATOR	3	
+-----+			
.	OPERATOR	1, 28	
+-----+			
;	OPERATOR	5, 6, 9, 10, 13, 14, 17, 20, 21, 24, 25,	
		28, 31, 32, 33, 37, 38, 39, 40, 41, 42,	
		44	
+-----+			

=	OPERATOR	5, 6, 9, 10, 13, 14, 17, 20, 21, 24, 25,	
		28, 31, 32, 33, 37, 38, 39, 40, 41, 42	
+-----+-----+-----+			
{	OPERATOR	3	
+-----+-----+-----+			
}	OPERATOR	45	
+-----+-----+-----+			
*	OPERATOR_ARITHM	31	
	ETIC		
+-----+-----+-----+			
+	OPERATOR_ARITHM	38	
	ETIC		
+-----+-----+-----+			
<	OPERATOR_RELATI	1	
	ONAL		
+-----+-----+-----+			
>	OPERATOR_RELATI	1	
	ONAL		
+-----+-----+-----+			
-0	LITERAL_INT	10, 14	
+-----+-----+-----+			
-6789	LITERAL_INT	6	
+-----+-----+-----+			
0	LITERAL_INT	39, 44	
+-----+-----+-----+			
12345	LITERAL_INT	5	
+-----+-----+-----+			
7654	LITERAL_INT	10	
+-----+-----+-----+			
89	LITERAL_INT	39	
+-----+-----+-----+			
"Cadena vá	LITERAL_STR	31	
lida"			
+-----+-----+-----+			
#65	LITERAL_CHAR	33	
+-----+-----+-----+			
'a'	LITERAL_CHAR	32	
+-----+-----+-----+			
0b1101	LITERAL_BINARY	17	

0123	LITERAL_OCTAL	9	
0x1	LITERAL_HEX	28	
0x1A3F	LITERAL_HEX	13	
-0.9876	LITERAL_DOUBLE	21	
-5.67E-3	LITERAL_DOUBLE	25	
1.23e4	LITERAL_DOUBLE	24	
12.34	LITERAL_DOUBLE	38	
3.14159	LITERAL_DOUBLE	20	

Caso de prueba 5: Palabras reservadas

```
#include <stdio.h>

int main() {
    auto intVar = 5;           // auto
    int b = intVar + 3;        // int
    char c = 'A';              // char
    const int d = 10;          // const
    unsigned long g = 100L;
    signed short h = -10;
    static int counter = 0;
    float e = 3.14f;           // float
    double f = 5.0;            // double
    unsigned long g = 100L;    // unsigned long
    signed short h = -10;      // signed y short
    enum days { Mon, Tue };    // enum
    struct point {              // struct
        int x, y;
    };
}
```

```

typedef int entero;      // typedef'
extern int i;            // extern'
static int counter = 0;  // static'
void function() {        // void
    continue;           // continue
}

return 0;
}

```

Resultados:

Token	Tipo de Token	Linea
auto	KEYWORD	2
char	KEYWORD	4
const	KEYWORD	5
continue	KEYWORD	21
double	KEYWORD	10
enum	KEYWORD	13
extern	KEYWORD	18
float	KEYWORD	9
int	KEYWORD	1, 3, 5, 8, 15, 17, 18, 19
long	KEYWORD	6, 11
return	KEYWORD	25
short	KEYWORD	12

signed	KEYWORD	7, 12	
+-----+	+-----+	+-----+	+-----+
static	KEYWORD	19	
+-----+	+-----+	+-----+	+-----+
struct	KEYWORD	14	
+-----+	+-----+	+-----+	+-----+
typedef	KEYWORD	17	
+-----+	+-----+	+-----+	+-----+
unsigned	KEYWORD	11	
+-----+	+-----+	+-----+	+-----+
void	KEYWORD	20	
+-----+	+-----+	+-----+	+-----+
Mon	ID	13	
+-----+	+-----+	+-----+	+-----+
Tue	ID	13	
+-----+	+-----+	+-----+	+-----+
b	ID	3	
+-----+	+-----+	+-----+	+-----+
c	ID	4	
+-----+	+-----+	+-----+	+-----+
counter	ID	8, 19	
+-----+	+-----+	+-----+	+-----+
d	ID	5	
+-----+	+-----+	+-----+	+-----+
days	ID	13	
+-----+	+-----+	+-----+	+-----+
e	ID	9	
+-----+	+-----+	+-----+	+-----+
entero	ID	17	
+-----+	+-----+	+-----+	+-----+
f	ID	10	
+-----+	+-----+	+-----+	+-----+
function	ID	20	
+-----+	+-----+	+-----+	+-----+
g	ID	6, 11	
+-----+	+-----+	+-----+	+-----+
h	ID	7, 12	
+-----+	+-----+	+-----+	+-----+
i	ID	18	

+-----+-----+-----+-----+			
intVar	ID	2, 3	
+-----+-----+-----+-----+			
main	ID	1	
+-----+-----+-----+-----+			
point	ID	14	
+-----+-----+-----+-----+			
shrt	ID	7	
+-----+-----+-----+-----+			
statc	ID	8	
+-----+-----+-----+-----+			
unsigned	ID	6	
+-----+-----+-----+-----+			
x	ID	15	
+-----+-----+-----+-----+			
y	ID	15	
+-----+-----+-----+-----+			
(OPERATOR	1, 20	
+-----+-----+-----+-----+			
)	OPERATOR	1, 20	
+-----+-----+-----+-----+			
,	OPERATOR	13, 15	
+-----+-----+-----+-----+			
;	OPERATOR	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,	
		15, 16, 17, 18, 19, 21, 25	
+-----+-----+-----+-----+			
=	OPERATOR	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19	
+-----+-----+-----+-----+			
{	OPERATOR	1, 13, 14, 20	
+-----+-----+-----+-----+			
}	OPERATOR	13, 16, 22, 26	
+-----+-----+-----+-----+			
+	OPERATOR_ARITHM	3	
	ETIC		
+-----+-----+-----+-----+			
-10	LITERAL_INT	7, 12	
+-----+-----+-----+-----+			
0	LITERAL_INT	8, 19, 25	
+-----+-----+-----+-----+			

10	LITERAL_INT	5	
+-----+	+-----+	+-----+	+-----+
100L	LITERAL_INT	6, 11	
+-----+	+-----+	+-----+	+-----+
3	LITERAL_INT	3	
+-----+	+-----+	+-----+	+-----+
5	LITERAL_INT	2	
+-----+	+-----+	+-----+	+-----+
'A'	LITERAL_CHAR	4	
+-----+	+-----+	+-----+	+-----+
3.14f	LITERAL_DOUBLE	9	
+-----+	+-----+	+-----+	+-----+
5.0	LITERAL_DOUBLE	10	
+-----+	+-----+	+-----+	+-----+

Caso de prueba 6: General

```
#include <stdio.h>

int main() {
    // Palabras reservadas
    int x = 10;           // int
    float y = 5.5f;       // float
    char z = 'c';         // char
    const int max = 100;   // const
    double pi = 3.14159;   // double
    unsigned int u = 5;    // unsigned
    signed int s = -3;     // signed
    enum colors { RED, GREEN, BLUE }; // enum
    struct Point {         // struct
        int x, y;
    };

    // Usando typedef
    typedef int entero;    // typedef
    entero a = 5;          // usando typedef

    // Operaciones
```

```

int sum = x + 20;          // Suma
int product = x * 2;       // Multiplicación
double division = (double)x / 4; // División
int modulo = x % 3;        // Módulo

// Operadores bit a bit
int andBit = x & 1;        // AND bit a bit
int orBit = x | 1;         // OR bit a bit
int xorBit = x ^ 1;        // XOR bit a bit
int notBit = ~x;           // NOT bit a bit

// Desplazamientos
int leftShift = x << 1;    // Desplazamiento a la izquierda
int rightShift = x >> 1;   // Desplazamiento a la derecha

// Errores intencionales
float wrongFloat = 2.5;    // Error: debería ser 'float'
int 2ndVar = 10;           // Error: no puede comenzar con un dígito
const float pi = 3.14;     // Error: redeclaración de 'const'
int l = a @ b;             // Operador inválido
int n = a // b;            // Comentario mal formado

return 0;
}

```

Errores esperados:

- Error en la línea 38: Dígito antes de id '2ndVar'.
- Error en la línea 28: Caracter desconocido '@'.

Resultados:

Errors:

Digit before id: 2ndVar in 38

Character unknown: @ in 40

+-----+-----+-----+-----+			
Token	Tipo de Token	Linea	
+-----+-----+-----+-----+			
char	KEYWORD	5, 32, 33, 34	

+-----+-----+-----+			
const	KEYWORD	6, 39	
+-----+-----+-----+			
double	KEYWORD	7, 22(2)	
+-----+-----+-----+			
enum	KEYWORD	10	
+-----+-----+-----+			
float	KEYWORD	4, 39	
+-----+-----+-----+			
int	KEYWORD	1, 3, 6, 8, 9, 12, 16, 20, 21, 23, 26, 2	
		7, 28, 29, 38, 40, 41	
+-----+-----+-----+			
return	KEYWORD	43	
+-----+-----+-----+			
signed	KEYWORD	9	
+-----+-----+-----+			
struct	KEYWORD	11	
+-----+-----+-----+			
typedef	KEYWORD	16	
+-----+-----+-----+			
unsigned	KEYWORD	8	
+-----+-----+-----+			
BLUE	ID	10	
+-----+-----+-----+			
GREEN	ID	10	
+-----+-----+-----+			
Point	ID	11	
+-----+-----+-----+			
RED	ID	10	
+-----+-----+-----+			
a	ID	17, 40, 41	
+-----+-----+-----+			
andBit	ID	26	
+-----+-----+-----+			
b	ID	40	
+-----+-----+-----+			
colors	ID	10	
+-----+-----+-----+			
division	ID	22	

+-----+-----+-----+			
entero	ID	16, 17	
+-----+-----+-----+			
f	ID	32	
+-----+-----+-----+			
floaat	ID	37	
+-----+-----+-----+			
g	ID	33	
+-----+-----+-----+			
h	ID	34	
+-----+-----+-----+			
l	ID	40	
+-----+-----+-----+			
main	ID	1	
+-----+-----+-----+			
max	ID	6	
+-----+-----+-----+			
modulo	ID	23	
+-----+-----+-----+			
n	ID	41	
+-----+-----+-----+			
notBit	ID	29	
+-----+-----+-----+			
orBit	ID	27	
+-----+-----+-----+			
pi	ID	7, 39	
+-----+-----+-----+			
product	ID	21	
+-----+-----+-----+			
s	ID	9	
+-----+-----+-----+			
sum	ID	20	
+-----+-----+-----+			
u	ID	8	
+-----+-----+-----+			
wrongFloat	ID	37	
+-----+-----+-----+			
x	ID	3, 12, 20, 21, 22, 23, 26, 27, 28, 29	
+-----+-----+-----+			

xorBit	ID	28	
+-----+	+-----+	+-----+	+-----+
y	ID	4, 12	
+-----+	+-----+	+-----+	+-----+
z	ID	5	
+-----+	+-----+	+-----+	+-----+
(OPERATOR	1, 22	
+-----+	+-----+	+-----+	+-----+
)	OPERATOR	1, 22	
+-----+	+-----+	+-----+	+-----+
,	OPERATOR	10(2), 12	
+-----+	+-----+	+-----+	+-----+
;	OPERATOR	3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 16, 17,	
		20, 21, 22, 23, 26, 27, 28, 29, 32, 33,	
		34, 37, 38, 39, 40, 43	
+-----+	+-----+	+-----+	+-----+
=	OPERATOR	3, 4, 5, 6, 7, 8, 9, 17, 20, 21, 22, 23,	
		26, 27, 28, 29, 32, 33, 34, 37, 38, 39,	
		40, 41	
+-----+	+-----+	+-----+	+-----+
{	OPERATOR	1, 10, 11	
+-----+	+-----+	+-----+	+-----+
}	OPERATOR	10, 13, 44	
+-----+	+-----+	+-----+	+-----+
%	OPERATOR_ARITHM	23	
	ETIC		
+-----+	+-----+	+-----+	+-----+
*	OPERATOR_ARITHM	21, 32	
	ETIC		
+-----+	+-----+	+-----+	+-----+
+	OPERATOR_ARITHM	20	
	ETIC		
+-----+	+-----+	+-----+	+-----+
/	OPERATOR_ARITHM	22	
	ETIC		
+-----+	+-----+	+-----+	+-----+
&	OPERATOR_BITWIS	26	
	E		
+-----+	+-----+	+-----+	+-----+

^	OPERATOR_BITWIS	28	
	E		
+-----+-----+-----+-----+			
	OPERATOR_BITWIS	27	
	E		
+-----+-----+-----+-----+			
~	OPERATOR_BITWIS	29	
	E		
+-----+-----+-----+-----+			
-3	LITERAL_INT	9	
+-----+-----+-----+-----+			
0	LITERAL_INT	43	
+-----+-----+-----+-----+			
1	LITERAL_INT	26, 27, 28	
+-----+-----+-----+-----+			
10	LITERAL_INT	3, 38	
+-----+-----+-----+-----+			
100	LITERAL_INT	6	
+-----+-----+-----+-----+			
2	LITERAL_INT	21	
+-----+-----+-----+-----+			
20	LITERAL_INT	20	
+-----+-----+-----+-----+			
3	LITERAL_INT	23	
+-----+-----+-----+-----+			
4	LITERAL_INT	22	
+-----+-----+-----+-----+			
5	LITERAL_INT	8, 17	
+-----+-----+-----+-----+			
"Cadena vá	LITERAL_STR	32	
lida"			
+-----+-----+-----+-----+			
#65	LITERAL_CHAR	34	
+-----+-----+-----+-----+			
'a'	LITERAL_CHAR	33	
+-----+-----+-----+-----+			
'c'	LITERAL_CHAR	5	
+-----+-----+-----+-----+			
2.5	LITERAL_DOUBLE	37	

3.14	LITERAL_DOUBLE	39	
3.14159	LITERAL_DOUBLE	7	
5.5f	LITERAL_DOUBLE	4	

Manual de usuario

Instalación

Para construir y ejecutar el proyecto, es necesario tener Java instalado en tu sistema. Sigue estos pasos para configurar el proyecto:

1. Clona el repositorio:

```
git clone https://github.com/AlonsoNav/CCompilerJFlex.git
cd your-repo
```

2. Genera el archivo CLexer:

```
java -jar lib/jflex-full-1.9.1.jar src/scanner/CLexer.flex
```

3. Genera el archivo Parser:

```
java -jar lib/java-cup-11b.jar -parser Parser -symbols Symbol
src/parser/Parser.cup
```

4. Compila el proyecto:

```
javac -d bin -sourcepath src -cp lib/java-cup-11b.jar
src/app/ParserMain.java src/app/ScannerMain.java
src/scanner/CLexer.java src/scanner/Token.java
src/scanner/TokenType.java src/parser/Parser.java
src/parser/Sym.java
```

Uso

Para ejecutar el compilador con un archivo de entrada, utiliza el siguiente comando:

```
java -cp bin app.ParserMain input_file
```

Bitácora

Fecha: 26-08-2024

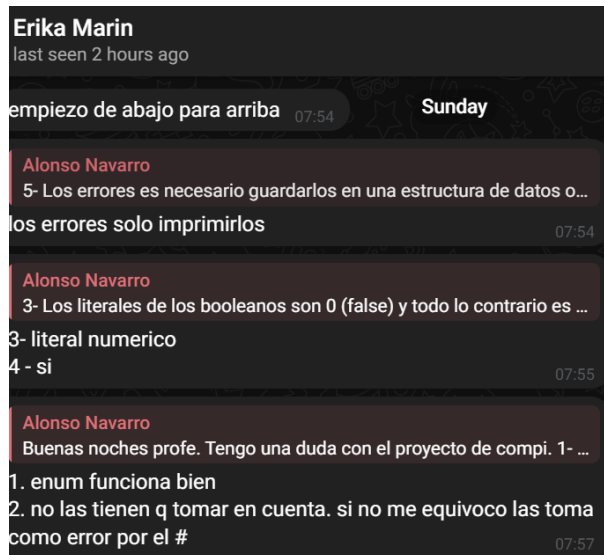
En la primera reunión del equipo de trabajo, se acordó que CV se encargará de las expresiones regulares de los operadores y del formato de impresión de la tabla. AN diseñará la estructura de los tokens y sus errores, así como las expresiones regulares de los identificadores y palabras reservadas. VG se responsabilizará de los literales. Por último, se decidió que la documentación se realizará en LaTeX y que GitHub será utilizado como sistema de control de versiones.

Fecha: 01-09-2024

La profesora responde las siguientes consultas:

- Nosotros ya manejamos los diferentes tipos de tokens en un enum. Queremos agregar subtipos pensando a futuro, pero no sé si sea mejor manejar cada subtipo de token como una clase aparte o si todo chorreado en un enum funciona. ¿Usted qué me recomienda?
- ¿Qué hacemos con las directivas para el procesador como `#include` o `#define`?
- Los literales de los booleanos son 0 (false) y todo lo contrario es true eso lo tomamos como un literal numérico no importa?
- Manejamos también sufijos (U: unsigned, L: long...)?
- Los errores es necesario guardarlos en una estructura de datos o solo con imprimirlos basta?

Lo siguiente son las respuestas de la profesora:



Fecha: 03-09-2024

La profesora responde la siguiente consulta:

Usted comentó que cosas como "1ejemplo", debería ser un error y no que los separe como "1": literal y "ejemplo": id. Hay otras cosas que se pegan, por ejemplo con las directivas:

```
#include <stdio.h >
```

lo separa como

```
#: error
```

```
include: id
```

```
<operador
```

```
stdio: id
```

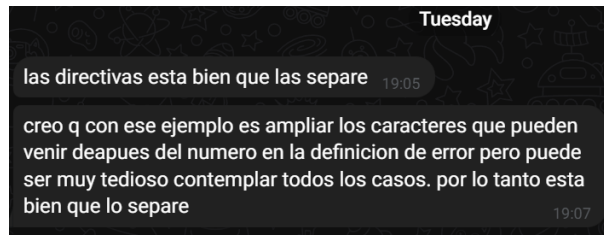
```
. operador
```

```
h: id
```

```
>operador
```

Entonces yo ya tengo definido el error genérico de "1ejemplo", pero con esto estaba pensando en meter más caracteres en ese mismo error, pero es que si fuera un "5<identificador" en algún condicional ya cromá.

La respuesta de la profesora es:



Fecha: 04-09-2024

Luego de tener todo el scanner corriendo correctamente se decide que lo siguiente es finalizar la documentación. CV se encargará de la introducción, AN de la estrategia de solución y VG de las lecciones aprendidas. Las demás secciones son redactadas en conjunto. Finalmente, se acuerda que cada integrante hará dos casos de prueba y documentará lo encontrado.

Bibliografía

- [1] Klein, G., Rowe, S., & Décamps, R. (marzo de 2023). *JFlex User's Manual*. JFlex Team. En: <https://www.jflex.de/manual.html>.