INSTITUTO TECNOLÓGICO DE COSTA RICA

Proyecto #1: Scanner

Escuela de Ingeniería en Computación
Compiladores e Intérpretes IC-5701

Alonso Navarro Carrillo, c. 2022236435

Carlos Venegas Masis, c. 2022153870

Valeria, c.

Ing. Ericka Marín Schumann
II Semestre 2024

Tabla de contenidos

Introducción

Este proyecto se sitúa en la primera etapa del desarrollo de un compilador para el lenguaje de programación C, conocida como el Análisis Léxico. El objetivo principal de esta etapa es diseñar y construir un scanner que sea capaz de identificar y clasificar los diferentes tokens que conforman un programa en C. Para lograr esto, se utilizó la herramienta JFlex, la cual permite definir expresiones regulares que describen los patrones de los tokens a reconocer.

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 listado de tokens encontrados	100%	
Mostrar tipo de token, línea y cantidad de apariciones por cada token	100%	
Manejar 4 tipos grandes (operadores, literales, ids, palabras reservadas) de tokens	100%	
Ignorar comentarios en línea y bloque	100%	
Identificar todos los operadores válidos de C	100%	
Identificar todos los literales válidos de C	100%	
Identificar todos los identificadores válidos de C	100%	
Identificar todas las palabras reservadas de C	100%	
Definir errores léxicos	100%	

Lecciones aprendidas

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: # es un token inválido.
- Error en la línea 2: # es un token inválido.
- Error en cáscada en línea 20: Comentario de bloque sin cerrar.

Es importante resaltar que aunque el proyecto indique que no se pueden tener errores en cáscada, 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_ARITHMETIC	17, 18
+	OPERATOR_ARITHMETIC	15, 16
:	OPERATOR_RELATIONAL	2
<	OPERATOR_RELATIONAL	1
>	OPERATOR_RELATIONAL	1, 2
?	OPERATOR_RELATIONAL	2
10	LITERAL_INT	13
20	LITERAL_INT	14, 19
"Este es otro string con /* comentario en bloque */"	LITERAL_STR	18
"Este es un string con // comentario en linea"	LITERAL_STR	17

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. Compila el proyecto:

```
javac -d bin -sourcepath src src/app/Main.java
src/scanner/CLexer.java .\src\scanner\Token.java
.\src\scanner\TokenType.java
```

Uso

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

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

También puedes enviar la salida a un archivo .txt con

```
java -cp bin app.Main input_file > output.txt
```

Bitácora

Fecha: 26-08-2024

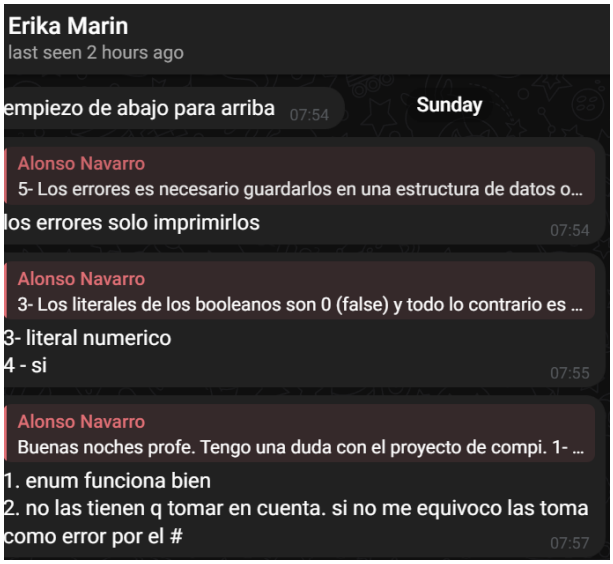
En la primera reunión del equipo de trabajo, se acordó que CV se encargará de los expresiones regulares de los operadores y del formato de impresión de la tabla. AN diseñará la estructura de los tokens y sus errores, así como las expresiones regulares de los identificadores y palabras reservadas. VG 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

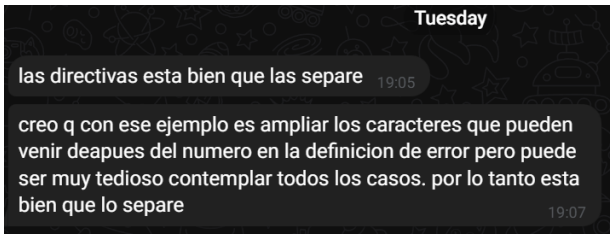
. operador

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 croma.

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`.