

Actividad 2. Preparando analizador léxico

El analizador léxico es la primera fase del proceso de compilación y su función es leer el código fuente carácter por carácter para transformarlo en una secuencia de tokens (unidades léxicas significativas). El lexer agrupa caracteres en lexemas y los clasifica en categorías como palabras reservadas (public, class), identificadores (nombres de variables), literales (números, cadenas), operadores (+, =, !=) y delimitadores (, , ;). Durante este proceso, elimina elementos irrelevantes como espacios en blanco y comentarios, mantiene una tabla de símbolos para distinguir palabras reservadas de identificadores, y reporta errores léxicos como caracteres inválidos o cadenas mal formadas. Lea cuidadosamente cada inciso y realizar lo solicitado.

- Para la parte de generación crear un repositorio en github y recuerde tenerlo organizado y separado por clases o módulos que considere necesarios.
- Para la parte escrita entregar un pdf con los ejercicios realizados de manera legible y ordenada.

Problema 1: 25%

Considere los siguientes fragmentos de código:

Fragmento 1 - Código en C:

```
1 float limitedSquare(x)
2 float x;
3 {
4     /* returns x-squared, but never more than 100 */
5     return (x<=-10.0||x>=10.0)?100:x*x;
6 }
```

Fragmento 2 - Código HTML:

```
1 Here is a photo of <b>my house</b>:
2 <p><img src = "house.gif"><br>
3 See <a href = "morePix.html">More Pictures</a> if you liked that one.<p>
```

- Realice el análisis léxico completo de ambos fragmentos de código. Para cada fragmento, identifique todos los lexemas y clasifíquelos por tipo.
- Determine cuáles lexemas de cada fragmento deben obtener valores léxicos asociados (atributos).
- Para cada lexema que requiera un valor léxico, especifique el tipo de valor que debe asociarse, el valor específico, y justifique por qué es necesario.
- Diseñe una estructura de datos unificada que pueda representar tokens de diferentes lenguajes de programación. Su estructura debe ser lo suficientemente flexible para manejar tanto lenguajes como C como lenguajes de marcado como HTML. Demuestre su uso representando al menos 5 tokens de cada fragmento.

Análisis Léxico
 Fragmento 1

Lexema	tipo
float	Keyword
limitedSquare	Id (function)
(Puntuación
x	Id (variable)
)	Puntuación
{	Puntuación
/* ---- */	Comentario (no lo reconoce)
return	Keyword
(puntuación
x	Id (variable)
;	Puntación
<	Operador
=	Operador
-10	Constante
	Operador (or)
x	Id (variable)
>	Operador
=	Operador
10	Constante
)	Puntuación
?	Puntación
100	Constante
:	Operador
X	Id (variable)
*	Operador
X	Id (variable)
;	puntuaciòn
}	Puntuaciòn

Fragmento 2

Lexema	Tipo
Here is a photo of	Texto
<	Delimiter
b	Operador
>	Delimiter close
My house	Texto
<	Delimiter
/b	Operador close
>	Delimiter close
:	
<	Delimiter
P	Operador
>	Delimiter close
<	Delimiter
Img	Operador
Source	Operador
=	operador
“	Delimiter
House.gif	String
“	Delimiter close
>	Close
<	Delimiter
Br	Operador
>	Delimiter close
See	Text
<	Delimiter
a	Operador
href	Operador
=	operador
“	Delimiter
morePix.html	String
“	Delimiter close
>	Close
More Pictures	text
<	Delimiter
/a	Operador close
>	Delimiter close
if you liked that one.	text
<	Delimiter
P	Operador
>	Delimiter close

Lexemas que si requieren atributos

Fragmento 1

Lexema	Tipo	¿Por qué necesita atributo?
limitedSquare	Identificador	Se necesita su nombre para tabla de símbolos
x	Identificador	Debe diferenciarse de otros identificadores
-10.0	Literal real	El valor numérico es necesario
10.0	Literal real	El valor numérico es necesario
100	Literal entero	Se usa directamente en la expresión
float	Palabra reservada	Asociada a un tipo de dato

No necesitan

Tipo	Ejemplos
Operadores	+, *, <=,
Delimitadores	{, }, (,), ;

Fragmento 2

Lexema	Tipo	¿Por qué necesita atributo?
"house.gif"	Literal string	Contiene la ruta del recurso
"morePix.html"	Literal string	URL del enlace
src	Atributo HTML	Indica tipo de atributo
href	Atributo HTML	Indica tipo de atributo
Texto plano	Texto	Se muestra/renderiza

No necesita

Tipo	Ejemplos
Etiquetas	, <p>,
Delimitadores	<, >
Operador	=

Especificar tipo de atributo

Fragmento 1

Lexema	Tipo de atributo	Valor asociado	Justificación
limitedSquare	string	"limitedSquare"	Identificar función
x	string	"x"	Identificar variable
-10.0	float	-10.0	Usado en comparación
10.0	float	10.0	Usado en comparación
100	int	100	Valor de retorno
float	enum / tipo	FLOAT	Define tipo de dato

Fragmento 2

Lexema	Tipo de atributo	Valor asociado	Justificación
"house.gif"	string	"house.gif"	Ruta de imagen
"morePix.html"	string	"morePix.html"	Enlace
src	string / enum	SRC	Tipo de atributo
href	string / enum	Href	Tipo de atributo
Texto plano	string	Texto literal	Renderizado

Estructura de datos

La estructura de datos propuesta para representar los tokens se puede diseñar unificada para que pudiera utilizarse tanto en lenguajes de programación como C, como en lenguajes de marcado como HTML. Cada token contiene un campo que indica su tipo (por ejemplo, palabra reservada, identificador, literal, operador o etiqueta), el lexema correspondiente tal como aparece en el código fuente, y un atributo opcional que almacena información adicional cuando es necesaria, como el valor numérico de un literal o el nombre de un identificador. Además, la estructura incluye información sobre el lenguaje de origen del token, así como la línea y la posición donde fue encontrado, lo cual facilita el manejo de errores y el análisis posterior. Este diseño permite manejar de manera flexible distintos tipos de tokens sin depender de un lenguaje específico.

Problema 2

Problema 2: 25%

Considere el siguiente fragmento de código en Java que contiene errores léxicos:

```
1 public int calculate Total(int x, double y) {  
2     int result = x * y;  
3     String message = "Result is: + result;  
4     double pi = 3.14.59;  
5     return result;  
6 }
```

- Identifique todos los errores léxicos presentes en el código.
- Para cada error léxico identificado, proponga una estrategia de recuperación específica que permita al analizador léxico continuar el análisis. Justifique por qué su estrategia es apropiada.
- Explique la diferencia entre un error léxico y un error sintáctico, dando un ejemplo de cada uno basado en el código anterior.

Errores Léxicos

- Calculate Total: se debe de escribir junto si no lo tomaría como dos identificadores
- "Result is: + result; falta un signo que cierra “ ” ”
- double pi = 3.14.59; falla por el temade que es un numero don doble decimal

Estrategia de recuperación

- Calculate Total: tomar solo un identificador, o calculate, o total, y reportar el error, así permite continuar el análisis sin bloquear y terminar el proceso
- "Result is: + result; Cerrar la cadena automáticamente, si no lo tomaría el documento en adelante todo como un string
- double pi = 3.14.59; ignorar el siguiente punto decimal, y tomar solamente 3.14. Así conserva este valor y continua

Diferencia entre errores

Error Léxico

double pi = 3.14.59;

Ocurre cuando los caracteres no pueden formar tokens válidos. El lexer no reconoce 3.14.59 como número válido.

Error sintáctico

int result = x * y;

Los tokens son correctos, pero la operación viola las reglas del lenguaje (tipos incompatibles). Ocurre cuando los tokens son válidos, pero la estructura del lenguaje es incorrecta.

Ejercicio 3

Explicación del funcionamiento del scanner

El scanner funciona leyendo el código fuente carácter por carácter y agrupando estos caracteres en tokens según reglas básicas del lenguaje. Cuando detecta letras, forma identificadores o palabras reservadas; si encuentra números, construye literales numéricos validando que tengan un formato correcto; y al encontrar comillas, reconoce cadenas de texto. Los símbolos como operadores y delimitadores se tokenizan directamente. Durante todo el proceso, el analizador mantiene el control de la línea y la columna donde aparece cada token, lo que facilita la detección y reporte de errores léxicos.

Modificaciones para implementar recuperación de errores

Para implementar recuperación de errores léxicos, el analizador podría modificarse para no detener la ejecución al encontrar un token inválido. En lugar de eso, se pueden registrar los errores y continuar el análisis ignorando o corrigiendo parcialmente el lexema incorrecto, por ejemplo truncando números mal formados o cerrando automáticamente cadenas no terminadas al final de la línea. Estas estrategias permiten que el lexer detecte múltiples errores en una sola ejecución, haciendo el análisis más robusto y útil para el programador.

Modificaciones para lenguajes sin espacios (como el japonés)

Si el código estuviera escrito en un idioma que no utiliza espacios para separar palabras, el proceso de scanning tendría que basarse completamente en reglas internas del lenguaje y no en separadores visibles. El analizador debería apoyarse en diccionarios léxicos, patrones estructurales y el contexto para decidir dónde termina un token y comienza otro. Además, sería necesario permitir múltiples interpretaciones temporales de los tokens hasta que el contexto sintáctico permita desambiguarlos, haciendo el proceso de tokenización más complejo que en lenguajes con separación por espacios.

Link del repositorio

https://github.com/Emadlgg/dise-o_lenguajes.git