UNIVERSIDAD DEL VALLE DE GUATEMALA

Diseño de lenguajes de programación

Sección 30

Ing. Pablo Koch

[M2] Actividad Práctica: Introducción al análisis léxico

Ihan Marroquin – 23108

GUATEMALA, 29 de enero del 2025

Problema 1: 25%

Considere los siguientes fragmentos de código:

Fragmento 1 - Código en C:

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

Fragmento 2 - Código HTML:

```html
Here is a photo of <b>my house</b>:
<p><img src = "house.gif"><br>
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.

Fragmento 1:

No.	Lexema	Tipo	Valores léxicos asociados (¿sí/no?)	Tipo de valor léxico asociado	Valor léxico asociado	Justificación
1	`float`	Keyword	No	-	-	Palabra reservada; el token en sí basta.
2	`limitedSquare`	Identificador (ID)	Sí	string ()	`"limitedSquare"`	Necesario para la tabla de símbolos.s

3	(	Puntuación	No	-	-	Marca inicio de lista de parámetros.
4	x	Identificador (ID)	Sí	string	"x"	Identificador del parámetro.
5	)	Puntuación	No	-	-	Cierre de lista de parámetros.
6	float	Keyword	No	-	-	Declaración del tipo del parámetro.
7	x	Identificador (ID)	Sí	string	"x"	Nombre del parámetro en la declaración.
8	;	Puntuación	No	-	-	Separador de declaración.
9	{	Puntuación	No	-	-	Inicio de bloque.
10	/* returns x-squared, but never more than 100 */	Comentario	No	-	–	Se ignora el comentario
11	return	Keyword	No	-	-	Palabra reservada de control de flujo.
12	(	Puntuación	No	-	-	Abre la expresión.

13	x	Identificador (ID)	Sí	string	"x"	Operando; referencia a variable.
14	<=	Operador.	No	-	-	Operador.
15	-	Operador.	No	-	-	Operador.
16	10.0	Constante	Sí	float	10.0	Valor numérico.
17	||	Keyword	No	-	-	-
18	x	Identificador (ID)	Sí	string	"x"	Referencia a variable.
19	>=	Operador	No	-	-	Operador; no requiere atributo.
20	10.0	Constante	Sí	float	10.0	Igual que arriba.
21	)	Puntuación	No	-	-	Cierra expresión.
22	?	Operador.	No	-	-	Operador; token suficiente.

23	`100`	Constante	Sí	int	`100`	Literal entero; used para valor máximo.
24	`:`	Puntuación.	No	-	-	Separador del operador ternario.
25	`x`	Identificador (ID)	Sí	string	`"x"`	Operando.
26	`*`	Operador.	No	-	-	Operador.
27	`x`	Identificador (ID)	Sí	string	`"x"`	Operando.
28	`;`	Puntuación.	No	-	-	Final de sentencia.
29	`}`	Puntuación.	No	-	-	Fin de bloque/función.

Fragmento 2:

No.	**Lexema**	**Tipo**	**¿Valores léxicos asociados?**	**Tipo de valor léxico asociado**	**Valor léxico asociado**	**Justificación**
1	`Here is a photo of`	Constantes	Sí	string	`"Here is a photo of "`	Texto literal fuera de tags
2	`<b>`	Opening tag	No	-	-	Marca inicio de tag.

3	my house	Constantes	Sí	string	"my house"	Texto interior de la etiqueta.
4		Closing tag	No	-	-	Marca inicio de tag de cierre.
5	:	Puntuación	No	-	–	Carácter de puntuación.
6	<p>	Opening tag	No	-	-	Inicio de <p>.
7	<img	Opening tag	No	-	-	Inicio de <img.
8	src	Identificador (ID)	Sí	string	"src"	Nombre del atributo que indica el origen de la imagen.
9	=	Operador	No	-	-	Asignación entre nombre de atributo y su valor.
10	"house.gif"	Constantes	Sí	string	"house.gif"	Valor literal del atributo src.
11	>	Closing tag	No	-	-	Cierra el tag <img ...>.
12	 	Opening tag	No	-	-	Inicio de .
13	See	Constantes	Sí	string	"See "	Texto literal antes del enlace.
14	<a	Opening tag	No	-	-	Inicio de <a.

15	`href`	Identificador (ID)	Sí	string	`"href"`	Nombre del atributo que indica destino del enlace.
16	`=`	Operadores	No	-	-	Asigna el valor al atributo `href`.
17	`"morePix.html"`	Constantes	Sí	string	`"morePix.html"`	Valor del atributo `href`
18	`>`	Closing tag	No	-	-	Cierre del start tag `<a ...>`.
19	`More Pictures`	Constantes	Sí	string	`"More Pictures"`	Texto que se muestra como enlace.
20	`</a>`	Closing tag	No	-	-	Inicio del cierre `</a>`.
21	`if you liked that one.`	Constantes	Sí	string	`" if you liked that one."`	Texto posterior al enlace.
22	`<p>`	Opening tag	No	-	-	Inicio de `<p>`.

Estructura de datos unificada:

- Lenguaje: C o HTML
- Lexema
- Tipo: Keyword, Operador, Identificador (ID), Constante o Puntuación
- Tipo_Valor_Asociado: int, String, Float, null, etc.
- Valor_Asociado: 10.0, “My House”, “See”, null, etc.

Ejemplos C:

1. float
    a. Lenguaje: C
    b. Lexema: float
    c. Tipo: Keyword
    d. Tipo_Valor_Asociado: null
    e. Valor_Asociado: null
2. `limitedSquare`
    a. Lenguaje: C
    b. Lexema: `limitedSquare`
    c. Tipo: Identificador (ID)
    d. Tipo_Valor_Asociado: String
    e. Valor_Asociado: `limitedSquare`
3. `return`
    a. Lenguaje: C
    b. Lexema: `return`
    c. Tipo: Keyword
    d. Tipo_Valor_Asociado: null
    e. Valor_Asociado: null
4. (
    a. Lenguaje: C
    b. Lexema: (
    c. Tipo: Puntuación
    d. Tipo_Valor_Asociado: null
    e. Valor_Asociado: null
5. 100
    a. Lenguaje: C
    b. Lexema: `100`
    c. Constante
    d. Tipo_Valor_Asociado: int
    e. Valor_Asociado: 100

Ejemplos HTML:

1. `Here is a photo of`
    a. Lenguaje: HTML
    b. Lexema: `Here is a photo of`
    c. Tipo: Constante
    d. Tipo_Valor_Asociado: String
    e. Valor_Asociado: `Here is a photo of`
2. `<b>`
    a. Lenguaje: HTML
    b. Lexema: `<b>`
    c. Tipo: Opening tag
    d. Tipo_Valor_Asociado: null
    e. Valor_Asociado: null
3. `</b>`
    a. Lenguaje: HTML
    b. Lexema: `</b>`
    c. Tipo: Closing tag
    d. Tipo_Valor_Asociado: null
    e. Valor_Asociado: null
4. `href`
    a. Lenguaje: HTML
    b. Lexema: `href`
    c. Tipo: Identificador (ID)
    d. Tipo_Valor_Asociado: String
    e. Valor_Asociado: “`href`”
5. `"house.gif"`
    a. Lenguaje: HTML
    b. Lexema: `"house.gif"`
    c. Tipo: Constante
    d. Tipo_Valor_Asociado: String
    e. Valor_Asociado: `"house.gif"`

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 detectados:

- Error léxico 1 — Línea 3: cadena no terminada
  - Texto problemático: String message = "Result is: + result;
    - Por qué es léxico: el analizador léxico leyó un " (comilla doble) y no encontró la comilla de cierre hasta el final de la línea/archivo, por lo que no puede formar un token válido.
- Error léxico 2 — Línea 4: literal numérico mal formado
  - Texto invalido: double pi = 3.14.59;
    - Por qué es léxico: la secuencia 3.14.59 no encaja en la sintaxis de literales numéricos de Java.

Estrategias de recuperación por cada error léxico:

- Error léxico 1 — Línea 3: cadena no terminada
  - Solución: colocar comillas luego de la palabra is, para la resolución del problema léxico:
    - “Result is” + result;
  - Justificación: Esta corrección permite que el flujo del compilador recupere sincronía con el parser.
- Error léxico 2 — Línea 4: literal numérico mal formado (puntos múltiples)
  - Solución: Eliminar el segundo punto de la declaración de pi
    - pi = 3.1459
  - Justificación: Al eliminar el segundo punto, el flujo del compilador vuelve al funcionar ya que al detectar un segundo punto en una misma declaración de double no puede continuar

Diferencia entre error lexico y sintactico:

- Error léxico: ocurre cuando el analizador léxico no puede transformar la secuencia de caracteres en una secuencia de tokens válidos.
    - Ejemplo: la cadena sin comillas de cierre en la línea 3 es un error léxico: no existe un tokenválido porque falta la " final.
- Error sintáctico: ocurre cuando la secuencia de tokens no encaja en la gramática del lenguaje
    - Ejemplo: la línea 1 “public int calculate Total(int x, double y) {“ produce los tokens:

KW public | KW int | ID calculate | ID Total | '(' | KW int | ID x | ',' | KW double | ID y | ')' | '{'

Aquí no hay un token inválido: calculate y Total son dos identificadores válidos. El error es que, según la gramática de Java, después de public int se espera un único identificador como nombre del método, no dos.

Problema 3: 50%

Considere el siguiente fragmento de código en Java:

```
1  public class PotionBrewer {
2      // Ingredient costs in gold coins
3      private static final double HERB_PRICE = 5.50;
4      private static final int MUSHROOM_PRICE = 3;
5      private String brewerName;
6      private double goldCoins;
7      private int potionsBrewed;
8
9      public PotionBrewer(String name, double startingGold) {
10         this.brewerName = name;
11         this.goldCoins = startingGold;
12         this.potionsBrewed = 0;
13     }
14
15     public static void main(String[] args) {
16         PotionBrewer wizard = new PotionBrewer("Gandalf, the Wise", 100.0);
17         String[] ingredients = {"Mandrake Root", "Dragon Scale", "Phoenix Feather"};
18
19         wizard.brewHealthPotion(3, 2);  // 3 herbs, 2 mushrooms
20         wizard.brewHealthPotion(5, 4);
21
22         wizard.printStatus();
23     }
24
25     /* Brews a potion if we have enough gold */
26     public void brewHealthPotion(int herbCount, int mushroomCount) {
27         double totalCost = (herbCount * HERB_PRICE) + (mushroomCount * MUSHROOM_PRICE
       );
28         if (totalCost <= this.goldCoins) {
29             this.goldCoins -= totalCost;  // Deduct the cost
30             this.potionsBrewed++;
31             System.out.println("Success! Potion brewed for " + totalCost + " gold.");
32         } else {
33             System.out.println("Not enough gold! Need: " + totalCost);
34         }
35     }
```

```
36
37      // Prints the current brewer status
38      public void printStatus() {
39          System.out.println("\n=== Brewer Status ===");
40          System.out.println("Name: " + this.brewerName);
41          System.out.println("Gold remaining: " + this.goldCoins);
42          System.out.println("Potions brewed: " + this.potionsBrewed);
43      }
44  }
```

Para este problema, debe implementar un analizador léxico (tokenizador) que procese el código anterior utilizando técnicas vista en clase (No use automatas, ni expresiones regulares para la solución).

- Diseñe e implemente un tokenizador.
- Implemente una **tabla de símbolos**.
- Genere como salida:
  - Una lista secuencial de todos los tokens encontrados con su clasificación
  - El contenido final de la tabla de símbolos mostrando todos los identificadores
  - El número de línea y posición donde se encontró cada token
- Explique detalladamente cómo funciona su scanner y el proceso de tokenización.
- Explique que modificaciones tendría que hacer a su código para implementar recuperación de errores.
- Si el código estuviera en japones, un idioma donde no se utilizan espacios que modificaciones tendría que hacer a su proceso de scanning.

```
PS C:\Users\Usuario\Desktop\7mo semestre\diseño de lenguajes de programacion>  c:; cd 'c:\Users\Usuario\Desktop\7mo semestre\diseño de lenguajes de programacion'; & 'C:\Program Files\Java\jdk-11\bin\java.exe' '-cp' 'C:\Users\Usuario\AppData\Roaming\Code\User\workspaceStorage\8538f06919b482b052fa0c86dcdc0bc2\redhat.java\jdt_ws\diseño de lenguajes de programacion_c813a3f0\bin' 'SimpleJavaTokenizer'
=== Tokens Encontrados ===
  1:1    "public"                Reservada
  1:8    "class"                 Reservada
  1:14   "PotionBrewer"          Identificador
  1:27   "{"                     Puntuacion
  3:5    "private"               Reservada
  3:13   "static"                Reservada
  3:20   "final"                 Reservada
  3:26   "double"                Reservada
  3:33   "HERB_PRICE"            Identificador
  3:44   "="                     Operador
  3:46   "5.50"                  Constante    <double>
  3:50   ";"                     Puntuacion
  4:5    "private"               Reservada
  4:13   "static"                Reservada
  4:20   "final"                 Reservada
  4:26   "int"                   Reservada
  4:30   "MUSHROOM_PRICE"        Identificador
  4:45   "="                     Operador
  4:47   "3"                     Constante    <int>
  4:48   ";"                     Puntuacion
  5:5    "private"               Reservada
  5:13   "String"                Reservada
  5:20   "brewerName"            Identificador
  5:30   ";"                     Puntuacion
  6:5    "private"               Reservada
  6:13   "double"                Reservada
  6:20   "goldCoins"             Identificador
  6:29   ";"                     Puntuacion
  7:5    "private"               Reservada
  7:13   "int"                   Reservada
  7:17   "potionsBrewed"         Identificador
```

```
=== Tabla de Símbolos ===
PotionBrewer        clase       -          1:14   apariciones=4
HERB_PRICE          campo       double     3:33   apariciones=2
MUSHROOM_PRICE      campo       int        4:30   apariciones=2
brewerName          campo       String     5:20   apariciones=3
goldCoins           campo       double     6:20   apariciones=5
potionsBrewed       campo       int        7:17   apariciones=4
name                campo       String     9:32   apariciones=2
startingGold        parametro   double     9:45   apariciones=2
main                desconocido -         15:24   apariciones=1
args                desconocido -         15:38   apariciones=1
wizard              desconocido -         16:22   apariciones=4
ingredients         desconocido -         17:18   apariciones=1
brewHealthPotion    desconocido -         19:16   apariciones=3
printStatus         desconocido -         22:16   apariciones=2
herbCount           campo       int       26:38   apariciones=2
mushroomCount       parametro   int       26:53   apariciones=2
totalCost           campo       double    27:16   apariciones=5
System              desconocido -         32:13   apariciones=6
out                 desconocido -         32:20   apariciones=6
println             desconocido -         32:24   apariciones=6
```

- Resumen del scanner:
    - Qué hace: Recorre el texto fuente y detecta piezas básicas: palabras reservadas, identificadores, constantes, operadores y signos de puntuación.
    - Cómo funciona: tiene un bucle que mira un carácter a la vez, salta espacios y comentarios, si encuentra una comilla construye una cadena, si encuentra dígitos construye un número, si encuentra letras construye un identificador o palabra reservada, para operadores y puntuación usa listas de símbolos simples y compuestos.
    - Seguimiento de contexto: guarda la línea y la columna donde aparece cada token y mantiene una “tabla de símbolos” con identificadores, su tipo y cuántas veces aparecen.
    - Salida: genera dos cosas:
        - una lista secuencial de tokens con su clasificación y posición.
        - el contenido final de la tabla de símbolos con los identificadores detectados.
- Modificaciones para implementar recuperación de errores:
    - Lo principal a añadir:
        - Un mecanismo de reporte de errores que registre tipo de error y posición y lo incluya en la salida.
        - Detectar errores locales como, carácter inesperado, literal mal cerrado, número mal formado y en lugar de lanzar o terminar, marcar un token de tipo error y continuar.
        - Registrar suficientes mensajes claros y ubicaciones para que el desarrollador identifique y arregle los problemas en el código fuente.
        - Añadir pruebas unitarias con ejemplos malformados para verificar que el scanner recupera y continúa.
- Si el código estuviera en japonés: qué cambiaría en el proceso de scanning
    - Problema principal: en japonés escrito no hay espacios que separen palabras y el tokenizador actual asume separadores para distinguir tokens.
    - Solución: introducir un paso de segmentación que divide la secuencia de caracteres en “candidatos de palabras” antes de aplicar las reglas del scanner, o integrar esa lógica dentro del scanner:
        - Usar un segmentador/analizador morfológico como un diccionario, que indique límites de palabra y ya con esa segmentación, el tokenizador puede reconocer palabras reservadas y nombres.
        - Aplicar la regla de “máxima coincidencia” para identificar palabras clave y operadores.
        - Para identificadores, permitir secuencias contiguas de caracteres japoneses como un identificador completo aunque no haya espacios, utilizar una lista de palabras reservadas en ASCII/unicode para detectar las palabras clave cuando se apliquen.

Repo:
https://github.com/Ihan-Marroquin/Actividad_2.git