Informe del Proyecto

Compiladores e intérpretes

Franco Zanardi

LU: 116256

Etapa 1: Analizador léxico

Compilación y ejecución del proyecto

Para compilar el código fuente primero se debe ubicar en la ruta donde se haya descomprimido el *zip* descargado. Luego, deberá ejecutar el siguiente comando:

> *> javac -encoding UTF-8 -d ./out "@sources.txt"*

Una vez compilado, para ejecutarlo deberá acceder a la carpeta generada *out*, estando allí podrá ejecutar el programa haciendo uso del siguiente comando:

> *> java ar.edu.uns.cs.minijava.Main <ruta del archivo fuente de minijava>*

Tokens reconocidos por el analizador léxico

Antes de comenzar, debemos declarar algunos conjuntos y explicar algunas convenciones adoptadas para facilitar el entendimiento de las expresiones regulares.

Cuando definimos la expresión regular para un token llamado x la notamos $E(x)$. Nuestras expresiones regulares están definidas sobre el alfabeto $\Sigma = \{x \mid x\ es\ cualquier\ carácter\ unicode\ válido\ o\ x\ es\ el\ carácter\ especial\ EOF\}$

A su vez, se definen los siguientes conjuntos:

$Cm = \{x \mid x\ es\ una\ letra\ minúscula\ válida\ en\ el\ alfabeto\ inglés\}$

$CM = \{x \mid x\ es\ una\ letra\ mayúscula\ válida\ en\ el\ alfabeto\ inglés\}$

$C = Cm \cup CM$

$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

También, se define el carácter especial EOFque representa el final del archivo. A su vez, el carácter ⏎ representa el carácter o la secuencia de caracteres que representan una nueva línea (independientemente del sistema operativo).

Finalmente, aclaradas estas convenciones, las expresiones regulares asociadas a cada token reconocido por el analizador léxico son:

- **Palabras claves.**

  A continuación mostramos el nombre de token asociado a cada palabra clave y su expresión regular de una manera genérica:

$Sea\ x \in Pr\ donde\ Pr = \{class,\ extends,\ static,\ dynamic,\ void,$
$boolean,\ char,\ int,\ String,\ public,\ private,\ if,\ else,\ for,$
$return,\ this,\ new,\ null,\ true,\ false\}$

$Tenemos\ que$:

$$E(pr_x) = x$$

$donde\ pr_x\ representa\ el\ nombre\ del\ token\ x,\ el\ cual\ será\ pr\_\ concatenado\ a\ x.$

- **Identificador de método y variable.**

$$E(idMetVar) = xy^* \qquad tal\ que\ x \in Cm,\ y \in C \cup D \cup \{\_\}$$

- **Identificador de clase.**

$$E(idClase) = xy^* \qquad tal\ que\ x \in CM,\ y \in C \cup D \cup \{\_\}$$

- **Literal entero.**

$$E(entero) = d\{1,9\} \qquad tal\ que\ d \in D$$

Siguiendo esta notación de llaves para indicar cuánto se repite el carácter a su izquierda.

- **Literal carácter.**

$$E(carácter) = '\,c_0\,' \mid '\backslash c_1\,'$$

$$tal\ que\ c_0 \in \Sigma - \{⏎, \backslash, ', EOF\},\ c1 \in \Sigma - \{⏎, ', EOF\}$$

- **Literal string.**

$$E(string) = "(c_0 | \backslash c_1)^*\ "$$

$$tal\ que\ c_0 \in \Sigma - \{⏎, \backslash, ", EOF\},\ c1 \in \Sigma - \{⏎, ", EOF\}$$

- **Signos de puntuación.**

$$E(parentesis\_abre) = ($$

$$E(parentesis\_cierra) =)$$
$$E(llave\_abre) = \{$$
$$E(llave\_cierra) = \}$$
$$E(punto\_y\_coma) = ;$$
$$E(coma) = ,$$
$$E(punto) = .$$

- **Operadores.**

$$E(mayor) = >$$
$$E(menor) = <$$
$$E(mayor\_igual) = >=$$
$$E(menor\_igual) = <=$$
$$E(negacion) = !$$
$$E(distinto) = !=$$
$$E(comparacion) = ==$$
$$E(suma) = +$$
$$E(resta) = -$$
$$E(producto) = *$$
$$E(division) = /$$
$$E(modulo) = \%$$
$$E(and) = \&\&$$
$$E(or) = ||$$

- **Asignación.**

$$E(asignacion) = =$$
$$E(incrementor) = ++$$
$$E(decrementor) = --$$

- **EOF.**

$$E(eof) = EOF$$

Errores detectables por el analizador léxico

- ***x* no es un símbolo válido**: se presenta cuando se lee un símbolo *x* no reconocido por el analizador léxico.
- ***x* no es un carácter válido**: se presenta cuando se lee un literal carácter *x* inválido. El analizador léxico comenzará a leer un literal carácter cuando encuentre una comilla simple en su estado inicial. Así, un literal carácter inválido será aquel que comience con comilla simple pero que luego no respete coincida con la expresión regular asociada a los literales caracteres.
- ***x* debe finalizar con */ para cerrar el bloque de comentarios**: se presenta cuando se comienza a leer un bloque de comentarios *x* pero este nunca es cerrado con los caracteres */.
- ***x* debe finalizar con comillas dobles para ser un string válido**: se presenta cuando se comienza a leer un literal string *x*, pero no se encuentran las comillas dobles en la línea para que finalicen la cadena correctamente.
- ***x* debe finalizar con tres comillas dobles seguidas (""") para ser un string en bloque válido**: se presenta cuando se comienza a leer un string en bloque *x*, pero no se encuentran las tres comillas esperadas para finalizar el string en bloque.
- ***x* es un entero que supera el límite establecido de 9 dígitos**: se muestra cuando se lee un entero *x* que posee más de 9 dígitos.

Explicación sobre la solución adoptada para el analizador léxico

A continuación se explicará cómo dividimos el proyecto en esta primera etapa y cuáles fueron las principales decisiones de diseño.

El proyecto se podría decir que cuenta con tres módulos:

- **El gestor de archivos**: busca abstraer al analizador léxico de cómo se maneja el archivo en sí.
- **El analizador léxico**: es el analizador léxico en cuestión.
- **El módulo encargado de mostrar los resultados por pantalla**: abstrae al analizador léxico de cómo se van a mostrar los datos por pantalla.

El gestor de archivos

Para la implementación del gestor de archivos decidimos utilizar únicamente la clase *FileReader* y leer carácter a carácter. Las principales decisiones de diseño asociadas a este módulo son:

- En el momento que se abre el archivo se lee por completo. El resultado se guarda en una lista de Strings. Cada elemento de la lista será un String que contenga una línea del archivo. Esto se realiza con el objetivo de facilitar el manejo del archivo (poder volver a líneas anteriores fácilmente, entre otras).
- A medida que se guarda el contenido del archivo, la codificación de una nueva línea adopta una codificación uniforme. Esta codificación uniforme es la adoptada por los sistemas UNIX, es decir el carácter *line feed* (0x0A en la tabla ASCII). Para adoptar esta codificación uniforme, todo carácter o secuencia de caracteres **LF** (0x0A), **CRLF** (0x0D 0x0A) o **CR** (0x0D) es interpretado como una nueva línea, por lo que es traducido a **LF** (0x0A) y almacenado así en el String que contiene dicha línea del archivo (artículo sobre las diferentes representaciones de una nueva línea). Se escogió esta solución dado que la clase *BufferReader* en su método *readLine()*, interpreta cualquiera de esas tres codificaciones mencionadas como una nueva línea (fuente: Oracle BufferReader documentation).
- Siguiendo con el inciso anterior, todo los módulos restantes del programa se podrán abstraer de la codificación de una nueva línea e interpretarla como el carácter *line feed*.

El analizador léxico

Para la implementación del analizador léxico en sí, decidimos hacer uso de la opción de implementar un autómata finito. Para representar cada estado del autómata utilizamos un método diferente. A continuación se detallan las principales decisiones de diseño:

- Para evitar tener una clase con todos los métodos del autómata (lo cual dificultaría su mantenimiento a futuro), se optó por crear un paquete llamado *automata*, el cual contiene todas las clases que implementan el autómata finito en cuestión.
- Todas estas clases usan el patrón de diseño *singleton* y todos sus métodos son accesibles únicamente dentro del paquete en cuestión.

- Además, cada clase representa un fragmento del autómata, donde los estados que contiene ese fragmento guardan una relación entre sí (generalmente los estados en conjunto reconocen algún token o un conjunto de tokens similares).
- Estas clases heredan de una clase abstracta llamada *Automata*, en su mayoría es una herencia para reutilización de código y no por comportamiento.
- Dentro del paquete hay una clase llamada *HandlerAutomata*, la cual también implementa el patrón de diseño *singleton*. Sin embargo, esta clase es accesible fuera del paquete *automata*. Esta tiene el estado inicial y es la clase que se encarga de unir los fragmentos del autómata. Además es la clase que se comunica con el exterior del paquete.
- Para la implementación del estado inicial en la clase *HandlerAutomata*, se buscó una alternativa a hacer una extensa secuencia de *if-then-else* para cada "ramificación" del autómata a partir del estado inicial. La alternativa usada fue usar un *HashMap* de carácter a *LazyTokenEvaluation*.
- *LazyTokenEvaluation* es una interfaz, la cual únicamente tiene un método *eval* que retorna un *Token* y lanza una excepción *LexicalException*. Esta interfaz se implementa, utilizando expresiones *lambda*, cada vez que se agrega una entrada al *HashMap* mencionado.
- Toda la implementación del autómata usa una clase llamada *CharacterUtils* la cual define qué es una letra válida, qué es un dígito válido, etc. De esta forma se busca una abstracción en el autómata sobre qué caracteres son válidos.

El módulo encargado de mostrar los resultados por pantalla

Para mostrar los mensajes por pantalla se optó por crear una clase *InfoDisplay*, la cual se encarga de mostrar los principales mensajes asociados al analizador léxico. Esta clase también es la encargada de mostrar los mensajes de error obtenidos a partir de la clase *LexicalException*. Las principales decisiones a resaltar sobre cómo se muestran los mensajes son:

- Cuando se produce un error cuyo lexema es multilínea, el código de error imprime únicamente la primera línea y se le concatena el string “...”, para indicar que hay más de una línea en error.
- El número de línea y columna en error mostrado es donde comienza el error. A su vez, el error elegante señala la columna por donde comienza el error.
- El mensaje descriptivo sobre el error mostrado, muestra el lexema completo, incluyendo todas sus líneas.

Logros que se intentan alcanzar en esta etapa

- Imbatibilidad Léxica
- Bloques de Texto como Strings
- Reporte de Error Elegante
- Columnas
- Multi-detección de Errores Léxicos