Práctica. Primera Fase

Desarrollo de analizadores léxicos

En esta primera parte debe realizarse el siguiente trabajo:

1) Desarrollo manual de un analizador léxico para **Tiny(0)**, un subconjunto de **Tiny** (véase el **apéndice A**). Para ello, deberá entregarse:
   - Un apartado en la memoria con las siguientes secciones:
     - Enumeración de las clases léxicas de **Tiny(0)**. Para cada clase debe incluirse, además, una descripción informal, en lenguaje natural.
     - Una especificación formal del léxico del lenguaje mediante definiciones regulares.
     - Diseño de un analizador léxico para el lenguaje mediante un diagrama de transiciones.
   - Una implementación manual, en Java, del analizador léxico. Debe proporcionarse, además un programa de prueba que acepte como argumento el archivo a procesar, y genere como salida una descripción legible de la secuenciad de tokens reconocida, y, si procede, un mensaje legible de error léxico detectado. Dicha implementación deberá, además, estar acondicionada para su funcionamiento y prueba a través del juez (véase el **apéndice B**).
2) Desarrollo de un analizador léxico completo para **Tiny** mediante la herramienta **JFlex**. Para ello, deberá entregarse:
   - Un apartado en la memoria con las siguientes secciones:
     - Enumeración de las clases léxicas de **Tiny**. Para cada clase debe incluirse, además, una descripción informal, en lenguaje natural.
     - Una especificación formal del léxico del lenguaje mediante definiciones regulares.

     Importante: (obviamente) en esta segunda parte de la memoria **no** hay que incluir diagrama de transiciones
   - Una implementación basada en **JFlex** del analizador léxico. Para ello deberá entregarse la especificación de entrada a **JFLex**, y todos los archivos Java adicionales requeridos por la implementación. Debe proporcionarse, además, un programa de prueba que acepte como argumento el archivo a procesar, y genere como salida una descripción legible de la secuencia de tokens reconocida, y, si procede, un mensaje legible de error léxico detectado. . Dicha implementación deberá estar también acondicionada para su funcionamiento y prueba a través del juez (véase el **apéndice B**).

La memoria deberá incluir una portada en la que aparezcan los nombres y apellidos de los integrantes del grupo, y el número de grupo.

Fechas:

- Presentación en el laboratorio de las memorias: **lunes 3 de febrero 2024**
- Presentación en el laboratorio de las implementaciones: **lunes 10 de febrero de 2024**
- Fecha límite de entrega: **Viernes 14 de febrero de 2024, a las 23:59h**.

La entrega final se realizará:

- A través del campus virtual, en un único .zip. Dicho archivo debe contener: (i) un documento PDF `memoria_lexico.pdf` con la memoria requerida; (ii) una carpeta `implementacion_manual`, en el interior de la cuál debe incluirse la implementación manual del analizador léxico para **Tiny(0)**; (iii) una carpeta `implementacion_jflex`, en el interior de la cuál debe incluirse la implementación jflex del analizador léxico para **Tiny**; (iv) una carpeta `pruebas_tiny_0` con distintos programas de prueba que permiten probar el analizador léxico para **Tiny(0)**; y (v) una carpeta `pruebas_tiny` con distintos programas de prueba que permitan probar el analizador léxico para **Tiny**. La entrega debe ser realizada solamente por un miembro del grupo.
- A través del juez **DomJudge** de la asignatura se entregará la implementación manual del analizador léxico para Tiny(0), y la implementación JFlex del analizador léxico completo para Tiny.

Apéndice A

Tiny(0) es un subconjunto de **Tiny** que incluye únicamente las siguientes características:

- Declaraciones de variables, con tipos básicos **int**, **real** y **bool**.
- Únicamente instrucciones eval. Las expresiones asociadas incluyen únicamente:
  - Literales enteros, reales, y booleanos (**true** y **false**).
  - Variables.
  - Operadores aritméticos +,-,* y /, menos unario (-), los operadores lógicos **and**, **or** y **not**, los operadores relacionales (<,>,<=,>=,==,!=), y el operador de asignación (=).
- Las prioridades y asociatividades de estos operadores son las de **Tiny**. Como en **Tiny**, pueden utilizarse paréntesis para alterarlas.

Ejemplo de programa **Tiny(0)**

```
{
  real peso;
  bool pesado
  &&
  @ peso = (45.0 * 12e-56) / -2.05;
  @ pesado = (peso >  10.0) or (peso / 2 <= +0.0)
}
```

Apéndice B

Para poder validar y entregar vuestros trabajos en el juez tenéis que preparar una clase `DomJudge`, ubicada en el paquete por defecto (es decir, en la raíz del proyecto, sin cláusula `package`). Esta clase tendrá un método `main` que, utilizando el analizador léxico, irá imprimiendo, uno por línea, los *lexemas* de los tokens reconocidos, o ERROR en caso de error léxico. En el caso de las palabras reservadas, los correspondientes lexemas deberán escribirse entre <...> y en minúsculas (por ejemplo, `<while>`).

Para conseguir esto, necesitaréis también tener en cuenta las siguientes indicaciones:

- Cuando el analizador descubra un error léxico, en lugar de abortar la ejecución, deberá levantar una excepción. Os recomiendo usar, como tipo de la excepción, una subclase de `RuntimeException` para evitar sobrecargar distintos métodos con cláusulas `throws`. A fin de que pueda seguir funcionando el `main` de prueba (el que muestra un listado legible de clases léxicas), deberéis capturar allí la excepción, obtener el mensaje encapsulado en la misma (podéis construir la excepción con dicho mensaje), e imprimirlo. En el caso del `main` de la clase `DomJudge`, deberéis capturar también la excepción, pero, en este caso, imprimir simplemente ERROR.
- El `main` de `DomJudge` leerá por la entrada estándar, y escribirá por la salida estándar. Para que el analizador lea por la entrada estándar, hay que encapsular en un `InputStreamReader` a `System.in` (el *reader* resultante puede pasarse al analizador).
- En la implementación manual, cuando descubráis un error, aparte de levantar la excepción, debéis obtener el siguiente carácter (llamar al método `sigCar`).
- El `main` de `DomJudge` irá recuperando tokens, e imprimiendo los lexemas. En el caso de las "unidades multivaluadas", el método `lexema` proporciona directamente el lexema. En el caso de las univaluadas, podéis añadir a `ClaseLexica` un constructor que permita indicar el *lexema* cuando se declaren los distintos enumerados, y un método para recuperar dicho lexema. En la implementación manual del analizador en el campus virtual se implementa esta idea.
- En la implementación *jflex*, **deberéis añadir la opción %public a la especificació**n (si no, y si la clase que se genera está en un paquete, no será visible para `DomJudge`).

En las implementaciones del analizador manual y el analizador *jflex* en el campus virtual se pueden ver todos estos detalles. Lo único que no aparece es la captura de la excepción en la clase `alex.Main` (implementación *jflex*), ni en el método `main` de `AnalizadorLexicoTiny` (implementación manual), pero esto lo podéis hacer vosotros/as.

Una vez que habéis preparado vuestro proyecto para poder ser validado en DomJudge, **tenéis que empaquetarlo como un .zip**, que es el archivo que subiréis al juez. Dicho .zip deberá contener todo lo que hay dentro de la carpeta **src** de vuestro proyecto (lo que hay en la carpeta, no la carpeta src en sí). O, en otras palabras, empaquetará todos los fuentes de vuestro proyecto en un .zip (únicamente

fuentes, no incluyáis otro tipo de archivo, porque pueda aumentar el peso y no ser aceptado por el juez).
Así mismo, en caso de la implementación *jflex*, necesitaréis incluir, en la raíz del proyecto, junto a DomJudge.java, un archivo llamado **ruta_jflex**. Dentro de este archivo debe estar el 'path' relativo de la especificación jflex (por ejemplo, si esta especificación está en la carpeta **alex**, el contenido de **ruta_jflex** deberá ser **alex/** Ojo con la /, que hay que ponerla, es la ruta relativa completa de la carpeta en la que está la entrada a *jflex*). Así mismo, el archivo con la entrada a *jflex* debe llamarse obligatoriamente **spec.jflex**.
Los ejemplos del CV (analizadores para el lenguaje `Eval`) son .zip empaquetados en el formato que puede subirse al juez. Podéis consultar en ellos todos los detalles (incluido el de **ruta_jflex** en la implementación *jflex*).
Una vez que tenéis el archivo .zip con vuestra entrega, se sube al juez como una solución codificada en un lenguaje ficticio que he llamado **procleng** (al subir un .zip es la única opción que os va a salir).
Es imprescindible seguir todos los convenios de empaquetado descritos, ya que el 'ejecutor' asociado a procleng es un 'script' que asume dichos convenios en el .zip que recibe como entrada, a fin de compilar y ejecutar el proyecto sobre los distintos casos de prueba.
En el concurso hay tres problemas dados de alta:

- `01 Analizador léxico Tiny(0)`, para la entrega de la implementación manual del analizador para Tiny(0)
- `02 Analizador léxico Tiny`, para la entrega de la implementación *jflex* del analizador para Tiny(1)
- EJEMPLO-LEXER. Aquí puede enviarse cualquiera de los dos .zip para el analizador léxico para Eval que hay en el campus virtual.

Ejemplos de entrada / salida:

Ante una entrada como la que sigue (no tiene que ser un programa Tiny válido, basta con que sea tokenizable):

```
_hola  35e-5 56 % And
```

el *main* de la clase `DomJudge` implementada debería producir una salida del tipo:

```
_hola
35e-5
56
%
<and>
EOF
```

y, ante una entrada como

un falso comentario

una salida como

```
ERROR
un
falso
comentario
EOF
```

Observad que el proceso no se detiene cuando se descubre un error: el carácter que ha producido el error simplemente se descarta, se levanta una excepción, se captura en el *main*, se imprime ERROR, y el proceso continúa (es por eso que en la implementación manual, cuando se descubre un error, antes de levantar la excepción debe llamarse a `sigCar`; en la implementación *jflex* esto se hace automáticamente, ya que ha casado el patrón `[^]`).