Trabajo Práctico Especial nº 1

Materia: Autómatas, Teoría de Lenguajes y Compiladores.

Profesores:

- Santos, Juan Miguel
- Arias Roig, Ana María
- Pose, Alberto Miguel

Alumnos:

- Magnorsky, Alejandro
- Mata Suárez, Andrés

Fecha de entrega: 06/05/2011

2011

Índice

Consideraciones realizadas

Asumimos que los nombres de los elementos sólo podían estar formados por palabras con letras minúsculas del idioma inglés (incluyendo el símbolo “!” que estaba en el archivo de nombres de ejemplos). A su vez, asumimos que el usuario no puede poner enters en el medio del comando ya que, debido a que el segundo elemento no puede tener enters en el medio, dicho comportamiento sería inconsistente. Puede haber tantos tabs y espacios como quiera en cualquier lugar del comando.

Cuando un usuario que se encuentra ejecutando la aplicación ingresa un comando inválido, es decir, que no es de la forma elementoA + elementoB, entonces se imprime un mensaje de error pero la aplicación continúa ejecutándose. En cambio, si el usuario quiere usar archivos de nombres de elementos o de magias que no tienen un formato válido, la aplicación le avisa con un mensaje de error adecuado y finaliza ya que no es lógico que siga ejecutándose con los datos mal cargados.

Con respecto a los argumentos en línea de comandos, se optó, a modo de evitar complicaciones innecesarias, dar prioridad al orden de ejecución de cada comando. De ese modo, se permite introducir varios comandos en una misma ejecución sin que haya conflictos entre ellos ni comportamiento no deseado. Por ejemplo, que el comando -m no colisione con el -n si no fueron cargados los nombres aún, o bien, que al introducir primero el comando -t y luego el -n, se ejecute -t sobre la lista de elementos nueva. Dicho esto, el orden de ejecución es: -n y -m, -h, -t, -b, -e.

Por otro lado, con la misma justificación de evitar posibles comportamientos no ideales, se tomó la decisión de, para los casos de múltiple ingreso del mismo comando para aquellos que requieran de un parámetro (ya sea archivo o elemento), sólo tomar como válido el argumento de la última ocurrencia del comando.

A modo de ejemplo, si se ingresara:

$> java -jar Alchemy.jar -e water -t -h -e fire -b earth

Se ejecutarían los comandos en el siguiente orden y de la siguiente manera:

-h → Imprime mensaje de ayuda.

-t → Imprime los elementos terminales.

-b earth → Se muestran aquellos elementos básicos necesarios para formar 'earth'.

-e fire → Se muestran aquellos elementos derivados de 'fire' (se ignora por completo el parámetro 'water' ingresado en una primera instancia).

Descripción del desarrollo del TP

Decidimos implementar la aplicación usando capas.

La primera capa, sería una especie de View-Controller, está formada por la clase Alchemy que utiliza el AlchemyLexer para parsear las instrucciones que le llegan por entrada estándar. Además es la encargada de interpretar los argumentos por línea de comandos e imprimir las respuestas que le llegan de la capa de servicios.

La segunda capa es la capa de Servicios, formada por Alchemist que usa SpellBook para resolver algunos requisitos.

La tercera capa es la de acceso a los datos donde están incluídas las clases ElementsLoader y MagicLoader, que hacen uso de los respectivos lexers para parsear los archivos que contienen la información de nombres y magias.

Por otro lado, está la capa del Model conformada por Element y Potion.

Además, implementamos un Junit Test Case para testear el parseo de los comandos, que forma la parte más importante de la aplicación. El mismo usa dos archivos, uno con todos comandos inválidos y el otro con todos válidos. Verifica si la salida es la esperada, comparándola con los archivos ya creados que contienen el comportamiento deseado.

Para los archivos .jflex que parsean la entrada estándar y el archivo de magias, usamos la transición de estados que provee JFlex. El archivo .jflex que parsea los nombres de elementos tiene una lógica más sencilla así que usamos sólo expresiones regulares para matchear las cadenas correctas y detectar si había algún error.

Dificultades encontradas en el desarrollo del TP

Al principio, nos costó encontrar un ejemplo simple del uso de JFlex cuya sintaxis difiere un poco de la de Lex. Habiendo encontrado un buen ejemplo nos resultó fácil implementar métodos dentro del .jflex, agregar imports, definir estados, variables, etc.

El parseo de la entrada estándar y del archivo de magias presentó la dificultad de encontrar una forma que validara únicamente eso, lo cual llevo a realizar más de una implementación. Al principio no usamos estados pero implementando el parseo de magic y utilizando dicha herramienta nos pareció muy útil y decidimos emplearla también para validar la entrada estándar.

Futuras extensiones

Podría agregársele la condición de unir más de 2 elementos entre sí. En dicho caso, primero habría que considerar si la agregación de elementos sigue siendo conmutativa y asociativa. Habría que modificar los .jflex para que la validación contemple la unión de más de 2 elementos. Además las pociones tendrían una lista de elementos de tamaño dinámico.

Otra modificación posible sería hacer que el orden de la unión influya en el elemento resultante. Dicho cambio es más fácil de implementar que el anterior.

Conclusión

Aplicando JFlex, que brinda manejo de expresiones regulares, estados, y otras herramientas, desarrollamos una aplicación que distingue minuciosamente entre lo que es un formato aceptado de otro que no, tanto en la entrada estándar como en archivos. De esta forma proporcionamos mayor robustez y confiabilidad al programa.