Procesadores del Lenguaje

PECL2: Construcción de un analizador sintáctico

Mario Ríos Muñoz
09047759L
9 Noviembre 2014

Indice:

Abstracto:

El objetivo de esta práctica es familiarizarse con el funcionamiento de la utilidad JCup, aprender a conectar la parte del análisis léxico (JLex) con el análisis sintáctico (JCup), diseñar una gramática para un leguaje dado, e implementar el análisis de la misma con las herramientas ya citadas.

A continuación se explican los aspectos más destacables y menos evidentes de la implementación de la práctica.

Nomenclatura y estilo seguido:

El autor ha decidido denotar los símbolos terminales como cadena de mayúsculas. Las etiquetas se han denotado como el tipo ('A' de abrir, 'C' de cerrar) seguido del nombre del bloque que delimitan. Por ejemplo, para el bloque de fecha, las etiquetas son ADATE, CDATE.

Los símbolos no terminales se han denotado como cadena de minúsculas. Además de los no terminales más obvios, el autor ha decidido declarar y usar ciertos no terminales auxiliares para facilitar la recursividad y la legibilidad.

Un ejemplo de esto se puede apreciar en la regla de fecha:

```
bloque_descripcion→ADESCR identificacion fecha CDESCR
fecha→ADATE DATE CDATE | ε
```

Lo mismo se podría haber denotado con la regla:

```
bloque_descripcion→ADESCR AIDENT TEXT CIDENT ADATE DATE CDATE CDESCR
                  | ADESCR AIDENT TEXT CIDENT CDESCR
```

pero el autor considera la primera opción mucho mas legible.

Se ha tratado de escribir el código de la manera más limpia posible, indentando cuidadosamente todas las líneas, y evitando líneas de longitud mayor a las 80 columnas, para facilitar el visionado del código en ventanas pequeñas.

Estructura de archivos y clases:

La implementación de esta práctica se ha distribuido entre los siguientes ficheros:

- Analizador.java: Es la clase de más alto nivel. Contiene el método main ejecutable, se asegura de que los parámetros recibidos en la llamada cumplen los requisitos (extensión de archivos correcta p.e.), y se encarga de instanciar el analizador léxico y el sintáctico.
- insClass.java: Cada instancia de esta clase almacena una instrucción aceptada del texto analizado. La implementación de esta clase es similar a la suministrada por el profesor en la documentación de la práctica.
- parser.cup: Fichero de especificación sintáctica. Implementa la gramática del lenguaje y todos las subtareas necesarias para el análisis sintáctico.
- parser.java: Implementación del autómata LALR(1) que ejecuta el análisis sintáctico. Esta clase es generada automáticamente por la herramienta JCup a partir del fichero anterior.

- storeClass.java: Cada instancia de esta clase almacena un registro o grupo de registros, de los aceptados por la gramática en el texto analizado. La implementación de esta clase ha sido ligeramente modificada respecto a la provista por el profesor.
- sym.java: Tabla de símbolos terminales reconocidos por la gramática. Esta clase es también autogenerada por JCup a partir de parser.cup
- UpdComp.lex: Especificación léxica del lenguaje. Versión extendida (en cuanto a lexemas reconocidos) del realizado en la práctica anterior.
- UpdComp.lex.java: Implementación del autómata que realiza en análisis léxico. Esta clase es generada automáticamente por JLex a partir de la especificación anterior.

Acerca de la modificación de storeClass:

Debido a la forma en que se planteó la gramática del lenguaje, fue necesario añadir un método que insertarse un Stack de registros después de ser instanciada la clase. Considerando el fragmento de la gramática en cuestión:

grupo→AGROUP contenido_registro AREGISTERS sub_registros CREGISTERS CGROUP

contenido_registro→ANAME NAME CNAME AUSE use CUSE ABITSIZE BITSIZE CBITSIZE regbitcode

Donde 'grupo' denota un grupo de registros.

El motivo de la producción 'contenido_registro' es que el alumno se percató de que un grupo tenía los mismos atributos que un registro, además de otro atributo adicional. Para evitar repetir código, se creó un no terminal a partir de la regla común a ambas secciones.

Es en la producción contenido_registro en la que se crea la clase storeClass. Como a la hora de reducir esta regla no se sabe si se está en un registro o en un grupo, no hay forma de añadir ese posible subconjunto de registros mediante el constructor. Tiene que hacerse al reducir por la 1ª regla (grupo) especificada. Por lo tanto, es necesario un método público, que permita introducir en la case un Stack con el subconjunto de registros que se han ido capturando en el análisis.

Más adelante se darán más detalles sobre la captura de información del texto analizado.

Interpretación de la documentación:

A continuación se especifica cómo se han interpretado ciertos aspectos del lenguaje a analizar:

Según la documentación, el atributo de entrada de un registro (<in>…</in>), debe ser un identificador de registro o las palabras reservadas DATA o MEM, delimitados por corchetes y separados por comas. Se entiende que el identificador de registro es el nombre del mismo, que según la especificación léxica debe ir entre comillas simples. Por lo tanto, esta gramática acepta en el bloque <in>…</in> la cadena ['A',DATA,MEM] pero no [A,DATA,MEM].

Con el bloque<out>…</out> se llega a la misma conclusión que en el caso anterior. Se acepta <out>['A']</out> pero no <out>[A]</out>

En el caso del bloque <behav> … </behav>, se entiende que el contenido delimitado por corchetes y separado por comas es un texto que carece de espacios en blanco.

De la misma forma que un archivo .java no contiene errores de sintaxis, se ha interpretado que para el fichero .upd debe pasar lo mismo. Por ello, el axioma de este lenguaje puede anularse, y si se intenta parsear un fichero vacío, el analizador no encontrará errores sintáctico.

Aspectos a destacar de la implementación:

Sobre el tratamiento de errores:

Tal y como se decía en la documentación, los errores se capturan en los tres grandes bloques principales (descripción, almacenamiento e instrucciones), mediante producciones de error.

Para personalizar los mensajes de error, se ha re-escrito el método report_error. Con las modificaciones añadidas, el mensaje de error queda de la siguiente forma:

Error sintactico en linea [nº de linea]: '[token que ha generado el error]'

Para disponer del número de línea y contenido del token que ha generado el error, desde el analizador léxico se llama al constructor de la clase Symbol con los siguientes parámetros:

Symbol(sym.ADESCR, yyline, yyline, yytext())

Se pasa yyline dos veces porque el constructor requiere obligatoriamente de dos enteros, aunque para este caso sólo se necesite el número de línea. Con la función yytext se captura el valor real del token, para ser tratado posteriormente por las acciones sintácticas.

Sobre la captura de contenido:

Una de las características que se pedía del analizador sintáctico, es que fuese capturando el contenido de las sentencias aceptadas por la gramática, para luego ser imprimidos por pantalla a modo de informe.

JCup se basa en un analizador LALR(1), que como su nombre indica, es ascendente. Esto quiere decir que se parte de un conjunto de terminales, y se va subiendo en el árbol sintáctico hasta llegar al axioma. Por lo tanto se necesitaba una forma de poder 'arrastrar' todos los datos capturados de reducción en reducción.

Además, como en las reglas recursivas que definen una lista de atributos, se necesita capturar toda la lista, y los elementos de esta lista aparecen en orden inverso al real en el fichero, se optó para estos casos usar un Stack.

A continuación se presenta un ejemplo de como se ha hecho uso de dicho Stack en la mayoría de producciones de la gramática:

```
behav ::= behav:be COMA ACORCHETE TEXT:b CCORCHETE {:   be.push(b);
                                                       RESULT = be; :}
        | ACORCHETE TEXT:b CCORCHETE {: Stack stack = new Stack();
                                        stack.push(b);
                                        RESULT = stack;:}
        ;
```

Siempre se actúa de la misma manera, en la producción no recursiva se instancia el Stack, y se acumula el primer token. Se devuelve como valor de la producción (que es de tipo Stack) la nueva pila creada.

En las producciones recursivas, se acumula el token aceptado en la pila del no terminal.

La creación de las diferentes clases ya depende del bloque en el que se realice.

Para el caso del bloque de instrucciones, se tiene una clase por cada bloque de instrucción capturado. Por tanto cuando se reduce la regla de instrucción, se crea dicha clase a partir de los valores de los elementos que la componen:

```
instruccion ::= AINS AOPCODE TEXT:op COPCODE entrada salida AINSBITCODE
                insbitcode:ins CINSBITCODE ABEHAV behav:be CBEHAV CINS
                {:  insClass instruction = new insClass(op,ins,be);
                    RESULT = instruction;:}
                ;
```

```
<use>”Index"</use>
<use>
        "AllPurpose","Accumulator","ProgramPC","Index","FlagVector",
        “StackPointer"
</use>
```

Para el caso del bloque de almacenamiento, ya se ha hablado ligeramente de se capturan los datos. La clase se crea al reducir el no terminal contenido_registro. Este no terminal puede haber sido reducido en un bloque registro o en un bloque grupo. Si es el caso del bloque registro, se llama al metodo setRegister. Si es el bloque de grupo, se llama al metodo setGroup y setRegisters. A este último se le pasa un Stack con el conjunto de subregistros del bloque.

Finalmente, para el bloque de descripción, no es necesario el uso de ninguna clase. A lo sumo, se tendrá que devolver la fecha y el identificador. Ambos son de tipo String, por lo que se pueden concatenar. A la hora de reducir la regla de descripción, se concatenan ambos valores (si no hubiese fecha se concatenaría como null), y se devuelven como un único String.

Finalmente, al reducir la regla del axioma, se imprime el valor de cada uno de los bloques. En el caso de almacenamiento e instrucciones, se hace mediante el método toString. Para evitar errores de NullPointerException, causados en potencia por un error sintáctico, se comprueba que el valor no sea null. En caso de que lo fuese, se le da un valor “N/A”, y es lo que se imprime.

Otros aspectos:

Se pedía validar que el contenido de la fecha tuviese sentido. Para ello se ha usado un SimpleDateFormat. El formato deseado es de tipo “dd/MM/yyyy”. La forma de actuar es: se parsea el valor de fecha. Si esto causa excepción se captura, se imprime un warning y se actualiza el valor de fecha a la cadena “Date Wrong”. Si por lo contrario todo es correcto, se devuelve el valor de fecha.

Se ha optado por sólo imprimir un warning y no lanzar un error de sintaxis porque, en teoría, una fecha correcta es sintácticamente correcto. Es más un error de semántica, y esta práctica sólo trata hasta el análisis sintáctico

---

Casos de prueba:

Se entregan dos fichero de prueba .upd: correcto.upd y con_fallos.upd (los nombres decriben suficientemente bien el objetivo de cada uno).

Hay que dejar claro que ha pesar de entregar sólo dos ficheros, se han hecho más pruebas de las que estos representan. Por ejemplo, las pruebas de aquellas secciones opcionales se han llevado a cabo comentando aquella sección a probar, comprobando que no da error y descomentando posteriormente el bloque.

Algunas de las pruebas realizadas se comentan a continuación:

Pruebas sobre la corrección de la gramática:

Omisión de bloques optativos no da error: Se han comentado los bloques de fecha, in, out y se ha ejecutado un análisis. Tal como se espera, el resultado no ha arrojado errores.

Omisión de bloques obligatorios da error: Se han comentado algunos de los bloques obligatorios, de manera aleatoria. Para todos los casos el resultado ha sido el esperado: error de sintaxis.

Alteración del orden en bloque almacenamiento: En la práctica se da a entender que el orden de los medios de almacenamiento tiene que ser siempre uno o más registros, seguido de cero o más grupos. Se ha probado a analizar un fichero en el que primero va un grupo y después un registro. El resultado ha sido error de sintaxis, como debía ser.

Comprobación de producciones recursivas: Para todos los casos en los que se pueda tener un único elemento (del tipo que sea) o una lista de los mismos, se usa definición recursiva. Para comprobar que esta recursividad está funcionando apropiadamente, se ha probado ha introducir un solo campo, y a introducir varios campos seguidos (separados por coma si la gramática así lo indica). En ambos casos la sentencia ha sido aceptada. Por poner un ejemplo concreto:

ambos casos son aceptados.

Pruebas sobre la corrección de las acciones sintácticas:

Validación apropiada del atributo fecha: Se pedía comprobar que el valor de la fecha tuviese sentido. Se ha probado ha introducir una fecha correcta y una correcta. En ambos casos, el análisis sigue con normalidad. La única diferencia es que (tal y como se esperaba) cuando la fecha es incorrecta se imprime un 'warning: fecha incorrecta' y el valor de fecha que aparece en el informe es 'Date Wrong'.

Omisión de bloque no lanza excepciones al imprimir el informe: Se comprueba que cuando se omite un bloque principal entero (lo que genera error de sintaxis) en el informe aparece "N/A", y cuando se omite un atributo de algún bloque, el valor que aparece en el informe es 'null'.

Los datos del informe aparecen en el orden en que se encuentran en el texto: Se cumple siempre, a pesar de hacer un análisis de abajo a arriba, los datos aparecen en el orden correcto, gracias al uso de la estructura Stack.

Comentarios sobre el entorno de desarrollo:

Para la realización de esta práctica se ha usado netbeans, con una particularización: La creación de la implementación de los analizadores a partir de los ficheros de especificación .lex y .cup se ha realizado mediante un makefile que automatiza la tarea.

Es muy probable que dicho makefile no funcione en el ordenador del profesor, dado que para llamar a los comandos se utiliza una ruta absoluta, y el profesor puede haber instalado las herramientas en directorios diferentes. Aún así, dicho makefile es entregado junto con el resto de ficheros.

En netbeans, se ha tenido que añadir los directorios de JLex y JCup a la fuente del proyecto, para que los import de las clases de cada uno sean válidos. Es probable que el profesor tenga que repetir esta tarea en su IDE a la hora de corregir la práctica.