Procesadores de lenguajes. Práctica 3.

LA HERRAMIENTA JAVACC: ESPECIFICACIÓN LÉXICA

Índice

1.1 JavaCC

1.2 Descarga e instalación de JavaCC Plugin Eclipse

1.3 Descarga e instalación de JavaCC en raíz

1.4 Sintaxis de JavaCC
- 1.4.1 Opciones JavaCC
- 1.4.2 Analizador y definiciones JavaCC
- 1.4.3 Especificación léxica de la gramática de JavaCC
- 1.4.4 Especificación sintáctica de la gramática de JavaCC

1.5 Expresiones regulares en JavaCC

1.6 Acciones asociadas al reconocimiento de expresiones regulares en JavaCC

1.7 Ejemplo práctico

1.1 JavaCC

- Java Compiler Compiler (JavaCC), herramienta de generación automática de analizadores gramaticales basada en Java. Es un metacompilador.
- Analiza un fichero de entrada, que contiene la descripción de una gramática, y generar un fichero de salida, escrito en Java, que contiene la especificación de un analizador léxico y de un analizador sintáctico para la gramática.
- Características de la herramienta:
  - Herramienta más utilizada en Java.
  - Se basa en un análisis sintáctico descendente recursivo.
  - Por defecto JavaCC analiza gramáticas de tipo LL(1), pero permite LL(K) o adaptativo.
  - Especificaciones léxica y sintáctica de la gramática a analizar en el mismo fichero.

1.1 JavaCC

- Características de la herramienta:
  - Especificaciones léxica y sintáctica de la gramática a analizar en el mismo fichero.
  - La especificación léxica se basa en expresiones regulares y la especificación sintáctica utiliza el formato EBNF.
  - Hay dos utilidades además de la herramienta principal:
    - JJTree para crear automáticamente un generador de árboles sintácticos
    - JJDoc para generar automáticamente la documentación de la gramática en formato HTML.
  - La gestión de errores léxicos y sintácticos se basa en excepciones y contiene información respecto al origen del error y su posición.
  - Existe un plugin para Eclipse que facilita la edición y ejecución de la herramienta dentro del desarrollo de cualquier aplicación Java.

1.2 Descarga e instalación de JavaCC Plugin Eclipse

- Eclipse --> Help --> Eclipse Marketplace --> Search “javacc” --> install JavaCC Eclipse Plug-in 1.6.1 --> seleccionar todos y aceptar
- Terminada la instalación se reiniciará el entorno
- Creamos un proyecto java (Java Proyect)
- Creamos un package
- Añadimos un nuevo fichero sobre src --> New --> Other --> JavaCC Example File --> seleccionamos el package y le damos un nombre a la clase.
- Al crearlo se ejecutará sólo.
- En Eclipse es imprescindible guardar el archivo antes de ejecutarlo por segunda vez.

1.3 Descarga e instalación de JavaCC en raiz

- Si descargamos https://github.com/javacc/javacc/archive/javacc-7.0.13.zip y descomprimimos y observamos las carpetas no está la carpeta bin.
- Hay un archivo build que si lo ejecutamos nos pide ant. Ant se descarga de apache-ant https://downloads.apache.org/ant/binaries/apache-ant-1.10.13-bin.zip
- Si lo descomprimimos en raíz y añadimos la carpeta /bin a path podremos ejecutar ant.
- Ahora volvemos al directorio de javacc, con una cmd ejecutamos “ant dist” generará los .jar todas las herramientas.
- En el build.bat no funciona binary-distribution.
- Se podría usar como .jar.

1.4 Sintaxis de JavaCC

- Extensión .jj son archivos de entrada que contienen la especificación de la gramática a analizar. Estos archivos se dividen en cuatro partes:
  - 1ª. Seleccionar opciones, valor del Lookahead, métodos estáticos o no, distinguir minúsculas y mayúsculas, etc.
  - 2ª. Definir nombre del analizador e incluir el código Java que se va a añadir directamente a este analizador, nombre de paquete, claúsulas import, cabecera de definición de clase, constructores de clase, variables de instancia o métodos utilizados en la definición de la semántica de la gramática.
  - 3ª. Especificación léxica de la gramática.
  - 4ª. Descripción de las reglas sintácticas de la gramática.

1.4.1 Opciones JavaCC

- BUILD_PARSER
- BUILD_TOKEN_MANAGER
- CACHE_TOKENS
- CHOICE_AMBIGUITY_CHECK
- COMMON_TOKEN_ACTION
- DEBUG_LOOKAHEAD
- DEBUG_PARSER
- DEBUG_TOKEN_MANAGER
- ERROR_REPORTING
- FORCE_LA_CHECK
- GENERATE_ANNOTATIONS
- GENERATE_BOILERPLATE
- GENERATE_CHAINED_EXCEPTION
- GENERATE_GENERICS
- GENERATE_STRING_BUILDER
- GRAMMAR_ENCODING
- IGNORE_CASE
- JAVA_TEMPLATE_TYPE
- JAVA_UNICODE_ESCAPE
- JDK_VERSION
- KEEP_LINE_COLUMN
- LOOKAHEAD
- NO_DFA
- OTHER_AMBIGUITY_CHECK
- OUTPUT_DIRECTORY
- PARSER_SUPER_CLASS
- SANITY_CHECK
- STATIC
- SUPPORT_CLASS_VISIBILITY_PUBLIC
- TOKEN_EXTENDS
- TOKEN_FACTORY
- TOKEN_MANAGER_USES_PARSER
- UNICODE_INPUT
- USER_CHAR_STREAM
- USER_TOKEN_MANAGER

1.4.1 Opciones JavaCC

- LOOKAHEAD permite especificar el número de tokens a considerar para predecir la regla de producción a expandir. Se pueden dar gramáticas LL(k).
- STATIC métodos estáticos para inicializar las variables, hay que hacer uso del método ReInit(). En estático no es posible ejecución en paralelo.
- UNICODE_INPUT permite generar analizadores que tomen como entrada ficheros en formato Unicode. Por defecto los ficheros de entrada se toman en formato ASCII.
- IGNORE_CASE se usa para lenguajes en los que sean independientes del uso de mayúsculas o minúsculas, como SQL.

1.4.2 Analizador y definiciones JavaCC

```
PARSER_BEGIN(MiGramatica) /* definición del nombre del analizador */

/* codigo Java utilizado en la descripción del analizador */

import java.io.*;

public class MiGramatica {
   ...
}

PARSER_END(MiGramatica)
```

1.4.2 Ejemplo

```
PARSER_BEGIN(Example)

/** Simple brace matcher. */
public class Example {


  /** Main entry point. */
  public static void main(String args[]) throws
ParseException {
    Example parser = new Example(System.in);
    parser.Input();
  }

}

PARSER_END(Example)
```

1.4.3 Especificación léxica de la gramática de JavaCC

- Esta especificación se basa en cuatro tipos de declaraciones:
  - TOKEN define expresiones regulares que representan tokens del lenguaje que serán enviados al analizador sintáctico por parte del método getNextToken().
  - SKIP describe expresiones regulares de categorías que son filtradas por el analizador léxico, blancos y comentarios del lenguaje.
  - SPECIAL_TOKEN tokens que no son enviados directamente al analizador sintáctico, sino que se añaden en el campo specialToken del siguiente token reconocido.
  - MORE reconoce una secuencia que se considera un prefijo del siguiente token.

- Tras reconocer un MORE seguido de un TOKEN O SPECIAL_TOKEN. El lexema MORE se añade al comienzo del lexema TOKEN O SPECIAL_TOKEN.

- Es raro el uso de MORE y SPECIAL_TOKEN.

1.4.3 Especificación léxica de la gramática de JavaCC

```
* especificación léxica de la gramática */

TOKEN: /* Identificadores */
{
   < ID: ( <LETTER> )+ ( "_" | "$" | "#" | <DIGIT> | <LETTER> )* >
  | < #LETTER: ["A"-"Z", "a"-"z"] >
  | < #DIGIT: ["0"-"9"] >
}

SKIP:
{
    " "
  | "\n"
  | "\r"
  | "\t"
}
```

1.4.4 Especificación sintáctica de la gramática de JavaCC

- Esta especificación se corresponde a la definición de las reglas de producción asociadas a cada símbolo no terminal de la gramática.

```
tipo identificador ( lista_de_parámetros ) :
  {
    código_java
  }
  {
     descripción_EBNF
  }
```

1.4.4 Especificación sintáctica de la gramática de JavaCC

- JavaCC se basa en un análisis sintáctico descendente recursivo.
- Cada símbolo no terminal genera una función.
- La cabecera de la declaración sintáctica corresponde a la cabecera de la función asociada al símbolo no terminal descrito.
- El tipo se refiere al tipo de dato que devuelve la función, puede ser simple o un objeto de clase.
- El identificador se corresponde al nombre del símbolo no terminal, que coincide con el nombre de la función asociada.
- La lista de parámetros se refiere a los parámetros a utilizar en la llamada a la función y se describe igual que en un método en Java.
- El código Java es incluido literalmente y se suele utilizar para declarar variables que se usaran.

1.4.4 Especificación sintáctica de la gramática de JavaCC

- La descripción EBNF de la regla de producción permite utilizar todas las operaciones de este formato:
  - Disyunción '|'
  - Clausura '(' expresión ')'*
  - Clausura positiva '(' expresión ')'+
  - Opcionalidad '[' expresión ']'

- Para referirnos a tokens (símbolos terminales) se utiliza '<' nombre_token '>'.
- Para referirnos a símbolos no terminales se utiliza la expresión 'identificador(lista_parámetros), se introduce una llamada a la función asociada al símbolo no terminal.
- Los símbolos terminales devuelven un valor de tipo Token.

1.4.4 Especificación sintáctica de la gramática de JavaCC

```
* especificación sintáctica de la gramática */


void CreateTable() :
{
  Token token;
}
{
    <CREATE> <TABLE>
    token = <ID> { System.out.println("Table: " + token.image); }
    "(" ColumnList() ")"
}
```

1.5 Expresiones regulares en JavaCC

- Hay cuatro formas de escribir expresiones regulares en JavaCC´:
  - expresión_regular ::= “cadena”
  - expresión_regular ::= < identificador : expresión_regular_compleja >
  - expresión_regular ::= < identificador >
  - expresión_regular ::= < EOF >

- 1ª Para definir patrones constantes, ejemplo: “/*” o “\n” se suelen utilizar en declaraciones SKIP o MORE, no se asigna identificador a la expresión regular.

- 2ª Definen expresiones regulares complejas y les asigna un identificador. En declaraciones TOKEN o SPECIAL_TOKEN. Si comienza con # no define categoría léxica sino que es auxiliar.

- 3ª y 4ª Representan la referencia a otras expresiones regulares.

1.5 Expresiones regulares en JavaCC

- Las expresiones regulares complejas pueden contener cadenas de caracteres (“import”)
- Listas de caracteres entre corchetes (**["a"-"z","A"-"Z","0"-"9"]**)
- Referencias a otras expresiones regulares **<DIGITO>**
- Referencia con operadores de clausura **(<DIGITO>)***
- Referencia con operadores de clausura positiva **(<DIGITO>)+**
- Disyunciones **("import" | "IMPORT")**
- Opcionalidad **(<DIGITO>)?**
- Para representar cualquier carácter excepto algunos se utiliza ~["\\"]

1.5 Expresiones regulares en JavaCC

```
TOKEN:  {                                                    >
    <MAS: "+">                                               }
  |   <MENOS: "-">
  |   <POR: "*">                                             SKIP: {
  |   <DIV: "/">                                                " "
  |   <PARAB: "(">                                            |   "\n"
  |   <PARCE: ")">                                            |   "\r"
  |       <NUM:   (["0"-"9"])+   (   "."   (["0"-"9"])*   )?  |   "\t"
(<EXPONENT>)? >                                                }
  |   < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+
```

1.6 Acciones asociadas al reconocimiento de expresiones regulares en JavaCC

- La especificación de JavaCC permite incluir código Java que será ejecutado al reconocer una determinada expresión regular.
- Este código se incluye entre llaves tras la definición de la expresión regular.

```
TOKEN : {
    <CLASS: “class”>   { System.out.println(“Palabra class reconocida”); }
  |  <INTERFACE: “interface”>
}
```

1.6 Acciones asociadas al reconocimiento de expresiones regulares en JavaCC

- Dentro del código Java que describe las acciones se puede acceder a las siguientes variables y métodos del analizador léxico:
  - Image(StringBuffer): contiene la cadena de caracteres reconocida por medio de la expresión regular.
  - lengthOfMatch(int): longitud de la cadena reconocida. Sin contar MORE.
  - curLexState(int): código del contexto léxico actual. Los contextos se verán más adelante.
  - inputStream(InputStream): flujo de datos de entrada del analizador. En el momento de ejecutar la acción se posiciona en el último carácter reconocido, el método read() devuelve el siguiente carácter.
  - matchedToken(Token): categoría léxica que se va a devolver al analizador sintáctico, valor que se devuelve al método getNextToken() del analizador léxico.
  - Void SwitchTo(int código): modifica el contexto léxico actual.