Java CUP

Cup es un software escrito en lenguaje Java, que genera una compilador para un lenguaje determinado utilizando el método de parsing ascendente LALR. Fue desarrollado en el Instituto de Tecnología de Georgia (EE.UU.), tiene la característica de permitir introducir acciones semánticas escritas en dicho lenguaje.

Un archivo Cup está formado por cinco bloques principales:

1. Definición de paquete y sentencias import.
2. Código de usuario.
3. Listas de símbolos
4. Asociatividad y precedencia.
5. Gramática

Definición de paquete y sentencias import

En esta sección se indican opcionalmente las clases que se necesitan importar. También puede indicarse el paquete al que se quieren hacer pertenecer las clases generadas.

Ej.

Import java_cup.runtime.*;

Código de usuario.

En esta sección se puede incluir código Java que el usuario desee incorporar en el analizador sintáctico que se va a obtener con CUP. Existen varias partes del analizador sintáctico generado con CUP en donde se puede incluir código de usuario. Cup engloba en una clase no pública aparte (incluida dentro del fichero parser.java) a todas las acciones semánticas especificadas por el usuario:

action code {: bloque_java :}

En esta sección pueden declararse variables, funciones, etc. todas de tipo estático ya que no existen objetos accesibles mediante los que referenciar componentes no estáticos. Todo lo que aquí se declare será accesible a las acciones semánticas.

También es posible realizar declaraciones Java dentro de la propia clase parser, mediante la declaración:

parser code {: bloque_java :}

lo cual es muy útil para incluir el método main() que arranque nuestra aplicación.

Las siguientes dos declaraciones sirven para realizar la comunicación con el analizador léxico:

- init with {: bloque_java :}

ejecuta el código indicado justo antes de realizar la solicitud del primer token; el objetivo puede ser abrir un fichero, inicializar estructuras de almacenamiento, etc.

- scan with {: bloque_java :}
  esta declaración permite especificar el bloque de código que devuelve el siguiente token a la entrada.

Listas de símbolos

En esta sección se definen todos los terminales y no terminales de la gramática. Pueden tenerse varias listas de terminales y no terminales, siempre y cuando éstos no se repitan. Cada lista de terminales se hace preceder de la palabra terminal y cada lista de no terminales se precede de las palabras non terminal. Los elementos de ambas listas deben estar separados por comas, finalizando ésta en punto y coma.

Con estas listas también es posible especificar el atributo de cada símbolo, ya sea terminal o no, sabiendo que éstos deben heredar de Object forzosamente, es decir, no pueden ser valores primitivos. Los símbolos cuyo tipo de atributo sea coincidente se pueden agrupar en una misma lista a la que se asociará dicho tipo justo detrás de la palabra terminal, de la forma:

terminal NombreClase terminal1, terminal2, etc.;

o bien

non terminal NombreClase noTerminal1, noTerminal2, etc.;

Asociatividad y precedencia.

Esta sección permite resolver los conflictos desplazar/reducir ante determinados terminales.

- precedence left terminal1, terminal2, etc.;
  opta por reducir en vez de desplazar al encontrarse un conflicto en el que el siguiente token es terminal1 o terminal2, etc.
- precedence right terminal1, terminal2, etc.;
  opta por desplazar en los mismos casos.
- precedence nonassoc terminal1, terminal2, etc.;
  produciría un error sintáctico en caso de encontrarse con un conflicto desplazar/reducir en tiempo de análisis.

Pueden indicarse tantas cláusulas de este tipo como se consideren necesarias, sabiendo que el orden en el que aparezcan hace que se reduzca primero al encontrar los terminales de la últimas listas, esto es, mientras más abajo se encuentre una cláusula de precedencia, más prioridad tendrá a la hora de reducir por ella. De esta manera, es normal encontrarse cosas como:

precedence left SUMAR, RESTAR;

precedence left MULTIPLICAR, DIVIDIR;

lo que quiere decir que, en caso de ambigüedad, MULTIPLICAR y DIVIDIR tiene más prioridad que SUMAR y RESTAR, y que todas las operaciones son asociativas a la izquierda

Gramática.

La ultima sección incluye la gramática a reconocer, expresada mediante reglas de producción que deben finalizar en punto y coma, y donde el símbolo '**::=**' significa 'el no terminal se define como'.

La gramática puede comenzar con una declaración que diga cuál es el axioma inicial. Caso de omitirse esta cláusula se asume que el símbolo distinguido o inicial es la primera regla definida. Dicha declaración se hace de la forma:

start with noTerminal;

Es posible cambiar la prioridad y precedencia de una regla que produzca conflicto desplazar/reducir indicando la cláusula:

%prec terminal

Las reglas de producción pueden incluir acciones semánticas, que son delimitadas por los símbolos '**{:**' y '**:}**'. En estas acciones puede incluir bloques de código Java, siendo de especial importancia los accesos a los atributos de los símbolos de la regla. El atributo del antecedente se llama RESULT, mientras que los atributos de los símbolos del consecuente son accesibles mediante los nombres que el desarrollador haya indicado para cada uno ellos. Este nombre se indica a la derecha del símbolo en cuestión(terminal/no terminal) y separado de éste por el caracter '**:**'. Su utilización dentro de la acción semántica debe ser coherente con el tipo asignado al símbolo cuando se lo indicó en la lista de símbolos. Debe prestarse especial atención cuando se utilicen acciones semánticas intermedias ya que, en tal caso, los atributos de los símbolos accesibles del consecuente sólo podrán usarse en modo lectura, lo que suele obligar a darles un valor inicial desde JFlex.

Ej:

expr ::= expr MAS termino ;

se le daría significado semántico de la forma:

expr ::= expr:eX MAS termino:tE {:RESULT = new Integer(eX.intValue() + tE.intValue()); :};

El símbolo error permite la recuperación de errores mediante el método panic mode.

EJECUCIÓN

Una vez que hemos creado el fichero CUP, para obtener las clases Java que implementan el analizador sintáctico correspondiente hay que ejecutar:

```
java java_cup.Main opciones < sintactico_CUP
```

sintactico _CUP es el fichero en formato CUP que hemos creado para nuestro analizador sintáctico.

Es posible especificar algunas opciones, de la forma:

```
java java_cup.Main  opciones < ficheroEntrada
```

Algunas de las opciones más interesantes son:

- **package nombrePaquete**: las clases **parser** y **sym** se ubicarán en el paquete indicado.
- **parser nombreParser**: el analizador sintáctico se llamará **nombreParser** y no **parser**.
- **symbols nombreSímbolos**: la clase que agrupa a los símbolos de la gramática se llamará **nombreSímbolos** y no **sym**.
- **expect numero**: por defecto, Cup aborta la compilación cuando se encuentra con un conflicto reducir/reducir. Si se desea que el comportamiento sea idéntico al de PCYacc, esto es, escoger la primera regla en orden de aparición en caso de dicho conflicto, es necesario indicarle a Cup exactamente el número de conflictos reducir/reducir que se esperan. Ello se indica mediante esta opción.
- **progress**: hace que Cup vaya informando a medida que va procesando la gramática de entrada.
- **dump**: produce un volcado legible de la gramática, los estados del autómata generado por Cup, y de las tablas de análisis. Si se desea obtener sólo las tablas de análisis (utilizadas para resolver los conflictos), puede hacerse uso de la opción más concisa **dump_states** que omite la gramática y las tablas de análisis.

Para que Cup se comunique convenientemente con el analizador léxico, éste debe implementar la interface **java_cup.runtime.Scanner**, definida como:

```
public interface Scanner {
public Symbol next_token() throws java.lang.Exception;
}
```

next_token() devuelve un objeto de la clase Symbol que representa el siguiente Token de la cadena de Tokens que será la entrada para realizar el análisis sintáctico.

Para que nuestra aplicación funcione es necesario incluir una función **main()** de Java que construya un objeto de tipo **parser**, le asocie un objeto de tipo **Scanner** (que realizará el análisis lexicográfico), y arranque el proceso invocando a la función **parser()** dentro de una sentencia **try-catch**. Esto se puede hacer de la forma:

```
parser code {:
        public static void main(String[] arg){
                /* Crea un objeto parser */
                parser parserObj = new parser();
                /* Asigna el Scanner */
                Scanner miAnalizadorLexico = new Yylex(new InputStreamReader(System.in));
                parserObj.setScanner(miAnalizadorLexico);
                try{ parserObj.parse();
                }catch(Exception x){
                System.out.println("Error fatal.");
                }
        }
        :};
```

Cuando se produce un error en el análisis sintáctico, CUP invoca a los siguientes métodos:

```
public void syntax_error(Symbol s);

public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception;
```

Una vez que se produce el error se invoca el método syntax_error. Después, se intenta recuperar el error (el mecanismo que se utiliza para recuperar errores no se explica en este documento; si no se hace nada especial, el macanismo para recuperar errores falla al primer error que se produzca). Si el intento de recuperar el error falla, entonces se invoca el método unrecovered_syntax_error. El objeto de la clase Symbol representa el último Token consumido por el analizador. Estos métodos se pueden redefinir dentro de la declaración parser code {: ... :}.

JFlex

JFlex incorpora la opción **%cup** que equivale a las cláusulas:

```
%implements java_cup.runtime.scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
        return new java_cup.runtime.Symbol(sym.EOF);
%eofval}
%eofclose
```

El analizador lexicográfico debe devolver los *tokens* al sintáctico en forma de objetos de la clase **Symbol**. Esta clase, que se encuentra en el paquete **java_cup.runtime** y que, por tanto, debe ser importada por el lexicográfico, posee dos constructores:

• Symbol(int n). Haciendo uso de las constantes enteras definidas en la clase **sym**, este constructor permite crear objetos sin atributo asociado.

• Symbol(int n, Object o). Análogo al anterior, pero con un parámetro adicional que constituye el valor del atributo asociado al *token* a retornar.