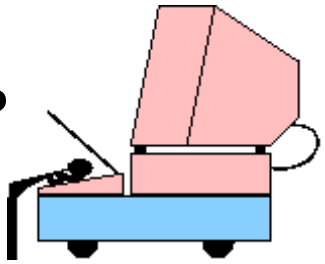
Breve Introducción a CUP



Definición de una gramática en el formato de entrada de CUP

Introducción

NOTA: este documento contiene una breve descripción de algunos de los elementos soportados por CUP. Existen muchas opciones o funcionalidades de CUP que no están descritas aquí. Por lo tanto, se recomienda consultar la documentación de CUP, disponible en la página Web "oficial" de CUP.

Como ejemplo para ilustrar como es un fichero de entrada para CUP, utilizaremos el fichero desarrollado para el lenguaje de programación presentado en clase:

```
package Parser;

import java_cup.runtime.*;
import AST.*;

parser code {:
public void syntax_error(Symbol s) {
  report_error("Error de sintaxis en linea " + s.left, null);
}

public void unrecovered_syntax_error(Symbol s) throws
  java.lang.Exception {
  report_fatal_error("", null);
}
:};

terminal PUNTOCOMA, ASOP, IF, THEN, ENDIF, IGUALQUE, PROG, IN, OUT, LOCAL,
MAS, ABRELLAVE, CIERRALLAVE, COMA, PAREN, TESIS;

terminal Integer CENT;
terminal Boolean CLOG, TIPO;
terminal String IDENT;

non terminal Prog Prog;
non terminal LDecl In, Out, Local, LDecl;
non terminal Decl Decl;
non terminal LVar LVar;
non terminal Sentencia Sent, SentSimp, Body;
non terminal Asignacion Asign;
non terminal Condicional Cond;
non terminal Exp Exp;

precedence left IGUALQUE;
precedence left MAS;
```

```
start with Prog;

Prog ::= PROG IDENT:i1 In:i2 Out:o Local:l Body:b {:RESULT=new
         Progv1(i1, i2, o, l, b); :}
     |   PROG IDENT:i1 In:i2 Out:o Body:b       {:RESULT=new
         Progv2(i1,i2,o,b); :} ;

In   ::= IN LDecl:l                             {:RESULT=l; :} ;

Out  ::= OUT LDecl:l                            {:RESULT=l; :} ;

Local::= LOCAL LDecl:l                          {:RESULT=l; :} ;

LDecl::= Decl:d PUNTOCOMA                       {:RESULT=new
         LDecl2(d); :}
     |   Decl:d PUNTOCOMA LDecl:l2              {:RESULT=new
         LDecl1(d, l2); :} ;

Decl ::= TIPO:t LVar:l                          {:RESULT=new
         Decl(t.booleanValue(), l); :} ;

LVar ::= IDENT:i                                {:RESULT=new
         LVar2(i); :}
     |   IDENT:i COMA LVar:l                    {:RESULT=new
         LVar1(i,l); :} ;

Body ::= ABRELLAVE Sent:s CIERRALLAVE           {:RESULT=s; :} ;

Sent ::= SentSimp:s1 PUNTOCOMA Sent:s2          {:RESULT=new
         SentenciaCompuesta(s1, s2); :}
     |   SentSimp:s PUNTOCOMA                   {:RESULT=s; :} ;

SentSimp::= Asign:s                             {:RESULT=s; :}
        |   Cond:s                              {:RESULT=s; :} ;


Asign::= IDENT:id ASOP Exp:e           {:RESULT=new
         Asignacion(id, e); :} ;

Cond::= IF Exp:e THEN Sent:s1 ENDIF
                                               {:RESULT=new
        Condicional(e, s1); :} ;


Exp::= CLOG:c                                {:RESULT=new
          ConstanteBooleana(c.booleanValue()); :}
      |   IDENT:s                                {:RESULT=new
          Variable(s); :}
      |   Exp:e1 MAS Exp:e2            {:RESULT=new Suma(e1,
           e2); :}
      |   CENT:n                                  {:RESULT=new
               ConstanteEntera(n.intValue()); :}
      |   PAREN Exp:e TESIS                     {:RESULT= e; :}
      |   Exp:e1 IGUALQUE Exp:e2              {:RESULT=new
               IgualQue(e1, e2); :} ;
```

Un fichero de entrada para CUP consta de las siguientes partes:

1. Definición de paquete y sentencias `import`.
2. Sección de código de usuario.
3. Declaración de símbolos terminales y no terminales.
4. Declaraciones de precedencia.
5. Definición del símbolo inicial de la gramática y las reglas de producción.

A continuación se describen cada una de estas partes.

Definición de paquete y sentencias `import`.

En esta sección se incluyen las construcciones para indicar que las clases Java generadas a partir de este fichero pertenecen a un determinado `package` o importar las clases Java necesarias. Para ambas cosas se utiliza exactamente la misma sintaxis que en Java. Esta parte contendrá como mínimo la siguiente línea:

```
import java_cup.runtime.*;
```

Nótese que puesto que las clases java generadas por CUP importan las clases del paquete `java_cup.runtime`, el directorio donde se encuentre este paquete deberá estar en el `classpath` tanto al compilar las clases como al ejecutar un programa en Java que las use (este paquete está incluido en las clases de CUP).

Sección de código de usuario.

En esta sección se puede incluir código Java que el usuario desee incluir 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. Aquí vamos a ver solamente una de ellas. La declaración:

```
parser code {:
...
:};
```

permite incluir código Java en la clase `parser`, generada por CUP. Como veremos más adelante, se pueden redefinir aquí los métodos que se invocan como consecuencia de errores de sintaxis.

Declaración de símbolos terminales y no terminales.

En esta sección se declaran los símbolos terminales y no terminales de la gramática que define el analizador sintáctico que deseamos producir. Tanto los símbolos no terminales como los símbolos terminales pueden, opcionalmente, tener asociado un objeto Java de una cierta clase. Por ejemplo, en el caso de un símbolo no terminal, esta clase Java puede representar los subárboles de sintaxis abstracta asociados a ese símbolo no terminal. En el caso de los símbolos terminales (o Tokens), el objeto Java representa el dato asociado al Token (por ejemplo un objeto de la clase `Integer` que represente el valor de una constante, o un `String` que represente el nombre de un identificador).

Para declarar símbolos terminales y no terminales se utiliza la siguiente sintaxis:

```
terminal [<nombre_clase>] nombre1, nombre2, ... ;
non terminal [<nombre_clase>] nombreA, nombreB, ... ;
```

Por ejemplo, en el fichero CUP de ejemplo que se presentó al principio del enunciado la declaración de símbolos terminales y no terminales era esta:

```
terminal PUNTOCOMA, ASOP, IF, THEN, ENDIF, IGUALQUE, PROG, IN, OUT, LOCAL,
MAS, ABRELLAVE, CIERRALLAVE, COMA, PAREN, TESIS;

terminal Integer CENT;
terminal Boolean CLOG, TIPO;
terminal String IDENT;

non terminal Prog Prog;
non terminal LDecl In, Out, Local, LDecl;
non terminal Decl Decl;
non terminal LVar LVar;
non terminal Sentencia Sent, SentSimp, Body;
```

```
non terminal Asignacion Asign;
non terminal Condicional Cond;
non terminal Exp Exp;
```

En este ejemplo, podemos ver definiciones de terminales que no tienen ningún objeto Java asociado (en la declaración `terminal PUNTOCOMA, ASOP, ...`), definiciones de terminales que tienen un objeto Java asociado (todas las demás) y definiciones de símbolos no terminales que tienen un objeto Java asociado (todas).

Como se ha estudiado en clase de teoría, un analizador sintáctico generado con un generador de analizadores LR (tipo al que pertenece CUP), durante el proceso del análisis sintáctico guarda internamente una cadena de símbolos gramaticales (símbolos terminales y no terminales) que representa el estado en el que se encuentra el analizador en cada paso del análisis sintáctico.

Cada uno de los símbolos gramaticales del estado del analizador sintáctico se representa en CUP por medio de un objeto de la clase `Symbol`:

```
public class Symbol {
  public int sym;
  public int left, right;
  public Object value;

  public Symbol(int id, int l, int r, Object o) {
    ...
  }

  public Symbol(int id, Object o) {
    ...
  }

  public Symbol(int id) {
    ...
  }
...
}
```

El significado de los atributos de un objeto de la clase `Symbol` es el siguiente:

- `sym`: identifica cuál es el símbolo terminal o no terminal representado por el objeto. A cada símbolo terminal y no terminal definidos en el documento se les asocia un número diferente que le identifica.
- `value`: si el objeto de la clase `Symbol` representa un símbolo terminal o no terminal que tenga asociado un objeto Java, dicho objeto se guarda en el atributo `value`.
- `left` y `right`: aunque su uso no es imprescindible, habitualmente se utilizan para identificar la línea de texto en la que comienza (`left`) y termina (`right`) la parte del programa que se corresponde con ese símbolo terminal o no terminal. Se usan cuando se produce un error de sintaxis para indicar el lugar donde se encuentra el error.

Declaraciones de precedencia.

En CUP, es posible definir niveles de precedencia y la asociatividad de símbolos terminales (como se verá más adelante, también se pueden definir niveles de precedencia y asociatividad ligándolos a reglas de producción concretas).

En el fichero CUP de ejemplo que se presento al principio de este documento, las declaraciones de precedencia eran:

```
precedence left IGUALQUE;
precedence left MAS;
```

Como podemos ver, las declaraciones de precedencia de un fichero CUP consisten en una secuencia de construcciones que comienzan por la palabra clave `precedence`. A continuación, viene la declaración de asociatividad, que puede tomar los valores `left`, `right` y `nonassoc`. Finalmente, la construcción termina con una lista de símbolos terminales separados por comas, seguido del símbolo punto y coma.

La precedencia de los símbolos terminales viene definida por el orden en que aparecen las construcciones `precedence`. Los terminales que aparecen en la primera construcción `precedence` son los de menor precedencia, a continuación vienen los de la segunda construcción `precedence` y así sucesivamente, hasta llegar a la última, que define a los terminales con mayor precedencia. En el ejemplo, el terminal `IGUALQUE` tiene menor precedencia que el terminal `MAS`.

La asociatividad se aplica cuando no es posible resolver una ambigüedad utilizando reglas de precedencia. La asociatividad de un símbolo terminal puede ser:

- `left`: el terminal asocia por la izquierda.
- `right`: el terminal asocia por la derecha.
- `nonassoc`: es un error de sintaxis que el árbol de derivación contenga dos derivaciones consecutivas en las que aparezca ese símbolo terminal. Por ejemplo, si hubiésemos declarado `IGUALQUE` como `nonassoc`, la expresión `a=b=c` produciría un error de sintaxis.

Definición del símbolo inicial de la gramática y las reglas de producción.

Para definir el símbolo inicial de la gramática se utiliza la construcción `start with...`:

```
start with Prog;
```

Para ver cómo definir las reglas de producción de la gramática veamos un ejemplo:

```
Prog ::= PROG IDENT:i1 In:i2 Out:o Local:l Body:b {:RESULT=new
         Progv1(i1, i2, o, l, b); :}
     |   PROG IDENT:i1 In:i2 Out:o Body:b       {:RESULT=new
         Progv2(i1,i2,o,b); :} ;
```

Para definir todas las reglas de producción que tengan a un mismo símbolo no terminal como antecedente, se escribe el símbolo no terminal en cuestión (en el ejemplo, `Prog`), seguido de `::=` y a continuación las reglas de producción que le tengan como antecedente, separadas por el símbolo `|` (en el ejemplo hay 2 reglas de producción). Después de la última regla de producción se termina con punto y coma.

Si nos fijamos, por ejemplo, en el consecuente de la primera regla de producción, vemos que está formado por una secuencia de símbolos terminales y no terminales (`PROG IDENT:i1 In:i2 Out:o Local:l Body:b`), algunos de los cuales llevan adyacente un símbolo de dos puntos seguido de un identificador. Recordemos que cuando se presentó la declaración de símbolos terminales y no terminales, se dijo que éstos podían tener asociado un objeto Java. Los identificadores que vienen después de un símbolo terminal o no terminal representan variables Java en las que se guarda el objeto asociado a ese símbolo terminal o no terminal. Estas variables Java pueden ser utilizadas en la parte `{: ... :}` que viene a continuación.

Entre `{: ... :}` se incluye el código Java que se ejecutará cuando se reduzca la regla de producción en la que se encuentre dicha cláusula `{: ... :}`.

Si el no terminal antecedente tiene asociada una cierta clase Java, obligatoriamente dentro de la cláusula `{: ... :}` habrá una sentencia `RESULT=...;`. El objeto Java guardado en la variable `RESULT` será el objeto Java que se asocie al no terminal antecedente cuando se reduzca la regla de producción en la que se encuentre esa cláusula `{: ... :}`.

A una regla de producción también se le puede asociar directamente una precedencia. Por ejemplo:

```
expr ::= MENOS expr:e
         {: ... :}
         %prec UMINUS
```

La directiva `%prec` permite asociar una precedencia a una regla de producción equivalente a la precedencia definida en la sección de declaraciones de precedencia al terminal que viene a continuación de `%prec` (en el ejemplo `UMINUS`).

Creación de las clases Java

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 < fichero_CUP
```

Donde `fichero_CUP` es el fichero en formato CUP que hemos creado para crear un analizador sintáctico. Para ver las opciones existentes (todas elllas son opcionales) consultar el manual "oficial" de CUP.

Uso de las clases Java

Para utilizar las clases Java obtenidas con CUP para realizar un análisis sintáctico, se seguira el proceso descrito a continuación.

Como sabemos, el analizador sintáctico consume los tokens generados por el analizador léxico. En CUP, al crear el analizador sintáctico (que será un objeto de la clase `parser` creada por CUP al ejecutar `java_cup.Main`), se le pasa al constructor como argumento un objeto que es el analizador léxico:

```
lexer l= new ...;
parser par;
par= new parser(l);
```

El analizador léxico (en el código de ejemplo anterior, el objeto de la clase `lexer`) sólo debe de cumplir el requisito de ser de una clase Java que implemente el siguiente interfaz:

```
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. El primer Token que se le pasa al analizador sintáctico al invocar `next_token` será el de más a la izquierda. Un analizador léxico obtenido con JLex se ajusta a estos requsitos.

Para realizar el análisis sintáctico se invoca el método `parse()`, que devuelve un objeto de la clase `Symbol` que representa al símbolo no terminal raíz del árbol de derivación que genera la cadena de Tokens de entrada. Si queremos obtener el objeto Java asociado a dicho símbolo no terminal, deberemos acceder al atributo `value` del objeto de la clase `Symbol` obtenido:

```
p= par.parse().value;
```

Como el objeto almacenado en el atributo `value` es de la clase `Object`, normalmente se realizará un casting para convertirlo a la clase adecuada, como por ejemplo:

```
p= (Prog) par.parse().value;
```

Gestión de errores en CUP

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 {: ... :}`.

En el ejemplo, se redefinen los métodos `syntax_error` y `unrecovered_syntax_error`:

```
parser code {:
public void syntax_error(Symbol s) {
  report_error("Error de sintaxis en linea " + s.left, null);
}

public void unrecovered_syntax_error(Symbol s) throws
  java.lang.Exception {
  report_fatal_error("", null);
}
:};
```

Se supone que el atributo `left` del objeto de la clase `Symbol` que representa cada Token tiene un valor igual al número de la línea en la que estaba el Token en el fichero de entrada del programa que está siendo analizado.

Last Revision: 01/23/2015 19:08:29