Adela del Río Ortega
Raquel Yanes Megías

i. A cerca de la Versión 0.10 del CUP

La versión 0.10 de CUP añade muchos cambios y características nuevas sobre las versiones anteriores a la versión 0.9. Estos cambios intentan hacer el CUP más parecido a su precursor, YACC. Por consiguiente, las especificaciones del analizador gramatical 0.9 para el CUP no son compatibles y será necesario leer el apéndice C del nuevo manual para poder escribir las nuevas especificaciones. La nueva versión, sin embargo, le da al usuario más poder y posibilidades, haciendo las especificaciones del analizador gramatical más fáciles de escribir.

1.-Introducción y Ejemplo

Este manual describe el funcionamiento básico y el empleo del constructor de analizadores gramaticales CUP(Constructor of Useful Parsers) basado en el lenguaje Java. El CUP es un sistema para generar analizadores sintáctico LALR de especificaciones simples. Desempeña el mismo papel que el programa ampliamente usado YACC [1] y de hecho presenta la mayoría de las características de YACC. Sin embargo, el CUP se escribe en Java, usa especificaciones incluyendo código Java, y produce los analizadores gramaticales implementados en Java.

Aunque este manual cubra todos los aspectos del sistema CUP, es relativamente breve, y asume que usted tiene al menos los conocimientos básicos de análisis de LR. Los conocimientos básicos de YACC también son muy útiles para entender cómo funcionan las especificaciones en el CUP. Están disponibles una serie de manuales de construcción de compiladores (como [2,3]) que cubren este material, y hablan del sistema YACC (que es bastante similar a éste) con un ejemplo específico.

La utilización del CUP implica la creación de una especificación simple basada en la gramática, para lo cual es

necesario un analizador gramatical así como la construcción de un escáner capaz de reagrupar caracteres en tokens significativos (como palabras clave, números, y símbolos especiales).

Como ejemplo simple, considere un sistema para evaluar expresiones aritméticas simples con números enteros. Este sistema leería expresiones de la entrada estándar (con un punto y coma como terminador), los evaluaría, e imprimiría el resultado sobre la salida estándar. Una gramática para la entrada a tal sistema podría ser la siguiente:

```
expr_list ::= expr_list expr_part | expr_part
expr_part ::= expr ';'
expr      ::= expr '+' expr | expr '-' expr | expr '*' expr
            | expr '/' expr | expr '%' expr | '(' expr ')'
            | '-' expr | number
```

Para especificar un analizador gramatical basado en esta gramática, nuestro primer paso es identificar y nombrar el juego de símbolos terminales que aparecerán en la entrada, y el juego de símbolos no terminales. En este caso, los no terminales son:

expr_list, expr_part and expr .

Para los símbolos terminales podríamos escoger los siguientes nombres:

SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, NUMBER, LPAREN, y RPAREN

El usuario experimentado notará un problema con la susodicha gramática. Es ambigua. Una gramática ambigua es aquella que, dada una entrada determinada, puede reducir las partes de la entrada de dos modos diferentes, es decir, puede dar dos respuestas distintas. Tome dicha gramática, considerando por ejemplo la siguiente entrada:

3 + 4 * 6

La gramática puede o evaluar 3 + 4 y luego multiplicar siete por seis, o bien evaluar 4 * 6 y luego añadir tres. Las

versiones más antiguas de CUP forzaban al usuario a escribir gramáticas no ambiguas, pero ahora existe una forma que permite al usuario especificar preferencias y asociatividades para los símbolos terminales. Esto quiere decir que la gramática ambigua anterior puede ser usada, siempre que se hayan especificado el orden de preferencias y asociatividades. Se explicará de forma más detallada más adelante; pero basándonos en lo dicho podemos construir una pequeña especificación de CUP como sigue:

```
// Especificación de CUP para un evaluador de expresiones simples(sin acciones)

import java_cup.runtime.*;

/* Preliminares para establecer y usar el escáner.  */
init with {: scanner.init();            :};
scan with {: return scanner.next_token(); :};

/* Símbolos terminales (tokens devueltos por el escáner). */
terminal          SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal          UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;

/* No terminales */
non terminal          expr_list, expr_part;
non terminal Integer    expr, term, factor;

/* Preferencias*/
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* La gramática */
expr_list ::= expr_list expr_part
          | expr_part
          ;
expr_part ::= expr SEMI
            ;
expr      ::= expr PLUS expr
```

```
        | expr MINUS expr
        | expr TIMES expr
        | expr DIVIDE expr
        | expr MOD expr
        | MINUS expr %prec UMINUS
        | LPAREN expr RPAREN
        | NUMBER
        ;
```

Más adelante consideraremos cada una de las partes de la sintaxis de la especificación más detalladamente. Sin embargo, aquí rápidamente podemos ver que la especificación contiene cuatro partes principales. La primera parte proporciona declaraciones preliminares y mixtas para especificar cómo debe ser generado el analizador sintáctico, y suministrar las partes del código necesario en tiempo de ejecución. En este caso indicamos que las clases java_cup.runtime deberían ser importadas, luego debería suministrarse un pequeño trozo de código de inicialización y otro que invoque al escáner para recuperar el siguiente token de entrada. La segunda parte de la especificación declara terminales y no terminales, y asocia clases de objeto con cada uno. En este caso, los terminales son declarados o sin tipo, o de tipo entero. El tipo especificado del terminal o del no terminal es el tipo del valor que contienen esos símbolos terminales o no terminales. Si no se especifica ningún tipo, el símbolo, terminal o no terminal, no lleva ningún valor. En este ejemplo, los símbolos que no están asociados con ningún tipo son aquellos que no van a contener ningún valor. La tercera parte especifica la preferencia y asociatividad de los símbolos terminales. La última declaración de preferencia le da la preferencia más alta a sus terminales. La parte final de la especificación contiene la gramática.

Para producir un analizador sintáctico a partir de esta especificación usamos el generador CUP. Si esta especificación estuviera guardada en un fichero parser.cup, entonces (al menos en un sistema UNIX)podríamos invocar al CUP usando el siguiente comando:

```
java java_cup.Main < parser.cup
```

En este caso, el sistema producirá dos ficheros fuente java que contendrán las partes del analizador generado: sym.java y parser.java. Como cabe esperar, estos dos ficheros contienen las declaraciones de las clases sym y parser. La clase sym contiene una serie de declaraciones de constantes, una para cada símbolo terminal. Esto es lo que generalmente usa el escáner para referirse a los símbolos (como por ejemplo en el siguiente trozo de código "return new Symbol(sym.SEMI);"). La clase parser implementa el analizador directamente.

La especificación anterior, aunque construye un analizador sintáctico completo, no realiza ninguna acción semántica &emdash; esto sólo indicará el éxito o fracaso de un análisis. Para calcular e imprimir los valores de cada expresión, debemos integrar el código Java dentro del analizador para realizar acciones en los distintos puntos. En el CUP, las acciones se presentan en *cadenas de código* contenidas entre delimitadores de la forma {: y :} (podemos verlo en las cláusulas *init with* y *scan with* del ejemplo anterior). En general, el sistema registra todos los caracteres que estén dentro de los delimitadores, pero no trata de comprobar que contenga código Java válido.

A continuación se muestra una especificación CUP más completa para nuestro ejemplo(con acciones integradas en varios puntos de la gramática):

```
// Especificación de CUP para un evaluador de expresiones simples (con acciones atribuidas)

import java_cup.runtime.*;

/* Preliminares para establecer y usar el escáner.  */
init with {: scanner.init();            :};
scan with {: return scanner.next_token(); :};

/* Símbolos terminales (tokens devueltos por el escáner). */
terminal          SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal          UMINUS, LPAREN, RPAREN;
terminal Integer  NUMBER;
```

```
/* No terminales */
non terminal          expr_list, expr_part;
non terminal Integer    expr;

/* Preferencias */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* La gramática */
expr_list ::= expr_list expr_part
            |
           expr_part;

expr_part ::= expr:e
            {: System.out.println("= " + e); :}
           SEMI
            ;

expr     ::= expr:e1 PLUS expr:e2
            {: RESULT = new Integer(e1.intValue() +
e2.intValue()); :}
            |
           expr:e1 MINUS expr:e2
           {: RESULT = new Integer(e1.intValue() -
e2.intValue()); :}
            |
           expr:e1 TIMES expr:e2
            {: RESULT = new Integer(e1.intValue() *
e2.intValue()); :}
            |
           expr:e1 DIVIDE expr:e2
            {: RESULT = new Integer(e1.intValue() /
e2.intValue()); :}
            |
           expr:e1 MOD expr:e2
            {: RESULT = new Integer(e1.intValue() %
e2.intValue()); :}
            |
           NUMBER:n
```

```
        {: RESULT = n; :}
        |
      MINUS expr:e
        {: RESULT = new Integer(0 - e.intValue()); :}
        %prec UMINUS
        |
      LPAREN expr:e RPAREN
        {: RESULT = e; :}
        ;
```

Aquí podemos ver varios cambios. El más importante, el código que se ejecutará en varios puntos del analizador incluido dentro de cadenas de código delimitadas por {: y :}. Además, se han colocado etiquetas a varios símbolos en la parte derecha de las producciones. Por ejemplo en:

```
expr:e1 PLUS expr:e2
      {: RESULT = new Integer(e1.intValue() +
e2.intValue()); :}
```

el primer símbolo no terminal expr se ha etiquetado con e1 y el segundo con e2. El valor de la parte izquierda de la producción está siempre implícitamente etiquetada como RESULT.
Cada uno de los símbolos que aparece en una producción, en tiempo de ejecución, viene representado por un objeto d tipo *symbol* en la pila de análisis. Las etiquetas hacen referencia al atributo *value* de estos objetos. En la expresión *expr:e1 PLUS expr:e2*, e1 y e2 hacen referencia a objetos de tipo entero(*integer*). Estos objetos están en los campos *value* de los objetos de tipo *Symbol* y representan a dichos símbolos no terminales sobre la pila de análisis. *RESULT* es de tipo entero también, ya que el no terminal resultante *expr* fue declarado de tipo entero. Este objeto se convierte en el atributo *value* de un nuevo objeto *Symbol*.

Para cada etiqueta, se declaran dos variables más accesibles al usuario. Se pasan una etiqueta de valor izquierdo y otra de derecho en la cadena de código, de modo que el usuario pueda averiguar dónde está el lado izquierdo y derecho de cada terminal o no terminal en la entrada. El

nombre de estas variables es el nombre de etiqueta, más *left* o *right*. por ejemplo, dada la parte derecha de una producción *expr:e1 PLUS expr:e2* el usuario no sólo podría acceder a las variables *e1* y *e2*, sino también a *e1left*, *e1right*, *e2left* y *e2right*, estas variables son de tipo int(entero).

El último paso en la creación de un analizador sintáctico es crear un analizador léxico . Este es el responsable de leer los caracteres individuales, eliminando elementos como los espacios blancos y los comentarios, reconociendo qué símbolos terminales de la gramática representa cada grupo de caracteres, y luego devolviendo al analizador sintáctico objetos *Symbo*l que representen a estos símbolos. Los símbolos terminales serán recuperados con una llamada a la función de escáner. En el ejemplo, el analizador sintáctico llamará a *scanner.next_token()* . Esta función debería devolver objetos de tipo *java_cup.runtime.Symbol*. Este tipo es muy diferente del *java_cup.runtime.symbol* de versiones anteriores del CUP. Estos objetos *Symbol* contienen el atributo v*alue* de tipo *Object*, cuyo valor debería fijarlo el analizador léxico. Este atributo hace referencia al valor de ése símbolo, y el tipo de objeto de *value* debería ser el mismo con el que se declaró en las declaraciones *terminal* y *non terminal*(símbolos terminales y no terminales)*.* En el ejemplo que estamos tratando, si el analizador léxico quisiera pasar un token NUMBER, debería crear un objeto *Symbol* cuyo atributo *value* debería contener un objeto de tipo entero(int). Los objetos *Symbol* correspondientes a símbolos terminales y no terminales sin valor tienen un campo *value null*.

El código contenido en la cláusula *init with* de la especificación se ejecutará antes de que se solicite ningún token. Atendiendo al código que aparezca en la cláusula *scan with* se pedirán cada uno de los tokens. En realidad, la forma exacta en que el analizador léxico los captura es algo más complicada; sin embargo nótese que cada llamada a la función *scanner* debería devolver una nueva ejemplar de *java_cup.runtime.Symbol* (o una subclase). Estos objetos *Symbol* son anotados con la información de analizador sintáctico y apilados en una pila; la reutilización de los objetos aparecerán en las anotaciones de analizador

gramatical siendo revueltas. Desde Cup 0.10j, debería detectarse la reutilización de *Symbol*, en cuyo caso el analizador gramatical lanzará un Error diciendo que debe arreglar el escáner.

En la siguiente sección se dará una explicación más detallada y formal de todas las partes de una especificación CUP. La sección 3 describe opciones para compilar en CUP. La sección 4 comenta los detalles de cómo personalizar un analizador sintáctico de CUP, mientras la sección 5 habla del interfaz de analizador léxico añadido en el CUP 0.10j. La sección 6 considera la recuperación de error. Finalmente, la Sección 7 proporciona una conclusión.

2. Sintaxis de Especificación

Ahora que hemos visto un pequeño ejemplo, presentamos una descripción completa de todas las partes de una especificación de CUP. Una especificación tiene cuatro secciones con un total de ocho partes específicas (sin embargo, la mayor parte de estas son opcionales). Una especificación consiste en:

- Especificaciones de paquete (package) y de importaciones(import),
- Componentes de código de usuario,
- Símbolo (terminal y no terminal) listas,
- Declaraciones de preferencia,
- La gramática.

Cada una de estas partes debe aparecer en el orden presentado aquí. (Se da una gramática completa para el lenguaje de especificación en el Apéndice A.). Los detalles de cada parte de la especificación son descritos en las subdivisiones que siguen.

Especificaciones de paquete (package) y de importaciones (import)

Una especificación comienza con las declaraciones opcionales del paquete de programas y de importación. Estos tienen la misma sintaxis, y juegan el mismo papel, que las declaraciones de paquete e importación encontradas en un programa normal de Java. Una declaración de paquete de programas es de la forma:

package name;

donde el nombre name es un identificador de paquete de programas de Java, posiblemente en varias partes separadas por ".". En general, CUP emplea convenciones léxicas de Java. Así por ejemplo, se permiten ambos estilos de comentarios de Java son, y los identificadores se construyen comenzando con una letra, el signo dólar (el $), o subrayado (_), seguido(opcionalmente) de cero o más letras, números, signos de dólar, y subrayado.

Después de una declaración de paquete de programas opcional, puede haber cero o más declaraciones de importación. Como en un programa de Java estos tienen la forma:

import package_name.class_name;

o

import package_name.;*

La declaración de paquete de programas indica en qué paquete de programas estarán el sym y las clases de analizador gramatical que son generadas por el sistema. Cualquier declaración de importación que aparezca en la especificación también aparecerá en el archivo de fuente para que la clase del analizador gramatical permita el uso de varios nombres para dicho paquete en el código de acción suministrado por el usuario.

Componentes de Código de Usuario

Después de las declaraciones opcionales de paquete e importación existe una serie de declaraciones opcionales que permiten la inclusión del código de usuario como parte del analizador gramatical generado (ver Sección 4 para una descripción completa de cómo el analizador gramatical usa este código). Como una parte del archivo del analizador gramatical, se produce una clase separada no pública para contener todas las acciones de usuario integradas. La primera sección de declaración de código de acción permitirá incluir el código en esta clase. Las rutinas y variables empleadas por el código integrado en la gramática normalmente se colocarán en esta sección (un ejemplo típico podría ser rutinas de manipulación de tabla de símbolo). Esta declaración toma la forma:

action code {: ... :};

donde {: ...:} es una cadena de código cuyo contenido se colocará directamente dentro de la declaración de la clase *action class*.

Después de la declaración de *action code* hay una declaración opcional de *parser code*. Esta declaración permite colocar directamente métodos y variables dentro de la clase del analizador gramatical generada. Aunque esto sea menos común, puede ser provechoso cuando personalice el analizador gramatical &emdash; es posible por ejemplo, incluir métodos de exploración dentro del analizador gramatical y/o anular las rutinas para informar errores por defecto. Esta declaración es muy similar a la declaración de *action code* y toma la forma

parser code {: ... :};

De nuevo, el código de la cadena de código es colocado directamente en la definición de clase del analizador gramatical generada.

Lo siguiente en la especificación es la declaración opcional *init* que tiene la forma:

init with {: ... :};

Esta declaración proporciona el código que será ejecutado por el analizador gramatical antes de que este pida el primer token. Esto se suele usar para inicializar el escáner, así como varias tablas y otras estructuras de datos que podrían ser necesarias para acciones semánticas. En este caso, el código dado en la cadena de código forma el cuerpo de un método void dentro de la clase *parser*.

La sección de código de usuario final (opcional) de la especificación indica cómo debería el analizador gramatical pedir el siguiente token desde el escáner. Esto tiene la forma:

scan with {: ... :};

Como con la clásula *init*, el contenido de la cadena de código forma el cuerpo de un método en el analizador gramatical generado. Sin embargo, en este caso el método devuelve un objeto de tipo *java_cup.runtime.Symbol*. Por consiguiente el código encontrado en la cláusula *scan with* debería devolver tal valor. Ver sección 5 para obtener información sobre el comportamiento por defecto si se omite la sección *scan with*.

Desde CUP 0.10j las secciones de *action code, parser code, init code* y *scan with* pueden aparecer en cualquier orden. Sin embargo, deben preceder a las listas de símbolo.

Listas de símbolo

Después del código suministrado por el usuario viene la primera parte requerida de la especificación: las listas de símbolo. Estas declaraciones son responsables de llamar y suministrar un tipo para cada símbolo terminal y no terminal que aparece en la gramática. Como se indicaba arriba, cada símbolo terminal y no terminal es representado en tiempo de ejecución con un objeto de Símbolo. En el caso de los terminales, estos son devueltos por el escáner y colocados sobre la pila del analizador gramatical. El analizador léxico debería poner el valor del terminal en la variable *value* del

objeto. En el caso de no terminales estos sustituyen una serie de objetos Symbol en la pila del analizador gramatical siempre que la parte derecha de alguna producción sea reconocida. Para indicarle al analizador gramatical qué tipo de objetos deben usarse para cada símbolo, terminal o no terminal se utilizan las declaraciones de terminales. Estas toman la forma:

terminal *classname name1, name2,* ...;
non terminal *classname name1, name2,* ...;
terminal *name1, name2,* ...;

y

non terminal *name1, name2,* ...;

Donde *classname* puede ser un nombre constituido por varias partes separadas con "." . El classname especificado representa el tipo del valor del terminal o no terminal. Cuando se tiene acceso a estos valores a través de etiquetas, los usuarios usan el tipo declarado. El classname puede ser de cualquier tipo. Si no se da ningún classname, entonces el terminal o no terminal no lleva asociado ningún valor sino una etiqueta que hace referencia a él y contiene un valor nulo. Desde el CUP 0.10j, se puede especificar la declaración de no terminales "nonterminal" (sin espacio) así como el original "non terminal" .

Los nombres de terminales y no terminales no pueden ser palabras reservadas de CUP; estos incluyen "code", "action", "parser", "terminal", "non", "nonterminal", "init", "scan", "with", "start", "precedence", "left", "right", "nonassoc", "import", y "package".

Declaraciones de Preferencia y Asociatividad

La tercera sección, que es opcional, especifica las preferencias y la asociatividad de terminales. Esto es útil para análisis con gramáticas ambiguas, como se ha hecho en el ejemplo de anterior. Hay tres tipos de declaraciones de preferencia/asociatividad:

precedence left *terminal*[, *terminal*...];

```
precedence right    terminal[, terminal...];
precedence nonassoc terminal[, terminal...];
```

La lista separada por comas indica que esos terminales deberían tener la asociatividad especificada en ese nivel de preferencia y la preferencia de esa declaración. El orden de preferencia, de lo más a lo menos importante, de abajo hacia arriba. Así, podemos declarar que la multiplicación y la división tienen una preferencia mayor que la suma y la resta:

```
precedence left  ADD, SUBTRACT;
precedence left  TIMES, DIVIDE;
```

Establecer el orden de preferencia permite resolver problemas de ambigüedad a la hora de hacer cambios en la pila. Por ejemplo, considerando la entrada al analizador gramatical del ejemplo 3 + 4 * 8, el analizador gramatical no sabe si hay que reducir 3 + 4 o cambiar el '*' en la pila. Sin embargo, desde que se establece una preferencia mayor para el '*' que para la '+', se cambiará en la pila y la multiplicación será realizada antes que la suma.

CUP asigna a cada uno de sus terminales una preferencia según estas declaraciones. Cualquier terminal que no aparezca en esta declaración tiene la preferencia más baja. CUP también asigna a cada una de sus producciones una preferencia. Esa preferencia es igual a la preferencia del último terminal en la producción. Si la producción no tiene terminales, entonces tiene la preferencia más baja. Por ejemplo, expr:: = expr TIMES expr tendría la misma preferencia que TIMES. Cuando hay un conflicto de cambio/reducción, el analizador gramatical determina si el terminal a cambiar o la producción a reducir tiene una preferencia más alta. Si el terminal o la producción tiene la preferencia más alta se produce el cambio. Si tienen la misma preferencia, la asociatividad del terminal determina qué ocurre.

Se asigna una asociatividad a cada terminal usado en las declaraciones de preferencia/asociatividad. Las tres asociatividades son izquierda, derecha y nonassoc. Las

asociatividades también son usadas para resolver conflictos de cambio/reducción, pero sólo en el caso de igual preferencia. Si la asociatividad del terminal que puede ser cambiado es izquierda entonces se realiza una reducción. Es decir, si la entrada es una cadena de sumas, como 3 + 4 + 5 + 6 + 7, el analizador gramatical siempre los reducirá de izquierda a derecha, en este caso, comenzando con 3 + 4. Si la asociatividad del terminal es derecha, este es cambiado en la pila, produciéndose las reducciones de derecha a izquierda. Así, si SUMA fue declarada con asociatividad derecha, el 6 + 7 sería reducido primero en dicha cadena. Si un terminal es declarado como nonassoc, entonces dos ocurrencias consecutivas de terminales no asociativos de igual preferencia generan un error. Esto es útil para operaciones de comparación. Por ejemplo, si la cadena de entrada es 6 == 7 == 8 == 9, el analizador gramatical debería generar un error. Si '==' es declarado como nonassoc entonces se generará un error.

Todos los terminales no usados en las declaraciones de preferencia/asociatividad se tratan con la preferencia más baja. Si se produce un error de cambio/reducción que implique a dos terminales de este tipo, no se puede resolver, como ocurre con los conflictos anteriores; en ese caso se informará de ello.

La Gramática

La sección final de una declaración de CUP proporciona la gramática. Esta sección, opcionalmente, comienza con una declaración de la forma:

start with *non-terminal*;

Esto indica qué símbolo no terminal es *el principio o el objetivo* no terminal para el análisis. Si este no se declara explícitamente, entonces se usará el no terminal de la parte izquierda de la primera producción. Al finalizar un análisis con éxito, CUP devuelve un objeto de tipo *java_cup.runtime. Symbol*. El valor de la variable que instancia este símbolo contiene el resultado final de la reducción.

La gramática en si misma sigue a la declaración de *start* opcional. Cada producción en la gramática tiene una parte izquierda no terminal seguido del símbolo ":: = ", al que a su vez siguen de una serie de cero o más acciones, símbolos terminales o no terminales, seguidos por una asignación de preferencia contextual (opcional), y terminado con un punto y coma (;).

Cada símbolo de la derecha puede ser etiquetado por un nombre. Los nombres de etiqueta aparecen después del nombre de símbolo separado por dos puntos (:). Los nombres de etiqueta deben ser únicos dentro de la producción, y se pueden usar dentro del código de acción para referirse al valor del símbolo. Con la etiqueta, se crean dos variables más, que son la etiqueta *left* y la etiqueta *right*. Estos son valores enteros que contienen las posiciones izquierda y derecha del terminal o no terminal en el archivo de entrada. Estos valores deben ser inicializados correctamente en los terminales en el analizador léxico. Los valores izquierdo y derecho se propagan a los no terminales a los que las producciones reducen.

Si hay varias producciones para el mismo no terminal, deben declararse juntas. En este caso las producciones comienzan con el no terminal y ":: = ", seguido de múltiples partes derechas cada una separada por una barra (|). El conjunto completo de producciones se termina con un punto y coma.

Las acciones aparecen en la parte derecha como cadenas de código (p.ej., el código Java dentro de los delimitadores {: ...:}). El analizador gramatical las ejecuta en el momento en que reconoce la parte izquierda de la producción de la acción. (Note que el escáner habrá devuelto el token una vezpasado el punto de la acción, ya que el analizador gramatical necesita esta señal suplementaria lookahead para el reconocimiento.)

Las asignaciones de preferencia contextuales van a continuación de todos los símbolos y acciones de la parte derecha de la producción cuya preferencia se está asignando.

La asignación de preferencia contextual permite asignar a una producción una preferencia no basada en el último terminal que aparezca en ella. Se muestra un buen ejemplo en la especificación del analizador gramatical de la muestra:

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

expr ::=  MINUS expr:e
        {: RESULT = new Integer(0 - e.intValue()); :}
        %prec UMINUS
```

Aquí, la producción se declara con la preferencia de UMINUS. Así, el analizador gramatical puede dar al signo MINUS(-) dos preferencias diferentes, dependiendo si es un menos unario (indicador de signo) o una operación de substracción.

3.- Introducción y Ejemplo

Como hemos mencionado anteriormente, el Cup está escrito en java. Para invocarlo necesitamos usar el intérprete de java para invocar el método *java_cup.Main(),* pasando un array de cadenas que contiene las opciones. Suponiendo que tenemos una máquina unix, la forma más sencilla de hacerlo suele ser invocarlo directamente desde la línea de comandos con un comando como el siguiente:

Java java_cup.Main options < inputfile

Una vez ejecutado, Cup espe ra encontrar un fichero de especificación en la entrada estándar y genera dos ficheros fuente Java como salida.

Además del fichero de especificación, podemos pasarle distintas opciones para variar el comportamiento de CUP. Las opciones legales están documentadas en *Main.java* e incluyen:

- *-package name*(nombre de paquete).

Especifica que las clases *parser* y *sym* deben ser ubicadas en el paquete nombrado. Por defecto, la especificación se coloca en el código generado (de ahí que por defecto las clases sean incluidas en un paquete de nombre *unnamed*).

- *-parser name*(nombre del analizador sintáctico)

  El analizador sintáctico de salida y el código de acción se meten en un fichero (y clase) con el nombre dado, en lugar de “parser”, que es el que tiene por defecto.

- *-symbols name*(nombre de los símbolos)

  El código constante de símbolos que se mete en una clase con el nombre dado en lugar de “sym”(por defecto).

- *-interface*(Interfaz)

  El código constante de símbolos se trata como una interfaz en lugar de cómo una clase.

- *-nonterms*(sin terminales)

  Usa constantes para los no terminales en la clase sym (o el nombre que le hayamos dado). La clase parser no necesita estas constantes, por lo que, normalmente, no son de salida. Sin embargo, puede ser muy útil hacer referencia a estas constantes cuando se está haciendo una traza de un analizador sintáctico generado.

- -expect *number* (para indicar el número de conflictos esperados)

  Durante la construcción del analizador el sistema puede detectar la posibilidad de ocurrencia de una situación ambigua en tiempo de ejecución. Es el Llamado conflicto. En general, el analizador sintáctico puede ser incapaz de decidir si hay que cambiar (leer otro símbolo) o reducir (sustituir la parte derecha de una producción por su lado

izquierdo). A esto se le llama conflicto cambio/reducción. Asimismo, puede darse el caso de que el analizador sintáctico no sea capaz de decidir entre la reducción de dos producciones diferentes. Es lo que se conoce como conflicto reducción/reducción.
Normalmente, si se dan uno o varios de estos conflictos, la generación del analizador sintáctico es abortada. Sin embargo, en ciertos casos cuidadosamente considerados puede resultar ventajoso romper arbitrariamente tal conflicto. En este caso el Cup usa la convención YACC y resuelve los conflictos cambio/reducción con el cambio, y los de reducción/reducción usando la producción con " mayor prioridad " (el que se ha declarado primero en la especificación). Para permitir la rotura automática de conflictos debe darse la opción -expect indicándose exactamente cuántos conflictos se esperan. No se informa de los conflictos resueltos por preferencias y asociatividades.

- compact_red (reducción-compacta)

  La inclusión de esta opción permite una optimización de consolidación de tabla que implica reducciones. En particular, esto permite que la reducción más común de la entrada en cada fila de la tabla de análisis de acciones se establezca como reducción por defecto para esa fila. Esto típicamente ahorra un espacio considerable en las tablas, que pueden llegar a ser muy grandes. Esta optimización tiene el efecto de sustituir todas las entradas de error de una fila por la reducción de entrada por defecto. Aunque esto pueda parecer peligroso, o incluso claramente incorrecto, esto no afecta a la corrección del analizador gramatical. En particular, algunos cambios de este tipo son inherentes en analizadores gramaticales LALR (cuando se comparan con analizadores gramaticales canónicos LR), y los analizadores gramaticales finales no seguirán leyendo una vez detectado un token erróneo. El analizador gramatical, sin embargo, puede efectuar reducciones erróneas antes de descubrir el error, esto puede degradar la capacidad del analizador gramatical para la

recuperación de errores. (Ver referencia [2] pp. 244-247 o referencia [3] pp. 190-194 para una explicación completa de esta técnica de consolidación.)

Esta opción generalmente se usa para trabajar en torno a las limitaciones del bytecode de java para los tamaños del código de inicialización de tablas. Sin embargo, CUP 0.10h introdujo una codificación de cadenas para las tablas del analizador gramatical que no estaban sujetas a las limitaciones del método estándar de tamaño. Por consiguiente, el empleo de esta opción no se va a requerir para gramáticas grandes.

- -nowarn(sin avisos)

  Esta opción suprime los mensajes de aviso (como opposed los mensajes de error) generados por el sistema.

- -nosummary(sin resumen)

  Normalmente, el sistema imprime al final de la ejecución un resúmen listando una serie de elementos tales como el número de símbolos terminales y no terminales, los estados del análisis, etc. Esta opción suprime este resumen.

- -progress

  Con esta opción, el sistema imprime mensajes cortos indicando su avance a través de varias partes del proceso de generación del analizado sintáctico.

- -dump_grammar (volcado de gramática)

- -dump_states (volcado de estados)

- -dump_tables (volcado de tablas)

- -dump(volcado)

Con estas opciones, el sistema produce un volcado de la gramática, los estados del analizador construido(requeridos a veces para resolver conflictos) y las tablas de análisis(rara vez necesarias), respectivamente; todos ellos legibles para el hombre. La opción –dump puede usarse para producir todos estos volcados.

- -time

  Esta opción añade estadísticas detalladas de los tiempos al resumen normal de resultados. Generalmente esto resulta sólo de gran interés para los encargados de mantener el propio sistema.

- -debug

  Esta opción produce información del sistema obtenida mediante grandes trazas internas llevadas a cabo durante su ejecución. Generalmente esto resulta sólo de gran interés para los encargados de mantener el propio sistema.

- -nopositions

  Esta opción evita que el CUP genere código para propagar los valores de las partes izquierda y derecha de terminales a no terminales, y luego de no terminales a otros terminales. Si el analizador sintáctico no va a usar los valores de las partes derecha e izquierda de las producciones, entonces se ahorrará tiempo de ejecución en la computación al no tener que generar estas propagaciones de posición.
  Esta opción también evita que se generen las etiquetas para las variables, tanto de la parte izquierda como de la derecha, por lo que, cualquier referencia a ellas provocará un error.

- -noscanner

CUP 0.10j introduce la integración de escáner mejorado y una nueva interfaz, java_cup.runtime.Scanner. Por defecto, el analizador gramatical generado hace referencias a esta interfaz, lo cual significa que no se pueden usar estos analizadores con versiones anteriores a la 0.10j.Si el analizador sintáctico no usa las nuevas características de la integración de escáner, en ese caso se debe especificar la opción –noscanner para suprimir las referencias a java_cup.runtime.Scanner y permitir la compatibilidad con compiladores anteriores. No obstante, es muy poco probable que existan razones para hacer tal cosa.

- -version

  Si se invoca al CUP con esta opción, se imprimirá la versión en uso del CUP y se detendrá. Esto permite la comprobación automática de la versión del CUP para la creación de ficheros(Makefiles), la instalación de scrips y otras aplicaciones que se requieran.

4. Personalización del Analizador gramatical

Cada analizador gramatical generado consta de tres clases generadas. La clase *sym* (que puede ser renombrada usando la opción -symbols) simplemente contiene una serie de constantes enteras, una para cada terminal. Los no terminales también se incluyen si se da la opción -nonterms. El fichero fuente para la clase *parser* (que puede ser renombrada usando la opción -parser) en realidad contiene dos definiciones de clase, la clase pública *parser*, que implementa el analizador gramatical real, y otra clase no pública (llamada CUP$action) que contiene todas las acciones de usuario contenidas en la gramática, así como el código de la declaración del código de acción. Además del código suministrado por el usuario, esta clase contiene un método: CUP$do_action que consiste en una declaración para seleccionar y ejecutar varios fragmentos del código de acción suministrado por el usuario. En general, todos los nombres

que comienzan con el prefijo CUP$ se reservan para usos internos del código generado por CUP.

La clase parser contiene el analizador gramatical generado. Es una subclase de java_cup.runtime.lr_parser que implementa un sistema general para la manipulación de las tablas asociadas a un analizador tiupo LR.. La clase parser generada proporciona una serie de tablas para ser usadas por la estructura general. Se proporcionan tres tablas:

la tabla de producción
: proporciona el número de símbolo del no terminal de la parte izquierda, con la longitud de la parte derecha, para cada producción de la gramática,

la tabla de acción
: indica qué acción (cambio, reducción o error) debe tomarse en cada símbolo de previsión (lookahead) cuando se encuentre en cada estado, y

la tabla de reducción-goto
: indica a qué estado cambiar después de reducir (bajo cada no terminal de cada estado).

(Note que las tablas de acción y reducción-goto no se almacenan como simples arrays, sino que usan una estructura de "lista" comprimida para ahorrar una cantidad significativa de espacio. Ver comentarios del código fuente del sistema runtime para mas detalles.)

Más allá de las tablas de análisis, el código generado (o heredado) proporciona una serie de métodos que se pueden usar para personalizar el analizador gramatical generado. Algunos de estos métodos son suministrados por el código situado en la parte de la especificación y se pueden personalizar directamente de esa forma. Los demás los proporciona la clase de base lr_parser y se pueden reemplazar con nuevas versiones (mediante la declaración de código del analizador gramatical) para personalizar el sistema. Los métodos disponibles para la personalización incluyen:

public void user_init()

El analizador gramatical llama a este método antes de que el escáner efectúe la petición del primer token. El cuerpo de este método contiene el código de la cláusula init de la especificación.

public java_cup.runtime.Symbol scan()

Este método incluye el escáner y es llamado cada vez que el analizador gramatical necesita un nuevo terminal. El cuerpo de este método es suministrado por la cláusula scan with de la especificación, si existe; en otro caso, devuelve getScanner () .next_token ().

public java_cup.runtime.Scanner getScanner()

Devuelve el escáner por defecto. Ver sección 5.

public void setScanner(java_cup.runtime.Scanner s)

Establece el escáner por defecto. Ver sección 5.

public void report_error(String message, Object info)

Este método debería ser llamado siempre que se emita un mensaje de error. En la implementación por defecto de este método, el primer parámetro proporciona el texto de un mensaje que se imprimido en System.err y el segundo parámetro simplemente se ignora. Es muy típico reemplazar este método para proporcionar un mecanismo de informe de error más sofisticado.

public void report_fatal_error(String message, Object info)

Este método debería ser llamado siempre que se produzca un error irrecuperable. Responde llamando a report_error (), luego aborta el análisis llamando al método del analizador gramatical done_parsing (), y finalmente lanza una excepción. (En general done_parsing() debería ser llamado en cualquier punto en que el analizador gramatical necesitase terminar pronto).

public void syntax_error(Symbol cur_token)

El analizador gramatical llama a este método en cuanto se detecta un error de sintaxis (pero antes se intenta la recuperación de error). En la implementación por defecto este llama a: report_error (" error de Sintaxis ", null); .

public void unrecovered_syntax_error(Symbol cur_token)

El analizador gramatical llama a este método si es incapaz de reponerse de un error de sintaxis. En la implementación por defecto este llama a:

report_fatal_error (" no podía reparar y seguir analizando", null);.

protected int error_sync_size()

El analizador gramatical llama a este método para determinar cuántos tokens se deben analizar satisfactoriamente para considerar exitosa una recuperación de error. La implementación por defecto devuelve 3. No son recomendados valores por debajo de 2. Ver la sección sobre la recuperación de error para más detalles.

El análisis en sí mismo es realizado por el método público Symbol parser(). Este método comienza por obtener referencias a cada una de la tablas de análisis, luego inicializa un objeto de CUP$action (mediante la llamada a protected void init_actions ()). Después este llama a user_init (), luego trae el primer token con una llamada a scan(). Finalmente, comienza a analizar. El análisis sigue hasta que se llama a done_parsing()(esto se hace automáticamente, por ejemplo, cuando el analizador gramatical acepta). Entonces se devuelve un Símbolo con el valor de la variable conteniendo el RESULTADO de la producción inicial, o null, si no hay ningún valor.

Además del analizador gramatical normal, el sistema runtime también proporciona una versión de depuración del analizador gramatical. Funciona exactamente igual que el analizador gramatical normal, pero imprime mensajes de depuración (llamando a public void debug_message (String mess) cuya implementación por defecto imprime un mensaje en System.err).

Basado en estas rutinas, la invocación de un analizador gramatical CUP se hace generalmente con código como el siguiente:

```
/* crear un objeto analizador */
parser parser_obj = new parser();

/* abrir archivos de entrada, etc. aquí */
     Symbol parse_tree = null;
```

```
    try {
      if (do_debug_parse)
        parse_tree = parser_obj.debug_parse();
      else
        parse_tree = parser_obj.parse();
    } catch (Exception e) {
/* hacer la limpieza aquí - - posiblemente lanza de nuevo la e
*/
} finally {
/* hacer el cierre aquí */
 }
```

5.- La Interfaz Scanner

En CUP 0.10j, la integración del escáner ha sido mejorada de acuerdo a las sugerencias hechas por David MacMahon. Los cambios facilitan la incorporación de Jlex y otros escáners generados automáticamente en analizadores gramaticales CUP.

Para usar el nuevo código, su escáner debe implementar la interfaz java_cup.runtime.Scanner, definida de la siuiente manera:

```
package java_cup.runtime;

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

Además de los métodos descritos en la sección 4, la clase java_cup.runtime.lr_parser tiene dos nuevos métodos de acceso, setScanner() y getScanner(). La implementación de scan() por defecto es:

```
public Symbol scan() throws java.lang.Exception {
    return getScanner().next_token();
  }
```

El analizador sintáctico generado también contiene un constructor que toma al objeto Scanner y con él llama a

setScanner(). Como consecuencia, en la mayoría de los casos se pueden omitir las directivas init with y scan with. Lo único que hay que hacer es crear el analizador sintáctico con una referencia al escáner deseado:

```
/* creación de un objeto parser*/
parser parser_obj = new parser(new my_scanner());
```

o establecer el escáner después de crear el objeto parser:

```
/* creación de un objeto parser */
parser parser_obj = new parser();
/* establecer el escáner */
parser_obj.setScanner(new my_scanner());
```

Debe notarse que el analizador sintáctico usa símbolos de previsión (look-ahead), por lo que se recomienda no reinicializar el escáner en mitad de un análisis. Si se intenta usar la implementación de scan() por defecto, sin llamar a setScanner(),saltará un NullPointerException.

Como ejemplo de integración del scanner, todo lo que se requiere para el uso de un scanner JLex con CUP son las tres líneas siguientes en la entrada del generador del analizador léxico:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
```

Se anticipa que la directiva %cup del JLex abreviará las tres directivas anteriores en la siguiente versión de JLex. Así, la invocación del analizador sintáctico con el escáner del JLex queda sencillamente:

```
parser parser_obj = new parser( new Yylex(
some_InputStream_or_Reader));
```

Nótese que aún hay que mantener EOF correctamente; el código JLex para hacerlo puede ser:

```
%eofval{
    return sym.EOF;
%eofval}
```

Donde sym es el nombre de la clase symbol para el analizador sintáctico generado.

El ejemplo simple_calc de la distribución del CUP ilustra el uso de las características de la integración del escáner con un escáner no generado automáticamente.

6. Recuperación de Error

Un aspecto final importante de la construcción de analizadores gramaticales con CUP es el apoyo a la recuperación de error sintáctica. CUP usa los mismos mecanismos de recuperación de error que YACC. En particular, este mantiene un símbolo de error especial (denotado simplemente como error). Este símbolo juega el papel de un no terminal especial que, en vez de ser definido por producciones, combina una secuencia de entrada errónea.

El símbolo de error sólo entra en juego si se detecta un error de sintaxis. Si se detecta un error de sintaxis, el analizador gramatical trata de sustituir alguna parte del token del flujo de entrada con el error y luego seguir analizando. Por ejemplo, podríamos tener producciones como:

```
stmt ::= expr SEMI | while_stmt SEMI | if_stmt SEMI | ... |
          error SEMI
        ;
```

Esto indica que si ninguna de las producciones normales para stmt se puede enlazar con la entrada, debería declararse un error de sintaxis, y la recuperación debería hacerse saltando los tokens erróneos (equivalente a encontrarlos y sustituirlos por *error*) hasta un punto en el que el análisis pueda seguirse con un punto y coma (y el contexto adicional que legalmente sigue una declaración). Un error se considera recuperado si y sólo si un número suficiente de tokens superior al símbolo error han podido ser analizados satisfactoriamente. (El

número de tokens requerido viene determinado por el método error_sync_size() del analizador gramatical y por defecto es 3).

Expresamente, el analizador gramatical primero busca el estado más cercano a la cima de la pila de análisis que tiene una transición de salida por debajo de error. Generalmente se trabaja desde las producciones que representan construcciones más detalladas (como una clase específica de declaración) hasta las producciones que representan construcciones más generales (como la producción general para todas las declaraciones o una producción que representa una sección entera de declaraciones), hasta llegar a un lugar en el que se haya previsto una producción de recuperación de error. Una vez que el analizador gramatical está situado en una configuración que tiene una recuperación de error inmediata (por llenado completo de pila hasta el primero de esos estados), el analizador gramatical comienza a saltar tokens hasta encontrar un punto en el que pueda seguir analizando. Después de deshacerse de cada token, el analizador gramatical intenta continuar con el análisis de la entrada (sin ejecutar cualquier acción semántica integrada). Si el analizador gramatical puede analizar satisfactoriamente el número requerido de tokens, entonces la entrada es devuelta al punto de recuperación y el análisis se continúa normalmente (ejecutando todas las acciones). Si no se puede continuar con el análisis lo suficiente, entonces se descartará otra señal y el analizador gramatical otra vez tratará de seguir analizando. Si se llega al final de la entrada sin haber realizado una recuperación satisfactoria (o no se ha encontrado ningún estado de recuperación de error conveniente en la pila de análisis con el que empezar) entonces la recuperación de error falla.

7.- Conclusión

Este manual ha descrito brevemente el sistema de generación de un analizador sintáctico CUP LALR. CUP ha sido diseñado para desempeñar el mismo papel que el ya conocido sistema generador de analizadores sintácticos YACC, pero está escrito y opera en su totalidad con código java, en lugar

de C o C++. Se pueden encontrar detalles adicionales sobre el comportamiento del sistema en el código fuente del generador de analizadores gramaticales y del compilador. Consúltese la página de CUP, indicada a continuación, para acceder a la documentación API del sistema y su ejecución.

Este documento cubre la versión o.10j del sistema.
Visite la
página de CUP:
http://www.cs.princeton.edu/~appel/modern/java/CUP/ para obtener la información más reciente, las instrucciones para bajarse el sistema; aquellas noticias adicionales sobre CUP. Los informes de errores y comentarios del sistema que se estime deban saber los desarrolladores, envíense al encargado de mantener CUP, C. Scott Ananian, a cananian@alumni.princeton.edu.

CUP fue originalmente escrito por Scott Hudson, en agosto de 1995.

Fue extendido para soportar versiones anteriores por Frank Flannery, en julio de 1996.

Las mejoras an corrido a cargo de C. Scott Ananian, el encargado de mantener CUP, desde diciembre de 1997 hasta la actualidad.

Referencias

[1] S. C. Johnson, "YACC &emdash; Yet Another Compiler Compiler", CS Technical Report #32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing, Reading, MA, 1986.

[3] C. Fischer, and R. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings Publishing, Redwood City, CA, 1991.

Apéndice A. Gramática para Archivos de Especificación CUP (0.10j)

```
java_cup_spec      ::= package_spec import_list code_parts
                       symbol_list precedence_list start_spec
                       production_list
package_spec       ::= PACKAGE multipart_id SEMI | empty
import_list        ::= import_list import_spec | empty
import_spec        ::= IMPORT import_id SEMI
code_part          ::= action_code_part | parser_code_part |
                     init_code | scan_code
code_parts         ::= code_parts code_part | empty
action_code_part   ::= ACTION CODE CODE_STRING
opt_semi
parser_code_part   ::= PARSER CODE CODE_STRING
opt_semi
init_code          ::= INIT WITH CODE_STRING opt_semi
scan_code          ::= SCAN WITH CODE_STRING opt_semi
symbol_list        ::= symbol_list symbol | symbol
symbol             ::= TERMINAL type_id declares_term |
                     NON TERMINAL type_id declares_non_term |
                       NONTERMINAL type_id declares_non_term |
                       TERMINAL declares_term |
                       NON TERMINAL declares_non_term |
                       NONTERMIANL declared_non_term
term_name_list     ::= term_name_list COMMA new_term_id
| new_term_id
non_term_name_list ::= non_term_name_list COMMA
new_non_term_id |
                     new_non_term_id
declares_term      ::= term_name_list SEMI
declares_non_term  ::= non_term_name_list SEMI
precedence_list    ::= precedence_l | empty
precedence_l       ::= precedence_l preced + preced;
preced             ::= PRECEDENCE LEFT terminal_list SEMI
                     | PRECEDENCE RIGHT terminal_list SEMI
                     | PRECEDENCE NONASSOC terminal_list
SEMI
```

```
terminal_list      ::= terminal_list COMMA terminal_id |
terminal_id
start_spec         ::= START WITH nt_id SEMI | empty
production_list    ::= production_list production | production
production         ::= nt_id COLON_COLON_EQUALS rhs_list
SEMI
rhs_list           ::= rhs_list BAR rhs | rhs
rhs                ::= prod_part_list PERCENT_PREC term_id |
                       prod_part_list
prod_part_list     ::= prod_part_list prod_part | empty
prod_part          ::= symbol_id opt_label | CODE_STRING
opt_label          ::= COLON label_id | empty
multipart_id       ::= multipart_id DOT ID | ID
import_id          ::= multipart_id DOT STAR | multipart_id
type_id            ::= multipart_id
terminal_id        ::= term_id
term_id            ::= symbol_id
new_term_id        ::= ID
new_non_term_id    ::= ID
nt_id              ::= ID
symbol_id          ::= ID
label_id           ::= ID
opt_semi           ::= SEMI | empty
```

Apéndice B. Un ejemplo sencillo de Scanner

```
// Ejemplo Simple de la Clase Scanner

import java_cup.runtime.*;
import sym;

public class scanner {
  protected static int next_char;

  /* avanza un caracter en la lectura de la entrada */
  protected static void advance()
    throws java.io.IOException
```

```
    { next_char = System.in.read(); }

  /* initialize the scanner / se inicializa el scanner*/
  public static void init()
    throws java.io.IOException
    { advance(); }

  /* reconoce y devuelve el siguiente token completo */
  public static Symbol next_token()
    throws java.io.IOException
    {
      for (;;)
        switch (next_char)
          {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
              /* parse a decimal integer */
              int i_val = 0;
              do {
                i_val = i_val * 10 + (next_char - '0');
                advance();
              } while (next_char >= '0' && next_char <= '9');
            return new Symbol(sym.NUMBER, new
Integer(i_val));

            case ';': advance(); return new Symbol(sym.SEMI);
            case '+': advance(); return new Symbol(sym.PLUS);
            case '-': advance(); return new
Symbol(sym.MINUS);
            case '*': advance(); return new
Symbol(sym.TIMES);
            case '/': advance(); return new
Symbol(sym.DIVIDE);
            case '%': advance(); return new Symbol(sym.MOD);
            case '(': advance(); return new
Symbol(sym.LPAREN);
            case ')': advance(); return new
Symbol(sym.RPAREN);

            case -1: return new Symbol(sym.EOF);
```

```
          default:
            /* en este escáner se ignora todo lo demás */
            advance();
          break;
        }
    }
};
```

Apéndice C: Incompatibilidades entre CUP 0.9 Y CUP 0.10

La versión 0.10a de CUP es una importante revisión de CUP. Los cambios son significativos, lo cual implica que no sea compatible con viejas versiones. Los cambios consisten en:

- Una interfaz léxica diferente,
- Nuevas declaraciones de terminales/no terminales,
- Referencias a etiqueta diferentes,
- Un modo diferente de pasar los resultados (RESULT),
- Nuevos valores de posición y propagación,
- El analizador gramatical ahora devuelve un valor,
- Declaraciones de preferencia de terminales y
- Regla de asignación de preferencia contextual

Interfaz léxica

CUP ahora interactúa con el analizador léxico de una manera completamente diferente. En las liberaciones anteriores, se usaba una clase nueva para cada tipo distinto de terminal. Esta versión, sin embargo, usa sólo una clase: La clase Symbol. La clase Symbol tiene tres variables de objeto que son significativas para el analizador gramatical cuando pasa información desde el analizador léxico. La primera es la variable de caso value(valor). Esta variable contiene el valor de ese terminal. Es del tipo declarado como tipo terminal en el fichero de especificación del analizador gramatical. Las otras dos variables de caso son left(izquierda) y right(derecha). Estas deberían contener el valor entero del

lugar en el archivo de entrada, en el sentido de carácter, en el que se encontró ese terminal.

Para más información, refiérase al manual sobre escáners.

Declaraciones de Terminales/No terminales

Las declaraciones de terminales y no terminales ahora se pueden realizar de dos formas distintos para indicar los valores de los terminales o no terminales. Las declaraciones anteriores de la forma

terminal *classname terminal* [, *terminal ...*];

aún funcionan. El classname, sin embargo indica el tipo del valor del terminal o no terminal, y no indica el tipo de objeto colocado sobre la pila de análisis. Una declaración, como:

terminal *terminal* [, *terminal ...*];

indica que los terminales que aparecen en la lista no contienen ningún valor.

Para más información, diríjase al manual sobre declaraciones

Referencias a etiqueta

Las referencias a etiqueta no se refieren al objeto sobre la pila de análisis, como en el antiguo CUP, sino más bien al valor de la variable de caso value del objeto Symbol que representa a ese terminal o no terminal. Así, las referencias al valor de terminales y no terminales son directas, a diferencia del antiguo CUP, donde las etiquetas se referían a objetos que contenían el valor del terminal o no terminal.

Para más información, diríjase al manual sobre etiquetas.

El valor de RESULT

La variable RESULT se refiere directamente al valor del no terminal al cual una regla reduce, más que al objeto de la pila de análisis. Así, RESULT es del mismo tipo que el no terminal al cual reduce, como se declaró en la declaración de no terminales. De nuevo, la referencia es directa, y no a algo que contendrá los datos.
Para más información, diríjase al manual sobre RESULT.

Propagación de Posición

Para cada etiqueta, se declaran dos variables más, que son la etiqueta más a la izquierda *left* o la etiqueta más a la derecha *right*. Estas se corresponden con las ubicaciones izquierda y derecha del flujo de entrada a las que vienen ese terminal o no terminal. Estos valores se propagan desde los terminales de entrada, de modo que el no terminal del principio deberá tener un valor izquierdo de 0 y un valor derecho de la posición del último carácter leído.

Para más información, refiérase al manual sobre posiciones.

Valor devuelto

Una llamada a parse() o debug_parse() devuelve un objeto Symbol. Este Símbolo es el no terminal inicial, así que la variable de caso value contiene la asignación final de RESULT.

Preferencia

CUP ahora tiene terminales precedentes, una nueva sección de declaración, que aparece entre las declaraciones de terminales y no terminales y la gramática especifica la preferencia y la asociatividad de las reglas. Las declaraciones son de la forma:

```
    precedence {left| right | nonassoc} terminal[,   terminal ...];
...
```

A los terminales se les asigna una preferencia, donde los terminales de la misma línea tienen la misma preferencia, y las declaraciones de preferencia que aparecen más abajo en

la lista de declaraciones de preferencia tienen la mayor preferencia. Left(Izquierda), right(derecha) y nonassoc(sin asociatividad) especifican la asociatividad de estos terminales. La asociatividad izquierda contribuye a la reducción de conflictos, la derecha a un cambio(movimiento) sobre el conflicto, y nonassoc a un error sobre el conflicto. De ahí, que ahora se puedan usar gramáticas ambiguas.

Para más información, refiérase al manual sobre la preferencia.

Preferencia Contextual

Finalmente el nuevo CUP añade la preferencia contextual. Una producción se puede declarar como sigue:

lhs ::= *{right hand side list of terminals, non-terminals and actions}*
%prec *{terminal}*;

entonces esta producción tendría una preferencia igual al terminal especificado después del %prec. Así, los conflictos cambio/reducción se pueden resolver por el contexto. Note que la parte %prec *terminal* viene después de todas las cadenas de acciones. No aparece antes de la última cadena de acción.

Para más información, refiérase al manual sobre la preferencia contextual.
Estos cambios han sido implementados por:

Frank Flannery
Departamento de Informática
Universidad de Princeton

Apéndice D. Errores de programación

En esta versión del CUP es difícil para las frases de acción semántica(código java referente a producciones) acceder al método report_error y otros métodos y objetos definidos en la directiva *parser code*.

Esto es porque las tablas de análisis(y la maquinaria de análisis) están en un objeto(perteneciente a la clase parser o el nombre que se le haya puesto en la directiva -parser), y las acciones semánticas están en otro objeto(de la clase CUP$actions).

Sin embargo, existe una forma de hacerlo, aunque es poco elegante. El objeto action tiene un atributo privado y final llamado parser que apunta al objeto de análisis. De esta manera, se puede acceder a los métodos y la variables de caso del analizador en las acciones semánticas como sigue:

```
parser.report_error(message,info);
x = parser.mydata;
```

Quizás, esto no será necesario en una revisión futura, y tales métodos y variables, como report_error y mydata estarán disponibles directamente desde las acciones semánticas; lo lograremos uniendo los objetos “parser” y “actions”.

Para obtener una lista de cualquier otro conocimiento actual sobre errores de programación en CUP, ver http://www.cs.princeton.edu/~appel/modern/java/CUP/bugs.html.

Apéndice E : Anotaciones de cambio

0.9e
: Marzo de 1996, version original de Scott Hudson.

0.10a
: Agosto de 1996, varios cambios de importancia a la interfaz.

0.10b
: Noviembre de 1996, arregla algunos errores secundarios de programación.

0.10c

Julio de 1997, arregla un error de programación relacionado con declaraciones de preferencia.

0.10e

Septiembre de 1997, arregla un error de programación introducido en 0.10c relacionado con la preferencia nonassoc. Gracias a Tony Hosking por informar sobre el error de programación y aportar el arreglo.. También reconoce el carácter retorno de carro así como el espacio blanco y arregla otros pequeños errores de programación.

0.10f

Diciembre de 1997, fue una revisión de mantenimiento. La fuente de CUP fue mejorada para JDK 1.

0.10g

Marzo de 1998, añade nuevos rasgos y arregla viejos errores de programación. El comportamiento de las asignaciones de RESULT fue normalizado, y se arregló un problema que exitía con las producciones implícitas de principio. La gramática de CUP fue ampliada para permitir el tipo arrays para terminales y no terminales, y se añadió una bandera de línea de mando para permitir la generación de un interfaz de symbol, mejor que la clase. Los errores de programa asociados a múltiples invocaciones de un objeto del analizador gramatical simple y múltiples clases de CUP en un paquete de programas se han solventado. También se puso al día la documentación.

0.10h-0.10i

Febrero de 1999, son revisiones de mantenimiento.

0.10j

Julio de 1999, se amplió la gramática de entrada de CUP para permitir una mayor flexibilidad y se mejoró la integración del escáner mediante la interfaz java_cup.runtime.Scanner.