JavaCC

Análisis sintáctico con JavaCC

Álvar Arnaiz González

2020

Departamento de Ingeniería Informática
Universidad de Burgos

Tabla de contenidos

1. JavaCC

¿Cómo funciona?

JavaCC es un generador de analizadores recursivos descendentes escritos en Java: análisis LL(k).

El fuente JavaCC se compone de una sucesión de producciones:

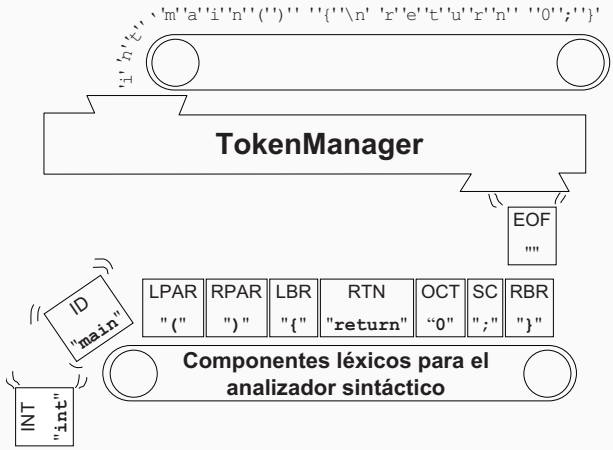
- Definiciones regulares que dan lugar al TokenManager.
- Producciones BNF que dan lugar al parser.

Genera un programa Java con un procedimiento para cada no terminal en cuyo cuerpo se pueden producir llamadas a otros procedimientos asociados con otros no terminales.

La misión del TokenManager es formar componentes léxicos (tókenes) con los caracteres de entrada para pasárselos al analizador recursivo descendente.

El parser de JavaCC utilizar el Token Manager para realizar el análisis léxico, es decir, crear los tókenes que usará el parser.

El analizador sintáctico realiza un análisis recursivo descendente en base a las producciones que el usuario haya definido en el fichero «`.jj`».



Código fuente de un parser en JavaCC

```
options={...}

// Definición del parser
PARSER_BEGIN(nombre_parser)
...
class nombre_parser ... {
    ...
}
...
PARSER_END(nombre_parser)

// Definición Token Manager
TOKEN:
{
  ...
}

// Definición de producciones BNF
void produccion (...) :
{
  ... // Declaración de variables, invocación de métodos...
}
{
  ... // Consecuente de la producción y acciones asociadas
}
```

2. Token Manager

Como se indicó, las producciones de expresión regular pueden ser de varios tipos. Para el análisis sintáctico conviene tener clara la diferencia:

- `TOKEN` Se genera una instancia de la clase `Token` con el lexema reconocido y se lo pasa al analizador sintáctico.
- `SKIP` Se reconoce el lexema que concuerda con la expresión regular pero no se envía al analizador sintáctico: se descarta. Sí realizará las acciones léxicas asociadas.
- `MORE` Se genera una instancia de la clase `Token` con el lexema reconocido, pero no lo devuelve al analizador sintáctico aún, será el prefijo del siguiente token. Es el equivalente a `yymore()` en Flex.
- `SPECIAL_TOKEN` Se genera un token especial que no será enviado al analizador sintáctico. Se puede acceder a este token especial mediante el campo `specialToken` de la clase `Token` del lexema reconocido inmediatamente después[1].

[1] No se utilizará durante las prácticas.

3. Definición del parser

Esta porción de código define la clase que implementará el parser. Queda definido por el código entre las directivas `PARSER_BEGIN` y `PARSER_END`.

Además de los métodos asociados a los no terminales de la gramática, y que serán generados por JavaCC, se pueden definir métodos y variables auxiliares.

Para iniciar el análisis, el parser deberá invocar a la función que representa el axioma de la gramática:

```
PARSER_BEGIN(Test)
public class Test {
    public static void main(String args[]) throws ParseException {
        // Instanciar un objeto de la clase Test
        Test parser = new Test(System.in);
        // Iniciar el análisis
        parser.axioma();
    }
}
PARSER_END(Test)
```

Las producciones se definen mediante notación BNF.

Cada producción se transformará en una función de Java con su valor de retorno y sus posibles parámetros de entrada.

En la producción puede aparecer código Java entre llaves: acciones semánticas asociadas.

```
tipoRetorno nombreProducción (listaParámetros):
{
    // Código Java. Por ejemplo: definición de variables.
}
{
    subProducción() <TOKEN>
      {/* acción semántica */}
      ...
}
```

Definición de producciones: ejemplo

Producciones para calcular sumas y restas:

```
void Axioma() :
{ int res; }
{
  res = Expression() ";"  { System.out.println("Resultado: " + res); }
}

int Expression() :
{ int res, val; }
{
  res = Factor()  ( "+" val = Factor()  { res = res + val; }
                  | "-" val = Factor()  { res = res - val; } )*
  { return res; }
}

int Factor() :
{ Token tk;
  int res; }
{
  ( tk = <NUM> { res = Integer.parseInt(tk.image); }
  | "(" res = Expression() ")"
  )
  { return res; }
}
```

4. Opciones del parser

Opciones

Como ya se ha visto, JavaCC permite definir opciones en la generación del parser, se realiza mediante:

```
options = { NOMBRE_OPCIÓN = valor; }
```

Estas opciones se escribirán al inicio del fichero «`.jj`», antes de la definición del parser.

En este punto, las más interesantes son:

- `DEBUG_TOKEN_MANAGER` Valor booleano que indica si se desea activar el modo «`debug`» en el parser generado.
- `DEBUG_PARSER` Valor booleano que indica si se desea activar el modo «`debug`» en el parser generado. Puede estar en combinación con el «`debug`» del Token Manager o no.
- `LOOKAHEAD` Valor entero que indica si se desea modificar el valor de k del analizador LL(k) generado.
- `IGNORE_CASE` Valor booleano que indica si el parser debe ignorar mayúsculas/minúsculas.

5. Lookahead

Como se ha indicado, JavaCC genera analizadores recursivos descendentes LL(k).

Aunque la gramática no sea LL(1) para algunas producciones, es posible hacer un análisis de la misma dando a JavaCC información adicional en los puntos de conflicto. Posibles soluciones:

1. Indicando que localmente la decisión se base en más de un token de entrada.
2. Utilizando lookahead sintáctico (se especifican los elementos sintácticos que tienen que estar presentes para utilizar una determinada producción) o semántico (se especifica mediante una expresión Java la condición que tiene que verificar para utilizar una determinada producción).
3. Reescribiendo la gramática para evitar el conflicto.

Suponer un conflicto como el siguiente:

```
void a():{} { <ID> b() | < ID> c() }
```

- Solución 1: Lookahead de dos tókenes.

```
void a():{}{ LOOKAHEAD(2) <ID> b() | < ID> c() }
```

- Solución 2: Lookahead sintáctico.

```
void a():{}{ LOOKAHEAD(<ID> b()) <ID> b() | <ID> c() }
```

- Solución 3: Reescritura (opción recomendada).

```
void a():{}{ <ID> ( b() | c() ) }
```


J. Community.
JavaCC tutorials.
`https://javacc.github.io/javacc/tutorials/`.


C. García-Osorio.
Apuntes de las prácticas de procesadores del lenguaje: JavaCC.