Análisis semántico

@Dec 1, 2020 8:45 AM-10:30 AM

Instalación

⚠ Si se usa CUP junto a JFLEX, debe añadirse la especificación `%cup` en la *sección II* del fichero JFLEX.

⚠ Por otra parte, si usas Linux por tu cuenta, debes añadir las bibliotecas que usa CUP en el *classpath*.

Instalar Jflex y Cup en Ubuntu - Shakaran

Para la instalación es necesario tener habilitados los repositorios Universe. Podemos buscar los paquetes necesarios en Synaptic llamados jflex y

https://shakaran.net/blog/2009/02/instalar-jfl

ex-y-cup-en-ubuntu/

CUP

```
import java_cup.runtime.*;

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

/* 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;

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

/* Definición de 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;
```

Secciones

📖 Al contrario que en JFLEX, cuyas secciones se diferenciaban mediante `%%`, las secciones de CUP son detectadas mediante una **lista de palabras reservadas** (`terminal`, `non terminal`, `precedence`...).

Importación de la librería

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

Control de los métodos de clase

```
/* Preliminares para montar el escáner */
init with {: scanner.init(); :};
scan with {: return scanner.next_token(); :};
```

- Gestiona los métodos de clase `parser.java`.
- Sirve para incluir código.
- Por lo general, no es necesario definirlo.

Definición de los tokens terminales

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

- Equivale al conjunto T de la gramática.
- Se asigna un identificador a cada **token terminal**.
- Pueden asignarse tipos, dotando a un token con **atributos** asociados a ese tipo.
- *Suelen escribirse en mayúsculas, por convención.*

Definición de los tokens no-terminales

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

- Equivale al conjunto N de la gramática.
- Se asigna un identificador a cada **token no-terminal**.
- Pueden asignarse tipos, dotando a un token con **atributos** asociados a ese tipo.
- *Suelen escribirse en minúsculas, por convención.*

Precedencias de operación

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

- La precedencia situada **más abajo** es la **más prioritaria**.
- Se define la asociatividad de los tokens:
  - Hacia la izquierda (`left`), que se resuelve con la **acción *reducción***.
  - Hacia la derecha (`right`), que se resuelve con la **acción *desplazamiento***.
  - Nula (`nonassoc`), que se usa cuando un token no genera conflictos; y no se resuelve.

 CUP calcula automáticamente **todos lo ítems** de las reglas de la gramática definida.

Ejemplos:

Sea un código en CUP cuyas precedencias y cuya gramática se definen a continuación, y los ítems I_a, I_b y I_c, **¿cómo se resuelven los conflictos que presenta cada ítem?**

$$I_a = \{E \to E\cdot + E, E \to E * E\cdot\}$$
$$I_b = \{E \to E\cdot * E, E \to E + E\cdot\}$$
$$I_c = \{E \to E\cdot + E, E \to E + E\cdot\}$$

```
precedence left PLUS, MINUS;   // 1
precedence left TIMES, DIVIDE; // 2
```

```
expr ::= expr PLUS expr    // 1
       | expr MINUS expr   // 1
       | expr TIMES expr   // 2
       | expr DIVIDE expr; // 2
```

Existe un conflicto en I_a entre las acciones: *desplazamiento* y *reducción*.

- $E \to E\cdot + E$ produce ***desplazamiento***: tiene **prioridad 1** por el terminal $+$ (**PLUS**).
- $E \to E * E\cdot$ produce ***reducción***: tiene **prioridad 2** por el terminal $*$ (**TIMES**).

$\text{Prioridad}(+) < \text{Prioridad}(*) \implies$ El conflicto se resuelve mediante ***reducción***.

Existe un conflicto en I_b entre las acciones: *desplazamiento* y *reducción*.

- $E \to E\cdot * E$ produce ***desplazamiento***: tiene **prioridad 2** por el terminal $*$ (**TIMES**).
- $E \to E + E\cdot$ produce ***reducción***: tiene **prioridad 1** por el terminal $+$ (**PLUS**).

$\text{Prioridad}(*) > \text{Prioridad}(+) \implies$ El conflicto se resuelve mediante ***desplazamiento***.

Existe un conflicto en I_c entre las acciones: *desplazamiento* y *reducción*.

- $E \to E\cdot + E$ produce ***desplazamiento***: tiene **prioridad 1** por el terminal $+$ (**PLUS**).

- $E \to E + E \cdot$ produce ***reducción***: tiene **prioridad 1** por el terminal $+$ (**PLUS**).

$\text{Prioridad}(+) = \text{Prioridad}(+) \implies$ El conflicto se resuelve mediante las reglas de asociatividad.

Como el terminal $+$ (PLUS) está definido con asociatividad a la izquierda (`left`), el conflicto se resuelve mediante ***reducción***.

Sea un código en CUP cuyas precedencias y cuya gramática se definen a continuación, se ha encontrado que falla para operaciones con números negativos.

```
precedence left PLUS, MINUS;   // 1
precedence left TIMES, DIVIDE; // 2
```

```
expr ::= expr PLUS expr    // 1
       | expr MINUS expr   // 1
       | expr TIMES expr   // 2
       | expr DIVIDE expr  // 2
       | NUM;
```

La solución sería definir la operación (expresión) de «menos unario», de forma que si se detecta un número `n` de la forma `-n`, entienda que es un número negativo.

¿Cómo se definiría la operación de «menos unario»?

Habría que definir un nuevo token de un símbolo no-terminal para el «menos unario».

Al mismo tiempo, en la gramática, añadir una nueva regla para interpretar el nuevo token.

```
precedence left PLUS, MINUS;   // 1
precedence left TIMES, DIVIDE; // 2
precedence left UMINUS;        // 3
```

```
expr ::= expr PLUS expr    // 1
       | expr MINUS expr   // 1
       | expr TIMES expr   // 2
       | expr DIVIDE expr  // 2
       | UMINUS expr       // 3
       | NUM;
```

Habiendo creado `UMINUS` en la tercera línea de la sección de precedencias, este token posee la mayor prioridad y por tanto, se reconocerán antes los números negativos que las propias operaciones de resta en el análisis semántico.

Sin embargo, aparece el siguiente problema, ¿cómo indicar en JFLEX cuándo un símbolo `-` describe una resta y cuándo un número negativo? Esto resulta en un código excesivamente tedioso.

Por otra parte, puede usarse el token `MINUS`, de forma que se reconocería el mismo símbolo `-` para las 2 operaciones, ignorando el problema anterior con JFLEX.

Lo único que habría que tratar es la prioridad, ya que debe ser mayor que la de resta; es decir, debe reconocerse un número negativo antes que una operación de resta. Si no se

tratara la prioridad esa regla tendría la misma prioridad que la de la operación resta y podría no ejecutarse correctamente.

El cambio final sería aplicar una nueva regla que use el token `MINUS`, con la prioridad del token `UMINUS`; para ello se usa la instrucción `%prec`, resultando el siguiente código:

```
expr ::= expr PLUS expr           // 1
       | expr MINUS expr          // 1
       | expr TIMES expr          // 2
       | expr DIVIDE expr         // 2
       | MINUS expr %prec UMINUS  // 3
       | NUM;
```

Definición de la gramática

```
/* Definición de la gramática */
expr_list ::= expr_list expr_part
            | expr_part;

expr_part ::= expr:e SEMI    {: RESULT = e; :};

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

$$L \to LP$$
$$L \to P$$
$$P \to E;$$
$$E \to E + E$$
$$E \to E - E$$
$$E \to E * E$$
$$E \to E/E$$
$$E \to E\%E$$
$$E \to -E$$
$$E \to (E)$$
$$E \to n$$

- La gramática se estructura mediante reglas del tipo $A \to B, \quad A \in N \quad B \in N \cup T.$

▼ Las reglas de la gramática admiten acciones y atributos.

- **Acciones →** bloque `{: :}` que ejecuta código en java.
- **Atributos →** identificadores de tokens de la regla que pueden usarse en las acciones.

📖 Cada regla tiene asignada una prioridad correspondiente a **la prioridad del primer símbolo no-terminal** para resolver los conflictos de la tabla SLR(1).

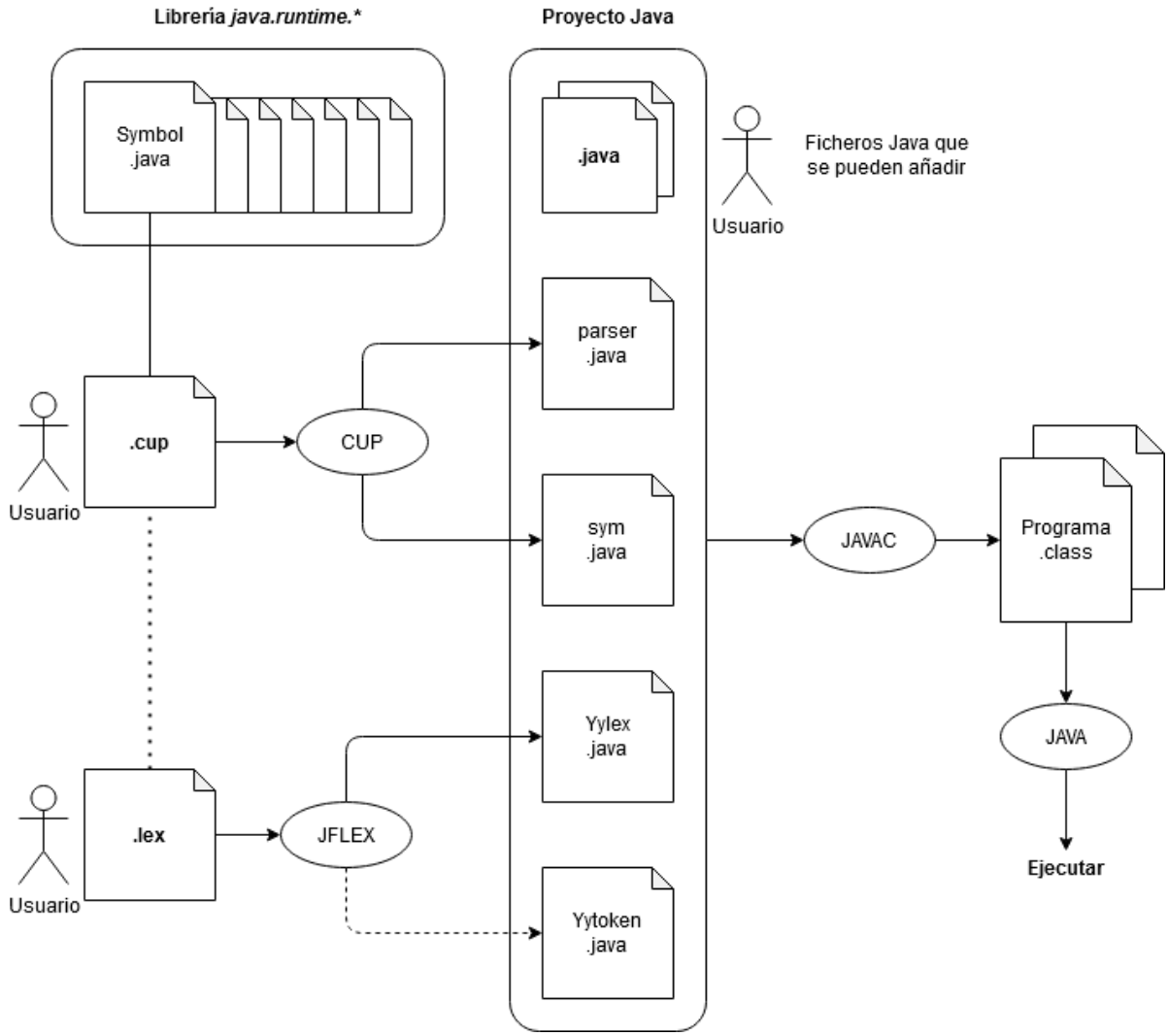
🧠 `RESULT` es un token predefinido de la clase `Symbol.java`.

🧠 `regla` `%prec` `terminal` → La regla adquiera la prioridad asociada al terminal.

⚠️ Una ***lista de expresiones*** se define de forma recursiva, como:
otra *lista de expresiones* seguido de una *expresión parcial*;
o bien, una sola *expresión parcial* (caso base).

⚠️ Una ***expresión parcial*** se define como: una *expresión* seguido de *un token de fin de línea*.

Ficheros



Estructura de un proyecto Java que usa CUP y JFLEX

Librería java_cup.runtime

```
class Symbol {
    public Symbol (int n);
    public Symbol (int n, Object o);

    // ...
}
```

Genera los tokens de los símbolos terminales.

Estos serán definidos en el fichero `sym.java` cuando sea creado por el compilador de CUP.

Se encarga de combinar el análisis léxico de JFLEX con el análisis semántico de CUP.

`Symbol.java` es a CUP lo que `Yytoken.java` es a JFLEX.

Generados por CUP

```
class sym {
    public static int MAS = 45;
    public static int MEN = 46;
    public static int POR = 47;
    public static int DIV = 48;

    // ... Generado automáticamente
}
```

Contiene una lista de tokens de los símbolos no-terminales con un numérico asignado.

Este valor solo es identificativo y se genera al azar, sin ningún tipo de patrón.

Salvo el valor 0, reservado para el token predefinido `EOF` que indica fin de fichero.

```
class parser {

    // ... Generado automáticamente

}
```

Incluye un analizador sintáctico basado en LALR(0), equivalente a un analizador SLR(1) con tablas no-ambiguas.

🧠 Las tablas SLR(1) no-ambiguas coinciden con las tablas LALR(0).

Escritos por el usuario

```
import java_cup.runtime.*;

// Terminales
terminal              MAS, MEN, POR, DIV, UMEN;
terminal              FIN;
terminal Integer      NUM;

// No-terminales
non terminal          lineas;
non terminal Integer  linea, exp;

// Precedencias
precedence left MAS, MEN;
precedence left POR, DIV;
precedence left UMEN;

// Reglas
lineas ::= lineas linea
         | linea:valor     {: System.out.println(valor); :};

linea ::= exp:e FIN        {: RESULT = e; :};

exp ::= exp:a MAS exp:b    {: RESULT = a + b; :}
      | exp:a POR exp:b    {: RESULT = a * b; :}
      | NUM:n              {: RESULT = n;     :};
```

```
import java_cup.runtime.*;

%%

%cup    // Indica al fichero que utilice Symbol.java en lugar de Yytoken.java
```

```
%%

"+"            {return new Symbol(sym.MAS);}
"-"            {return new Symbol(sym.MEN);}
"*"            {return new Symbol(sym.POR);}
"/"            {return new Symbol(sym.DIV);}

0|[1-9][0-9]*  {Integer val = new Integer(yytext())
                return new Symbol(sym.NUM, val);}

[\ \t\n\r]     { /* Ignorar */ }

[^]            {System.out.println("Carácter inesperado: " + yytext());}
```