Proyecto 2: Analizador sintáctico

Entrega 1 de Octubre de 2018

1 Descripción

El presente proyecto tiene como objetivo construir el analizador sintáctico que verificará que la secuencia de átomos, que reconoce el analizador léxico, tenga sentido bajo la gramática de *p* (gramática.org).

Las herramientas que utilizaremos para construir el analizador sintáctico son *byaccj* y *jflex*.

1.1 Analizador léxico

Actualmente el analizador léxico no devuelve átomos, sólo los imprime. Como parte de la realización de este proyecto deben regresarse los átomos en el formato especificado en el analizador sintáctico y *uno a la vez* (recordemos que al reconocer los átomos de indentación se imprimen más de uno cuando se empata la regla de los espacios).

1.2 Analizador sintáctico

El analizador sintáctico no incluye la generación del *árbol de sintáxis abstracta* pero hay que tener en cuenta su futura existencia a la hora de transformar la gramática de *p*, que está en formato *EBNF*, a la sintáxis de *byaccj* ya que internamente no es lo mismo reconocer una gramática con recursión a la derecha que una con recursión a la izquerda, como pudo observarse en la práctica anterior. Veamos un ejemplo:

```
/* arith_expr: term (('+'|'-') term)* */

/*    test: (term ('+'|'-'))* term   */
test: term
    | aux8 term
;
```

```
/*    aux8: (term ('+'|'-'))+        */
aux8: term MAS  { }
     | term MENOS { }
     | aux8 term MAS  { }
     | aux8 term MENOS  { }
;
```

La expansión natural de esa regla sería:

```
/* arith_expr: term (('+'|'-') term)* */
arith_expr: term
          | term aux8
;
aux8: MAS term
    | MENOS term
    | MAS term aux8
    | MENOS term aux8
;
```

Recordando que el reconocimiento utiliza una pila para almacenar los símbolos que en algún momento va a reducir, el comportamiento, obviando algunos pasos, sería el siguiente ante la cadena `2 + 5 - 4`:

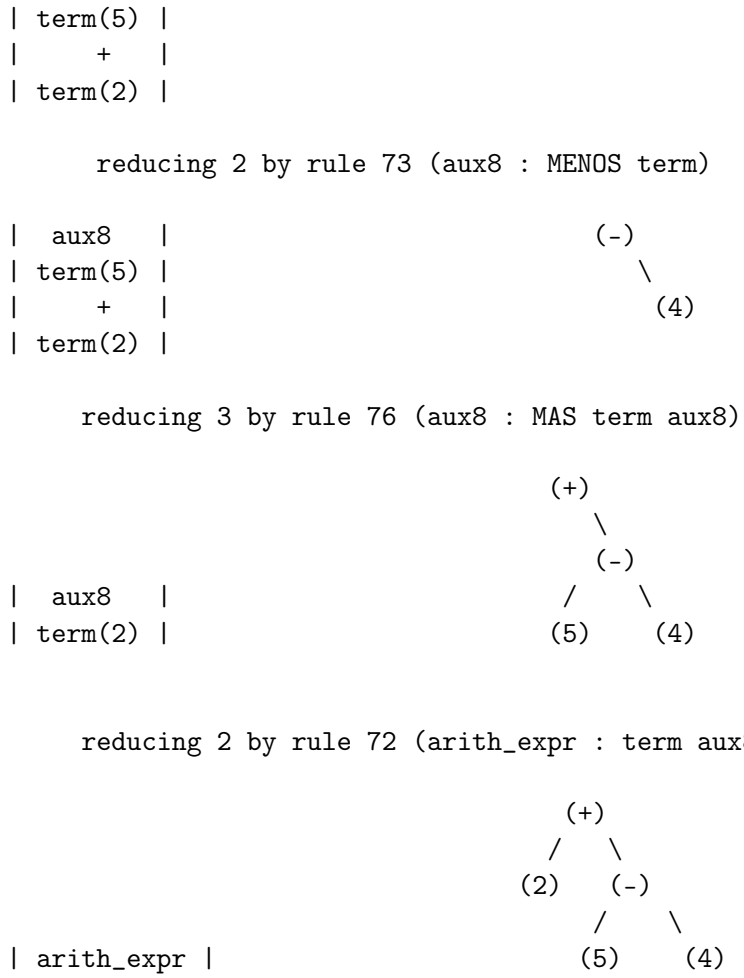
```
| term(2) |

|    +    |
| term(2) |


| term(5) |
|     +   |
| term(2) |

|    -    |
| term(5) |
|     +   |
| term(2) |

| term(4) |
|    -    |
```



De los anterior podemos observar dos inconvenientes:

1. No tan grave pero innecesario, guardar todos los símbolos en la pila antes de hacer alguna reducción.

2. Lo más grave es que se tiene asociatividad derecha.

Para este proyecto el reconocimiento sintáctico no se ve afectado por la dirección de la recursión, pero es importante tomar en cuenta que en las siguientes etapas sí afectará la dirección de la recursión. Por lo que se sugiere que desde esta etapa se expandan adecuadamente las reglas (con recursividad izquierda y entendiendo el uso de la pila y las reducciones).

2 Byaccj

Para facilitar la realización del proyecto se expondrán las siguientes características de *byaccj*:

- **Depuración**. El objeto del analizador sintáctico cuenta con una atributo booleano que cuando es verdadero imprime las acciones que va realizando el analizador sintáctico. Por omisión es falsa.

```
public static void main(String args[]) throws IOException {
    Parser yyparser = new Parser(new FileReader(args[0]));
    yyparser.yydebug = true; //Activamos la depuración
    yyparser.yyparse();
}
```

- **Descripción completa de la gramática y del analizador**. *byaccj* puede generar un archivo, `y.output`, en el que escribe de manera legible la gramática y expone los posibles conflictos que pueda tener la misma. Especifica el tipo de conflicto, el estado y la regla que lo causó.

```
$ byaccj -v -J  Parser.y
$ ls
Parser.y   ParserVal.java  Parser.java  y.output
```

3 Ejercicios

1. Modificar el analizador léxico para que regrese un único átomo por cada regla empatada.

2. Construir un archivo, que pueda ser recibido por *byaccj* para generar el analizador sintáctico del compilador para p. El analizador sintáctico no debe tener ningún conflicto (shift/reduce ó reduce/reduce).

3. Escribir un readme en el que explique como debe ser probado su código.