OLyC2

Gramática

Andrés Ricardo Ismael Guzmán - 201010425

Gramáticas 1

J#Jison 2

Expresiones Regulares. 2

Precedencia utilizada. 3

Cantidad de símbolos terminales. 3

Enumeración de los símbolos terminales. 3

Cantidad de símbolos no terminales. 4

Explicación de cada uno de los símbolos no terminales (cuál fue su uso dentro de la gramática) 4

Gramática funcional describiendo cada una de las acciones 7

Gramáticas

J#Jison

Expresiones Regulares.

IDENTIFIER	`[_a-zA-Z][a-zA-Z0-9_]*`

INTEGER LITERAL	**Macros Utilizadas**
	`D [0-9]`
	`NZ [1-9]`
	`Ds ("0"|{NZ}{D}*)`
	`{DS}`

STRING LITERAL	**Macros Utilizadas**
	`escapechar [\'\"\\bfnrtv]`
	`escape \\{escapechar}`
	`acceptedcharsdouble [^\"\\]+`
	`stringdouble {escape}|{acceptedcharsdouble}`
	`stringliteral (\"{stringdouble}*\")`

DOUBLE LITERAL	**Macros Utilizadas**
	`D [0-9]`
	`NZ [1-9]`
	`Ds ("0"|{NZ}{D}*)`
	`({Ds}"."{Ds}+)`

CHARACTER LITERAL	**Macros Utilizadas**
	`escapechar [\'\"\\bfnrtv]`
	`escape \\{escapechar}`
	`acceptedcharsdouble [^\"\\]+`
	`stringdouble {escape}|{acceptedcharsdouble}`
	`charliteral (\'{stringdouble}\')`

BOOLEAN LITERAL	“true” | “false”

NULL LITERAL	“null”
SEMICOLON, COMMA, LBRACE, RBRACE, LPAREN, RPAREN, COLON, LBRACK, RBRACK, ARROW,QUESTION, PEQ, DOLLAR, DOT	“;” “,” ”{“ “}“ “(“ “)” “:” “[“ “]” “?” “:=” “$” “.”
EQ	“=”
PLUSPLUS, MINUSMINUS	“++” “--”
CONST, VAR, GLOBAL	“const” “var” “global”
PLUS, MINUS, NOT, DIV, MOD, POW, MULT	“+” “-” “!” “/” “%” “^^” “*”
LT, LTEQ, GT, GTEQ, AND, OR, XOR, EQEQEQ	“<” “<=” “>” “>=” “&&” “||” “^” “===”
IF, ELSE, SWITCH, CASE, DEFAULT, FOR, BREAK, CONTINUE, WHILE, DO, THROW, TRY, CATCH, IMPORT, DEFINE, AS, STRC, RETURN	“if” “else” “switch” “case” “default” “for” “break” “continue” “while” “do” “throw” “try” “catch” “import” “define” “as” “strc” “return”
BOOLEAN, INTEGER, DOUBLE, CHAR, VOID	“boolean” “integer” “double” “char” “void”
Comentarios	"//".* "/*" <comment>"*/" <comment>. \s+

Precedencia utilizada.

```
%left PEQ, EQ

%left ELSE

%left LBRACK

%left DOT
```

Cantidad de símbolos terminales.

66

Enumeración de los símbolos terminales.

1. PEQ 2. LBRACE 3. RBRACE 4. LPAREN 5. RPAREN 6. LBRACK 7. RBRACK 8. COMMA 9. QUESTION 10. COLON 11. SEMICOLON 12. DOLLAR 13. FINAL 14. VAR 15. GLOBAL 16. VOID 17. IMPORT	18. IF 19. ELSE 20. WHILE 21. DO 22. FOR 23. BREAK 24. CONTINUE 25. SWITCH 26. CASE 27. DEFAULT 28. TRY 29. CATCH 30. THROW 31. BOOLEAN_LITERAL 32. DEFINE 33. AS 34. NEW	35. RETURN 36. BOOLEAN 37. INTEGER 38. DOUBLE 39. CHAR 40. LTEQ 41. LT 42. REQEQ 43. EQEQ 44. GTEQ 45. GT 46. NOTEQ 47. OROR 48. POT 49. XOR 50. ANDAND 51. NOT 52. EQ	53. PLUSPLUS 54. PLUS 55. MINUSMINUS 56. MINUS 57. MULT 58. DIV 59. MOD 60. DOT 61. NULL_LITERAL 62. IDENTIFIER 63. DOUBLE_LITERAL 64. INTEGER_LITERAL 65. STRING_LITERAL 66. CHARACTER_LITERAL

Cantidad de símbolos no terminales.

92

Explicación de cada uno de los símbolos no terminales (cuál fue su uso dentro de la gramática)

1. translation_unit	1. Este es el root, es una lista de posibles opciones
2. literal	2. producción para todas las hojas de las expresiones
3. primitive_type	3. Tipos primitivos
4. numeric_type	4. Producción para tipo numérico
5. integral_type	5. Tipo entero
6. `floating_point_type`	6. Tipo double
7. array_type	7. Nombres y primitivos en la forma de arreglo
8. name	8. Producción para simple name y qualified name
9. simple_name	9. IDENTIFICADOR
10. qualified_name	10. Producción para nombres del tipo A.B
11. import_declarations	11. Declaración para import
12. type_declarations	12. Lista para los tipos
13. type_declaration	13. Declaración de dos posibilidades
14. struct_declaration	14. Declaración de struct
15. struct_body	15. Cuerpo del struct
16. struct_variable_declaration	16. Definición de la forma en que vienen las variables
17. class_member_declaration	17. Declaración de los miembros que pueden venir: definición de variable y función
18. modifiers	18. const, var y global
19. field_declaration	19. Declaración de variable
20. variable_declarators	20. Declaración de variable sin punto y coma
21. variable_declarator	21. Id de la variable
22. variable_declarator_id	22. Id de la variable //hotfix para evitar conflictos
23. method_declaration	23. method_header y method_body
24. method_header	24. tipo y method_declarator
25. method_declarator	25. Id y lista de argumentos
26. formal_parameter_list	26. Lista de parámetros formales

27. formal_parameter	27. parámetro formal (tipo id)
28. method_body	28. body de la función
29. array_initializer	29. dos tipos de inicialización de arreglo
30. variable_initializers	30. Lista de tipos de inicialización de variable
31. variable_initializer	31. Inicialización de variable
32. block	32. lista de sentencias delimitada por { }
33. block_statements	33. lista de sentencias de un bloque
34. block_statement	34. switch con todas las sentencias posibles
35. variable_declaration_statement	35. declaración de variable con ;
36. variable_declaration	36. declaración de variable sin ;
37. special_declaration	37. tipo de declaración con :=
38. statement	38. Todas las posibles sentencias dentro de un método
39. statement_without_trailing_substatement	39. hotfix para evitar problemas de if colgante (no funcionó)
40. if_then_statement	40. if sin sentencia else
41. if_then_else_statement	41. if con sentencia else
42. while_statement	42. sentencia while
43. for_statement	43. sentencia for
44. for_init	44. primer hijo del for
45. for_update	45. tercer hijo del for
46. throw_statement	46. sentencia throw
47. try_statement	47. sentencia try catch
48. catches	48. lista de catches
49. catch_clause	49. sentencia catch como tal
50. statement_expression	50. expresión con delimitador ;
51. empty_statement	51. ; como tal
52. expression_statement	52. expresión
53. switch_statement	53. sentencia switch
54. switch_block_statement_groups	54. grupo de sentencias de switch
55. switch_labels	55. etiquetas case default de switch
56. do_statement	56. sentencia do while
57. expression_list	57. lista de expresiones con comas intercaladas
58. break_statement	58. sentencia break
59. continue_statement	59. sentencia continue
60. return_statement	60. sentencia return
61. primary	61. expresión que agrupa ciertas no terminales
62. primary_no_new_array	62. expresión con el grupo de no terminales faltantes
63. class_instance_creation_expression	63. setencia str IDENTIFIER()
64. argument_list	64. lista de argumentos para función
65. array_creation_expression	65. creación de arreglo
66. dim_exprs	66. lista dimensiones con expresiones

67. dim_expr 68. dims 69. field_access 70. method_invocation 71. array_access 72. postfix_expression 73. postincrement_expression 74. unary_expression 75. predecrement_expression 76. cast_expression 77. unary_expression_not_plus_minus 78. power_expression 79. preincrement_expression 80. multiplicative_expression 81. additive_expression 82. relational_expression 83. equality_expression 84. exclusive_or_expression 85. conditional_and_expression 86. conditional_or_expression 87. assignment_expression 88. assignment 89. left_hand_side 90. assignment_operator 91. expression 92. constant_expression	adentro 67. dimensión con expresión 68. lista de dimensión sin expresion [] 69. acceso a un objeto 70. llamada a función 71. acceso a un arreglo 72. expresión postfija 73. expresión postfija ++ 74. expresiones unarias 75. expresión prefija -- 76. casteos 77. expresión unaria sin menos o mas 78. potencia 79. expresión preija ++ 80. producción para producto, división y módulo 81. producción para suma, resta 82. producción con todos los relacionales >,>=,<=,< 83. Producción para igualdad ===, ==, != 84. Producción para xor 85. Producción para and 86. Producción para or 87. asignación 88. asignación 89. lado izquierdo de la asignación 90. operador = 91. expresión 92. expresión nuevamente

Gramática funcional describiendo cada una de las acciones

Básicamente las acciones tomadas fueron para la generación de un AST. Haciendo uso de las clases de javascript, en cada terminal se fue creando un nodo y según fuera el caso este nodo se sintetiza o se agrupa con otros para dar la estructura del árbol deseada.

Estructura del lenguaje

```
translation_unit :
    import_declarations SEMICOLON type_declarations EOF {
        $$ = new AST("PROGRAM", null, @1.first_line, @1.first_column, $1, ...$3);
        JSharpRoot = $$;
    }
    | type_declarations EOF {
        $$ = new AST("PROGRAM", null, @1.first_line, @1.first_column, ...$1);
        JSharpRoot = $$;
    }
    ;
```

En la primera producción se referencia al primer nodo y se obtiene la raíz del árbol.

Se toma el caso que sólo pueda venir una declaración ya que eso mencionaba el enunciado. Que pudiese venir import únicamente o la lista de declaraciones.

Esta producción contiene las 2 posibles declaraciones para una función. Como los parámetros pueden ser opcionales, se comprueba si los nodos traen algo o no. Se toma de referencia el terminal y se le cambia el tipo para construir el AST

Tipos de Literales

```
// 19.3) Lexical Structure.
literal :
    INTEGER_LITERAL {
        $$ = new AST("INTEGER_LITERAL", parseInt($1), @1.first_line, @1.first_column);
    }
    |   DOUBLE_LITERAL {
        $$ = new AST("DOUBLE_LITERAL", parseFloat($1), @1.first_line, @1.first_column);
    }
    |   BOOLEAN_LITERAL {
        $$ = new AST("BOOLEAN_LITERAL", $1 == 'true', @1.first_line, @1.first_column);
    }
    |   CHARACTER_LITERAL {
        $$ = new AST("CHARACTER_LITERAL", $1, @1.first_line, @1.first_column);
    }
    |   STRING_LITERAL {
        $$ = new AST("STRING_LITERAL", $1.substring(1,$1.length-1), @1.first_line, @1.first_column);
    }
    |   NULL_LITERAL {
        $$ = new AST("NULL_LITERAL", null, @1.first_line, @1.first_column);
    }
    ;
```

Se crean nodos de los tipos de datos del lenguaje, estos se crean a partir de las reglas y expresiones regulares definidas anteriormente. Son un

Sentencias

```
/* statements */
statement :
    statement_without_trailing_substatement {
    $$ = $1;
    }
    |   if_then_statement {
    $$ = $1;
    }
    |   if_then_else_statement {
    $$ = $1;
    }
    |   while_statement {
    $$ = $1;
    }
    |   for_statement {
    $$ = $1;
    }
    ;
```

statement sirve para agrupar para la recursividad de la lista como tal. Así que lo único que se hace es reducir el nodo y agregarlo a la lista o al nodo necesario

Bloques de Instrucciones

```
538
539  // 19.11) Blocks and Statements
540  block : LBRACE block_statements RBRACE {
541          $$ = $2;
542          $$.changeType("BLOCK");
543      }
544      | LBRACE RBRACE {
545          $$ = new AST("BLOCK",null,@1.first_line, @1.first_column);
546      }
547      ;
548
549  block_statements :
550      block_statement {
551          $$ = new AST("STMT_LIST",null,@1.first_line, @1.first_column,$1) ;
552      }
553      |   block_statements block_statement {
554          $$ = $1;
555          $$.addChild($2);
556      }
557      ;
558
559  block_statement :
560      variable_declaration_statement {
561          $$ = $1;
562      }
563      |   statement {
564          $$ = $1;
565      }
566      ;
567
```

block_statements es simplemente una lista de sentencias, al igual que las otras sentencias, en el inicio de la recursividad se crea el contenedor que agrupará el resto de sentencias, solo se van agregando los bloques a la lista de hijos.

Instrucciones Especiales

```
statement_without_trailing_substatement :
    block {
    $$ = $1;
    }
    |   empty_statement {
    $$ = $1;
    }
    |   expression_statement {
    $$ = $1;
    }
    |   switch_statement {
    $$ = $1;
    }
    |   do_statement {
    $$ = $1;
    }
    |   break_statement {
    $$ = $1;
    }
    |   continue_statement {
    $$ = $1;
    }
    |   return_statement {
    $$ = $1;
    }
    |   throw_statement {
    $$ = $1;
    }
    |   try_statement {
    $$ = $1;
    }
    ;
```

Este es el workaround para el if-else colgante. Al igual que en los anteriores sólo se sube el nodo.

```
statement_expression :
    expression {
    $$ = $1;
    }
    ;


empty_statement :
    SEMICOLON {
    $$ = new AST("NO_OP", null, @1.first_line, @1.first_column, $1);
    }
    ;

expression_statement :
    expression SEMICOLON {
        $$ = new AST("EXPRESSION_STMT", null, @1.first_line, @1.first_column, $1);
    }
    ;
```

De las producciones vistas en la foto anterior se toman sólo los nodos necesarios. En el caso del If tomamos el nodo del IF y en la primera casilla agregamos la condición de verdadero o falso seguido de la sentencia. Este caso es para un if sin else colgante

Sentencia IF

```
666
667  if_then_statement :
668      IF LPAREN expression RPAREN statement {
669      $$ = new AST('IF', null, @1.first_line, @1.first_column, $3, $5);
670      }
671      ;
672
673  if_then_else_statement :
674          IF LPAREN expression RPAREN statement
675              ELSE statement {
676      $$ = new AST('IF', null, @1.first_line, @1.first_column, $3, $5, $7);
677      }
678      ;
679
```

Aca se crea el nodo para la sentencia IF se crea un nodo principal como padre. Se añaden los hijos: hijo 1 la condición, hijo 2 la lista de sentencias, hijo 3 la lista de sentencias else. En caso de que hijo 3 sea nulo, no se traduce nada al momento de interpretar.

Sentencia SWITCH

```
811  switch_statement :
812      SWITCH LPAREN expression RPAREN switch_block {
813          $$ = new AST('SWITCH', null, @1.first_line, @1.first_column, $3, $5);
814      }
815      ;
816  switch_block :
817      LBRACE switch_block_statement_groups switch_labels RBRACE {
818          $$ = $2;
819          $2.addChild($3);
820      }
821      |   LBRACE switch_block_statement_groups RBRACE {
822          $$ = $2;
823      }
824      |   LBRACE switch_labels RBRACE {
825          $$ = new AST("SWITCH_BODY", null, @1.first_line, @1.first_column, $2);
826      }
827      |   LBRACE RBRACE {
828          $$ = new AST("SWITCH_BODY", null, @1.first_line, @1.first_column);
829      }
830      ;
831
832  switch_block_statement_groups :
833      switch_block_statement_group {
834          $$ = new AST("SWITCH_BODY", null, @1.first_line, @1.first_column,$1);
835      }
836      |   switch_block_statement_groups switch_block_statement_group {
837          $$ = $1;
838          $$.addChild($2);
839      }
840      ;
```

```
lock_statement_group :
ch_labels block_statements {
$$ = $1;
$1.addChild($2);



abels :
ch_label {
$$ = new AST('CASE_LABEL_LIST', null, @1.first_line, @1.first_column, ...$1);

switch_labels switch_label {
$$ = $1;
$$.addChild(...$2);



abel :
 constant_expression COLON {
$$ = [$2];

DEFAULT COLON {
$$ = [new AST("DEFAULT", null, @1.first_line, @1.first_column)]
```

Se crea un nodo para la sentencia SWITCH. Este contiene dos hijos: el primero la expresión que se evaluara y el segundo que que es un switch_block.

La producción de switch_block agrupa los diferentes tipos de cuerpos que el switch puede contener desde una lista de expresiones hasta un cuerpo vacío.

La producción de switch_block_statement_groups es una lista de expresiones junto a sus sentencias.

La producción de switch_block_statement_group define la estructura de una lista de expresiones junto a sus sentencias.

La producción de switch_labels define una lista de casos con su expresión (case 1: case 2

case 22-3: case n:).

La producción de switch_label agrupa las opciones para el label de cada bloque, podría ser un case (case 8:) o el defecto (default:).

Sentencia WHILE

```
679
680 while_statement :
681     WHILE LPAREN expression RPAREN statement {
682     $$ = new AST('WHILE', null, @1.first_line, @1.first_column, $3, $5);
683     }
684     ;
```

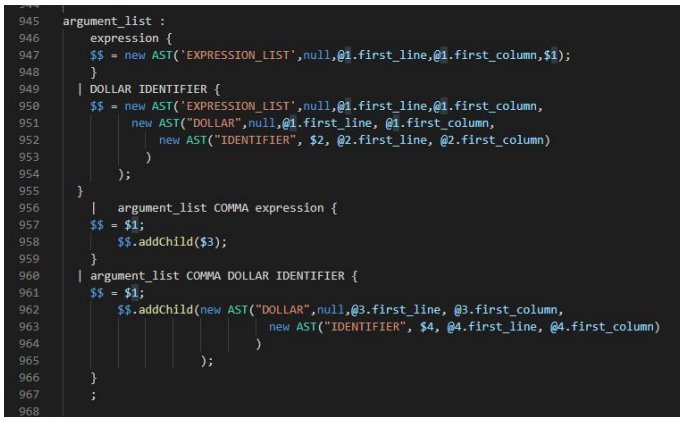
La sentencia while cuenta de una estructura muy simple,, la condición y luego la lista de sentencias a ejecutar si la condición fue cumplida.

Sentencias de Escape

```
888 break_statement :
889     BREAK SEMICOLON {
890     $$ = new AST("BREAK", null, @1.first_line, @1.first_column);
891     }
892     ;
893
894 continue_statement :
895     CONTINUE SEMICOLON {
896     $$ = new AST("CONTINUE", null, @1.first_line, @1.first_column);
897     }
898     ;
899
900 return_statement :
901     RETURN SEMICOLON {
902     $$ = new AST("RETURN", null, @1.first_line, @1.first_column);
903     }
904     | RETURN expression SEMICOLON {
905     $$ = new AST("RETURN", null, @1.first_line, @1.first_column,$2);
906     }
907     ;
908
909
```

Sentencias de escape, sólo se sube el nodo que se desea; en el caso de return si existe una expresión asociada se añade y se sube como tal.

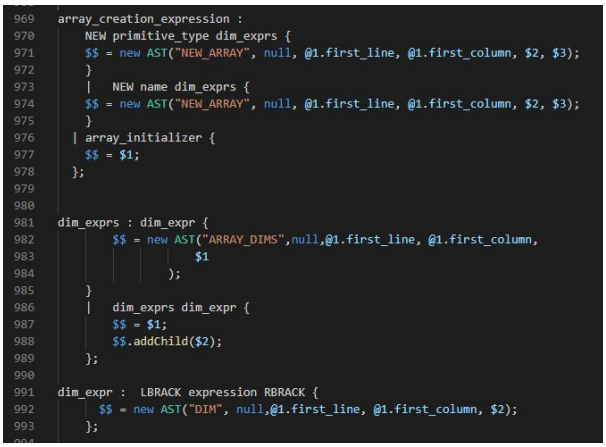
Lista de parámetros



```
argument_list :
    expression {
    $$ = new AST('EXPRESSION_LIST',null,@1.first_line,@1.first_column,$1);
    }
  | DOLLAR IDENTIFIER {
    $$ = new AST('EXPRESSION_LIST',null,@1.first_line,@1.first_column,
          new AST("DOLLAR",null,@1.first_line, @1.first_column,
              new AST("IDENTIFIER", $2, @2.first_line, @2.first_column)
            )
          );
  }
    |   argument_list COMMA expression {
    $$ = $1;
      $$.addChild($3);
    }
  | argument_list COMMA DOLLAR IDENTIFIER {
    $$ = $1;
      $$.addChild(new AST("DOLLAR",null,@3.first_line, @3.first_column,
                                  new AST("IDENTIFIER", $4, @4.first_line, @4.first_column)
                                )
                  );
    }
    ;
```

Inicio de expresiones, argument_list es para la lista de parámetros que puede tener una función o epsilon en caso que no tenga ninguno.

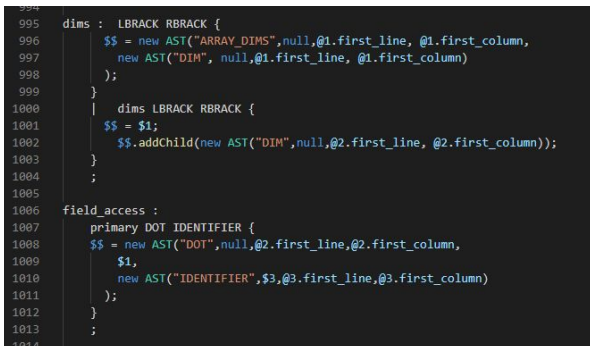
Arreglos



```
array_creation_expression :
    NEW primitive_type dim_exprs {
    $$ = new AST("NEW_ARRAY", null, @1.first_line, @1.first_column, $2, $3);
    }
    |   NEW name dim_exprs {
    $$ = new AST("NEW_ARRAY", null, @1.first_line, @1.first_column, $2, $3);
    }
  | array_initializer {
    $$ = $1;
  };


dim_exprs : dim_expr {
        $$ = new AST("ARRAY_DIMS",null,@1.first_line, @1.first_column,
                    $1
                );
    }
    |   dim_exprs dim_expr {
        $$ = $1;
        $$.addChild($2);
    };

dim_expr :  LBRACK expression RBRACK {
      $$ = new AST("DIM", null,@1.first_line, @1.first_column, $2);
    };
```



```
dims :  LBRACK RBRACK {
      $$ = new AST("ARRAY_DIMS",null,@1.first_line, @1.first_column,
        new AST("DIM", null,@1.first_line, @1.first_column)
      );
    }
    |   dims LBRACK RBRACK {
      $$ = $1;
        $$.addChild(new AST("DIM",null,@2.first_line, @2.first_column));
    }
    ;

field_access :
    primary DOT IDENTIFIER {
    $$ = new AST("DOT",null,@2.first_line,@2.first_column,
        $1,
        new AST("IDENTIFIER",$3,@3.first_line,@3.first_column)
      );
    }
    ;
```

La producción de array_creation_expression agrupa las tres formas para la creación de un array, ya sea un array de valor primitivo (String, Int, Bool...) o de un objeto (perro, gato...). Se cuenta con una producción para la creación de arrays inicializados. Recibe como hijos el tipo de dato y la lista de dimensiones.

La producción dim_exprs sirve para hacer una lista de dimensiones ([2][3]..), la lista va añadiendo las expresiones como hijos.

La producción dim_expr define la estructura de la definición de una dimensión.

La producción dims es una lista de dimensiones vacías el cual sirve para la definición de el numero de dimensiones de los los arrays (String[][][]).

Invocación de Métodos

```
method_invocation :
    name LPAREN argument_list RPAREN  {
            let type = native_functions.find(item => item == $1.getValue());
            $$ = new AST((type === undefined?"FUNCTION_CALL":"NATIVE_FUNCTION_CALL"),null,@1.first_line,@1
    }
    | name LPAREN RPAREN {
        let type = native_functions.find(it=>it==$1.getValue());
        $$ = new AST((type === undefined?"FUNCTION_CALL":"NATIVE_FUNCTION_CALL"),null,@1.first_line,@1.fir

    }
    |   primary DOT IDENTIFIER LPAREN RPAREN {
            $$ = new AST("FUNCTION_CALL",null,@3.first_line,@3.first_column,
                new AST("DOT", null, @2.first_line, @2.first_column, $1, new AST("IDENTIFIER",$3,@3.first_
    }
    |   primary DOT IDENTIFIER LPAREN argument_list RPAREN {
            $$ = new AST("FUNCTION_CALL",null,@3.first_line,@3.first_column,
                new AST("DOT", null, @2.first_line, @2.first_column,
                    $1,
                    new AST("IDENTIFIER",$3,@3.first_line,@3.first_column)
                ),
                $5
            );
    }
    ;
```

Esta producción encapsula todas las formas de invocación de los métodos. Lo que varia entre estas es el número de parámetros con las que cuenta el método. Como hijos tiene el nombre del método y la lista de parámetros.

Expresiones

```
unary_expression :
      preincrement_expression {
        $$ = $1;
    }
    | predecrement_expression {
        $$ = $1;
    }
    | PLUS unary_expression {
        $$ = new AST("PLUS", null,  @1.first_line, @1.first_column, $2);
    }
    | MINUS unary_expression {
        $$ = new AST("MINUS", null,  @1.first_line, @1.first_column, $2);
    }
    | unary_expression_not_plus_minus {
        $$ = $1;
    };
```

```
postfix_expression :
      primary {
        $$ = $1;
    }
    | name {
        $$ = $1;
    }
    | postincrement_expression {
        $$ = $1;
    }
    | postdecrement_expression {
        $$ = $1;
    };
```

A partir de los literales (producciones anteriores) acá se permite la negación tanto aritmética como lógica. La división en producciones es para la precedencia.

```
power_expression :
    unary_expression {
    $$ = $1;
  }
  | power_expression POT unary_expression {
    /* FIX THE RECURSION */
    $$ = new AST("POW", null, @2.first_line,@2.first_column);

    //we check for lhs (if lhs type == ^) we must rotate
    if($1.getType() == tree_types.types.POW && !$1.grouped){
          let tmp = $1.deleteAt(1)[0];
          $$.addChild(tmp);
          $$.addChild($3);
          $1.addChild($$);
          $$ = $1;
    }else{
            $$.addChild($1,$3);
        }
  };
```

Esta es la única lista de operaciones binarias de las que tenemos que hacer hincapié, ya que la asociatividad debe ser por la derecha. por lo que deben de intercambiarse los nodos como se muestra en la producción.

A partir de acá las operaciones binarias son las mismas y sólo se dividen en distintas producciones para que se comporte como se desea. En todos los casos son listas y el escape es la salida a otra lista. Si es una operación binaria se devuelve el operador con sus 2 hijos.

```
multiplicative_expression :
      power_expression {
        $$ = $1;
      }
    | multiplicative_expression MULT power_expression {
        $$ = new AST("MULT", null, @2.first_line,@2.first_column, $1,$3);
      }
    | multiplicative_expression DIV power_expression {
        $$ = new AST("DIV", null, @2.first_line,@2.first_column, $1,$3);
      }
    | multiplicative_expression MOD power_expression {
        $$ = new AST("MOD", null, @2.first_line,@2.first_column, $1,$3);
      };
```

```
additive_expression :
      multiplicative_expression {
        $$ = $1;
      }
    | additive_expression PLUS multiplicative_expression {
        $$ = new AST("PLUS", null, @2.first_line,@2.first_column, $1,$3);
      }
    | additive_expression MINUS multiplicative_expression {
        $$ = new AST("MINUS", null, @2.first_line,@2.first_column, $1,$3);
      };
```

```
relational_expression :
      additive_expression {
        $$ = $1;
      }
    | relational_expression LT additive_expression {
        $$ = new AST("LT", null, @2.first_line,@2.first_column, $1,$3);
      }
    | relational_expression GT additive_expression {
        $$ = new AST("GT", null, @2.first_line,@2.first_column, $1,$3);
      }
    | relational_expression LTEQ additive_expression {
        $$ = new AST("LTEQ", null, @2.first_line,@2.first_column, $1,$3);
      }
    | relational_expression GTEQ additive_expression {
        $$ = new AST("GTEQ", null, @2.first_line,@2.first_column, $1,$3);
      };
```

```
equality_expression :
    relational_expression {
        $$ = $1;
    }
    |   equality_expression EQEQ relational_expression {
        $$ = new AST("EQEQ", null, @2.first_line,@2.first_column, $1,$3);
    }
    |   equality_expression REQEQ relational_expression {
        $$ = new AST("REQEQ", null, @2.first_line,@2.first_column, $1,$3);
    }
    |   equality_expression NOTEQ relational_expression {
        $$ = new AST("NOTEQ", null, @2.first_line,@2.first_column, $1,$3);
    };
```

```
conditional_and_expression :
      exclusive_or_expression {
        $$ = $1;
      }
    | conditional_and_expression ANDAND exclusive_or_expression {
        $$ = new AST("ANDAND", null, @2.first_line,@2.first_column, $1,$3);
      };

conditional_or_expression :
      conditional_and_expression {
        $$ = $1;
      }
    | conditional_or_expression OROR conditional_and_expression {
        $$ = new AST("OROR", null, @2.first_line,@2.first_column, $1,$3);
      };

conditional_expression :
    conditional_or_expression {
        $$ = $1;
    }
    | conditional_or_expression QUESTION expression COLON conditional_expression {
        $$ = new AST("QUESTION", null, @2.first_line,@2.first_column, $1,$3,$5);
    };
```

```
conditional_expression :
    conditional_or_expression {
        $$ = $1;
    }
    | conditional_or_expression QUESTION expression COLON conditional_expression {
        $$ = new AST("QUESTION", null, @2.first_line,@2.first_column, $1,$3,$5);
    };
```

A este nivel (el más alto) se realiza el operador ternario. Al igual que en los anteriores se usa QUESTION de raíz e hijo uno es condición, hijo dos es primera expresión, hijo tres es segunda expresión. Semánticamente se revisa que las dos expresiones sean del mismo tipo.