PL2 - Gramáticas: Analizadores Léxico y Sintáctico.

Blanca Calderón González
Raquel Fernández Cajoto
Franck Michael Fierro Chicaiza
Daniel Medina García
Miguel Virtus Rubio

ÍNDICE

1. ANALIZADOR LÉXICO

El analizador léxico se encarga de descomponer una entrada de caracteres en una secuencia de tokens. La definición del lenguaje que hemos realizado es la siguiente:

1.1. Palabras reservadas

Una palabra reservada tiene un significado gramatical especial para el lenguaje y no puede ser utilizada como un identificador. Como se trata del analizador léxico solo vamos a tener que definir el nombre de las palabras reservadas. Hemos establecido las siguientes:

Condicional if

Está compuesto por las palabras reservadas:

```
IF: 'if';
ELSEIF: 'else if';
ELSE: 'else';
```

Las cuales constituyen entre las tres constituyen el token condicional.

Bucle while

```
WHILE: 'while';
```

Función

```
FUNCTION: 'function';
```

Let

```
LET: 'let';
```

Se va a utilizar para la declaración de variables, aunque a nivel léxico solo hay que definir la palabra que lo caracteriza.

Return

```
RETURN: 'return';
```

1.2. Separadores

Los elementos separadores considerados que generarán un token son los **paréntesis** (izquierdo y derecho), **corchetes** (izquierdo y derecho), el **punto y coma** y la **coma** simple.

Estos se utilizarán al declarar funciones (función nombre () {}), en algunas palabras reservadas como los condicionales (if(condición) {}) y los bucles (while (condición) {}).

Además los paréntesis serán clave para diferenciar los signos de los números negativos y positivos de los operadores matemáticos más adelante, esto se hará de la siguiente forma:

Ejemplo: 4 + (-8); → ponemos paréntesis en -8 para indicar que el signo de este número es negativo, si fuese una operación de resta sería 4 - 8.

Tokens:

```
CI: '{';
CD: '}';
PI: '(';
PD: ')';
PUNTOCOMA: ';';
COMA: ',';
```

1.3. Operadores lógicos

Los operadores lógicos serán:
Unión (OR: ||)
Intersección (AND: &&)
Exclusividad (XOR: ##) y la negación (NEG: !).
Estos operadores nos servirán más adelante para los condicionales.

Tokens:

```
AND:'&&';
OR:'||';
XOR:'##';
NEG:'!';
```

1.4. Operadores matemáticos

Los operadores matemáticos son los siguientes: **suma** '+', **resta** '-', **multiplicación** '*', **división** '/'. **exponencial** '^' y **módulo** '%'.
Los operadores matemáticos indican el tipo de operación que se realizará entre variables y literales numéricos.

Tokens:

```
SUM: '+';
REST: '-';
MULT: '*';
DIV: '/';
EXP: '^';
MOD: '%';
```

1.5. Operadores de asignación.

En este caso solo contaremos con un elemento que será el igual (=) y que nos servirá para la **asignación** de valores a las variables.

Token: `IGUAL: '=';`

1.6. Operadores de comparación

Dentro de estos encontramos las comparaciones básicas como son **menor que** (<), **mayor que** (>), **iguales** (==), **distinto de** (!=), **menor o igual que** (<=) y **mayor o igual que** (>=).
Estos tokens nos servirán a la hora de llevar a cabo los elementos condicionales y bucles.

Tokens:

```
MENORQUE: '<';
MENORIGUALQUE: '<=';
MAYORQUE: '>';
MAYORIGUALQUE: '>=';
IGUALA: '==';
DISTINTO: '!=';
```

1.7. Comentarios

Para capturar los comentarios usamos los siguientes símbolos: para recoger un **comentario de línea** empleamos '%%%' al inicio del comentario, mientras que el **comentario de bloque** se emplea '***' al inicio y final del comentario.
Además, hemos añadido un **'skip'** para los comentarios, por lo que el analizador léxico se encargará de ignorar las cadenas que contengan el patrón de comentario y no se lo pasará al sintáctico.

Tokens:

Comentarios de línea: `LINEA: '%%%'.*?'\n'->skip;`

Comentarios de bloque: `BLOQUE: '***'.*?'***'->skip;`

1.8. Espacios

Mediante el **skip** le indicamos al analizador léxico que no tenga en cuenta los **espacios**, de forma que se los salta sin pasarlos al analizador sintáctico, ya que no lo tiene en cuenta.

Token: `WS: [ \b\n\t\r]+ -> skip;`

1.9. Identificadores

Hemos definido dos tipos de identificadores para nuestro código.
En primer lugar tenemos las letras, que van a poder ser **cualquier carácter definido en el alfabeto** desde la A hasta la Z.
Como se trata de un lenguaje **case sensitive** hemos tenido en cuenta tanto mayúsculas como minúsculas: `LETRA: [A-Z | a-z].`
En segundo lugar tenemos los **nombres de variables**, para su definición hemos establecido lo siguiente: `VAR: (LETRA | '_')(LETRA | DIGITO | '_')*`, de forma que van a tener que empezar por una letra o por el carácter "_", pero nunca van a poder comenzar por un dígito, pero si contenerlo en su identificación tras la aparición de un letra o el carácter "_".

1.10. Tipos de datos

Dentro de tipos de datos hemos definido como básicos **numérico** y **texto**.
A continuación, se describen más detalles sobre cada uno:

1.10.1. Numérico

Al definir los tipos numéricos hemos establecido los siguientes:
Dígito va a ser de tipo fragment ya que no vamos a poder generar un token que se corresponda únicamente con un dígito, sino que va a tener que formar parte de otro token para poder ser usado, como por ejemplo de los enteros o los reales.

Fragment: `fragment DIGITO: [0-9];`

Con **signo** ocurre lo mismo que con dígito, va a ser de tipo fragment ya que va a tener que formar parte de otro token para ser utilizado.

Fragment: `fragment SIGNO: ('-' | '+');`

Entero va a contener todas las combinaciones posibles de los números enteros, y también va a ser tipo fragment, ya que en nuestro lenguaje sólo existe un tipo numérico compuesto tanto por enteros como reales.

Fragment: `fragment REAL: ENTERO?'.'DIGITO+;`

Real va a contener todas las combinaciones posibles de los números reales, y también va a ser tipo fragment, ya que va a formar parte del tipo numérico.

Fragment: `fragment REAL: ENTERO?'.'DIGITO+;`

Numérico es el tipo de números que se pueden definir en nuestro lenguaje, va a poder ser tanto un número real como un número entero.

Token: `NUMERICO: ENTERO | REAL;`

1.10.2. Texto

Para reconocer un dato de tipo texto hemos utilizado la siguiente composición de fragmentos:

El fragmento **Escape** nos ayuda a recoger tokens que contienen **caracteres de escape**, es decir aquellos que empiezan con '\'.

Fragmento: `fragment ESCAPE: '\\'[bntr"\\];`

El token **Texto** utiliza el carácter "" para identificar el inicio y final del token, dentro de la estructura de inicio y final encontramos que la cadena puede puede estar vacía, tener varios caracteres de escape o que puede contener cualquier carácter el cual representamos con la expresión (.)*?.

Token: `TEXTO: '"' (ESCAPE|.)*? '"';`

1.10.3. Booleano

Booleano en la definición de nuestro lenguaje va a poder tener dos posibles valores: 'verdadero' o 'falso'.

Token: `BOOLEANO: 'true'|'false';`

2. ANALIZADOR SINTÁCTICO

El analizador sintáctico construye una representación intermedia del programa, construye una árbol de análisis aplicando las reglas de la gramática y comprobar el orden en que el analizador léxico le va entregando los tokens es válido.

2.1. Axioma

El **axioma** es el elemento inicial que forma el analizador sintáctico y desde el que se parte para analizar el fichero.
En nuestro axioma encontramos `(func puntocoma)* EOF`, `(func puntocoma)*` indica que nuestro programa debe estar compuesto por **cero o más funciones** con un **punto y coma al final** de cada una (al principio se llevó a cabo un analizador mixto que podía contener tanto sentencias sueltas como funciones, pero se tomó la decisión de que aceptase únicamente funciones ya que se quería controlar que no se pudiese encontrar un return suelto no perteneciente a ninguna función ya pensado en el analizador semántico a realizar en el futuro) y `EOF` que es el final del archivo.

Regla: `(func puntocoma)* EOF;`

2.2. Sentencias

Las **sentencias** de nuestro lenguaje pueden estar compuestas por declaraciones, asignaciones o expresiones o condicionales o bucles o valores de retorno o bloques, y siempre finalizarán con un punto y coma.

Regla:

```
sentencia: (decl|asig|expr|cond|bucle|dev|bloque) puntocoma;
```

Ejemplos:

```
let x = 10;
x + y;
if (x > y){};
while (true) {
      “hola”;
      function saludar() {
            4 * 5;
      };
};
```

Dichos elementos los vamos a explicar más detalladamente a continuación.

2.3. Declaraciones

Las **declaraciones** están definidas en el analizador sintáctico como **decl**, dentro de ellas podremos definir la **declaración de una variabl**e mediante la palabra reservada **let** (definida en el analizador léxico) y el nombre de la variable, además también podrá realizar la **asignación de un valor** a la nueva variable declarada.

Regla: `decl: let (identificador | asig);`

Ejemplos :

```
let x;
let x = 10;
```

2.4. Asignaciones

Las **asignaciones** están definidas en el analizador sintáctico como **asig**, dentro de ellas definiremos un **identificador**, el cual se corresponderá con el nombre que identifique a la variable, después del identificador irá un igual “=”, y tras él irá una **expresión**.

Regla: `asig: identificador igual expr;`

Ejemplos:

```
y = 10 + 5;
x = y - z;
cadena = “patata”;
```

2.5. Expresiones

Las expresiones están definidas como **expr**, y en ellas abordaremos la gran mayoría de las composiciones posibles de nuestro código.
Podrán ser identificadores, cadenas, booleanos, polinomios, la llamada a una función, un signo numérico, un numérico.
También, si sigue la sintaxis establecida se puede poner una operación entre paréntesis, y realizar todas las operaciones de las que disponemos (suma, resta, multiplicación, división, módulo, exponente, lógico) entre dos expresiones.

Reglas:

```
expr:   pi expr pd                      |
        expr opermodexp expr            |
        expr opermuldiv expr            |
        expr opersumrest expr           |
        expr opercomparacion expr       |
        expr operlogico expr            |
```

```
                operneg exp                         |
                identificador                       |
                cadena                              |
                booleano                            |
                polinomio                           |
                llamadafuncion                      |
                signonumerico                       |
                numerico                            ;
```

Ejemplos:

```
(-3)+4;
x^6+9;
false;
_variable;
var45;
nombre_a;
7;
(+99);
(6 % 4);
8 * 5;
```

2.6. Bloque de código

Los bloques de código nos permiten armar el cuerpo de **condicionales, bucles y funciones**.
La estructura que sigue es la siguiente, el bloque comienza con el **carácter '{'** y finaliza con el **carácter '}'**.
La estructura puede encontrarse vacía o puede contener varias **sentencias**.

Regla:

```
bloque: ci sentencia* cd;
```

Ejemplo:

```
{                           //inicio bloque
      x + y;                //sentencia 1
      5 > 4;                //sentencia 2
      let variable = 10;    //sentencia 3
};                          //final de bloque
```

2.7. Condicionales

Los condicionales de nuestro analizador sintáctico van a estar compuesto por dos reglas gramaticales:

- La condición está definida como **cond** y contendrá la definición general de la sintaxis del condicional if.

- El cuerpo de la condición definido como **cuerpocondicion** contendrá las expresiones a evaluar de los condicionales if, y se utilizará dentro de cond.

Reglas:

```
cond: if cuerpocondicion bloque (elseif cuerpocondicion bloque)*
(else bloque)?;
cuerpocondicion: pi expr pd;
```

Ejemplo:

```
if (x > 7) {
      x = 7;
      y = 10 + x;
} else {
      x = x + 1;
};
```

2.8. Bucles

Los bucles definidos en el analizador sintáctico son del tipo **buclewhile**, para construir una estructura de tipo bucle utilizamos el siguiente patrón:
La estructura del bucle comienza con la palabra reservada **while** que indica el comienzo del bucle, a continuación se introduce la **condición** que finaliza el bucle utilizando la regla **cuerpocondicion** y la estructura finaliza con la regla **bloque** que contendrá todas las sentencias del bucle.

Regla:

```
buclewhile: while cuerpocondicion bloque;
```

Ejemplo:

```
while (x > 5) {
      a = a + i;
      i = i + 1;
};
```

2.9. Llamada a funciones

Las llamadas a funciones están compuestas por cuatro reglas gramaticales:

- Argumento: expresión que será pasada a la función.
Regla gramatical: argumento: expr;

- Argumentos: debido a que la función puede recibir más de un argumento los argumentos van a estar compuestos de argumentos separados por comas.

Regla: `argumentos: argumento (coma argumento)*;`

- El **cuerpo de argumentos** contendrá los argumentos que le pasamos a la función entre paréntesis, y está definidida como **cuerpoargumentos**.

Regla: `cuerpoargumentos: pi argumentos? pd;`

- **Llamada a la función** contendrá el nombre de la función y los argumentos pasados, está definida como **llamadafuncion**.

Regla: `llamadafuncion: nombrefuncion cuerpoargumentos`

Ejemplos:

```
encenderLuz(intensidad,encendido,apagado)
funcionVacia()
```

2.10. Declaración de funciones

Definidos por la regla gramatical **func** y contiene function (el cual es equivalente al token FUNCTION generado por el analizador léxico), nombrefuncion que puede ser una variable (token VAR) o una letras (token LETRA), es decir, el nombre de la función puede ser una letra o símbolo _ seguida de ninguna o varias letras, dígitos o símbolo _.
Después de nombrefuncion debe estar **cuerpofuncion** formado por un paréntesis izquierdo (pi = PI), cero o un parámetros (incluye uno o más parámetros separados por comas) y un paréntesis derecho (pd = PD).
Por último, encontramos un bloque (este a su vez está compuesto de cero o más sentencias delimitadas por un corchete izquierdo al inicio y otro derecho al final).

Regla gramatical:

```
func: function nombrefuncion cuerpofuncion bloque;
```

Ejemplo:

```
function nombre (parámetro)
{
      x = 5;
```

}

2.11. Retorno de funciones

El retorno de las funciones se encuentran definidas por la regla dev el cual contiene **return** (token RETURN) seguida de cero o una **expresión** (expresión expr puede ser a su vez una exp rodeada por paréntesis, una operación modular, exponencial, multiplicación, división, resta, suma, comparación, operaciones lógica, identificador, cadena, booleano, polinomio, llamadafuncion, signonumerico y numerico).
Estos dos elementos pueden aparecer cero o una vez, es decir, podemos encontrarnos return; y sería válido.

Regla: `dev: return expr?`

Ejemplos:

```
return 4+5;
return x < 4;
return;
return y;
```

2.12. Operadores

Dentro de este campo encontramos varios tipos de operadores, estos son:

2.12.1. Operaciones de comparación:

Identificado con las reglas opercomparacion y podrá ser opermenorque (token MENORQUE <), **opermayorque** (token MAYORQUE >), **opermayorigualque** (token MAYORIGUALQUE >=), **opermenorigualque** (token MENORIGUALQUE <=), **operiguala** (token IGUALA ==) y **operdistinto** (token DISTINTO !=).
Estos operadores son útiles para los condicionales.

Ejemplos:

opermenorque → x **<** y;

opermayorque → a **>** b;

opermayorigualque → c **>=** 8;

opermenorigualque → d **<=** 67;

operiguala→ x **==** y;

operdistinto → d **!=** f;

2.12.2. Operaciones lógicas:

Se corresponde con operlogico el cual contiene **operand** (token AND), **operor** (token OR), **operxor** (token XOR) y **operneg** (token NEG).

Ejemplos:

operand → x = 4 && y > 8;

operor → y = 5 || y <= 9;

operxor → z = 87 ## k = 2;

operneg → t != 3;

2.12.3. Operaciones de multiplicación y división:

A estos pertenecen **opermul** y **operdiv** correspondiendo opermul al token MULT (*) y operdiv al token DIV (/);

Ejemplos:

opermul → 4 * 6;

operdiv → 8 / 2;

2.12.4. Operaciones modulares y exponenciales:

Las reglas son **opermod** que contienen token MOD (%) y **operexp** con token EXP (^) respectivamente.

Ejemplos:

opermod → 6 % 3;

operexp → x^2;

2.12.5. Operaciones de suma y resta:

En esta parte nos encontramos las reglas **opersum** (token SUM) y **operrest** (token REST).

Ejemplos:

opersum → 4 + 7;

operrest → 7 - 6;

NOTA → El **orden de precedencia** de los operadores corresponden al orden en el que aparecen las reglas correspondientes en el analizador sintáctico

llevándose a cabo primero las operaciones de potenciación y modulo, y por último la suma y la resta.
Además, este órden se podrá cambiar introduciendo elementos que queremos llevar a cabo primero entre paréntesis (Ej. 5* (7 + 2) en este caso se hará primero la operación de suma dentro de los paréntesis y luego la multiplicación).

2.13. Polinomios

En cuanto a los polinomios, los hemos definido como **polinomio**. Esta regla nos indica que se debe comenzar y terminar un polinomio con unas comillas simples.
En el interior de estas, debemos introducir un monomio o una suma de monomios.
Los monomios son definidos como **monomio**.
Estos pueden ser un numérico, una letra o un numérico y una letra juntos, todos ellos seguidos de cero o más operadores de exponenciación junto con un numérico. De esta forma, admitimos en nuestra gramática la posibilidad de crear polinomios con exponentes elevados a otros exponentes.

Reglas:

```
polinomio: comillasimple monomio (opersumrest monomio)* comillasimple;
monomio: (numerico letra|numerico|letra) (operexp numerico)*;
```

Ejemplos:

```
‘3x^3-x^2+5x+2’;
‘2x^5’;
```

2.14. Parámetros

Los parámetros se definen como **parametros**, esta regla se encarga de definir la estructura de datos que podemos pasar a funciones.
La estructura está definida por una regla **parámetro** que nos indica que el parámetro puede ser el identificador de una variable. A su vez, esta última se utiliza en la regla **parametros** que está formada siempre por una regla **parametro**, en el caso de haber más parámetros se utiliza utiliza una coma para separar cada uno de ellos.

Reglas:

```
parametros: parametro (coma parametro)*;
parametro: identificador;
```

Ejemplos:

```
function funcion1 (var1){};
function funcion2 (var1, var2, var3){};
```

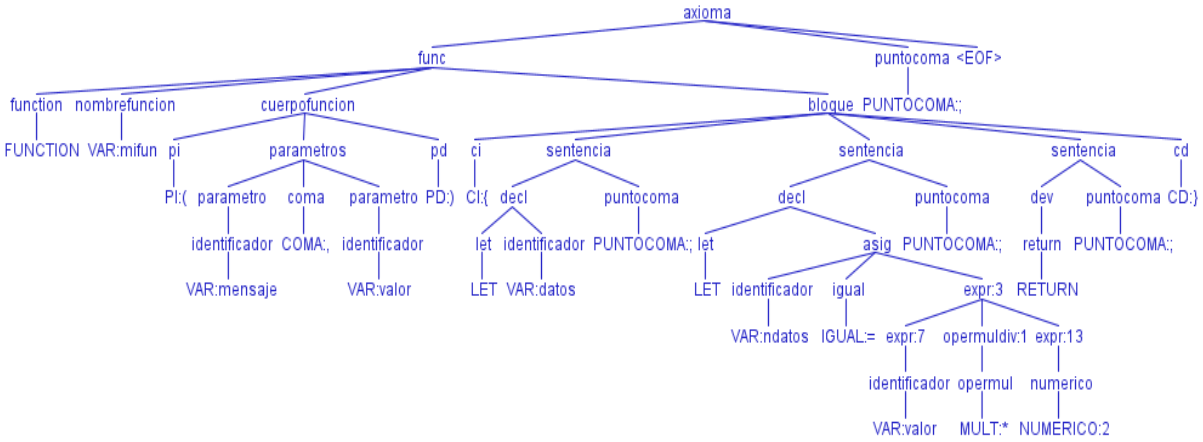
2.15. Elementos del lexer

Los elementos definidos en el analizador léxico se los hemos pasado al analizador sintáctico, asignándole a reglas gramaticales los tokens definidos en el lenguaje.

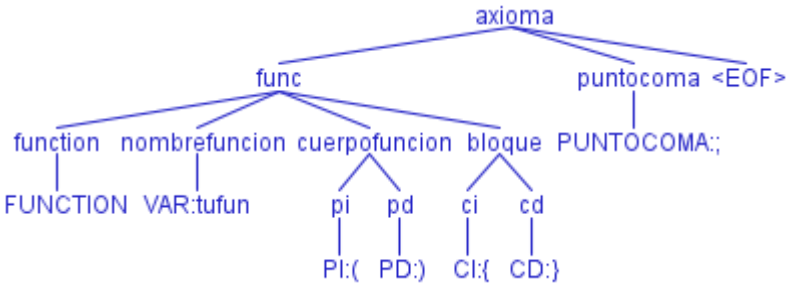
3. ESTRUCTURA AST

Como el ejemplo nos muestra un árbol de un gran tamaño, hemos decidido mostrar un árbol de cada una de las funciones.

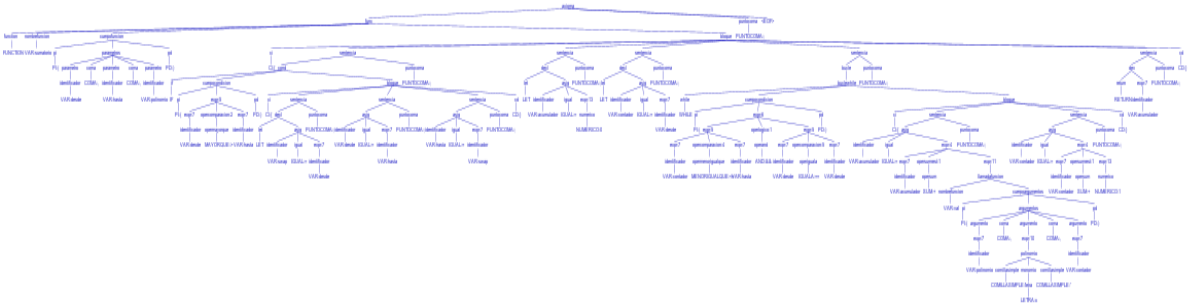
3.1. Función mifun():



3.2. Función tufun():



3.3. Función sumatorio():



3.4. Función test():