Oskar Arturo Gamboa Reyes
A01173648
07 de mayo de 2025
BabyDuck - Entrega #2

Léxico y Sintáxis

Herramienta Seleccionada

Para poder realizar los Lexers y Parsers, use la herramienta ANTLR ya que nativamente trabaja con Java y al ser mi lenguaje de programación seleccionado es una gran opción, además que es extremadamente sencilla de usar y recibe gramáticas poco simplificadas. Para poder crear el proyecto en Java use Maven ya que permite una conexión con ANTLR a través de plugins, el programa se llama y se encuentra dentro de la carpeta *duck*. Las reglas léxicas y gramaticales están definidas en el formato de ANTLR (.g4) como podemos ver son prácticamente idénticas al formato anterior por lo que crear este archivo fue muy sencillo.

Reglas Léxicas Usadas

- PROGRAM : 'program';
- MAIN : 'main';
- END : 'end';
- VAR : 'var';
- VOID : 'void';
- PRINT : 'print';
- WHILE : 'while';
- DO : 'do';
- IF : 'if';
- ELSE : 'else';
- INT_T : 'int';
- FLOAT_T : 'float';
- ASSIGN : '=';
- GT : '>';
- LT : '<';
- NE : '!=';
- SEMICOLON : ';';
- COMMA : ',';
- COLON : ':';
- LPAREN : '(';
- RPAREN : ')';
- LBRACE : '{';
- RBRACE : '}';
- PLUS : '+';

- MINUS : '-';
- MULT : '*';
- DIV : '/';
- ID : [a-zA-Z_][a-zA-Z_0-9]*;
- CTE_INT : [0-9]+;
- CTE_FLOAT : [0-9]+ '.' [0-9]+;
- CTE_STRING: "" (~["\\] | '\\' .)* "";
- WS : [\t\r\n]+ -> skip;
- LBRACKET: '[';
- RBRACKET: ']';

Agregué el token WS para ignorar espacios, saltos de línea y de párrafos, esto como una medida inicial para afrontar este problema. Si en un futuro decido preocuparme por la indentación puedo agregar otras reglas que involucren estos caracteres.

Gramática Formal

programa : PROGRAM ID SEMICOLON vars funcs MAIN body END ;

vars : VAR varDef | ;
varDef : ID varDefID COLON type SEMICOLON multipleVar ;
varDefID : COMMA ID varDefID | ;
multipleVar : varDef | ;

funcs : VOID ID LPAREN funcArguments RPAREN LBRACKET vars body RBRACKET SEMICOLON funcs | ;
funcArguments : ID COLON type multipleFuncArguments | ;
multipleFuncArguments : COMMA funcArguments | ;

body : LBRACE bodyStatement RBRACE ;
bodyStatement : statement bodyStatement | ;

statement : assign
| condition
| cycle
| f_call
| print ;

assign : ID ASSIGN expresion SEMICOLON ;

Oskar Arturo Gamboa Reyes
A01173648
07 de mayo de 2025
BabyDuck - Entrega #2

```
condition : IF LPAREN expresion RPAREN body elsePart SEMICOLON ;
elsePart : ELSE body | ;

cycle : WHILE LPAREN expresion RPAREN DO body SEMICOLON ;

print : PRINT LPAREN printExpresion multiplePrint RPAREN SEMICOLON ;
printExpresion : expresion | CTE_STRING ;
multiplePrint : COMMA printExpresion multiplePrint | ;

expresion : exp expresionFinal ;
expresionFinal : GT exp
               | LT exp
               | NE exp
               | ;

exp : termino expFinal ;
expFinal : PLUS exp
         | MINUS exp
         | ;

termino : factor terminoFinal ;
terminoFinal : MULT termino
         | DIV termino
         | ;

factor : LPAREN expresion RPAREN
         | sumaResta id_cte ;

sumaResta : PLUS
         | MINUS
         | ;

id_cte : ID
         | CTE_INT
         | CTE_FLOAT ;

f_call : ID LPAREN f_call_expresion RPAREN SEMICOLON ;
f_call_expresion : expresion m_f_call_expresion
```

| ;

m_f_call_expresion : COMMA expresion m_f_call_expresion
| ;

type : INT_T | FLOAT_T ;

Test

Para correr los casos creé un pequeño programa en Java (Main.Java), este permite leer varios archivos .txt con el lenguaje creado (se encuentran en la carpeta *main/resources/casos)* e imprime su estado de aceptación, en el caso de ser rechazados indica la posición y el error encontrado.

Comando para correr programa

```
mvn clean compile exec:java -Dexec.mainClass="com.ok.Main"
```

En el caso de los casos que adjunte los primeros 4 deben de ser aceptados y los 2 últimos rechazados.

Caso 1:

```
program p1;
var x, y : int;
z : float;
main {
  x = 5;
  print(x);
}
end
```

Caso 2:

Oskar Arturo Gamboa Reyes
A01173648
07 de mayo de 2025
BabyDuck - Entrega #2

```
program p2;
void suma(a : int, b : float)[
  var c : float;
  {c = a + b;
  print(c);}
];
main {
  x = 1;
  while (x < 5) do {
    if (x > 0) {
      print("positivo");
    };
    x = x + 1;
  };
}
end
```

Caso 3:

```
program p3;
void saludar(nombre : int)[
  {print("Hola");}
];
main {
  saludar(3);
}
end
```

Caso 4:

Oskar Arturo Gamboa Reyes
A01173648
07 de mayo de 2025
BabyDuck - Entrega #2

```
program p4;
main {
  a = 3.5;
  b = a * (2 + 1.5);
  if (b != 0.0) {
    print("ok");
  };
}
end
```

Caso 5:

```
program p5;
main {
  x = 10
  print(x);
}
end
```

Caso 6:

```
program p6;
main {
  y = 2;
  if y > 1 {
    print("fail");
  };
}
end
```

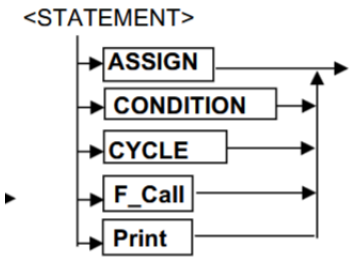
Oskar Arturo Gamboa Reyes
A01173648
07 de mayo de 2025
BabyDuck - Entrega #2

```
Analizando: test1.txt
ACEPTADO

Analizando: test2.txt
ACEPTADO

Analizando: test3.txt
ACEPTADO

Analizando: test4.txt
ACEPTADO

Analizando: test5.txt
RECHAZADO: Error de sintaxis en línea 4:2 -> no viable alternative at input 'print'

Analizando: test6.txt
RECHAZADO: Error de sintaxis en línea 4:5 -> missing '(' at 'y'
```

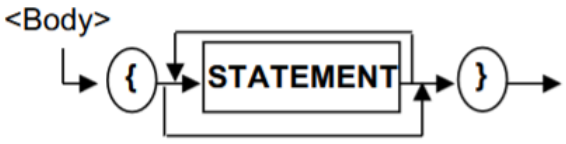
Comentarios

Encontré dos características del lenguaje que no me parecen adecuadas.

- No se pueden poner variables en el BODY:
  Se debe a que STATEMENT no puede adquirir el valor de una variable.



  Esto resulta en que no se pueden definir variables en el main.
- Se tienen que poner corchetes en el BODY de las funciones:
  Debido a la definición del BODY



  Las funciones adquieren esta forma:

Oskar Arturo Gamboa Reyes
A01173648
07 de mayo de 2025
BabyDuck - Entrega #2

```
void suma(a : int, b : float)[
  var c : float;
  {c = a + b;
  print(c);}
];
```

Los corchetes están de más y no son necesarios para poder definir la función.

Estos son aspectos opcionales que podrían ser modificados en entregas posteriores en caso de ser necesario.

Semántica

Estructura Cubo Semántico

Para el cubo semántico use un HashMap de 3 dimensiones que me permite una búsqueda eficiente para validar operaciones entre los dos diferentes tipos de variables que existen en el lenguaje (float e int). En el caso de que sea posible da como resultado el tipo de variable de hacer la operación.

Las operaciones que implementé en este avance: "+", "-", "*", "/", "<", ">", "!=", "=".
Los tipos de variables implementadas: “int” y “float”.

La función “resultado” es la utilizada para hacer llamada a esta estructura en cualquier parte del código y será usada por la semántica en futuros avances.

Estructura Directorio de Funciones

El Directorio de Funciones (Clase) es un HashMap de Funciones (Clase) que a su vez es un HashMap de Variables (Clase). Use HashMaps ya que me permite la búsqueda rápida y eficiente de variables y funciones.

Directorio de Funciones:
Propiedades:

- funciones: HashMap de funciones.

Funciones:

- agregarFuncion(): Validar que la función no exista y agregarla al directorio.
- obtenerFuncion(): Obtener la función a partir del id.
- existeFuncion(): Comprobar que una función no exista a partir del id.

- todas(): Regresa todas las funciones en el directorio.
- toString(): Regresa las funciones como un texto para poder imprimir y hacer pruebas.

Función:
Propiedades:
- tablaVariables: HashMap de variables, se inicializa un HashMap vacío al crear una función.
- parametros: Una lista de los parámetros de la función como id.

Funciones:
- toString(): Regresa la función como un texto para poder imprimir y hacer pruebas.
- agregarVariable(): Validar que la variable no exista y agregala a la tabla.

Variable:
Propiedades:
- tipo: el tipo de la variable (en este caso int o float).

Funciones:
- toString(): Regresa la variable como un texto para poder imprimir y hacer pruebas.

Puntos Neurálgicos

ANTLR genera un árbol de análisis sintáctico a partir del código al ejecutar el parser. Este árbol contiene un nodo por cada regla gramatical reconocida, lo que permite acceder directamente a los elementos (tokens y subreglas) mediante objetos de tipo context.

Debido a esta particularidad de ANTLR ya no es necesario insertar puntos neurálgicos manuales en la gramática para recolectar información semántica, se puede usar un listener generado por ANTLR, el cual reacciona automáticamente al entrar o salir de cada regla.

En este proyecto, se creó un listener personalizado que analiza la semántica del programa durante el recorrido del árbol. Estas son las funciones clave implementadas en el listener:

- enterPrograma: Crea la función principal del programa (guardada con el nombre "global" internamente) y se registra el nombre real del programa para evitar colisiones.

- enterVarDef: Se agrega una variable a la función actual. Si ya existe marca error. Aquí también se comprueba si ya existe una función global con el mismo nombre.
- enterFuncs: Agrega funciones al directorio de funciones, si ya existe marca error. Aquí tambien comprueba que no se llame de la misma forma que el programa.
- procesarArgumentos: Agrega argumentos como variables a la función actual.
- getDirectorio: Esta función me permite obtener el directorio en cualquier punto de la ejecución.
- exitFuncs: Permite salir de la función cuando ya se guardaron todas las variables.

También se implementó en el main un print a partir de la función del Directorio de Funciones toString(), donde imprime el directorio para comprobar su funcionalidad.

```
? Analizando: test3.txt
ACEPTADO
Directorio de Funciones:
DirectorioFunciones{
{saludar=Funcion{tipoRetorno='void', parametros=[int], tablaVariables={nombre=Variable{tipo='int'}}}, global=Funcion{tip
oRetorno='void', parametros=[], tablaVariables={}}, hola=Funcion{tipoRetorno='void', parametros=[int], tablaVariables={n
ombre=Variable{tipo='int'}}}}
}
```

Y en el caso de cometer un error semántico imprimir el problema.

```
? Analizando: test1.txt
RECHAZADO: Variable 'x' ya fue declarada globalmente. No puede redeclararse en 'suma'
```

Comando para correr codigo:

```
mvn exec:java -Dexec.mainClass="com.ok.Main"
```

Link GitHub:
https://github.com/oskgamart/Compiladores/tree/da3381c7b14b91d9ed9b014a28c468f2a223cae5/Scanners%20y%20Parsers