



1.- Diseñar las Expresiones Regulares que representan a los diferentes elementos de léxico que ahí aparecen.

Elemento	Expresión Regular	Definición
Identificador (id)	[a-zA-Z_][a-zA-Z0-9_]*	Una letra o guion bajo seguido de cero o más letras, dígitos o guiones bajos.
Número entero (CTE_INT)	[0-9]+	Uno o más dígitos consecutivos.
Número flotante (CTE_FLOAT)	[0-9]+\.[0-9]+	Uno o más dígitos, un punto decimal, y uno o más dígitos.
Cadena (CTE_STRING)	"[^"]*"	Comillas dobles rodeando cualquier número (incluyendo cero) de caracteres distintos de comillas.
Operadores aritméticos	[+ - * /]	Suma, resta, multiplicación o división.
Operadores relacionales	(< > == !=)	Menor que, mayor que, igual, diferente.
Asignación	=	El signo igual usado para asignar valores.

Puntuación	[(){}\\[,;:]	Paréntesis, llaves, corchetes, coma, punto y coma o dos puntos.
Palabras reservadas	\\b(program main end if else while do print int float void var)\\b	Palabras especiales del lenguaje delimitadas por bordes de palabra.

2.- Listar todos los Tokens que serán reconocidos por el lenguaje

PROGRAM → program

MAIN → main

END → end

INT → int

FLOAT → float

VOID → void

IF → if

ELSE → else

WHILE → while

DO → do

PRINT → print

VAR → var

IDENTIFIER → [a-zA-Z_][a-zA-Z0-9_]*

CTE_INT → [0-9]+

CTE_FLOAT → [0-9]+\\.[0-9]+

CTE_STRING → "[^"]*"

ASSIGN → =

PLUS → +

MINUS → -

MULTIPLY → *

DIVIDE → /

REL_OP → <|>|==|!=

LPAREN → (

RPAREN →)

LBRACE → {

RBRACE → }

LBRACKET → [

RBRACKET →]

COMMA → ,

SEMICOLON → ;

COLON → :

3.- Diseñar las reglas gramaticales (Context Free Grammar) equivalentes a los diagramas.

// Regla Inicial

<programa> ::= PROGRAM IDENTIFIER SEMICOLON <vars> <funcs> MAIN
<Body> END

// Declaración de Variables

<vars> ::= VAR <var_declaration_list> | ϵ

<var_declaration_list> ::= IDENTIFIER COLON <type> SEMICOLON
<var_declaration_list_tail> |

<var_declaration_list_tail> ::= IDENTIFIER COLON <type> SEMICOLON
<var_declaration_list_tail> | ϵ

// Tipos de Datos

<type> ::= INT | FLOAT

// Código

<Body> ::= LBRACE <statement_list> RBRACE

<statement_list> ::= <statement> <statement_list> | ϵ

// Tipos de Statements

<statement> ::= <assign> | <condition> | <cycle> | <f_call> | <print>

// Asignación

<assign> ::= IDENTIFIER ASSIGN <expresion> SEMICOLON

// Condicional

<condition> ::= IF LPAREN <expresion> RPAREN <Body> <else_part>

<else_part> ::= ELSE <Body> | ϵ

// Ciclo While

<cycle> ::= WHILE LPAREN <expresion> RPAREN DO <Body> SEMICOLON

// Escritura

<print> ::= PRINT LPAREN <expresion> RPAREN SEMICOLON

// Llamada a Función: id ([lista_argumentos]) ;

<f_call> ::= IDENTIFIER LPAREN <arg_list> RPAREN SEMICOLON

// Lista de Argumentos para llamada a función

<arg_list> ::= <arg_list_non_empty> | ε

<arg_list_non_empty> ::= <expresion> <args_tail>

<args_tail> ::= COMMA <expresion> <args_tail> | ε

// Expresiones

<expresion> ::= <exp> <rel_op_exp_opt>

<rel_op_exp_opt> ::= REL_OP <exp> | ε

// Expresión Aritmética (Sumas y Restas)

<exp> ::= <termino> <exp_prime>

<exp_prime> ::= PLUS <termino> <exp_prime> | MINUS <termino> <exp_prime> | ε

// Término (Multiplicaciones y Divisiones)

<termino> ::= <factor> <termino_prime>

<termino_prime> ::= MULTIPLY <factor> <termino_prime> | DIVIDE <factor>

<termino_prime> | ε

// Factor (Unidad mínima de una expresión)

<factor> ::= LPAREN <expresion> RPAREN | PLUS <factor> | MINUS <factor> |
IDENTIFIER | CTE_INT | CTE_FLOAT | CTE_STRING

// Definición de Funciones

<funcs> ::= <func_decl> <funcs> | ε

<func_decl> ::= VOID IDENTIFIER LPAREN <params> RPAREN SEMICOLON

<vars> <Body> SEMICOLON

// Parámetros de función

<params> ::= <param_list_non_empty> | ε

<param_list_non_empty> ::= IDENTIFIER COLON <type> <params_tail>

<params_tail> ::= COMMA IDENTIFIER COLON <type> <params_tail> | ε

1. Investigar Herramientas de Generación Automática de Compiladores

- **PLY (Python Lex-Yacc):** Es una implementación pura en Python de las herramientas clásicas Lex y Yacc. Está diseñado para ser similar a las herramientas tradicionales y es conocido por ser bueno para fines educativos debido a sus detallados informes de error. No tiene dependencias externas.
- **Lark:** Es una biblioteca de análisis sintáctico moderna que puede manejar cualquier gramática libre de contexto usando algoritmos como Earley (bueno para ambigüedad) o LALR(1) (rápido, similar a PLY). Genera el árbol de sintaxis automáticamente y tiene buena documentación.
- **ANTLR (ANother Tool for Language Recognition):** Es un generador de analizadores muy potente que soporta múltiples lenguajes destino, incluido Python. Requiere una herramienta (generalmente Java) para generar el código del analizador a partir de la gramática, aunque el *runtime* para ejecutar el analizador generado sí está disponible en Python. Seleccionar la que, a tu juicio, conecte mejor con el lenguaje de desarrollo (y que además tenga buena documentación)

2. Seleccionar la Herramienta

Para Python, tanto PLY como Lark son excelentes opciones con buena documentación.

Pros: Es puramente Python, no requiere pasos de compilación externos, está muy inspirado en las herramientas estándar Lex/Yacc, y está diseñado con un enfoque educativo, proporcionando buenos diagnósticos de error. La documentación parece adecuada. Su implementación directa en Python usando funciones y decoradores puede sentirse natural.

3. Test-Plan para BabyDuck

Objetivo:

Asegurar que el Scanner y Parser identifican correctamente los tokens del lenguaje BabyDuck, construyen adecuadamente las estructuras sintácticas y detectan errores en los casos apropiados.

Estrategia de Pruebas

Reconocimiento de Tokens: Verificar que las palabras reservadas, identificadores, operadores y delimitadores sean correctamente identificados.

Errores Léxicos: Comprobar que símbolos no permitidos o identificadores mal formados sean detectados como errores.

Errores Sintácticos: Validar que errores de sintaxis como paréntesis desbalanceados, expresiones incompletas o estructuras de control mal formadas sean capturados.

Casos Borde: Probar entradas mínimas o atípicas (por ejemplo, programas vacíos, nombres de variables muy largos, etc.).

Reglas Léxicas (Scanner):

Se definieron usando **funciones decoradas** en PLY. Cada token tiene su expresión regular asociada directamente como un docstring o mediante decorador `@TOKEN`.

```
def t_IDENTIFIER(t):  
    r'[a-zA-Z_][a-zA-Z0-9_]*'  
  
    t.type = reserved.get(t.value, 'IDENTIFIER')  
  
    return t
```

Reglas Sintácticas (Parser):

Se definieron como **funciones normales**, donde la especificación de la gramática se da en el **docstring** de cada función, respetando la sintaxis BNF.

```
def p_program(p):  
    """program : PROGRAM IDENTIFIER SEMICOLON vars_opt funcs_opt MAIN body  
    END"""
```

```
print("Programa válido")

p[0] = ("program", p[2], p[4], p[5], p[7]) # Ejemplo simple de resultado
```

Manejo de errores:

Se implementó la función especial `p_error(p)` para capturar errores sintácticos:

```
def p_error(p):

    if p:

        print(f"ERROR DE SINTAXIS: Token inesperado '{p.value}' (Tipo: {p.type}) en
línea {p.lineno}")

    else:

        print("ERROR DE SINTAXIS: Fin inesperado del archivo (EOF)")

    raise SyntaxError
```

Caso de Prueba	Entrada (resumen del código)	Resultado esperado
1	Programa válido con declaraciones de <code>entero</code> , <code>flotante</code> , y estructura <code>si</code>	Análisis exitoso. Scanner detecta tokens correctamente. Parser genera AST.
2	Programa con error léxico (<code>@</code> usado en una expresión)	Scanner detecta error léxico en el carácter <code>@</code> . No pasa a parser.
3	Programa con error sintáctico (falta de <code>;</code> tras asignación)	Scanner procesa tokens. Parser lanza error sintáctico al no encontrar <code>;</code> .
4	Programa válido con expresión matemática correcta	Análisis exitoso. Scanner y parser procesan correctamente.

5	Programa con error de sintaxis (falta cierre de { })	Scanner procesa tokens. Parser reporta error por estructura incompleta.
6	Programa con múltiples declaraciones (a, b, c, d)	Análisis exitoso. Scanner reconoce múltiples variables. Parser genera AST correcto.
7	Programa con número mal formado (10..5)	Scanner detecta error léxico en número malformado. No pasa a parser.

Para el análisis semántico de BabyDuck, se implementaron las siguientes estructuras de datos principales:

Cubo Semántico (cubo_semantico)

- Estructura: Diccionario de Python anidado (dict[str, dict[str, dict[str, str]]]).
- Formato: cubo_semantico[operador][tipo_izq][tipo_der] -> tipo_resultado
- Propósito: Almacena las reglas de validación de tipos para todas las operaciones binarias (aritméticas, relacionales) y la asignación. Define qué combinaciones de tipos son válidas para cada operador y cuál es el tipo de dato resultante.
- Justificación: Un diccionario anidado permite una consulta rápida y directa ($O(1)$ promedio) de la validez y el resultado de una operación dados los tipos de los operandos. Es una representación estándar y eficiente para este tipo de reglas tabuladas. Centraliza la lógica de tipos.

Operaciones Principales:

- Consulta: Realizada por la función check_semantic(op, tipo_izq, tipo_der) que busca la entrada correspondiente y devuelve el tipo_resultado o lanza una excepción (TypeError) si la combinación es inválida.

Directorio de Funciones (function_directory)

- Estructura: Diccionario de Python (dict[str, dict]).
- Formato: function_directory[nombre_funcion] -> {'type': tipo_retorno, 'vars': tabla_variables, ...}
- Propósito: Almacena información sobre cada función definida en el programa, incluyendo su tipo de retorno y una referencia a su tabla de variables local. También incluye una entrada especial para el ámbito 'global'.
- Justificación: Permite un acceso rápido a la información de cualquier función por su nombre. La estructura de diccionario es flexible para añadir más atributos a las funciones si es necesario (e.g., lista de parámetros, dirección de inicio para generación de código).

Operaciones Principales:

- add_function(name, return_type): Añade una nueva función al directorio. Incluye validación para evitar declarar funciones con el mismo nombre dos veces.
- lookup_function(name) (Potencial): Buscaría una función y devolvería su información (actualmente no se usa explícitamente, pero add_function la usa implícitamente para la validación).

Tablas de Variables (vars dentro de function_directory)

- Estructura: Diccionario de Python (dict[str, dict]), anidado dentro de la entrada de cada función en function_directory.
- Formato: function_directory[nombre_func][vars][nombre_var] -> {'type': tipo_variable, 'scope': 'local', ...}
- Propósito: Almacena información sobre cada variable declarada dentro de un ámbito específico (global o una función). Guarda al menos el tipo de la variable.
- Justificación: Asocia naturalmente las variables a su función contenedora. El acceso por nombre de variable dentro del ámbito de la función es eficiente. Permite verificar fácilmente si una variable ya ha sido declarada en ese ámbito y buscar su tipo cuando se utiliza.

Operaciones Principales:

- add_variable(func_name, var_name, var_type): Añade una variable a la tabla de la función especificada (func_name). Incluye validación para evitar declarar variables con el mismo nombre dos veces dentro del mismo ámbito.
- lookup_variable(func_name, var_name): Busca una variable en la tabla de la función especificada y devuelve su tipo. Lanza una excepción si la variable no está declarada en ese ámbito.

Referencias:

Beazley, D. M. (2001). *PLY (Python Lex-Yacc)*. Retrieved from <http://www.dabeaz.com/ply/>

Lark-parser. (2024). *Lark - a modern parsing library for Python*. Retrieved from <https://github.com/lark-parser/lark>

Parr, T. (2023). *ANTLR (ANother Tool for Language Recognition)*. Retrieved from <https://www.antlr.org/>