1. Expresiones Regulares (Léxico)

Identificadores (id): Comienzan con una letra, seguida de letras o números.

LETRA = [a-zA-Z]

DIGITO = [0-9]

ID = {LETRA}({LETRA}|{DIGITO})*

Constantes Enteras (cte_int): Uno o más dígitos.

CTE_INT = {DIGITO}+

Constantes Flotantes (cte_float): Dígitos, un punto decimal, y más dígitos.

CTE_FLOAT = {DIGITO}+\.{DIGITO}+

Constantes String (cte_string): Caracteres entre comillas dobles.

CTE_STRING = \"[^"]*\"

Palabras Reservadas (Keywords): Son literales fijos.

program
main
end
var
int
float
void
if
else
while
do
print

Operadores:

+ (Suma)
- (Resta)
* (Multiplicación)
/ (División)
> (Mayor que)
< (Menor que)
!= (Diferente de)
= (Asignación)

Separadores / Puntuación:

; (Punto y coma)
: (Dos puntos)
, (Coma)
{ (Llave izquierda)
} (Llave derecha)
((Paréntesis izquierdo)
) (Paréntesis derecho)

2. Lista de Tokens

Estos son los nombres de los tokens que el analizador léxico (scanner) generaría:

TOKEN_PROGRAM (para program)
TOKEN_MAIN (para main)
TOKEN_END (para end)
TOKEN_VAR (para var)
TOKEN_INT (para int)
TOKEN_FLOAT (para float)
TOKEN_VOID (para void)
TOKEN_IF (para if)
TOKEN_ELSE (para else)
TOKEN_WHILE (para while)
TOKEN_DO (para do)
TOKEN_PRINT (para print)
TOKEN_ID (para identificadores, ej. miVariable)
TOKEN_CTE_INT (para constantes enteras, ej. 123)
TOKEN_CTE_FLOAT (para constantes flotantes, ej. 3.1416)
TOKEN_CTE_STRING (para constantes string, ej. "Hola")
TOKEN_PLUS (para +)
TOKEN_MINUS (para -)
TOKEN_MULT (para *)
TOKEN_DIV (para /)
TOKEN_GT (para >)
TOKEN_LT (para <)
TOKEN_NE (para !=)
TOKEN_ASSIGN (para =)
TOKEN_SEMICOLON (para ;)
TOKEN_COLON (para :)
TOKEN_COMMA (para ,)
TOKEN_LBRACE (para {)
TOKEN_RBRACE (para })
TOKEN_LPAREN (para ()
TOKEN_RPAREN (para))

3. Gramática Libre de contexto

<programa> ::= TOKEN_PROGRAM TOKEN_ID TOKEN_SEMICOLON [<vars>] [<funcs>] TOKEN_MAIN <cuerpo> TOKEN_END

<vars> ::= TOKEN_VAR <lista_vars_decl> | ε

<lista_vars_decl> ::= <declaracion> [<lista_vars_decl>]

<declaracion> ::= TOKEN_ID TOKEN_COLON <tipo> TOKEN_SEMICOLON

<tipo> ::= TOKEN_INT | TOKEN_FLOAT

<funcs> ::= <funcion> [<funcs>] | ε

<funcion> ::= TOKEN_VOID TOKEN_ID TOKEN_LPAREN [<parametros>] TOKEN_RPAREN TOKEN_LBRACE [<vars>] <cuerpo> TOKEN_RBRACE

<parametros> ::= TOKEN_ID TOKEN_COLON <tipo> [<mas_parametros>] | ε

<mas_parametros> ::= TOKEN_COMMA TOKEN_ID TOKEN_COLON <tipo> [<mas_parametros>] | ε

<cuerpo> ::= TOKEN_LBRACE [<lista_estatutos>] TOKEN_RBRACE

<lista_estatutos> ::= <estatuto> [<lista_estatutos>] | ε

<estatuto> ::= <asignacion> TOKEN_SEMICOLON
| <condicion> TOKEN_SEMICOLON
| <ciclo> TOKEN_SEMICOLON
| <llamada_func> TOKEN_SEMICOLON
| <impresion> TOKEN_SEMICOLON

<asignacion> ::= TOKEN_ID TOKEN_ASSIGN <expresion>

<condicion> ::= TOKEN_IF TOKEN_LPAREN <expresion> TOKEN_RPAREN <cuerpo> [<parte_else>]

<parte_else> ::= TOKEN_ELSE <cuerpo> | ε

<ciclo> ::= TOKEN_WHILE TOKEN_LPAREN <expresion> TOKEN_RPAREN TOKEN_DO <cuerpo>

<impresion> ::= TOKEN_PRINT TOKEN_LPAREN <lista_impresion> TOKEN_RPAREN

<lista_impresion> ::= <arg_impresion> [<mas_args_impresion>]

<arg_impresion> ::= <expresion> | TOKEN_CTE_STRING

<mas_args_impresion> ::= TOKEN_COMMA <arg_impresion>
[<mas_args_impresion>] | ε

<llamada_func> ::= TOKEN_ID TOKEN_LPAREN [<lista_argumentos>]
TOKEN_RPAREN

<lista_argumentos> ::= <expresion> [<mas_argumentos>] | ε

<mas_argumentos> ::= TOKEN_COMMA <expresion> [<mas_argumentos>] | ε

<expresion> ::= <exp> [<op_relacional> <exp>]

<op_relacional> ::= TOKEN_GT | TOKEN_LT | TOKEN_NE

<exp> ::= <termino> [<op_suma_resta> <exp>]

<op_suma_resta> ::= TOKEN_PLUS | TOKEN_MINUS

<termino> ::= <factor> [<op_mult_div> <termino>]

<op_mult_div> ::= TOKEN_MULT | TOKEN_DIV

<factor> ::= TOKEN_LPAREN <expresion> TOKEN_RPAREN
| TOKEN_ID
| <cte>

<cte> ::= TOKEN_CTE_INT | TOKEN_CTE_FLOAT

Comparación de herramientas

Característica	Flex (Scanner)	Bison (Parser)	ANTLR (Scanner + Parser)	JavaCC (Scanner + Parser)	Otros (Ej: PEG, Combinators)
Tipo	Generador de Scanner	Generador de Parser	Generador Scanner+Parser	Generador Scanner+Parser	Bibliotecas/Generadores
Algoritmo Parsing	N/A (Exp. Regulares)	LALR(1), GLR	LL(*), Adaptativo	LL(k)	Recursive Descent, Packrat
Notación Gramática	N/A	BNF (similar a Yacc)	EBNF	EBNF	PEG, Código Host
Lenguaje Salida	C, C++	C, C++, Java, D	Java, C#, Python, JS, C++, Go, Swift, PHP	Java, C++	Varia (Java, Scala, Haskell, C++, Python...)
Integración Lex/Parse	Separado (requiere Parser)	Separado (requiere Scanner)	Integrado	Integrado	Generalmente Integrado
Generación AST	No directa	Requiere acciones semánticas	Soporte integrado (listeners/visitors)	Soporte integrado	Generalmente integrado
Manejo Errores	Básico	Moderado (requiere esfuerzo)	Avanzado	Moderado	Variable, a menudo manual
Soporte Unicode	Limitado/Indirecto	N/A	Sí	Sí	Variable
Licencia	BSD	GPLv3 con excepción	BSD	BSD	Variable (MIT, Apache, etc.)
Madurez/Actividad	Muy Maduro, Estable	Muy Maduro, Estable	Maduro, Muy Activo	Maduro, Menos Activo	Variable
Curva Aprendizaje	Moderada (RegEx)	Moderada/Alta (LR)	Moderada (LL)	Moderada (LL)	Variable
IDE/Herramientas	No específicas	No específicas	Sí (ANTLRWorks, plugins)	No específicas	Variable

Para el desarrollo del compilador del lenguaje BabyDuck, se seleccionaron las herramientas Flex y Bison.

Madurez y Estabilidad: Flex y Bison son herramientas extremadamente maduras, probadas en innumerables proyectos durante décadas.

Estándar de Facto en C/C++: Son las herramientas tradicionales y estándar para generar scanners y parsers en C y C++. Se integran de forma natural en flujos de trabajo basados en C/C++, que es un lenguaje común para la implementación de compiladores.

Rendimiento: Los analizadores generados en C por Flex y Bison son conocidos por su alta velocidad de ejecución, lo cual es importante para herramientas de compilación.

Disponibilidad: Forman parte del conjunto de herramientas estándar en la mayoría de los sistemas operativos tipo Unix (Linux, macOS) y están fácilmente disponibles para Windows. No suelen requerir dependencias complejas.

Recursos Educativos: Existe una vasta cantidad de documentación, tutoriales, libros y ejemplos disponibles para Flex y Bison, lo que facilita el aprendizaje y la resolución de problemas. Son herramientas frecuentemente utilizadas en cursos de teoría de compiladores.

Adecuación al Problema: Para un lenguaje relativamente simple como BabyDuck, las capacidades de Flex para el léxico definido con expresiones regulares y Bison para la gramática, son perfectamente adecuadas. No se requieren las características más avanzadas de herramientas como ANTLR.

4. Reglas Léxicas (Scanner - Flex)

Las expresiones regulares se definieron en un archivo `scanner.l` utilizando la sintaxis de Flex:

Sección de Definiciones: Se usaron alias para patrones comunes (LETRA [a-zA-Z], DIGITO [0-9).

Sección de Reglas (`%% ... %%`)**:** Se especificaron pares patrón { acción }.

Los patrones corresponden a las expresiones regulares definidas (ej. {ID}, {CTE_INT}, "program", "+", etc.).

Para tokens con valor asociado (IDs, constantes), se utiliza la variable yylval (definida en Bison) para almacenar el valor.

Se incluye una regla para manejar caracteres no reconocidos.

5. Reglas Gramaticales (Parser - Bison)

La gramática libre de contexto se definió en un archivo `parser.y` usando la sintaxis de Bison:

Sección de Declaraciones (`%{ ... %}` y `%token`, `%union`, etc.):

Se declaró la unión `yylval` para manejar los tipos de datos asociados a los tokens.

Se declararon todos los nombres de los tokens (`%token TOKEN_PROGRAM TOKEN_ID ...`), asociando aquellos con valores a un campo de la unión (`%token <sval> TOKEN_ID`).

Se definió el símbolo inicial de la gramática (`%start programa`).

Sección de Reglas Gramaticales (`%% ... %%`):

Las reglas se escribieron en una notación similar a BNF:
`simbolo_no_terminal : cuerpo_regla_1 | cuerpo_regla_2 ... ;`

El cuerpo de la regla consiste en secuencias de terminales y no terminales.