Entrega 2: Analizador Léxico y Sintáctico para Little Duck
(Expandido con Entrega 3: Análisis Semántico)

Jesús Daniel Martínez García René Dario Tapia Alcaraz
Monterrey, Nuevo León, a 11 de Mayo del 2025

Introducción

Este documento contiene la definición de las expresiones regulares, la lista de tokens, y las reglas gramaticales que serán reconocidas por el compilador Little Duck. El propósito de esta tarea es diseñar e implementar un análisis léxico y sintáctico que permita identificar y estructurar los componentes del código escrito en este lenguaje. Esta entrega se basa en la Entrega 1 y la expande con el desarrollo del parser. Posteriormente, se detalla la fase de análisis semántico, incluyendo el diseño del Directorio de Funciones, Tablas de Variables, Cubo Semántico y las validaciones correspondientes.

1 Expresiones Regulares.

A continuación, se presentan las expresiones regulares que corresponden a los diferentes elementos léxicos que se identifican en Little Duck.

1.1 Palabras Reservadas

Las palabras reservadas son identificadores que tienen un significado especial dentro del lenguaje y no pueden ser utilizadas como nombres de variables u otros identificadores. Las siguientes son las palabras reservadas:

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

1.2 Identificadores y Constantes

- **Identificadores (id):** `id [a-zA-Z][a-zA-Z0-9]*`
- **Constantes Enteras (cte_int):** `cte_int [0-9]+`
- **Constantes Flotantes (cte_float):** `cte_float [0-9]+.[0-9]+`
- **Constantes de Texto (cte_string):** `cte_string "([^"\\]|\\.)*"`

1.3 Operadores y Delimitadores

- **Operadores Aritméticos:** `+`, `-`, `*`, `/`
- **Operadores de Comparación:** `==`, `!=`, `<`, `>` (Asumimos que `=` es para asignación y `==` para comparación. Para claridad, separamos ASSIGN: `'='` y EQ_COMP: `'=='`)
- **Operador de Asignación:** `=`
- **Delimitadores:** `(`, `)`, `{`, `}`, `;`, `,`, `:`

2 Lista de Tokens

Los tokens que serán reconocidos por el compilador se describen a continuación:

1. PROGRAM: palabra clave `program`
2. MAIN: palabra clave `main`
3. END: palabra clave `end`
4. VAR: palabra clave `var`
5. INT: palabra clave `int`
6. FLOAT: palabra clave `float`
7. VOID: palabra clave `void`
8. IF: palabra clave `if`
9. ELSE: palabra clave `else`
10. WHILE: palabra clave `while`
11. DO: palabra clave `do`
12. PRINT: palabra clave `print`
13. ID: identificadores
14. CTE_INT: constantes enteras
15. CTE_FLOAT: constantes flotantes
16. CTE_STRING: cadenas de texto
17. PLUS: operador `+`
18. MINUS: operador `-`
19. MULT: operador `*`
20. DIV: operador `/`
21. ASSIGN: operador `=` (Modificado de EQUAL para ser asignación)
22. NEQ: operador `!=`
23. LT: operador `<`

24. GT: operador `>`
25. LPAREN: delimitador `(`
26. RPAREN: delimitador `)`
27. LBRACE: delimitador `{`
28. RBRACE: delimitador `}`
29. SEMICOLON: delimitador `;`
30. COMMA: delimitador `,`
31. COLON: delimitador `:`
32. **NUEVO:** EQ_COMP: operador `==` (Para comparación de igualdad)
33. **NUEVO:** DOT: delimitador `.` (Para fin de programa `end .`)

3 Desarrollo de los Analizadores Léxico y Sintáctico

3.1 Investigación y Selección de Herramienta: ANTLR

Para el desarrollo de los analizadores léxico y sintáctico del lenguaje se investigaron varias herramientas de generación automática. Entre las consideradas se estaban:

- **Lex/Flex y Bison/Yacc:** Herramientas clásicas en entornos Unix para C/C++. Flex genera lexers y Bison genera parsers. Son potentes y eficientes, pero requieren la integración de dos herramientas separadas y su sintaxis puede ser menos intuitiva para algunos desarrolladores.
- **JFlex y CUP (o SableCC):** Alternativas para Java. JFlex es un generador de lexers similar a Flex, y CUP es un generador de parsers LALR similar a Yacc. SableCC es otra opción que genera parsers y AST walkers.
- **PLY (Python Lex-Yacc):** Una implementación de Lex y Yacc en Python. Es muy conveniente si el resto del proyecto se desarrolla en Python, pero su rendimiento puede no ser comparable al de herramientas que generan código compilado.
- **ANTLR (ANother Tool for Language Recognition):** Es un potente generador de parsers (LL(*)) que puede generar analizadores léxicos, sintácticos, árboles de sintaxis (parse trees), y ofrece mecanismos de "listeners" y "visitors" para recorrer dichos árboles. Soporta múltiples lenguajes de destino (Java, Python, C#, C++, Go, Swift, JavaScript, Dart).

Selecciónnamos **ANTLR** como la herramienta principal debido a las siguientes ventajas:

- **Generación Integrada de Lexer y Parser:** ANTLR permite definir la gramática léxica y sintáctica en uno o varios archivos `.g4` interrelacionados, generando ambos componentes de forma coordinada.
- **Soporte Multilenguaje:** La capacidad de generar el compilador en varios lenguajes (se eligió Java para este proyecto, por su robustez y ecosistema) ofrece flexibilidad.

- **Excelente Documentación y Comunidad Activa:** ANTLR cuenta con una documentación exhaustiva ("The Definitive ANTLR 4 Reference") y una comunidad grande y activa, lo que facilita la resolución de dudas y problemas.
- **Construcción Automática de Parse Trees y Mecanismos de Recorrido:** Esto es fundamental para las etapas posteriores del compilador (análisis semántico, generación de código intermedio).
- **Manejo Avanzado de Errores:** Proporciona mecanismos de recuperación de errores bastante sofisticados.
- **Sintaxis Intuitiva:** La sintaxis para definir las gramáticas en ANTLR es legible y expresiva.

3.2 Formato de las Reglas de Construcción (Archivos de Gramática .g4)

Las reglas para la construcción, tanto las expresiones regulares para el léxico como las reglas gramaticales para la sintaxis, se definen utilizando el formato específico de ANTLR v4, almacenadas en archivos con extensión `.g4`.

- **Reglas Léxicas (Tokens):** Se definen mediante expresiones regulares asignadas a nombres de tokens en mayúsculas. Por ejemplo: `ID: [a-zA-Z][a-zA-Z0-9]*;`
- **Reglas Gramaticales (Parser):** Se definen usando una notación similar a EBNF (Extended Backus-Naur Form), con nombres de reglas en minúsculas. Por ejemplo: `asignacion: ID ASSIGN expresion SEMICOLON;`

3.3 Archivo de Gramática del Lexer (LittleDuckLexer.g4)

El archivo `LittleDuckLexer.g4` define las reglas léxicas del lenguaje. Cada token se describe a través de expresiones regulares.

```
// LittleDuckLexer.g4
lexer grammar LittleDuckLexer;

// Palabras Reservadas
PROGRAM : 'program';
MAIN    : 'main';
END     : 'end';
VAR     : 'var';
INT     : 'int';
FLOAT   : 'float';
VOID    : 'void';
IF      : 'if';
ELSE    : 'else';
WHILE   : 'while';
DO      : 'do';
PRINT   : 'print';

// Identificadores
```

```
ID : [a-zA-Z][a-zA-Z0-9]*;

// Constantes
CTE_INT    : [0-9]+;
CTE_FLOAT  : [0-9]+ '.' [0-9]+;
CTE_STRING : '"' (ESC | ~["\\])* '"'; // Maneja escapes básicos
fragment ESC : '\\' (_ | 'u' HEX HEX HEX HEX) ; // Para escapes
fragment HEX : [0-9a-fA-F] ;

// Operadores
PLUS    : '+';
MINUS   : '-';
MULT    : '*';
DIV     : '/';
ASSIGN  : '=';    // Asignación
EQ_COMP : '==';   // Comparación de igualdad
NEQ     : '!=';
LT      : '<';
GT      : '>';

// Delimitadores
LPAREN    : '(';
RPAREN    : ')';
LBRACE    : '{';
RBRACE    : '}';
SEMICOLON : ';';
COMMA     : ',';
COLON     : ':';
DOT       : '.'; // Para el 'end.'

// Ignorar espacios en blanco y comentarios
WS : [ \t\r\n]+ -> skip;
// COMMENT : '//' ~[\r\n]* -> skip; // Ejemplo de comentario de una línea
// ML_COMMENT : '/*' .*? '*/' -> skip; // Ejemplo de comentario multilínea
```

3.3.1 Explicación del Archivo de Gramática del Lexer

- **Palabras Reservadas:** Se definen explícitamente como literales (ej. `'program'`). ANTLR les dará prioridad sobre reglas más generales como `ID`.
- **Identificadores y Constantes:** Definidos con sus respectivas expresiones regulares. `CTE_FLOAT` fue corregida para requerir un punto decimal. `CTE_STRING` se mejoró para manejar secuencias de escape.
- **Operadores y Delimitadores:** Definidos como literales. Se distingue `ASSIGN` (`=`) de `EQ_COMP` (`==`). Se añade `DOT`.
- **Espacios en Blanco y Comentarios:** La regla `WS` utiliza `-> skip` para indicar al lexer que ignore estos caracteres. Se incluyen ejemplos comentados para futuros tipos de comentarios.

3.4 Generación del Lexer

Para generar el código del lexer en Java a partir de `LittleDuckLexer.g4`:

3.5 Archivo de Gramática del Parser (LittleDuckParser.g4)

El archivo `LittleDuckParser.g4` define la estructura sintáctica de Little Duck, usando los tokens definidos en `LittleDuckLexer.g4`.

```
// LittleDuckParser.g4
parser grammar LittleDuckParser;

options { tokenVocab = LittleDuckLexer; } // Importa tokens de LittleDuckLexer

// Regla inicial
programa : PROGRAM ID SEMICOLON dec_vars? bloque_principal END DOT;

dec_vars : VAR dec_var_aux+;
dec_var_aux : lista_ids COLON tipo SEMICOLON;
lista_ids : ID (COMMA ID)*;
tipo : INT | FLOAT | VOID; // VOID podría ser solo para funciones

bloque_principal : MAIN LPAREN RPAREN bloque;

bloque : LBRACE estatuto* RBRACE; // Un bloque puede tener cero o más estatutos

estatuto : asignacion
         | condicion
         | ciclo_w
         | ciclo_do_w // Renombrado para claridad
         | escritura
         ;

asignacion : ID ASSIGN expresion SEMICOLON;

escritura : PRINT LPAREN print_args RPAREN SEMICOLON;
print_args : expresion (COMMA expresion)*
           | CTE_STRING (COMMA (expresion | CTE_STRING))* // Permitir imprimir múltiples cosas
           ;

condicion : IF LPAREN expresion RPAREN bloque (ELSE bloque)?;

ciclo_w : WHILE LPAREN expresion RPAREN DO bloque; // Como en el OCR original

ciclo_do_w : DO bloque WHILE LPAREN expresion RPAREN SEMICOLON;

// Expresiones
expresion : exp_comp ( (EQ_COMP | NEQ | LT | GT ) exp_comp )*; // Operadores de comparación

exp_comp : exp_arit ( (PLUS | MINUS) exp_arit )*; // Adición y sustracción
```

```
exp_arit : termino ( (MULT | DIV) termino )*; // Multiplicación y división

termino : LPAREN expresion RPAREN   // Expresión entre paréntesis
        | (PLUS | MINUS)? factor    // Factor con signo opcional
        ;

factor : ID
       | CTE_INT
       | CTE_FLOAT
       // | llamada_func // Podría añadirse en el futuro
       ;

// Nota: CTE_STRING no se incluye en 'factor' para expresiones aritméticas/lógicas,
// solo en 'escritura'. Si se permitieran operaciones con strings, se añadiría aquí.
```

3.5.1 Explicación del Archivo de Gramática del Parser

- `options { tokenVocab = LittleDuckLexer; }`: Indica a ANTLR que los tokens utilizados en este parser están definidos en `LittleDuckLexer.g4`.
- `programa`: Define la estructura general de un programa Little Duck. Termina con `END DOT`.
- `dec_vars, dec_var_aux, lista_ids, tipo`: Reglas para la declaración de variables.
- `bloque_principal`: Define el punto de entrada `main`.
- `bloque`: Una secuencia de estatutos encerrados entre llaves `{}`.
- `estatuto`: Define los diferentes tipos de sentencias permitidas (asignación, condición, ciclos, escritura).
- `expresion, exp_comp, exp_arit, termino, factor`: Definen la jerarquía y precedencia de operadores para las expresiones aritméticas y de comparación. Utilizamos recursión por la izquierda indirecta para manejar la asociatividad y precedencia.
    - `expresion`: Nivel más alto, maneja operadores de comparación (`==`, `!=`, `<`, `>`).
    - `exp_comp`: Siguiente nivel, maneja suma y resta (`+`, `-`).
    - `exp_arit`: Siguiente nivel, maneja multiplicación y división (`*`, `/`).
    - `termino`: Elementos básicos de una expresión, incluyendo expresiones entre paréntesis y factores con signo opcional.
    - `factor`: Los operandos más básicos: identificadores y constantes numéricas.
-
- `escritura`: Permite imprimir expresiones y/o constantes de texto.

3.6 Generación del Parser

Para generar el código del parser a partir de `LittleDuckParser.g4`:

3.7 Diseño del Plan de Pruebas

Para asegurar que los analizadores léxico y sintáctico funcionen correctamente, tenemos un Test Plan detallado.

3.7.1 Objetivos del Test Plan

- **Lexer:**
    - Verificar que todos los tokens definidos (palabras reservadas, identificadores, constantes, operadores, delimitadores) sean correctamente identificados.
    - Confirmar que los espacios en blanco y comentarios sean ignorados.
    - Validar el manejo de errores léxicos (símbolos no reconocidos).
- **Parser:**
    - Verificar que las construcciones gramaticales válidas sean aceptadas (programas completos, declaraciones, estatutos, expresiones).
    - Confirmar que se respeten las reglas de precedencia y asociatividad de operadores.
    - Asegurar que los errores de sintaxis sean detectados y reportados adecuadamente (ej. falta de punto y coma, llaves no balanceadas, estructura incorrecta).

3.7.2 Estrategia de Pruebas del Lexer

- **Prueba de palabras reservadas:** Código fuente con cada palabra reservada, aislada y en combinación.
    - Ej: `program main end var int float void if else while do print`
- **Prueba de identificadores:** Nombres de variables válidos e inválidos.
    - Ej: `valida variable1 v_1 _invalido 1invalido var@#!`
- **Prueba de constantes:**
    - Enteras: `123 0 98765`
    - Flotantes: `0.5 123.456 7.0`
    - Cadenas: `"Hola" "Cadena con \"escapes\"" ""`
- **Prueba de operadores y delimitadores:** Combinaciones de todos los operadores y delimitadores.

- **Prueba de errores léxicos:** Caracteres o secuencias no permitidas.
    - Ej: `int x = 5 #esto es un error;`
-

3.7.3 Estrategia de Pruebas del Parser

Se crearán archivos de prueba `.ld` (Little Duck) con diversos casos:

Programa mínimo válido: `littleduck`

```
program Test1;
main()
{
}
```

Declaración de variables: `littleduck`

```
program TestVars;
var
  i, j, k : int;
  x, y : float;
  flag : int;
main()
{
  // sin estatutos
}
```

Asignaciones y expresiones aritméticas: `littleduck`

```
program TestExpr;
var i, j : int; result : float;
main()
{
  i = 10;
  j = i * 2 + (5 - 3) / 2;
  result = i + j * 0.5;
}
```

Estructuras de control (if, else, while, do-while): `littleduck`

```
var count : int;
main()
{
  count = 0;
  if (count < 5) {
    print("Menor que 5");
  } else {
    print("Mayor o igual a 5");
  }
```

```
  while (count < 3) do {
    count = count + 1;
    print("Iteracion while:", count);
  }

  do {
    count = count - 1;
    print("Iteracion do-while:", count);
  } while (count > 0);
}
```

Pruebas de escritura (print): `littleduck`

```
program TestPrint;
var num : int; name : string; // Asumiendo que string es un tipo o CTE_STRING puede ser usado
main()
{
  num = 100;
  print("El numero es:", num, "y su doble es:", num * 2);
  print("Hola Mundo");
}
```

- end.
- *(Nota: La declaración*
- **Casos de error sintáctico:**
  - Falta de `;`: `program Error1; var x: int main() { x = 5 } end.`
  - Llaves desbalanceadas: `program Error2; main() { x = 1; end.`
  - Palabra clave mal usada: `program Error3; var x: integer; main() {} end.` (integer no es int)
  - Expresión mal formada: `program Error4; main() { x = 5 + * 3; } end.`
-

3.7.4 Herramientas para las Pruebas

ANTLR TestRig (grun): Utilidad de ANTLR que permite probar gramáticas rápidamente contra entradas de prueba, mostrando el árbol de análisis sintáctico (parse tree) o la secuencia de tokens.

```
# Para probar el lexer
grun LittleDuckLexer tokens -tokens mi_archivo_de_prueba.ld

# Para probar el parser (y ver el árbol gráfico)
```

- grun LittleDuckParser programa -gui mi_archivo_de_prueba.ld
- **Pruebas Unitarias:** Se pueden escribir pruebas unitarias en Java (usando JUnit, por ejemplo) que alimenten cadenas de texto al lexer y parser generados, y verifiquen que

no se produzcan errores para entradas válidas, o que se reporten los errores esperados para entradas inválidas.

5. Análisis Semántico

Tras la correcta identificación de tokens (análisis léxico) y la validación de la estructura gramatical (análisis sintáctico), la siguiente fase crucial en la compilación es el análisis semántico. Esta etapa se encarga de verificar el significado del código fuente y asegurar que las construcciones sintácticamente válidas también tengan sentido lógico y coherencia dentro del lenguaje Little Duck.

5.1 Introducción al Análisis Semántico

El análisis semántico tiene como objetivos principales:

1. **Gestión de Símbolos:** Mantener un registro de todos los identificadores (variables, funciones) declarados en el programa, junto con su información relevante (tipo, ámbito, etc.). Esto se logra mediante el Directorio de Funciones y las Tablas de Variables.
2. **Chequeo de Tipos:** Verificar que los operandos en las expresiones sean de tipos compatibles con los operadores utilizados y que las asignaciones sean válidas. El Cubo Semántico es la herramienta central para esta tarea.
3. **Validación de Ámbito:** Asegurar que los identificadores se utilicen correctamente según las reglas de ámbito (por ejemplo, una variable debe ser declarada antes de ser usada en su ámbito correspondiente).
4. **Control de Flujo:** Validar que las expresiones en estructuras de control (como `if`, `while`) produzcan resultados compatibles con una condición booleana.

Para Little Duck, nos enfocaremos en la creación del Directorio de Funciones (aunque inicialmente solo contendrá `main` y el ámbito global), las Tablas de Variables, el Cubo Semántico y la identificación de los puntos neurálgicos en la gramática donde se realizarán estas validaciones.

5.2 Directorio de Funciones (ProcTable)

El Directorio de Funciones (o "ProcTable") almacenará información sobre cada función definida en el programa. Para la versión actual de Little Duck, esto incluirá principalmente la función `main` y un concepto de ámbito "global" para variables declaradas fuera de `main` (aunque la gramática actual las anida bajo `program` antes de `main`).

5.2.1 Estructura y Justificación

- **Estructura Seleccionada:** Un `HashMap` (o diccionario) es una elección adecuada.

    - **Clave:** Nombre de la función (String, ej: "main", "global").
    - **Valor:** Un objeto o estructura que contenga los detalles de la función.
-
- **Justificación:**
    - **Eficiencia de Búsqueda:** Permite un acceso rápido (promedio O(1)) a la información de una función por su nombre.
    - **Flexibilidad:** Fácil de añadir nuevas funciones si el lenguaje se expande.
    - **Claridad:** Representa de forma natural la colección de funciones.
-

5.2.2 Información Almacenada por Función

Cada entrada en el Directorio de Funciones (representada por el objeto valor) contendrá:

- **Nombre:** (String) El identificador de la función.
- **Tipo de Retorno:** (Enum/String: `VOID`, `INT`, `FLOAT`). Para `main` y el ámbito "global" en Little Duck, esto sería `VOID`.
- **Tabla de Variables:** Una referencia o instancia de una Tabla de Variables (VarTable) para las variables locales de esta función (o variables globales si es la entrada "global").
- **(Opcional para futuras expansiones):**
    - Lista de Parámetros: (Lista de objetos {nombre_param, tipo_param})
    - Dirección de inicio de código (para generación de código).
    - Contadores de recursos (número de variables de cada tipo, tamaño de temporales).
-

5.2.3 Operaciones Principales

- `addFunction(name, returnType)`: Añade una nueva función al directorio. Verifica si ya existe para evitar redefiniciones. Crea su VarTable asociada.
- `getFunction(name)`: Retorna el objeto con la información de la función.
- `doesFunctionExist(name)`: Verifica si una función ya ha sido declarada.
- `addVariableToFunction(functionName, varName, varType)`: Delega la adición de una variable a la VarTable de la función especificada.
- `getVariableFromFunction(functionName, varName)`: Busca una variable en la VarTable de la función.

5.3 Tablas de Variables (VarTable)

Cada función (incluyendo el ámbito global) tendrá su propia Tabla de Variables para almacenar información sobre los identificadores declarados dentro de ese ámbito.

5.3.1 Estructura y Justificación

- **Estructura Seleccionada:** Un `HashMap` (o diccionario) por cada ámbito.
    - **Clave:** Nombre de la variable (String).
    - **Valor:** Un objeto o estructura que contenga los detalles de la variable.
-
- **Justificación:**
    - **Eficiencia de Búsqueda:** Rápido acceso a la información de una variable por su nombre dentro de su ámbito.
    - **Manejo de Ámbito:** Naturalmente encapsula las variables de un scope específico.
-

5.3.2 Información Almacenada por Variable

Cada entrada en una Tabla de Variables contendrá:

- **Nombre:** (String) El identificador de la variable.
- **Tipo:** (Enum/String: `INT`, `FLOAT`).
- **(Opcional para futuras expansiones):**
    - Dirección de memoria virtual asignada.
    - Valor (si es una constante, aunque las constantes tienen su propio token).
    - Dimensiones (si se implementaran arreglos).
-

5.3.3 Operaciones Principales

- `addVariable(name, type)`: Añade una nueva variable a la tabla del ámbito actual. Verifica si ya existe en *ese ámbito* para evitar re-declaraciones locales.
- `getVariable(name)`: Retorna el objeto con la información de la variable.
- `doesVariableExist(name)`: Verifica si una variable ya ha sido declarada en el ámbito actual.
- `getVariableType(name)`: Retorna el tipo de la variable.

5.3.4 Manejo de Ámbitos (Scopes)

- Se mantendrá una referencia al "ámbito actual", que será la clave de la función activa en el Directorio de Funciones (ej: "global", "main").
- Cuando se declara una variable (`var ...`), se añade a la VarTable del ámbito actual.
- Cuando se usa un identificador en una expresión, se busca primero en la VarTable del ámbito actual (`main`). Si no se encuentra y el lenguaje permitiera variables globales accesibles desde funciones, se buscaría en la VarTable del ámbito "global". (Para Little

Duck, la gramática anida `var` antes de `main`, implicando que estas son las "globales" o las únicas disponibles para `main`).

5.4 Cubo Semántico

El Cubo Semántico es una estructura de datos multidimensional que define el tipo resultante de aplicar un operador a uno o dos operandos de tipos específicos. También se usa para validar la compatibilidad de tipos en asignaciones.

5.4.1 Propósito y Diseño

- **Propósito:** Sistematizar el chequeo de tipos en expresiones y asignaciones. Evita una cascada de `if-else` statements.
- **Diseño:** Conceptualmente, es una tabla donde las dimensiones son: `Operador`, `TipoOperando1`, `TipoOperando2` (si es binario), y el valor en la celda es el `TipoResultado`. Si la operación no es válida, se indica un tipo `ERROR`.

5.4.2 Definición para Little Duck

Tipos de datos en Little Duck: `INT`, `FLOAT`.
Operadores Aritméticos: `+`, `-`, `*`, `/`
Operadores de Comparación: `==`, `!=`, `<`, `>`
Operador de Asignación: `=`
(Para los operadores de comparación, el "resultado" no es un tipo de dato almacenable en Little Duck como `BOOLEAN`, sino una condición que se evalúa. El cubo puede retornar un tipo especial `BOOLEAN_CONDITION` o simplemente validar compatibilidad y dejar que la gramática maneje el flujo).

Operador	Operando 1	Operando 2	Tipo Resultado
`+,-,*`	`INT`	`INT`	`INT`
`+,-,*`	`INT`	`FLOAT`	`FLOAT`
`+,-,*`	`FLOAT`	`INT`	`FLOAT`
`+,-,*`	`FLOAT`	`FLOAT`	`FLOAT`
`/`	`INT`	`INT`	`FLOAT` (División puede resultar en float)
`/`	`INT`	`FLOAT`	`FLOAT`

/	FLOAT	INT	FLOAT
/	FLOAT	FLOAT	FLOAT
==, !=,<,>	INT	INT	BOOLEAN_CONDITION (o INT si se mapea a 0/1)
==, !=,<,>	FLOAT	FLOAT	BOOLEAN_CONDITION (o INT)
==, !=,<,>	INT	FLOAT	BOOLEAN_CONDITION (o INT)
==, !=,<,>	FLOAT	INT	BOOLEAN_CONDITION (o INT)
=	INT	INT	INT
=	FLOAT	FLOAT	FLOAT
=	FLOAT	INT	FLOAT (Promoción permitida en asignación)
=	INT	FLOAT	ERROR (Pérdida de precisión, no permitida sin cast explícito)
Otros	*Cualquiera*	*Cualquiera*	ERROR

Nota:

5.5 Puntos Neurálgicos y Validaciones Semánticas

Estos son los puntos en las reglas gramaticales (del `LittleDuckParser.g4`) donde se deben insertar acciones semánticas:

1. **Regla**
    - Al inicio (`PROGRAM ID SEMICOLON`): Crear la entrada "global" (o el nombre del programa) en el Directorio de Funciones con tipo `VOID`. Establecer este como ámbito actual.
    - Al final (`END DOT`): Limpiar/finalizar el análisis semántico.
2.
3. **Regla**
    - `lista_ids COLON tipo SEMICOLON`:
        - Para cada `ID` en `lista_ids`:
            - Obtener el `tipo` (`INT` o `FLOAT`).
            - Intentar añadir la variable (`ID`, `tipo`) a la VarTable del ámbito actual.

        - **Validación:** Si `addVariable` detecta que el `ID` ya existe en la VarTable del ámbito actual, reportar "Error: Variable [ID] doblemente declarada".
    -
  -
4.
5. **Regla**
   - Al encontrar `MAIN`:
     - Intentar añadir "main" al Directorio de Funciones con tipo `VOID`.
     - **Validación:** Si "main" ya existe, error (aunque en Little Duck solo hay uno).
     - Establecer "main" como el ámbito actual. Su VarTable hereda/puede ver las variables globales (o en Little Duck, las declaradas antes de `main` son el único otro scope).
   -
6.
7. **Regla**
   - Si es `ID`:
     - Buscar `ID` en la VarTable del ámbito actual ("main"). Si no está, buscar en "global".
     - **Validación:** Si `ID` no se encuentra en ningún ámbito accesible, reportar "Error: Variable [ID] no declarada".
     - Obtener y propagar el tipo del `ID`.
   -
   - Si es `CTE_INT`: Propagar tipo `INT`.
   - Si es `CTE_FLOAT`: Propagar tipo `FLOAT`.
   - Si es `LPAREN expresion RPAREN`: Propagar el tipo de `expresion`.
8.
9. **Reglas de Expresión (**
   - Para cada operador (`+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`):
     - Obtener los tipos de los operandos (izquierdo y derecho).
     - Consultar el Cubo Semántico: `resultado_tipo = Cubo[operador][tipo_op1][tipo_op2]`.
     - **Validación:** Si `resultado_tipo` es `ERROR`, reportar "Error: Tipos incompatibles en operación [operador] entre [tipo_op1] y [tipo_op2]".
     - Propagar `resultado_tipo`.
   -
10.
11. **Regla**

    - Obtener tipo de `ID` (LHS) (verificando que esté declarada).
    - Obtener tipo de `expresion` (RHS).
    - Consultar el Cubo Semántico para el operador = : `resultado_tipo = Cubo[ASSIGN][tipo_LHS][tipo_RHS]`.
    - **Validación:** Si `resultado_tipo` es `ERROR`, reportar "Error: Tipo incompatible para asignación. No se puede asignar [tipo_RHS] a [ID] de tipo [tipo_LHS]".
12.
13. **Reglas de Estatutos de Control (**
    - `IF LPAREN expresion RPAREN ...`
    - `WHILE LPAREN expresion RPAREN ...`
    - `DO bloque WHILE LPAREN expresion RPAREN ...`
        - Obtener el tipo de la `expresion` de la condición.
        - **Validación:** Verificar que este tipo sea `BOOLEAN_CONDITION` según el Cubo Semántico (es decir, el resultado de una comparación válida, o si el lenguaje permite, que sea numérico). Si no, reportar "Error: Se esperaba una expresión booleana/comparación en la condición de [IF/WHILE/DO-WHILE]".
    -
14.
15. **Regla**
    - Para cada `expresion` o `CTE_STRING` en `print_args`:
        - Obtener el tipo de la `expresion`.
        - **Validación:** (Generalmente, la mayoría de los tipos básicos y strings son imprimibles. Little Duck permite expresiones y `CTE_STRING`. No se requiere una validación de tipo estricta aquí más allá de que la expresión sea válida).
    -
16.

5.6 Integración con ANTLR (Listeners/Visitors)

Para implementar estas acciones semánticas, se utilizará el patrón Listener o Visitor proporcionado por ANTLR sobre el árbol de sintaxis (Parse Tree) generado por el parser.

- **Listeners:** Se crea una clase que hereda de `LittleDuckBaseListener.java`. Se sobrescriben los métodos `enterRule()` y `exitRule()` para las reglas gramaticales relevantes (los puntos neurálgicos).
    - `enterRule()`: Útil para acciones al inicio de una regla (ej: crear un nuevo ámbito).

    - `exitRule()`: Útil para acciones al final de una regla, cuando ya se procesaron sus hijos (ej: chequeo de tipos en una expresión, donde los tipos de los operandos ya fueron determinados).
-
- **Visitors:** Se crea una clase que implementa `LittleDuckVisitor.java`. Se sobrescriben los métodos `visitRule()`. El programador controla explícitamente la visita a los nodos hijos y puede retornar valores. Es potente para cálculos que dependen de los resultados de los nodos hijos (como la evaluación de tipos en expresiones).

Para la gestión de tablas de símbolos y chequeo de tipos, una combinación o el uso extensivo de `exitRule()` en un Listener suele ser efectivo. Por ejemplo, en `exitAsignacion()`, los tipos del ID y de la expresión ya habrán sido (idealmente) calculados y almacenados por las visitas a sus respectivos nodos, listos para ser comparados. Se usarán pilas auxiliares para manejar los operandos y tipos durante el procesamiento de expresiones.

6. Conclusión

El desarrollo del analizador léxico (scanner) y el analizador sintáctico (parser) para el lenguaje Little Duck utilizando ANTLR ha progresado significativamente. Se han definido las expresiones regulares para los tokens y las reglas gramaticales para la estructura del lenguaje. Los archivos de gramática `LittleDuckLexer.g4` y `LittleDuckParser.g4` han sido creados y detallados. Con la adición de la fase de análisis semántico, se han diseñado las estructuras de datos clave como el Directorio de Funciones, las Tablas de Variables y el Cubo Semántico, además de identificar los puntos neurálgicos para realizar las validaciones de significado y coherencia.

El plan de pruebas diseñado abarca tanto el lexer como el parser, y se extenderá para cubrir las validaciones semánticas, con el objetivo de verificar la correcta identificación de tokens, la validación de la estructura sintáctica, y la coherencia semántica del lenguaje. Las herramientas proporcionadas por ANTLR, como TestRig (grun), junto con la implementación de Listeners/Visitors para las acciones semánticas, serán cruciales para la depuración y validación inicial.

Las próximas etapas del proyecto se centrarán en la implementación detallada del análisis semántico, utilizando los árboles de sintaxis generados por el parser y las estructuras definidas, para verificar la coherencia y el significado del código fuente. Esto será seguido por la generación de código intermedio y, finalmente, código objeto. Este documento servirá como base y referencia continua para estas futuras fases.