Universidad de Córdoba

Escuela Politécnica Superior de Córdoba

Grado de Ingeniería informática

Segundo cuatrimestre del tercer curso del año academico 2024/2025

Especialidad: computación

Procesadores de lenguajes

Intérprete de pseudocódigo en ingles: interpreter

Autores:

Sergio Alcántara Avilés
Rafael David Tortosa Bueno

Córdoba 22 de junio de 2025

Índice

Índice de figuras

1 Introducción

En este trabajo se partida del código proporcionado por el profesor en el ejemplo17 de la practica 2 de la asignatura de procesadores de lenguajes. En conjunto, el trabajo ha consistido en extender dicha gramática y árbol AST para dar soporte a nuevos constructos de control de flujo, operadores y funciones; garantizando que Flex reconozca las nuevas palabras reservadas y que Bison maneje correctamente los tipos semánticos y genere el árbol de sintaxis adecuado.

En este documento se exploraran y explicaran los siguientes campos:

- Lenguaje de pseudocódigo: donde se verán los componentes léxicos (palabras reservadas, identificadores ...) así como las sentencias.
- Tabla de símbolos: resumen y descripción de las clases utilizadas.
- Análisis léxico: descripción del fichero de flex utilizado para definir y reconocer los componentes léxicos.
- Análisis sintáctico: descripción del fichero de bison utilizado para describir la gramática de contexto libre.
- Código de AST: resumen y breve descripción de las clases utilizadas.
- Funciones auxiliares: resumen de las funciones auxiliares que se hayan codificado.
- Modo de obtención del intérprete: nombre y descripción de cada directorio y fichero.
- Modo de ejecución del intérprete: formas y usos de los modos de ejecución del trabajo
- Ejemplos: resumen y breve descripción de los ejemplos implementados para el trabajo.
- Conclusiones: puntos fuertes y puntos débiles del intérprete desarrollado y reflexión sobre el trabajo.
- Bibliografía o referencias web.
- Anexos.

El resultado de todo lo nombrado anteriormente debe ser un interprete capaz de entender, comprobar y ejecutar pseudocódigos sencillos de forma precisa, documentando los errores que se cometen y notificando los mismos.

2 Lenguaje de pseudocódigo

2.1 Componentes léxicos o tokens

2.1.1 Palabras reservadas

Sentencias de control

- read, read_string
- print
- if, then, else, end_if
- while, do, end_while
- repeat, until, for, end_for, from, step, to
- switch, case, default, end_switch

Manejo de la pantalla

- clear_screen, place

Funciones predefinidas

- sin, cos, sqrt, log, log10, exp, sqrt, integer, abs.

Constantes predefinidas

- pi, e, gamma, phi, deg.

Operador matemático

- mod.

Constantes y operadores lógicos

- true, false.
- or, and, not.

Observaciones:

- No se distinguirá entre mayúsculas ni minúsculas.
- Las palabras reservadas no se podrán utilizar como identificadores.

2.1.2 Identificadores

Características:

- Estarán compuestos por una serie de letras, dígitos y el símbolo de subrayado "_".
- Deben comenzar por una letra.
- No se distinguirá entre mayúsculas ni minúsculas.
- No podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.

Identificadores válidos

- dato, dato_1, dato_1_a

Identificadores no válidos

- _dato, dato_, dato__1.

2.1.3 Número

- Se utilizarán números enteros, reales de punto fijo y reales con notación científica.
- Todos ellos serán tratados conjuntamente como números.

2.1.4 Cadena

- Estará compuesta por una serie de caracteres delimitados por comillas simples.
- 'Ejemplo de cadena'.
- Deberá permitir la inclusión de la comillas simples, el salto de línea y la tabulación. (\', \n, \t).

2.1.5 Operadores

Operador de asignación

- asignación: :=

Operadores aritméticos

- Suma: + (unario: + 2 y binario: 2 + 3)
- Resta: - (unario: - 2 y binario: 2 - 3)
- Producto: *
- División: /
- División entera: //

- Módulo: mod
- Potencia: ^

Operador alfanumérico

- Concatenación: ||

Operadores relacionales de números y cadenas

- Menor que: <
- Menor o igual que: <=
- Mayor que: >
- Mayor o igual: >=
- Igual que: =
- Distinto que: <>

Operadores lógicos

- Disyunción lógica: or
- Conjunción lógica: and
- Negación lógica: not

2.1.6 Comentarios

- De varias líneas: delimitados por el símbolos (* y *)
- De una línea: Todo lo que siga al carácter # hasta el final de la línea.

2.1.7 Fin de sentencia

- Punto y coma: ; (Se utilizará para indicar el fin de una sentencia).

2.2 Componentes léxicos

2.3 Sentencias

Asignación

- identificador := expresión numérica
  - Declara a identificador como una variable numérica y le asigna el valor de expresión numérica.
  - Las expresiones numéricas se formarán con números, variables numéricas y operadores numéricos.

- identificador := expresión alfanumérica
  - Declara a identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.
  - Las expresiones alfanuméricas se formarán con cadenas, variables alfanuméricas y el operador alfanumérico de concatenación (`||`).

Lectura

- read (identificador)
  - Declara a identificador como variable numérica y le asigna el número leído.
- read_string (identificador)
  - Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).

Escritura

- print (expresión numérica)
  - El valor de la expresión numérica es escrito en la pantalla.
- print (expresión lógica)
  - El valor de la expresión lógica es escrito en la pantalla.
- print (expresión alfanumérica)
  - La cadena (sin comillas exteriores) es escrita en la pantalla.
  - Se debe permitir la interpretación de comandos de saltos de línea (`\n`) y tabuladores (`\t`) que puedan aparecer en la expresión alfanumérica.
  - `print('\t Introduzca el dato \n');`
  - `Introduzca el dato`

Sentencias de control

- Sentencia condicional simple
  - `if` condición
    `then` lista de sentencias
    `end_if`
- Sentencia condicional compuesta
  - `if` condición
    `then` lista de sentencias
    `else` lista de sentencias
    `end_if`

- Bucle `while`
  - `while` condición `do`
    lista de sentencias
    `end_while`
- Bucle `repeat`
  - `repeat`
    lista de sentencias
    `until` condición
- Bucle `for`
  - `for` identificador
    `from` expresión numérica 1
    `to` expresión numérica 2
    `[step]` expresión numérica 3
    `do`
    lista de sentencias
    `end_for`
  - **Notas:**
    - El paso (`step`) es opcional; en su defecto, tomará el valor 1.
    - Se debe controlar que el bucle esté bien definido, es decir, que el intervalo de valores del identificador no sea nulo o infinito.
- Sentencia `switch`
  - `switch (expresión)`
    `case` expresión 1: lista de sentencias
    `case` expresión 2: lista de sentencias
    `...`
    `[default:` lista de sentencias`]`
    `end_switch`

Nota al pie: Una condición será una expresión relacional o una expresión lógica compuesta.

Comandos especiales

- `clear_screen`
  - Borra la pantalla.
- `place(expresión numérica1, expresión numérica2)`
  - Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

3 Tabla de símbolos

En esta sección se recogen las clases pertenecientes a la creación de la tabla de símbolos:

Clase	Descripción
Builtin	Definition of atributes and methods of Builtin class
Keyword	Definition of atributes and methods of Keyword class
LogicalVariable	Definition of atributes and methods of LogicalVariable class
NumericVariable	Definition of atributes and methods of NumericVariable class
StringVariable	Definition of a string variable
Symbol	Definition of the class Symbol
SymbolInterface	Prototype of the pure virtual methods
Table	Definition of atributes and methods of Table class
TableInterface	Specificatons of the pure virtual methods
Variable	Definition of atributes and methods of Variable class

4 Análisis léxico

El análisis léxico es la primera fase del procesamiento de lenguajes. Su objetivo es leer la entrada fuente carácter a carácter, agrupar los símbolos en lexemas y producir una secuencia de *tokens* que será consumida por el analizador sintáctico. En este proyecto se ha utilizado **Flex** para generar automáticamente el analizador léxico a partir de expresiones regulares.

4.1 Descripción del fichero `interpreter.l`

El fichero `interpreter.l` contiene la especificación del analizador léxico. Su estructura se divide en varias partes:

- **Cabecera en C++ (`%...%`):** Incluye archivos necesarios como `interpreter.tab.h`, definiciones para el AST, la tabla de símbolos, y la gestión de errores. También se define la función auxiliar `toLower` para convertir identificadores a minúsculas.
- **Definiciones léxicas:** Se definen macros para expresar de forma clara patrones regulares como:
  - `DIGIT`, `LETTER`, `IDENTIFIER` para dígitos, letras e identificadores.
  - `NUMBER1`, `NUMBER2`, `NUMBER3` y `EXPONENT` para reconocer números reales y en notación científica.
  - `CADENA` para representar cadenas de caracteres entre comillas simples.
- **Estados léxicos:** Se definen los estados `COMMENT` y `ERROR` para gestionar comentarios multilínea y errores léxicos respectivamente.
- **Reglas léxicas:** Cada regla consta de una expresión regular y una acción asociada:
  - Se ignoran espacios, tabulaciones y líneas de comentario (`#`).
  - Los saltos de línea incrementan el contador `lineNumber`.

  - Los identificadores se convierten a minúsculas y se validan según ciertas restricciones léxicas (por ejemplo, no pueden comenzar o terminar en _ ni contener __).
  - Se reconocen palabras clave como `mod`, `not`, `or` y `and`.
  - Los números se transforman a tipo `double` mediante `atof` o `strtod` y se asocian al token `NUMBER`.
  - Las cadenas se procesan eliminando las comillas y gestionando caracteres escapados (`\n`, `\t`, etc.).
  - Los operadores (`+`, `-`, `:=`, `>=`, `||`, etc.) se vinculan con los tokens sintácticos correspondientes.
  - Los comentarios delimitados por `(*` y `*)` se ignoran por completo.
  - Los errores léxicos se capturan en el estado `ERROR`, acumulando los caracteres inválidos y generando una advertencia mediante la función `warning`.

- **Fin de fichero:** Cuando se detecta el final del archivo (≪`EOF`≫), se retorna `0` para indicar el término del análisis.

El fichero `interpreter.l` implementa un analizador léxico basado en autómatas finitos deterministas (AFD), ya que las expresiones regulares utilizadas definen lenguajes regulares. Este componente actúa como interfaz entre el código fuente y el análisis sintáctico, facilitando la conversión del texto en una secuencia de símbolos reconocibles por el intérprete.

5 Análisis sintáctico

5.1 Descripción general

El análisis sintáctico ha sido implementado con la herramienta `Bison`, mediante una gramática libre de contexto que define la estructura del lenguaje interpretado. El fichero `interpreter.y` contiene la definición de los símbolos terminales y no terminales, las reglas de producción y las acciones semánticas asociadas.

5.2 Símbolos de la gramática

5.2.1 Símbolos terminales

Los símbolos terminales son los componentes léxicos definidos en el analizador léxico. Entre los principales se incluyen:

- **Palabras clave:** `IF`, `THEN`, `ELSE`, `ENDIF`, `WHILE`, `DO`, `ENDWHILE`, `REPEAT`, `UNTIL`, `FOR`, `FROM`, `TO`, `STEP`, `ENDFOR`, `SWITCH`, `CASE`, `DEFAULT`, `ENDSWITCH`, `READ`, `READ_STR`, `PRINT`, `CLEAR`, `PLACE`.

- **Operadores:** `PLUS`, `MINUS`, `MULTIPLICATION`, `DIVISION`, `MODULO`, `FLOOR_DIVISION`, `POWER`, `EQUAL`, `NOT_EQUAL`, `LESS_THAN`, `LESS_OR_EQUAL`, `GREATER_THAN`, `GREATER_OR_EQUAL`, `AND`, `OR`, `NOT`, `CONCATENATION`, `ASSIGNMENT`.

- **Símbolos de puntuación:** `SEMICOLON`, `COLON`, `COMMA`, `LPAREN`, `RPAREN`, `LETFCURLYBRACKET`, `RIGHTCURLYBRACKET`.

- **Tipos de datos:** `NUMBER`, `STRING`, `BOOL`, `VARIABLE`, `CONSTANT`, `BUILTIN`, `UNDEFINED`.

5.2.2 Símbolos no terminales

Entre los símbolos no terminales definidos en la gramática destacan:

- `program`, `stmtlist`, `stmt`, `exp`, `cond`, `asgn`, `print`, `read`, `block`, `repeat`, `for`, `switch`, `case`, `switchlist`, `listOfExp`, `restOfListOfExp`.

5.3 Reglas de producción

Las producciones parten del símbolo inicial `program`, que representa el conjunto de sentencias de un programa:

```
program : stmtlist { root = new lp::AST($1); }

stmtlist : /* vacío */
         | stmtlist stmt
         | stmtlist error

stmt : SEMICOLON
     | asgn SEMICOLON
     | print SEMICOLON
     | read SEMICOLON
     | for
     | switch
     | repeat
```

5.4 Acciones semánticas

Cada producción está asociada a una acción semántica escrita en C++, que construye los nodos del árbol de sintaxis abstracta (AST). A continuación se destacan las más relevantes para las sentencias de control.

a) Sentencia `if`

```
if: IF controlSymbol cond THEN stmtlist ENDIF {
     $$ = new lp::IfStmt($3, new lp::BlockStmt($5));
}
| IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF {
     $$ = new lp::IfStmt($3, new lp::BlockStmt($5), new lp::BlockStmt($7));
}
```

Crea nodos `IfStmt` que encapsulan una condición y uno o dos bloques de sentencias.

b) Bucle `while`

```
while: WHILE controlSymbol cond DO stmtlist ENDWHILE {
     $$ = new lp::WhileStmt($3, new lp::BlockStmt($5));
}
```

Representa un bucle de condición previa.

c) Bucle `repeat-until`

```
repeat: REPEAT stmtlist UNTIL controlSymbol cond SEMICOLON {
     $$ = new lp::RepeatStmt(new lp::BlockStmt($2), $5);
}
```

Representa un bucle de condición posterior.

d) Bucle `for`

```
for: FOR VARIABLE FROM exp TO exp STEP exp DO stmtlist ENDFOR {
     $$ = new lp::ForStmt($2, $4, $6, new lp::BlockStmt($10), $8);
}
| FOR VARIABLE FROM exp TO exp DO stmtlist ENDFOR {
     $$ = new lp::ForStmt($2, $4, $6, new lp::BlockStmt($8), NULL);
}
```

El paso es opcional. Se representa con la clase `ForStmt`, que gestiona el identificador, los extremos del rango, el paso (si existe) y el bloque de instrucciones.

e) Sentencia `switch`

```
switch: SWITCH controlSymbol LPAREN exp RPAREN switchlist ENDSWITCH {
     $$ = new lp::CaseBlockStmt($4, $6, NULL);
}
| SWITCH ... DEFAULT COLON stmtlist ENDSWITCH {
     $$ = new lp::CaseBlockStmt($4, $6, new lp::BlockStmt($9));
}

case: CASE controlSymbol exp COLON stmtlist {
     $$ = new lp::SwitchStmt($3, new lp::BlockStmt($5));
}
```

Cada `case` se encapsula en un nodo `SwitchStmt` y se agrupa en un `CaseBlockStmt`. El bloque por defecto es opcional.

5.5 Apoyo gráfico

A continuación se presentan diagramas generados automáticamente con `Doxygen`, que muestran la estructura jerárquica de las clases asociadas al análisis sintáctico y al árbol de sintaxis abstracta (AST), así como las relaciones entre las sentencias de control.

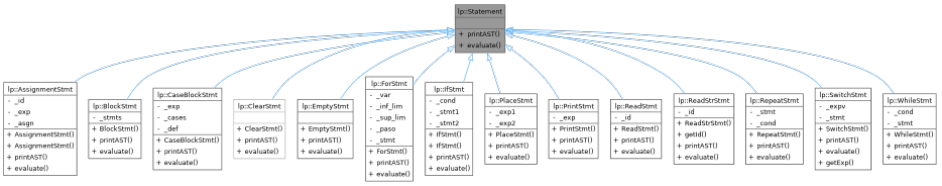

Figura 1: Jerarquía de clases que heredan de `Statement`.

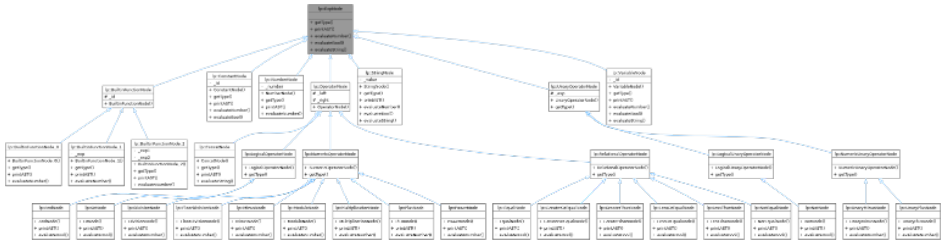

Figura 2: Jerarquía de nodos de expresión derivados de `ExpNode`.

6 Código de AST

En esta sección se recogen las clases pertenecientes a la creación de la tabla de símbolos:

Clase	Descripción
AndNode	Definition of atributes and methods of AndNode class
AssignmentStmt	Definition of atributes and methods of AssignmentStmt class
AST	Definition of atributes and methods of AST class
BlockStmt	Definition of atributes and methods of BlockStmt class
BuiltinFunctionNode	Definition of atributes and methods of BuiltinFunctionNode class
BuiltinFunctionNode_0	Definition of atributes and methods of BuiltinFunctionNode_0 class
BuiltinFunctionNode_1	Definition of atributes and methods of BuiltinFunctionNode_1 class
BuiltinFunctionNode_2	Definition of atributes and methods of BuiltinFunctionNode_2 class
BuiltinParameter0	Definition of atributes and methods of BuiltinParameter0 class
BuiltinParameter1	Definition of atributes and methods of BuiltinParameter1 class
BuiltinParameter2	Definition of atributes and methods of BuiltinParameter2 class
CaseBlockStmt	Definition of attributes and methods of CaseBlockStmt class
ClearStmt	
ConcatNode	Nodo para representar la concatenación de dos cadenas
Constant	Definition of atributes and methods of Constant class
ConstantNode	Definition of atributes and methods of ConstantNode class
DivisionNode	Definition of atributes and methods of DivisionNode class
EmptyStmt	Definition of atributes and methods of EmptyStmt class
EqualNode	Definition of atributes and methods of EqualNode class
ExpNode	Definition of atributes and methods of ExpNode class
FloorDivisionNode	
ForStmt	Definition of atributes and methods of ForStmt class
GreaterOrEqualNode	Definition of atributes and methods of GreaterOrEqualNode class
GreaterThanNode	Definition of atributes and methods of GreaterThanNode class

Clase	Descripción
IfStmt	Definition of atributes and methods of IfStmt class
LessOrEqualNode	Definition of atributes and methods of LessOrEqualNode class
LessThanNode	Definition of atributes and methods of LessThanNode class
LogicalConstant	Definition of atributes and methods of LogicalConstant class
LogicalOperatorNode	Definition of atributes and methods of LogicalOperatorNode class
LogicalUnaryOperatorNode	Definition of atributes and methods of UnaryOperatorNode class
MinusNode	Definition of atributes and methods of MinusNode class
ModuloNode	Definition of atributes and methods of ModuloNode class
MultiplicationNode	Definition of atributes and methods of MultiplicationNode class
NotEqualNode	Definition of atributes and methods of NotEqualNode class
NotNode	Definition of atributes and methods of UnaryPlusNode class
NumberNode	Definition of atributes and methods of NumberNode class
NumericConstant	Definition of atributes and methods of NumericConstant class
NumericOperatorNode	Definition of atributes and methods of NumericOperatorNode class
NumericUnaryOperatorNode	Definition of atributes and methods of UnaryOperatorNode class
OperatorNode	Definition of atributes and methods of OperatorNode class
OrNode	Definition of atributes and methods of OrNode class
PlaceStmt	Definition of attributes and methods of PlaceStmt class
PlusNode	Definition of atributes and methods of PlusNode class
PowerNode	Definition of atributes and methods of PowerNode class
PrintStmt	Definition of atributes and methods of PrintStmt class
ReadStmt	Definition of atributes and methods of ReadStmt class
ReadStrStmt	Class to represent a read string statement
RelationalOperatorNode	Definition of atributes and methods of RelationalOperatorNode class
RepeatStmt	Definition of atributes and methods of RepeatStmt class
Statement	Definition of atributes and methods of Statement class
StringNode	Definition of attributes and methods of StringNode class
SwitchStmt	Definition of attributes and methods of SwitchStmt class
UnaryMinusNode	Definition of atributes and methods of UnaryMinusNode class
UnaryOperatorNode	Definition of atributes and methods of UnaryOperatorNode class
UnaryPlusNode	Definition of atributes and methods of UnaryPlusNode class
VariableNode	Definition of atributes and methods of VariableNode class
WhileStmt	Definition of atributes and methods of WhileStmt class

7 Funciones auxiliares

En el desarrollo del procesador de lenguajes, se han codificado diversas funciones auxiliares que permiten extender la funcionalidad del lenguaje con operaciones predefinidas. Estas funciones no forman parte directa del núcleo del lenguaje definido por el usuario, sino que se ofrecen como utilidades ya implementadas. A continuación, se resumen y clasifican:

Funciones matemáticas

Estas funciones permiten realizar operaciones numéricas avanzadas o conversiones útiles:

- `sin(x)` – Calcula el seno de `x`.
- `cos(x)` – Calcula el coseno de `x`.
- `atan(x)` – Calcula la arcotangente de `x`.
- `atan2(y, x)` – Devuelve el ángulo cuya tangente es `y/x`, considerando el cuadrante.
- `log(x)` – Logaritmo neperiano de `x`.
- `log10(x)` – Logaritmo en base 10 de `x`.
- `exp(x)` – Devuelve e^x, la exponencial de `x`.
- `sqrt(x)` – Raíz cuadrada de `x`.
- `integer(x)` – Convierte `x` a su parte entera (truncamiento).
- `abs(x)` – Valor absoluto de `x`.

Estas funciones están disponibles como funciones *built-in* dentro del lenguaje, y su comportamiento sigue el estándar de las funciones matemáticas en C++ o bibliotecas similares.

Funciones alfanuméricas

Permiten realizar operaciones de entrada/salida y manipulación básica de cadenas:

- `print(x)` – Muestra por pantalla el valor de `x`, sea numérico, lógico o cadena.
- `read_string(var)` – Lee una cadena desde la entrada estándar y la almacena en una variable alfanumérica especificada.

8 Modo de obtención del intérprete

El intérprete se organiza en varios directorios que agrupan los distintos componentes del sistema. A continuación, se describe el propósito de cada directorio y el contenido de sus ficheros:

Directorios y ficheros

- **ast**
  Contiene las definiciones y métodos asociados al árbol de sintaxis abstracta (AST).
  - `ast.cpp`: Implementación de las funciones de la clase `AST`.
  - `ast.hpp`: Declaración de la clase `AST`.
- **error**
  Módulo para la gestión de errores durante la ejecución del intérprete.
  - `error.cpp`: Implementación de funciones para la recuperación de errores.
  - `error.hpp`: Prototipos de las funciones de gestión de errores.
- **includes**
  Ficheros auxiliares y macros comunes al resto del sistema.
  - `macros.hpp`: Definición de macros para control de pantalla y otros elementos comunes.
- **parser**
  Componentes del análisis léxico y sintáctico.
  - `interpreter.l`: Fichero del analizador léxico (scanner), utilizado por `flex`.
  - `interpreter.y`: Fichero del analizador sintáctico (gramática), utilizado por `bison`.
- **table**
  Módulo de la tabla de símbolos y entidades semánticas del lenguaje.
  - `builtin.cpp / .hpp`: Código y declaración de la clase `Builtin`, que representa funciones predefinidas.
  - `builtinParameter0.cpp / .hpp`: Código y declaración de funciones built-in con 0 parámetros.
  - `builtinParameter1.cpp / .hpp`: Código y declaración de funciones built-in con 1 parámetro.
  - `builtinParameter2.cpp / .hpp`: Código y declaración de funciones built-in con 2 parámetros.
  - `constant.cpp / .hpp`: Código y declaración de la clase `Constant`, para constantes del lenguaje.
  - `init.cpp / .hpp`: Inicialización de la tabla de símbolos con elementos predefinidos.
  - `keyword.cpp / .hpp`: Código y declaración de la clase `Keyword`, que gestiona palabras reservadas.
  - `logicalConstant.cpp / .hpp`: Código y declaración de constantes lógicas.
  - `logicalVariable.cpp / .hpp`: Código y declaración de variables lógicas.
  - `mathFunction.cpp / .hpp`: Funciones matemáticas disponibles como funciones auxiliares.
  - `numericConstant.cpp / .hpp`: Código y declaración de constantes numéricas.

  - `numericVariable.cpp / .hpp`: Código y declaración de variables numéricas.
  - `stringVariable.hpp`: Declaración de variables alfanuméricas (de tipo cadena).
  - `symbol.cpp / .hpp`: Código y declaración de la clase `Symbol`, elemento base de la tabla de símbolos.
  - `symbolInterface.hpp`: Declaración de la clase abstracta `SymbolInterface`.
  - `table.cpp / .hpp`: Código y declaración de la clase `Table`, que representa la tabla de símbolos.
  - `tableInterface.hpp`: Declaración de la clase abstracta `TableInterface`.
  - `variable.cpp / .hpp`: Código y declaración de la clase `Variable`, superclase de todas las variables del lenguaje.

- **Raíz del proyecto**
  - `interpreter.cpp`: Programa principal del intérprete, que integra todas las fases (análisis léxico, sintáctico, semántico y ejecución).

9 Modo de ejecución del intérprete

El intérprete desarrollado permite dos modos de ejecución: modo interactivo y modo desde fichero. A continuación, se describe el comportamiento en cada caso.

Modo interactivo

Si el intérprete se ejecuta sin argumentos en la línea de comandos, entra automáticamente en modo interactivo. En este modo, el usuario puede introducir instrucciones del lenguaje directamente a través del teclado, y el sistema las analiza, interpreta y ejecuta en tiempo real.

Este modo es útil para realizar pruebas rápidas, ejecutar expresiones sencillas o depurar el comportamiento del lenguaje paso a paso.

```
$ ./interpreter.exe
```

Modo desde fichero

El intérprete también puede ejecutar programas escritos previamente en un fichero. Para ello, basta con proporcionar el nombre del fichero como argumento al invocar el programa. El fichero debe tener extensión `.p` y debe existir en el sistema.

```
$ ./interpreter.exe programa.p
```

Si el fichero no existe o no tiene la extensión adecuada, el intérprete mostrará un mensaje de error indicando la causa y finalizará la ejecución.

Este modo es el más apropiado para ejecutar programas completos, previamente preparados y almacenados en disco.

10 Ejemplos

Podemos encontrar dos tipos de ejemplos en el trabajo, Aquellos facilitados por el profesor, para probar el interprete y capacidad de interpretar y su correcta ejecución estos son:

- binario.p: código que transforma número decimales en su representación binaria.
- menu.p: menú con diversas opciones matemáticas.
- conversion.p: se comprueba la conversión de tipo de una variable.
- test_switch.p: código de ejemplo para ver el correcto funcionamiento de switch.

Estos códigos se pueden ejecutar y comprobar en el código principal. Ademas de los facilitados por el profesorado se han implementado otros ejemplos algo mas simples con el fin de comprobar mas funciones implementadas durante la elaboración de este trabajo. Estos son:

- error.txt: fichero con extensión errónea, al intentar ejecutarlo el interprete debe devolver un fallo aclarando la necesidad de que el archivo se ".p".
- es_primo.p: código que comprueba que el número insertado por teclado es primo o no.
- fibonacci.p: código que muestra por pantalla los n primeros números de la sucesión de fibonacci.
- for.p: código de prueba para el bucle for.
- if_while.p: código que prueba un if y un while para demostrar su correcto funcionamiento.
- repeat.p: código de prueba para el bucle repeat.

Al igual que los anteriores es posible ejecutar todos estos códigos en una ejecución del trabajo ademas se ha facilitado la implementación del pseudocodigo fibonacci.p en esta memoria en el apartado de Anexos 13.

11 Conclusiones

A lo largo de este trabajo se ha desarrollado un intérprete funcional y extensible para un lenguaje de pseudocódigo, partiendo del código base proporcionado por el profesor. El diseño se ha llevado a cabo con una estructura modular, utilizando herramientas estándar como `Flex` para el análisis léxico y `Bison` para el análisis sintáctico, y construyendo un árbol de sintaxis abstracta (AST) detallado y jerárquico.

A continuación, se presentan los principales aspectos positivos y puntos mejorables detectados durante el desarrollo:

Puntos fuertes

- **Ampliación del lenguaje:** Se han incorporado múltiples constructos sintácticos y semánticos como bucles (`for`, `while`, `repeat`), condicionales anidados, operadores lógicos y aritméticos, y comandos especiales.
- **Diseño modular:** La organización del proyecto en directorios independientes facilita la escalabilidad, el mantenimiento y la depuración del sistema.
- **Análisis léxico robusto:** El analizador léxico desarrollado con `Flex` reconoce con precisión identificadores, cadenas, números, operadores y palabras clave, además de incluir validaciones adicionales y gestión de errores léxicos.
- **Gramática sintáctica completa:** El analizador sintáctico, implementado con `Bison`, incluye una gramática libre de contexto capaz de interpretar correctamente las estructuras del lenguaje, junto con acciones semánticas que generan el AST.
- **Estructura del AST coherente:** Las clases del árbol de sintaxis abstracta representan fielmente la jerarquía de expresiones y sentencias, y están preparadas para facilitar la ejecución o futura compilación del código.
- **Funciones y constantes predefinidas:** El lenguaje incluye funciones matemáticas útiles (`sqrt`, `log`, `abs`, ...) y constantes habituales (`pi`, `e`, ...), accesibles de forma directa por el usuario.
- **Versatilidad en la ejecución:** El intérprete permite tanto la ejecución interactiva como la ejecución de programas almacenados en ficheros, lo cual lo hace adecuado para distintos contextos de uso y pruebas.
- **Validación mediante ejemplos:** Se han proporcionado varios ejemplos de uso (como `binario.p`, `conversion.p`, ...) que demuestran la correcta ejecución del intérprete y su aplicabilidad práctica.

Puntos débiles y aspectos mejorables

- **Análisis semántico limitado:** No se han implementado comprobaciones exhaustivas de tipos o errores semánticos como el uso de variables no declaradas, lo cual puede afectar la robustez en casos complejos.
- **Ausencia de funciones definidas por el usuario:** Actualmente, el lenguaje no permite la definición de funciones por parte del usuario, lo que restringe la modularidad y la reutilización de código.
- **Falta de estructuras de datos avanzadas:** No se contempla el uso de arrays, listas u otras estructuras dinámicas, lo que podría ser una extensión interesante para enriquecer el lenguaje.
- **Gestión de errores en tiempo de ejecución:** Aunque existen mecanismos para advertencias y errores léxicos, se podría ampliar la retroalimentación ofrecida al usuario durante la ejecución con mensajes más detallados y localización precisa de errores.

Reflexión final

El proyecto desarrollado constituye una base sólida para la construcción de un lenguaje interpretado. La implementación ha permitido aplicar conocimientos teóricos de análisis léxico, sintáctico y semántico, así como afianzar conceptos de orientación a objetos en C++. A pesar de las limitaciones señaladas, el intérprete resultante es completamente funcional y está preparado para ser ampliado en futuras versiones.

12 Bibliografía

- Aho, A. V., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, and tools (2nd ed.). Addison-Wesley.
- Appel, A. W. (1998). Modern compiler implementation in C. Cambridge University Press.
- Nystrom, R. (2018). Crafting interpreters. Retrieved June 23, 2025, from `https://craftinginterpreters.com/`
- Cooper, K. D., & Torczon, L. (2011). Engineering a compiler (2nd ed.). Morgan Kaufmann.
- Crenshaw, J. W. (1998). Let's build a compiler. Retrieved June 23, 2025, from `http://compilers.iecc.com/crenshaw/`
- Tang, J. (2012). Write yourself a Scheme in 48 hours. Retrieved June 23, 2025, from `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`

13 Anexos

13.1 Códigos de ejemplo implementados por los alumnos: fibonacci.p

```
print(‘Inserte el cuantos digitos de la sucesion quiere obtener’);

read(n);

sigiente := 1;
anterior := 0;
actual := 0;
i := 0;

for i from 1 to n do
    actual := sigiente + anterior;
    print(actual);
    anterior := sigiente;
    sigiente := actual;
end_for

print(‘Final del programa’);
```