

PL2 - Memoria

Rubén Barroso, Marcos Fuster, Cristian Márquez

Primera parte: árbol sintáctico para un lenguaje limitado (la búsqueda del tesoro)

Segunda parte: jugando a la búsqueda del tesoro

Tercera parte: árbol sintáctico para un lenguaje simple (MiniB)

Primera parte: árbol sintáctico para un lenguaje limitado (la búsqueda del tesoro)

Implementación base

Tomando como ejemplo el archivo ejemplo.field:

```
"Mapa de pruebas"
"El Titanic" te da 50 puntos
"El Santa Maria" te da 20 puntos
"La Chatarra" te da 1 puntos
"El Titanic" está enterrado en 3,1
"El Santa Maria" está enterrado en 2,3
"La Chatarra" está enterrado en 3,2
"La Chatarra" está enterrado en 1,4
"La Chatarra" está enterrado en 4,1
```

Podemos diferenciar 3 tipos de enunciados:

- El título, que aparece una única vez
- Los puntuación que aporta cada barco
- Las localización de cada barco, pudiendo haber varios con el mismo nombre

Vemos que las cadenas de las que nos interesa extraer información se encuentran entre comillas y que las oraciones del mismo tipo presentan una estructura idéntica, lo cual nos ayuda a poder establecer la gramática de la siguiente forma:

```
grammar MapaTesoro;

// Parser Rules
mapa      : titulo puntos* localizacion* EOF ;
titulo    : STRING NEWLINE ;
puntos    : STRING 'te da' NUMBER 'puntos' NEWLINE ;
localizacion : STRING 'está enterrado en' coordenada NEWLINE ;
coordenada : NUMBER ',' NUMBER;

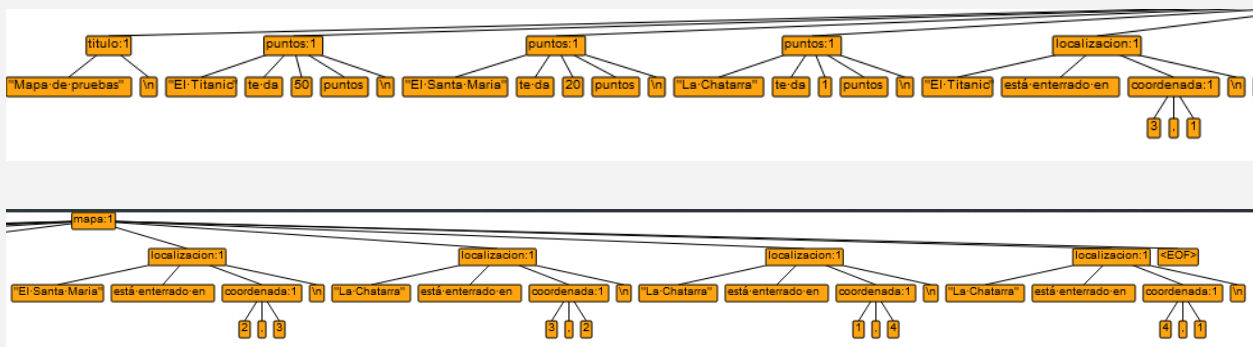
// Lexer Rules
STRING    : '"' (~["\r\n"])* '"' ;
NUMBER    : [0-9]+ ;
NEWLINE   : [\r\n]+ ;
WS        : [ \t]+ -> skip ;
```

Así podemos diferenciar los distintos enunciados que se pretenden interpretar, la definición del valor de cada barco, y la situación en el mapa, de uno o varios barcos de cada tipo.

Con nuestra gramática establecida, y por medio de ANTLR, podemos crear ya el AST de nuestro archivo.




```
$ antlr4-parse Mapa.g4 mapa -gui
<árbol en pestaña nueva>
```

Exportando a svg:



Además, a partir de esta gramática podemos crear las herramientas que necesitaremos para trabajar con el archivo desde nuestro código.

```
$ antlr4 -Dlanguage=Python3 MapaTesoro.g4
```

 MapaTesoroLexer.py	30/10/2024 13:14	Python File	5 KB
 MapaTesoroListener.py	30/10/2024 13:16	Python File	3 KB
 MapaTesoroParser.py	30/10/2024 13:14	Python File	15 KB

Primera mejora

Una mejora que podemos aplicar, tanto a la gramática como al archivo fuente en sí, es la de añadir una **frase que establezca el tamaño** del mapa a construir.

Por ejemplo, podríamos añadir en el archivo, justo después del título, una frase del estilo: "El mapa tiene _ filas y _ columnas"

```
"Mapa de pruebas"
El mapa tiene 5 filas y 5 columnas
"El Titanic" te da 50 puntos
"El Santa Maria" te da 20 puntos
"La Chatarra" te da 1 puntos
"El Titanic" está enterrado en 3,1
"El Santa Maria" está enterrado en 2,3
"La Chatarra" está enterrado en 3,2
"La Chatarra" está enterrado en 1,4
"La Chatarra" está enterrado en 4,1
```

La gramática, por tanto, quedaría de la siguiente manera:

```
grammar MapaTesoro;

// Parser Rules
mapa      : titulo tamaño? puntos* localizacion* EOF ;
tamaño    : STRING 'tiene' NUMBER 'filas' NUMBER 'columnas' NEWLINE ;
titulo    : STRING NEWLINE ;
puntos    : STRING 'te da' NUMBER 'puntos' NEWLINE ;
localizacion : STRING 'está enterrado en' coordenada NEWLINE ;
coordenada : NUMBER ',' NUMBER;

// Lexer Rules
STRING    : '"' (~["\r\n"])* '"' ;
NUMBER    : [0-9]+ ;
NEWLINE   : [\r\n]+ ;
WS        : [ \t]+ -> skip ;
```

Sería algo engorroso contrastar el tamaño aportado con los valores de las coordenadas y su validez. Así que mantendremos el tamaño como una aportación opcional:

El tamaño mínimo del mapa vendrá determinado por los valores *x* e *y* máximos de las coordenadas, y éste se podrá ampliar con la nueva regla, para dispersar los barcos y aumentar la dificultad, así como evitar que siempre haya un objetivo en cada borde, lo cuál ocurre usando el tamaño mínimo necesario.

Segunda mejora

Otra mejora podría ser la de ampliar la gramática para implementar nuevas mecánicas del juego, como por ejemplo, casillas dañinas.

Al igual que determinamos en este archivo la puntuación y la localización de cada barco hundido, podemos definir una serie de obstáculos o trampas que reduzcan la vida del jugador.

```
grammar MapaTesoro;

// Parser Rules
mapa      : titulo tamaño? puntos* localizacion*;
titulo    : STRING NEWLINE ;
tamaño    : 'El mapa tiene' NUMBER 'filas y' NUMBER 'columnas' NEWLINE ;
puntos    : STRING 'te da' NUMBER 'puntos' NEWLINE ;
daño      : STRING 'hace ' NUMBER 'puntos de daño' NEWLINE ;
localizacion : STRING 'está enterrado en' coordenada NEWLINE ;
coordenada : NUMBER ',' NUMBER;

// Lexer Rules
// Mapa de pruebas
STRING    : '"' (~["\r\n"])* '"' ;
NUMBER    : [0-9]+ ;
NEWLINE   : [\r\n]+ ;
WS        : [ \t]+ -> skip ;
```

Con esto hemos añadido una nueva regla para determinar el daño que hace un obstáculo. Seguimos la misma estructura que para definir los puntos de los barcos, de hecho, podemos reutilizar la regla de 'localización' para situar también los obstáculos.

A todo esto, necesitamos una vida que restar, y un jugador que la tenga, así que podemos dividir el archivo en la configuración del mapa y la del jugador tal que:

```
grammar MapaTesoro;

// Parser Rules
nivel      : mapa jugador | jugador mapa;
mapa       : 'Mapa:' NEWLINE titulo tamaño? puntos* daño* localizacion*;
titulo     : STRING NEWLINE ;
tamaño     : 'El mapa tiene' NUMBER 'filas y' NUMBER 'columnas' NEWLINE ;
puntos     : STRING 'te da' NUMBER 'puntos' NEWLINE ;
daño       : STRING 'hace ' NUMBER 'puntos de daño' NEWLINE ;
localizacion : STRING 'está enterrado en' coordenada NEWLINE ;
coordenada : NUMBER ',' NUMBER;

jugador     : 'Jugador:' NEWLINE nombre puntos? vida? ;
vida        : 'Tiene ' NUMBER ' puntos de vida' NEWLINE;

// Lexer Rules
STRING      : '"' (~["\r\n"])* '"' ;
NUMBER      : [0-9]+ ;
NEWLINE     : [\r\n]+ ;
WS          : [ \t\r\n]+ -> skip ;
```

Un ejemplo de uso de esta gramática sería el siguiente:

```
Mapa:
"En medio del Atlántico"
El mapa tiene 4 filas y 4 columnas

"El Titanic" te da 50 puntos
"El Santa Maria" te da 20 puntos
"La Chatarra" te da 1 puntos

"Bomba" hace 10 puntos de daño
"Mina" hace 20 puntos de daño

"El Titanic" está enterrado en 3,1
"El Santa Maria" está enterrado en 2,3
"La Chatarra" está enterrado en 3,2
"La Chatarra" está enterrado en 1,4
"La Chatarra" está enterrado en 4,1
"Bomba" está enterrado en 2,2
"Bomba" está enterrado en 4,2
"Bomba" está enterrado en 1,3
"Mina" está enterrado en 3,3
"Mina" está enterrado en 4,3
```

Con toda esta información, más que un mapa estamos definiendo un nivel propio del juego, que desarrollaremos acto seguido, y nos facilita la creación de distintos niveles, en los que podemos ajustar la dificultad según queramos con los elementos que tenemos a nuestra disposición.

AST en texto plano

Una vez aplicadas las mejoras mencionadas anteriormente, lo siguiente es escribir el AST generado en forma de texto plano, en un archivo de texto.

Para ello, haremos uso de un Listener, más concretamente, podemos implementar el `ParseTreeListener` que proporciona ANTLR.

```
class ToStringTreeListener(ParseTreeListener):
    def __init__(self, parser):
        self.indent = 0
        self.output = ""
        self.parser = parser

    def enterEveryRule(self, ctx):
        rule_name = ctx.__class__.__name__.replace("Context", "")
        self.output += "|" * self.indent + rule_name + "\n"
        self.indent += 1

    def exitEveryRule(self, ctx):
        self.indent -= 1
```



Nuestro listener, que hereda de `ParseTreeListener`, va llevando cuenta de los sangrados del árbol, añadiendo cuando entramos en una nueva rama, y restando al salir de ella.

También tenemos que almacenar la salida al archivo, y necesitamos introducir el parser correspondiente al crear el objeto, para más tarde poder acceder a los nombres simbólicos.

Además, esto lo hace funcional para cualquier parser y gramática, por lo que luego nos servirá a la hora de escribir el AST para el MiniB.


```
def visitTerminal(self, node):
    token_text = node.getText()
    token_type = node.getSymbol().type
    token_name = (
        self.parser.symbolicNames[token_type]
        if token_type < len(self.parser.symbolicNames)
        else str(token_type)
    )

    self.output += "| " * self.indent

    if token_name == "<INVALID>" or token_text == "<EOF>":
        self.output += token_text
    elif token_text == "\n":
        self.output += token_name
    else:
        self.output += f"{token_name}: {token_text}"

    self.output += "\n"
```

Esta función es la que se invoca cada vez que llegamos a un nodo terminal, del cual queremos saber tanto el texto que contiene, como el tipo de nodo.

Para este último, primero lo buscamos en la tabla de símbolos y, si no está, lo intentamos convertir a string.

Después de esto, lo único que hacemos es añadir el nombre y / o el texto del token, según como sea más legible, sin perder información.

Una vez ejecutado un código como el siguiente:

```
def main():
    with open("ejemplo.field") as file:
        mapa = Mapa(file.read())

    with open("mapa.tree", "w") as file:
        file.write(mapa.as_tree())

    print("Tree succesfully written in 'mapa.tree'")
    return 0
```

siendo **Mapa** una clase que, en el constructor, toma una string y la procesa, haciendo uso del lexer y el parser contruidos anteriormente, guardando, de momento,

Mapa		
Titulo STRING: "Mapa de pruebas" NEWLINE	Localizacion STRING: "El Titanic" está enterrado en Coordenada NUMBER: 3 , NUMBER: 1 NEWLINE	
Tamaño El mapa tiene NUMBER: 5 filas y NUMBER: 5 columnas NEWLINE	Localizacion STRING: "El Santa Maria" está enterrado en Coordenada NUMBER: 2 , NUMBER: 3 NEWLINE	
Puntos STRING: "El Titanic" te da NUMBER: 50 puntos NEWLINE	Localizacion STRING: "La Chatarra" está enterrado en Coordenada NUMBER: 3 , NUMBER: 2 NEWLINE	, NUMBER: 4 NEWLINE
Puntos STRING: "El Santa Maria" te da NUMBER: 20 puntos NEWLINE	Localizacion STRING: "La Chatarra" está enterrado en Coordenada NUMBER: 4 , NUMBER: 1 NEWLINE	STRING: "La Chatarra" está enterrado en Coordenada NUMBER: 4 , NUMBER: 1 NEWLINE
Puntos STRING: "La Chatarra" te da NUMBER: 1 puntos NEWLINE	Localizacion STRING: "La Chatarra" está enterrado en Coordenada NUMBER: 1 ,	<EOF>

Segunda Parte: Jugando a la Búsqueda del Tesoro

A continuación, presentaremos el juego de búsqueda de tesoros, seguido de una explicación sobre las mejoras implementadas conforme a lo descrito en el apartado anterior.

Clases

Barco: En primer lugar, crearemos un objeto de tipo "recompensa", al que hemos llamado Barco. Este objeto representará el tesoro que el jugador deberá descubrir. Para definir este objeto, es necesario almacenar la siguiente información:

- **self.nombre:** identificador único con el que nos referiremos al barco.
- **self.puntos:** valor de puntos que el jugador obtendrá al encontrar este tesoro.
- **self.coordenadas:** ubicación específica del barco dentro del mapa.

Obstáculo: Similar al diseño de los barcos, pero el valor a almacenar en vez de los puntos que vale, es el daño que ocasiona a nuestro jugador.

- **self.nombre:** identificador único con el que nos referiremos al obstáculo.
- **self.daño:** puntos de daño que restará a la vida del jugador desenterrar este objeto.
- **self.coordenadas:** ubicación específica del obstáculo dentro del mapa.

Jugador: Contiene los datos pertenecientes al jugador en cuestión, puntos que lleva acumulados a lo largo de la partida, y vida que le queda.

Nivel: Aquí contendremos toda la información extraíble del archivo fuente.

Los componentes definidos en este, queremos guardarlos en un tablero de juego, que podemos llamar "mapa".

Este mapa se representa mediante una matriz bidimensional, en el cual cada celda puede contener un barco, un obstáculo, o estar vacía.

Para estructurar el mapa y gestionar el contenido, hemos ampliado la clase Mapa anterior, ahora llamada **Nivel**, que incluye las siguientes funciones principales:

- **__init__:** Inicializa el nivel parseando el mapa y almacenando datos clave, como el título del mapa, los barcos y otros elementos de juego. Con esta información, crea la matriz que representa el mapa y coloca los barcos en las posiciones designadas.

-
- **try_cord**: Supone la principal interacción que realiza el juego con la clase Nivel. Dadas unas coordenadas de la matriz, devuelve el contenido de esa posición (barco o obstáculo si hay, y en su defecto, **None**).
 - **as_map** y **as_matrix**: Permiten visualizar el mapa en texto plano, mostrando el contenido de cada casilla para referencia o revisión.
 - **as_tree**: Proporciona una representación del AST en texto plano, ya vista en el apartado anterior.
 - **parse_map**: Gestiona todo el proceso de parsear el texto del archivo fuente y extraer los datos, usando todas las herramientas de ANTLR generadas (Lexer, Parser, Listener), para obtener los barcos y obstáculos, así como el AST.

Juego: Haciendo uso de todo lo anterior, podemos construir el juego en sí, componente principal de esta segunda parte.

Para ello crearemos un nivel por cada archivo fuente disponible (3 en nuestro caso), y desarrollaremos la lógica que permita interactuar con el mapa, superar niveles, ganar puntos, perder vida...

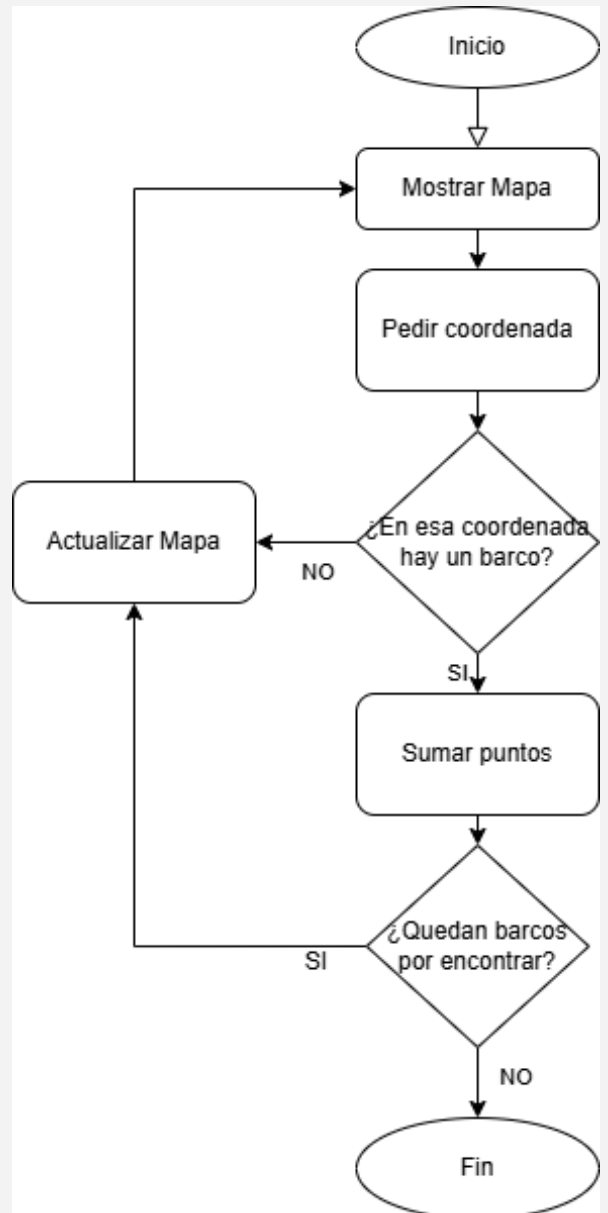
Sin embargo, aunque mantenemos el uso de obstáculos, la vida del jugador no se establecerá desde los archivos fuente como en la mejora expuesta anteriormente.

Vamos a encadenar un nivel con otro, teniendo que sobrevivir el jugador a todos manteniendo la vida inicial sin recibir curas, por lo que la vida que se establezcan en los niveles que sigan al primero no tendrá ningún efecto. Mejor desechamos la idea y la establecemos manualmente dentro del juego, al crear el objeto Jugador.

Interacción con el Usuario

Para permitir la interacción del jugador con el juego, hemos diseñado un bucle de juego que guía al usuario a través de una serie de pasos, modelados en un diagrama de estados. A continuación, se detallan estos pasos fundamentales:

1. **Mostrar la matriz:** En esta etapa, se presenta el mapa al jugador. Dependiendo de las respuestas anteriores (si el jugador ya ha seleccionado algunas casillas), se indicará en el mapa si en las posiciones anteriormente elegidas había un barco, un obstáculo, o estaban vacías.
2. **Pedir coordenadas:** Se solicita al jugador que introduzca las coordenadas de la casilla que desea seleccionar. Este paso recoge el input del usuario y marca el inicio de la búsqueda en el mapa.
3. **Validar coordenada:** La coordenada ingresada por el jugador se verifica para asegurar que sea válida, es decir, que se encuentre dentro de los límites del mapa y que no haya sido seleccionada previamente. Luego, se comprueba si la casilla contiene un barco o no.
4. **Sumar puntos:** Si el jugador acierta y encuentra un barco en la casilla elegida, se suman los puntos correspondientes a su contador de puntuación.
5. **Actualizar mapa:** Tras seleccionar una casilla, el mapa se actualiza para reflejar el estado actual del juego, indicando las casillas que el jugador ya ha seleccionado, ya sea con o sin barco.



El juego concluye una vez que el jugador ha descubierto todos los tesoros, momento en el cual se termina la partida.

Mejoras Propuestas

En este apartado, detallamos las mejoras introducidas en el juego de búsqueda del tesoro, diseñadas para enriquecer la experiencia del jugador. Estas modificaciones abarcan desde ajustes mecánicos hasta la implementación de una interfaz gráfica interactiva basada en Tkinter, una biblioteca estándar de Python. Cada característica contribuye a un entorno de juego más atractivo, dinámico y desafiante. A continuación, se describen estas mejoras, algunas de ellas ya han sido detalladas anteriormente, sin embargo ahora son pertinentes en la implementación del juego de manera detallada:

Mapa de Longitud Variable

La dimensión del mapa ahora es configurable dentro del archivo del nivel, permitiendo variar el tamaño de la matriz que representa el terreno del juego. Esta flexibilidad permite adaptarse tanto a los principiantes como a los jugadores avanzados, brindando una experiencia personalizada.

Sistema de Vida del Jugador

La mecánica del juego incluye un sistema de vidas que penaliza al jugador al encontrar obstáculos en el mapa.

- Funcionamiento:
 - Cada obstáculo encontrado reduce la barra de vida del jugador.
 - Si la vida llega a cero, el jugador pierde la partida.

Esto se representa visualmente mediante una barra de progreso gráfica que disminuye proporcionalmente con cada impacto. Brindando un indicador visual claro y constante del estado del jugador.

Progresión por Niveles

El juego se organiza en tres niveles de dificultad creciente. La vida del jugador se conserva entre niveles, aumentando la tensión y la importancia de cada decisión.

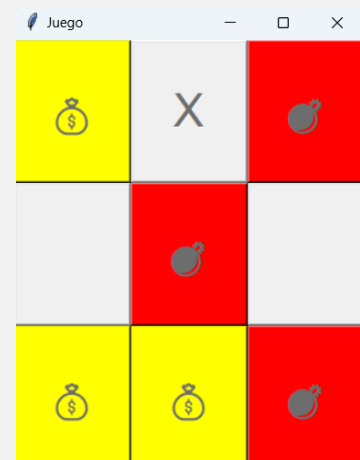
Al completar un nivel, el mapa actual se borra y se genera uno nuevo con mayor número de obstáculos y tesoros. Se notificará al jugador sobre el inicio del nuevo nivel mediante ventanas emergentes.

Interfaz Gráfica Interactiva

Para mejorar la experiencia del usuario, se ha implementado una interfaz gráfica basada en Tkinter, que permite al jugador interactuar con el juego de manera intuitiva mediante botones y notificaciones visuales.

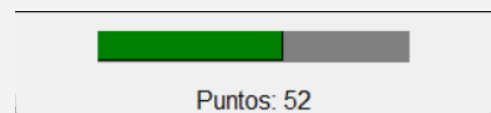
- **Características del Diseño:**

- Un layout en forma de matriz compuesto por botones. Cada botón representa una casilla del mapa, y el jugador puede interactuar con ellas para explorar el terreno.
- Iconografía y Colores:
 - Las casillas con obstáculos y tesoros tienen iconos distintivos.
 - Estas tienen además, colores llamativos que las diferencian claramente del resto.



- **Información del jugador:**

- Se utiliza texto y barras gráficas para informar sobre el progreso del jugador:
 - Puntos acumulados: Muestran los puntos que suman los tesoros encontrados.
 - Barra de vida: Indica visualmente el estado de salud del jugador.



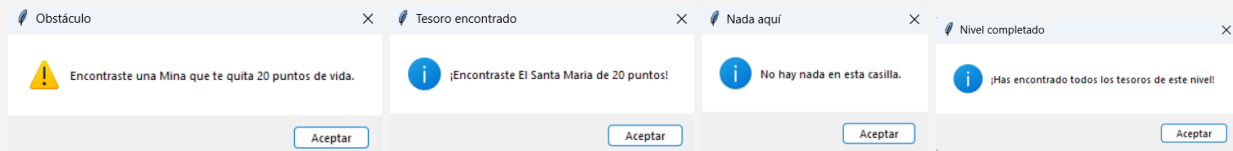
Notificaciones y Mensajes Informativos

El jugador es informado en tiempo real sobre las consecuencias de sus acciones.

- **Mensajes Emergentes:**

- Notificaciones que informan si el jugador ha encontrado un tesoro, un obstáculo o una casilla vacía.
- Mensajes de fin de nivel, victoria o derrota.

Estos mensajes tienen el propósito de brindar retroalimentación inmediata, mejorando el entendimiento del jugador sobre su progreso.



Con estas mejoras, el juego de búsqueda del tesoro no solo se convierte en un reto para los jugadores, sino que también asegura una buena experiencia a través de una interfaz intuitiva. La progresión por niveles, la implementación del sistema de vida y la interacción gráfica directa aportan una mejor mecánica de juego.

Ejecución

Para la ejecución del juego deberemos de correr el archivo de python llamado Juego.py desde la carpeta src del proyecto.

Tercera parte: árbol sintáctico para un lenguaje simple (MiniB)

Implementación base

Teniendo en cuenta los diversos ejemplos sobre el funcionamiento del lenguaje MiniB se ha construido la siguiente gramática:

```
grammar Basic;
// PARSER RULES
program:      (statement NEWLINE+)* statement? EOF;
statement:    letStmt | opStmt | printStmt | inputStmt | ifStmt | forStmt | whileStmt | repeatStmt | keyStmt;
letStmt:      LET ID '=' expression;
opStmt:       ID '=' expression;
printStmt:    PRINT expression;
inputStmt:    INPUT STRING_LITERAL ID;
ifStmt:       IF condition THEN NEWLINE (statement NEWLINE)* (ELSE NEWLINE (statement NEWLINE)*)? END;
forStmt:      FOR ID '=' expression TO expression NEWLINE (statement NEWLINE)* NEXT;
whileStmt:    WHILE condition NEWLINE (statement NEWLINE)* END;
repeatStmt:   REPEAT NEWLINE (statement NEWLINE)* UNTIL condition;
keyStmt:      CONTINUE | EXIT;
condition:    expression comparisonOp expression | expression;
comparisonOp: '<' | '>' | '<=' | '>=' | '=';
expression:   expression op expression | '(' expression ')' | functionCall | NUMBER | STRING_LITERAL | ID;
op:          '+' | '-' | '*' | '/' | MOD;
functionCall : VAL '(' expression ')' | LEN '(' expression ')' | ISNAN '(' expression ')';
```

```
// LEXER RULES
LET : 'LET' | 'let';
PRINT : 'PRINT' | 'print' ;
INPUT : 'INPUT' | 'input' ;
IF : 'IF' | 'if' ;
ELSE : 'ELSE' | 'else' ;
FOR : 'FOR' | 'for' ;
TO : 'TO' | 'to' ;
NEXT : 'NEXT' | 'next' ;
WHILE : 'WHILE' | 'while' ;
REPEAT : 'REPEAT' | 'repeat' ;
UNTIL : 'UNTIL' | 'until' ;
CONTINUE : 'CONTINUE' | 'continue' ;
EXIT : 'EXIT' | 'exit' ;
END : 'END' | 'end' ;
THEN : 'THEN' | 'then' ;
MOD : 'MOD' | 'mod' ;
VAL : 'VAL' | 'val' ;
LEN : 'LEN' | 'len' ;
ISNAN : 'ISNAN' | 'isnan' ;
```

```
ID: [a-zA-Z_][a-zA-Z0-9_]*;
NUMBER: [0-9]+ ('.' [0-9]+)?;
STRING_LITERAL: '"' (~["\r\n])* '"';
NEWLINE: '\r'? '\n';
COMMENT: 'REM' ~[\r\n]* NEWLINE -> skip;
WS: [ \t]+ -> skip;
```

La gramática de MiniB organiza un programa como una secuencia de `statements`, que pueden estar separados por uno o más saltos de línea y finalizan con `EOF`(END OF FILE). Cada `statement` corresponde a un tipo específico de sentencia, como `letStmt` para declaraciones de variables, `opStmt` para asignaciones, o `printStmt` para impresión, entre otros. Las instrucciones de control de flujo (`ifStmt`, `forStmt`, `whileStmt`, y `repeatStmt`) permiten incluir más `statements` anidados en su interior, dando lugar a una estructura de programa jerárquica.

Las expresiones (`expression`) pueden ser operaciones matemáticas, llamadas a funciones (`functionCall`), o valores literales, y soportan paréntesis para anidar o dar prioridad a subexpresiones. MiniB distingue entre diferentes operadores (`op` y `comparisonOp`) para operaciones matemáticas y comparaciones, lo que define el formato permitido en las condiciones (`condition`) de los controles de flujo.

Soporta las estructuras de control de flujo más comunes, como `IF...THEN...ELSE`, `FOR...NEXT`, `WHILE...END` y `REPEAT...UNTIL`, permitiendo usar condiciones y bucles. Además, incluye las instrucciones de entrada y salida `INPUT` y `PRINT`. Para declarar variables se utiliza `LET`, y se pueden asignar valores directamente a través de expresiones simples. También permite realizar operaciones matemáticas con `+`, `-`, `*`, `/` y `MOD`, además de soportar comparaciones (`<`, `>`, `<=`, `>=`, `=`). Además soporta las instrucciones tanto en mayúscula como minúscula.

La gramática incluye funciones como `VAL`, `LEN` e `ISNAN`, que permiten convertir a numérico, obtener la longitud de una cadena o verificar si un valor no es numérico. También permite el uso de comentarios con `REM`, ignorando el resto de la línea. Los espacios y tabulaciones son ignorados, por lo que la sintaxis es flexible y facilita la legibilidad.

Mejoras

Una mejora para avanzar hacia la versión completa de BASIC sería **el soporte para operadores lógicos** (`AND`, `OR`, `NOT`). Esta mejora permite construir condiciones más complejas y cercanas a la funcionalidad típica de BASIC.

Para esta mejora se ha añadido el reconocimiento de `AND`, `OR`, `NOT` tanto en mayúsculas como en minúsculas al igual que en el resto de la gramática. También se

ha añadido una nueva regla para el parser `logicalOp` para manejar los `AND`, `OR` y por último y más importante se ha modificado la regla `condition` para tener en cuenta las nuevas posibilidades.

Gramática actualizada:

```
whileStmt:    WHILE condition NEWLINE (statement
repeatStmt:   REPEAT NEWLINE (statement NEWLIN
keyStmt:      CONTINUE | EXIT;

condition:
| expression comparisonOp expression
| NOT condition
| condition logicalOp condition
| expression;
logicalOp: AND | OR;

comparisonOp: '<' | '>' | '<=' | '>=' | '=';
expression:  expression op expression | '(' e
```

(tan sólo partes que han presentado cambios por simplicidad):

```
// LEXER RULES
NOT : 'NOT' | 'not';
AND: 'AND' | 'and';
OR:  'OR' | 'or';
LET : 'LET' | 'let';
PRINT : 'PRINT' | 'print';
INPUT : 'INPUT' | 'input';
```

AST en texto plano

Una vez aplicadas las mejoras mencionadas anteriormente, lo siguiente es escribir el AST generado en forma de texto plano, en un archivo de texto.

Para ello, se reutiliza lo que ya se hizo para la [Parte 1](#), el `ToStrListener` el cual, como se hizo se uso de funciones generales como `entryEveryRule()` y `visitTerminal()`, se puede aplicar a esta gramática sin ningún tipo de cambio.

De esta forma es sencillo crear el AST en texto plano de cualquiera de los ejemplos.

```

def as_tree(tree):
    """
    Recorre e imprime el mapa en forma de árbol detallado.
    Args:
    |   tree (ParseTree): El árbol de análisis sintáctico a recorrer.
    Returns:
    |   str: La representación del árbol en forma de cadena.
    """
    printer = ToStrTreeListener(BasicParser)
    walker = ParseTreeWalker()
    walker.walk(printer, tree)

    return printer.output


def main(argv):
    """
    Función principal que procesa el archivo de entrada y recorre el árbol de
    análisis sintáctico.

    Args:
    |   argv (list): Lista de argumentos de la línea de comandos. Se espera que
    |               el segundo argumento sea el nombre del archivo de entrada.
    """
    input_stream: FileStream = FileStream(argv[1])
    lexer: BasicLexer = BasicLexer(input_stream)
    stream: CommonTokenStream = CommonTokenStream(lexer)
    parser: BasicParser = BasicParser(stream)
    tree: ParseTree = parser.program()

    with open("basic.tree", "w") as f:
        f.write(as_tree(tree))

```

AST del ejemplo while.bas:

```

Program
| Statement
| | LetStmt
| | | LET: LET
| | | ID: x
| | | =
| | | Expression
| | | NUMBER: 5
| NEWLINE:

| Statement
| | LetStmt
| | | LET: LET
| | | ID: y
| | | =
| | | Expression
| | | NUMBER: 10
| NEWLINE:

| Statement
| | LetStmt
| | | LET: LET
| | | ID: z
| | | =
| | | Expression
| | | NUMBER: 15
| NEWLINE:

| NEWLINE:

| Statement
| | IfStmt
| | | IF: IF
| | | Condition
| | | | Condition
| | | | | Condition
| | | | | Expression
| | | | | ID: x
| | | | | ComparisonOp
| | | | | <
| | | | | Expression
| | | | | ID: y
| | | | LogicalOp

```

```

| | | AND: AND
| | | Condition
| | | | Expression
| | | | | ID: y
| | | | ComparisonOp
| | | | | <
| | | | Expression
| | | | | ID: z
| | | LogicalOp
| | | OR: OR
| | | Condition
| | | | NOT: NOT
| | | | Condition
| | | | | Expression
| | | | | ID: z
| | | | ComparisonOp
| | | | | =
| | | | Expression
| | | | | NUMBER: 10
| | THEN: THEN
| | NEWLINE:

| | Statement
| | | PrintStmt
| | | | PRINT: PRINT
| | | | Expression
| | | | | STRING_LITERAL: "Condition met"
| | NEWLINE:

| | ELSE: ELSE
| | NEWLINE:

| | Statement
| | | PrintStmt
| | | | PRINT: PRINT
| | | | Expression
| | | | | STRING_LITERAL: "Condition not met"
| | NEWLINE:

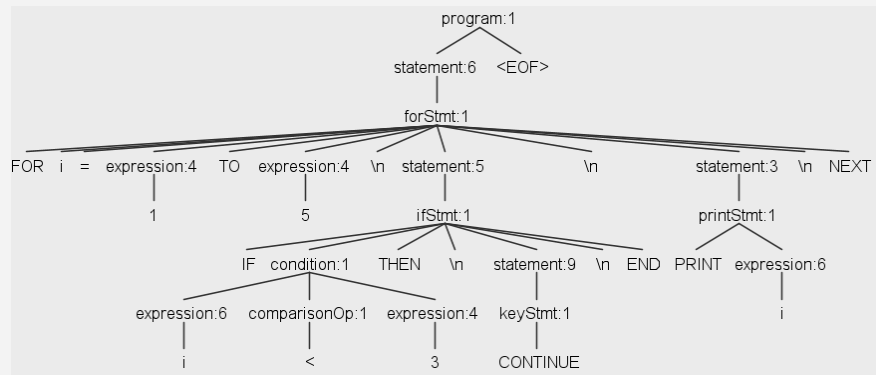
| | END: END
| <EOF>

```

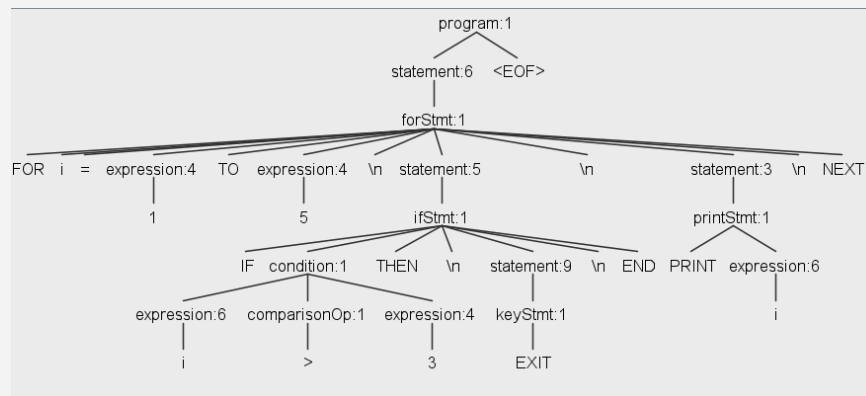
Testing

Para poner a prueba la gramática se comprueba con todos los ejemplos de prueba además de uno añadido para probar las nuevas funcionalidades introducidas en la mejora con **AND**, **OR**, **NOT**.

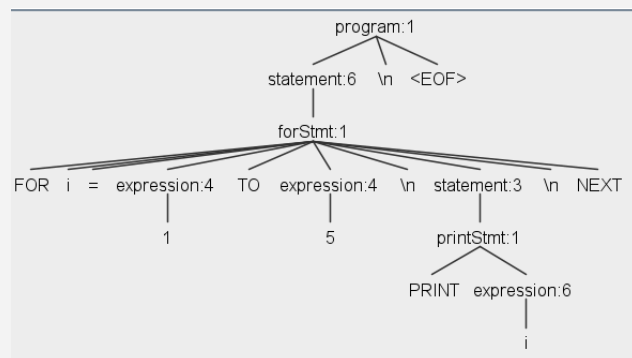
for_continue.bas:



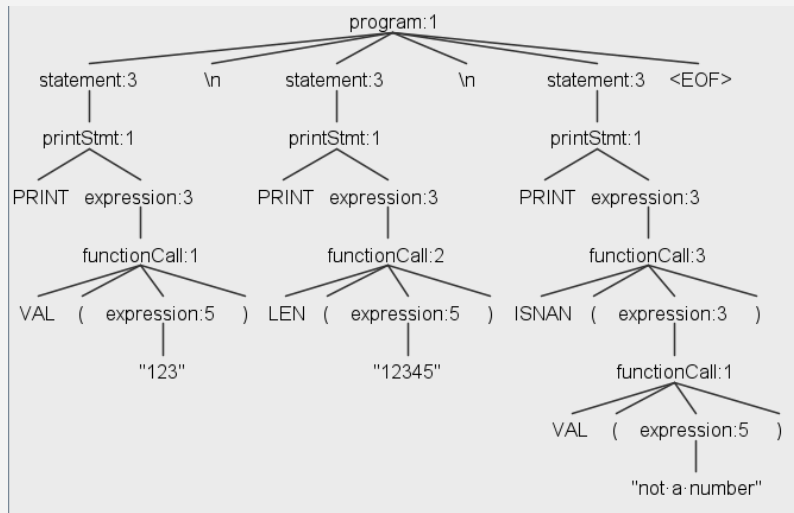
for_exit.bas:



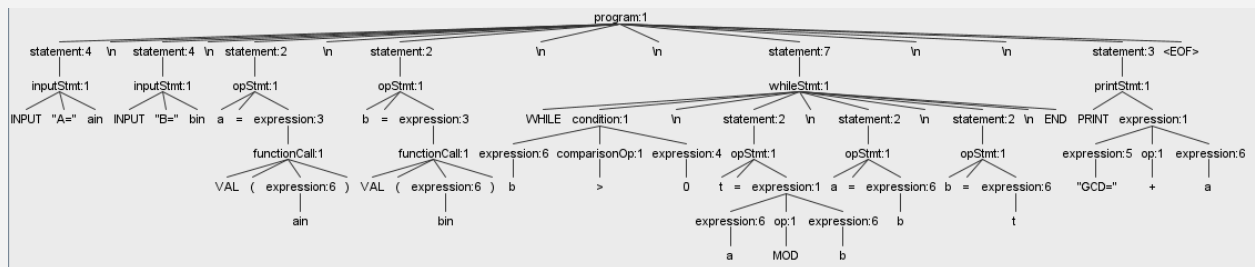
for_simple.bas:



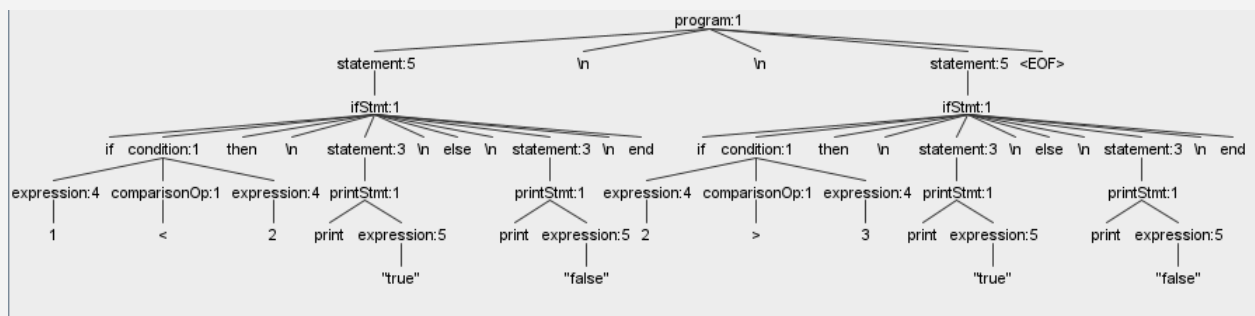
functions.bas:



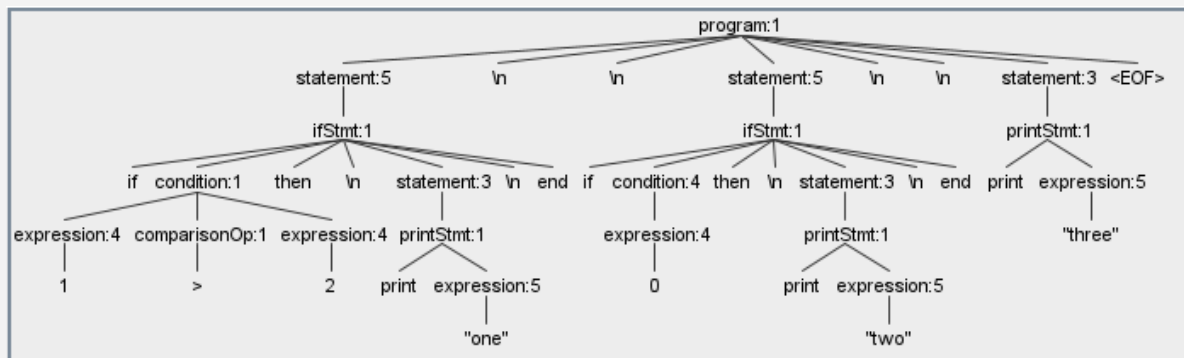
gcd_euclid.bas:



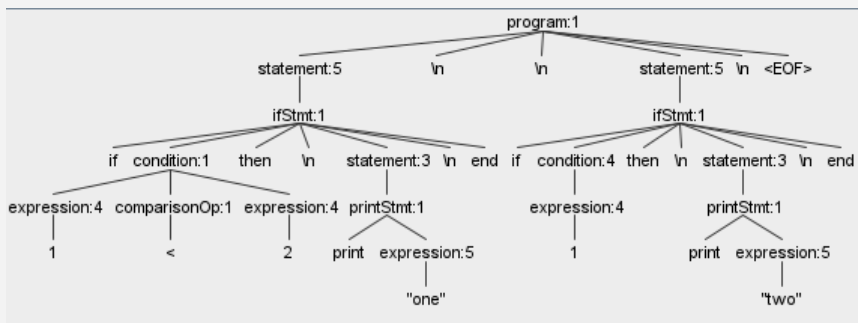
if_else.bas:



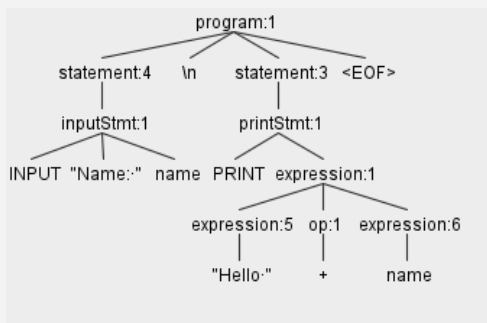
if_simple_false.bas:



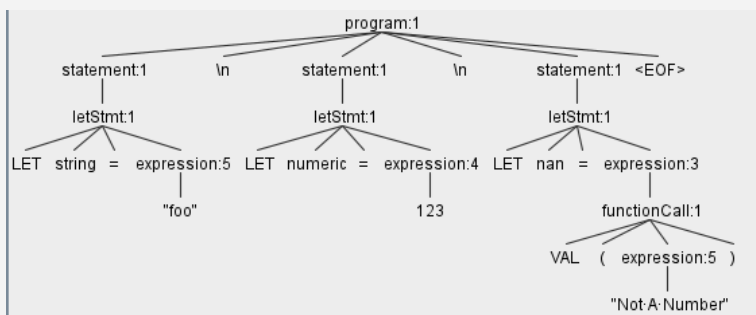
if_simple_true.bas:



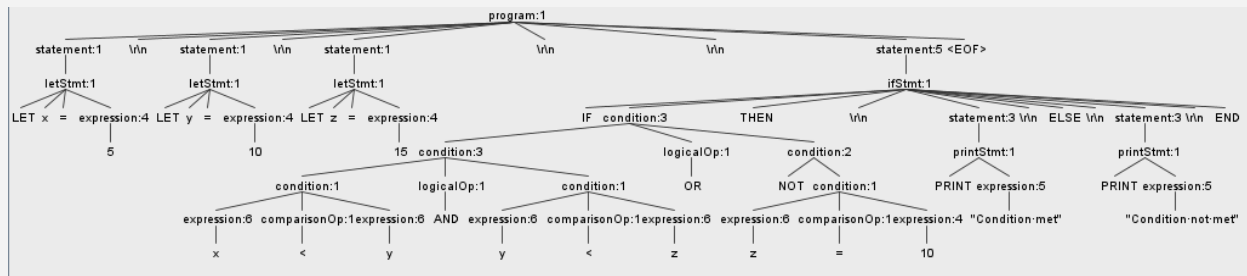
input.bas:



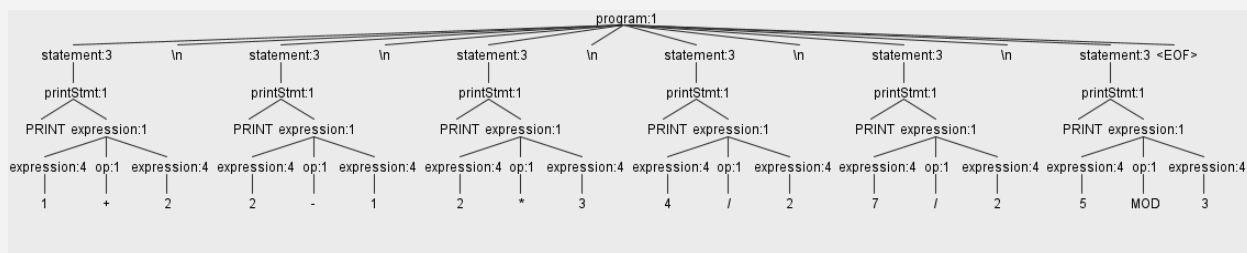
let.bas:



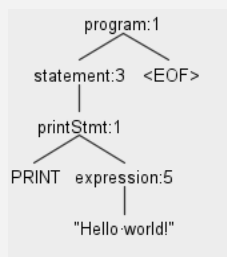
operadores_condicionales.bas:



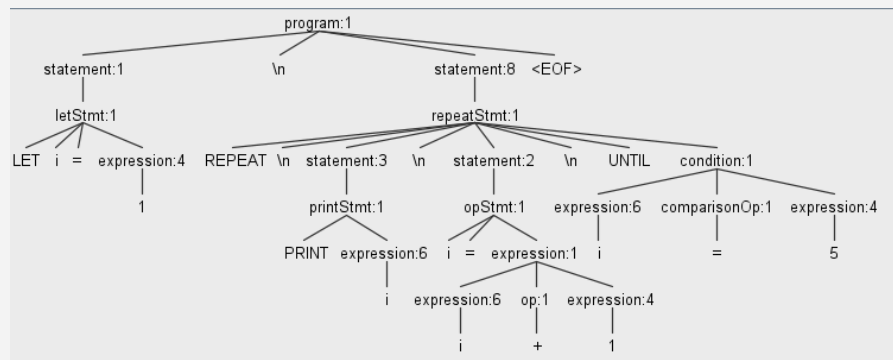
operations.bas:



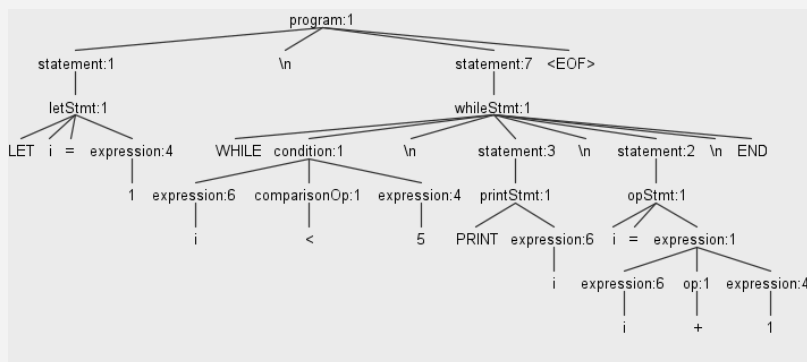
print.bas:



repeat.bas:



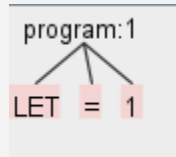
while.bas:



Ahora se comprueba lo que pasa concretamente en la ocurrencia de un error sintáctico y otro de tipo:

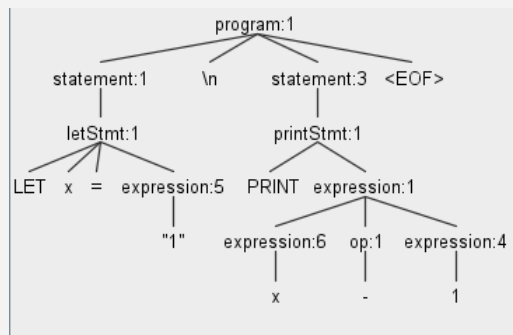
Ejemplos > `error_sintactico.bas`

```
1 LET = 1
```



Ejemplos > `error_tipo.bas`

```
1 LET x = "1"
2 PRINT x - 1
```



```
PS C:\Users\ruben\Desktop\Uni\4\Lenguaje\ProcLenguaje_PL2\ejemplos> python C:\Users\ruben\Desktop\Uni\4\Lenguaje\ProcLenguaje_PL2\src\basic\basicMejorado.py error_tipo.bas
PS C:\Users\ruben\Desktop\Uni\4\Lenguaje\ProcLenguaje_PL2\ejemplos> python C:\Users\ruben\Desktop\Uni\4\Lenguaje\ProcLenguaje_PL2\src\basic\basicMejorado.py error_sintactico.bas
line 1:4 no viable alternative at input 'LET='
```

Nótese como el error de tipo no es detectado mientras que el error sintáctico si lo es, siendo indicado con la frase `line 1:4 no viable alternative at input 'LET='`. Esta nos indica que tras el `LET` nuestra gramática no permite un operador como `=`, espera un `ID` tal y como especificamos en nuestra gramática con

`letStmt: LET ID '=' expression;`

Esto se debe a que no es trabajo del analizador sintáctico encontrar errores sobre el funcionamiento del código en sí mismo, sino tan solo comprobar que la sintaxis funciona de acuerdo a la gramática establecida.

Por último se comprueba el ejemplo añadido para comprobar la nueva funcionalidad. Se obtiene además el AST correspondiente para verificar que la estructura es la correcta:

Ejemplos > `operadores_condicionales.bas`

```
1 LET x = 5
2 LET y = 10
3 LET z = 15
4
5 IF x < y AND y < z OR NOT z = 10 THEN
6     PRINT "Condition met"
7 ELSE
8     PRINT "Condition not met"
9 END
```

Program

```
| Statement
| | LetStmt
| | | LET: LET
| | | ID: x
| | | =
| | | Expression
| | | | NUMBER: 5
| NEWLINE:

| Statement
| | LetStmt
| | | LET: LET
| | | ID: y
| | | =
| | | Expression
| | | | NUMBER: 10
| NEWLINE:

| Statement
| | LetStmt
| | | LET: LET
| | | ID: z
| | | =
| | | Expression
| | | | NUMBER: 15
| NEWLINE:

| NEWLINE:

| Statement
| | IfStmt
| | | IF: IF
| | | Condition
| | | | Condition
| | | | | Condition
| | | | | Expression
| | | | | | ID: x
| | | | | ComparisonOp
| | | | | | <
| | | | | Expression
| | | | | | ID: y
```

```
| | | LogicalOp
| | | | AND: AND
| | | Condition
| | | | Expression
| | | | | ID: y
| | | | ComparisonOp
| | | | | <
| | | | Expression
| | | | | ID: z
| | | LogicalOp
| | | | OR: OR
| | | Condition
| | | | NOT: NOT
| | | | Condition
| | | | | Expression
| | | | | | ID: z
| | | | ComparisonOp
| | | | | =
| | | | Expression
| | | | | NUMBER: 10
| | THEN: THEN
| | NEWLINE:

| | Statement
| | | PrintStmt
| | | | PRINT: PRINT
| | | | Expression
| | | | | STRING_LITERAL: "Condition met"
| | NEWLINE:

| | ELSE: ELSE
| | NEWLINE:

| | Statement
| | | PrintStmt
| | | | PRINT: PRINT
| | | | Expression
| | | | | STRING_LITERAL: "Condition not met"
| | NEWLINE:

| | END: END
| <EOF>
```