

INSTITUTO TECNOLÓGICO DE CHETUMAL

ALUMNO: Palma Sanchez Darwin Giovanni.

Semestre: 9°

DOCENTE: May Canché Isaías.

ASIGNATURA: Lenguajes y Autómatas II.

TAREA: Documentación del compilador al lenguaje LMPN (Lenguaje Mágico Para Niños).



Tabla de contenido


Lenguaje LMPN y su desarrollo.	3
Gramática del lenguaje.	4
Archivos generados por ANTLR4.	7
Compilador del lenguaje: compilador.py.	8
Repositorio del proyecto.	13

Lenguaje LMPN y su desarrollo.

El proyecto del lenguaje LMPN (Lenguaje Mágico Para Niños), fue desarrollado usando Python 3 y ANTLR 4.



Hace del ANTLR4 el cual es una herramienta que viene en forma de .jar:

 antlr-4.13.2-complete.jar

Y el runtime de ANTLR4 el cual se instala desde el pib en Python:

```
1. pip install antlr4-python3-runtime
```

El lenguaje es pensado para que un niño de entre 8 a 10 años pueda usarlo. Esto se consigue al traducir las expresiones más básicas de los lenguajes de programación, aun idioma español y usando palabras que sean más fáciles para ellos. Se busca ser lo más claro y conciso para que al pequeño no le sea difícil poder entender y familiarizarse con el lenguaje. Contempla los fundamentos de un lenguaje de programación y lo más básico para que un niño pueda aprender y usar su creatividad para realizar pequeños programas.

Gramática del lenguaje.

Las reglas de la gramática del lenguaje fueron definidos en el archivo SimpleLang.g4. Dichas reglas son las siguientes:

Establece que el programa inicia con “Empecemos!” y finaliza con “Hasta luego!”:

```
//Regla para el programa
program: 'Empecemos!' statement* 'Hasta luego!';
```

Las sentencias que reconocerá el lenguaje:

```
//Sentencias
statement
: varDeclaration      # DeclaracionVariable
| functionDeclaration # DeclaracionFuncion
| ifStatement         # Condicional
| loopStatement       # Ciclo
| printStatement      # Escribir
| aumentar            # AumentarEnUno
| disminuir           # DisminuirEnUno
| potencia            # Potencias
| raizCuadrada        # RaizCua
| expr ';'            # Expresion
;
```

Como se declaran las variables y funciones:

```
//Declaraciones
varDeclaration: tipo ID '=' expr ';;';
functionDeclaration: 'hacer esto' ID '(' ')' '{' statement* '}';
```

Declaración de la estructura que deberá seguir el condicional if, else y los bucles:

```
//Condicionales y bucles
ifStatement: 'Si pasa esto' '(' expr ')' '{' statement* '}' ('Sino' '{' statement* '}')?;
loopStatement: 'Repetir mientras que' '(' expr ')' '{' statement* '}';
```

Tipos de variables en el lenguaje:

```
//Tipos de variables y expresiones
tipo: 'numero' | 'texto' | 'logico';
```

Estructura de como poder aumentar, disminuir, elevar al cuadrado y raíz cuadrada:

```
//++ y --
aumentar: 'aumentar' '(' ID ')' ' ';';
disminuir: 'disminuir' '(' ID ')' ' ';';

//Potencia y raíz
potencia: 'potencia' '(' ID ')' ' ';';
raizCuadrada: 'raizCuadrada' '(' ID ')' ' ';';
```

Todas las expresiones del lenguaje:

```
expr
: expr 'y que' expr          # And
| expr 'o que' expr          # Or
| expr 'menor que' expr      # MenorQue
| expr 'mayor que' expr      # MayorQue
| expr 'igual que' expr      # IgualQue
| expr 'menor igual a' expr  # MenorIgualQue
| expr 'mayor igual a' expr  # MayorIgualQue
| expr 'diferente de' expr   # DiferenteDe
| '(' expr ')'               # Parentesis
| 'potencia' '(' expr ')'     # PotenciasExpr
| 'raizCuadrada' '(' expr ')' # RaizCuaExpr
| expr '*' expr              # Suma
| expr '/' expr              # Resta
| expr '+' expr              # Multiplicacion
| expr '-' expr              # Division
| ID                          # ID
| INT                         # Int
| STRING                      # String
| BOOL                        # Boolean
| expr ',' expr              # EscribirDos
;
```

Están todos los comparativos, la potencia/raíz y las demás expresiones importantes del lenguaje como el uso de comas o paréntesis.

La estructura para imprimir en consola y como hacer comentarios:

```
//Para imprimir
printStatement: 'escribir' '(' expr ')' ';';

//Comentarios
LINE_COMMENT: '//' ~[\r\n]* -> skip;
```

Los tokens más básicos:

```
//Tokens
ID: [a-zA-Z_][a-zA-Z0-9_]*;
INT: [0-9]+;
STRING: '"' .*? '"';
BOOL: 'verdadero' | 'falso';
WS: [ \t\r\n]+ -> skip;
```

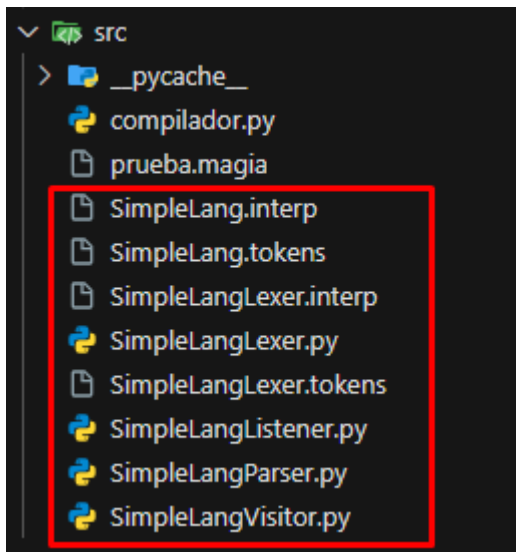
El id correspondiente al nombre de una variable, los tipos de datos: INT, STRING y BOOL, además de los espacios en blanco (White Space).

Archivos generados por ANTLR4.

Al usar los comandos:

```
1. java -jar C:\antlr\antlr-4.13.2-complete.jar -Dlanguage=Python3 -o src
   grammar/SimpleLang.g4
2. java -jar C:\antlr\antlr-4.13.2-complete.jar -Dlanguage=Python3 -visitor -o src
   grammar/SimpleLang.g4
```

Te genera los siguientes archivos (excluyendo el compilador.py y el programa de prueba.magia):



Dichos archivos son generados en automático por el ANTLR4 para poder hacer uso de la herramienta del Lexer, Parser y el Visitor en el compilador.

Compilador del lenguaje: compilador.py.

Primero que todo se debe importar el antlr4 y las clases que generó:

```
from antlr4 import *

#Clases generadas por ANTLR4
from SimpleLangLexer import SimpleLangLexer
from SimpleLangParser import SimpleLangParser
from SimpleLangVisitor import SimpleLangVisitor
```

La clase SimpleLangVisitor es mi clase principal, ya que se encargará de validar y traducir las instrucciones del programa.magia:

```
class SimpleLangVisitor(ParseTreeVisitor):

    def __init__(self):
        #Diccionario para almacenar variables
        self.variables = {}
```

Se puede ver igual como creamos un diccionario al inicio para poder guardar las variables.

Recorre todas las sentencias del programa:

```
#Funcion para procesar el programa completo
def visitProgram(self, ctx: SimpleLangParser.ProgramContext):
    #Recorre todas las declaraciones del programa
    for statement in ctx.statement():
        self.visit(statement)
```

Se encarga de la correcta declaración de variables y su interpretación, verifica correctamente su estructura propuesta en la gramática:

```
#Metodo para procesar las declaraciones de variables
def visitVarDeclaration(self, ctx: SimpleLangParser.VarDeclarationContext):
    #Obtencion del nombre y valor de las variables
    var_name = ctx.ID().getText()
    value = ctx.expr().getText()

    #Traduccion y evaluacion de las variables
    value = self.translate_statement(value)
    value = self.replace_logical_operators(value)
    python_code = f"{var_name} = {value}"

    #Ejecucion del codigo en lenguaje python
    exec(python_code, globals())
```


Se encarga de la validación e interpretación de las estructuras condicionales y cuando va a entrar en un segmento if o else:

```
#Metodo para manejar declaraciones condicionales
def visitIfStatement(self, ctx: SimpleLangParser.IfStatementContext):
    #Obtencion de la condicion y su cuerpo
    condition = ctx.expr().getText()
    condition = self.replace_logical_operators(condition)
    if_body = self.translate_statement(ctx.statement(0).getText())

    #Traduccion y validacion
    python_code = f"if {condition}:\n    {if_body.replace('\n', '\n    ')}"
    if ctx.statement(1) is not None:
        else_body = self.translate_statement(ctx.statement(1).getText())
        python_code += f"\nelse:\n    {else_body.replace('\n', '\n    ')}"

    #Ejecucion del codigo en lenguaje python
    exec(python_code, globals())
```

De igual forma, valida e interpreta las estructuras de bucles y sus condiciones para parar:

```
#Metodo para manejar los bucles
def visitLoopStatement(self, ctx: SimpleLangParser.LoopStatementContext):
    #Obtencion de la condicion y el cuerpo del bucle
    condition = ctx.expr().getText()
    condition = self.replace_logical_operators(condition)
    loop_body = "\n    ".join([self.translate_statement(statement.getText()) for statement in ctx.stateme

    #Traduccion
    python_code = f"while {condition}:\n    {loop_body}"

    #Ejecucion del codigo en lenguaje python
    exec(python_code, globals())
```

Revisa y evalúa que se cumpla la estructura para imprimir algo en consola:

```
#Manejo de los imprimir
def visitPrintStatement(self, ctx: SimpleLangParser.PrintStatementContext):
    #Obtencion de la sentencia y valor
    value = ctx.expr().getText()

    #Validacion y traduccion
    value = self.replace_logical_operators(value)
    value = self.translate_statement(value)
    python_code = f"print({value})"

    #Ejecucion del codigo en lenguaje python
    exec(python_code, globals())
```

Traduce las sentencias con sus respectivas contrapartes:

```
#Traductor de las sentencias
def translate_statement(self, statement_text):
    if "escribir" in statement_text:
        return statement_text.replace("escribir", "print")

    if "aumentar" in statement_text:
        statement_text = statement_text.replace("aumentar", "").strip("(")
        statement_text = statement_text.replace(");", "").strip("(")
        return f"{statement_text} += 1;"

    if "disminuir" in statement_text:
        statement_text = statement_text.replace("disminuir", "").strip("(")
        statement_text = statement_text.replace(");", "").strip("(")
        return f"{statement_text} -= 1;"

    if "potencia" in statement_text:
        statement_text = statement_text.replace("potencia", "")
        statement_text = statement_text.replace(");", "").strip("(")
        return f"{statement_text} ** 2"

    if "raizCuadrada" in statement_text:
        statement_text = statement_text.replace("raizCuadrada", "")
        statement_text = statement_text.replace(");", "").strip("(")
        return f"{statement_text} ** 0.5"
    return statement_text
```

Podemos apreciar como devuelve lo equivalente en Python.

Traduce los operadores lógicos obtenidos por la gramática y el visitor para interpretarlos a lenguaje Python.

```
#Traductor de los operadores logicos
def replace_logical_operators(self, expression):
    expression = expression.replace('igual que', '==')
    expression = expression.replace('menor que', '<')
    expression = expression.replace('mayor que', '>')
    expression = expression.replace('menor igual a', '<=')
    expression = expression.replace('mayor igual a', '>=')
    expression = expression.replace('diferente de', '!=')
    expression = expression.replace('y que', 'and')
    expression = expression.replace('o que', 'or')

    return expression
```

Inicio del método main:

```
#Metodo main
def main():
    if len(sys.argv) != 2:
        print("Error: debes ejecutar el programa de la siguiente forma: python main.py <archivo.magia>")
        sys.exit(1)

    input_file = sys.argv[1]
```

Validación del archivo para corroborar que sea .magia:

```
try:
    #Verifica que el archivo tenga la extension .magia
    if not input_file.endswith(".magia"):
        print(f"Error: El archivo '{input_file}' no tiene la extensión '.magia'.")
        sys.exit(1)

    #Obtien el programa .magia
    input_stream = FileStream(input_file, encoding="utf-8")
```

El archivo obtenido es pasado por las 3 herramientas brindadas por ANTLR4:

```
#Hace uso del lexer y parser proporcionados por ANTLR4
lexer = SimpleLangLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = SimpleLangParser(token_stream)
```

El compilador crea el árbol de sintaxis del programa escaneado:

```
#Crea el arbol de sintaxis del programa
tree = parser.program()
```

Evaluación y recorrido de todo el árbol con ayuda del visitor:

```
#El visitor evalua el arbol y compilador
visitor = SimpleLangVisitor()
visitor.visit(tree)
```

Validación en caso de que no se encuentre el archivo:

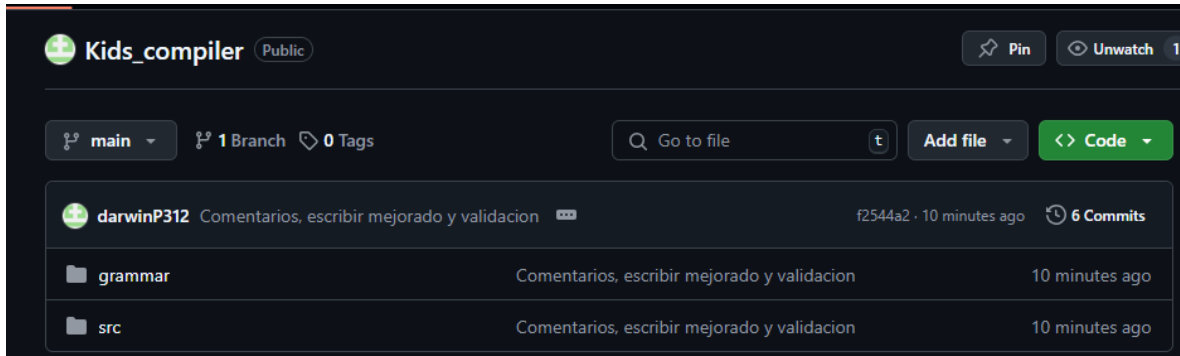
```
except FileNotFoundError:  
    print(f"Error: El archivo '{input_file}' no existe.")  
    sys.exit(1)
```

Repositorio del proyecto.

El proyecto con el compilador para el lenguaje LMPN (Lenguaje Mágico Para Niños), está en un repositorio en Github.

El link es el siguiente (se puede clonar o descargar):

https://github.com/darwinP312/Kids_compiler.



Se realizó el desarrollo usando Github y se puede apreciar sus diferentes versiones.