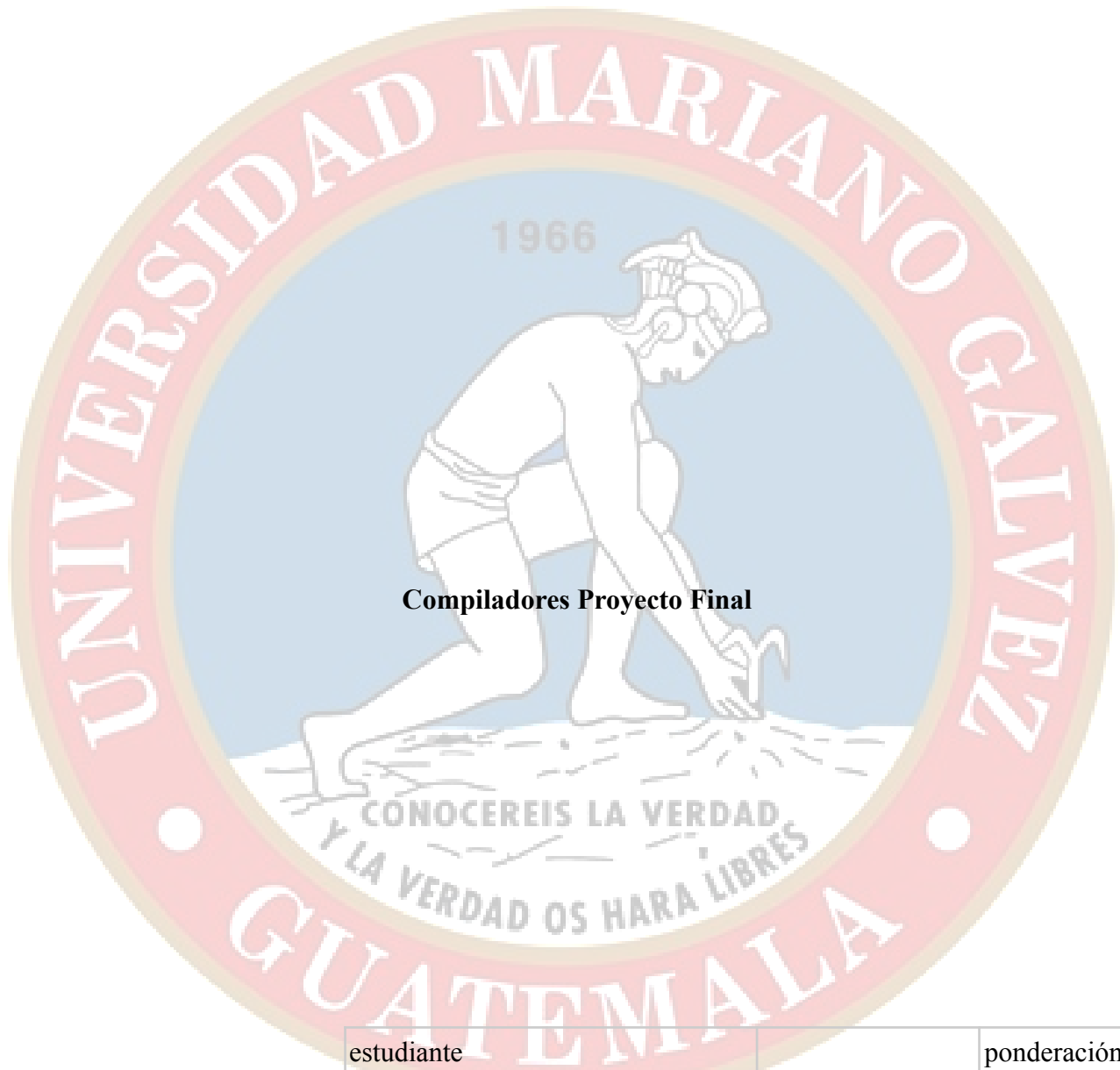


Universidad Mariano Gálvez Sede Boca Del Monte, Jornada Sabatina
Compiladores - 12025-7690-035-B
Inge. Ezequiel Urizar



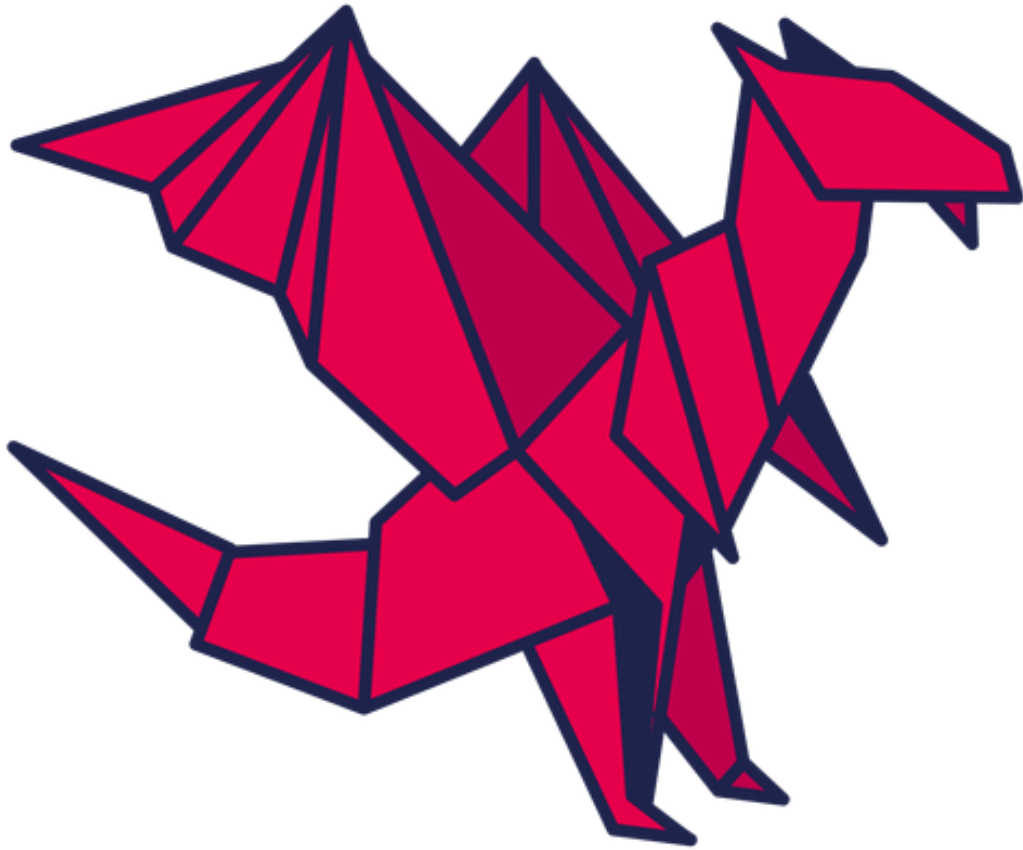
Compiladores Proyecto Final

estudiante		ponderación
Estiven Joel Laferre Guevara	7690-22- 2644	100/100
Melany Dayreli Romero Samayoa	7690-22- 48	100/100
Jose Armando Lopez Cruz	7690-21-2673	100/100

Sección B

Guatemala 23 de mayo del 2025

Antrax(<https://github.com/estiven-lg/Anthrax.git>)



Compilador Antrax - README

Descripción general

Antrax es un compilador desarrollado con **ANTLR**, **Clang** y **LLVM Lite**, capaz de traducir código fuente personalizado (.nx) a **LLVM IR** y binarios ejecutables. El proyecto ofrece una cadena de herramientas completa, desde el análisis sintáctico hasta la generación de código, así como un entorno de desarrollo integrado (IDE).

Estructura del proyecto

antlr4_data/	# Gramática ANTLR y archivos generados
├─ Antrax.g4	# Definición de gramática principal
├─ AntraxLexer.py	# Analizador léxico generado
├─ AntraxParser.py	# Analizador sintáctico generado
├─ AntraxIRVisitor.py	# Visitante personalizado para generación de IR
└─ ...	# Otros archivos auxiliares generados por ANTLR
Antrax_compiler/	# Componentes del compilador
└─ compiler.py	# Lógica principal del compilador
Antrax_IDE/	# IDE gráfico
├─ GUI.py	# Interfaz gráfica
└─ assets/	# Recursos visuales del IDE (iconos, etc.)
bin/	# Binarios generados
exe/	# Ejecutables para Windows
ll/	# Archivos intermedios LLVM IR
nx/	# Archivos de ejemplo (.nx)
obj/	# Archivos objeto intermedios
main.py	# Punto de entrada del compilador

Características principales

- **Análisis léxico/sintáctico:** Frontend basado en ANTLR.
 - **Generación de IR:** Utiliza LLVM Lite para representar el código.
 - **Optimización:** Soporte de pases de optimización con LLVM.
 - **Generación de código:** Producción de LLVM IR, objetos y ejecutables.
 - **IDE integrado:** Interfaz gráfica con resaltado de sintaxis y controles de compilación.
-

Dependencias

- Python 3.9+
- ANTLR 4
- LLVM (con bindings para Python)
- Clang
- LLVM Lite

Paquetes de Python

```
pip install antlr4-python3-runtime llvmlite
```

Dependencias del sistema (Linux/Unix)

```
sudo apt-get install llvm clang x86_64-w64-mingw32-gcc
```

Instalación

1. Clona este repositorio:

```
git clone https://github.com/usuario/antrax-compiler.git
cd antrax-compiler
```

2. Instala las dependencias necesarias.
-

Uso

Línea de comandos

```
python main.py [comando] [opciones]
```

Opciones disponibles:

- `-o salida` - Especifica el nombre del archivo de salida.
- `-emit-llvm` - Genera solo el IR de LLVM.
- `-optimize` - Aplica optimizaciones.

IDE

Ejecuta la interfaz gráfica:

```
python Antrax_IDE/GUI.py
```

Ejemplos

- Archivos de entrada `.nx` disponibles en el directorio `nx/`
 - Salidas IR generadas en `ll/`
-

Proceso de compilación

1. Análisis léxico/sintáctico:

```
lexer = AntraxLexer(InputStream(source_code))
stream = CommonTokenStream(lexer)
parser = AntraxParser(stream)
tree = parser.root()
```

2. Generación de IR:

```
visitor = AntraxIRVisitor()
module_ir = visitor.visit(tree)
```

3. Optimización:

```
opt -S -O2 input.ll -o optimized.ll
```

4. Generación de binarios:

Linux:

```
clang -o salida input.ll
```

Windows:

```
llc -filetype=obj -mtriple=x86_64-pc-windows-gnu input.ll  
x86_64-w64-mingw32-gcc -o salida.exe input.obj
```

Comandos disponibles

<code>python main.py visit archivo.nx</code>	# Analiza el árbol de derivación
<code>python main.py llvm archivo.nx salida.ll</code>	# Genera LLVM IR
<code>python main.py exec archivo.nx</code>	# Ejecuta el código directamente
<code>python main.py bin archivo.nx salida</code>	# Genera binario ejecutable

Ejemplo completo de flujo

```
# Generar IR\python main.py llvm nx/test1.nx output.ll  
  
# Optimizar IR  
opt -S -O2 output.ll -o optimized.ll  
  
# Ejecutable en Linux  
clang optimized.ll -o programa  
  
# Ejecutable en Windows  
llc -filetype=obj -mtriple=x86_64-pc-windows-gnu optimized.ll  
x86_64-w64-mingw32-gcc optimized.obj -o programa.exe
```

Detalle de los pasos del proceso

1. Generación de Código Intermedio: 0.002 segundos
En este paso, el código fuente escrito en Antrax se traduce a un código intermedio, generalmente LLVM IR (.ll). Esta representación es más cercana al lenguaje máquina, pero aún independiente de la arquitectura del sistema operativo.
2. Optimización de Código Intermedio: 0.091 segundos
Aquí se aplica un conjunto de reglas para mejorar el rendimiento del código intermedio. Estas optimizaciones pueden incluir eliminación de código redundante, simplificación de expresiones, mejoras en el uso de memoria, entre otros.
3. Guardando Código Intermedio: 0.000 segundos
El archivo .ll optimizado es guardado. En este caso, el proceso fue tan rápido que el tiempo registrado es prácticamente cero.
4. Generación de Archivo Binario: 0.316 segundos
Se compila el código intermedio optimizado hacia un archivo ejecutable binario (.exe para Windows). Este archivo es el que podrá ser ejecutado directamente en el sistema operativo.
5. Ejecución de Archivo Binario: 0.002 segundos
Finalmente, el archivo ejecutable se ejecuta, y este es el tiempo que tardó el programa en correr.

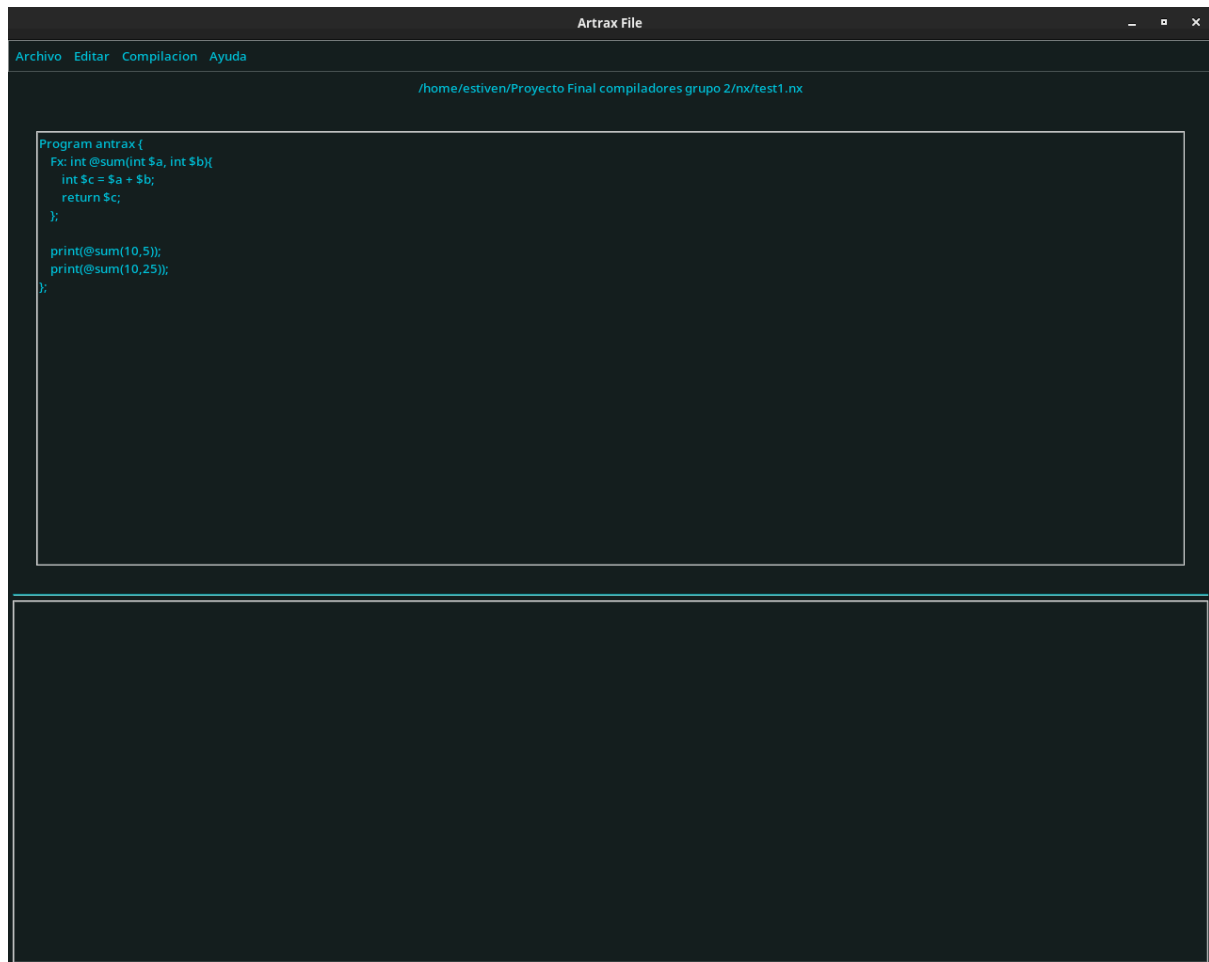
```
Generación de Código Intermedio: 0.002 segundos
Optimización de Código Intermedio: 0.091 segundos
Guardando Código Intermedio: 0.000 segundos
Generación de Archivo Binario: 0.316 segundos
Ejecución de Archivo Binario: 0.002 segundos
=====
0
1
2
3
4
Iteracion
Iteracion
Iteracion
Iteracion
Iteracion

Fin de ejecución
=====
```

El IDE de Antrax

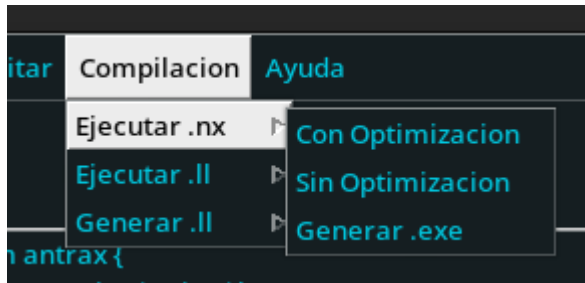
El IDE de Antrax es una herramienta sencilla que incluye un editor de texto, una consola, opciones para guardar y recuperar archivos, y lo más importante: la opción de compilación.

Es en este punto donde el IDE ofrece varias alternativas para trabajar con el código escrito en Antrax.



Opciones del menú de compilación: Ejecutar .nx, Ejecutar .ll y Generar .ll

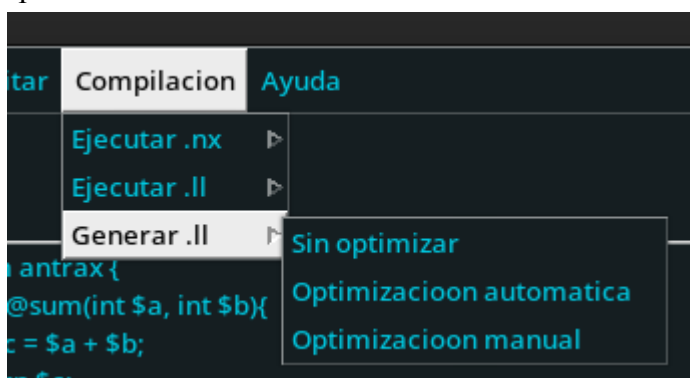
- **Ejecutar .nx:** Permite ejecutar directamente el código Antrax que ha sido ingresado por el usuario. Ofrece varias opciones: ejecutar con o sin optimización, y generar un archivo .exe compatible con Windows.



- **Ejecutar .ll:** Una vez que se ha generado el archivo intermedio en formato LLVM IR (.ll), esta opción permite ejecutar el código desde ese nivel intermedio. Al igual que en Ejecutar .nx, se puede elegir entre diferentes niveles de optimización.



- **Generar .ll:** Esta opción es un paso previo necesario para usar "Ejecutar .ll", ya que se encarga de generar los archivos en formato LLVM IR. Además, permite seleccionar entre tres tipos de optimización: sin optimizar, optimización automática y optimización manual.



Optimizaciones en el Compilador Antrax

Función Intermediate_Code_Optimization

Esta función es responsable de aplicar optimizaciones al código intermedio LLVM IR. Veamos su funcionamiento detallado

```
def Intermediate_Code_Optimization(module_ir, passes="-O2"):
    """Optimiza el código intermedio"""
    input_path = "./ll/temp.ll"
    output_path = "./ll/optimized.ll"

    with open(input_path, "w") as f:
        f.write(str(module_ir))

    subprocess.run(["opt", "-S", passes, input_path, "-o", output_path])

    with open(output_path, "r") as f:
        optimized_ir = f.read()
    return optimized_ir
```

El Optimizador opt de LLVM

La herramienta opt es parte del conjunto de herramientas de LLVM y realiza transformaciones y optimizaciones en código LLVM IR.

Parámetros clave:

- -S: Indica que la entrada/salida es en formato legible (textual)
- passes: Especifica el nivel/pases de optimización (en este caso "-O2")
- -o: Archivo de salida

Niveles de Optimización

El compilador Antrax usa principalmente estos niveles:

1. -O0: Sin optimizaciones (modo debug)
2. -O1: Optimizaciones básicas
3. -O2: Optimizaciones moderadas (equilibrio entre rendimiento y tiempo de compilación)

4. -O3: Optimizaciones agresivas (máximo rendimiento)
5. -Os: Optimización para tamaño de código
6. -Oz: Optimización agresiva para tamaño

Optimizaciones típicas aplicadas en -O2

Cuando usas -O2, el optimizador aplica entre otras:

1. Eliminación de código muerto (DCE): Remueve código que no afecta los resultados
2. Propagación de constantes: Reemplaza variables con valores conocidos en tiempo de compilación
3. Inline de funciones: Reemplaza llamadas a funciones pequeñas con su contenido
4. Eliminación de subexpresiones comunes: Detecta y elimina cálculos redundantes
5. Simplificación de CFG: Optimiza el flujo de control del programa
6. Loop optimizations: Optimiza bucles (rotación, desenrollado parcial)

Diseño y reglas de la gramática del Lenguaje v1

Para el diseño de el lenguaje Antrax nos basamos en lenguajes modernos, buscamos mezclar la simpleza de Python pero la buena estructura de lenguajes como Java, C#.

```
grammar Antrax;  
  
// definicion de la estructura del programa  
root: 'Program' 'antrax' '{' stat+ '}' EOF?;
```

Nuestro lenguaje nace con el nombre de Antrax, y en la imagen anterior demuestra la estructura que cada programa debe iniciarse con la siguiente cláusula.

```
Program antrax {  
    //Codigo  
}
```

Antrax se inspira en la estructura de cada programa en lenguajes como Java y C#, pero sus referencias también incluyen PHP, pues como podemos observar en la imagen, el nombre de cada variable debe iniciar con \$, siendo el TOKEN ID, el conjunto de reglas que cada variable. Siendo mas especifico, cada variable en Antrax debe iniciar con \$ e iniciar con al menos una letra, Ejemplos: \$x, \$nombre123, \$ap12Mar, son ejemplos validos de variables en Antrax, \$123, \$1manzana y \$@umbrella, no son validos.

```
ID: '$' [a-zA-Z_]([a-zA-Z_0-9])*;  
varDecl: ID '=' expr;
```

```
stat:
|   varDecl ';'
|   | ifStat ';'
|   | whileStat ';'
|   | forStat ';'
|   | printStat ';;'
```

Estas son Stats, que conforman cada una de las instrucciones que conforman por ahora lo que Antrax es.

- varDecl: que es la declaración de variable
- ifStat, que es la estructura de control if de toda la vida
- whileStat: que es el ciclo while,
- forStat: que es el ciclo for,
- printStat: que es la acción de imprimir el resultado

```
if ($x < $y) {
    print("x es menor que y");
} else {
    print("x es mayor o igual que y");
};
```

Como se ve en la imagen, cada una de estas sentencias deben finalizar con ;, Ejemplo grafico:

```
/ ifStat:
/ |   'if' '(' expr ')' '{' ifAction '}' (
|   |   'else' '{' elseAction '}'
|   )?;
```

```
whileStat: 'while' '(' expr ')' '{' stat+ '}';
```

```
printStat: 'print' '(' (expr | STRING) ')';
```

Donde cabe aclarar que stat+, indica que por lo menos, al instanciar alguna de estas sentencias debemos agregar al menos una sentencia.

Ahora, dentro de Antrax, dado que este es un lenguaje sencillo y enfocado a realizar operaciones matemáticas básicas, pensamos en una regla que defina cada una de las operaciones soportadas, su nombre es expr, que viene de expresión matemática.

Expr es recursiva consigo misma, además dentro de esta regla se contemplaron operadores booleanos.

```
expr:
  expr (MULT | DIV) expr
  | expr (PLUS | MINUS) expr
  | '(' expr ')'
  | NUM
  | expr (LT | GT | LEQ | GEQ | EQ | NEQ) expr
  | ID;
```

En la anterior imagen vemos desglosados varios Token de los cuales veremos a continuación su definición formal.

```
// Definición de tokens
ID: '$' [a-zA-Z_]([a-zA-Z_0-9])*;
NUM: [0-9]+ ('.' [0-9]+)?;
STRING: '"' .*? '"';

// Operadores aritméticos
PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';

// Operadores de comparación
LT: '<';
GT: '>';
LEQ: '<=';
GEQ: '>=';
EQ: '==';
NEQ: '!=';
```

En donde cabe aclarar que el token NUM acepta números reales, es decir, enteros y decimales son reconocidos bajo el mismo token, esto por simplicidad dada la meta de esta gramática.

Expr puede derivarse en cada una de las formas vistas en la imagen anterior, lo que permite a Antrax manejar expresiones mas grandes.

Diseño y reglas de la gramática del Lenguaje v2

```
stat:
    varDecl ';'
  | varAsg ';'
  | ifStat ';'
  | whileStat ';'
  | forStat ';'
  | printStat ';'
  | retStat ';'
  | funcStat ';'
  | funcCall ';'
  | tryStat ';'
  ;
```

En la actualización de nuestra gramática, agregamos funciones y el bloque de código, Try-catch para manejo en código de errores.

Estas sentencias pueden verse como las derivaciones: funcStat, funcCall, tryStat, retStat. Siendo estas, el código de la función, el llamado de una función, el bloque try-catch, y la cláusula return.

Este sería la definición formal de la regla para el bloque try-catch, que se desglosa en dos diferentes scopes: tryAction y catchAction. Que se explican por sí mismas.

```
//tryCatch statement
tryStat: 'try' '{' tryAction '}' 'catch' ID '{' catchAction '}' ;
```

Estas reglas también pueden derivarse en más statements, lo que permite el código dentro de los bloques.

```
//acciones de trycatch
tryAction: stat+;
catchAction: stat+;
```

Antrax contempla el uso de funciones, estas mantienen una estructura similar a Java, donde se especifica el tipo de objeto a retornar, el nombre además de que cada función debe comenzar con “Fx:” para hacer el código más entendible.

```
// estructura de una funcion
funcStat: 'Fx:' TYPE FID '(' params? ')' '{' stat+ '}' ;
// estructura de los parametros de una funcion
params: (param (',' param)*);
param: TYPE ID;
//sentencia return
retStat: RETURN (STRING | expr) ' ';

// llamamos a la funcion
funcCall: FID '(' (expr (',' expr)*)? ')';
```

Params, son los parámetros, FID, es el nombre de la función, hacemos diferencia entre el ID, que es el nombre de una variable, y FID, que es para funciones.


Pasemos ahora con las variables, se implemento el tipado estricto y la reasignacion de valor.

```
// estructura de una declaración de variable  
varDecl: TYPE ID '=' expr;  
// reasignacion de variable  
varAsg: ID '=' expr;
```

Ahora, además de esto, se debe mencionar que para llamar a una función, es requerido utilizar @ antes del nombre, Ejemplo: @funcion.

LLVM IR generado

Código ingresado:



```
Program antrax {  
  Fx: int @sum(int $a, int $b){  
    int $c = $a + $b;  
    return $c;  
  };  
  int $res = @sum(4,5);  
  
  print($res);  
};
```

IR Resultante:

```
; ModuleID = "AntraxModule"
target triple = "x86_64-pc-linux-gnu"
target datalayout = ""

define i32 @main()
{
entry:
  %"$res" = alloca i32
  %".2" = call i32 @sum(i32 4, i32 5)
  store i32 %".2", i32* %"$res"
  %"$res.1" = load i32, i32* %"$res"
  %".4" = bitcast [4 x i8]* @fmt to i8*
  %".5" = call i32 (i8*, ...) @printf(i8* %".4", i32 %"$res.1")
  ret i32 0
}

declare i32 @printf(i8* %".1", ...)

@fmt = internal constant [4 x i8] c"%d\0a\00"
@fmt_str = internal constant [4 x i8] c"%s\0a\00"
define i32 @sum(i32 %".1", i32 %".2")
{
entry:
  %"$a" = alloca i32
  store i32 %".1", i32* %"$a"
  %"$b" = alloca i32
  store i32 %".2", i32* %"$b"
  %"$c" = alloca i32
  %"$a.1" = load i32, i32* %"$a"
  %"$b.1" = load i32, i32* %"$b"
  %"addtmp" = add i32 %"$a.1", %"$b.1"
  store i32 %"addtmp", i32* %"$c"
  %"$c.1" = load i32, i32* %"$c"
  ret i32 %"$c.1"
}
```