UNIVERSIDAD TECNOLÓGICA DE SANTIAGO (UTESA)

Sistema Corporativo
Facultad de Arquitectura e Ingeniería
Carrera de Ingeniería en Sistemas Computacionales

Compiladores
Proyecto final
The Compiler

Presentado a:
Prof. Iván Mendoza

Presentado por:

Alberto de Jesús García Peña	2-17-1097
Guillermo Santos Marzan	2-18-0494
Jean Luciano Ureña	2-17-1955

Santiago, provincia Santiago de Los Caballeros,
República Dominicana.
Agosto, 2022.

ÍNDICE

INTRODUCCIÓN

En este documento se detallarán conceptos teóricos y prácticos utilizados para la elaboración de nuestro compilador, nuestros objetivos a realizar, así como una breve introducción al funcionamiento del mismo. Además, contaremos al final con una serie de imágenes que mostrarán el despliegue de nuestro compilador.

CAPÍTULO 1: DESCRIPCIÓN DEL PROYECTO

1.1. PROBLEMA

Los compiladores generalmente son poco intuitivos para los desarrolladores, creando una barrera para el desarrollo de programas complejos. Algunos están desarrollados para que solo ejecuten una clase o archivo del lenguaje y los que son multi-archivos requieren una previa configuración, lo cual hace que la herramienta sea compleja de utilizar.
Otro inconveniente es que detectan errores poco descriptivos, los cuales no especifican el origen del error o no presentan la falla original del programa. También, los compiladores hacen uso de funciones principales, donde la declaración de esta es obligatoria para iniciar la ejecución de un programa, lo cual nos limita a declarar nuestras propias funciones afines a lo que queremos ejecutar tan pronto inicie la ejecución de nuestro programa.

1.2. PLANTEAMIENTO DE LA SOLUCIÓN

Nuestro compilador permitirá seleccionar un directorio, donde podremos crear carpetas y archivos para organizar nuestro código como nos plazca, además tendrá los mensajes de errores apuntando a la línea y sentencia la cual lo provocó, junto con el nombre del archivo.

No será necesaria la declaración de una función “main”, sino que en un archivo se puede colocar todo su código directamente, y el compilador se encargará de recogerlo y crear la función main.

Por complejidad, la solución elegida es un lenguaje del paradigma funcional con similitudes sintácticas a JavaScript y F#, además de tener como objetivo el framework de .NET y el lenguaje al cual apuntan todos estos, IL.

1.3. OBJETIVOS DEL PROYECTO

1.3.1. OBJETIVO GENERAL

Desarrollar un compilador que sea fácil de usar y tenga la capacidad de ejecutar todos sus procesos de forma exitosa ante las acciones que el usuario le indique.

1.3.2. OBJETIVOS ESPECÍFICOS

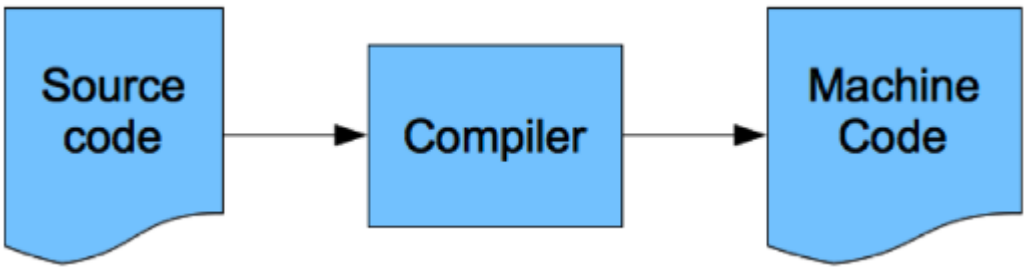
- Traducir el código de forma correcta y eficiente.
- Tener una interfaz amigable al usuario.
- Mejorar la compatibilidad con los diferentes sistemas operativos.
- Mejorar la compatibilidad con el framework .NET.
- Disminuir el consumo de recursos al momento de compilar.
- Brindar retroalimentación de forma constante y detallada sobre los posibles errores que se puedan encontrar en el código.

CAPÍTULO 2: MARCO TEÓRICO

2.1. INTRODUCCIÓN A LOS COMPILADORES

2.1.1. DEFINICIÓN DE UN COMPILADOR

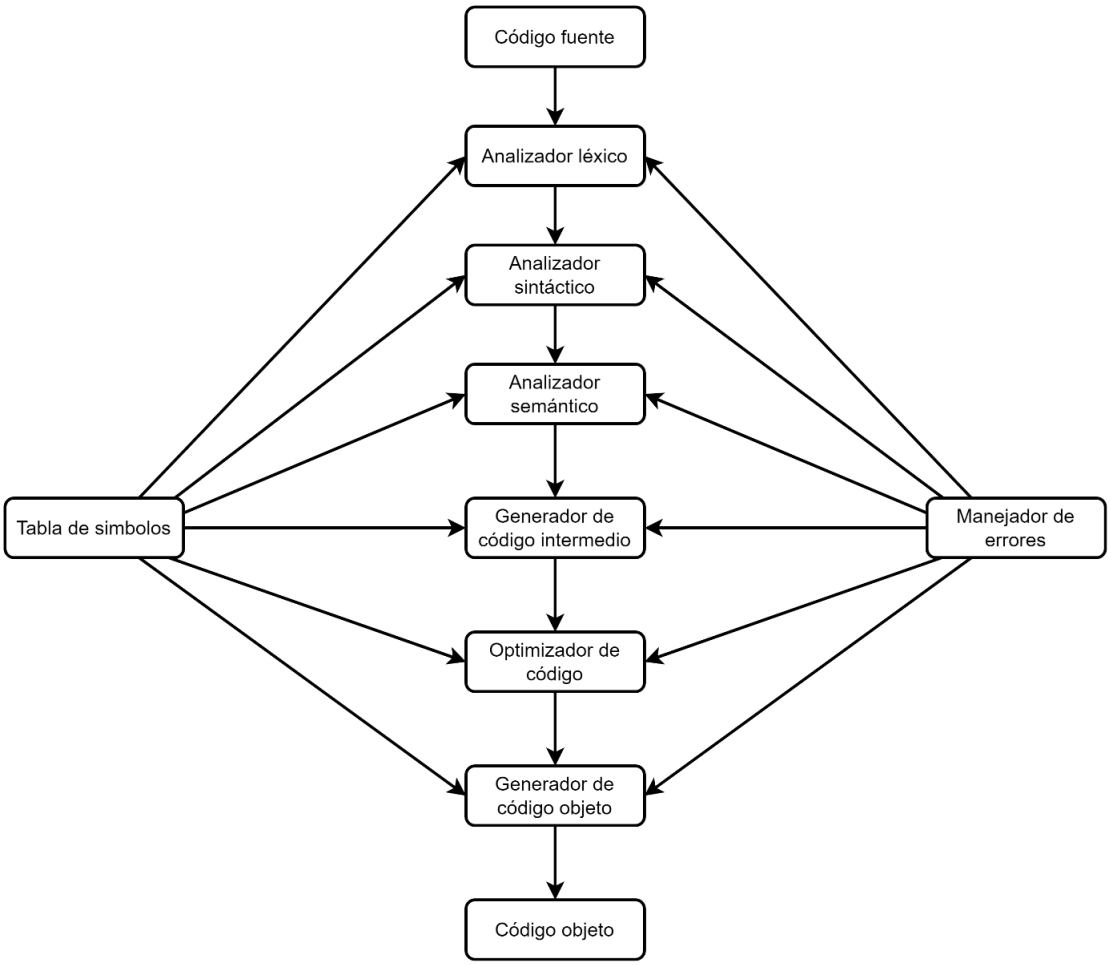
Un compilador es un programa que tiene como objetivo cambiar el código fuente a un lenguaje de máquina más legible por una computadora. El resultado de la compilación se llama código objeto y es lo que el procesador toma para poder realizar una instrucción a la vez.



Funcionamiento básico de un compilador.

El compilador es importante para poder comprender la comunicación entre una máquina y un lenguaje de alto nivel. Sabiendo bien el funcionamiento intermedio del mismo, se podrá mejorar el desarrollo en lenguajes de alto nivel.

2.1.2. ESTRUCTURA DE UN COMPILADOR



Estructura de un compilador.

Un compilador está estructurado por los siguientes elementos:

Analizador léxico: Revisa los caracteres que conforman el programa fuente y los une en secuencias significativas, conocidas como lexemas.

Analizador sintáctico: Hace uso de los elementos principales de los tokens producidos por el analizador léxico para desarrollar una representación intermedia en forma de árbol que defina la estructura gramatical del flujo de tokens.

Analizador semántico: Identifica si los elementos del código fuente son significativos o no.

Generador de código intermedio: Traduce el código fuente a código de máquina.

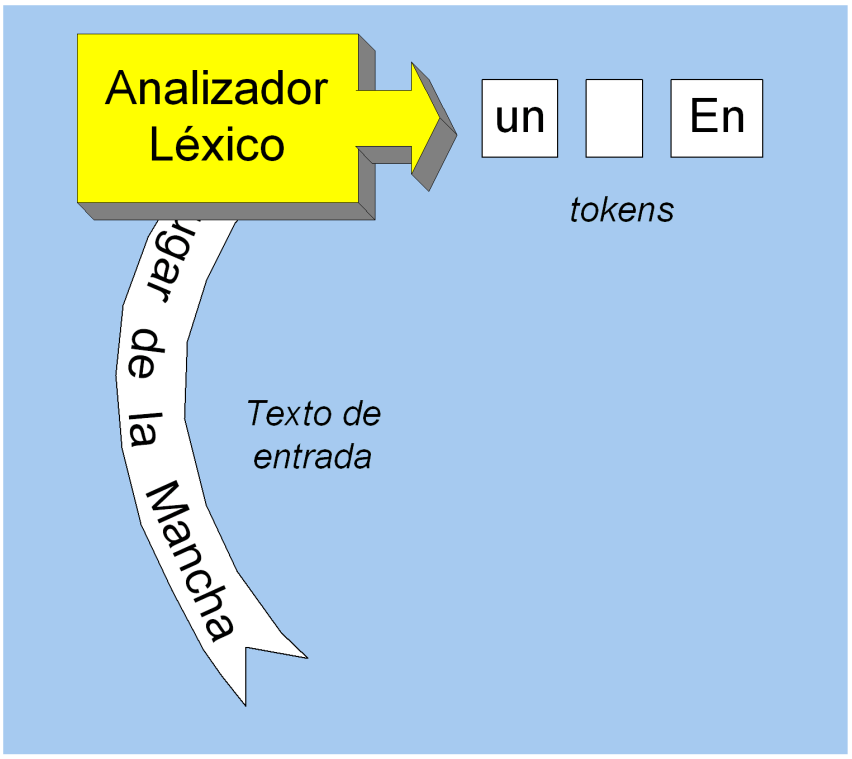
Optimizador de código: Transforma el código para que agote menos recursos y produzca más rapidez.

Generador de código objeto: Desarrolla un código que la máquina pueda entender y también guarda la asignación, la elección de instrucciones, etc. El código optimizado se transforma en código de máquina que luego forma la entrada al cargador y al enlazador.

Tabla de símbolos: Estructura de datos que contiene un registro para cada nombre de variable y campos para los atributos que tiene el nombre.

Manejador de errores: Detecta y avisa de cualquier error que suceda al compilar.

2.2. ANÁLISIS LÉXICO



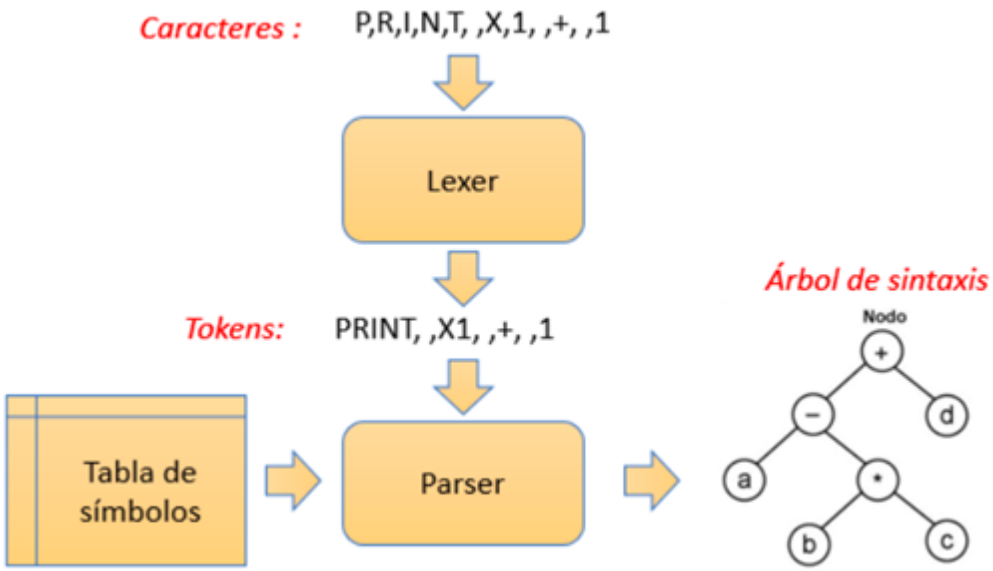
Ejemplo del funcionamiento de un analizador léxico.

En una de las fases que componen a un compilador. En esta, la cadena de caracteres que compone el código fuente se lee de izquierda a derecha y se reúnen en elementos léxicos, secuencias de caracteres que tienen un significado en conjunto.

Aquí entra en juego un término de suma importancia: token. Un token es una cadena que lleva un significado atribuido y, por lo tanto, definido. Su

estructura está compuesta por un nombre y un valor que puede ser opcional. El nombre del token es una clase de unidad léxica. Los nombres de token más comunes son: identificador, palabra clave, separador, operador, literal y comentario.

2.3. ANÁLISIS SINTÁCTICO



Funcionamiento del análisis sintáctico.

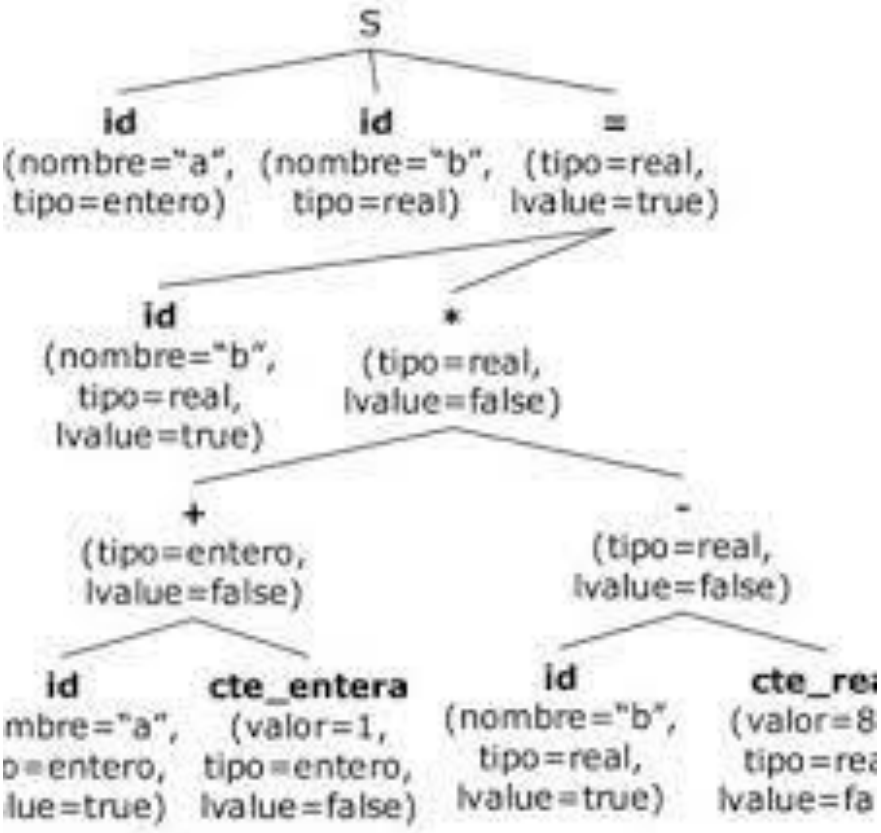
Es el análisis que nos dice el significado lógico de ciertas oraciones dadas o partes de esas oraciones. Este convierte los tokens provistos por el analizador léxico en otra estructura que facilita el análisis posterior y captura de la jerarquía de los mismos.

2.3.1. ANÁLISIS SINTÁCTICO DESCENDENTE

El análisis descendente trata de hallar entre las producciones de la gramática la deducción por la izquierda del símbolo de inicio para una cadena inicial. Se puede dividir en:

- Análisis sintáctico por descenso recursivo.
- Análisis sintácticos predictivos.
- Análisis sintáctico predictivo no recursivo.

2.4. ANÁLISIS SEMÁNTICO



Funcionamiento del análisis semántico.

Es la fase del compilador en donde se recoge la estructura sintáctica del código fuente y se verifica que sea semánticamente correcto.

2.4.1. TRADUCCIÓN DIRIGIDA POR LA SINTAXIS

Esta establece que el concepto de una construcción se encuentra relacionado directamente con su estructura sintáctica, y a la vez está simbolizada en su árbol de análisis sintáctico.

2.4.2. ATRIBUTOS

Los atributos son variables que ejemplifican una propiedad de un símbolo, sea no terminal o terminal. Algunos atributos pueden ser: tipo de dato, valor, posición en memoria, etc.

2.4.3. NOTACIONES PARA ASOCIAR REGLAS SEMÁNTICAS

2.4.3.1. DEFINICIÓN DIRIGIDA POR LA SINTAXIS

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

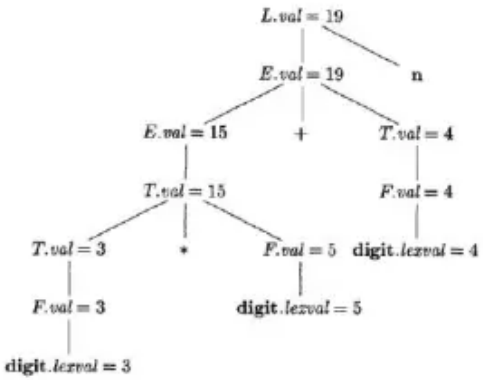


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

Ejemplo de una definición dirigida por la sintaxis.

Esta establece que el concepto de una construcción se encuentra relacionado directamente con su estructura sintáctica, y a la vez está simbolizada en su árbol de análisis sintáctico.

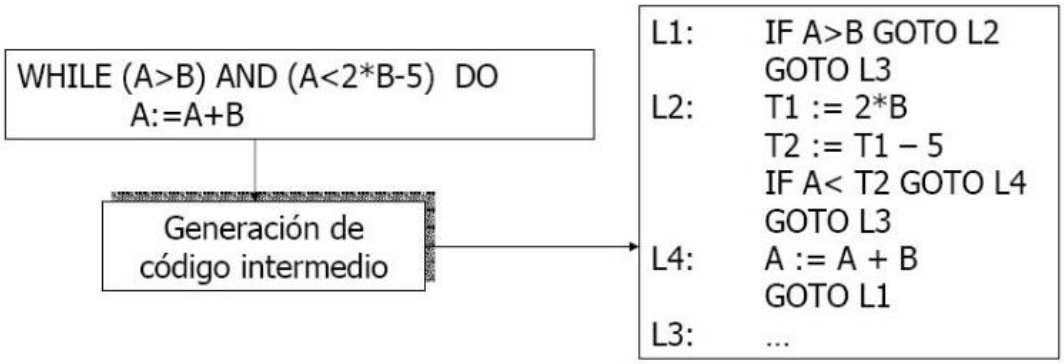
2.4.3.2. ESQUEMA DE TRADUCCIÓN

Producción	Reglas Semánticas
L → E n	print (E.val)
E → E_1 + T	E.val := E_1.val + T.val
E → T	E.val := T.val
T → T_1 * F	T.val := T_1.val x F.val
T → F	T.val := F.val
F → (E)	F.val := E.val
F → digito	F.val := digito.valex

Ejemplo de un esquema de traducción.

Esta es una gramática independiente del contexto en la que se insertaron fragmentos de código en las particiones derechas de cada una de sus reglas de producción.

2.5. GENERACIÓN DE CÓDIGO INTERMEDIO



Ejemplo del proceso de generación de código intermedio.

Esla etapa del compilador que genera las instrucciones en código intermedio u objeto para un nivel específico de una máquina. Estos códigos pueden ser:

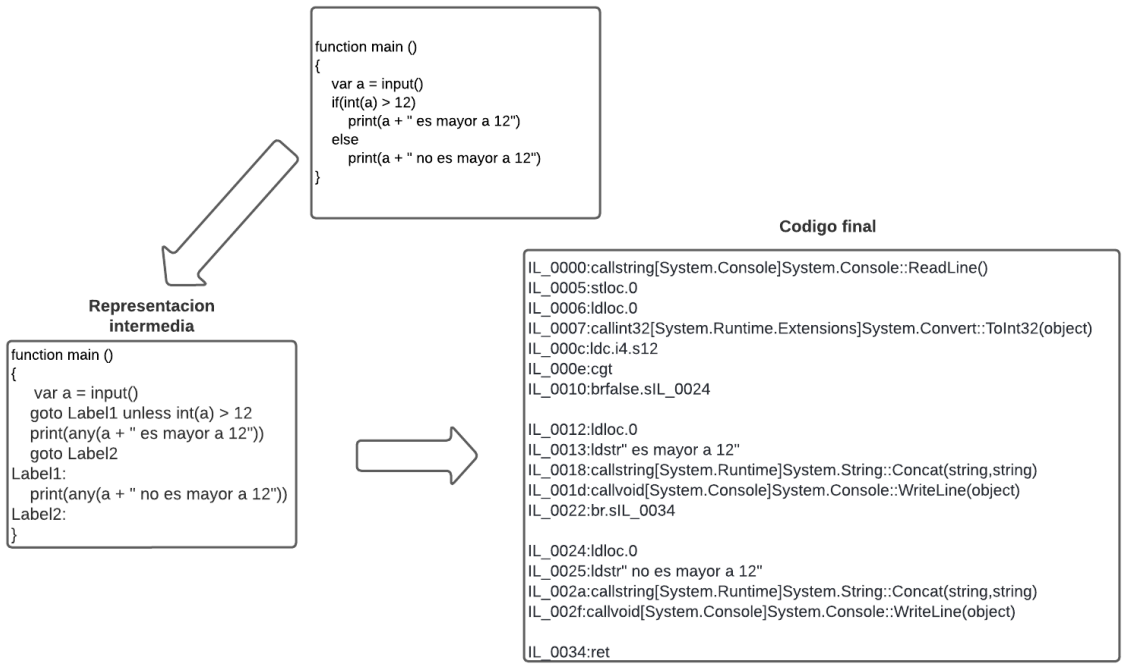
- **Lenguaje máquina ejecutable:** Puede cargar y ejecutar el programa de forma inmediata.

- **Lenguaje máquina reubicable:** Es la opción elegida por gran parte de los compiladores que existen, pero se necesita de un cargador y un enlazador para su funcionamiento.

- **Lenguaje ensamblador:** Puede usar instrucciones simbólicas y macros de ensamblador, pero sería necesario agregar el proceso de ensamblado.

- **Lenguaje de alto nivel:** Hace las cosas más simples, ya que muchos problemas los podrá solucionar el compilador de ese lenguaje.

2.6. GENERACIÓN DE CÓDIGO FINAL



Ejemplo de la conversión de un programa que recibe un número y dice si este es menor o mayor a 12.

Es la fase de la compilación luego de la de generación de código intermedio y la opcional fase de optimización de código intermedio. En esta las expresiones y sentencias del código intermedio son llevadas a expresiones y sentencias del leguaje final.

2.7. TABLAS DE SÍMBOLOS Y TIPOS

La tabla de símbolos son un conjunto de estructura de datos, las cuales cumplen una multitud de funciones, de esta identificamos los tokens, palabras claves y operadores aceptados por el lenguaje de programación. Existen varios tipos de tabla símbolos, como:

- **Tabla de símbolos global:** Esta se utiliza como el árbol sintáctico, y permite acceder a cualquier parte del programa a compilar.

- **Tabla de símbolos de alcance:** Esta se llena como resultado del análisis semántico, almacena las declaraciones de funciones y variables, así como sus llamadas, advierte, esta se usa para advertir de operaciones invalidas, llamada a funciones o variables no existentes o la inserción de parámetros faltantes o de más en una función.

2.8. ENTORNO DE EJECUCIÓN

Un entorno de ejecución viene siendo un espacio en el que reside nuestra aplicación compilada a la hora de ejecutarse en la maquina destino. En esta residen los módulos del programa, punto de entrada, referencias y bibliotecas, entre otras cosas que necesitara nuestro programa para ejecutarse.

Desde otro punto de vista, es un espacio que solicita nuestro programa al sistema operativo objetivo para poder funcionar.

2.9. OPTIMIZACIÓN DE CÓDIGO

La fase de optimización se encarga de mejorar el código intermedio del programa, de manera que, para la fase siguiente el código sea más rápido de ejecutar y utilice menos recursos de memoria. Al optimizar el código el compilador debe cumplir con las siguientes condiciones:
Existen dos tipos generales de optimización de código:

- **Optimizaciones dependientes de la máquina:** Estas se realizan luego de que el código objeto se ha generado y cuando este cambia en base a la arquitectura de la máquina destino. Utiliza registros de CPU y tiene la capacidad de usar referencias de memoria absolutas en lugar de las relativas, por lo que buscan sacar el mayor provecho de la jerarquía de memoria

- **Optimizaciones independientes de la máquina:** Son las optimizaciones que se realizan luego de obtenido el programa en su representación intermedia. Dependiendo del compilador u objetivos de este, estas optimizaciones pueden ser de distintas formas, desde la eliminación de código muerto (código no ejecutable), reducción de redundancia hasta la reducción de constantes.

Ejemplo de una optimización de código:

Tenemos la siguiente operación:

```
t1 = 8.0;
t2 = valor * t1;
t3 = fijo + t2;
comisión = t3;
```

Es posible optimizarla de la siguiente forma:

t2 = valor * 8.0;
comisión = fijo + t2;

2.9. MANEJO DE ERRORES

Como se muestra en este diagrama, el manejo de errores es un complemento de los compiladores que nos permite saber si hemos cometido algún error a la hora de escribir nuestro programa.

Funcionamiento del manejo de errores.

Este está presente en todas las fases del compilador de las cuales recoge información específica, como los tokens no reconocidos del analizador léxico, estructuras mal formadas del sintáctico y operaciones y llamadas de funciones o variables no validas en el analizador semántico, así como otros errores, como la falta de referencias en las fases de síntesis.

2.10. HERRAMIENTAS DE CONSTRUCCIÓN DE COMPILADORES

- **Generadores de analizadores sintácticos:** YACC y Bison, por ejemplo.

- **Generadores de escáneres:** LEX y FLEX, por ejemplo.

- **Motores de traducción:** Este nos ayuda a producir rutinas para recorrer un árbol.

- **Generadores de generadores de código:** Un ejemplo conocido es Mono.Cecil, una librería de C# que nos permite escribir aplicaciones en IL.

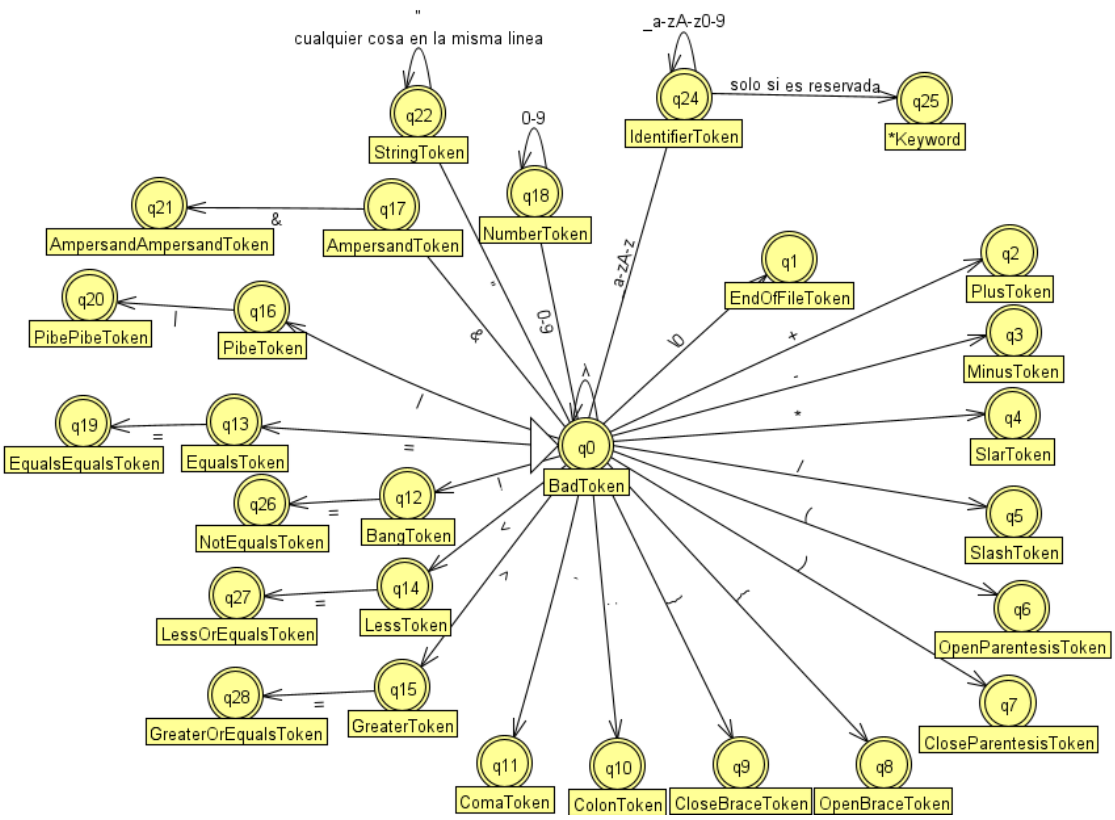
CAPÍTULO 3: DESARROLLO

3.1. ANALIZADOR LÉXICO

Nuestro analizador léxico es un analizador de 3 buffers (aunque por lo general solo se usan 2) el cual lee el texto de entrada token a token, es el primero de los analizadores en ser ejecutado y del cual dependen todos los demás.

El analizador léxico solo tiene 2 puntos de entrada, uno es a través del analizador sintáctico, el cual lo usa para cada token del texto origen, y otro es por la salida global (una clase llamada **Compilation**), la cual retorna todos los tokens de un texto dado.

3.1.1. AUTÓMATA DEL ANÁLISIS LÉXICO



Autómata del análisis léxico.

Este se encarga de dos cosas, identificar los tokens en el texto de entrada, y eliminar los comentarios y espacios en blanco. Esto último lo hace copiando algo que hace “Roslyn” (un compilador de C#), que es los almacena en trivia, unos tokens especiales que no se consideran parte del programa. Entre estos están los comentarios, tabulaciones, espacios en blanco etc.

Esto lo hace en dos pasos, primero busca toda la trivia antes de un token, luego busca el token, y finalmente busca toda la trivia después de ese token, luego de haber realizado esto, retorna el valor del token que se haya identificado.

El autómata está compuesto por dos clases:

- **Lexic Analyzer:** la clase principal del analizador léxico y donde también se encuentran las funciones **ReadTrivia** y **ReadToken**, que conforman al autómata del analizador léxico.

- **SyntaxFacts:** una clase que almacena las constantes del lenguaje, esta es una clase multipropósito, se encarga de la tarea de identificar el texto que conforma un token de valor fijo (operadores y palabras claves), identificar si tokens son trivia, palabras claves u operadores, e incluso la precedencia de los operadores, que es utilizada por el analizador sintáctico para determinar el orden de los nodos en el árbol sintáctico.

3.1.2. TABLA DE SÍMBOLOS

La tabla de símbolos se encuentra dispersa en todo el programa con diversos nombres, en resumen, este sería su contenido:

TRIVIA	
Partes del texto origen que, aunque no son tokens, cumplen una condición especial. Estas siempre están antes o después de un token, y pertenecen al token más cercano por delante o al token anterior a este.	
NOMBRE	**DEFINICIÓN**
SkippedTextTrivia	Todo texto que no sea un identificador, operador o palabra clave del lenguaje.
LineBreakTrivia	El final y/o inicio de una línea.

WhiteSpaceTrivia	Los caracteres de espacio en blanco, como las tabulaciones, espacios normales, entre otros.
SinglelineCommenTrivia	Comentarios de una sola línea. Estos inician con ***//***
MultiLineCommentTrivia	Comentarios de más de una línea. Estos inician con ***/**** y terminan con ****/***
TOKENS	
Identifican a todo texto aceptable o no por el lenguaje, y representan la unidad básica de la que se componen las sentencias y expresiones del lenguaje.	
BadToken	Todo texto que no sea un identificador, operador o palabra clave del lenguaje, dependiendo del caso, estos pueden convertirse en "**SkippedTextTrivia**"**.**
EndOfFileToken	Token que representa el final de un archivo.
NumberToken	Cualquier número entero
StringToken	Cualquier texto que este entre comillas ("")
PlusToken	+
MinusToken	-
StarToken	*
SlashToken	/
OpenParentesisToken	(
CloseParentesisToken	)
CommaToken	,
ColonToken	:
OpenBraceToken	{
CloseBraceToken	}
IdentifieToken	Guion bajo '_' o letra seguida de más guiones bajos, letras o números. Ejemplos: • _lol • L_ol1 • Lol_2 • lol
BangToken	!

EqualsToken	=
TildeToken	~
HatToken	^
AmpersandToken	&
AmpersandAmpersandToken	&&
PibeToken	|
PibePibeToken	||
BangEqualsToken	!=
EqualsEqualsToken	==
LessToken	<
LessOrEqualsToken	<=
GreaterToken	>
GreaterOrEqualsToken	>=
PALABRAS CLAVES	
BreakKeyword	break
ContinueKeyword	continue
DoKeyword	do
ElseKeyword	else
FalseKeyword	false
ForKeyword	for
FunctionKeyword	function
IfKeyword	if
LetKeyword	let
returnKeyword	return
ToKeyword	to
TrueKeyword	true
VarKeyword	var
WhileKeyword	while
NODOS	
Por así decirlo, son las unidades básicas del árbol sintáctico, de la cual se construye este.	
CompilationUnit	Unidad sintáctica que identifica a un documento
FunctionDeclaration	Unidad sintáctica que identifica la declaración de una función.
GlobalStatement	Unidad sintáctica que identifica una sentencia fuera de una función.
Parameter	Unidad que identifica a un parámetro de una función.

TypeClause	Clausula utilizada para asignarle un tipo a una función, parámetro o variable.
ElseClause	Cláusula que identifica lo que sucede si no se cumple la condición de un if.
EXPRESIONES	
Las expresiones del lenguaje, estas son obtenidas en el analizador sintáctico, y luego evaluadas en el analizador semántico. Por cada **expresión sintáctica (SyntaxExpression)** existe una **expresión enlazada (BoundExpression)** con el mismo nombre en el analizador semántico.	
LiteralExpression	Un valor literal, que no proviene de una operación o llamada de función.
NameExpression	La llamada a una variable.
UnaryExpression	Una operación con un operador unario.
BinaryExpression	Una operación con un operador binario.
ParenthesizedExpression	Cualquier expresión entre paréntesis.
AssignmentExpression	Una expresión de asignación.
CallExpression	Una expresión de llamada a una función.
ErrorExpression	Una expresión utilizada por el analizador semántico para indicar que hay una operación no valida.
ConversionExpression	Una "**CallExpression**" que le indica al analizador semántico que hay una conversión de tipos, dependiendo de si esta operación es válida o el tipo devuelto por la conversión es aplicable, puede o no convertirse en un "**ErrorExpression**".
SENTENCIAS	
Las sentencias del lenguaje, estas son obtenidas en el analizador sintáctico, y luego evaluadas en el analizador semántico, por cada sentencia sintáctica (**SyntaxStatement**) existe una sentencia enlazada (**BoundStatement**) en el analizador semántico.	

BlockStatement	Una secuencia de sentencias entre llaves ‘{}’.
VariableDeclarationStatement	Declaración de una variable.
IfStatement	Sentencia ‘if’.
WhileStatement	Sentencia ‘while’.
DoWhileStatement	Sentencia ‘do{}while’.
ForStatement	Sentencia ‘for’.
BreakStatement	Sentencia ‘break’
ContinueStatement	Sentencia ‘continue’
ReturnStatement	Sentencia ‘return ()’
ExpressionStatement	Una expresión cualquiera que también actúa como sentencia.
LabelStatement	Un label que apunta al inicio de una secuencia de operaciones en el código intermedio.
GotoStatement	Indica que se continuara ejecutando el programa desde el label especificado
ConditionalGotoStatement	Un goto que ira o no al label indicado dependiendo de una condición.
NopStatement	Código no alcanzable/ejecutable en el código intermedio.
OPERADORES UNARIOS	
Identity	PlusToken
Negation	MinusToken
LogicalNegation	BangToken
OnesComplement	TildeToken
OPERADORES BINARIOS	
Addition	PlusToken
Substraction	MinusToken
Multiplication	StarToken
Division	SlashToken
LogicalAnd	AmpersandAmpersandToken
LogicalOr	PibePibeToken
BitwiseAnd	AmpersandToken
BitwiseXor	PibeToken
Equals	EqualsEqualsToken
NotEquals	BangEqualsToken
Less	LessToken
LessOrEquals	LessOrEqualsToken
Greater	GreaterToken

GreaterOrEquals	GreaterOrEqualsToken
TIPO DE SÍMBOLOS	
Identificador en la tabla de símbolos que indica que representa una variable/función en específico.	
Function	Una función.
GlobalVariable	Una variable global (solo usado por el intérprete).
LocalVariable	Una variable local.
Parameter	Un parámetro.
Type	Un tipo de dato.
CLASIFICACIÓN	
Esta parte es especial, generaliza todo el contenido de la tabla de símbolos a algo fácil de comprender para el coloreo de sintaxis.	
Text	Todo token que no entre en las siguientes clasificaciones.
Keyword	Todo token que represente una palabra clave.
Identifier	Todo token que represente una variable o parámetro.
Number	Todo token que represente un numero
String	Todo token que represente un string.
Comment	Comentarios de una o varias líneas.
Operator	Todo operador binario o unario.

3.1.3. IMPLEMENTACIÓN DEL ANÁLISIS LÉXICO

Nuestra implementación del analizador léxico es a código puro. En la próxima imagen se puede ver la cabecera de nuestro analizador léxico.

```
3 references
internal sealed class LexicAnalyzer
{
    private readonly DiagnosticBag _diagnostics = new();
    private readonly SyntaxTree _syntaxTree;
    private readonly SourceText _text;
    private int _position;

    private int _start;
    private SyntaxKind _kind;
    private object? _value;
    private readonly ImmutableArray<SyntaxTrivia>.Builder _triviaBuilder = ImmutableArray.CreateBuilder<SyntaxTrivia>();

    2 references
    public LexicAnalyzer(SyntaxTree syntaxTree)
    {
        _syntaxTree = syntaxTree;
        _text = syntaxTree.Text;
    }
    2 references
    public DiagnosticBag Diagnostics => _diagnostics;
    23 references
    private char Current => Peek(0);
    5 references
    private char Lookahead => Peek(1);
    2 references
    private char Peek(int offset)
    {
        var index = _position + offset;
        if (index >= _text.Length)
            return '\0';
        return _text[index];
    }
```

Cabecera del analizador léxico.

Este cuenta con:

- **DiagnosticBag:** un objeto que se encarga de administrar los errores encontrados en los analizadores (mejor explicado en el punto 3.5 de manejo de errores).

- **SyntaxTree:** que representa el árbol sintáctico en el analizador sintáctico que quiere un token.

- **SourceText:** que representa un documento, la posición actual en el documento.
- **_start:** que representa el inicio de un token.

- **_kind:** que representa el tipo del token.

- **_value** que representa su valor.

- **_triviaBuilder:** que aloja a la trivia que pertenece al token.

Y, finalmente, los buffers:

- **Current:** representa el carácter actual que estamos revisando.
- **Lookahead:** es un buffer que apunta al siguiente carácter.
- **Peek:** es un buffer que puede apuntar a x caracteres por delante del carácter actual, este sirve de base para los 2 anteriores.

La próxima imagen muestra la función por la cual obtenemos un token del analizador léxico, lo primero que busca, es identificar la trivia anterior al token, luego al mismo token y finalmente la trivia después de este token, finalmente le pregunta a la clase **SyntaxFacts** si este token tiene texto registrado (solo los símbolos y palabras claves tienen texto único), si no lo tiene, entonces toma el texto que representa al token en el documento original como el texto que representa al token, y entonces retorna un token con todos los datos encontrados.

```
2 references
public SyntaxToken Lex()
{
    ReadTrivia(leading: true);

    var leadingTrivia = _triviaBuilder.ToImmutable();
    var tokenStart = _position;

    ReadToken();
    var tokenKind = _kind;
    var tokenValue = _value;
    var tokenLength = _position - _start;

    ReadTrivia(leading: false);
    var trailingTrivia = _triviaBuilder.ToImmutable();

    var tokenText = SyntaxFacts.GetText(tokenKind);
    if (tokenText == null)
        tokenText = _text.ToString(tokenStart, tokenLength);
    return new SyntaxToken(_syntaxTree, tokenKind, tokenStart, tokenText, tokenValue, leadingTrivia, trailingTrivia);
}
```

Función para obtener un token del analizador léxico.

Como se muestra en la próxima imagen, "**ReadTrivia**" es un método que lee el texto original mientras aun exista trivia que leer, una vez identifique una parte del texto que no lo sea, entonces este se detiene, la trivia acumulada será vaciada en una variable y pasada al token, como se muestra en la imagen anterior.

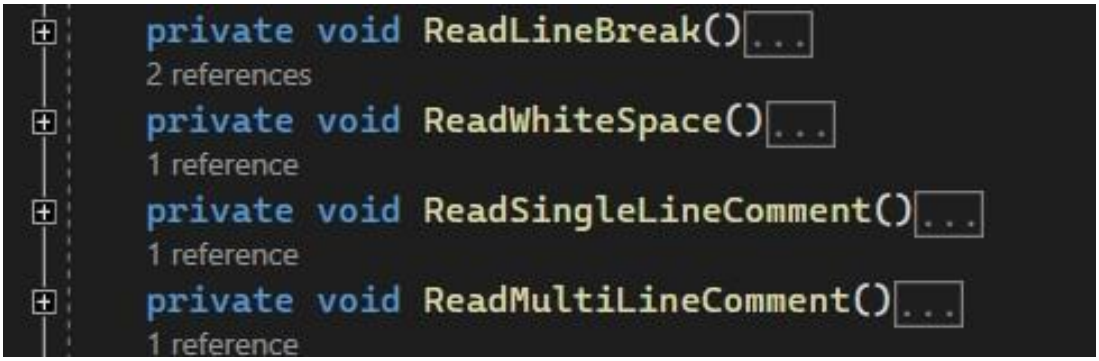
```
2 references
private void ReadTrivia(bool leading)
{
    _triviaBuilder.Clear();
    var done = false;

    while (!done)
    {
        _start = _position;
        _kind = SyntaxKind.BadToken;
        _value = null;

        switch (Current)
        {
            case '\0':
                done = true;
                break;
            case '/':
                if (Lookahead == '/')
                {
                    ReadSingleLineComment();
                }
                else if (Lookahead == '*')
                {
                    ReadMultiLineComment();
                }
                else
                {
                    done = true;
                }
                break;
            case '\n':
            case '\r':
                if (!leading)
                    done = true;
                ReadLineBreak();
                break;
            case ' ':
            case '\t':
                ReadWhiteSpace();
                break;
            default:
                if (char.IsWhiteSpace(Current))
                {
                    ReadWhiteSpace();
                }
                else
                {
                    done = true;
                }
                break;
        }
```

Método “ReadTrivia”.

Los métodos “**ReadLineBreak**”, “**ReadWhiteSpace**”, “**ReadSingleLineComment**” y “**ReadMultiLineComment**” son métodos llamados por “**ReadTrivia**” que le dice cuanto del texto original pertenece a la trivia en específico que está leyendo.



```
private void ReadLineBreak()...
2 references
private void ReadWhiteSpace()...
1 reference
private void ReadSingleLineComment()...
1 reference
private void ReadMultiLineComment()...
1 reference
```

Métodos utlizados por “ReadTrivia”.

El método “**ReadToken**” está compuesto por lo siguiente:

```
1 reference
private void ReadToken()
{
    _start = _position;
    _kind = SyntaxKind.BadToken;
    _value = null;

    switch (Current)...
}
1 reference
private void ReadString()...
1 reference
private void ReadNumberToken()...
2 references
private void ReadIdentifierOrKeywordToken()...
```

Método “ReadToken”.

Primero designa la posición en la que inicia el token como la posición actual en el texto original, y el tipo de token por default a “**BadToken**”, pues aún no sabemos si este es un token valido, y su valor a null. Luego entramos un switch que evalúa al token usando a los buffers (**Current** para los tokens de un carácter y **Current** + “**Lookahead** para los de 2) y cambia el tipo y valor del token según corresponda, para los tokens de tamaño indeterminado están los métodos “**ReadString**”, “**ReadNumberToken**” y “**ReadIdentifierOrKeywordToken**”, los cuales se encargan de determinar hasta cuando continua el token y para el caso del último método, si este es un identificador palabra clave.

Los “**BadToken**” y casos como los comentarios de más de una línea sin terminar (que les falte el ‘*/’ al final) o string sin terminar (que les falte el “ al final), son reportados aquí.

3.2. ANALIZADOR SINTÁCTICO

3.2.1. DEFINICIÓN DE GRAMÁTICAS LIBRES DEL CONTEXTO

EXPRESIONES	
LiteralExpression	<Value>
NameExpression	<IdentifierToken>
UnaryExpression	<Operador><Expresión>
BinaryExpression	<Expresión><Operador><Expresión>
ParenthesizedExpression	<OpenParentesisToken><Expresión><CloseParentesisToken>
AssignmentExpression	<IdentifierToken><EqualsToken><Expresión>
CallExpression	<IdentifierToken><OpenParentesisToken><SeparatedList<Expresión>><CloseParentesisToken>
ConversionExpression	Una "**CallExpression**" donde el <IdentifierToken> es el identificador de un tipo de dato.
SENTENCIAS	
BlockStatement	<OpenBraceToken> <Sentencias> <CloseBraceToken>
VariableDeclarationStatement	<VarKeyword / LetKeyword> <IdentifierToken> <TypeClause><EqualsToken> <Expresión>
IfStatement	<Ifkeyword> <Expresión> <Sentencia> <ElseClause>
WhileStatement	<WhileKeyword> <Expresión> <Sentencia>
DoWhileStatement	<DoKeyword> <Sentencia> <WhileKeyword> <Expresión>
ForStatement	<ForKeyword> <IdentifierToken> <EqualsToken> <Expresion><ToKeyword> <Expresion> <Sentencia>
BreakStatement	<BreakKeyword>
ContinueStatement	<ContinueKeyword>
ReturnStatement	<ReturnKeyword> <Expresión>
ExpressionStatement	<Expresión>
NODOS	
FunctionDeclaration	<FunctionKeyword> <IdentifierToken> <OpenParentesisToken><SeparatedList<Parameter>> <CloseParentesis> <TypeClause><BlockStatement>
GlobalStatement	<Sentencia>
Parameter	<IdentifierToken> <TypeClause>
TypeClause	<ColonToken> <IdentifierToken>
ElseClause	<ElseKeyword><Sentencia>

SeparatedList<T>	Secuencia de tipo T, compuesta por elementos y sus separadores. Solo es usado para parámetros y argumentos, y el separador usado es <ComaToken>.

3.2.2. TIPO DE ANALIZADOR SINTÁCTICO

Nuestro analizador sintáctico es de tipo LR predictivo, pues va asumiendo las siguientes partes de nuestras sintaxis y de no existir, entonces las genera, reporta un error y continua con su evaluación (esto forma parte del mecanismo para el manejo de errores, sección 3.5 para más información).

3.2.3. AUTÓMATA DEL ANÁLISIS SINTÁCTICO

El autómata del analizador sintáctico es una serie de métodos el cual tiene como entrada el árbol sintáctico objetivo obtenido a la hora de llamar al constructor del analizador sintáctico. Este método se llama "**ParseCompilationUnit**", el cual retorna una unidad de compilación con todos los nodos del árbol sintáctico (para nuestro compilador, una unidad de compilación es la raíz de un árbol sintáctico).

```
1 reference
public CompilationUnitSyntax ParseCompilationUnit()
{
    var members = ParseMembers();
    var endOfFileToken = MatchToken(SyntaxKind.EndOfFileToken);
    return new CompilationUnitSyntax(_syntaxTree, members, endOfFileToken);
}

1 reference
private ImmutableArray<MemberSyntax> ParseMembers()...

1 reference
private MemberSyntax ParseMember()...
1 reference
private MemberSyntax ParseFunctionDeclaration()...
1 reference
private SeparatedSyntaxList<ParameterSyntax> ParseParameterList()...
1 reference
private ParameterSyntax ParseParameter()...
1 reference
private MemberSyntax ParseGlobalStatement()...
```

Método "ParseCompilationUnit".

Esta función llama a la función "**ParseMembers**" la cual llama continuamente a la próxima función, y así continuamente hasta haber analizado todos los tokens. Por último, llama al método de manejo de tokens "**MatchToken**" para ver si el ultimo token es un "**EndOfFileToken**", y si no, entonces lo genera.

Como nota especial el método "**ParseStatement**" identifica el tipo de sentencia según el tipo del token actual, como se vio en el apartado

anterior, todas las sentencias tienen un token especifico el cual les da inicio, para el caso de las expresiones también se hace lo mismo.

```
7 references
private StatementSyntax ParseStatement()...
2 references
private BlockStatementSyntax ParseBlockStatement()...
1 reference
private StatementSyntax ParseVariableDeclaration()...
2 references
private TypeClauseSyntax? ParseOptionalTypeClause()...
2 references
private TypeClauseSyntax ParseTypeClause()...
1 reference
private StatementSyntax ParseIfStatement()...
1 reference
private ElseClauseSyntax? ParseOptionalElseClause()...
1 reference
private StatementSyntax ParseWhileStatement()...
1 reference
private StatementSyntax ParseDoWhileStatement()...
1 reference
private StatementSyntax ParseForStatement()...
1 reference
private StatementSyntax ParseBreakStatement()...
1 reference
private StatementSyntax ParseContinueStatement()...
1 reference
private StatementSyntax ParseReturnStatement()...
1 reference
private StatementSyntax ParseExpressionStatement()...
```

```
10 references
private ExpressionSyntax ParseExpression()...
2 references
private ExpressionSyntax ParseAssigmentExpression()...
3 references
private ExpressionSyntax ParseUnaryOrBinaryExpression(int parentPrecedence = 0)...
1 reference
private ExpressionSyntax ParsePrimaryExpression()...
1 reference
private ExpressionSyntax ParseParenthesizedExpression()...
1 reference
private ExpressionSyntax ParseBooleanLiteral()...
1 reference
private ExpressionSyntax ParseNumberLiteral()...
1 reference
private ExpressionSyntax ParseStringLiteral()...
1 reference
private ExpressionSyntax ParseNameOrCallExpression()...
1 reference
private ExpressionSyntax ParseCallExpression()...
1 reference
private SeparatedSyntaxList<ExpressionSyntax> ParseArguments()...
1 reference
private ExpressionSyntax ParseNameExpression()...
```

Diferentes métodos del analizador sintáctico.

3.2.3. IMPLEMENTACIÓN DEL ANÁLISIS SINTÁCTICO

Al igual que el analizador léxico, nuestro analizador sintáctico es un analizador a código puro de dos buffers, uno que apunta al token actual, y uno que apunta a un token x posiciones por delante o por detrás del token actual.

```
private readonly DiagnosticBag _diagnostics = new();
private readonly SyntaxTree _syntaxTree;
private readonly SourceText _text;
private readonly ImmutableArray<SyntaxToken> _tokens;
private int _position;
1 reference
public Parser(SyntaxTree syntaxTree)...
1 reference
public DiagnosticBag Diagnostics => _diagnostics;

5 references
private SyntaxToken Peek(int offset)
{
    var index = _position + offset;
    if (index >= _tokens.Length)
        return _tokens[_tokens.Length - 1];
    if (index < 0)
        return _tokens[0];
    return _tokens[index];
}
29 references
private SyntaxToken Current => Peek(0);
8 references
private SyntaxToken NextToken()
{
    var current = Current;
    _position++;
    return current;
}
36 references
private SyntaxToken MatchToken(SyntaxKind type)
{
    if (Current.Kind == type)
        return NextToken();
    _diagnostics.ReportUnexpectedToken(Current.Location, Current.Kind, type);
    return new SyntaxToken(_syntaxTree, type, Current.Position, null, null, ImmutableArray<SyntaxTrivia>.Empty, ImmutableArray<SyntaxTrivia>.Empty);
}
```

Componentes del analizador sintáctico.

Como se ve en la imagen anterior, el analizador sintáctico está compuesto por:

- **DiagnosticBag**: un objeto que se encarga de administrar los errores.
- **SyntaxTree**: el árbol sintáctico actual que se está analizando.
- **SourceText**: el documento actual.
- **_tokens**: un array de tokens obtenido del analizador léxico.
- **_position**: el token actual que se está analizando.

Buffers:

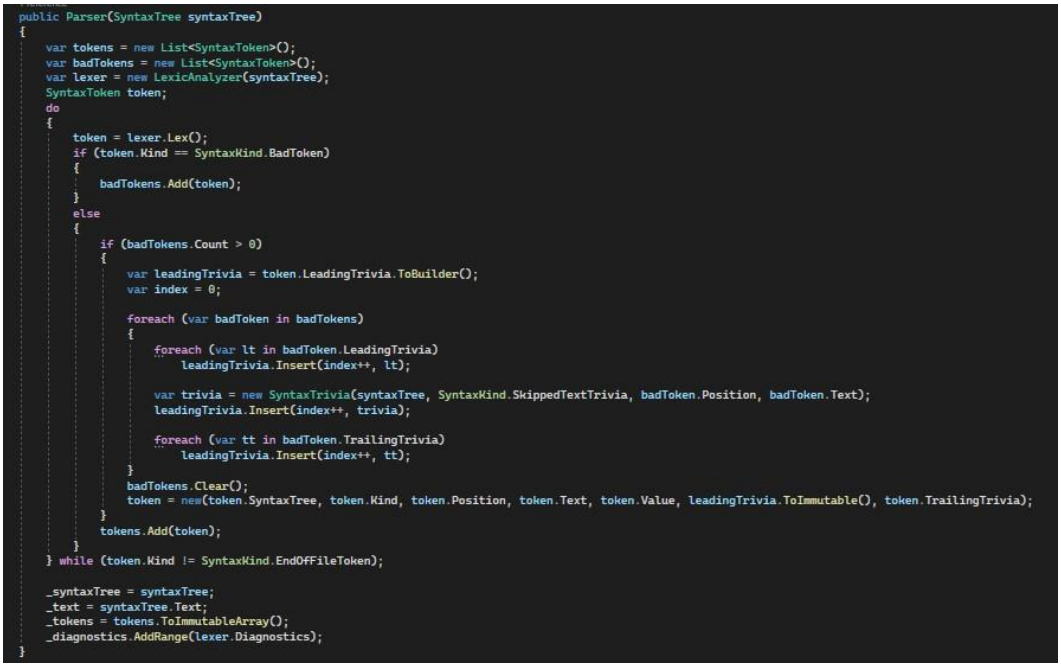
- **Peek:** un buffer que apunta a x tokens por delante o atrás del token actual.
- **Current:** un buffer que apunta al token actual.

Métodos de manejo de tokens:

- **NextToken**: un método que nos retorna el token actual y avanza nuestra posición hacia el próximo token.
- **MatchToken**: un método que compara el tipo de token actual con un tipo de token dado, de ser iguales entonces avanzamos la posición y retornamos el token actual llamando a "**NextToken**", de no serlo reportamos un token inesperado y generamos un token del tipo dado para llenar la posición (Mecanismo de manejo de errores).

"**MatchToken**" es el método principal utilizado para manejar los tokens en el analizador sintáctico y el motivo por el que este solo aumenta la posición actual si los tipos coinciden es debido a que el token que no coincida puede pertenecer a otra estructura gramatical, por lo que se mantiene la posición en este para buscar dicha estructura.

La manera en que se llena el array de tokens es en el constructor del analizador sintáctico, donde este llama continuamente al analizador léxico hasta encontrar todos lo tokens del archivo especificado.



```
public Parser(SyntaxTree syntaxTree)
{
    var tokens = new List<SyntaxToken>();
    var badTokens = new List<SyntaxToken>();
    var lexer = new LexicAnalyzer(syntaxTree);
    SyntaxToken token;
    do
    {
        token = lexer.Lex();
        if (token.Kind == SyntaxKind.BadToken)
        {
            badTokens.Add(token);
        }
        else
        {
            if (badTokens.Count > 0)
            {
                var leadingTrivia = token.LeadingTrivia.ToBuilder();
                var index = 0;

                foreach (var badToken in badTokens)
                {
                    foreach (var lt in badToken.LeadingTrivia)
                        leadingTrivia.Insert(index++, lt);

                    var trivia = new SyntaxTrivia(syntaxTree, SyntaxKind.SkippedTextTrivia, badToken.Position, badToken.Text);
                    leadingTrivia.Insert(index++, trivia);

                    foreach (var tt in badToken.TrailingTrivia)
                        leadingTrivia.Insert(index++, tt);
                }
                badTokens.Clear();
                token = new(token.SyntaxTree, token.Kind, token.Position, token.Text, token.Value, leadingTrivia.ToImmutable(), token.TrailingTrivia);
            }
            tokens.Add(token);
        }
    } while (token.Kind != SyntaxKind.EndOfFileToken);

    _syntaxTree = syntaxTree;
    _text = syntaxTree.Text;
    _tokens = tokens.ToImmutableArray();
    _diagnostics.AddRange(lexer.Diagnostics);
}
```

Método “Parser”.

Algo que realiza al final es recoger los errores que se hayan generado en el analizador léxico, pues estos deben llegar a la salida del programa.

3.3. REGLAS DEL LENGUAJE

Lenguaje: Spark

Palabras reservadas:

- break --> Palabra clave para salir de un bucle.
- continue --> Palabra clave para saltar directamente a la comparación en un bucle.
- do --> Palabra clave usada para iniciar una sentencia do-while.
- let --> declara una variable de solo lectura de cualquier tipo, requiere que se le asigne un valor en su declaración.
- var --> declara una variable de cualquier tipo, requiere que se le asigne un valor en su declaración.
- else --> Condición ‘Si no’ del if.
- false --> palabra clave para el bool false.

- for --> palabra clave para la definición de una declaración for.
- function --> palabra clave para iniciar una funcion.
- if --> palabra clave para la definición de una declaración if.
- return --> palabra clave que indica que termina una función, retornando o no un valor.
- to --> palabra clave para determinar el límite superior de una expresión for.
- true --> palabra clave para el bool true.
- while --> palabra clave para la definición de una declaración while.

Símbolos:

- +
- -
- *
- /
- (
-)
- {
- }
- !
- =
- ~
- <

- <=
- >
- >=
- &
- &&
- |
- ||
- ^
- !=
- ==

Tipos:

- bool.
- int.
- String.

Funciones internas del lenguaje:

- input() --> recibe una entrada del teclado, es equivalente al Console.ReadLine() en C#.
- print(value) --> imprime en consola un valor, es equivalente al Console.WriteLine(value) en C#.
- random(value) --> genera un numero random desde 0 hasta el valor dado, equivalente a crear un objeto de tipo random y llamar a su función en C#.

Declaración de funciones:

Una función se declara de la siguiente forma:

```
function nombre (parametro: tipo): tipo{
    return valor
}
```

Una función puede no tener tipo:

```
function nombre (parametro: tipo){
}
```

Puede no tener parámetros:

```
function nombre (): tipo{
    return valor
}
```

Puede no tener ambas:

```
function nombre (){
}
```

Una función sin tipo puede o no usar la palabra reservada "**return**", pero si tiene tipo, entonces está obligada a usar esta sentencia. Algo a tomar en cuenta a la hora de usar bucles y condicionales en una función, es que todos los caminos deben retornar.

Declaración de variables:

Las variables siempre deben estar inicializadas:

```
var a = 12
```

Estas pueden tener un tipo implícito o explícito:

```
Var a:int = 12 // Declaración de variable tipo int de manera explicita
Var a = 12 // Declaración de variable tipo int de manera implícita.
```

Una variable puede ser dinámica:

```
var a: any = 12 // Variable dinámica iniciada con valor int 12
```

a = “hola mundo!” // le asignamos un valor de tipo string a la variable dinámica.

Sentencias globales:

Las sentencias globales solo están permitidas cuando:

- No hay una función **‘main’** declarada.
- Todas las sentencias globales están en un mismo documento.

Conversiones:

Las únicas conversiones implícitas permitidas son las de cualquier tipo a ‘any’, cualquier otra conversión ha de ser de manera explícita, esto al menos que el valor sea del mismo tipo, en ese caso no es necesaria ninguna conversión:

```
var a:any = 12 // conversión implícita
a = “hola” // conversión implícita
a = true // conversión implícita
var b:string = string(12) // conversión explicita.
b = “hola” // no conversión.
b = string(true) // conversión explicita.
```

Esto también aplica para cualquier tipo de operación.

```
a = 12 * 5 > 12 // conversión implícitamente, pasando el valor resultado de esta expresión.
b = string(12 * 5 > 12) // conversión explicita.
```

Int está basado en el tipo entero de 32 bits System.Int32. Lo cual significa que las operaciones con números están limitadas a las aceptadas por System.Int32.

Conversiones aceptadas:

- int --> string
- bool --> string
- string --> int (Solo si el texto es un número, si no entonces el programa lanzara una excepción).

- string --> bool (Solo si el texto es true o false, no importan las mayúsculas o minúsculas).

- any --> cualquier otro tipo (Se necesita una conversión explicita. Si el dato en any no cumplir con la estructura del dato objetivo, entonces el programa lanzara una excepción).

- Cualquier otro tipo --> any

También existe unas conversiones triples, las cuales sirven para saltar la limitación de no poder convertir int a bool ni bool a int. Estas son las siguientes:

- int --> any --> bool

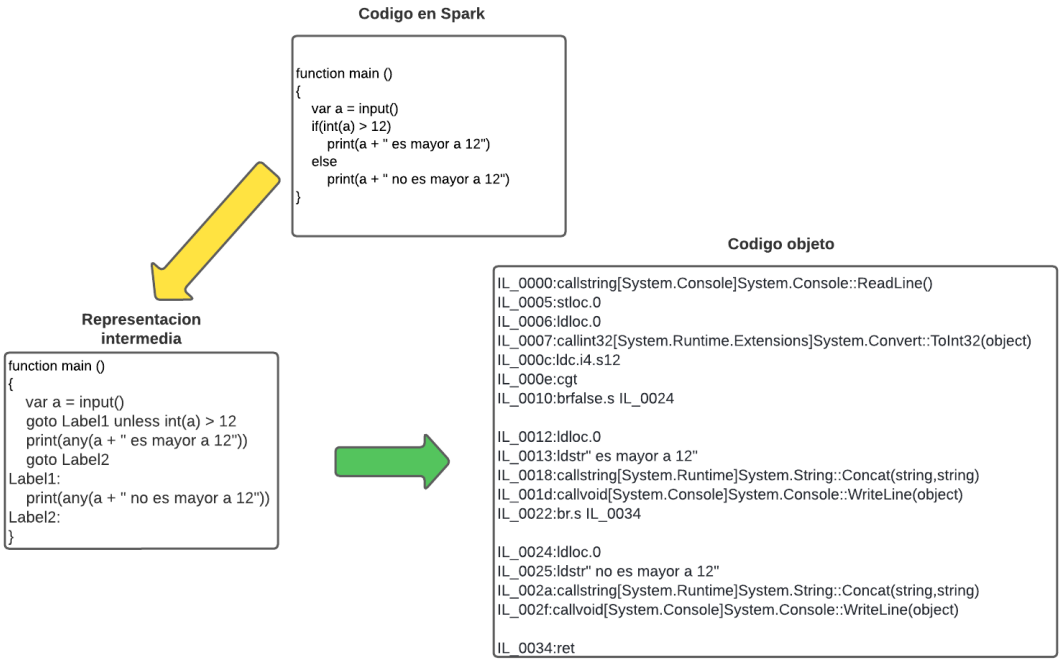
- bool --> any --> int

El lenguaje no tiene forma para manejar excepciones, lo que lo hace muy volátil, tomar en cuenta esto a la hora de programar en él.

3.4. GENERADOR DE CÓDIGO INTERMEDIO

La generación de código intermedio es manejada por 2 clases, "**BoundTreeRewriter**" y "**Lowerer**", lo que hacen estas clases es tomar la expresiones y sentencias en el "**BounTree**" que nos trae como resultado el analizador semántico, no todos los nodos del "BoundTree" son reescritos, ya que, como se ve en la próxima imagen, algunas sentencias conservan su misma forma en la representación intermedia, pues son lo suficientemente explicitas como para generar el código objeto.

Sentencias que si son reescritas son las llamadas a la función interna "print", la cual acepta un valor de tipo "any", al ser una conversion implicita, no es necesario escribirla en Codigo Spark, pero el compilador la realiza en la representacion intermedia. Otros elementos que son reescritos son los bucles for, while y do-while, además de la sentencia if-else, las cuales son reducidas a una serie de sentencias, gotos, conditionalgotos y labels, que más adelante simplifican su conversión a código objeto.



Proceso de la generación de código intermedio.

Como puede ver, aunque la estructura del código objeto sea totalmente distinta a la del código en Spark, si mantiene cierto parentesco con la representación intermedia, facilitando identificar que parte pertenece a que expresión o sentencia.

3.5. MANEJO DE ERRORES

El manejo de errores se divide en 2:

- **Reportado**: todas las fases del compilador tienen una "**DiagnosticBag**" con la cual reúnen los errores de fases anteriores y agregan sus propios errores, antes de compilar se pregunta si ha habido errores, de haberlos entonces no compila y los errores se le muestran al programador.

- **Recuperación**: las fases de análisis tienen la capacidad de recuperarse de un error, el analizador léxico lo hace reportando dichos errores como "**BadToken**", el analizador sintáctico lo hace generando los tokens faltantes para poder seguir con el análisis, el analizador semántico lo hace cambiando la parte de la estructura sintáctica por un "**ErrorExpression**", toda operación que contenga un "**ErrorExpression**" se convierte en otro "**ErrorExpression**" y se continúan con las demás.

La fase de compilación no necesita métodos de recuperación, pues si hay errores no hay motivos para compilar, por lo que simplemente se cancela. Todo error encontrado es mostrado al programador en una lista, conteniendo su ubicación e información sobre cuál es el error.

3.6. OPTIMIZACIÓN DE CÓDIGO A COMPILAR

El compilador hace 3 tipos de optimizaciones antes de compilar:

- **ConstantFolding:** reduce expresiones a valores literales si estas están compuestas por valores literales. Ejemplo:

  a = 12 * 2 --> a = 24

- **Eliminación de código muerto:** elimina el código no alcanzable/ejecutable del programa, esto ocurre luego del **constantfolding**, pueden ser sentencias y expresiones dentro de un bucle que nunca se ejecuta, luego de un return, etc.

- **ConstantFolding para strings**: esto lo hace justo antes de generar el código objeto, el motivo por el que se hace al final es para facilitar su tratamiento, ya que para realizar esto se debe aplanar toda la operación que se estén haciendo con los strings.

3.7. CREACIÓN DEL PROYECTO

3.7.1. PASOS PARA CREAR EL PROYECTO

1. **LexicAnalyzer**: Realizamos el analizador léxico, y denotamos los tokens que serían aceptados por el lenguaje.

2. **Parser**: Realizamos el analizador sintáctico, para que sea capaz de identificar la estructura del programa, con esto empezamos la "**CompilationUnit**", que también nos ayudó a poder trabajar con más de un archivo.

3. **Diagnostics**: Se hizo el manejador de errores "**DiagnosticBag**", el objeto "**SourceText**" y varios más para facilitar lidiar con errores y leer el texto de entrada.

4. **Binder:** El analizador semántico fue lo siguiente, seguido del "**BoundTreeRewriter**" y "**Lowerer**" para realizar la representación intermedia del programa, con el "**Lowerer**" y una clase más llamada

"**BoundProgram**" logramos hacer que se junten todos los archivos para realizar la compilación.

5. **Emit**: Realizamos el "**Emitter**" la clase que se encarga de escribir el programa objeto y una aplicación de consola llamada "**spksc**" que se encarga de obtener las referencias y juntar los archivos para realizar la compilación, con esto también se agregó un nuevo archivo para los programas, el cual es un ".**spksproj**" en el cual se identifica el framework de la app y también apunta a la ubicación de "**spksc**" para poder ser compilado.

6. **GUI**: Se creo la interfaz de usuario, empezando primero por el coloreado de las palabras, a la hora de escribir código, seguido por un objeto el cual se encarga de recolectar los archivos y otro que se encarga de leerlos para buscar los errores.

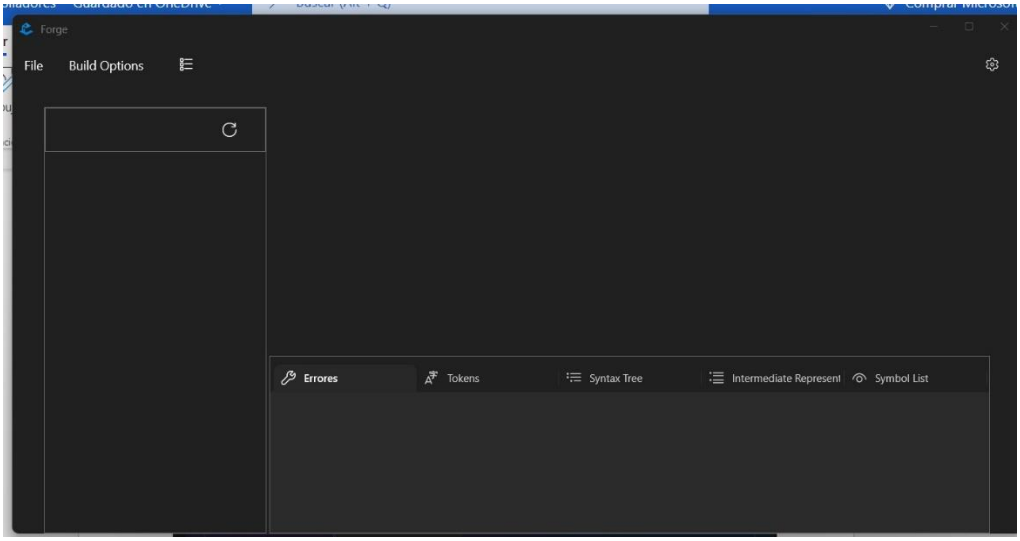
Lo último realizado para esto fue la capacidad de crear y abrir los proyectos, ya sea que contengan múltiples carpetas o no, y los botones "**Build**", "**Deploy**" y "**Build and Deploy**" para compilar y correr la aplicación.

3.7.2. EXPLICACIÓN DEL FUNCIONAMIENTO DEL PROYECTO

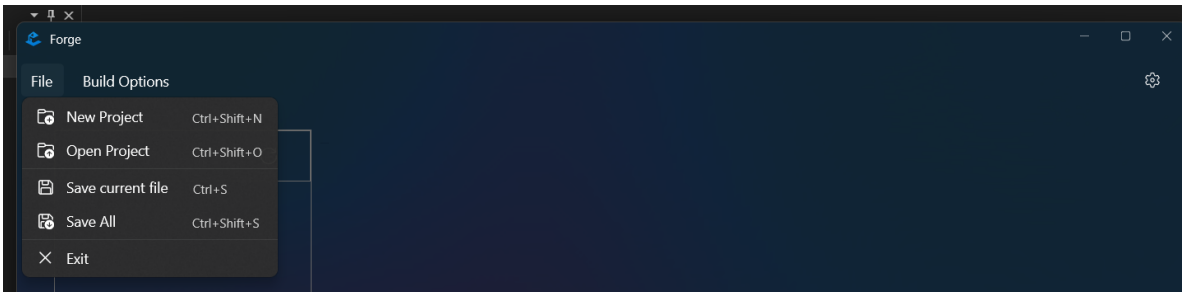
A la hora de abrir la aplicación, lo primero que se debe hacer es entrar al submenú de "**File**" y seleccionar la opción de "crear nuevo proyecto" o "abrir proyecto". Si se elige la opción de crear un proyecto, entonces se le pedirá elegir un nombre y ubicación en la que se alojará, si se elige la opción, se le pedirá buscar un archivo ".**spksc**" para obtener el proyecto. Los documentos aparecerán en un panel a la izquierda, donde se verán los folder y archivos que componen el proyecto. Si se quiere editar uno de estos documentos, entonces se le deberá dar doble clic para abrirlo. No se permite modificar folders ni borrar o crear archivos desde la app, para esto usar el explorador de archivos normal y luego presiona el botón "**Refresh**" en el panel izquierdo para cargar los cambios.

Una vez modificado, guardar los cambios y elegir una de las "**Build Options**", ya sea "**Build**" para compilar, "**Deploy**" para correr la aplicación, si es que ya está compilada, o "**Build and Deploy**" para compilar y luego correr la aplicación.

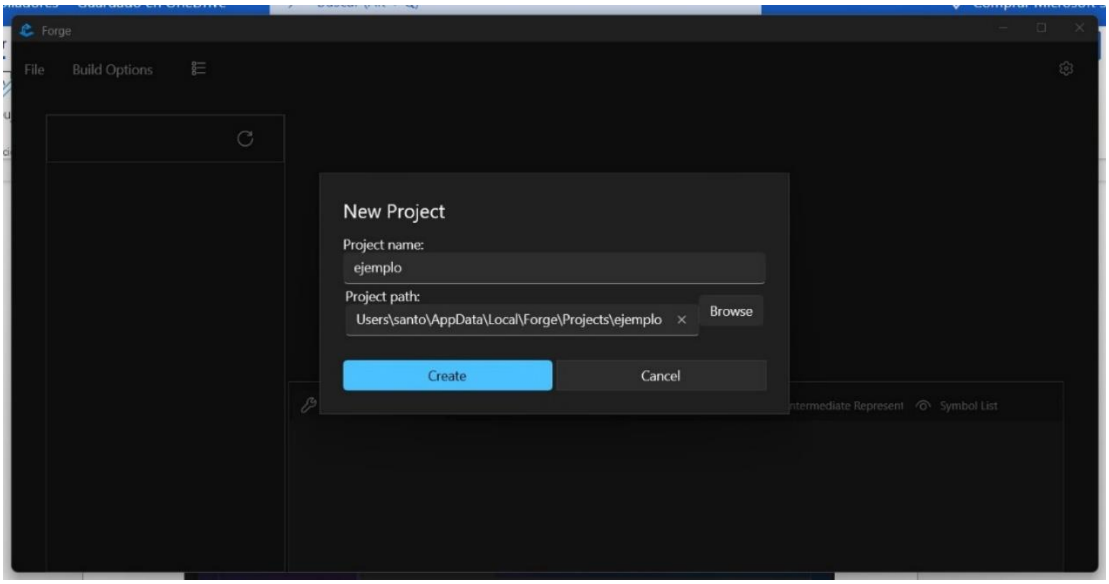
3.7.3. DESPLIEGUE DE LA APLICACIÓN



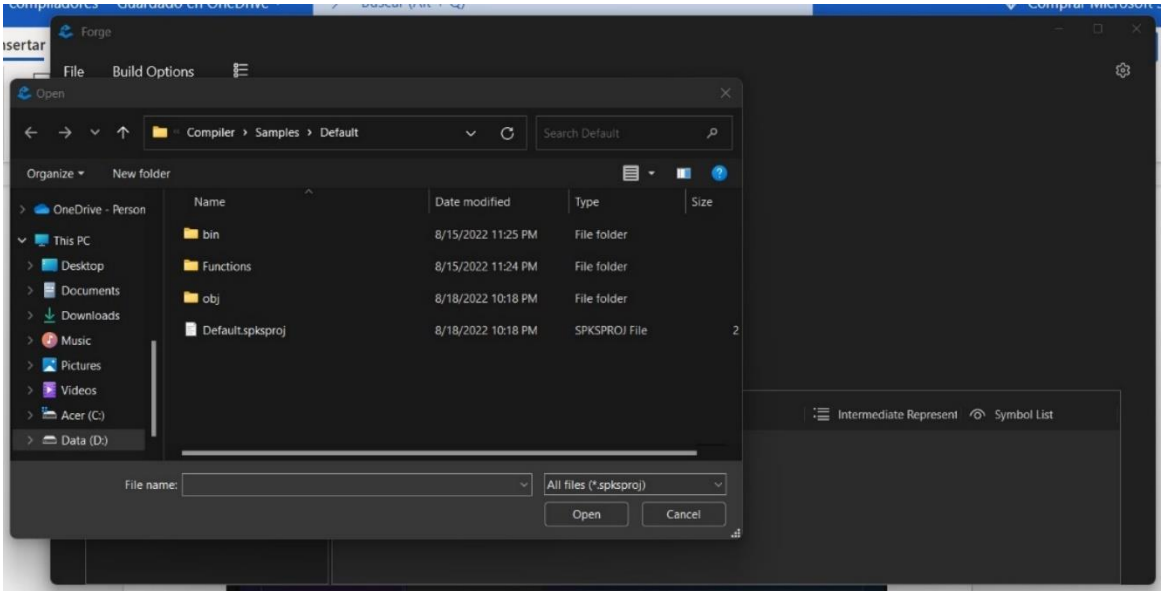
Despliegue del compilador.



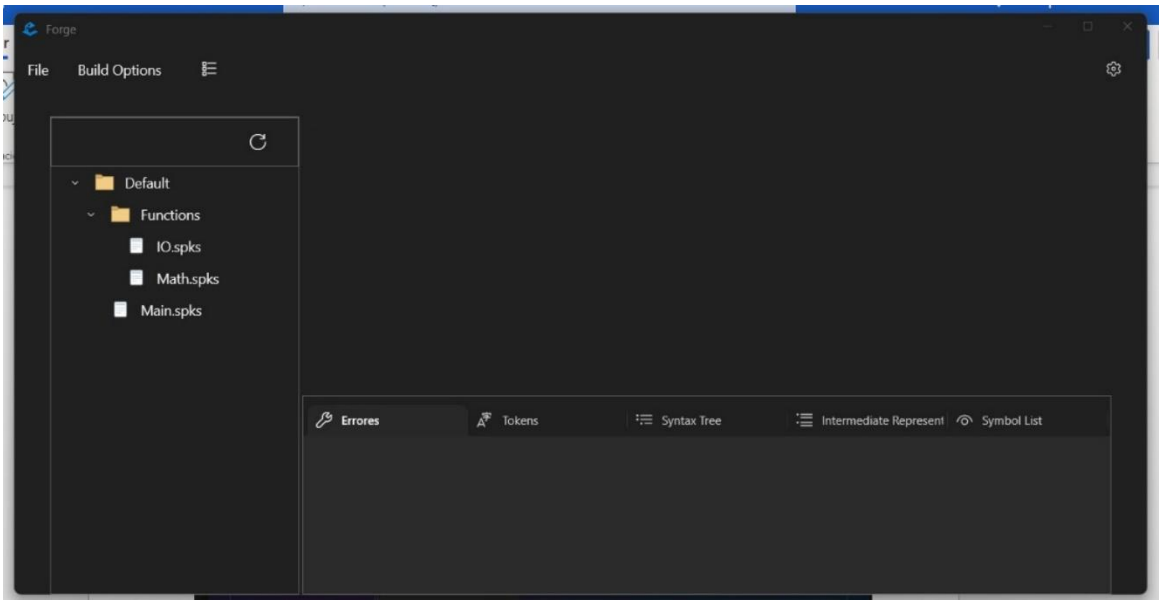
Menú “File”.



Crear nuevo proyecto.

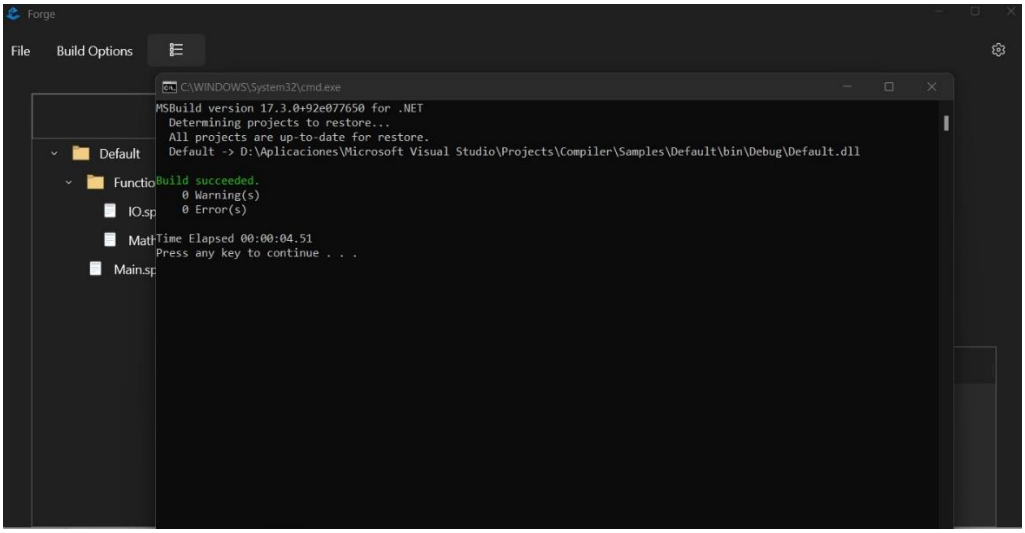


Abrir proyecto.

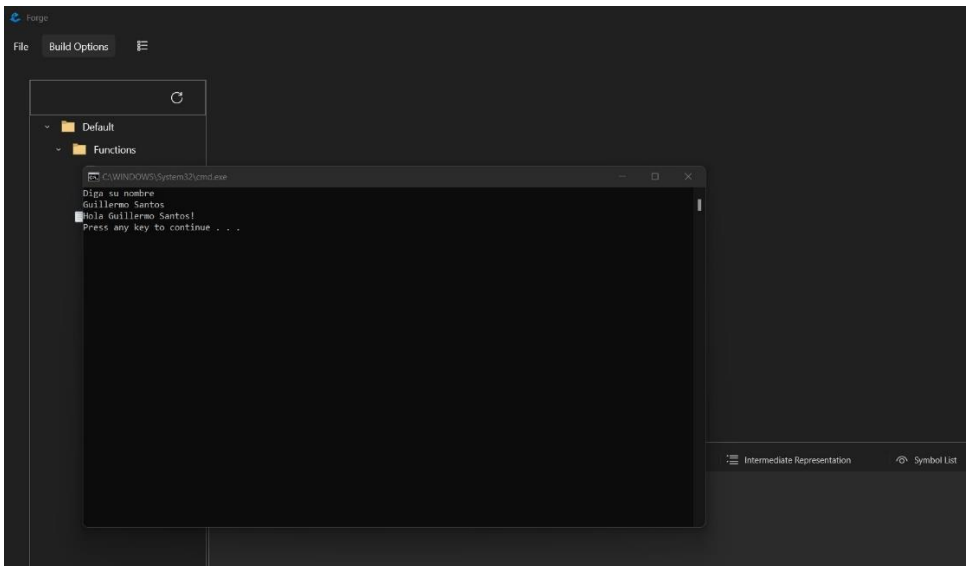


Un proyecto abierto en la aplicación.

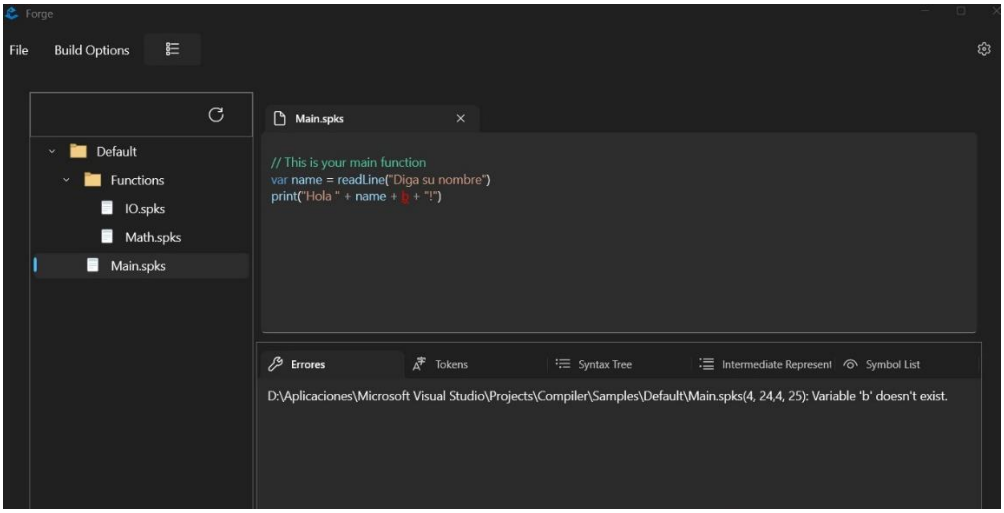
Menú “Build options”.



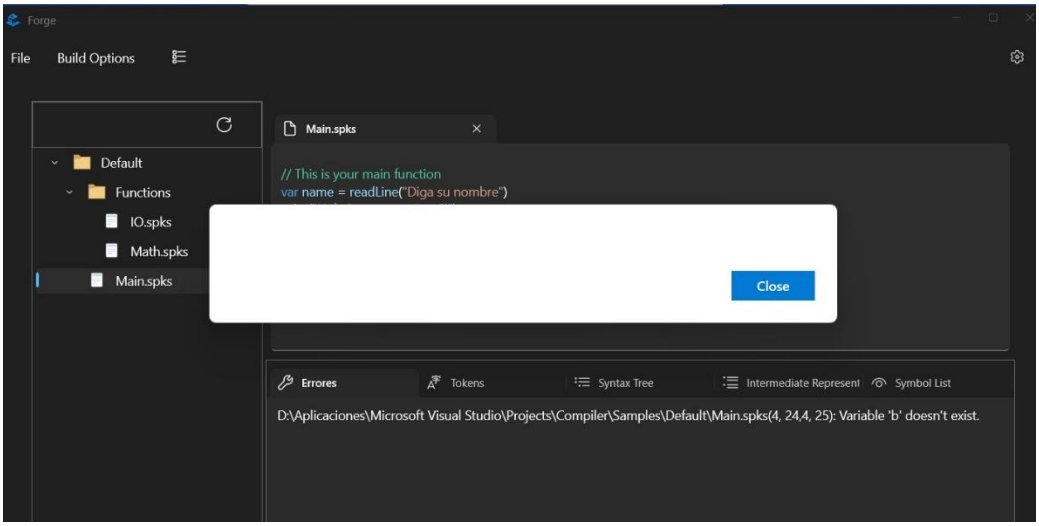
Compilación exitosa.



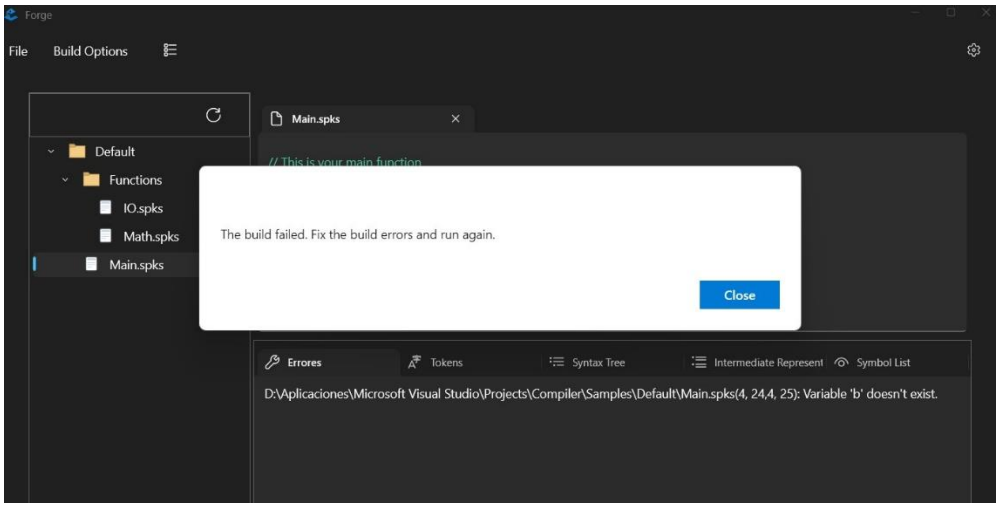
Despliegue del código objeto.



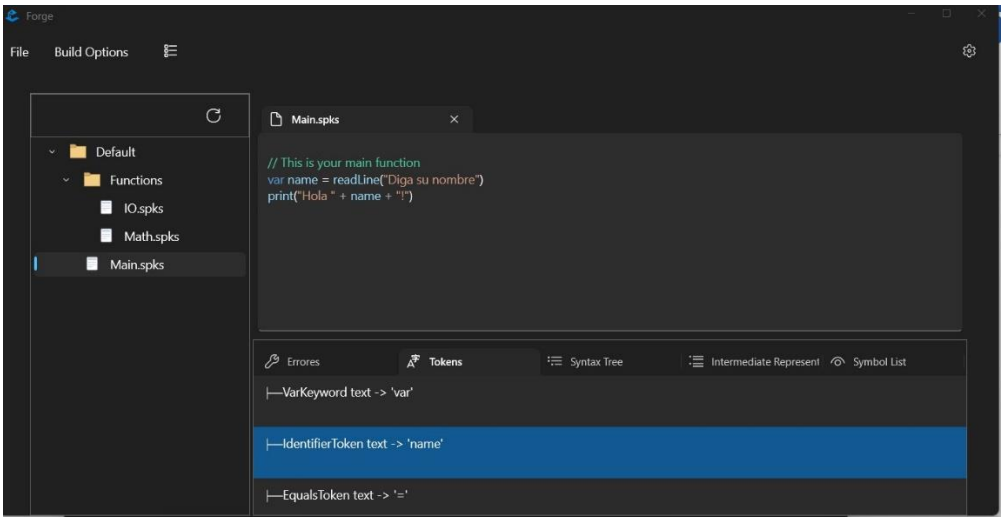
Mostrar errores.



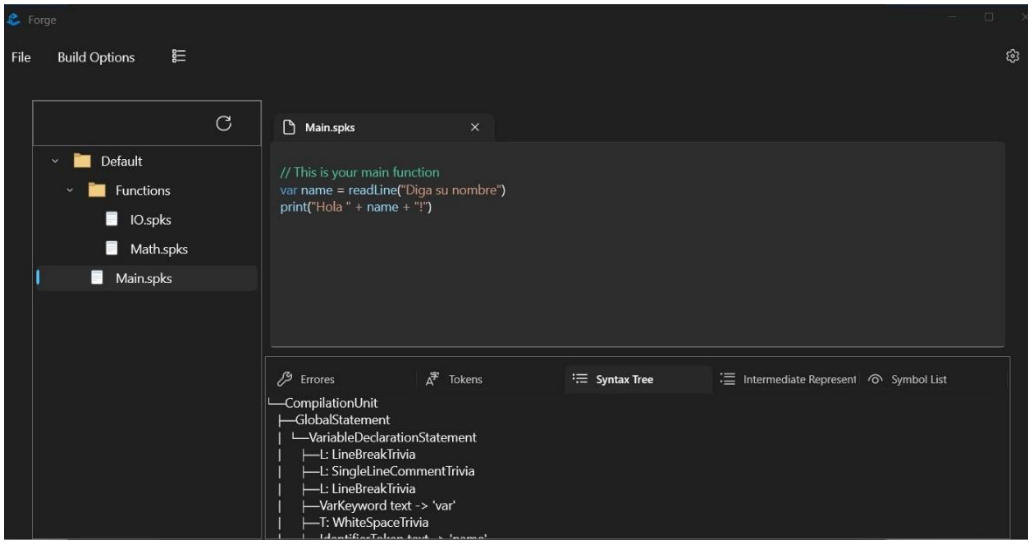
Compilación con errores (Sin mensaje de error, pues el problema ya se muestra en la ventana de errores).



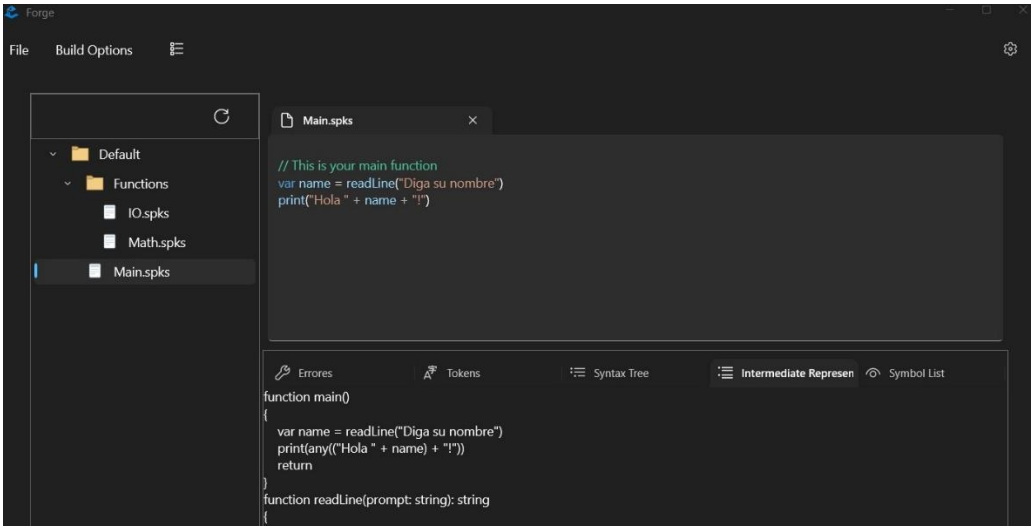
Despliegue con errores.



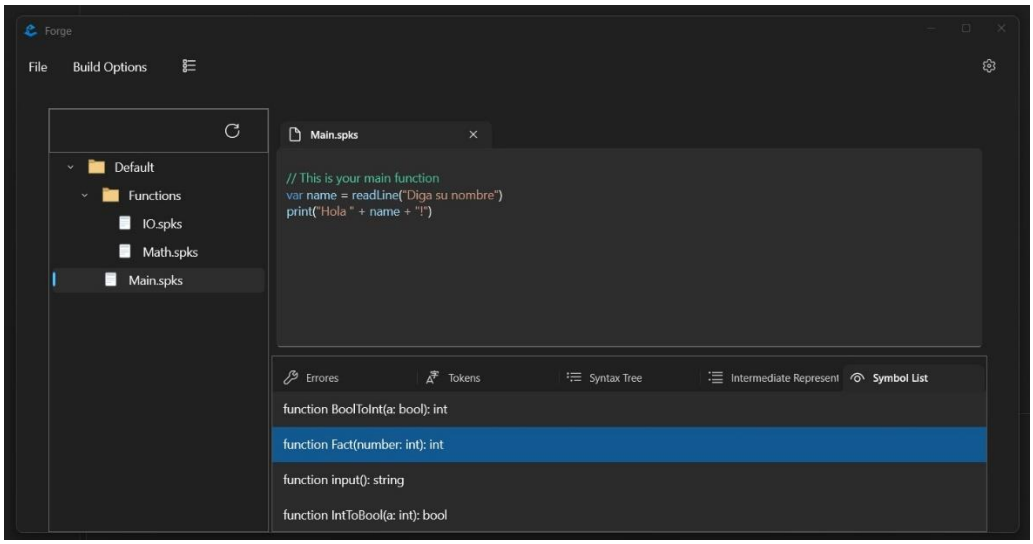
Tokens del documento abierto.



Árbol sintáctico del documento abierto.



Representación intermedia (todo el programa).



Lista de símbolos (todo el programa).

Configuración.

CONCLUSIÓN

Un compilador es una herramienta compleja para el desarrollo de programas, por lo que su demanda en cuanto a capacidad ha ido creciendo de forma constante, desde poder visualizar errores, la capacidad de depurar programas, entorno de equipos, etc. Esta es una tecnología que crece en conjunto con las capacidades de los lenguajes de programación, y esperamos que sigan trayendo herramientas útiles para el desarrollo.

Con nuestro compilador, mostramos un mínimo de lo que debe encontrarse en compiladores actuales, así que esperamos que esto nos sirva de experiencia en caso de que tengamos que darnos a la tarea de realizar uno nuevamente.

BIBLIOGRAFÍA

- **ILSpy:** ILSpy - Aplicaciones de Microsoft Store (Necesario para ver el código objeto).
- **Mono.Cecil:** NuGet Gallery | Mono.Cecil 0.11.4
- **Galería de controles WinUI 3:** WinUI 3 Gallery - Aplicaciones de Microsoft Store
- **Galería del MVVM Toolkit:** MVVM Toolkit Sample App - Microsoft Store Apps
- **WinUI 3 Doc:** Windows UI Library (WinUI) 3 - Windows apps | Microsoft Docs
- **Llamar a procesos C#:** Process Class (System.Diagnostics) | Microsoft Docs+
- **CMD pause:** pause | Microsoft Docs
- **Let's build a compiler series:** (2) Building a Compiler - YouTube
- **Link del proyecto en GitHub:** TheCompiler (github.com)