Universidad
Católica del Norte

Creación de un Compilador para un Lenguaje Propio (Frankenstein)

Taller de Compiladores

Fecha de Entrega:

17 de Junio de 2025

Autores:

- Benjamín Andrés Garces Zarate
- Pablo Andrés Jorquera Herrera

Índice

1. Introducción

El presente informe detalla el proceso de diseño y construcción de un compilador para un lenguaje de programación propio de alto nivel, como parte de los requisitos del Taller de Compiladores. El objetivo principal fue aplicar los conceptos teóricos de la compilación para crear un programa capaz de analizar, validar y traducir código fuente a un lenguaje de más bajo nivel, en este caso, **C++**.

Para lograr esto, se utilizaron las herramientas estándar de la industria: **Flex** para el análisis léxico y **Bison** para el análisis sintáctico. La lógica interna del compilador, incluyendo la construcción del Árbol de Sintaxis Abstracta (AST), el análisis semántico y la generación de código, se implementó en **C++**, utilizando el estándar C++17.

Este documento está dividido en dos secciones principales. La primera ofrece una descripción técnica detallada del diseño y la implementación de cada fase del compilador. La segunda sección sirve como un manual de usuario completo, explicando la sintaxis y las características del lenguaje desarrollado.

2. Diseño e Implementación del Compilador (Versión Técnica)

Esta sección describe la arquitectura interna y los mecanismos de implementación del compilador, detallando las estructuras de datos y algoritmos utilizados en cada fase del proceso.

2.1 Visión General del Flujo de Compilación

El compilador fue diseñado como un sistema monolítico secuencial que procesa el código fuente en cuatro fases distintas. La orquestación de este flujo se centraliza en la función **main** definida en **parser.y**, la cual invoca al parser **yyparse()**. A su vez, **yyparse()** colabora con el lexer **yylex()** y, una vez completado el análisis sintáctico, se inician explícitamente las fases de análisis semántico y generación de código sobre el Árbol de Sintaxis Abstracta (AST) resultante.

2.2. Fase 1: Análisis Léxico

El analizador léxico se implementó con **Flex**, con sus reglas definidas en **lexer.l**. Su principal responsabilidad es la tokenización del flujo de caracteres de entrada.

- **Seguimiento de Ubicación**: Se utilizó la opción **%option yylineno** para que Flex mantenga un registro automático del número de línea actual, información crucial para el reporte de errores en fases posteriores.
- **Transmisión de Valores**: Para comunicar los valores de los tokens al parser, se empleó la unión **yylval** definida en **parser.y**. Por ejemplo, al reconocer un número, su valor se convierte a entero mediante **atoi(yytext)** y se almacena en **yylval.ival**. Para identificadores y literales de cadena, se duplica la cadena con **strdup(yytext**) y se asigna a **yylval.sval** para evitar problemas de gestión de memoria.

- **Procesamiento de Literales**: Se implementó una lógica específica para los literales de cadena (**STRING_LITERAL**) que procesa secuencias de escape como **\n** y **\t**, reemplazándolas por sus caracteres correspondientes antes de pasar el valor al parser.
- **Reglas de Exclusión**: Se definieron reglas explícitas para descartar componentes no semánticos del lenguaje, como los espacios en blanco y los comentarios de una sola línea (//...), que son identificados y simplemente ignorados por el lexer.

2.3. Fase 2: Análisis Sintáctico y Construcción del AST (Versión Detallada)

El analizador sintáctico, o parser, implementado con **Bison** en **parser.y**, es el núcleo del front-end del compilador. Recibe la secuencia de tokens del lexer y verifica si la estructura del código es gramaticalmente correcta.

- **Gramática del Lenguaje**: Se especificó una gramática libre de contexto que establece las reglas de sintaxis del lenguaje. Esto incluye las reglas para declaraciones de variables, sentencias de control, expresiones y definición de funciones. Se definió la precedencia y asociatividad de los operadores para resolver ambigüedades.
- **Acciones Semánticas**: A medida que el parser reconoce una regla gramatical, ejecuta una acción en **C++** para construir una porción del **AST**. Estas acciones invocan a las funciones de fábrica (ej. **createBinaryOpNode**) definidas en **Ast.cpp**.

El producto de esta fase es el **Árbol de Sintaxis Abstracta (AST)**, la estructura de datos central que representa la lógica del programa.

Estructura de Datos del AST

El AST se diseñó como un árbol heterogéneo, donde cada nodo es de un tipo diferente según la construcción del lenguaje que representa. Todas las definiciones se encuentran en **Ast.h**.

- **A. El Nodo Base (Node):** Todos los nodos del árbol comparten una estructura base común que contiene información esencial:

```
struct Node { //Estructura generica y fundamental
              //(Todos los nodos especificos comparten los campos de este)
    NodeType type; //Guarda la etiqueta del nodo
    Node *next; //Puntero hacia el nodo siguiente
    int line; //Campo que guarda el numero de linea donde se encuentra
              //(Util para reportar errores)
    SymbolBasicType calculatedType; //Guarda el tipo resultante de una expresion
};
```

- **B. Nodos Especializados:** Para cada construcción sintáctica, se definió un **struct** específico que, además de los campos base, contiene punteros a sus nodos hijos. Esto crea la estructura jerárquica del árbol.

- **Nodo para Operaciones Binarias**: Contiene punteros a sus operandos izquierdo y derecho

```
typedef struct { //OPERACION QUE ACTUA SOBRE DOS VARIABLES
    NodeType type;
    Node *next;
    int line;
    SymbolBasicType calculatedType;
    int op; //Guarda el token del operador
    Node *left; //Puntero al nodo de la expresion que esta a la izquierda
    Node *right; //Puntero al nodo de la expresion que esta a la derecha
} BinaryOpNode;
```

- **Nodo para Condicionales if**: Contiene punteros para la condición, el bloque **then** y el bloque opcional **else**.

```
typedef struct { //IF o ELSE IF o ELSE
    NodeType type;
    Node* next;
    int line;
    SymbolBasicType calculatedType;
    Node* condition; //Puntero a la expresion booleana de la condicion
    Node* thenBranch; //Puntero al StatementListNode del bloque then
    Node* elseBranch; //Puntero al statement del bloque else
                      //(Tambien puede guardar un IfNode para generar un else if)
} IfNode;
```

Un desafío de diseño importante durante la construcción del **AST** fue el manejo de las declaraciones de múltiples variables (ej. **string a, b;**). Inicialmente, se utilizaba un único nodo de tipo (**NODE_TYPE**) compartido para todas las variables de la misma declaración, lo que llevaba a un error crítico de `double-free` durante la fase de liberación de memoria. Para solucionar esto de manera robusta, se implementó una función auxiliar, **duplicateTypeNode**, en **Ast.cpp**. Ahora, durante el análisis sintáctico, en lugar de compartir un puntero, se invoca a esta función para crear una copia única del nodo de tipo para cada variable declarada. Este enfoque garantiza que el **AST** mantenga una estructura de árbol estricta en este aspecto, permitiendo una de-alocación segura y predecible, y eliminando por completo el riesgo de corrupción de memoria.

2.4. Fase 3: Análisis Semántico

Esta fase verifica la coherencia lógica del AST construido.

- **Tabla de Símbolos:**

La implementación, encontrada en symbol_table.h y symbol_table.cpp, utiliza una estructura de datos **std::vector<std::unordered_map<std::string, SymbolTableEntry_CPP>>**. Esto funciona como una pila de tablas hash, donde cada tabla representa un ámbito anidado, permitiendo un manejo eficiente de los scopes.

El proceso de búsqueda de un símbolo (**lookupSymbol**) itera esta pila en orden inverso (**usando rbegin()**), lo que garantiza que las variables locales (en los ámbitos más internos) oculten a las globales (en los ámbitos más externos), implementando así el shadowing de variables.

Para interactuar con Bison (basado en C), se creó una fachada de C sobre la clase C++ **SymbolManager**. Funciones como **c_insert_variable y c_lookup_symbol** actúan como puentes, permitiendo que el código del parser manipule la tabla de símbolos orientada a objetos.

- **Verificador de Tipos:**

El núcleo de esta fase es la función recursiva **checkTypesInNodeRecursive**, que implementa un recorrido en profundidad (post-orden) sobre el AST.

Para las expresiones, la función **getAndCheckType** determina el tipo resultante de una operación. Por ejemplo, al encontrar un **BinaryOpNode**, invoca recursivamente a **getAndCheckType** sobre sus hijos izquierdo y derecho, y luego determina el tipo resultante basándose en el operador y los tipos de los operandos (ej. **int + float resulta en float**).

Un aspecto clave del diseño es la anotación del AST. A medida que el verificador determina el tipo de un nodo de expresión, almacena este tipo en el campo **calculatedType** del propio nodo. Esta información es luego utilizada por fases posteriores como la generación de código.

Para cada tipo de nodo de sentencia, se aplican reglas específicas. Por ejemplo, para un **IfNode**, se comprueba que el tipo de su nodo **condition**, obtenido a través de **getAndCheckType**, sea **SYM_TYPE_BOOL**.

2.5. Fase 4: Generación de Código

Una vez que el AST ha sido validado semánticamente, el generador de código, implementado en code_generator.cpp, lo traduce a código C++.

- **Estrategia de Buffering**: Se optó por un diseño que utiliza dos stringstream principales: **functions_buffer** y **main_buffer**. Al recorrer el AST, las definiciones de funciones (**NODE_FUNCTION_DEF**) se escriben en el primer buffer, mientras que el resto de las sentencias del ámbito global se escriben en el segundo. Al final, se ensambla el archivo final escribiendo primero todos los includes necesarios, luego el contenido del **functions_buffer**, y finalmente el **main_buffer** dentro de una función **int main() {}** de **C++**. Esta estrategia asegura que todas las funciones estén declaradas antes de ser llamadas en el main.
- **Traducción Recursiva:** La generación se realiza mediante un conjunto de funciones recursivas, principalmente **generate_statement_code** y **generate_expression_code**, que se especializan en traducir sentencias y expresiones, respectivamente.
- **Mapeo de Tipos:** Una función de ayuda, **map_type_to_cpp**, se encarga de traducir el enum interno **SymbolBasicType** (ej. **SYM_TYPE_STRING**) a su cadena de texto equivalente en **C++** (**std::string**), facilitando la declaración de variables y funciones en el código de salida.
- **Traducción de Nodos Complejos:** La traducción de nodos como **ForNode** demuestra el proceso en acción: se realizan llamadas separadas a **generate_expression_code** para las secciones de inicialización, condición e incremento, y a **generate_statement_code** para el cuerpo, mientras se formatea la salida de texto para que coincida con la sintaxis **for(...) {...}** de **C++**.

3. Manual de Usuario del Lenguaje

3.1. Introducción al Lenguaje

Este es un lenguaje de programación imperativo, de tipado estático y propósito general, diseñado para ser intuitivo y fácil de aprender. Su sintaxis es similar a la de lenguajes como C++ y JavaScript, pero con algunas simplificaciones y características únicas.

3.2. Tipos de Datos y Declaración de Variables

El lenguaje soporta cuatro tipos de datos básicos. Todas las variables deben ser declaradas antes de su uso.

- **int**: Números enteros (ej. 10, -5).
- **float**: Números de punto flotante (ej. 3.14, -0.5).
- **string**: Cadenas de texto (ej. “Hola mundo”).
- **bool**: Valores booleanos, que pueden ser **true** o **false**.

Sintaxis de declaración:

```
//Declaracion Simple

int numero;
float decimal;
string frase;
bool si;

//Declaracion con asignacion

int numeroAsig = 1;
float decimalAsig = 1.5;
string fraseAsig = "Hola mundo";
bool siAsig = true;

//Declaracion multi-variable

int x, y, z;
float i, j, k;
string palabra, frase;
bool si, no;
```

3.3. Comentarios

Los comentarios de una sola línea comienzan con //. Todo lo que sigue en esa línea es ignorado por el compilador.

```
// Esto es un comentario y no será ejecutado.
int x = 10; // También se puede comentar después del código.
```

3.4. Operadores

El lenguaje incluye un conjunto completo de operadores.

Categoría	**Operadores**	**Descripción**
Aritméticos	+, -, *, /	Suma, resta, multiplicación y división
Asignación	=, +=, -=, *=, /=	Asignación simple y compuesta
Comparación	==, !=, <, >, <=, >=	Igualdad, desigualdad y comparación
Lógicos	!, &&, ||	NOT (Lógico), AND(Lógico) y OR(Lógico)

3.5. Estructuras de Control

El lenguaje ofrece varias estructuras para controlar el flujo de ejecución.

Condicional if/else: Ejecuta un bloque de código si una condición es verdadera y, opcionalmente, otro si la anterior es falsa.

```
if (edad >= 18) {
    print("Es mayor de edad.");
} else {
    print("Es menor de edad.");
}
```

Esto debería mostrar por consola “Es mayor de edad.” si “edad” es mayor de 18 y “Es menor de edad.” si “edad” es menor que 18.

También se puede utilizar la sentencia “else if(condicional){}” para evitar la anidación de “if”.

Condicional unless: Es la inversa de if. Ejecuta un bloque de código si una condición es falsa.

```
bool esta_logueado = false;

unless (esta_logueado) {
    print("Por favor, inicie sesión.");
}
```

Esto debería mostrar por consola: Por favor, inicie sesión.

Bucle while: Repite un bloque de código mientras una condición sea verdadera.

```
int contador = 0;
while (contador < 5) {
    print(contador);
    contador = contador + 1;
}
```

Esto debería mostrar por consola números del 0 al 5

Bucle for: Un bucle clásico con inicialización, condición e incremento.

```
for (int i = 0; i < 10; i = i + 1) {
    print(i);
}
```

Esto debería mostrar por consola números del 0 al 9

break y continue Se pueden usar dentro de los bucles for y while.

Para salir del bucle se puede utilizar **break** y para saltar a la siguiente iteración se puede utilizar **continue**.

3.6. Funciones

Las funciones permiten agrupar código reutilizable. Deben tener un tipo de retorno, un nombre, una lista de parámetros y un cuerpo. Si una función no devuelve nada, se usa el tipo **void**.

Sintaxis de definición:

```
// Función que suma dos enteros
int sumar(int a, int b) {
    return a + b;
}

// Función que no retorna nada
void saludar(string nombre) {
    print("Hola, " + nombre);
}
```

Recursividad Nuestro lenguaje soporta funciones recursivas. Una función es recursiva si se llama a sí misma dentro de su propio cuerpo. Esto es muy útil para resolver problemas que pueden dividirse en subproblemas más pequeños y de la misma naturaleza.

```
// Calcula el factorial de un numero usando recursividad
int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1); // Llamada recursiva a si misma
    }
}
```

3.7. Entrada y Salida

El lenguaje provee dos funciones básicas para interactuar con el usuario.

```
print("Hola mundo");
```

Imprime el valor de una expresión en la consola, seguido de un salto de línea.

```
string variable_contenedora = "";
read(variable_contenedora);
```

Lee un valor desde el teclado y lo almacena en la variable especificada.

3.8. Características Avanzadas

Operador Pipe (|>) Este operador toma el resultado de la expresión a su izquierda y lo pasa como el primer argumento a la función de su derecha. Su objetivo es mejorar la legibilidad al encadenar operaciones.

```
// La siguiente línea es equivalente a: int resultado = factorial(5);
int resultado = 5 |> factorial;
print(resultado);
```

Esto debería mostrar por consola 120.

3.9. Ejemplo de Programa Completo

```
// Función principal que se ejecutara.
void main() {
    string saludo = "Bienvenido a la calculadora de factoriales!";
    print(saludo);

    bool seguir = true;

    while (seguir) {
        int numero;
        print(""); // Salto de linea
        print("Ingrese un numero (o -1 para salir):");
        read(numero);

        if (numero < 0) {
            seguir = false;
            print("Gracias por usar el programa.");
        } else {
            // Se usa el operador pipe para llamar a la funcion factorial
            int resultado = numero |> factorial;
            print("El factorial de " + numero + " es: " + resultado);

            // Ejemplo de un bucle 'for' y el operador 'unless'
            unless (resultado > 1000) {
                print("Dato curioso: este factorial tiene pocos digitos.");
                for (int i = 1; i <= 3; i = i + 1) {
                    print("Iteración " + i);
                }
            }
        }
    }
}
```

4. Conclusión

El desarrollo de este compilador ha sido un ejercicio exhaustivo y enriquecedor que ha permitido aplicar en la práctica los conceptos teóricos del diseño de lenguajes de programación. Se ha logrado construir un compilador robusto que cumple con todos los requisitos funcionales y técnicos solicitados, incluyendo análisis léxico, sintáctico, semántico y generación de código.

El resultado es un lenguaje de programación funcional y un compilador capaz de traducirlo a **C++**, demostrando un entendimiento profundo de las herramientas y los procesos involucrados. Este proyecto sienta las bases para futuras mejoras, como la optimización de código o la adición de características de lenguaje más avanzadas.