Informe de Compilador para Lenguaje de Programación.

Fundamentos de la Computación 2025-01.

Integrantes:
Benjamín Miranda Araya
Vicente Ruiz Escobar

Docente:
José Veas Muñoz

17 de junio de 2025.

Índice.

1. Introducción.

Los compiladores son una parte esencial a la hora de programar, debido a que es lo que nos permite que las instrucciones escritas en cierto lenguaje se puedan traducir a código de máquina, permitiendo que el computador pueda entender las instrucciones. Desarrollar un compilador es un trabajo extenso y complejo en el ámbito de las ciencias de la computación, pues esto permitirá comprender en profundidad el funcionamiento interno de los lenguajes de programación como los conocemos hoy en día.

En el siguiente documento se explicará y detallará el proceso llevado a cabo para la creación de un compilador desde 0 para un lenguaje de programación propio, donde se hará uso de análisis léxico, análisis sintáctico y generación de código. Estos 3 elementos son esenciales para un funcionamiento correcto en el compilador, por lo que será necesario hacer uso de herramientas externas para lograr este objetivo.

Cabe mencionar que este informe también profundizará en elementos como la definición de la gramática del lenguaje, el diseño del árbol AST, la estrategia de generación de código, y un manual de usuario detallado. El objetivo principal será demostrar el funcionamiento del compilador, tras aplicar el conocimiento adquirido a lo largo del desarrollo del proyecto.

2. Gramática del Lenguaje.

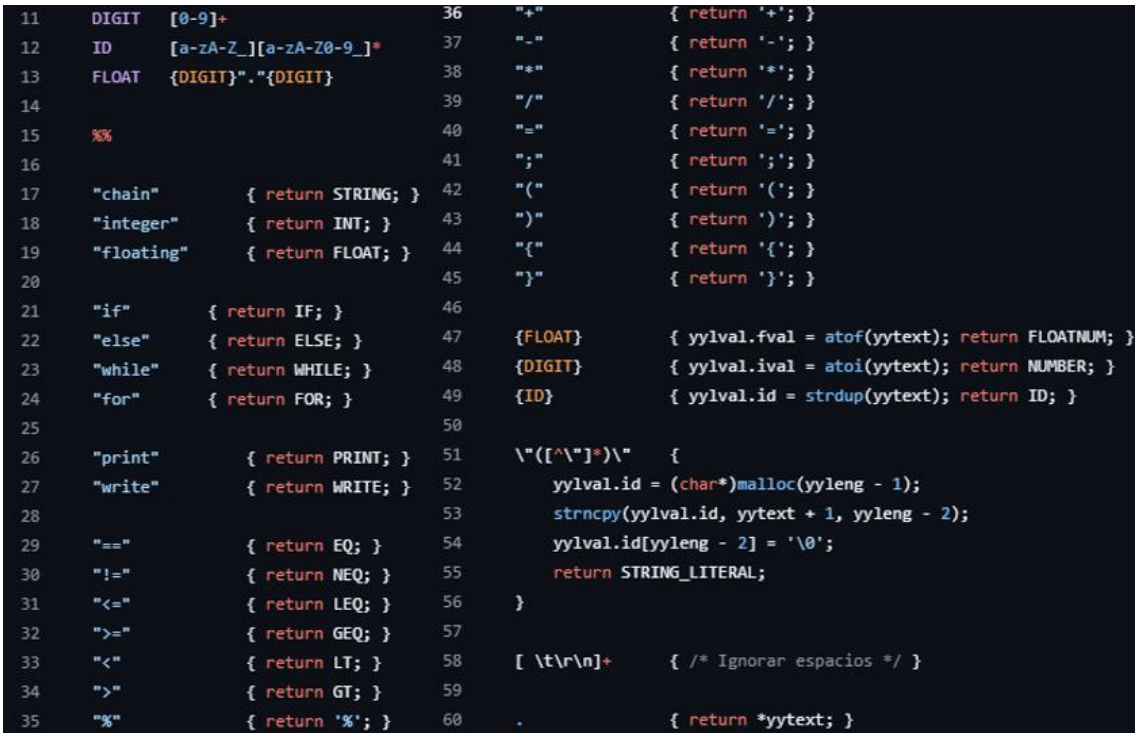
La gramática de un lenguaje de programación consiste en un conjunto de reglas formales donde se define la sintaxis del lenguaje, donde se pueden hallar elementos como símbolos terminales, símbolos no terminales, reglas de producción y un inicio. A continuación, se presentará el lenguaje de programación creado para el funcionamiento del compilador.

D++ es un lenguaje, inspirado en la sintaxis de C/C++ y de Python, el cuál cuenta con palabras reservadas definidas de manera distinta para distinguirse de estos. El lenguaje cuenta con las funciones básicas de todo lenguaje de programación:

- Declaración de variables (int, float, string)
- Condicionales (if, else)
- Ciclos (for, while)
- Entrada / Salida (input, print)
- Operaciones Aritméticas (+, -, *, /, %, **, >, <, >=, <=)
- Declaración de funciones con parámetros. (function nombreFunc()).

Para realizar esto, se hizo uso de Flex y Bison, herramientas clave a la hora de construir un compilador para un lenguaje de programación, esto será esencial para el análisis léxico y sintáctico que requerirá el compilador para interpretar el lenguaje. Para esto, se creó un archivo llamado “scanner.l” que define todos los elementos con los que va a contar el lenguaje y que el compilador pueda entender.

En el contexto de un compilador, un token representa un elemento que el compilador reconoce, como una palabra clave, un operador, un número, un delimitador, etc. Durante el análisis léxico, el programa escanea el texto del código fuente y convierte las secuencias de caracteres en una serie de tokens que luego serán utilizados por el análisis sintáctico para construir la estructura del programa.



```
DIGIT   [0-9]+
ID      [a-zA-Z_][a-zA-Z0-9_]*
FLOAT   {DIGIT}"."{DIGIT}

%%

"chain"         { return STRING; }
"integer"       { return INT; }
"floating"      { return FLOAT; }

"if"        { return IF; }
"else"      { return ELSE; }
"while"     { return WHILE; }
"for"       { return FOR; }

"print"         { return PRINT; }
"write"         { return WRITE; }

"=="            { return EQ; }
"!="            { return NEQ; }
"<="            { return LEQ; }
">="            { return GEQ; }
"<"             { return LT; }
">"             { return GT; }
"%"             { return '%'; }
"+"             { return '+'; }
"-"             { return '-'; }
"*"             { return '*'; }
"/"             { return '/'; }
"="             { return '='; }
";"             { return ';'; }
"("             { return '('; }
")"             { return ')'; }
"{"             { return '{'; }
"}"             { return '}'; }

{FLOAT}         { yylval.fval = atof(yytext); return FLOATNUM; }
{DIGIT}         { yylval.ival = atoi(yytext); return NUMBER; }
{ID}            { yylval.id = strdup(yytext); return ID; }

\"([^\"]*)\"    {
    yylval.id = (char*)malloc(yyleng - 1);
    strncpy(yylval.id, yytext + 1, yyleng - 2);
    yylval.id[yyleng - 2] = '\0';
    return STRING_LITERAL;
}

[ \t\r\n]+      { /* Ignorar espacios */ }

.               { return *yytext; }
```

Figura 1: Tokens del archivo Flex de D++. (Fuente: Elaboración propia.)

Toda esta información es lo que utilizará Bison como base para trabajar con la sintaxis del lenguaje, en la que se definen las reglas de producción para verificar que el código se haya escrito correctamente. Por ejemplo, supongamos que tenemos el siguiente código:

integer edad;
edad = 20;
print edad;

El análisis léxico del compilador (Flex) se encargará de tomar este código y convertirlo en los tokens para que se pueda procesar en la parte del análisis sintáctico del compilador (Bison). Para el ejemplo mostrado anteriormente, la conversión se vería de la siguiente manera:

INT ID ';' ID '=' NUMBER ';' PRINT ID ';'

3. Sintaxis del Lenguaje.

Para la parte del análisis sintáctico del compilador, se hizo uso de Bison, un generador de analizadores sintácticos que se encargará de definir las reglas gramaticales y construir un árbol de sintaxis abstracto (AST). De esta forma, las estructuras válidas del lenguaje se arman a partir de los tokens ya definidos anteriormente en el scanner desarrollado con Flex.

La sintaxis se compone de sentencias (stmt) y expresiones (expr), donde cada sentencia puede ser, por ejemplo, una declaración de variable, una instrucción, o una asignación. De la misma manera, una expresión es el componente que permite que se puedan realizar las operaciones aritméticas, las asignaciones, etc.

Por ejemplo, para definir la estructura de un código, se hace uso de una serie de sentencias que son válidas mediante notación BNF:

program → *stmt_list*
stmt_list → *stmt*
| *stmt_list stmt*

Donde la producción “program” indica que un programa se compone de una lista de sentencias, y la producción “stmt_list” indica que permiten múltiples sentencias consecutivas en el programa.

Entonces, para que un compilador pueda correr programas que contengan múltiples operaciones y acciones, se requiere una extensa serie de reglas para que todas las sentencias puedan detectarse y aplicarse correctamente a la hora de interpretar el código escrito, dichas reglas se detallan a continuación:

```
63    stmt:
64          INT ID ';'              { add_symbol($2, NODE_INT); $$ = make_decl_node($2, NODE_INT); }
65        | FLOAT ID ';'            { add_symbol($2, NODE_FLOAT); $$ = make_decl_node($2, NODE_FLOAT); }
66        | STRING ID ';'           { add_symbol($2, NODE_STRING); $$ = make_decl_node($2, NODE_STRING); }
67        | ID '=' expr ';'         { $$ = make_assign_node($1, (ASTNode*)$3); }
68        | PRINT expr ';'          { $$ = make_print_node((ASTNode*)$2); }
69        | WRITE ID ';'            { $$ = make_read_node($2, -1); }
70        | IF '(' expr ')' stmt %prec LOWER_THAN_ELSE
71                                  { $$ = make_if_node((ASTNode*)$3, (ASTNode*)$5, NULL); }
72        | IF '(' expr ')' stmt ELSE stmt
73                                  { $$ = make_if_node((ASTNode*)$3, (ASTNode*)$5, (ASTNode*)$7); }
74        | WHILE '(' expr ')' stmt  { $$ = make_while_node((ASTNode*)$3, (ASTNode*)$5); }
75        | '{' stmt_list '}'       { $$ = $2; }
76        | FOR '(' expr ';' expr ';' expr ')' stmt
77                                  { $$ = make_for_node((ASTNode*)$3, (ASTNode*)$5, (ASTNode*)$7, (ASTNode*)$9); }
78        ;
```

Figura 2: Sentencias del Lenguaje D++. (Fuente: Elaboración propia.)

Donde cada una de estas reglas definen el cómo se debe realizar alguna operación con las palabras reservadas del lenguaje, así mismo, dentro de las llaves se indica que cada elemento genera un nodo correspondiente que deberá ser convertido a código C.

Por otra parte, también se deben indicar cómo funcionan las debidas expresiones para poder realizar operaciones aritméticas, comparaciones, uso de valores literales y variables. Todas estas instrucciones permiten que la estructura de la sintaxis sea válida y funcional para los requerimientos del proyecto, además, también permite la implementación del árbol AST.

```
80    expr:
81          expr '+' expr      { $$ = make_binop_node("+", $1, $3); }
82        | expr '-' expr      { $$ = make_binop_node("-", $1, $3); }
83        | expr '*' expr      { $$ = make_binop_node("*", $1, $3); }
84        | expr '/' expr      { $$ = make_binop_node("/", $1, $3); }
85        | expr '%' expr      { $$ = make_binop_node("%", $1, $3); }
86        | expr EQ expr       { $$ = make_binop_node("==", $1, $3); }
87        | expr NEQ expr      { $$ = make_binop_node("!=", $1, $3); }
88        | expr LEQ expr      { $$ = make_binop_node("<=", $1, $3); }
89        | expr GEQ expr      { $$ = make_binop_node(">=", $1, $3); }
90        | expr LT expr       { $$ = make_binop_node("<", $1, $3); }
91        | expr GT expr       { $$ = make_binop_node(">", $1, $3); }
92        | ID '=' expr        { $$ = make_assign_node($1, (ASTNode*)$3); }
93        | NUMBER             { $$ = make_int_node($1); }
94        | ID                 { $$ = make_id_node($1); }
95        | FLOATNUM           { $$ = make_float_node($1); }
96        | STRING_LITERAL     { $$ = make_string_node($1); }
97    ;
```

Figura 3: Expresiones del Lenguaje D++. (Fuente: Elaboración propia.)

Para entender cómo funciona la lógica de las expresiones y sentencias, se puede demostrar con un ejemplo sencillo. Supongamos que tenemos el siguiente código que contiene declaración de variables, asignación, y salida por consola:

```
integer edad;
edad = 18;
print edad;
```

La primera línea de código le corresponde esta regla definida en el parser:

```
stmt → INT ID ';'  { add_symbol($2, NODE_INT); $$ = make_decl_node($2, NODE_INT); }
```

Aquí, el token ***"INT"*** detecta que esto corresponde a "integer", lo cuál fue definido en Flex anteriormente, y de la misma manera, ***"ID"*** detecta que este es el nombre de la variable. Esto se logra al haber definido anteriormente que ***"ID"*** puede ser cualquier combinación de valores alfanuméricos con la siguiente declaración:

```
ID      [a-zA-Z_][a-zA-Z0-9_]*
```

Finalmente, podemos apreciar que a esto le sigue una serie de instrucciones que se encargan de generar un nodo, el cuál será un paso clave para que el compilador pueda interpretar lo que se está escribiendo en el lenguaje y eventualmente nos permitirá transcribir nuestro código a código C. Para el resto de líneas se realiza el mismo proceso para cada línea con sus reglas correspondientes.

Para esto se hará uso de una estructura llamada "AST", la cuál está implementada en el compilador. Esta estructura, su funcionalidad y rol en el proyecto se detalla en el siguiente apartado del documento.

4. Árbol de Sintaxis Abstracto (AST).

El AST es una representación de la sintaxis de un código que se construye en forma de árbol, donde cada nodo representa una parte del código fuente escrito originalmente en el programa. A pesar de que el árbol omite ciertos elementos como la puntuación o las llaves, el AST es importante en un compilador debido a que este árbol define la lógica del programa, un paso esencial para su interpretación.

En este compilador, el AST se compone de un archivo Header y un archivo C, ambos archivos se encargan de definir la estructura del árbol y de cómo funciona la generación de nodos para el proyecto.

El Header posee las siguientes características:

- Define la estructura de los nodos del árbol, llamados “ASTNode”.
- Define los posibles tipos de nodos que puede generar el compilador en el apartado “NodeType”.
- El llamado a funciones encargadas de construir dichos nodos.
- La tabla de símbolos, encargada de identificar correctamente el tipo de variable que se declara en el programa.

Por otra parte, el archivo C contiene la implementación de toda la lógica que requiere lo definido en el Header:

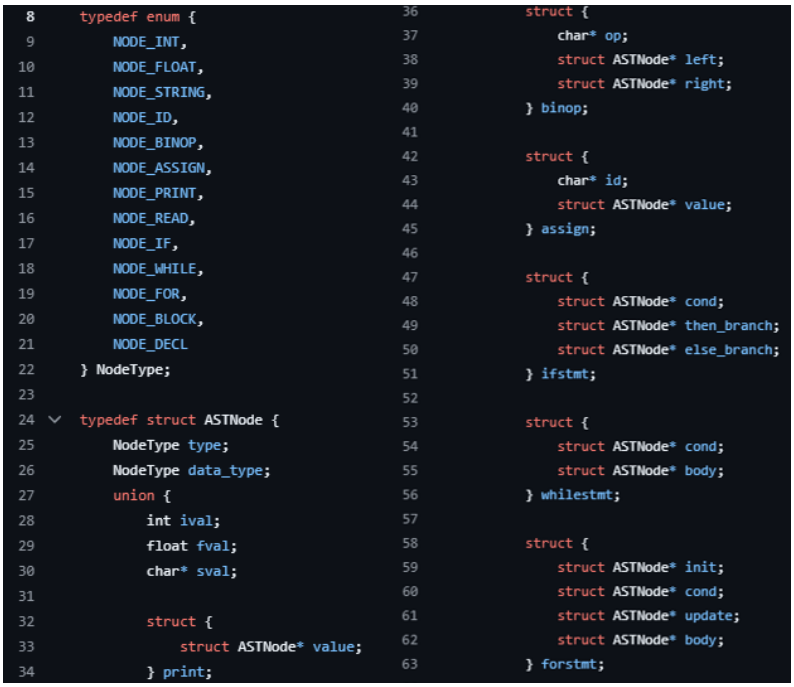
- Se generan los nodos haciendo uso de la función “malloc” en C, asignando memoria de forma dinámica.
- Se verifica la semántica del programa, donde se debe validar que la sintaxis sea correcta y que cada variable utilizada ya haya sido declarada anteriormente.
- Se implementa la lógica de la tabla de símbolos mencionada anteriormente.

Luego, la estructura de los nodos generados por el AST, como se mencionó anteriormente se compone de nodos “ASTNode” los cuáles se declaran en el archivo Header, donde dicha estructura incluye los siguientes elementos:

- Un campo “type” de tipo “NodeType” que se encarga de identificar el tipo de nodo en base al a lo que se ha leído desde el programa.
- Un campo “data_type” que indica el tipo de dato del nodo (int, float, string)
- Una unión que permite que cada tipo de nodo con sus datos pueda almacenarse correctamente en la memoria.

Para los tipos de nodos, se tiene un tipo por cada posible instrucción válida que el compilador necesite para interpretar el código del lenguaje. Estos tipos se detallan a continuación:

Tipo Nodo	Función
NODE_INT, NODE_FLOAT, NODE_STRING	Valores Literales para variables.
NODE_ID	Identificación de variables.
NODE_ASSIGN	Asignación de valor a una variable.
NODE_BINOP	Operaciones binarias. (Aritméticas)
NODE_IF, NODE_WHILE, NODE_FOR	Condicionales y ciclos.
NODE_PRINT, NODE_READ	Funciones de entrada/salida.
NODE_DECL	Declara una variable.
NODE_BLOCK	Define un bloque de sentencias agrupadas.



```c
typedef enum {
    NODE_INT,
    NODE_FLOAT,
    NODE_STRING,
    NODE_ID,
    NODE_BINOP,
    NODE_ASSIGN,
    NODE_PRINT,
    NODE_READ,
    NODE_IF,
    NODE_WHILE,
    NODE_FOR,
    NODE_BLOCK,
    NODE_DECL
} NodeType;

typedef struct ASTNode {
    NodeType type;
    NodeType data_type;
    union {
        int ival;
        float fval;
        char* sval;

        struct {
            struct ASTNode* value;
        } print;

        struct {
            char* op;
            struct ASTNode* left;
            struct ASTNode* right;
        } binop;

        struct {
            char* id;
            struct ASTNode* value;
        } assign;

        struct {
            struct ASTNode* cond;
            struct ASTNode* then_branch;
            struct ASTNode* else_branch;
        } ifstmt;

        struct {
            struct ASTNode* cond;
            struct ASTNode* body;
        } whilestmt;

        struct {
            struct ASTNode* init;
            struct ASTNode* cond;
            struct ASTNode* update;
            struct ASTNode* body;
        } forstmt;
```

Figura 4: Parte de la estructura del árbol AST. (Fuente: Elaboración propia.)

Si quisiéramos ver un ejemplo visual de cómo se comporta el árbol AST al compilar un archivo de nuestro lenguaje de programación, debemos partir por comprender que esto se hará mediante una jerarquía, organizando las operaciones según su función dentro del programa.

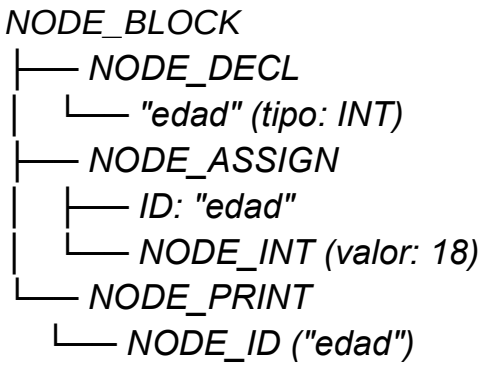
La jerarquía del AST permite que el compilador recorra todo el árbol en orden y genere el código C de forma correcta y ordenada, preservando el como estaba escrito el programa original y al mismo tiempo simplificando la tarea del análisis semántico y la validación de símbolos.

Por ejemplo, supongamos que nuevamente tenemos este código:

integer edad;
edad = 18;
print edad;

El código contiene 3 sentencias, declarando una variable tipo int, una asignación y una impresión, que corresponde a una función de salida.

La jerarquía de este código se vería de la siguiente manera:



Aquí, la raíz del árbol corresponde al nodo ***NODE_BLOCK***, debido a que el bloque de código contiene todo el programa. Dentro de este bloque, podemos encontrar 3 nodos, los cuáles corresponden a las 3 sentencias escritas anteriormente.

1. **NODE_DECL:** Representa la declaración de la variable “edad”, donde contiene el nombre de la variable y su tipo correspondiente.
2. **NODE_ASSIGN:** Representa la asignación de un valor a la variable edad, declarada anteriormente. Este nodo representa el nombre de la variable como un identificador, asegurándose de que exista, y hace uso de un ***NODE_INT*** como nodo hijo, el cuál representa el valor asignado que es 18 en este caso.
3. **NODE_PRINT:** Representa la acción de imprimir la variable edad, para el cuál necesita hacer uso de un ***NODE_ID*** para asegurarse de que no haya problemas con la variable seleccionada.

De esta forma es como se transforma cada línea de código del ejemplo anterior al árbol AST, manteniendo el orden de las reglas, la lógica y las operaciones. Preparando así el siguiente paso que es la generación de código, la cuál será realizada en el lenguaje C.

5. Generación de código C.

Para la parte final del compilador, utilizaremos el árbol AST creado anteriormente para poder recorrer los nodos y transcribir nuestro código original a su equivalente en código C. Esto nos retornará un archivo output en nuestro compilador que será totalmente ejecutable, contendrá todo el código original y se podrá compilar de igual manera que en un programa C hecho desde cero.

La razón de por qué se transforma a código C en lugar de código ensamblador o código máquina debido a que al ser un lenguaje más universal, este es más fácil de compilar en cualquier sistema que sea compatible mediante el compilador GCC, por lo que no es necesario transcribirlo al código de máquina para cada dispositivo.

También, la similitud y la facilidad con la que se pueden replicar funciones en C como lo pueden ser las condicionales, los ciclos, y variables, lo hace una opción más viable para el desarrollo del compilador. Esto hará que se pueda evitar trabajar con instrucciones de la CPU, disminuyendo la complejidad del proyecto y priorizando la funcionalidad del lenguaje de programación.

Para recorrer el AST, hay que recorrer cada nodo y generar su respectivo código correspondiente según el NodeType que vimos anteriormente, una vez hecho esto, se deben recorrer los nodos hijos que si llega a haberlos. De esta manera al transcribir el código se va a conservar la lógica y el flujo del programa original, respetando la indentación de los ciclos, el orden de las operaciones, etc.

En este compilador, la transición a código C se realiza mediante un archivo llamado “generarCodigo.c”, el cuál contiene la lógica de la función “generate_code” declarada anteriormente en el header del árbol AST.

```
void generate_code(FILE* out, ASTNode* node);
```

Donde FILE* out corresponde a un puntero que referencia al archivo de salida “output.c” y ASTNode* Es un nodo del árbol AST. Cabe mencionar que esta función es recursiva.

Tipo Nodo	Escritura en C.
NODE_INT, NODE_FLOAT, NODE_STRING	Se escriben directamente
NODE_ID	Se escribe directamente.
NODE_ASSIGN	Asigna los valores directamente de las variables, si es un string se usa la función strcpy.
NODE_BINOP	Imprime de forma recursiva el valor de la izquierda, el operador, y el valor de la derecha.
NODE_IF, NODE_WHILE, NODE_FOR	Traduce las condicionales y los ciclos a la sintaxis de C y aparta los bloques de código para cada caso. En el caso del ciclo For se representa como un while.
NODE_PRINT	Traduce la función a printf. Según el tipo (%d, %f o %s) se imprime la variable o literal.
NODE_DECL	Declara una variable ajustando el tipo.
NODE_BLOCK	Recorre un array de nodos.

Supongamos nuevamente que tenemos el siguiente código:

integer edad;
edad = 18;
print edad;

El compilador recorrerá este código y el AST se encargará de traducirlo a código C obteniendo el siguiente resultado en el archivo “output.c”

int edad;
edad = 18;
printf("%d\n", edad);

Obteniendo así, un código C totalmente compilable y capaz de funcionar de forma independiente del archivo escrito con nuestro lenguaje de programación, siendo esta la función representa la etapa final del desarrollo del compilador.

```
5   void generate_code(FILE* out, ASTNode* node) {
6       if (!node) return;
7
8       switch (node->type) {
9           case NODE_INT:
10              fprintf(out, "%d", node->ival);
11              break;
12
13          case NODE_FLOAT:
14              fprintf(out, "%f", node->fval);
15              break;
16
17          case NODE_STRING:
18              fprintf(out, "\"%s\"", node->sval);
19              break;
20
21          case NODE_ID:
22              fprintf(out, "%s", node->sval);
23              break;
24
25          case NODE_DECL:
26              switch (node->decl.decl_type) {
27                  case NODE_INT:
28                      fprintf(out, "int %s;\n", node->decl.id);
29                      break;
30                  case NODE_FLOAT:
31                      fprintf(out, "float %s;\n", node->decl.id);
32                      break;
33                  case NODE_STRING:
34                      fprintf(out, "char %s[100];\n", node->decl.id);
35                      break;
36                  default:
37                      break;
38              }
39              break;
```

Figura 5: Parte de la estructura de la función de generar código C. (Fuente: Elaboración propia.)

6. Ejemplo Práctico.

Una vez repasado todo el proceso de desarrollo y funcionamiento del compilador, analizaremos el funcionamiento de este compilador desde el principio mediante un ejemplo que contenga un poco de todas las operaciones que se pueden realizar con el lenguaje D++.

Finalmente, supongamos que tenemos el siguiente código:

```
integer x;
x = 0;

while (x < 5) {
   x = x + 1;

   if (x % 2 == 0) {
      print x;
   } else {
      print "impar";
   }
}
```

Donde tenemos declaraciones, asignaciones, ciclos, condicionales, operaciones aritméticas, y una salida por pantalla.

Primero, se inicia el proceso del análisis léxico con Flex, donde el compilador generará los tokens requeridos para los siguientes pasos. Estos tokens se verían de la siguiente forma:

INT ID ';'
ID '=' NUMBER ';'
WHILE '(' ID '<' NUMBER ')' '{'
ID '=' ID '+' NUMBER ';'
IF '(' ID '%' NUMBER EQ NUMBER ')' '{'
PRINT ID ';'
'}' ELSE '{'
PRINT STRING_LITERAL ';'
'}'
'}'

Ahora, Bison se encarga del análisis sintáctico y se deben aplicar las reglas definidas anteriormente en el compilador. Esto se vería de la siguiente manera:

x = x + 1 → make_binop_node(...) → make_assign_node("x", ...)
x % 2 == 0 → make_binop_node()
print x; print "impar"; → make_print_node(...)
if-else → make_if_node(...)
while → make_while_node(...)

Ya con las reglas a utilizar, se procede a armar el árbol AST con los nodos correspondientes a cada línea de código:

```
NODE_BLOCK
├── NODE_DECL ("x")
├── NODE_ASSIGN ("x") = NODE_INT(0)
└── NODE_WHILE
    ├── cond: NODE_BINOP("<")
    │       ├── NODE_ID("x")
    │       └── NODE_INT(5)
    └── body: NODE_BLOCK
        ├── NODE_ASSIGN ("x") = NODE_BINOP("+")
        │       ├── NODE_ID("x")
        │       └── NODE_INT(1)
        └── NODE_IF
            ├── cond: NODE_BINOP("==")
            │       ├── NODE_BINOP("%", NODE_ID("x"), 2)
            │       └── NODE_INT(0)
            ├── then_branch: NODE_PRINT(NODE_ID("x"))
            └── else_branch: NODE_PRINT(NODE_STRING("impar"))
```

Finalmente, obtenemos el siguiente código C, finalizando así el proceso del compilador:

```c
int main() {
    int x;
    x = 0;

    while (x < 5) {
        x = x + 1;

        if ((x % 2) == 0) {
            printf("%d\n", x);
        } else {
            printf("%s\n", "impar");
        }
    }

    return 0;
}
```

7. Conclusión.

El desarrollo de este proyecto requirió de muchas pruebas y errores, investigación y aplicación del conocimiento adquirido de las estructuras de datos, el lenguaje C y la teoría de compiladores. El proyecto abarcó todas las etapas del proceso típico de compilación: análisis léxico, análisis sintáctico, análisis semántico básico, construcción del árbol de sintaxis abstracta (AST) y generación de código ejecutable en lenguaje C.

Durante el primer mes de investigación, fue necesario comprender el funcionamiento de las herramientas **Flex** y **Bison**, ya que comprender como se utilizaban sería el paso clave para crear el lenguaje desde 0, permitiendo asignar las palabras reservadas para las operaciones del lenguaje, y de la misma forma utilizarlas para la generación del analizador léxico y el parser sintáctico.

También se pudo comprender el por qué generar los árboles AST son tan importantes a la hora de representar de manera ordenada los programas escritos en un lenguaje de programación. Ya que la sintaxis es mucho más legible de esta forma y puede ayudar a entender mejor los programas como si de un diagrama se tratara.

A nivel de experiencia, este proyecto reforzó los conocimientos técnicos esenciales para esta área de las ciencias de la computación, fomentando la constante mejora en la capacidad de diseño, abstracción y resolución de problemas. La construcción de un compilador es una tarea compleja, pero sirve para poner en práctica todos los conocimientos de los lenguajes y de la lógica que estos llevan por detrás.

8. Referencias

1) de Lenguajes, P., Autómatas, G. y., & de Ingeniería en Informática http://webdiis. unizar. es/asignaturas/LGA, C. C. (n.d.). *Introducción a Flex y Bison*. Unizar.Es. Retrieved June 16, 2025, from https://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf

2) (N.d.). Admb-project.org. Retrieved June 16, 2025, from https://www.admb-project.org/tools/flex/compiler.pdf

3) Pietro. (n.d.). *What is A programming Language Grammar?* Compilers. Retrieved June 16, 2025, from https://pgrandinetti.github.io/compilers/page/what-is-a-programming-language-grammar/

4) Daniel. (2024, September 17). *Teoría de los Lenguajes de Programación: todo lo que necesitas saber*. DataScientest. https://datascientest.com/es/teoria-de-los-lenguajes-de-programacion