Informe Taller de Fundamentos de la Computación

“Compilador de Minecraft”

Nombres: Benjamin Bustamante
Bryan Carvajal
Profesor : Jose Luis Veas.
Fecha de entrega : 17/06/2025

Objetivo

El objetivo de este proyecto es desarrollar un compilador para un lenguaje de programación propio utilizando las herramientas Flex y Bison para el análisis léxico y sintáctico, respectivamente, y C/C++ para la generación de código. A través de este proceso, buscamos comprender y experimentar con las etapas fundamentales de la compilación, como la generación de un árbol de sintaxis abstracta (AST), análisis semántico, y la generación de código ejecutable

1. La gramática del lenguaje propio

PALABRA CLAVE	POR LO QUE LO REEMPLAZAMOS
DIAMANTE	Palabra clave para tipo entero (int)
LAVA	Palabra clave para tipo flotante (float)
LIBRO	Palabra clave para tipo cadena (string)
ENDER	Palabra clave de control if
CREEPER	Palabra clave de control else
ZOMBIE	Palabra clave de control while
LETRERO	Palabra clave de control print
HORNO	Palabra clave de control read (input)
PORTAL	Palabra clave de control func (declarar función)
TESORO	Palabra clave de control return

2. El diseño del AST

El árbol AST queda representado al realizar uno de los pasos para compilar el taller, en concreto el cuarto paso (Get-Content Pruebas/prueba1_profe.txt | .\Compilador), al utilizar esto se genera el archivo “code.c” con toda nuestra sintaxis, y en la terminal se ve representado el árbol AST que se utilizó para la generación de este archivo.

Un ejemplo de el árbol que sale en la consola es (Este es un ejemplo simple):

Código por transformar a código en C:

ejemplosimple.txt

```
PORTAL suma(DIAMANTE a, DIAMANTE b) {
    DIAMANTE resultado;
    resultado = a + b;
    LETRERO(resultado);
    TESORO resultado;
}

DIAMANTE x;
DIAMANTE y;
LETRERO("Ingresa el primer numero:");
HORNO(x);
LETRERO("Ingresa el segundo numero:");
HORNO(y);

LETRERO("El resultado de la suma es:");
suma(x, y);
```

Árbol AST que se “printea” por consola:

```
PS C:\Users\bryan\Documents\Taller-Fundamentos> Get-Content ejemplosimple.txt | .\Compilador
Iniciando el compilador...
PORTAL suma:
  Parametro tipo =DIAMANTE id= a
  Parametro tipo =DIAMANTE id= b
  Decl: tipo=DIAMANTE id=resultado
  Asign: id=resultado
    Op: +
      Id: a
      Id: b
  LETRERO:
    Id: resultado
  TESORO
    Id: resultado
Decl: tipo=DIAMANTE id=x
Decl: tipo=DIAMANTE id=y
LETRERO:
LETRERO:
  LIBRO: "Ingresa el segundo numero:"
HORNO id=y
LETRERO:
  LIBRO: "El resultado de la suma es:"
Llamada suma
  Id: x
  Id: y
```

Archivo “code.c” que se genera a raíz del comando mencionado anteriormente:

```c
#include <stdio.h>
#define DIAMANTE int
#define LAVA float
#define LIBRO char*
#define ENDER if
#define CREEPER else
#define ZOMBIE while
#define LETRERO printf
#define HORNO scanf
#define PORTAL
#define TESORO return

DIAMANTE x;
DIAMANTE y;
DIAMANTE suma(DIAMANTE a, DIAMANTE b) {
DIAMANTE resultado;
resultado = (a + b);
LETRERO("%d\n", resultado);
TESORO resultado;
}

int main() {
LETRERO("%s\n", "Ingresa el primer numero:");
HORNO("%d", &x);
LETRERO("%s\n", "Ingresa el segundo numero:");
HORNO("%d", &y);
LETRERO("%s\n", "El resultado de la suma es:");
suma(x, y);

 return 0;
}

```

Salida por pantalla del ejemplo:

```
PS C:\Users\bryan\Documents\Taller-Fundamentos> ./programa
Ingresa el primer numero:
10
Ingresa el segundo numero:
20
El resultado de la suma es:
30
```

3. La descripción del proceso de generación de código

En esta parte del informe explicaremos paso a paso la creación y funcionamiento del compilador desarrollado a lo largo del proyecto. Nuestro compilador permite al usuario escribir programas usando una sintaxis especial, con palabras relacionadas al juego “Minecraft”, para luego traducirlos automáticamente en código ejecutable en lenguaje C.

Primer paso: Hicimos la creación del análisis léxico (utilizando Flex)

El primer paso en nuestro compilador fue definir cómo reconocer cada elemento del lenguaje escrito por el usuario. Para esto usamos la herramienta Flex, creando un archivo llamado “escaner.l“

Segundo paso: Hicimos la creación del análisis sintáctico (Utilizando Bison)

En la segunda etapa, utilizamos **Bison** para definir la estructura gramatical y sintáctica del lenguaje personalizado. Aquí establecemos cómo deben combinarse los tokens previamente identificados por Flex para formar instrucciones válidas.

En el archivo “parser.y”, definimos reglas claras para cada instrucción.

Tercer paso: Creación y Manejo del Árbol de Sintaxis Abstracta (AST)

Luego de analizar la sintaxis, es necesario representar internamente la estructura del programa. Para esto utilizamos el Árbol de Sintaxis Abstracta **(AST)**, que creamos y gestionamos mediante los archivos “nodoAst.c” y “nodoAst.h”.

El AST es una representación gráfica lógica del código. Cada instrucción es representada por nodos que tienen información clara sobre qué operación se realiza y qué elementos la componen.

Cuarto paso: Creación de tabla de símbolos para el análisis de la semántica.

En esta fase nos enfocamos en almacenar y gestionar información relevante sobre variables y funciones del programa. Usamos una estructura especial llamada “Tabla de Símbolos”, implementada en “tablaSimbolos.c” y “tablaSimbolos.h”.

La tabla guarda, por cada identificador, su nombre, tipo de dato y posición en memoria, entre otros detalles útiles para generar código final posteriormente.

Quinto paso: Generación de Código en C

En este paso, transformamos el AST en código ejecutable, generando automáticamente código C mediante los archivos “generadorDeCodigo.c” y “generadorDeCodigo.h”.

Cada nodo del AST es visitado, y dependiendo de su tipo, generamos la instrucción equivalente en C.

Sexto paso: Compilación y Ejecución del Programa Final

Finalmente, el código generado se guarda en un archivo llamado “code.c”. Para ejecutar el programa final, seguimos los siguientes pasos desde el terminal del sistema operativo:

1. bison -d parser.y (Genera los archivos correspondientes, parser.tab.h y parser.tab.c)
2. flex escaner.l (Genera su archivo correspondiente lex.yy.c)
3. gcc -o Compilador parser.tab.c lex.yy.c generadorDeCodigo.c nodoAst.c tablaSimbolos.c -lm (Genera el archivo .exe de Compilador)
4. Get-Content Pruebas/prueba1_profe.txt | .\Compilador (Le pasa el código al compilador para posteriormente crear el archivo “code.c”)
5. gcc -o programa code.c (Genera archivo .exe adjuntando lo que hay dentro del “code.c”)

6. ./programa (Se ejecuta el código realizado)

4. Ejemplos de programas en el lenguaje propio

Dentro del repositorio existe una carpeta llamada “Pruebas”, en dónde se encuentran varios ejemplos para utilizar en el compilador. En el presente informe haremos mención de tres de los archivos que existen dentro de la carpeta, estos serían: “prueba1_profe.txt”, “prueba2_profe.txt” y “prueba3_profe.txt”.

Salida por pantalla del “prueba1_profe.txt” :

```
PS C:\Users\56945\OneDrive\Documentos\GitHub\Taller-Fundamentos> ./programa
Ingresa el primer numero:
10
Ingresa el segundo numero:
2
El resultado de la suma es:
12
El resultado de la resta es:
8
El resultado de la multiplicacion es:
20
El resultado de la division es:
5
```

Salida por pantalla del “prueba2_profe.txt” :

```
PS C:\Users\56945\OneDrive\Documentos\GitHub\Taller-Fundamentos> ./programa
Ingresa el primer numero:
10
Ingresa el segundo numero:
2
El exponente es:
100
```

Salida por pantalla del “prueba3_profe.txt” :

```
PS C:\Users\56945\OneDrive\Documentos\GitHub\Taller-Fundamentos> ./programa
Ingresa el primer numero:
10
Ingresa el segundo numero:
2
El resultado de la suma es:
12
El resultado de la resta es:
8
El resultado de la multiplicacion es:
20
El resultado de la division es:
5
El exponente es:
100
```

5. Manual de usuario del lenguaje

Este lenguaje de programación ha sido diseñado con una temática inspirada en **Minecraft**, el famoso videojuego de mundo abierto. En este manual, se explican las palabras clave, operadores y su uso en el lenguaje de programación.

Palabras Clave del Lenguaje

A continuación, describiremos las principales palabras clave del lenguaje, sus significados y cómo utilizarlas en tu código.

1. Tipos de Datos

DIAMANTE (int)

El tipo de dato DIAMANTE representa números enteros. Son valores numéricos que no tienen decimales.

Antes: int cantidad = 100 ; **Ahora:** DIAMANTE cantidad = 100;

LAVA (float)

El tipo de dato LAVA representa números con decimales. Estos valores pueden tener una parte decimal.

Antes: float temperatura = 29.75; **Ahora:** LAVA temperatura = 29.75;

LIBRO (string)

El tipo de dato LIBRO se utiliza para representar cadenas de texto. Este tipo de dato es ideal para representar las palabras o frases.

Antes: string mensaje = "Bienvenido al mundo de Minecraft";

Ahora: LIBRO mensaje = "Bienvenido al mundo de Minecraft";

2. Palabras de Control

ENDER (if)

La palabra clave de control ENDER permite realizar una condición en el código. Si la condición se cumple, se ejecuta un bloque de código. Se utiliza para tomar decisiones en el flujo del programa.

Antes:

```
if (condición) {
   // Código que se ejecuta si la condición es verdadera
}
```

Ahora:

```
ENDER (condición) {
   // Código que se ejecuta si la condición es verdadera
}
```

CREEPER (else)

La palabra clave CREEPER es la opción alternativa al ENDER. Si la condición no se cumple, el bloque de código dentro de CREEPER se ejecutará .

Antes:

```
if(condición) {
    // Código que se ejecuta si la condición es verdadera
} else{
   // Código que se ejecuta si la condición es falsa
}
```

Ahora:

```
ENDER(condición) {
    // Código que se ejecuta si la condición es verdadera
} CREEPER{
   // Código que se ejecuta si la condición es falsa
}
```

ZOMBIE (while)

La palabra clave ZOMBIE permite realizar un bucle que se ejecuta de manera infinita hasta que la condición se vuelve falsa.

Antes:

```
while (condición) {
   // Código que se repite mientras la condición sea verdadera
}
```

Ahora:

```
ZOMBIE (condición) {
 // Código que se repite mientras la condición sea verdadera
}
```

3. Funciones y Retornos

PORTAL (func)

La palabra clave PORTAL se utiliza para declarar funciones en el lenguaje. Las funciones permiten organizar el código en bloques reutilizables .

Antes:

```
func miFuncion() {
    // Código de la función
}
```

Ahora:

```
PORTAL miFuncion() {
    // Código de la función
}
```

TESORO (return)

La palabra clave TESORO se usa para retornar un valor desde una función. Este valor es el "tesoro" que se obtiene al finalizar una tarea.

Antes: return 100 ; **Ahora:** TESORO 100;

LETRERO (print)

Muestra en pantalla el valor de una expresión y añade un salto de línea.

Antes:

```
int edad;
read(edad);
print("Tu edad es:");
print(edad);
```

Ahora:

```
DIAMANTE edad;
HORNO(edad);
LETRERO("Tu edad es:");
LETRERO(edad);
```

HORNO(read)

Pide un dato al usuario y lo guarda en la variable indicada.

Antes:

```
int edad;
read(edad);
print("Tu edad es:");
print(edad);
```

Ahora:

```
DIAMANTE edad;
HORNO(edad);
LETRERO("Tu edad es:");
LETRERO(edad);
```