UNIVERSIDAD PERUANA DE CIENCIAS APLICADAS

Teoría de Compiladores - CC218

Trabajo Final

INTEGRANTES

Juan Alejandro Benites Diaz	u202010449
Diego Fernando Humbser Meza	u202012711
Diego Eloy Quispe Palacin	u202012453

SECCIÓN

CC61

DOCENTE

Luis Martin Canaval Sanchez

CICLO 2023-1

1. INTRODUCCIÓN

En el ámbito de la programación, existen diversos lenguajes esotéricos que se caracterizan por presentar una sintaxis radicalmente distinta a los lenguajes de programación tradicionales. Estos lenguajes, en lugar de tener un uso práctico real, se centran en brindar entretenimiento y desafíos a los programadores. Uno de estos lenguajes esotéricos es el Brainfuck, el cual cuenta con una sintaxis muy minimalista y poco intuitiva.

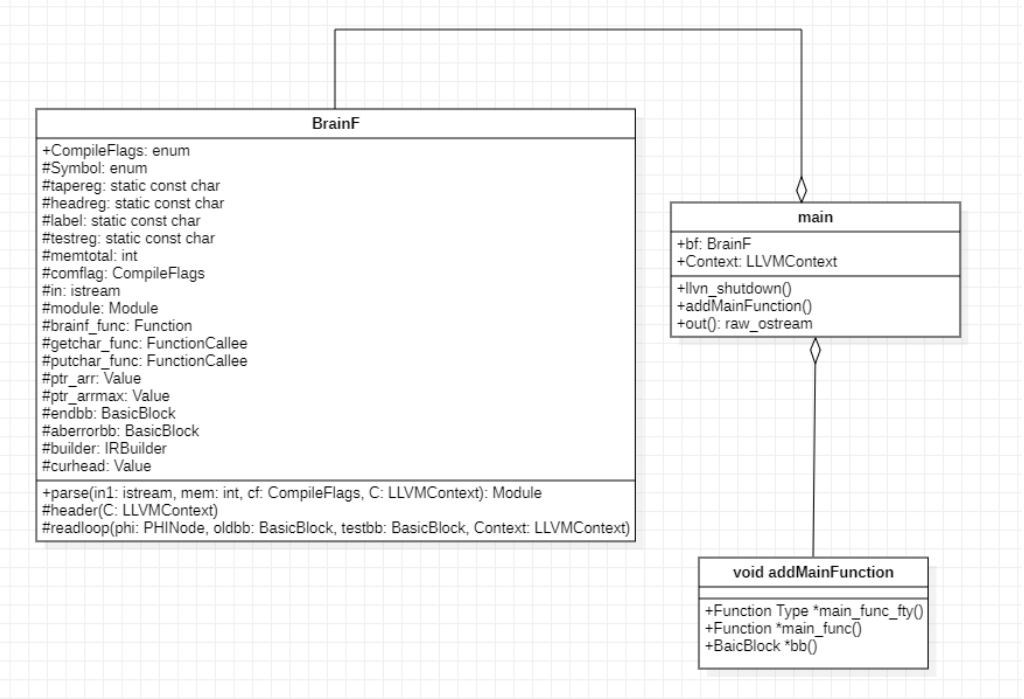
El objetivo de este trabajo es implementar un compilador para el lenguaje esotérico Brainfuck. Este compilador permitirá traducir programas escritos en Brainfuck a código ejecutable en LLVM, un popular framework para la generación de código intermedio y compilación de lenguajes de programación. Con la implementación de este compilador, se busca explorar los conceptos fundamentales de la compilación, así como también mejorar la comprensión de la estructura y funcionamiento de los lenguajes de programación.

2. MARCO TEÓRICO

- **Brainfuck:** Brainfuck es un lenguaje de programación esotérico inventado por Urban Müller en 1993. Su diseño se basa en la simplicidad extrema, utilizando solo ocho comandos para manipular un arreglo de memoria y realizar operaciones sobre él. El lenguaje Brainfuck se considera un desafío para los programadores, ya que requiere un enfoque mental diferente al de los lenguajes tradicionales. Los programas en Brainfuck están compuestos por una secuencia de comandos, donde cada comando se representa con un único carácter.
- **LLVM (Low-Level Virtual Machine):** LLVM es un framework de compilación de código abierto que proporciona herramientas y bibliotecas para la generación, optimización y ejecución de código de programación. LLVM utiliza una representación intermedia en forma de código de máquina de bajo nivel independiente de la arquitectura del procesador, lo que permite la portabilidad y optimización del código fuente en diferentes plataformas. La infraestructura de LLVM incluye un conjunto de herramientas como el compilador Clang y el optimizador LLVM, que facilitan la creación de compiladores eficientes y flexibles.
- **Teoría de Compiladores:** En el contexto de este trabajo, el objetivo es implementar un compilador para el lenguaje esotérico Brainfuck utilizando LLVM como infraestructura. Para lograr esto, se debe realizar un análisis léxico y sintáctico del código fuente en Brainfuck, generar código intermedio de LLVM correspondiente y optimizarlo cuando sea posible. Finalmente, el código intermedio se compila y se genera el código ejecutable que pueda ser utilizado para ejecutar programas en Brainfuck.

3. DESARROLLO

3.1. Diagrama de Clases



El programa está compuesto por 3 archivos principales, estos siendo Brain.cpp, BrainF.cpp y BrainFDriver.cpp. Estos ya están mostrados en el diagrama de clases.

Por un lado, tenemos la clase BrainF, la clase principal donde se crea todo el sistema para el lenguaje BrainFuck, aquí se crean bastante variables y funciones donde posteriormente se utilizarán para poder crear el lenguaje y estos serán usados finalmente en la función main. Por otro lado, tenemos a la función addMainFunction, donde su función es la de ir añadiendo funciones , y esta va acompañada de la función de BrainF. Finalmente tenemos la función main, donde todo el ambiente se crea y ejecuta.

3.2. Procedimiento de Compilación, Ejecución y Prueba del Compilador

I. Compilación del código fuente

- En el directorio raíz del proyecto, dentro del directorio 'src' donde se encuentran los archivos fuente del proyecto, crear el archivo de configuración principal de CMake, 'CMakeLists.txt', que define cómo se debe construir el proyecto.
- Este archivo contiene las instrucciones para configurar las opciones de compilación, los archivos fuente del proyecto y las dependencias:

```
cmake_minimum_required(VERSION 3.22)
project(brainf_compiler)

find_package(LLVM REQUIRED CONFIG)

include_directories(${LLVM_INCLUDE_DIRS})
add_definitions(${LLVM_DEFINITIONS})

add_executable(brainf_compiler BrainF.cpp
BrainFDriver.cpp)
```

```
llvm_map_components_to_libnames(llvm_libs BitWriter
Core ExecutionEngine MC MCJIT Support nativecodegen)
target_link_libraries(brainf_compiler ${llvm_libs})
```

- Crear un directorio de compilación separado, por ejemplo, 'build', en el directorio raíz del proyecto.
- Abrir una terminal y navegar hasta el directorio de compilación recién creado.
- Ejecutar el comando 'cmake' seguido de la ruta al directorio 'src' del proyecto (donde se encuentra el archivo CMakeLists.txt):

```
cmake ../src/
```

- Si todo está configurado correctamente, se generará un Makefile en el directorio 'build'.
- Ahora se podrán compilar los archivos fuente ejecutando el comando 'make':

```
make
```

- Esto compila los archivos fuente utilizando las bibliotecas de LLVM especificadas en el archivo CMakeLists.txt y generará el ejecutable 'brainf_compiler' en el directorio 'build'.

II. Ejecución del compilador de Brainf**k

- Crear un directorio 'test' en el directorio raíz del proyecto y asegurarse de tener un programa Brainf**k válido en un archivo con extensión '.bf'.
- Abrir una terminal y acceder al directorio 'build' que contiene el ejecutable 'brainf_compiler'.
- En la terminal, ejecutar el siguiente comando para compilar un programa Brainf**k:

```
./brainf_compiler [OPCIONES] ../test/<input_file>.bf
```

III. Análisis del código fuente Brainf**k

- El compilador cargará el archivo fuente Brainf**k especificado y lo leerá.
- Utilizará la clase 'BrainF' para analizar el programa Brainf**k y generar un módulo LLVM correspondiente.

IV. Verificación del módulo LLVM

- El compilador verificará el módulo LLVM generado utilizando la función 'verifyModule' de LLVM.
- Si el módulo no pasa la verificación, se mostrará un mensaje de error y se abortará el programa.

V. Generación de código objeto o ejecución Just-In-Time (JIT)

- Dependiendo de las '[OPCIONES]' de línea de comandos, el compilador realizará una de las siguientes 2 acciones:

    1. Si se especifica la opción '-jit', se ejecutará el programa Brainf**k utilizando el JIT (Just-In-Time) de LLVM.
    2. Si **no** se especifica la opción '-jit', se generará un archivo de código objeto (Bitcode) en el directorio actual utilizando la función `WriteBitcodeToFile` de LLVM.

VI. Ejecución del programa Brainf**k (si se seleccionó la opción '-jit')

- Se inicializará el objetivo nativo de LLVM y se creará un motor de ejecución (ExecutionEngine).
- El motor de ejecución ejecutará la función 'brainf' dentro del módulo LLVM generado.
- El resultado de la ejecución del programa Brainf**k se mostrará en la salida estándar.

VII. Ejecución del programa Brainf**k (si NO se seleccionó la opción '-jit')

- Una vez generado el archivo de código objeto (Bitcode), ejecutar el programa Brainf**k utilizando el intérprete 'lli':

```
lli ../test/<input_file>.bf.bc
```

- Se ejecutará el programa Brainf**k utilizando el intérprete LLVM.

VIII. Pruebas del compilador

- Para probar el compilador Brainf**k, se puede utilizar diferentes programas Brainf**k como entrada.
- Ejecutar el compilador con el archivo fuente Brainf**k deseado y verificar que el resultado de la ejecución sea el esperado.
- Probar programas Brainf**k simples y más complejos para verificar el correcto funcionamiento del compilador.

4. PRUEBAS

- Programa de Prueba #1:

```
,>++++[<------------>-],>++++[<------------>-]<[>+<-]<[>>+<<-]>>>+
+++[<++++++++++++>-]<.

INPUT: 33 -> OUTPUT: 6
INPUT: 45 -> OUTPUT: 9
INPUT: 55 -> OUTPUT:
```

Un programa que se encarga de sumar dos números enteros positivos y muestra el resultado si es menor a 10. Si la suma de los dos números es igual o mayor a 10, no mostrará el resultado de la suma.

- Programa de Prueba #2:

```
++++++++++[>+>+++>+++++++>++++++++++<<<<-]>>>----.>+++++.----.++++
```

```
+++++.++++++++.-----------------.+++++++++.-----.-----.++++++++++
.<<++.>>--------------.<<.>>+.+++++++++++++++.-----------------.+
+++++++.+++++.--------.+++++++++++++++.------------------.++++++++
.<<+.
```

Un programa que muestra una bienvenida al lenguaje de programación esotérico.

- Programa de Prueba #3:

```
,>++++[<------------>-]<[>+>+++++++[<++++++>-]+++[<++++>-]<<[>.+<-]
]

INPUT: 7 -> OUTPUT: 0123456
INPUT: D -> OUTPUT: 0123456789:;<=>?@ABC
```

Este programa muestra los números enteros menores hasta el 0 del número de entrada ingresado. Si se coloca un código ASCII, se muestran todas las representaciones de carácteres menores hasta el '0' (48).

- Programa de Prueba #4:

```
>,>++++[<------------>-],>++++[<------------>-]<[<[<+>>>+<<-]>>[<<
+>>-]<-]++++[<++++++++++++>>-]<<.

INPUT: 33 -> OUTPUT: 9
INPUT: 19 -> OUTPUT: 9
INPUT: 25 -> OUTPUT:
```

Este programa se encarga de multiplicar dos números enteros positivos y muestra el resultado si es menor a 10.

- Programa de Prueba #5:

```
>++++[<++++++++++>-]+[<++>-]>++[<+++++>-]>,>++++[<------------>-],
>++++[<------------>-]<[<[<<<.>>>>>+<<-]>>[<<+>>-]<<<<.>>>-]

INPUT: 55
OUTPUT:
*****
*****
*****
*****
*****
INPUT: 53
*****
*****
*****
```

Este programa muestra un rectángulo representado con el carácter '*' con las

dimensiones (mxn) que se le ingresan (m=largo, n=ancho | m,n < 10).

5. CONCLUSIONES

- Es gracias al trabajo y curso en general que nos ha otorgado una perspectiva diferente al mundo de compiladores, no solo hemos podido elaborarlos desde cero, si no que al entenderlo podemos crear ahora nuestros propios compiladores a nuestra propia comodidad.
- Este proyecto nos ha proporcionado una valiosa experiencia práctica en el desarrollo de compiladores, permitiéndonos comprender en mayor profundidad los aspectos teóricos y prácticos relacionados con la compilación de lenguajes de programación. El diseño detallado del diagrama de clases, el proceso paso a paso de compilación, ejecución y prueba, así como la creación de programas de prueba, han contribuido a fortalecer nuestra comprensión de los conceptos fundamentales de la compilación.
- Gracias a brainfuck y los lenguajes esotéricos es que nos otorgan experiencia valiosa no solo en el desarrollo del código para un lenguaje de programación y su compilador, si no para la compresión de los lenguajes de programación en general. De esta manera, nosotros tenemos la capacidad de comprender a profundidad estos y la capacidad de poder modificar el código fuente de bastantes lenguajes de programación o programas. Esto se puede aplicar para varios campos como lo es el de la ciberseguridad.