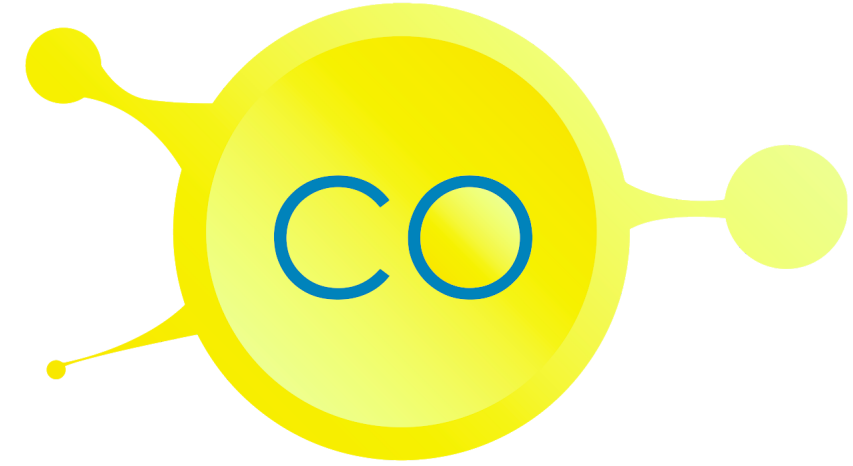


COMPILER by neuro Technologies

INTEGRANTES:

- **AGUILAR GONZÁLEZ OSCAR**
- **CAMPOS RODRÍGUEZ LEONARDO JOSÉ**
- **FLORES LICEA LARS ALAIN**
- **HERNÁNDEZ JIMÉNEZ DIANA LISSET**
- **JORGE ROMERO FERNANDO**
- **REYES MARÍN ALEXANDER**

ÍNDICE

Introducción

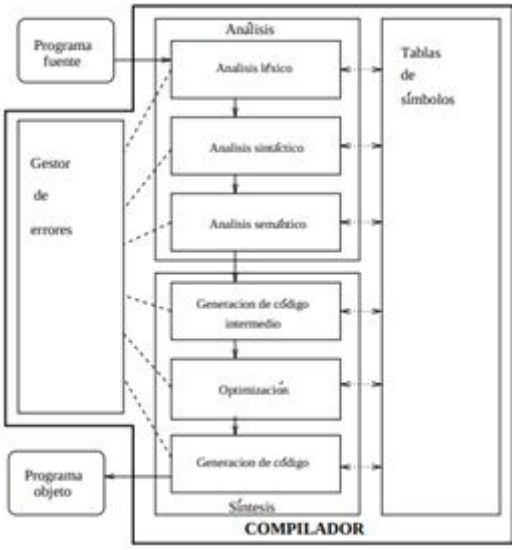
El primer compilador fue usado en 1952 y en el 2003 fue el momento culmen debido a que; el compilador fue optimizado y usado de manera modular,
Algo de historia sobre compiladores

- •Corrado Böhm: First compiler description 1951
- •Alick Glennie: First implemented compiler 1952
- •Grace Hopper: First use of the “Compiler” word 1952
- •John W. Backus: First commercial compiler 1957
- •Chris Lattner: First compiler toolkit (LLVM) 2003

Un compilador es un programa que transforma el código fuente escrito por el desarrollador en un lenguaje de programación de alto nivel en su expresión equivalente en lenguaje máquina, que puede ser interpretado por el procesador.

El proceso de convertir el código fuente a su representación en código de objeto se conoce como compilación.

Un compilador comprende: una unidad de front - end que modifica un código nativo en un código intermedio; una unidad de modificación de código que genera un código intermedio modificado, que es un código modificado del mismo código intermedio que genera el front - end, reemplazando una primera instrucción, satisfaciendo una condición pre configurada entre las instrucciones incluidas en el código intermedio, a una instrucción de bifurcación de condición; y una unidad de fondo que modifica el código intermedio modificado en un código de objeto. En consecuencia, el compilador puede evitar que un aumento del tiempo de ejecución aumente innecesariamente, cuando el código objeto se ejecuta en cierto hardware.



En la anterior imagen podemos observar la estructura de un compilador, se compone de un analizador léxico, sintáctico y semántico, una vez aplicado este análisis al código fuente, el compilador genera un

código intermedio, para después pasar por la optimización, al finalizar este paso, es cuando se genera el código intermedio modificado y por último el código objeto.

Un compilador ejecuta cuatro funciones principales.

Scaning: El escáner lee un elemento a la vez del código fuente y mantiene presente que elementos existen por línea de código.

Análisis Léxico: El compilador convierte la secuencia de elementos o caracteres que aparecen en el código fuente a una serie de cadenas de caracteres conocidos como “tokens” que son asociados a un a regla específica por el programa llamada analizador léxico. Una tabla de símbolos o estructura de datos análoga es usada por el analizador léxico para guardar las cadenas del código fuente que corresponden con el token generado.

Análisis Sintáctico: En este paso, el análisis de la sintaxis es realizado, esto conlleva procesar la información capturada para verificar si los tokens creados durante el análisis léxico están apropiadamente ordenados para su uso. El orden correcto de un conjunto de palabras clave o “keywords” para obtener el resultado deseado se conoce como sintaxis. El compilador tiene que examinar que el código fuente cumpla con la sintaxis esperada.

Análisis Semántico: Este paso está compuesto de varios procesos intermedios. Primero la estructura de tokens es verificada, así como el orden de estos de acuerdo con la gramática específica de cada lenguaje. El significado de la estructura de tokens es interpretado por el parser y el analizador para finalmente generar código intermedio, llamado también como código de objeto. El código de objeto incluye instrucciones que representan comandos interpretables por el procesador.

Es importante comprender los principios fundamentales de cómo diseñar e implementar un compilador, para esto se debe conocer previamente las ideas básicas que permitan construir un compilador, independientemente del lenguaje para el cual se cree.

En nuestra formación cómo ingenieros en computación es importante, ya que aprender a diseñar un compilador nos permite mejorar nuestra práctica programando, ya que nos permite incorporar perspectivas y conocimientos que, si bien no son fundamentales, sí son relevantes y necesarios para los conocimientos que se tienen hasta ahora, cómo lo son las estructuras e implementaciones de los lenguajes de programación.

neura Technologies

Plan de Proyecto

Nuestro plan de trabajo original consistía en hacer la entrega final en cuatro semanas, cada una estaba destinada a una tarea; la primera semana se investigaría la programación funcional y el modo de instalar y utilizar elixir, también se configurarán las máquinas instalando una versión estable de linux, git y elixir, también se daría un tiempo para aprender a usar los comandos básicos de en este caso Ubuntu 20.04 y git ya que, varios integrantes del equipo no habían tenido contacto con estas herramientas, al final de esta semana se desarrollaría la parte del depurador y también del análisis léxico (lexer), la segunda semana estaría destinada al análisis sintáctico (parser), así como la creación de un menú dónde se mandarán a llamar a las funciones, la tercera semana, estaría destinada para el generador de código y el ejecutable, así cómo hacer mejoras o correcciones en las entregas anteriores, por último la cuarta semana estaría destinada para hacer las pruebas correspondientes al compilador.

Arquitectura

La estructura general del compilador está fundamentada en las recomendaciones del tutorial de Nora Sandler. Por lo anterior el compilador está compuesto por cuatro módulos fundamentales, estos son: Lexer, Parser, Code Generator, Linker.

Los módulos anteriormente mencionados procesan el código de manera secuencial como se ve en el siguiente diagrama.

Sanitizer

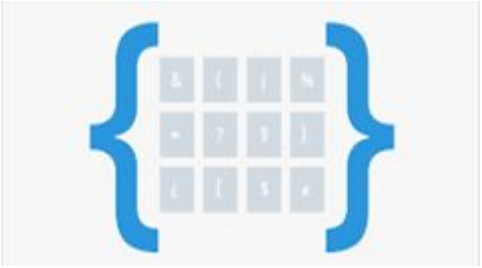
- Elimina caracteres innecesarios unicode

Lexer

- Sanitiza el código
- Crea lista inicial de expresiones

Parser

- Asigna Tokens
- Crea estructura de árbol AST



Code Generator

- Recorre AST
- Asigna tokens con equivalente en OC

Linker

- Llama al compilador gcc

La estructura siguiente es la general del compilador. en esta entrega es:

SANITIZER

Entradas	Salidas
Código fuente	Listas con elementos String

PROCEDIMIENTO

El sanitizador por medio de la instrucción String.trim() quita caracteres innecesarios como los saltos de línea, y espacios en blanco unicode y por medio de la instrucción split(), se hace la división de la cadena y genera una lista con datos tipo string igualmente.

LEXER

Entradas	Salidas
Lista con elementos string	Lista con tokens y lista con elementos string.

PROCEDIMIENTO

Entra primero a función de scanner donde tenemos una función map, esta función nos permite hacer operaciones sobre elementos de una lista de palabras que viene del sanitizador (es el primer parámetro), y como segundo parámetro tenemos las operaciones que se harán sobre de él. El segundo parámetro es la función lex_raw tokens() donde se obtienen se identifican los tokens en las cadenas del primer parámetro.

Al estar dentro de una función map, puede llegar a generar muchas listas, por lo que llegue a ser contraproducente, entonces implementamos la función flat_map(), que extrae todas las listas y las "aplana" para que quede una lista de elementos y no una lista de listas.

PARSER

Entradas	Salidas
Lista de tokens	Árbol AST

PROCEDIMIENTO

El parser se compone de 4 procedimientos dependientes, en el primer procedimiento se hace una llamada a una función que tiene la estructura de una programa básico en C, donde con la lista de tokens en orden valida la estructura del programa. Aquí se crea el árbol AST.

CODE GENERATOR

Entradas	Salidas
Árbol AST	archivo extensión.s

PROCEDIMIENTO

Utilizando el algoritmo de pos orden para recorrer el árbol, se recogen las operaciones en primera instancia y posteriormente expresiones de menor precedencia ordenando el código para que sea procesado.

LINKER

Entradas	Salidas
archivo extensión. s	ejecutable de 32 o 64 bits

PROCEDIMIENTO

Manda a llamar al linker de gcc para compilar el archivo con extensión .s que fue generado previamente en el Generador de código y así generar el ejecutable.

Pruebas

Se generaron una serie de pruebas de forma automatizada, con la intención de ahorrar tiempo y hacer al compilador más eficiente. Estas pruebas están basadas en las recomendaciones del tutorial de Nora Sandler, y son de vital importancia para indicar si el compilador funciona de manera correcta en muchas posibles entradas que el usuario pueda proporcionar.

Las pruebas se encuentran en dos carpetas dentro de otra carpeta llamada "***test***": "***valid***" e "***invalid***". La carpeta "*valid*" tiene **8** pruebas que contienen diferentes escenarios en el código y el compilador debe aprobar:

- **multi_digit.c:** en esta prueba el compilador debe aprobar un valor entero de retorno de más de un dígito, como el número "*100*".
- **newlines.c:** en esta prueba el compilador debe aprobar que cada token que forma parte del código esté escrito en una línea diferente.
- **no_newlines.c:** en esta prueba el compilador debe aprobar que todos los tokens que forman parte del código estén escritos en una sola línea.
- **return_0.c:** en esta prueba el compilador debe aprobar el número "*0*" como valor entero de retorno.
- **return002.c:** en esta prueba el compilador debe aprobar el número "002" como valor entero de retorno.
- **return_2.c:** en esta prueba el compilador debe aprobar el número "*2*" como valor entero de retorno. Este número es el que está especificado como valor entero de retorno para esta entrega.
- **spaces.c:** en esta prueba el compilador debe aprobar que haya más de un espacio entre los caracteres que forman parte del código.
- **tabulador.c:** en esta prueba se analiza la interacción que tiene el compilador con el separador tabulador.

En resumen, estas pruebas contienen diferentes escenarios en el código, como que se escriba en una o en varias líneas, o que el valor entero que se ingresa conste de uno o más dígitos, o que el código contenga uno o más espacios entre cada palabra.

Dentro de la carpeta hay un ***make*** que al momento de llamarlo con el comando "*make ejecutar*" ejecuta las pruebas automatizadas y crea un ensamblador de los programas.

La segunda carpeta, "*invalid*", tiene **11** pruebas que contienen diferentes escenarios en el código y el compilador debe rechazar:

- **doble_punto.c:** en esta prueba se hace un análisis de caracteres no reconocidos, en este caso es un conocido "*fat finger"* en el que se confunde un ";" con un ":".
- **missing_paren.c:** en esta prueba el compilador debe rechazar que falte alguno de los dos paréntesis dentro del código.
- **missing_retval.c:** en esta prueba el compilador debe rechazar que falte el valor entero de retorno.
- **no_brace.c:** en esta prueba el compilador debe rechazar que falte alguna de las dos llaves dentro del código.
- **no_semicolon.c:** en esta prueba el compilador debe rechazar que falte el punto y coma después del valor entero de retorno.
- **no_space.c:** en esta prueba el compilador debe rechazar que no haya espacio entre la palabra reservada "*return*" y el valor entero de retorno.
- **nt_missing.c:** en este caso se realiza una prueba de análisis en sintaxis en la cual se escribe "nt" en lugar de "int" generando un error en el valor entero de retorno.
- **return2_0.c:** en esta prueba el compilador debe rechazar que se ingrese un valor real, como "*2.0*", como valor de retorno.
- **returnC.c:** en esta prueba el compilador debe rechazar que se ingrese un caracter, como la letra "*c*", como valor de retorno.
- **returnLong.c:** en esta prueba el compilador debe rechazar que se ingrese un valor de tipo "*Entero largo*" (que es menor al número "- *32768*" y mayor al número "*32767*") como valor de retorno.
- **wrong_case.c:** en esta prueba el compilador debe rechazar que la palabra reservada "*return*" esté escrita con letras mayúsculas.

En resumen, estas pruebas contienen escenarios en los cuales haya algún error con el código, como que falte algún paréntesis o llave, o que los caracteres tengan un orden incorrecto, o que falte el punto y coma después del valor entero de retorno, o que la palabra reservada "*return*" esté escrita con letras mayúsculas.

Éstas pruebas están consideradas en un entorno en el que el usuario conoce cuáles son los principios básicos de un lenguaje, por lo cual no se preocupa en la optimización de un código o de presentar advertencias para mejorar los procesos de un código.

Dentro de la carpeta "***test***" hay un ***make*** que al momento de llamarlo con el comando "*make correr_pruebas*" ejecuta las pruebas automatizadas de las carpetas "*valid*" e "in*valid*". Pero antes que nada se deberá ejecutar un ***make*** que se encuentra dentro de la carpeta "*valid*", el trabajo de este ***make*** es generar un código fuente/ensamblador/ejecutable de los códigos válidos en C.

neura Technologies

Conclusiones y dificultades

AGUILAR GONZÁLEZ OSCAR

Trabajar con un equipo en proyectos escolares anteriores ya era algo complicado, pero trabajar en un proyecto con roles establecidos es aún más difícil. Principalmente la organización y coordinación con todo el equipo y cumplir con los tiempos de entrega es algo realmente importante puesto que eso definirá la forma de trabajo del equipo.

Algo a tener en cuenta que aumentó un poco la dificultad, fue analizar un código que no fue hecho por nosotros, con una arquitectura no diseñada por nosotros, a esto se le añade la dificultad de hacer en un lenguaje del cual no teníamos conocimiento, pero adaptarse a este fue vital para cumplir con tiempos de entrega.

Un conocimiento adquirido fue que para que un equipo funcione correctamente, todos deben de saber un poco de las áreas adyacentes, esto con el fin de agilizar el proyecto.

CAMPOS RODRÍGUEZ LEONARDO JOSÉ

Se cumplió el objetivo en esta primera entrega, ya que se logró diseñar e implementar cada componente que conforma nuestro compilador. Sin duda fue un gran reto adaptarse a un nuevo paradigma de programación con el Lenguaje Elixir, y también fue algo nuevo, en lo personal, trabajar de lleno con el Sistema Operativo LINUX. Se presentaron algunas complicaciones al intentar generar y unir algunas partes del compilador, pero se lograron resolver de manera satisfactoria.

Como Tester, se lograron diseñar algunas pruebas para complementar las que proporciona Nora Sandler en su tutorial y hacer un poco más grande el panorama en cuanto a las entradas por parte del usuario se refiere. También, se lograron implementar dichas pruebas de forma automatizada y con esto el compilador resultó ser más eficiente. Se cumplió el objetivo de forma satisfactoria.

FLORES LICEA LARS ALAIN

Esta primera entrega ha sido una experiencia agridulce, por un lado el conocer; ¿Qué es?, ¿Qué hace?, ¿Cuál es la relevancia?. Del compilador, las diferencias entre un lenguaje compilado e interpretado, la inclusión de un nuevo lenguaje de programación (Elixir), el trabajar con un paradigma de programación (funcional) en el que sólo había visto simples ejemplos. El uso de herramientas como GitHub, Discord, trabajar de lleno y familiarizarse por completo con un nuevo sistema operativo (Linux), todo esto me ha parecido fascinante y ha abierto un gran interés sobre el tema de compiladores, debido a que, está conformado por conceptos vistos en: Sistemas Operativos, Lenguajes Formales y Autómatas, etc.

El otro lado, es un aspecto que sin duda ha afectado a todos, el no poder comunicarme de la manera idónea (verbal) con mis compañeros, etc. Ha repercutido en muchas confusiones, no poder delimitar

las tareas de mi rol en el equipo fue sin duda lo que más me costó, queriendo abarcar todo y a la vez nada. En el rol de arquitecto, la estructura del compilador está basada en las recomendaciones de Nora Sandler y lo más importante es responder; ¿Qué hacen los módulos? Sabiendo esto, podemos saber qué módulo poder optimizar.

Mediante esta primera entrega se analizó a fondo la forma en que trabaja un compilador, así como los componentes que lo integran.

HERNÁNDEZ JIMÉNEZ DIANA LISSET

Para esta primera entrega teníamos como objetivos conocer y saber emplear las partes de un compilador, comprender el cómo funcionan, considero que fue en cierta manera difícil aprender un entorno nuevo como lo es Elixir. Pero aprendimos acerca de cómo realizar el compilador por partes, qué era cada una de nuestras partes y qué debía realizar, aplicamos los conocimientos adquiridos en clase el cómo funcionaba el lexer o el parser, también el code generator, el árbol AST, lo difícil no fue plantear la idea si no de alguna manera comprender qué era lo que queríamos realizar, además de que se tuvo que aprender a trabajar en equipo, lo cual en cierta manera fue un poco difícil ya que cada uno debía aprender de su rol pero también el de los demás para poder apoyarnos. Este proyecto requiere que aprendamos a plantear nuestros conocimientos adquiridos a lo largo de la carrera y que podamos aprender a trabajar en equipo, por eso concluyo que nuestra primera entrega fue un éxito.

JORGE ROMERO FERNANDO

Como equipo se planteó la idea de conocer la estructura de un compilador, el cómo funciona, la arquitectura, y los diferentes componentes que tiene en back-end y front-end. En clases vimos la estructura que debe llevar un lexer, un parser y un limpiador.

Uno de los retos más interesantes que tuvimos con este proyecto fue la implementación e interpretación de pseudocódigos para Lexer y Parser, tuvimos que estudiar la sintaxis de Elixir y la forma en la que se manejan los datos para tener una idea más clara de cómo funcionan dichos algoritmos en el lenguaje. También se tuvo que analizar el cómo iban a interactuar ambas partes del compilador para poder establecer un tipo de dato o de estructura que beneficiara tanto al Lexer como al Parser.

Como tester uno de los problemas más claros que tuvimos fue establecer diferentes escenarios en los que el usuario pudiera tener algún error o alguna forma de organizar su código que pudiera causar interferencias con el compilador. Después de eso se analizó el ambiente general en el que se correría el sistema, en este último caso las pruebas fueron un poco más cerradas debido a que no está una parte con la que el usuario tenga gran contacto.

Con este proyecto se reforzaron nuestros conocimientos en investigación e implementación de soluciones para problemas enfocados al análisis de código.

REYES MARÍN ALEXANDER

Mientras se estaba analizando/creando/desarrollando la primera parte de este proyecto se pudieron notar varios puntos fuertes acerca del trabajo realizado:

1. El trabajo en equipo es un gran reto debido a que son diferentes formas de pensar mezcladas en una misma mesa de trabajo, pero la idea es saber cómo unirnos para formar un bloque, una unidad sólida.
2. Aprender un nuevo lenguaje de programación es todo un reto debido a que es un nuevo mundo que explorar con demasiados comandos que ver y entender para poder realizar un proyecto de tal magnitud.
3. Los roles de un equipo es muy importante ya que cada integrante debe meterse en su papel para poder cumplir su trabajo, esto se cumplio con éxito.
4. Hubo ciertas dificultades al momento de entablar comunicación debido a “n” cosas, pero se supieron resolver y al plantearse un problema se buscó la solución más óptima y que beneficiará a todos.

BIBLIOGRAFÍA

Nora Sandler. (2017). *Writing a C Compiler, Part 1*. Retrieved on October 1, 2020, from: https://norasandler.com/2017/11/29/Write-a-Compiler.html.

Eng. Norberto Jesus Ortigoza Marquez. (2020). *Compilers*. UNAM. Faculty of Engineering. Retrieved on November 1, 2020.