**Arquitectura Utilizada para Construir el Compilador de Lenguaje C**

Revisamos muchos tipos de arquitecturas para desarrollar el compilador, pero por la que pensamos que es mejor y útil para este problema es la arquitectura basa en capas.

Utilizamos esa arquitectura de base para poder adaptarla a los recursos y desarrolladores tan limitados con los que se cuentan en este caso para crear el compilador.

La arquitectura basada en capas se enfoca en la distribución de roles y responsabilidades de forma jerárquica proveyendo una forma muy efectiva de separación de responsabilidades. El rol indica el modo y tipo de interacción con otras capas, y la responsabilidad indica la funcionalidad que está siendo desarrollada.

El estilo de arquitectura basado en capas se identifica por las siguientes características:

• Describe la descomposición de servicios de forma que la mayoría de la interacción ocurre solamente entre capas vecinas.

• Las capas de una aplicación pueden residir en la misma maquina

• Los componentes de cada capa se comunican con otros componentes en otras capas a través de interfaces muy bien definidas.

**PRINCIPIOS FUNDAMENTALES**

Los principios comunes que se aplican cuando se diseña para usar este estilo de arquitectura incluyen:

• Abstracción. La arquitectura basada en capas abstrae la vista del modelo como un todo mientras que provee suficiente detalle para entender las relaciones entre capas.

• Funcionalidad claramente definida. El diseño claramente define la separación entre la funcionalidad de cada capa.

• Alta cohesion. Cada capa contiene funcionalidad directamente relacionas con la tarea de dicha capa.

• Reutilizable. Las capas permitiéndoles ser reutilizables en otros escenarios.

**BENEFICIOS**

Los principales beneficios del estilo de arquitectura basado en capas son:

• Abstracción. Las capas permiten cambios que se realicen en un nivel abstracto.

• Rendimiento. Distribuir las capas entre múltiples sistemas puede incrementar la escalabilidad, la tolerancia a fallos y el rendimiento.

• Mejoras en Pruebas. La capacidad de realizar pruebas se beneficia de tener una interfaces bien definidas para cada capa así como de la habilidad para cambiar a diferentes implementaciones de las interfaces de cada capa.

**Arquitectura propuesta**

**CAAM**

Escaner

Código fuente .c

Linker

Generador de código

Arbol AST

Parser

Lista de Tokens

AST

Saneador

Ensamblador

**CAAM**: Este será la estructura principal, la que contendrá a todas las demás estructuras que se diseñan a formato de capas. Des aquí se podrá llamar a cada una de las capas que tiene un trabajo específico, asi se podrá ver lo que realiza cada una de ellas asi como todo el resultado final.

**Saneador**: Tiene la función de quitar los espacios antes del código y después del código, asi sean saltos de línea o tabulaciones.

**Escaner**: Quita los espacios en blanco que haya entre el código y separa las palabras a demás de etiquetarlas para tener en una lista de tokens.

**Parser**: Aquí se revisa la sintaxis del código verificando que la lista de tokens sea la correcta.

**AST:** Es una herramienta que utiliza el parser para poder revisar la sintaxis del código, tambien llamado Árboles De Sintaxis Abstracta, este devuelve el árbol construido y revisado.

**Generador de código**: Con el árbol ya recibido, se procede a generar el código para un ensamblador pasando la sintaxis a su equivalente en código ensamblador.

**Linker:** es el encargado de combinar diferentes archivos con código objeto en un único archivo, y para esto hay dos tareas fundamentales, la resolución de símbolos y el “traslado” (relocation). Por un lado, los archivos de código objeto referencian y usan símbolos, el objetivo de la resolución de símbolos es asociar cada referencia con una única definición del símbolo.

**Estándares de codificación.**

Para los estándares de codificación se utilizará de la siguiente manera.

Para los módulos principales como podría ser el “Main”,”Lexer”, etc. Se utilizará el nombre de la función, sin espacios y cada palabra (de ser el caso) empezando con mayúsculas, también se usara el idioma en inglés, para que nos resulten más fáciles de identificar con los términos vistos en clase o en algún libro de teoría de compiladores, como se muestras con el generador de código, al cual se le llamara siguiendo esto Generador de Código = “Generator”.

Para las definiciones o funciones que se agregan con “def” se utilizara: El nombre de la función que realiza y en que modulo lo hace, todo separado por un guion bajo ( \_ ).

Por ejemplo: “defp compile\_file” --- Esto nos indica que compila el archivo.

defp print\_token\_list --- Este nos dice que mostrara la lista de tokens.

Ahora con los atomos, se nombrara con lo que debe contener dentro de la función, todo esto siguiendo lo visto para las cosas pasadas (Nombre en ingles,separadas con guion bajo),pero ahora para no confundirlas no empezaran con letra mayuscula,

Un ejemplo: :node\_name, :value, :left\_node --- Todo esto muestra que node\_name debe tener el nombre del nodo, value tiene el valor y left\_node nos indica que deberá usarse el nodo izquierdo

Y por último, las variables se llamaran de forma que nos ayuden a tener un entendimiento del código mas fluido, de esta forma se llamaran como lo que se necesita usar, asignar o guardar para poder programar de una manera mejor. Como por ejemplo: file\_name , code\_snippet, words. Etc.