***Imagen que contiene Logotipo

Descripción generada automáticamenteInstituto Politécnico Nacional***

***Escuela Superior de Ingeniería Mecánica y Eléctrica Unidad Culhuacán***

***Manual Tecnico***

***Materia: Compiladores***

***Alumnos:***

***Bautista Mendoza Saul***

***Velázquez Caudillo Mauricio Emiliano***

***Equipo 1***

***Profesor: Luis Efrén Veloz Ortiz***

***Grupo: 5CM24***

***TURNO MATUTINO***

***Introducción:***

*Un compilador es un Software que traduce un programa escrito en un lenguaje de programación de alto nivel (C / C ++, COBOL, etc.) en lenguaje de máquina. Un compilador generalmente genera lenguaje ensamblador primero y luego traduce el lenguaje ensamblador al lenguaje máquina. Una utilidad conocida como «enlazador» combina todos los módulos de lenguaje de máquina necesarios en un programa ejecutable que se puede ejecutar en la computadora.*

*El proceso de compilación está dividido en varias fases, cada una de las cuales desempeña un papel específico en la traducción de código fuente a código ejecutable. Estas fases pueden agruparse en dos grandes categorías: análisis y síntesis. En la etapa de análisis, el compilador descompone y comprende el programa fuente, mientras que en la etapa de síntesis se genera el código de salida. A continuación, se describen las principales fases de un compilador:*

1. ***Análisis léxico (Scanner)****: Esta es la primera fase del compilador. Su función principal es leer el código fuente y dividirlo en unidades básicas conocidas como «tokens», que representan palabras clave, identificadores, operadores, literales, entre otros. Además, elimina los espacios en blanco y comentarios del código. Esta etapa asegura que el programa fuente esté libre de caracteres inválidos.*
2. ***Análisis sintáctico (Parser)****: En esta fase, el compilador verifica que la secuencia de tokens generada en la fase anterior cumpla con la gramática del lenguaje de programación. El resultado de esta etapa es un árbol sintáctico (parse tree), que representa la estructura jerárquica del programa fuente.*
3. ***Análisis semántico****: Esta fase valida el significado del programa fuente. Se asegura de que las operaciones y estructuras sean semánticamente válidas. Por ejemplo, verifica que las variables estén declaradas antes de su uso y que los tipos de datos sean compatibles en las operaciones.*
4. ***Generación de código intermedio****: Una vez que el programa fuente ha sido validado sintáctica y semánticamente, se genera una representación intermedia del código. Este código intermedio es independiente de la máquina y sirve como una forma de puente entre el código fuente y el código de máquina.*
5. ***Optimización del código****: Esta fase mejora el código intermedio para hacerlo más eficiente en términos de velocidad de ejecución y uso de recursos. Las optimizaciones pueden incluir la eliminación de código redundante y la reorganización de instrucciones para mejorar el rendimiento.*
6. ***Generación de código****: El código intermedio optimizado se traduce en código máquina específico para la arquitectura del sistema donde se ejecutará. Este código debe ser eficiente y debe cumplir con las especificaciones del hardware.*
7. ***Ensamblado y enlace****: Finalmente, el código máquina generado se ensambla y se combina con otros módulos o bibliotecas necesarias para producir un programa ejecutable.*

*Cada una de estas fases tiene su propia importancia y contribuye al objetivo final de traducir correctamente un programa fuente en un ejecutable eficiente. El diseño de compiladores implica el conocimiento profundo de estas fases para garantizar que el proceso de traducción sea preciso y eficiente.*

***Objetivo General:***

*Desarrollar un compilador TINY, que sea capaz de ejecutar programas básicos usando nuestros propios tokens, reglas, entre otras.*

***Objetivos Específicos:***

* *Comprender la Teoría vista durante las clases*
* *Aplicar la teoría en la programación del compilador*
* *Establecer nuestras tablas de tokens y de errores del compilador*

***Contenido:***

*En este documento se va a explicar de manera detallada código el compilador, el código se adjuntará en la entrega, por lo que aquí se explicará el contenido del código:*

*Como primera instancia, se declaran las bibliotecas usadas en el código, así de las primeras declaraciones de las funciones:*

*Texto

Descripción generada automáticamente*

*En donde se implementa el* ***typedef enum*** *donde enumera los tipos posibles de tokens (palabras clave, identificadores, números, etc.). Se define un tipo de dato struct llamado* ***typedef*** *donde s representa un token con su tipo (TokenType) y su valor (value). Además de que* ***typedef struct Symbol*** *tambien se declara como un struct y* ***Symbol*** *representa una variable declarada, con nombre, tipo (DataType), y un puntero al siguiente nodo en una lista enlazada (para la tabla de símbolos).*

*Texto

Descripción generada automáticamenteEn la siguiente sección de código, se declara la tabla de símbolos que se tienen en el compilador, que es la siguiente:*

*Se declara una lista* ***const char\* keywords,*** *de las cuales se contienen algunos de los tokens que se van a ocupar en el compilador, además de que se nos indico que se incluyeran los nombres de los integrantes del equipo como un token y desempeñara un papel en el compilador, además de que también se incluye el carácter* ***NULL,*** *después se implementa una función llamada* ***int is\_keyword,*** *donde se va a verificar si una cadena que se ingrese por el usuario, contiene una palabra clave y la otra función* ***Symbol\* find\_symbol,*** *va a buscar una variable en la tabla de símbolos que definimos.*

*En la función* ***void free\_symbol\_table*** *se libera la memoria utilizada por la tabla de símbolos, recorriendo y liberando cada nodo de la lista enlazada, y en* ***void add\_symbol*** *añade un nuevo símbolo a la tabla, verifica si el símbolo ya existe* ***(find\_symbol)*** *y crea un nuevo nodo con malloc y lo inserta al inicio de la lista enlazada.*

*Texto

Descripción generada automáticamente*

*En la siguiente parte del código, se va a definir una de las funciones mas importantes del compilador que es la de* ***get\_next\_token,*** *la cual es la que se va a encargar de obtener los tokens que se estén ingresando y que de validar si estos están definidos, es decir, se salta los espacios en blanco, busca los identificadores, números, operadores, y caracteres desconocidos:*

*Texto

Descripción generada automáticamente*

*La otra función importante de las primeras fases del compilador es el* ***void parse\_declaration****, y sus funciones son analizar una declaración de variable:*

* *Verificar el tipo (int, float).*
* *Obtener el identificador y lo agrega a la tabla de símbolos.*
* *Comprobar la asignación (=) y el fin de la declaración (;).*

*Texto

Descripción generada automáticamente*

*La otra función importante de las primeras fases del compilador es el* ***void parse\_declaration****, de la cual analiza expresiones aritméticas, valida números, identificadores, y operadores y verifica que las variables usadas estén declaradas* ***(find\_symbol).***

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*La función* ***int operation*** *tiene como objetivo evaluar una expresión que sigue un formato específico, donde se realizan operaciones aritméticas entre números o variables declaradas previamente. Estas operaciones van a estar representadas por símbolos personalizados (#, $, &, %) y tienen un comportamiento similar a los operadores matemáticos tradicionales, donde el # es suma, $ es resta, & es multiplicación y % es división.*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*La función* ***parse\_expression*** *se encarga de analizar y procesar una expresión dada como entrada para evaluar su valor final. Y se encarga de leer y procesar una expresión desde una entrada carácter por carácter, manejando números, identificadores (o variables) y operadores, además de verificar la validez sintáctica y semántica de la expresión.*

*Texto

Descripción generada automáticamente*

*La función* ***parse\_identifier*** *se asegura de la correcta sintaxis y semántica de asignaciones en un lenguaje de programación. Combina la validación sintáctica (= y ;) con la evaluación de expresiones y el almacenamiento de valores, siendo clave en el procesamiento de declaraciones y asignaciones dentro de un analizador sintáctico.*

Texto

Descripción generada automáticamente

*La función parse\_keys se encarga de que se actúe como un manejador de bloques en un lenguaje de programación personalizado. Al representar { y } con palabras clave específicas (como "mauricioemilianovelazquezcaudillo"), garantiza que los bloques estén correctamente estructurados y que todas las instrucciones dentro de ellos sean analizadas.*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*parse\_if\_statement analiza declaraciones if, evalúa la condición relacional y procesa el bloque de código asociado. Valida la correcta estructura de la condición (paréntesis, expresiones, y operador) y utiliza parse\_keys para manejar el bloque {}.*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamente*

*La función* ***saul\_expression*** *analiza y ejecuta una estructura de control tipo bucle, similar a un for en lenguajes de programación convencionales. Evalúa las tres partes principales de un bucle: inicialización, condición, y actualización (incremento), además de procesar el bloque asociado al bucle.*

*Texto

Descripción generada automáticamente*

*Texto

Descripción generada automáticamenteLa función imp\_expr evalúa una expresión y devuelve su resultado. Sirve como un componente auxiliar para analizar y manejar expresiones, distinguiendo entre números literales, identificadores declarados previamente, y expresiones complejas.*

*Texto

Descripción generada automáticamente*

*La función* ***parse\_print*** *interpreta y procesa una instrucción de impresión. Se asegura de que la sintaxis sea correcta y luego evalúa los contenidos dentro de los paréntesis para imprimirlos. Soporta cadenas entre comillas dobles y expresiones evaluadas.*

*Texto

Descripción generada automáticamente*

*La función* ***rhaeshisar*** *procesa una instrucción que simula el comportamiento de una declaración return en otros lenguajes. Evalúa una expresión opcional y asegura que la sintaxis sea correcta, terminando con un punto y coma ;. Si no se proporciona una expresión, devuelve 0 por defecto.*

*Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente*

*Programa principal*

***Ejemplo del uso de palabras reservadas:***

*rhaeshis con = 3&3;*

*rhaeshis num = 9#3;*

*saul(rhaeshis i = 0; i < con ; i = i # 2)*

*mauricioemilianovelazquezcaudillo*

*vezof(i<con$2)mauricioemilianovelazquezcaudillo*

*num = 9#i;*

*voktys (num "Hola Mundo")*

*mauricioemilianovelazquezcaudillo*

*mauricioemilianovelazquezcaudillo*

***Tabla de tokens:***

|  |  |
| --- | --- |
| ***Token*** | ***Observaciones*** |
| vezof | If |
| noreth | else |
| kash | while |
| rhaeshisar | return |
| rhaeshis | int |
| dothraki | float |
| saul | for |
| voktys | printf |
| mauricioemilianovelazquezcaudillo | {} |
| NULL | Nulo |
| # | Suma |
| $ | Resta |
| & | Multiplicacion |
| % | Division |