TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Intérpretes, IC5701

Etapa Cuatro: Generador de Código

Estudiantes:
Samir Cabrera Tabash, 2022161229
Luis Urbina Salazar, 2023156802

Primer semestre del año 2025

Índice general

Capítulo 1

Generación de Código

1.1. Introducción

La generación de código constituye la etapa final y culminante del proceso de compilación del lenguaje Notch Engine. En esta fase crítica, el compilador transforma las estructuras de datos internas y la representación semántica del programa fuente en instrucciones ejecutables de código ensamblador TASM, completando así el ciclo completo de traducción desde el código fuente de alto nivel hasta el código objeto ejecutable.

Esta etapa representa la materialización práctica de todo el análisis previo realizado durante las fases de análisis léxico, sintáctico y semántico. La información recopilada en la tabla de símbolos semánticos, las verificaciones de tipos realizadas, y la estructura sintáctica validada del programa sirven como fundamento para la generación de código eficiente y correctamente estructurado.

1.1.1. Objetivos de la Generación de Código

El generador de código del compilador Notch Engine persigue varios objetivos fundamentales que garantizan la correcta traducción y ejecución del programa objetivo:

Correctitud Semántica

El código generado debe preservar fielmente la semántica original del programa fuente. Cada construcción del lenguaje Notch Engine debe traducirse a una secuencia equivalente de instrucciones en ensamblador que mantenga el comportamiento esperado. Esto incluye la correcta implementación de operaciones aritméticas, estructuras de control, manejo de variables y gestión de

memoria.

Gestión de Recursos

El generador debe administrar eficientemente los recursos del sistema objetivo, incluyendo registros del procesador, segmentos de memoria (datos y código), y la pila del sistema. Se implementa un control estricto para prevenir el desbordamiento de segmentos, constituyendo un error irrecuperable que debe detectarse durante la generación.

Interfaz con Runtime Library

Todas las operaciones complejas desarrolladas en etapas anteriores se integran en una biblioteca de tiempo de ejecución unificada (Runtime Library). Esta biblioteca proporciona las funcionalidades básicas del lenguaje como operaciones de entrada/salida, manipulación de cadenas, y gestión de estructuras de datos complejas.

1.1.2. Arquitectura del Generador

El generador de código se estructura siguiendo un diseño modular que separa claramente las responsabilidades y facilita el mantenimiento y extensión del sistema:

Tabla de Símbolos de Generación

Se mantiene una tabla de símbolos específica para la generación de código, independiente de la tabla semántica utilizada en etapas anteriores. Esta separación permite mantener información específica de generación como direcciones de memoria, offsets de variables, y etiquetas de código sin contaminar el análisis semántico.

Manejo de Segmentos

El generador implementa un sistema de gestión de segmentos que controla automáticamente el uso del espacio en los segmentos de datos y código. Se mantiene un registro preciso del espacio utilizado y disponible, generando errores irrecuperables cuando se alcancen los límites de capacidad.

Generación de Portadas

Cada programa generado incluye automáticamente una portada de identificación que contiene información del proyecto, autores, y metadatos relevantes. Esta característica asegura la trazabilidad y documentación adecuada del código generado.

1.1.3. Estrategia de Traducción

La traducción del código fuente a ensamblador sigue una estrategia dirigida por la sintaxis, donde cada construcción sintáctica del lenguaje Notch Engine tiene asociada una rutina específica de generación de código. Este enfoque garantiza la consistencia y facilita la verificación de la correctitud de la traducción.

Traducción de Expresiones

Las expresiones aritméticas y lógicas se traducen utilizando una estrategia de evaluación en pila, donde los operandos se cargan en la pila del sistema y los operadores se implementan mediante llamadas a rutinas especializadas de la Runtime Library.

Estructuras de Control

Las estructuras de control como `TARGET/HITMISS` (equivalentes a if/else) y los bucles se implementan mediante etiquetas y saltos condicionales, manteniendo la semántica original del flujo de control del programa fuente.

Gestión de Variables

Las variables se mapean a ubicaciones específicas en memoria, considerando su tipo de dato, alcance, y tiempo de vida. Se implementa un sistema de direccionamiento que permite el acceso eficiente tanto a variables locales como globales.

1.1.4. Manejo de Errores en Generación

Durante la fase de generación de código, todos los errores se consideran irrecuperables y provocan la terminación inmediata del proceso de compilación mediante una estrategia de pánico. Esta decisión de diseño reconoce que los errores en esta etapa típicamente indican problemas fundamentales que no pueden ser corregidos automáticamente.

Los errores de tiempo de ejecución del código generado siguen la misma filosofía, terminando la ejecución del programa de manera controlada y proporcionando información diagnóstica útil para la depuración.

1.1.5. Integración con Etapas Anteriores

El generador de código se integra seamlessly con las etapas previas del compilador, utilizando la información recopilada durante el análisis léxico, sintáctico y semántico. La tabla de símbolos semánticos proporciona información de tipos y alcances, mientras que el árbol sintáctico abstracto guía la estructura de la generación de código.

Esta integración permite aprovechar al máximo el trabajo realizado en etapas anteriores, minimizando la duplicación de esfuerzo y asegurando la consistencia global del compilador.

El resultado final de esta etapa es un programa ejecutable en ensamblador TASM que preserva fielmente la semántica del programa original en Notch Engine, listo para su ensamblado y ejecución en el sistema objetivo.

1.2. Análisis de Código

1.2.1. Chequeo de Variables No Inicializadas

Definición

Este chequeo se encarga de verificar si una variable está siendo utilizada antes de haber sido inicializada con un valor válido. Durante el análisis de flujo de datos, se rastrea el estado de inicialización de cada variable declarada, manteniendo un registro de aquellas que han recibido un valor y aquellas que permanecen en estado no inicializado.

Cuando se detecta el uso de una variable no inicializada, el sistema puede aplicar diferentes estrategias: emitir una advertencia y asignar un valor por defecto según el tipo de dato, o generar un error que detenga la compilación dependiendo de las políticas de seguridad establecidas.

Este mecanismo previene comportamientos impredecibles en tiempo de ejecución y ayuda a identificar errores lógicos en el código fuente, especialmente en casos donde el programador asume incorrectamente que una variable ha sido inicializada previamente.

Ubicación

Se ejecuta cada vez que se referencia una variable en una expresión, asignación o como parámetro de función. El análisis se realiza antes de la generación de código para garantizar que todas las variables utilizadas tengan valores válidos.

1.2.2. Chequeo de Código Muerto

Definición

El análisis de código muerto (dead code) identifica segmentos de código que nunca serán ejecutados durante la ejecución normal del programa. Esto incluye instrucciones que aparecen después de declaraciones de terminación como `WORLDSAVE`, declaraciones de retorno incondicionales, o bloques que se encuentran en ramas de control inalcanzables.

La detección de código muerto es importante tanto para la optimización del programa final como para alertar al programador sobre posibles errores lógicos. El código inalcanzable puede indicar condiciones mal estructuradas, bucles con condiciones incorrectas o declaraciones de control de flujo mal ubicadas.

El análisis utiliza técnicas de análisis de flujo de control para determinar qué instrucciones pueden ser alcanzadas desde el punto de entrada del programa, marcando como código muerto aquellas que no pueden ser ejecutadas bajo ninguna circunstancia.

Ubicación

Se realiza durante el análisis de flujo de control del programa, después de construir el grafo de flujo de control y antes de la generación de código optimizado.

1.2.3. Chequeo de Tipos en Operaciones Binarias

Definición

Este chequeo valida la compatibilidad de tipos en operaciones binarias, asegurando que los operandos involucrados en operaciones aritméticas, lógicas o de comparación sean semánticamente compatibles. El sistema verifica que no se realicen operaciones inválidas como sumar una cadena (`SPIDER`) con un entero (`STACK`).

Para cada operador binario, existe un conjunto de reglas que define qué combinaciones de tipos son válidas. Por ejemplo, las operaciones aritméticas

requieren operandos numéricos (`STACK` o `GHAST`), mientras que las operaciones de concatenación pueden requerir tipos específicos como cadenas.

En casos donde los tipos no son directamente compatibles pero existe una conversión implícita válida, el sistema puede aplicar la conversión automáticamente o emitir una advertencia según las políticas de tipado establecidas.

Ubicación

Se activa al encontrar operadores aritméticos $(+, -, *, /)$, operadores lógicos $(\&\&, ||)$, operadores de comparación $(==, != , <, >)$ y otros operadores binarios durante el análisis semántico.

1.2.4. Chequeo de Límites en SHELF

Definición

Este chequeo específico para estructuras de tipo `SHELF` (equivalentes a arreglos) verifica que todos los accesos indexados se mantengan dentro de los límites declarados de la estructura. Se valida tanto que el índice no sea negativo como que no exceda el tamaño máximo definido durante la declaración.

El análisis puede ser estático (cuando los índices son constantes conocidas en tiempo de compilación) o dinámico (insertando verificaciones en tiempo de ejecución para índices calculados). Para índices estáticos, se puede determinar con certeza si el acceso es válido, mientras que para índices dinámicos se generan verificaciones de rango.

La violación de límites en arreglos es una fuente común de errores graves que pueden llevar a corrupción de memoria, accesos a datos no válidos o comportamientos impredecibles del programa.

Ubicación

Se ejecuta cada vez que se detecta un acceso indexado a una estructura `SHELF`, tanto en operaciones de lectura como de escritura.

1.2.5. Chequeo de Constantes Reasignadas

Definición

Este chequeo garantiza la inmutabilidad de las constantes declaradas con el tipo `OBSIDIAN`. Una vez que una constante ha sido inicializada con un valor, cualquier intento posterior de modificar dicho valor debe ser rechazado como un error semántico.

El sistema mantiene un registro de todas las constantes declaradas y sus valores asociados. Durante el análisis, cada operación de asignación se verifica contra este registro para asegurar que no se esté intentando modificar una constante existente.

Este mecanismo es fundamental para mantener las garantías de inmutabilidad que proporcionan las constantes, permitiendo optimizaciones del compilador y asegurando comportamientos predecibles en el código.

Ubicación

Se activa en cada operación de asignación, verificando si el identificador de destino corresponde a una constante previamente declarada.

1.2.6. Chequeo de Argumentos en Llamadas a Funciones

Definición

Este chequeo valida que las llamadas a funciones o procedimientos se realicen con el número correcto de argumentos y que cada argumento sea del tipo esperado según la signatura de la función. Se verifica tanto la cantidad de parámetros como la compatibilidad de tipos entre los argumentos proporcionados y los parámetros formales.

Para cada función declarada, se almacena su signatura completa incluyendo el número de parámetros, sus tipos y el tipo de retorno. Durante una llamada a función, esta información se utiliza para validar que la invocación sea correcta.

En caso de discrepancias, el sistema puede permitir conversiones implícitas de tipos compatibles o rechazar la llamada completamente dependiendo de la severidad de la incompatibilidad.

Ubicación

Se ejecuta en cada punto donde se realiza una llamada a función o procedimiento, antes de generar el código de invocación.

1.2.7. Chequeo de Retorno en Funciones

Definición

Este chequeo asegura que las funciones que declaran un tipo de retorno específico efectivamente retornen un valor del tipo correcto en todos los ca-

minos de ejecución posibles. Se verifica que no existan rutas de ejecución que terminen sin una declaración de retorno válida.

El análisis examina todos los caminos posibles a través del cuerpo de la función, identificando aquellos que no terminan en una declaración de retorno explícita. Para funciones con tipo de retorno, esto constituye un error semántico que debe ser corregido.

También se valida que el tipo del valor retornado sea compatible con el tipo declarado en la signatura de la función, aplicando las mismas reglas de compatibilidad de tipos utilizadas en otras verificaciones.

Ubicación

Se realiza al finalizar el análisis del cuerpo de cada función, verificando todos los caminos de ejecución posibles.

1.2.8. Chequeo de Archivos No Cerrados

Definición

Este chequeo identifica recursos de archivo (tipo `BOOK`) que han sido abiertos durante la ejecución del programa pero no han sido explícitamente cerrados antes de la terminación. La gestión incorrecta de recursos de archivo puede llevar a fugas de recursos y problemas de rendimiento.

El sistema mantiene un registro de todos los archivos abiertos y sus estados correspondientes. Al final del análisis del programa, se verifica que todos los archivos abiertos hayan sido cerrados apropiadamente mediante las operaciones correspondientes.

Esta verificación es especialmente importante en lenguajes que no cuentan con recolección automática de basura para recursos del sistema, donde la responsabilidad de liberar recursos recae completamente en el programador.

Ubicación

Se ejecuta como parte del análisis final del programa, antes de la declaración `WORLDSAVE`, verificando el estado de todos los recursos de archivo.

1.2.9. Chequeo de Bucles Infinitos

Definición

Este análisis identifica bucles que potencialmente nunca terminarán debido a condiciones que permanecen constantes o variables de control que nunca

se modifican dentro del cuerpo del bucle. Se examina si las variables utilizadas en la condición del bucle son modificadas en algún punto del cuerpo del bucle.

El análisis puede detecter bucles infinitos obvios (como `while(true)`) así como casos más sutiles donde las variables de control no se actualizan de manera que permita que la condición del bucle eventualmente se vuelva falsa.

Aunque algunos bucles infinitos pueden ser intencionales (como bucles principales de programas), en la mayoría de casos representan errores lógicos que deben ser señalados al programador.

Ubicación

Se activa al analizar estructuras de control repetitivas como `TARGET` con condiciones de bucle, examinando el cuerpo del bucle en busca de modificaciones a las variables de la condición.

1.2.10. Chequeo de Shadowing de Variables

ónDefinición Este chequeo detecta situaciones donde una variable declarada en un ámbito interno oculta (shadow) a una variable con el mismo nombre en un ámbito externo. Aunque sintácticamente válido, el shadowing puede crear confusión y errores lógicos difíciles de detectar.

El análisis mantiene un registro de los ámbitos anidados y las variables declaradas en cada uno. Cuando se detecta una nueva declaración de variable, se verifica si existe una variable con el mismo nombre en algún ámbito externo accesible.

Dependiendo de las políticas del compilador, el shadowing puede generar una advertencia para alertar al programador sobre la potencial ambigüedad, sin necesariamente impedir la compilación.

Ubicación

Se ejecuta cada vez que se declara una nueva variable, verificando contra todas las variables declaradas en ámbitos externos accesibles.

1.2.11. Chequeo de Conversiones Implícitas Peligrosas

Definición

Este chequeo identifica conversiones automáticas de tipos que pueden resultar en pérdida de datos o comportamientos inesperados. Por ejemplo,

la conversión de un número en punto flotante (`GHAST`) a un entero (`STACK`) resulta en la pérdida de la parte fraccionaria.

Se analizan todas las asignaciones y operaciones que involucran tipos diferentes pero compatibles, evaluando si la conversión implícita puede resultar en pérdida de información o precisión. El sistema puede emitir advertencias para alertar al programador sobre estas situaciones.

Las conversiones consideradas peligrosas incluyen truncamiento de números flotantes, desbordamiento de enteros, y conversiones entre tipos de diferentes rangos de valores.

Ubicación

Se activa en asignaciones, operaciones aritméticas y paso de parámetros donde se detecten tipos diferentes que requieren conversión implícita.

1.2.12. Chequeo de Recursos sin Liberar

Definición

Este análisis identifica recursos del sistema (como memoria asignada dinámicamente para estructuras `SHELF`) que han sido asignados pero no han sido explícitamente liberados antes de que salgan de su ámbito de validez.

El sistema rastrea las operaciones de asignación y liberación de recursos, manteniendo un registro del estado de cada recurso. Al finalizar el análisis de un ámbito, se verifica que todos los recursos asignados en ese ámbito hayan sido apropiadamente liberados.

Esta verificación es crucial para prevenir fugas de memoria y otros problemas relacionados con la gestión de recursos que pueden degradar el rendimiento del programa o agotar los recursos del sistema. ónUbicación Se realiza al finalizar el análisis de cada ámbito donde se han asignado recursos, y como parte del análisis global antes de la terminación del programa.

1.2.13. Chequeo de Pre/Post Condiciones

Definición

Este chequeo valida que se cumplan las condiciones previas necesarias antes de ejecutar operaciones críticas. El ejemplo más común es la verificación de división por cero, pero puede extenderse a otras condiciones como acceso a punteros nulos o validación de rangos.

Para operaciones que tienen prerrequisitos específicos, el análisis examina los valores de los operandos (cuando son conocidos en tiempo de compila-

ción) o inserta verificaciones en tiempo de ejecución para garantizar que las condiciones se cumplan.

Este tipo de verificación es fundamental para la robustez del programa, previniendo errores en tiempo de ejecución que podrían resultar en terminación anormal del programa o comportamientos indefinidos.

Ubicación

Se ejecuta antes de operaciones críticas como divisiones, accesos a memoria, y otras operaciones que tienen precondiciones específicas.

1.2.14. Chequeo de Consistencia en WORLDNAME/-WORLDSAVE

Definición

Este chequeo verifica la estructura fundamental del programa, asegurando que comience con `WORLDNAME` y termine con `WORLDSAVE`. Esta verificación es crítica para la correcta interpretación del programa y establece los límites claros del código ejecutable.

Se valida no solo la presencia de estos tokens sino también su posición correcta dentro de la estructura del programa. `WORLDNAME` debe ser el primer token significativo, mientras que `WORLDSAVE` debe ser el último, sin código ejecutable después de él.

La ausencia o mal posicionamiento de estos elementos estructurales indica un error fundamental en la organización del programa que debe ser corregido antes de proceder con la compilación.

Ubicación

Se realiza como parte del análisis estructural inicial y final del programa, verificando la presencia y posición correcta de los delimitadores principales.

1.2.15. Chequeo de Rendimiento

Definición

Este análisis identifica patrones de código que pueden resultar en bajo rendimiento, como bucles anidados con operaciones costosas, accesos repetitivos a estructuras de datos complejas, o algoritmos con complejidad innecesariamente alta.

El sistema examina la estructura del código en busca de patrones conocidos que típicamente resultan en problemas de rendimiento. Esto incluye operaciones costosas dentro de bucles, accesos no optimizados a arreglos, y patrones de recursión ineficientes.

Aunque estos patrones no constituyen errores semánticos, su identificación temprana permite al programador optimizar el código antes de que se conviertan en problemas de rendimiento significativos en la ejecución.

Ubicación

Se ejecuta como parte del análisis de optimización, examinando estructuras repetitivas, llamadas a funciones, y patrones de acceso a datos para identificar oportunidades de mejora. “‘

1.3. Runtime Library

1.3.1. Introducción a la Runtime Library

La Runtime Library del compilador Notch Engine constituye una biblioteca unificada de rutinas en tiempo de ejecución que implementa todas las operaciones fundamentales del lenguaje. Esta biblioteca representa la culminación del trabajo realizado en etapas anteriores, integrando las operaciones aritméticas, lógicas, de comparación, entrada/salida y manipulación de datos desarrolladas durante el análisis léxico, sintáctico y semántico.

La biblioteca se genera automáticamente como un archivo ensamblador independiente (`runtime_library.asm`) que contiene todas las rutinas necesarias para soportar la ejecución de programas Notch Engine compilados.

1.3.2. Arquitectura de la Runtime Library

Estructura Modular

La Runtime Library se organiza en módulos funcionales que agrupan operaciones relacionadas:

- **Operaciones Aritméticas**: Suma, resta, multiplicación, división y módulo para tipos `STACK`
- **Operaciones de Comparación**: Todas las comparaciones relacionales y de igualdad
- **Operaciones Lógicas**: AND, OR, NOT y XOR para tipos `TORCH`

- **Operaciones de Incremento/Decremento**: Implementación de `soulsand` y `netherrack`
- **Entrada/Salida**: Rutinas para lectura y escritura de diferentes tipos de datos
- **Manipulación de Cadenas**: Operaciones básicas con tipos `SPIDER`

Convención de Llamadas

Todas las rutinas de la Runtime Library siguen una convención de llamadas estándar:

- Los parámetros se pasan a través de la pila del sistema
- El resultado se retorna en el registro AX para operaciones que retornan valores
- Se preservan los registros BP y otros registros críticos
- Se realiza limpieza automática de la pila al retornar

1.3.3. Implementación de Operaciones

Operaciones Aritméticas Enteras

Las operaciones aritméticas para el tipo `STACK` se implementan como rutinas optimizadas que manejan casos especiales:

SUMAR_ENTEROS: Implementa la operación `:+` sumando dos valores enteros pasados por pila.

RESTAR_ENTEROS: Implementa la operación `:-` restando el segundo operando del primero.

MULTIPLICAR_ENTEROS: Implementa la operación `:*` utilizando la instrucción IMUL para multiplicación con signo.

DIVIDIR_ENTEROS: Implementa la operación `:/` con verificación automática de división por cero, retornando 0 en caso de error.

MODULO_ENTEROS: Nueva rutina que implementa la operación módulo `:%`, retornando el resto de la división entera.

Operaciones de Incremento y Decremento

INCREMENTAR_ENTERO: Implementa la operación `soulsand`, incrementando en 1 el valor pasado como parámetro.

DECREMENTAR_ENTERO: Implementa la operación `netherrack`, decrementando en 1 el valor pasado como parámetro.

Operaciones de Comparación

Se implementan todas las operaciones de comparación necesarias para estructuras de control:

COMPARAR_IGUAL: Implementa `is`, comparando si dos valores son iguales.

COMPARAR_DIFERENTE: Implementa `isNot`, comparando si dos valores son diferentes.

COMPARAR_MAYOR: Implementa `>`, comparando si el primer valor es mayor que el segundo.

COMPARAR_MENOR: Implementa `<`, comparando si el primer valor es menor que el segundo.

COMPARAR_MAYOR_IGUAL: Implementa `>=`, comparando si el primer valor es mayor o igual que el segundo.

COMPARAR_MENOR_IGUAL: Implementa `<=`, comparando si el primer valor es menor o igual que el segundo.

Operaciones Lógicas Booleanas

Para el manejo del tipo `TORCH` se implementan operaciones lógicas estándar:

AND_LOGICO: Implementa la operación AND lógica entre dos valores booleanos.

OR_LOGICO: Implementa la operación OR lógica entre dos valores booleanos.

NOT_LOGICO: Implementa la negación lógica de un valor booleano.

XOR_LOGICO: Implementa la operación XOR lógica entre dos valores booleanos.

Operaciones de Entrada/Salida

La biblioteca incluye rutinas especializadas para cada tipo de dato:

LEER_ENTERO: Lee un valor entero desde la entrada estándar y lo retorna como tipo `STACK`.

MOSTRAR_ENTERO: Muestra un valor entero en la salida estándar con formato apropiado.

LEER_CARACTER: Lee un carácter individual desde la entrada estándar.

MOSTRAR_CARACTER: Muestra un carácter en la salida estándar.

LEER_BOOLEANO: Lee un valor booleano, aceptando representaciones textuales como .ºN/OFF".

MOSTRAR_BOOLEANO: Muestra un valor booleano como .ºN.º .ºFF"según corresponda.

1.3.4. Operaciones de Cadenas

Manipulación Básica

Para el tipo `SPIDER` se implementan operaciones fundamentales:

CONCATENAR_STRINGS: Une dos cadenas de caracteres creando una nueva cadena resultado.

LONGITUD_STRING: Calcula y retorna la longitud en caracteres de una cadena.

1.3.5. Generación Automática

Proceso de Creación

La Runtime Library se genera automáticamente mediante la clase `GeneradorConRuntime` que:

1. Crea el archivo `runtime_library.asm` con todas las rutinas necesarias
2. Incluye comentarios descriptivos y metadatos de generación
3. Implementa manejo de errores en operaciones críticas como división por cero
4. Optimiza el código generado para eficiencia en tiempo de ejecución

Integración con el Código Generado

El generador de código principal integra la Runtime Library mediante:

- Inclusión automática del archivo mediante directiva `INCLUDE`
- Generación de llamadas apropiadas a las rutinas según las operaciones detectadas
- Manejo automático de parámetros y valores de retorno
- Verificación de compatibilidad de tipos antes de generar llamadas

1.3.6. Manejo de Errores en Runtime

Errores Irrecuperables

La Runtime Library implementa detección y manejo de errores críticos:

- **División por cero**: Se detecta automáticamente y se retorna un valor seguro (0)
- **Desbordamiento de enteros**: Se detecta en operaciones que pueden causar overflow
- **Acceso fuera de rango**: Para operaciones con estructuras `SHELF`

Estrategia de Recuperación

Cuando se detecta un error en runtime:

1. Se registra el error interno para diagnóstico
2. Se retorna un valor por defecto seguro según el tipo de operación
3. Se continúa la ejecución para permitir terminación controlada
4. Se proporciona información de depuración cuando es posible

1.3.7. Optimizaciones Implementadas

Eficiencia en Llamadas

Las rutinas están optimizadas para:

- Minimizar el uso de registros y preservar estado
- Realizar limpieza automática de pila
- Utilizar instrucciones nativas del procesador cuando es posible
- Evitar operaciones redundantes o costosas

Gestión de Memoria

La biblioteca implementa gestión eficiente de memoria:

- Uso mínimo de memoria para variables temporales
- Reutilización de registros para operaciones múltiples
- Limpieza automática de recursos temporales
- Prevención de fragmentación de memoria

1.3.8. Compatibilidad y Portabilidad

Compatibilidad con TASM

Toda la Runtime Library está diseñada para ser compatible con el ensamblador TASM:

- Utiliza sintaxis estándar de TASM para todas las instrucciones
- Implementa convenciones de llamada compatibles
- Usa directivas de ensamblador apropiadas
- Mantiene compatibilidad con diferentes versiones de TASM

Extensibilidad

La arquitectura permite extensión futura:

- Adición de nuevas rutinas sin modificar las existentes
- Soporte para tipos de datos adicionales
- Integración de optimizaciones específicas de hardware
- Personalización de comportamiento según necesidades específicas

La Runtime Library representa así la base sólida sobre la cual se ejecutan todos los programas compilados con Notch Engine, proporcionando funcionalidad robusta, eficiente y extensible para todas las operaciones fundamentales del lenguaje.

1.4. Ejecución de Programa

1.4.1. Introducción al Proceso de Ejecución

La ejecución de programas Notch Engine compilados representa la culminación de todo el proceso de compilación. Una vez que el código fuente ha sido procesado a través de las etapas de análisis léxico, sintáctico, semántico y generación de código, el resultado es un programa ejecutable en ensamblador TASM que debe ser ensamblado y ejecutado en el sistema objetivo.

Esta etapa final involucra la coordinación entre el código generado por el compilador, la Runtime Library, y el sistema operativo objetivo para producir la ejecución correcta del programa original.

1.4.2. Arquitectura de Ejecución

Segmentación de Memoria

El programa ejecutable generado utiliza una arquitectura de memoria segmentada que divide el espacio de direcciones en tres segmentos principales:

Segmento de Pila (`STACK`): Reserva 256 palabras (512 bytes) para el manejo de la pila del sistema. Este segmento gestiona las llamadas a procedimientos, parámetros de funciones, variables locales temporales y el estado de ejecución durante las llamadas a la Runtime Library.

Segmento de Datos (`DATA`): Contiene todas las variables declaradas en el programa, constantes, cadenas literales y estructuras de datos como arreglos (`SHELF`). Este segmento se organiza secuencialmente según el orden de declaración de variables.

Segmento de Código (`CODE`): Alberga todas las instrucciones ejecutables del programa, incluyendo el procedimiento principal (`MAIN`), rutinas auxiliares y puntos de entrada y salida del sistema.

Inicialización del Sistema

Al iniciar la ejecución, el programa realiza una secuencia de inicialización crítica:

1. **Configuración de Segmentos**: Se establece la configuración inicial mediante `ASSUME` que define la asociación entre registros de segmento y segmentos de memoria.

2. **Inicialización de Registros**: El registro DS (Data Segment) se inicializa para apuntar al segmento de datos, permitiendo el acceso correcto a variables y constantes.

3. **Configuración de Pila**: El sistema configura automáticamente el segmento de pila y establece los punteros SP (Stack Pointer) y BP (Base Pointer) en posiciones iniciales válidas.

1.4.3. Modelo de Ejecución

Flujo de Control Principal

La ejecución comienza en el procedimiento `MAIN`, que actúa como punto de entrada único del programa. Desde este punto:

- Se ejecutan secuencialmente las instrucciones generadas a partir del código fuente

- Las estructuras de control (`TARGET/HITMISS`) se implementan mediante saltos condicionales y etiquetas
- Los bucles se ejecutan mediante combinaciones de comparaciones y saltos
- Las operaciones complejas se delegan a la Runtime Library mediante llamadas a procedimientos

Gestión de Variables y Memoria

Durante la ejecución, el sistema gestiona diferentes tipos de variables según su naturaleza:

Variables Simples (`STACK`, `GHAST`, `TORCH`): Se almacenan directamente en el segmento de datos con acceso directo mediante direccionamiento absoluto.

Constantes (`OBSIDIAN`): Se almacenan en el segmento de datos pero con protección implícita contra modificación (verificada durante la compilación).

Cadenas (`SPIDER`): Se almacenan como secuencias de bytes terminadas en '$' siguiendo las convenciones de DOS/TASM.

Arreglos (`SHELF`): Se implementan como bloques contiguos de memoria con cálculo de direcciones basado en índices y tamaño de elementos.

1.4.4. Integración con Runtime Library

Mecanismo de Llamadas

Todas las operaciones complejas del lenguaje Notch Engine se ejecutan mediante llamadas a rutinas de la Runtime Library:

1. **Preparación de Parámetros**: Los operandos se colocan en la pila en orden inverso (último parámetro primero).
2. **Invocación de Rutina**: Se ejecuta la instrucción `CALL` correspondiente a la operación deseada.
3. **Recuperación de Resultado**: El resultado se obtiene del registro AX tras la ejecución de la rutina.
4. **Limpieza de Pila**: La rutina se encarga automáticamente de limpiar los parámetros de la pila.

Operaciones Típicas

Durante la ejecución se realizan llamadas frecuentes a rutinas como:

- `SUMAR_ENTEROS`, `RESTAR_ENTEROS`, etc. para operaciones aritméticas
- `COMPARAR_IGUAL`, `COMPARAR_MAYOR`, etc. para evaluación de condiciones
- `MOSTRAR_ENTERO`, `LEER_CARACTER`, etc. para operaciones de entrada/salida
- `AND_LOGICO`, `OR_LOGICO`, etc. for operaciones lógicas booleanas

1.4.5. Manejo de Entrada/Salida

Operaciones `dropperTorch` y `dropperSpider`

Las operaciones de salida del lenguaje Notch Engine se traducen a llamadas específicas:

`dropperTorch`: Se traduce a la rutina `MOSTRAR_ENTERO` o `MOSTRAR_BOOLEANO` según el tipo de dato, mostrando el valor en la consola del sistema.

`dropperSpider`: Se traduce a `MOSTRAR_CARACTER` o rutinas de manejo de cadenas para mostrar texto en la salida estándar.

Operaciones de Lectura

Las operaciones implícitas de lectura se manejan mediante:

- `LEER_ENTERO` para captura de valores numéricos
- `LEER_CARACTER` para captura de caracteres individuales
- `LEER_BOOLEANO` para captura de valores lógicos con conversión automática

1.4.6. Control de Flujo en Ejecución

Estructuras Condicionales

Las estructuras `TARGET/HITMISS` se ejecutan mediante:

1. Evaluación de la condición mediante llamadas a rutinas de comparación
2. Salto condicional a la etiqueta correspondiente según el resultado
3. Ejecución del bloque `HITMISS` (then) o continuar al bloque alternativo
4. Confluencia en punto común tras completar la estructura condicional

Estructuras Repetitivas

Los bucles implícitos en las estructuras `TARGET` se ejecutan mediante:

- Establecimiento de etiqueta de inicio del bucle
- Evaluación de condición de continuación
- Salto condicional de salida del bucle si la condición es falsa
- Ejecución del cuerpo del bucle
- Salto incondicional de regreso al inicio para reevaluación

1.4.7. Terminación del Programa

Secuencia de Terminación Normal

Un programa Notch Engine termina normalmente mediante la secuencia:

1. Ejecución de la declaración `WORLDSAVE` traducida como punto de terminación
2. Ejecución de la secuencia de terminación del sistema operativo (`MOV AH, 4Ch; INT 21h`)
3. Liberación automática de recursos del sistema (memoria, archivos, etc.)
4. Retorno del control al sistema operativo con código de salida 0

Manejo de Errores en Tiempo de Ejecución

Cuando ocurren errores durante la ejecución:

- **Errores de Runtime Library**: Se manejan internamente retornando valores seguros
- **Errores del Sistema**: Se propagan al sistema operativo con códigos de error apropiados
- **Errores de Acceso a Memoria**: Generan terminación controlada con información diagnóstica
- **Desbordamiento de Pila**: Se detecta y maneja con terminación segura

1.4.8. Optimizaciones de Ejecución

Eficiencia en Runtime

Durante la ejecución, se aplican varias optimizaciones:

- **Reutilización de Registros**: Los valores se mantienen en registros cuando es posible para evitar accesos a memoria
- **Eliminación de Operaciones Redundantes**: El código generado evita cálculos innecesarios
- **Optimización de Saltos**: Se minimizan los saltos condicionales e incondicionales
- **Uso Eficiente de la Pila**: Se minimiza el uso de la pila para operaciones temporales

Gestión de Recursos

El sistema de ejecución implementa gestión eficiente de recursos:

- Liberación automática de memoria temporal tras operaciones complejas
- Reutilización de espacios de memoria para variables de ámbito limitado
- Optimización del uso de registros del procesador
- Minimización de accesos a memoria externa

1.4.9. Depuración y Diagnóstico

Información de Depuración

El código generado incluye información útil para depuración:

- Comentarios que indican la operación original del código fuente
- Etiquetas descriptivas que facilitan el seguimiento del flujo de ejecución
- Preservación de nombres de variables originales cuando es posible
- Información de línea y contexto en caso de errores

Herramientas de Diagnóstico

Para facilitar la depuración y análisis:

- El generador produce código con estructura clara y comentarios descriptivos
- Se mantiene trazabilidad entre código fuente y código generado
- Se proporcionan mensajes de error descriptivos en caso de fallas
- Se incluyen verificaciones de sanidad en puntos críticos

1.4.10. Compatibilidad y Portabilidad

Compatibilidad con Sistemas DOS

El código generado es totalmente compatible con:

- Sistemas DOS y compatibles (incluyendo emuladores modernos)
- Ensamblador TASM en diferentes versiones
- Arquitecturas x86 de 16 bits
- Convenciones estándar de llamadas del sistema DOS

Portabilidad del Código

La arquitectura de ejecución permite:

- Ejecución en diferentes entornos compatibles con x86
- Adaptación a diferentes tamaños de memoria disponible
- Personalización de la Runtime Library para necesidades específicas
- Extensión futura para arquitecturas adicionales

El modelo de ejecución implementado garantiza así que los programas Notch Engine compilados se ejecuten de manera correcta, eficiente y robusta en el sistema objetivo, preservando fielmente la semántica del programa original mientras aprovechan las capacidades del hardware y sistema operativo subyacente.

1.5. Prueba de Implementacion

Para comprobar la prueba de implementacion se va a usar de referencia el Examen 3 del curso de Introduccion a la Programacion (IC1802) en el primer semestre del 2023 a cargo del profesor Aurelio Sanabria.

En la carpeta de pruebas viene anexado el PDF con el enunciado completo, a continuacion se anexan imagenes de las preguntas y los codigos que dan solucion a las mismas.

1.5.1. Pregunta 1

Pregunta 1. Matriz simétrica

Una matriz simétrica es una matriz cuadrada que es **idéntica** a su transpuesta

Puesto matemáticamente: $A = A^T$

Puesto ejemplificamente:

$$A = \begin{vmatrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 0 \end{vmatrix} \qquad A^T = \begin{vmatrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 0 \end{vmatrix}$$

Programe una función que permita determinar si una matriz es una matriz simétrica

La función debe llamarse de la siguiente forma: **`es_matriz_simetrica(matriz)`** → `True/False`

donde:

1. **matriz** es una matriz de acuerdo a la definición vista en clase (sino '*Error01*')
2. **matriz** es una matriz cuadrada (sino '*Error02*')
3. La **salida** debe ser un valor de verdad *True* si es simétrica, *False* en caso contrario.

Figura 1.1: Pregunta 1

Codigo de Notch Engine

```
WorldName MatrizSimetrica:

Inventory
    Shelf[10][10] Stack matrizEjemplo;

CraftingTable
```

```
    Spell es_matriz_simetrica(Shelf :: matriz) -> Torch
    PolloCrudo
        $$ Verificar que el parámetro es una matriz (Error01)
        target chunk(matriz) >> Stack < 2 craft hit
        PolloCrudo
            respawn "Error01";
        PolloAsado

        $$ Obtener dimensiones de la matriz
        Stack filas = #(matriz);
        Stack columnas = #(matriz[0]);

        $$ Verificar que es cuadrada (Error02)
        target filas isNot columnas craft hit
        PolloCrudo
            respawn "Error02";
        PolloAsado

        $$ Comparar matriz con su transpuesta
        walk i set 0 to filas - 1 craft
        PolloCrudo
            walk j set 0 to columnas - 1 craft
            PolloCrudo
                target matriz[i][j] isNot matriz[j][i] craft hit
                PolloCrudo
                    respawn Off;
                PolloAsado
            PolloAsado
        PolloAsado

        $$ Si llegamos aquí, es simétrica
        respawn On;
    PolloAsado

SpawnPoint
    PolloCrudo
        $$ Ejemplo de uso
        $$ Matriz simétrica 3x3
        Shelf[3][3] Stack matriz1 = [
            [1, 2, 3],
            [2, 4, 5],
```

```
            [3, 5, 6]
        ];

        $$ Matriz no simétrica 3x3
        Shelf[3][3] Stack matriz2 = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]
        ];

        $$ No es matriz (Error01)
        Stack noMatriz = 5;

        $$ No es cuadrada (Error02)
        Shelf[2][3] Stack matrizRectangular = [
            [1, 2, 3],
            [4, 5, 6]
        ];

        $$ Pruebas
        dropperSpider("Matriz 1 (simétrica):");
        dropperTorch(es_matriz_simetrica(matriz1));

        dropperSpider("Matriz 2 (no simétrica):");
        dropperTorch(es_matriz_simetrica(matriz2));

        dropperSpider("No es matriz:");
        dropperSpider(es_matriz_simetrica(noMatriz) >> Spider);

        dropperSpider("Matriz rectangular:");
        dropperSpider(es_matriz_simetrica(matrizRectangular) >> Spider);
    PolloAsado

worldSave
```

1.5.2. Pregunta 2

Pregunta 2. Verificar ganador del *Felis silvestris catus*

Su profesor del semestre entrante los puso a hacer un proyecto programado de un juego de *Felis silvestris catus*. Como ustedes son muy previsores van a dejar listo desde este semestre una función que verifique si ya hay un ganador para el juego.

Ejemplo de matriz de entrada:

```
[['x', 'o', 'x'], ['o', 'x', 'x'], ['-', '-', '-']]
```

Nota: **x** es un jugador, **o** el otro jugador y **-** representa espacios en blanco. Programe una función que permita determinar si hay un ganador en el gato.

La función debe llamarse de la siguiente forma:

`verificar_ganador_gato(tablero, jugadorx)` → `True/False`

donde:

1. **tablero** es una matriz que contiene únicamente textos (sino '*Error01*')
2. **tablero** es una matriz cuadrada (sino '*Error02*')
3. **jugadorx** debe ser un texto de un solo caracter (sino '*Error03*')
4. **jugadorx** debe ser **'**x**'** o **'**o**'** únicamente (sino '*Error04*')
5. La **salida** debe ser un valor de verdad *True* si el jugadorx indicado es ganadorx, *False* en caso contrario.

Figura 1.2: Pregunta 2

Codigo de Notch Engine

```
WorldName VerificarGanadorGato:

Recipe
Spell verificarGato(Shelf Rune :: tablero; Rune :: jugador) -> Torch;

CraftingTable
Spell verificarGato(Shelf Rune :: tablero; Rune :: jugador) -> Torch
PolloCrudo
    $$ Validaciones básicas
    target #(tablero) isNot 9 craft hit
    PolloCrudo
        dropperSpider("Error02"); $$ No es cuadrado (3x3)
        respawn Off;
    PolloAsado

    target jugador isNot 'x' and jugador isNot 'o' craft hit
    PolloCrudo
```

```
        dropperSpider("Error04"); $$ Jugador no válido
        respawn Off;
    PolloAsado

    $$ Líneas ganadoras (por índice lineal)
    Shelf Shelf Stack lineas = [
        [0,1,2], [3,4,5], [6,7,8],  $$ Filas
        [0,3,6], [1,4,7], [2,5,8],  $$ Columnas
        [0,4,8], [2,4,6]            $$ Diagonales
    ];

    walk i set 0 to 7 craft
    PolloCrudo
        Shelf Stack linea = lineas[i];
        Stack a = linea[0];
        Stack b = linea[1];
        Stack c = linea[2];

        target tablero[a] is jugador and tablero[b] is jugador and tablero[c] is
        PolloCrudo
            respawn On;
        PolloAsado
    PolloAsado

    $$ No se encontró ganador
    respawn Off;
PolloAsado

SpawnPoint
Shelf Rune gato = ['x','o','x','o','x','x','-','-','-'];
Rune j = 'x';

Torch hayGanador = verificarGato(gato, j);

dropperTorch(hayGanador);

worldSave
```

1.6. Implementación del Generador de Código

1.6.1. Introducción a la Implementación

La implementación del generador de código del compilador Notch Engine se materializa principalmente en el archivo `mc_generacion.py`, que contiene una jerarquía de clases especializadas para diferentes aspectos de la generación de código. Esta implementación modular permite una separación clara de responsabilidades y facilita la extensión y mantenimiento del sistema.

La arquitectura implementada sigue un diseño orientado a objetos que progresa desde funcionalidades básicas hasta características avanzadas, culminando en un generador completo capaz de producir código ensamblador TASM totalmente funcional.

1.6.2. Arquitectura de Clases

Clase Base: GeneradorCodigo

La clase `GeneradorCodigo` proporciona la funcionalidad fundamental para la generación de código:

Gestión de Portadas: Genera automáticamente portadas de identificación que incluyen información del proyecto, autores (Cabrera Samir, Urbina Luis), fecha de generación, y metadatos del compilador.

Control de Segmentos: Implementa un sistema robusto de gestión de segmentos de memoria con límites configurables (4KB para datos, 4KB para código) y verificación automática de desbordamiento.

Tabla de Símbolos de Generación: Mantiene una tabla independiente específica para la generación que almacena información como direcciones de memoria, tipos de datos, y metadata específica de cada símbolo.

Plantilla Base: Genera la estructura fundamental del programa ASM incluyendo directivas `ASSUME`, definición de segmentos, y procedimiento principal.

Clase Extendida: GeneradorConRuntime

La clase `GeneradorConRuntime` extiende las capacidades básicas incorporando la Runtime Library:

Integración de Runtime Library: Proporciona métodos para incluir automáticamente la Runtime Library en el código generado, ya sea mediante inclusión directa o referencia externa.

Generación de Operaciones: Implementa la traducción de operaciones del lenguaje Notch Engine a llamadas apropiadas de la Runtime Library, manejando la preparación de parámetros y recuperación de resultados.

Creación Automática de Runtime: Incluye funcionalidad para generar automáticamente el archivo completo de Runtime Library con todas las rutinas necesarias.

Clase Completa: GeneradorCompleto

La clase `GeneradorCompleto` proporciona funcionalidad avanzada para estructuras de control complejas:

Control de Flujo: Implementa generación de código para estructuras condicionales y repetitivas mediante etiquetas y saltos.

Gestión de Contextos: Mantiene una pila de contextos para manejar estructuras anidadas y garantizar la correcta generación de etiquetas únicas.

Optimizaciones: Incorpora optimizaciones básicas como eliminación de saltos redundantes y reutilización de registros.

1.6.3. Proceso de Generación de Runtime Library

Estructura de la Runtime Library Generada

El método `crear_archivo_runtime` genera un archivo ASM completo que contiene:

Rutinas Aritméticas:

```
- SUMAR_ENTEROS: Implementa operación :+
- RESTAR_ENTEROS: Implementa operación :-
- MULTIPLICAR_ENTEROS: Implementa operación :*
- DIVIDIR_ENTEROS: Implementa operación :/ con manejo de división por cero
- MODULO_ENTEROS: Implementa operación :% con protección de errores
```

Rutinas de Incremento/Decremento:

```
- INCREMENTAR_ENTERO: Implementa soulsand
- DECREMENTAR_ENTERO: Implementa netherrack
```

Rutinas de Comparación:

```
- COMPARAR_IGUAL: Implementa is
- COMPARAR_DIFERENTE: Implementa isNot
- COMPARAR_MAYOR: Implementa >
- COMPARAR_MENOR: Implementa <
- COMPARAR_MAYOR_IGUAL: Implementa >=
- COMPARAR_MENOR_IGUAL: Implementa <=
```

Rutinas Lógicas:

```
- AND_LOGICO: Implementa operaciones AND
- OR_LOGICO: Implementa operaciones OR
- NOT_LOGICO: Implementa operaciones NOT
- XOR_LOGICO: Implementa operaciones XOR
```

Rutinas de Entrada/Salida:

```
- LEER_ENTERO: Lectura de valores STACK
- MOSTRAR_ENTERO: Salida de valores STACK
- LEER_CARACTER: Lectura de caracteres individuales
- MOSTRAR_CARACTER: Salida de caracteres
- LEER_BOOLEANO: Lectura de valores TORCH
- MOSTRAR_BOOLEANO: Salida de valores TORCH como ON/OFF
```

Rutinas de Cadenas:

```
- CONCATENAR_STRINGS: Unión de cadenas SPIDER
- LONGITUD_STRING: Cálculo de longitud de cadenas
```

Características de Implementación

Cada rutina de la Runtime Library implementa:

- **Convención de Llamadas Estándar**: Uso de pila para parámetros, retorno en AX, preservación de registros
- **Manejo de Errores**: Detección y manejo seguro de condiciones excepcionales
- **Optimización**: Código eficiente con mínimo uso de recursos
- **Documentación**: Comentarios descriptivos y metadatos de generación

1.6.4. Generación de Código Principal

Proceso de Traducción de Operaciones

El método `generar_operacion_aritmetica` implementa la traducción directa de operaciones Notch Engine:

```
def generar_operacion_aritmetica(self, operador, var_resultado, operando1, opera
    operaciones_runtime = {
        ':+': 'SUMAR_ENTEROS',
        ':-': 'RESTAR_ENTEROS',
        ':*': 'MULTIPLICAR_ENTEROS',
        ':/': 'DIVIDIR_ENTEROS',
        ':%': 'MODULO_ENTEROS'
    }

    codigo_operacion = [
        f"    ; Operación: {var_resultado} = {operando1} {operador} {operando2}"
        f"    PUSH {operando2}    ; Segundo operando",
        f"    PUSH {operando1}    ; Primer operando",
        f"    CALL {rutina}       ; Llamar rutina de Runtime Library",
        f"    MOV {var_resultado}, AX  ; Guardar resultado"
    ]
```

Este proceso garantiza que cada operación del lenguaje fuente se traduzca correctamente a una secuencia equivalente de instrucciones ASM que utiliza la Runtime Library.

Declaración de Variables

El método `declarar_variable` maneja la asignación de espacio en el segmento de datos:

- Verifica disponibilidad de espacio en el segmento de datos
- Asigna direcciones de memoria apropiadas según el tipo de dato
- Actualiza la tabla de símbolos de generación
- Genera las directivas ASM correspondientes (DW, DB, etc.)

Control de Flujo

Para estructuras de control, se implementan métodos especializados:

Generación de Etiquetas: Sistema automático de generación de etiquetas únicas para evitar conflictos en programas complejos.

Estructuras Condicionales: Traducción de `TARGET/HITMISS` a secuencias de comparación y salto condicional.

Bucles: Implementación de estructuras repetitivas mediante etiquetas de inicio y salto condicional de salida.

1.6.5. Archivos de Prueba y Validación

Programa de Prueba (programa_test.asm)

El archivo `programa_test.asm` representa un ejemplo del código generado automáticamente:

```
;=============================================
;               NOTCH ENGINE COMPILER
;=============================================
; Generado por: Cabrera Samir, Urbina Luis
; Fecha: 12/06/2025 21:47
; Proyecto: Compilador para Notch Engine
; Etapa 4: Generador de Código
;=============================================

ASSUME CS:CODE, DS:DATA, SS:STACK

STACK SEGMENT STACK
    DW 256 DUP(?)
STACK ENDS

DATA SEGMENT
    numero DW 10
    ; Operación: resultado = numero :+ 5
    PUSH 5    ; Segundo operando
    PUSH numero    ; Primer operando
    CALL SUMAR_ENTEROS       ; Llamar rutina de Runtime Library
    MOV resultado, AX  ; Guardar resultado
DATA ENDS

CODE SEGMENT
MAIN PROC
    MOV AX, DATA
    MOV DS, AX
    ; Código principal generado
    ; Terminar programa
    MOV AH, 4Ch
    INT 21h
MAIN ENDP
CODE ENDS
END MAIN
```

Este archivo demuestra la estructura completa generada automáticamente, incluyendo portada, segmentos, variables, operaciones y terminación apropiada.

Programa de Prueba de Runtime (test_runtime_library.asm)

El archivo `test_runtime_library.asm` contiene pruebas exhaustivas de la Runtime Library:

Pruebas Aritméticas: Verificación de suma, resta, multiplicación, división y módulo con valores conocidos.

Pruebas Lógicas: Validación de operaciones AND, OR, NOT con diferentes combinaciones de valores booleanos.

Pruebas de E/S: Verificación de lectura y escritura para diferentes tipos de datos.

Pruebas de Manejo de Errores: Validación del comportamiento en condiciones excepcionales como división por cero.

Script de Pruebas (test_generador.py)

El archivo `test_generador.py` implementa pruebas automatizadas:

```
def test_mc_generacion_directo():
    from mc_generacion import GeneradorConRuntime

    gen = GeneradorConRuntime()

    # Crear Runtime Library
    resultado = gen.crear_archivo_runtime("runtime_generado.asm")

    # Generar código de prueba
    gen.declarar_variable("numero", "STACK", "10")
    gen.generar_operacion_aritmetica(":+", "resultado", "numero", "5")
    gen.finalizar_programa()
    gen.guardar_archivo("programa_test.asm")
```

Este script verifica automáticamente:

- Creación correcta de la Runtime Library
- Generación apropiada de código principal
- Funcionalidad de declaración de variables y operaciones
- Proceso completo de finalización y guardado

1.6.6. Control de Flujo Avanzado

Archivo de Prueba de Control de Flujo (test_control_flujo_corregido.asm)

Este archivo demuestra la implementación de estructuras de control complejas:

Estructuras Condicionales Anidadas: Implementación de múltiples niveles de `TARGET/HITMISS` con etiquetas apropiadas.

Bucles con Condiciones Complejas: Demostración de bucles con múltiples condiciones de salida y variables de control.

Combinación de Estructuras: Integración de diferentes tipos de estructuras de control en un programa cohesivo.

Generación de Etiquetas Únicas

El sistema implementa un algoritmo robusto para generar etiquetas únicas:

- Uso de contadores incrementales para cada tipo de estructura
- Prefijos descriptivos que indican el tipo de estructura (`IF_`, `WHILE_`, `END_`)
- Verificación de unicidad para evitar conflictos en programas complejos
- Mantenimiento de contexto para estructuras anidadas

1.6.7. Integración y Compatibilidad

Runtime Generado (runtime_generado.asm)

El archivo `runtime_generado.asm` representa la salida del proceso automático de generación de Runtime Library:

- Contiene todas las rutinas necesarias para la ejecución de programas Notch Engine
- Incluye metadatos de generación y documentación automática
- Implementa manejo robusto de errores y condiciones excepcionales
- Optimizado para eficiencia en tiempo de ejecución

Runtime Library Base (runtime_library.asm)

El archivo `runtime_library.asm` sirve como plantilla y referencia:

- Define la estructura estándar para todas las rutinas
- Establece convenciones de codificación y documentación
- Proporciona implementaciones de referencia para operaciones críticas
- Facilita la extensión y mantenimiento de la biblioteca

1.6.8. Manejo de Errores y Robustez

Errores de Generación

El sistema implementa detección y manejo comprehensivo de errores:
Desbordamiento de Segmentos: Verificación automática antes de cada adición de código o datos.
Operadores No Soportados: Validación de operadores antes de generar código correspondiente.
Variables No Declaradas: Verificación de existencia en tabla de símbolos antes de uso.
Tipos Incompatibles: Validación de compatibilidad de tipos antes de generar operaciones.

Estrategia de Recuperación

Cuando se detectan errores durante la generación:

1. Se registra el error con información detallada de contexto
2. Se intenta recuperación automática cuando es posible
3. Se genera código alternativo seguro si la recuperación no es factible
4. Se termina la generación con mensaje descriptivo en casos irrecuperables

1.6.9. Optimizaciones y Eficiencia

Optimizaciones Implementadas

El generador incorpora varias optimizaciones:

Reutilización de Registros: Mantiene valores en registros cuando es beneficioso.

Eliminación de Operaciones Redundantes: Detecta y elimina cálculos innecesarios.

Optimización de Saltos: Minimiza la cantidad de saltos condicionales e incondicionales.

Gestión Eficiente de Pila: Optimiza el uso de la pila para parámetros y variables temporales.

Métricas de Rendimiento

El sistema proporciona estadísticas detalladas:

- Número total de variables declaradas
- Uso actual y disponible de segmentos de datos y código
- Cantidad de operaciones generadas por tipo
- Número de llamadas a Runtime Library
- Eficiencia de uso de memoria

1.6.10. Extensibilidad y Mantenimiento

Arquitectura Modular

La implementación facilita extensiones futuras:

- Separación clara entre generación básica y características avanzadas
- Interfaces bien definidas entre componentes
- Posibilidad de agregar nuevos tipos de datos sin modificar código existente
- Soporte para nuevas operaciones mediante extensión de la Runtime Library

Documentación y Trazabilidad

El código generado incluye:

- Comentarios automáticos que describen cada operación
- Metadatos de generación con fecha, versión y autores
- Trazabilidad entre código fuente y código generado
- Información de depuración para facilitar el mantenimiento

1.6.11. Casos de Uso y Ejemplos Prácticos

Generación de Operaciones Complejas

El sistema maneja casos complejos como:

Expresiones Anidadas: Evaluación de expresiones con múltiples operadores mediante uso apropiado de la pila y llamadas secuenciales a la Runtime Library.

Asignaciones Múltiples: Generación eficiente de código para asignaciones que involucran múltiples variables y operaciones.

Operaciones con Constantes: Optimización especial para operaciones que involucran valores constantes conocidos en tiempo de compilación.

Gestión de Tipos de Datos

El generador maneja apropiadamente todos los tipos del lenguaje Notch Engine:

Tipos Simples: `STACK` (enteros), `GHAST` (flotantes), `TORCH` (booleanos), `SPIDER` (cadenas).

Tipos Compuestos: `SHELF` (arreglos) con cálculo automático de direcciones y verificación de límites.

Constantes: `OBSIDIAN` con verificación de inmutabilidad durante la generación.

Archivos: `BOOK` con manejo de recursos y verificación de estado.

1.6.12. Integración con el Sistema Completo

Interfaz con Etapas Previas

El generador se integra seamlessly con:

Analizador Léxico: Utiliza información de tokens para generar comentarios descriptivos y mantener trazabilidad.

Analizador Sintáctico: Sigue la estructura sintáctica validada para garantizar correctitud en la generación.

Analizador Semántico: Utiliza información de tipos y alcances de la tabla de símbolos semánticos para generar código apropiado.

Salida Final

El resultado del proceso de generación incluye:

- Archivo principal ASM con el programa compilado
- Archivo de Runtime Library con todas las rutinas necesarias
- Archivos de prueba para validar la funcionalidad
- Documentación automática y metadatos de generación
- Estadísticas de compilación y uso de recursos

La implementación del generador de código representa así una solución robusta, eficiente y extensible que completa exitosamente el proceso de compilación del lenguaje Notch Engine, produciendo código ensamblador TASM de alta calidad que preserva fielmente la semántica del programa fuente mientras proporciona las herramientas necesarias para la ejecución, depuración y mantenimiento del software generado.