IceBox Compiler

- Darío Fragas González C411
- Abraham González Rivero C412

1. Detalles técnicos

A continuación se describe como compilar un código dado en COOL a MIPS. Ejecute:

```
cool.py [-h] [--output_file OUTPUT_FILE] [--lexer] [--parser] [-t] [-c]
              [--log-level LOG_LEVEL]
              input_file

IceBox Compiler

argumentos posicionales:
  input_file : el archivo que donde se encuentra el código fuente.

argumentos opcionales:
  -h, --help : muestra este mensaje y termina la ejecución.
  --output_file: el archivo de salida( default: el mismo que se dio como entrada pero con .mips como extensión).
  --lexer: ejecuta solo etapa de tokenización del lexer.
  --parser: ejecuta hasta la etapa de parsing.
  -t: printea los tokens que dio el lexer.
  -c, --cil: da como salida un programa de CIL en output.cil
  --log-level LOG_LEVEL : setea el modo del log(default:INFO).
```

Estructura y organización de los directorios del proyecto a partir de la carpeta *src*:

```
.
├── codegen
│      ├── __init__.py
│      ├── cil_ast.py
│      ├── cil_to_mips.py
│      ├── cocl_to_cil.py
│      ├── mips_ast.py
│      └──  mips_codegen.py
├── parsing
│      ├── __init__.py
│      ├── ast.py
│      ├── lex.py
│      └── parser.py
├── semantic
│      ├── __init__.py
│      ├── context.py
│      ├── scope.py
│      ├── type_builder.py
│      ├── type_checker.py
│      ├── type_collector.py
│      └── types.py
├─ test_cool
│      ├── dispatch.cl
│      └── formals.cl
├─ utils
│      ├── loggers.py
│      └── visitor.py
├── cool.py
├── coolc.sh
├── makefile
├── Readme.md
```

2. Arquitectura del Compilador:

IceBox Compiler, diseñado para traducir código escrito en COOL (Classroom Object-Oriented Language) a MIPS (Microprocessor without Interlocked Pipeline Stages), se estructura en tres módulos fundamentales: parsing, semantic y codegen. Cada uno de estos módulos encapsula una fase crítica en el proceso de compilación, abordando desafíos únicos inherentes a la traducción y optimización del código fuente.

Análisis Lexicográfico

El análisis léxico, implementado en `lex.py`, juega un papel crucial en el proceso de compilación del lenguaje COOL. Esta etapa transforma una secuencia lineal de caracteres en una secuencia de tokens, que son las unidades fundamentales de significado en el lenguaje. Los tokens pueden ser literales, palabras clave, identificadores, números, cadenas, entre otros.

La Clase `CharacterStream`

Esta clase es esencial en el proceso de tokenización, ya que permite una manipulación detallada y eficiente de la cadena de caracteres del código fuente. Con métodos como `next_char()` y `peek_char()`, ofrece la capacidad de avanzar en la cadena y observar el siguiente carácter sin consumirlo, respectivamente. El método `get_position()` devuelve la posición actual en la cadena (línea y columna), lo cual es crucial para la generación de mensajes de error significativos y precisos. La función `reset()` reinicia el estado interno del objeto para un nuevo análisis, lo que es especialmente útil en múltiples pasadas o reanálisis del código fuente.

Clasificación de Tokens

La enumeración `TokenType` define los tipos de tokens que el lexer puede identificar. Esto incluye identificadores de objetos, constantes enteras, constantes de cadena, diferentes símbolos y operadores (punto, coma, dos puntos, etc.), así como tokens especiales como comentarios de una línea y de múltiples líneas, y el fin de archivo (EOF).

La clase `Token` representa un token individual con tres propiedades críticas: tipo, valor y posición. Esto permite no solo identificar la naturaleza del token, sino también su valor específico y su ubicación en el código fuente, elementos esenciales para el análisis y la depuración.

Implementación del Lexer

El `Lexer` es el motor principal de esta fase. Se inicializa con una instancia de `CharacterStream`, y se encarga de procesar esta secuencia de caracteres para producir tokens. El método `fetch_token()` es responsable de la lógica de tokenización: identifica el tipo de token basado en el carácter actual y, en algunos casos, en el siguiente carácter (lookahead).

El lexer maneja diferentes categorías de tokens:

- Identificadores y palabras clave: detecta secuencias de caracteres alfabéticos, diferenciando las palabras clave de los identificadores.
- Números: captura secuencias numéricas.
- Cadenas: maneja cadenas entre comillas, incluyendo el escape de caracteres especiales.
- Comentarios: identifica y omite comentarios de una línea y de múltiples líneas.
- Operadores y puntuación: reconoce símbolos y operadores específicos del lenguaje COOL.

Durante el proceso de tokenización, el lexer también lleva un registro de errores, como cadenas sin cerrar, caracteres nulos en cadenas o EOF inesperado en comentarios. Esto es crucial para proporcionar retroalimentación detallada y útil durante la depuración y la corrección de errores en el código fuente.

Finalmente, el método `lex()` coordina el proceso de tokenización, llamando repetidamente a `fetch_token()` hasta que se alcanza el final de la cadena de entrada, y devolviendo una lista de tokens junto con cualquier error encontrado.

En resumen, el módulo de análisis léxico de IceBox es un sistema refinado y robusto, diseñado para descomponer eficientemente el código fuente en COOL en tokens significativos, preparándolos para las etapas subsiguientes de análisis sintáctico y semántico. La implementación refleja una consideración cuidadosa de los aspectos prácticos y teóricos del análisis léxico, asegurando precisión, eficiencia y una sólida base para el proceso de compilación.

Análisis Sintáctico:

El análisis sintáctico, llevado a cabo en `parser.py`, es un proceso crítico que transforma la secuencia de tokens generada por el análisis léxico en un Árbol de Sintaxis Abstracta (AST). Este proceso es fundamental en la compilación ya que establece la estructura jerárquica del programa, permitiendo su posterior análisis y optimización.

Producciones

```
 program := [[class]]+
class ::= class TYPE [inherits TYPE] { [[feature; ]]*}
feature := ID method
                 | ID attribute

                 | ID : TYPE [ <- expr]
method := ( [ formal [[, formal]]*] ) : TYPE { expr }
formals := formal [[, formal]]*
expr ::=ID <- expr
            | expr [@TYPE].ID( [ expr [[ , expr ]]*] )
            | ID( [ expr  [[ , expr ]] *] )
            | if expr then expr else expr if
            | while expr loop expr pool
            | { [[ expr ; ]]+}
            | let ID : TYPE [<-expr] [[,ID : TYPE [<- expr ]]]*in expr
            | case expr of [[ID : TYPE => expr; ]]+esac
            |new TYPE
            |isvoid expr
            |expr+expr
            |expr– expr
            |expr*expr
            |expr/expr
            |ẽxpr
            |expr<expr
            |expr<=expr
            |expr=expr
            |not expr
            |(expr)
            |ID
            |integer
            |string
            |true
            |false
```

Creación y Estructura del AST

El AST, definido en `ast.py`, es una representación jerárquica del código fuente, donde cada nodo representa una construcción del lenguaje (como clases, métodos, expresiones). Estos nodos son diseñados para ser visitables (patrón Visitor), permitiendo operaciones posteriores sobre ellos. Por ejemplo, `ProgramNode` representa la raíz del AST y contiene una lista de `ClassNode`, cada uno representando una clase en COOL.

Cada nodo del AST almacena información crucial como su tipo, valor y posición en el código fuente. Esta información es vital para el análisis semántico y la generación de código.

Proceso de Parsing

El proceso de parsing inicia con la creación de una instancia de `Parser`, que toma como entrada la lista de tokens generada en la fase léxica. El parser procesa los tokens siguiendo las reglas gramaticales del lenguaje COOL y construye el AST correspondiente.

- El método `parse()` coordina el proceso de parsing, invocando diferentes métodos para manejar las distintas construcciones del lenguaje.
- `program()` y `parse_class()` manejan la creación de nodos para el programa completo y para cada clase, respectivamente.
- `parse_feature()` se encarga de analizar las características de las clases, como atributos y métodos.
- `parse_method()` y `parse_attribute()` son métodos específicos para analizar métodos y atributos dentro de las clases.
- `parse_formals()` y `parse_formal()` manejan los argumentos formales de los métodos.

El parser utiliza el método `eat()` para consumir tokens de la secuencia y avanzar en el análisis, asegurándose de que los tokens consumidos sigan la gramática del lenguaje.

Manejo de Expresiones

La complejidad del análisis sintáctico aumenta considerablemente al manejar expresiones. El método `parse_expression()` es un componente esencial en este proceso, encargándose de analizar expresiones aritméticas, lógicas, condicionales, bucles y asignaciones. Este método, junto con otros como `parse_if()`, `parse_while()`, `parse_let()`, `parse_case()`, entre otros, gestiona la complejidad inherente a las expresiones en COOL, construyendo la estructura jerárquica correspondiente en el AST.

Gestión de Errores

El parser también está equipado para manejar errores sintácticos. Mediante el registro y notificación de errores, proporciona retroalimentación útil para la depuración y corrección del código fuente. Esto es crucial para guiar a los desarrolladores en la resolución de problemas en sus programas.

En resumen, el módulo de análisis sintáctico de IceBox es un componente que transforma una secuencia lineal de tokens en una estructura de árbol que refleja la

jerarquía y la estructura del código fuente COOL. Este proceso no solo implica seguir las reglas gramaticales del lenguaje, sino también manejar la complejidad de las distintas construcciones sintácticas y proporcionar una base sólida para el análisis semántico y la generación de código.

Análisis Semántico

El análisis semántico es una fase crítica en la compilación del lenguaje COOL, donde se verifica la correcta aplicación de las reglas semánticas del lenguaje. Esta fase asegura que el programa no solo esté sintácticamente correcto, sino que también sea lógicamente coherente.

Estructura y Procedimiento

1. **Preparación**: Antes de comenzar el análisis, se prepara un contexto (`Context`) que actúa como un diccionario de todos los tipos (clases) definidos en el programa. Este contexto se enriquece con tipos incorporados como `Object` , `Int` , `String` , `Bool` , `IO` .
2. **Recorridos por el AST**: Se realizan tres recorridos principales sobre el AST, obtenido tras el análisis sintáctico:
    - **Recolección de Tipos**: Utilizando `TypeCollector` , se recolectan todos los tipos definidos en el programa, incluidas las clases definidas por el usuario. Cada tipo se almacena en el contexto.
    - **Construcción de Tipos**: El `TypeBuilder` procesa cada tipo recolectado, definiendo sus atributos y métodos. Aquí se realizan comprobaciones como la no redefinición de atributos heredados y la coherencia en la definición y sobreescritura de métodos.
    - **Chequeo de Tipos**: Por último, `TypeChecker` realiza la verificación de tipos en todas las expresiones y estructuras del programa. Se asegura que las operaciones, asignaciones y llamadas a métodos sean coherentes con los tipos de los elementos involucrados. Además computa y almacena el tipo estático de cada expresión.

Detalle del Análisis Semántico: TypeCollector, TypeBuilder y TypeChecker

1. TypeCollector

El `TypeCollector` es responsable de la primera fase de análisis semántico, donde se recolectan y registran todos los tipos (clases) definidos en el programa COOL.

- **Funcionamiento**: Visita cada `ClassNode` en el AST y agrega su correspondiente `Type` al `Context` .
- **Manejo de Errores**:
    - Redefinición de Tipos: Reporta errores si una clase se define más de una vez.
    - Tipos Reservados: Reporta errores si se definen clases con nombres reservados del lenguaje (como `Object` , `Int`).
    - Clase Principal Faltante: Verifica y reporta si falta la clase principal `Main` , que es obligatoria en COOL.

2. TypeBuilder

El `TypeBuilder` construye la estructura interna de cada tipo identificado por el `TypeCollector` , definiendo atributos y métodos, y asegurando la correcta aplicación de las reglas de herencia.

- **Funcionamiento**: Recorre el AST hasta los `ClassNode` , por cada `Type` recolectado, definine sus atributos y métodos. Gestiona las relaciones de herencia y asegura la coherencia en la definición de tipos.
- **Manejo de Errores**:
    - Herencia de Tipos No Definidos: Reporta errores si una clase hereda de una clase no definida o inexistente.
    - Herencia de Tipos Reservados: Reporta errores si una clase intenta heredar de tipos reservados como `Int` , `Bool` , `String` o `SELF_TYPE` .
    - Redefinición de Atributos: Reporta errores si se intenta redefinir un atributo heredado.
    - Conflicto en Redefinición de Métodos: Reporta errores si un método redefinido no coincide en el tipo o número de argumentos con su definición original en la clase padre.

3. TypeChecker

El `TypeChecker` es la última fase del análisis semántico, donde se verifica que cada expresión, asignación, llamada a método y estructura de control en el AST utilice los tipos correctamente según las reglas del lenguaje COOL.

- **Funcionamiento**: Visita cada nodo del AST y valida el uso correcto de tipos. Esto incluye la conformidad de los tipos en operaciones binarias, la verificación de tipos en asignaciones y llamadas a métodos, y el uso correcto de tipos en estructuras de control.
- **Manejo de Errores**:
    - Uso Incorrecto de Tipos en Expresiones: Reporta errores si se realizan operaciones entre tipos incompatibles (por ejemplo, sumar un `Int` y un `String`).
    - Asignaciones de Tipo Incorrecto: Reporta errores si el tipo del valor asignado no se conforma con el tipo de la variable.
    - Llamadas a Métodos Incorrectas: Reporta errores si se llama a un método que no existe en el tipo dado o si los tipos de los argumentos no son compatibles.
    - Uso Incorrecto en Estructuras de Control: Reporta errores si las expresiones en estructuras de control como `if` o `while` no son de tipo `Bool` .
    - Uso Incorrecto de `self` : Reporta errores si se utiliza incorrectamente la palabra reservada `self` en el programa.

Estas tres fases del análisis semántico - TypeCollector, TypeBuilder y TypeChecker - forman una secuencia lógica y rigurosa para asegurar la correcta aplicación de las reglas semánticas del lenguaje COOL. Cada fase tiene su responsabilidad específica y un conjunto de errores que maneja, garantizando así que el programa resultante no solo sea sintácticamente correcto sino también semánticamente coherente, listo para la generación de código.

Generación de código

En el proceso de compilación, uno de los pasos más críticos y complejos es la generación de código. Es aquí donde el lenguaje de alto nivel, como COOL, se transforma en un conjunto de instrucciones que una máquina puede entender y ejecutar. Sin embargo, saltar directamente a un lenguaje de máquina específico o código objeto puede ser extraordinariamente desafiante y poco práctico. Aquí es donde entra en juego el concepto de un Lenguaje Intermedio (IR, por sus siglas en inglés).

El IR actúa como un puente entre el alto nivel de abstracción del lenguaje fuente y el bajo nivel del código de máquina. Este nivel intermedio simplifica el proceso de traducción al descomponerlo en dos etapas más manejables: primero, de COOL a IR, y luego, del IR al código de máquina. Este enfoque tiene múltiples ventajas:

1. **Simplificación de la Compilación**: El IR proporciona una abstracción que es más fácil de manipular que el código de máquina directo, permitiendo resolver

problemas complejos paso a paso.

2. **Independencia del Hardware**: Al utilizar un IR, el compilador no está atado a un conjunto específico de instrucciones de hardware, lo que facilita la portabilidad del código fuente a diferentes arquitecturas de máquinas.

3. **Optimización**: El IR permite realizar optimizaciones de código intermedias antes de la generación final del código de máquina, mejorando así el rendimiento y la eficiencia del programa compilado.

En el caso de IceBox, se utiliza el Class Intermediate Language (CIL), definido en clases, como IR. CIL es un lenguaje de máquina simplificado con soporte para operaciones orientadas a objetos. No tiene concepto de herencia y trata todas las variables como enteros de 32 bits, lo que requiere una interpretación basada en el contexto de uso.

La generación de código en CIL implica convertir las estructuras complejas y las expresiones del lenguaje COOL en un conjunto de instrucciones de tres direcciones en CIL. Este paso intermedio es crucial para asegurar que el código generado sea eficiente, optimizado y, sobre todo, funcional en la arquitectura objetivo.

Traducción de COOL a CIL en IceBox

El proceso de traducir COOL a CIL implica varios pasos detallados y funciones específicas. Aquí se detalla cada parte de este proceso:

Configuración Inicial y Funciones Auxiliares

- **Inicialización**: Se crean listas para almacenar tipos (`self.types`), código (`self.code`) y datos (`self.data`). Además, se establecen contadores para gestionar identificadores únicos y se inicializa el contexto semántico.
- **Funciones Auxiliares**: Se definen métodos para generar identificadores únicos (`generate_next_string_id` y `next_id`), convertir nombres de funciones (`to_function_name`), y manejar nombres de datos y atributos (`to_data_name` y `to_attr_name`).

Registro de Tipos y Funciones

- **Registro de Tipos (`register_type`)**: Crea un nuevo nodo de tipo en CIL y lo añade a la lista de tipos.
- **Registro de Funciones (`register_function`)**: Crea un nuevo nodo de función en CIL y lo añade a la lista de código.

Creación de Estructuras CIL

- **ProgramNode (`visit__ProgramNode`)**: Se recorren todos los tipos en el contexto y se preparan sus atributos y métodos para su mapeo en CIL. Se genera la función principal (`main`) en CIL y se agregan funciones integradas (builtin).
- **ClassNode (`visit__ClassNode`)**: Por cada clase en COOL, se registra un tipo en CIL, se agregan sus atributos y métodos, y se genera su función de inicialización.
- **AttributeNode (`visit__AttributeNode`)**: Se crea una función de inicialización para cada atributo, asignando valores predeterminados o expresiones de inicialización.
- **MethodNode (`visit__MethodNode`)**: Se convierte cada método COOL en una función CIL, procesando sus parámetros y cuerpo.

Traducción de Expresiones

- **Asignaciones (`visit__AssignNode`)**: Se traduce la asignación de COOL a CIL, manejando la asignación a variables locales, parámetros o atributos.
- **Llamadas a Métodos (`visit__MethodCallNode` y `visit__DispatchNode`)**: Se manejan llamadas a métodos, diferenciando entre llamadas estáticas y dinámicas.
- **Control de Flujo (`visit__IfNode` y `visit__WhileNode`)**: Se traducen estructuras de control como `if` y `while` a sus equivalentes en CIL, utilizando etiquetas y saltos condicionales.
- **Let (`visit__LetNode`)**: Se maneja la declaración de variables locales con posibles inicializaciones.
- **Operadores Binarios (`visit__BinaryOperatorNode`)**: Se convierten operaciones binarias (como suma, resta, comparaciones) a instrucciones CIL.
- **Booleanos, Enteros y Cadenas (`visit__BooleanNode` , `visit__IntegerNode` , `visit__StringNode`)**: Se cargan valores literales directamente en CIL. Las cadenas se añaden a la sección DATA.
- **Negación y Complemento (`visit__NotNode` , `visit__PrimeNode`)**: Se manejan operaciones unarias como la negación y el complemento.
- **Case (`visit__CaseNode`)**: Se implementa la lógica para la expresión `case` , manejando la selección dinámica de ramas en tiempo de ejecución.

Funciones integradas y manejo de errores

- **Funciones Integradas (Builtin)**: Se agregan funciones integradas de COOL como `abort` , `type_name` , `copy` , y operaciones de entrada/salida.
- **Manejo de Errores**: Se incluyen estrategias para el manejo de errores en tiempo de ejecución, como en la selección de ramas en `case` .

Este proceso detallado asegura una traducción precisa y eficiente del código COOL a su representación intermedia en CIL, sentando las bases para la posterior generación de código MIPS.

De CIL a MIPS

Esta fase consiste en traducir el Abstract Syntax Tree (AST) de CIL a un AST de MIPS, un lenguaje de bajo nivel más cercano al hardware. El objetivo es generar un código que pueda ser ejecutado por un emulador de SPIM, una versión del procesador MIPS.

Estructura y Representación en Memoria

En MIPS, las secciones `.data` y `.text` contienen, respectivamente, los datos constantes (como cadenas de texto) y las instrucciones del programa. La representación en memoria es crucial, especialmente para instancias de tipos y la tabla de métodos virtuales. Esta última facilita las llamadas a métodos no estáticos, con cada tipo almacenando una etiqueta (`label`) y un conjunto de métodos asociados.

En MIPS, la representación en memoria de los objetos y tipos es fundamental. Cada objeto se almacena en bloques de memoria, con el primer bloque reservado para la información del tipo (metadata). Los atributos y métodos se almacenan en bloques consecutivos. Esta organización permite un acceso eficiente a los atributos y la ejecución de métodos.

Tipo de Atributo	Atributo 1	Atributo 2	...	Atributo n
Dirección	Dirección + 4	Dirección + 8		Dirección + 4*n

Tratamiento de Tipos `Int`, `String` y `Bool` por valor

En la compilación de CIL a MIPS, los tipos `Int`, `String` y `Bool` presentan un desafío único debido a sus características inherentes. La decisión de tratar estos tipos por valor en lugar de por referencia tiene implicaciones significativas en cómo se manejan en el proceso de compilación. Aquí detallamos cómo se aborda cada uno de estos tipos.

1. Int y Bool

`Int` y `Bool` son tipos básicos que se representan típicamente como valores enteros en muchos lenguajes de bajo nivel, incluido MIPS. En el contexto de la compilación de CIL a MIPS, estos tipos se manejan de la siguiente manera:

- **Representación**: Tanto `Int` como `Bool` se representan mediante valores enteros de 32 bits. En MIPS, esto significa que se pueden almacenar directamente en registros o en la pila.
- **Operaciones**: Las operaciones aritméticas o lógicas en `Int` y `Bool` se realizan directamente utilizando instrucciones MIPS estándar para enteros. Por ejemplo, la adición de dos enteros o la evaluación de una expresión booleana se maneja mediante las instrucciones de suma o comparación de MIPS. Se hacen anotaciones en CIL que permite los llamados a métodos de estos tipos así como el `type of` computarse sin necesidad de acceder al tipo.
- **Eficiencia**: Al tratar estos tipos por valor, se evita la sobrecarga asociada con el manejo de referencias a objetos, lo que resulta en un rendimiento mejorado, especialmente en operaciones que son muy frecuentes como cálculos numéricos o evaluaciones lógicas.

2. String

El tipo `String` representa un caso más complejo debido a su naturaleza como una secuencia de caracteres. Sin embargo, en el contexto de CIL a MIPS, se toman decisiones específicas para tratar `String` de manera más eficiente:

- **Inmutabilidad y Copia**: las cadenas son inmutables, lo que significa que cualquier operación que parezca modificar una cadena en realidad resulta en la creación de una nueva cadena. Este enfoque simplifica la representación de las cadenas y su manipulación, permitiendo tratarlas como valores inmutables.
- **Representación en Memoria**: Una cadena se representa con dos atribútos, uno para el tamaño y otro como un puntero a una secuencia de caracteres. Sin embargo, en términos de operaciones, se trata como un valor.
- **Operaciones**: Las operaciones con cadenas, como la concatenación o la comparación, se manejan mediante funciones especializadas en MIPS. Estas operaciones trabajan con las direcciones de las cadenas pero no modifican las cadenas originales sino que crean copias de la instancia, reservando memoria y alojando en nuevos espacios.

Llamadas a Métodos en Tiempo Lineal

Las llamadas a métodos, tanto estáticas como dinámicas, se manejan de manera eficiente. Para un método estático, se realiza un salto directo al `label` correspondiente. En el caso de una llamada dinámica, se busca la instancia en la pila, se obtiene el tipo y se calcula la dirección del método basado en su índice, permitiendo así ejecutar el método correcto en tiempo lineal.

Manejo de VOID

En MIPS, `Void` se maneja como un tipo especial. Se asigna una dirección específica para representar `Void`, y cuando un objeto COOL tiene valor `void`, esta dirección se utiliza para indicarlo. Esto es esencial para expresiones como `isvoid`, donde se compara la dirección de un objeto con la dirección de `Void` para determinar si el objeto es nulo.

Funciones Específicas en MIPS

Algunas operaciones no tienen una traducción directa en MIPS y requieren implementaciones específicas. Ejemplos notables son `STR Compare` y `COPY BYTES`.

- **STR Compare**: Esta función compara dos cadenas caracter por caracter. Se utilizan registros específicos para cargar las direcciones de las cadenas y se itera sobre cada caracter comparándolos.
- **COPY BYTES**: Se utiliza para copiar una cantidad específica de bytes de una ubicación de memoria a otra. Es esencial para operaciones como la concatenación de cadenas o la copia de objetos.

Implementación de TypeOf y Manejo de Input

El TypeOf es una operación crucial que determina el tipo dinámico de un objeto en tiempo de ejecución. Se implementa cargando la dirección de memoria del tipo almacenado en el objeto.

El manejo del input, especialmente para cadenas, implica leer datos del buffer de entrada y almacenarlos en una ubicación de memoria asignada dinámicamente. Esta operación asegura que las entradas del usuario se manejen de manera eficiente y segura.

Stack y Frame Pointer

En MIPS, el stack es una estructura de datos tipo LIFO (Last In, First Out) utilizada para almacenar información temporal durante la ejecución de un programa. El stack pointer (`SP_REG`) apunta a la parte superior del stack. El frame pointer (`FP_REG`), por otro lado, se usa para mantener un punto de referencia constante al inicio de un frame de función, facilitando el acceso a variables locales y argumentos.

Proceso de Llamado a una Función

1. **Preparación para el Llamado**:
    - **Guardar Argumentos**: Los argumentos de la función se empujan al stack. Esto se hace utilizando el método `visit__ArgNode`, que coloca los argumentos en el stack antes de realizar la llamada a la función.
    - **Actualizar el Stack Pointer**: El stack pointer se ajusta para reservar espacio para las variables locales y argumentos de la función.
2. **Establecimiento del Nuevo Frame**:
    - **Guardado del Frame Pointer Actual**: Se guarda el valor actual del frame pointer (`FP_REG`) en el stack. Esto es importante para poder restaurar este valor una vez que la función haya terminado, manteniendo la integridad del stack.
    - **Actualización del Frame Pointer**: El frame pointer se actualiza para apuntar a la parte superior del stack actual. Esto marca el inicio del nuevo frame de la función.
3. **Ejecución de la Función**:
    - **Instrucciones de la Función**: Se ejecutan las instrucciones de la función, que pueden incluir operaciones aritméticas, acceso a variables, llamadas a otras funciones, etc.
    - **Acceso a Variables Locales y Argumentos** : El acceso a variables locales y argumentos se realiza a través de offsets relativos al frame pointer. Esto permite un acceso consistente sin importar los cambios en el stack pointer.
4. **Finalización de la Función**:
    - **Restauración del Frame y Stack Pointers**: Antes de finalizar la función, se restaura el frame pointer al valor que tenía antes de la llamada a la función. El stack pointer también se ajusta para deshacer los cambios realizados al inicio de la función.
    - **Retorno de Valor**: Si la función retorna un valor, este se coloca generalmente en un registro específico (`A1_REG`) antes de retornar al caller.
5. **Retorno al Caller**:
    - **Recuperación de Dirección de Retorno y Frame Pointer Anterior**: La dirección de retorno (`RA_REG`) y el frame pointer previo se recuperan del stack, permitiendo que el programa continúe su ejecución desde el punto donde se realizó la llamada a la función.
    - **Ajuste del Stack Pointer**: Se ajusta el stack pointer para remover cualquier dato relacionado con la función que acaba de finalizar.

Este proceso asegura que cada función tenga su propio espacio de trabajo en el stack, y que las variables locales y argumentos se manejen de forma aislada y segura. Además, permite que las funciones se llamen recursivamente sin interferir entre sí.

Proceso de Compilación Paso a Paso

1. Configuración Inicial y Manejo de Registros

- **Registro de Nombres**: Se definen los nombres de los registros disponibles en MIPS, como `t0`, `t1`, etc., y registros de argumentos como `a0`, `a1`, etc.
- **Registro para Procedimientos Integrados**: Se definen registros específicos (`s0`, `s1`, etc.) para ser utilizados en procedimientos integrados como `COPY_BYTES` y `STR_CMP`.
- **Manejador de Memoria (`MemoryManager`)**: Esta clase se encarga de administrar el uso de registros, manteniendo un registro de los utilizados y disponibles.

2. Visitando Nodos del AST

- **Nodos de Programa (`visit__ProgramNode`)**: Aquí se procesan los tipos (`dottypes`), los datos (`dotdata`), y las funciones (`dotcode`) del programa CIL. Se generan las secciones `.data` y `.text` para MIPS.
- **Nodos de Tipo (`visit__TypeNode`)**: Para cada tipo en CIL, se crea una entrada en la sección `.data` de MIPS, que incluye una tabla de métodos virtuales.
- **Nodos de Función (`visit__FunctionNode`)**: Cada función CIL se traduce a un conjunto de instrucciones MIPS. Se maneja la reserva de espacio para variables locales y se guardan las direcciones de retorno.

3. Traducción de Instrucciones Específicas

- **Asignaciones, Operaciones Aritméticas, y Control de Flujo**: Instrucciones como `AssignNode`, `PlusNode`, `MinusNode`, `GotoNode`, etc., son traducidas a un conjunto de instrucciones equivalentes en MIPS. Esto incluye manejo de operaciones aritméticas y de control de flujo.
- **Manejo de Cadenas y Entrada/Salida**: Procesos para operaciones con cadenas (`LoadNode`, `ConcatNode`, `SubstringNode`) y E/S (`PrintNode`, `ReadNode`) que implican interacciones más complejas con la memoria y llamadas al sistema.

4. Procedimientos Especiales

- `COPY_BYTES` y `STR_CMP`: Son procedimientos MIPS para copiar bloques de bytes y comparar cadenas, respectivamente. Estos procedimientos son esenciales para operaciones que no se pueden traducir directamente de CIL a MIPS.

5. Consideraciones Finales

- **Manejo de Registros y Memoria**: Se utiliza un enfoque cuidadoso para el manejo de registros y memoria, asegurando que los valores se almacenen y recuperen correctamente durante la ejecución del programa, sin embargo no se lleva ningún mecanismo para limpieza o aprovechamiento de la memoria alojada.