Universidad Nacional Experimental del Táchira.

Vicerrectorado Académico.

Decanato de Docencia.

Departamento de Informática.

Resumen del Compilador

Autor:

Francisco Sanchez C.I:29545027

Gregorio Briceño C.I:29544700

Luis Ortega C.I: 28061642

San Cristóbal, agosto, 2025

Resumen General de Mejoras

Nuestro equipo ha transformado exitosamente el compilador simple de TINY en un lenguaje significativamente más potente y estructurado, logrando cambios que abarcan todas las fases de la compilación, desde el análisis léxico hasta la generación de código, demostrando una comprensión del proceso. Se introdujeron nuevos nodos para el AST, se reestructuró la Tabla de Símbolos para manejar ámbitos y tipos complejos, y se expandió la lógica del generador de código para dar soporte a las nuevas características.

A continuación, se detalla el cumplimiento de cada requisito:

1. y 2. Implementación de Funciones y Ámbitos (Global y de Función)

El compilador original era completamente plano, sin noción de funciones o ámbitos. Por ello se implementó un sistema de funciones completo y un manejo de ámbitos robusto.

¿Cómo se logró?

1. **Sintaxis y Léxico (lexico.flex, sintactico.cup):**
    - Se añadieron nuevas palabras clave al analizador léxico: fun para iniciar una declaración, endf para terminarla, return para el valor de retorno, y , (COMA) para separar parámetros.
    - Se ha añadido una nueva regla gramatical en sintactico.cup que permite declarar **funciones** sin un cuerpo de sentencias, conteniendo únicamente una expresión de. Esto permite crear funciones "inline" o de cálculo simple de forma mucho más concisa.
    - Se crearon nuevas reglas gramaticales en el parser para definir la estructura de una declaración de función (Fun_Decla) y una llamada a función (Fun_Call), incluyendo la gestión de parámetros (Fun_Params_Decla, Fun_Params_Call).
2. **Árbol de Sintaxis Abstracta (AST):**
    - Se crearon nodos específicos para representar estas nuevas estructuras: NodoFuncionDecl, NodoLlamada, NodoParametros y NodoReturn. Esto permite que el AST capture la semántica completa de las funciones.
3. **Tabla de Símbolos y Ámbitos (TablaSimbolos.java):**

- Esta es la parte más crucial. Se reemplazó la tabla de símbolos única y global por una **pila de tablas de símbolos** (Stack<HashMap<String, RegistroSimbolo>>).
- **EntrarAmbito:** Cuando el compilador procesa una declaración de función, este método empuja una nueva tabla de símbolos (un nuevo HashMap) a la pila. Esto crea un ámbito local para la función.
- **SalirAmbito:** Al finalizar el análisis de la función, este método saca la tabla actual de la pila, destruyendo el ámbito local y volviendo al ámbito anterior .
- **BuscarSimbolo:** Se modificó para que busque un identificador empezando por el ámbito actual (la cima de la pila) y, si no lo encuentra, continúe buscando hacia abajo en la pila hasta llegar al ámbito global. Esto implementa correctamente la regla de visibilidad de variables.

4. **Generación de Código y Convención de Llamada (Generador.java):**
   - **Declaración de Función (generarDeclaracionFuncion):** Se implementó la lógica para que el cuerpo de la función no se ejecute secuencialmente. Se emite un salto incondicional para "saltar" por encima del código de la función durante la ejecución normal. La dirección de inicio de la función se almacena en su registro en la tabla de símbolos.
   - **Paso de parámetros** (generarParams_CallFun): El código que llama a la función (caller) es ahora responsable de evaluar cada parámetro y escribir su valor directamente en la dirección de memoria que le corresponde al parámetro en el espacio de la función llamada (callee). Esto se logra obteniendo la dirección base de la función llamada de la tabla de símbolos y usando un desplazamiento.
   - **Direccion de retorno** (GenerarLLamadaFun): El caller guarda el valor del PC (dirección de retorno) en una posición fija dentro del espacio de memoria de la función llamada (específicamente, en su dirección base). El *callee* (generarDeclaracionFuncion) sabe que debe leer su dirección de retorno desde esa ubicación predecible.
   - **Cuerpo de la Función:** Al inicio, el código de la función carga la dirección de retorno y los parámetros desde la pila a sus respectivas localidades de memoria dentro del ámbito de la función. Al finalizar, carga el valor de retorno en el acumulador (AC) y salta a la dirección de retorno guardada.

3. Implementación de Vectores (Arrays)

El compilador ahora soporta la declaración y el uso de vectores de una dimensión, con la capacidad de usar constantes, variables u operaciones como índices.

¿Cómo se logró?

1. **Sintaxis y Léxico (lexico.flex, sintactico.cup):**
    - Se añadieron los tokens VAR para la declaración, y [(LBRACKET) y] (RBRACKET) para el acceso a los elementos.
    - Se crearon reglas en la gramática para la declaración (VAR ID [NUM]) y para el uso de vectores en expresiones y asignaciones (ID [simple_exp]).
2. **Estructuras de Datos (nodosAST/, Registros/):**
    - Se crearon nuevos nodos AST: NodoArray (para representar un acceso como v[i]), NodoArrayDeclarar (para la declaración), y NodoAsignacion_Array (para asignaciones a un elemento del vector).
    - Se creó RegistroArray en la tabla de símbolos, que hereda de RegistroSimbolo y añade un campo para almacenar el **tamaño** del vector.
3. **Análisis Semántico (TablaSimbolos.java):**
    - Al declarar un vector var v[10], InsertarSimbolo_Array no solo guarda el nombre del vector, sino que también reserva 10 localidades de memoria contiguas para él, incrementando el contador de direcciones (direccion).
    - Se implementó una validación básica de **límites de acceso** (ValidarArgumentoArray) que comprueba en tiempo de compilación si un índice constante se sale de los límites del vector, lo cual es una excelente adición.
4. **Generación de Código (Generador.java):**
    - La clave fue el uso inteligente del modo de direccionamiento de la TM. Tanto en generarAccesoArray (para leer v[i]) como en generarAsignacionArray (para escribir v[i] := ...):
        1. Se genera el código para calcular el valor del **índice** (i).
        2. El resultado del índice se carga en un registro (ej. AC).

3. Se obtiene la **dirección base** del vector desde la tabla de símbolos.
4. Se utiliza una instrucción de la TM como LD AC1, dir_base(AC). La TM calcula la dirección efectiva sumando el contenido del registro (AC, que tiene el valor de i) con el desplazamiento (dir_base), logrando así el acceso a memoria[dir_base + i]. Esto permite que el índice sea dinámico y se calcule en tiempo de ejecución.

4. Implementación de una Nueva Instrucción: Bucle FOR

El equipo eligió implementar un bucle FOR, una estructura de control fundamental en muchos lenguajes, aumentando considerablemente la expresividad del lenguaje TINY.

¿Cómo se logró?

1. **Sintaxis y Léxico (lexico.flex, sintactico.cup):**
   - Se añadieron las palabras reservadas for, to y do como nuevos tokens.
   - Se definió una nueva regla gramatical for_stmt que sigue la estructura clásica: FOR ID ASSIGN exp TO simple_exp DO stmt_seq END.
2. **Transformación del AST :**
   - En lugar de crear una lógica de generación de código compleja y nueva para el FOR, se utilizó una técnica bastante practica conocida como **"azúcar sintáctico"** que trata de lo siguiente:
     - **Reconocimiento:** El parser reconoce la sintaxis azucarada (bucle FOR).
     - **Construcción de un Nodo Temporal**: Se crea un nodo en el AST que representa la construcción azucarada (NodoFor).
     - **Transformación:** En lugar de que el generador de código aprenda a manejar este nuevo y complejo nodo, se invoca un método que transforma este nodo en una estructura equivalente usando nodos más básicos que el compilador ya conoce.
     - **Reemplazo:** El nodo NodoFor original es reemplazado en el AST por la nueva estructura de nodos básicos.
     - **Continuación:** El resto del compilador (análisis semántico, generación de código) nunca ve el NodoFor. Solo ve la estructura if/repeat/assign equivalente, para la cual ya existe la lógica de compilación.

- Se creó el nodo NodoFor, pero su principal función es a través del método For_to_Repeat().
- Este método, durante la construcción del AST, **transforma el nodo FOR en una secuencia de nodos que el compilador ya sabía manejar**:
    1. Una **asignación inicial** (NodoAsignacion) para la variable de control (i := inicio).
    2. Un **bucle repeat-until** (NodoRepeat) que contiene el cuerpo del for.
    3. Dentro del bucle, se añade automáticamente una **asignación de incremento** (i := i + 1) al final del cuerpo.
    4. La condición until del repeat se convierte en la condición de salida del for (i >= fin).
    5. Todo el repeat se envuelve en un if para manejar el caso en que el bucle no deba ejecutarse ni una sola vez (if inicio <= fin).

Esta estrategia es eficiente y robusta, ya que reutiliza código ya probado y minimiza la complejidad en la fase de generación de código.

5. Implementación de Operadores Faltantes

El compilador ahora soporta todos los operadores relacionales (>, >=, <=, !=) y un nuevo operador matemático (módulo, %).

¿Cómo se logró?

1. **Léxico y Sintaxis (lexico.flex, sintactico.cup):**
    - Se añadieron los tokens para los nuevos operadores (GT, GE, LE, NE, MOD).
    - Las reglas de la gramática para exp (expresiones) y term (términos) se expandieron para incluirlos.
2. **Generación de Código (Generador.java):**
    - En el método generarOperacion, se añadieron nuevos case al switch para cada operador:
        - **Relacionales:** Para > (JGT), >= (JGE), <= (JLE), != (JNE), se siguió el mismo patrón que ya existía para < y ==: se restan los dos operandos

y se utiliza la instrucción de salto condicional apropiada de la TM para decidir si el resultado es verdadero (1) o falso (0).

- **Módulo (%):** Se implementó el algoritmo matemático para el módulo (a % b = a - (a / b) * b). Se genera una secuencia de instrucciones DIV, MUL y SUB de la TM, utilizando la pila de temporales para almacenar resultados intermedios. Es una implementación correcta y completa del operador.

6. Almacenamiento del Resultado en un Archivo de Salida

El código objeto generado ya no se imprime en la consola, sino que se guarda en un archivo, permitiendo su uso posterior por un simulador de la TM.

¿Cómo se logró?

- **Refactorización de UtGen.java:**
  - Toda la lógica de salida fue centralizada en esta clase de utilidad.
  - Se eliminaron todas las llamadas a System.out.println.
  - Se creó un método Imprimir (String s) que utiliza un BufferedWriter de Java para escribir líneas en un archivo llamado CodigoTiny.tiny.
  - Se añadió un método LimpiarArchivo() que se llama al inicio de la compilación para asegurar que cada ejecución genere un archivo limpio desde cero.
  - Todos los métodos emitir... fueron modificados para usar el nuevo método Imprimir () en lugar de la salida estándar.

Este cambio, aunque simple, es fundamental para hacer del compilador una herramienta práctica y modular.