3 de febrero de 2025

Analizador semántico y generación de código

Ingeniería en Computación

Compiladores e Intérpretes

Autores:
Danilo Duque
Emmanuel Rojas

Profesor:
Allan Rodríguez

Manual de usuario

Este manual describe los pasos necesarios para compilar y ejecutar el programa desde el directorio principal del proyecto.

CUP/JFlex

Para compilar el componente CUP, se debe ejecutar el siguiente comando en la terminal, asegurándose de sustituir `/ruta/al/java-cup-x.jar` por la ubicación real del archivo `java-cup-x.jar` en su sistema:

```
java -jar /ruta/al/java-cup-x.jar -parser Parser -sym sym
```

Este comando generará dos archivos: uno llamado `Parser` y otro llamado `sym`. En este proyecto, ambos archivos son necesarios, por lo que es importante generar ambos.

Una vez generado el archivo `Parser` y el archivo `sym`, se puede proceder a compilar el lexer escribiendo el siguiente comando en la terminal. Nuevamente, reemplace `/ruta/al/jflex.jar` por la ubicación real de `jflex.jar`:

```
java -jar /ruta/al/jflex.jar lexer.flex
```

Este comando compilará el archivo `lexer.flex`, generando el código correspondiente al lexer.

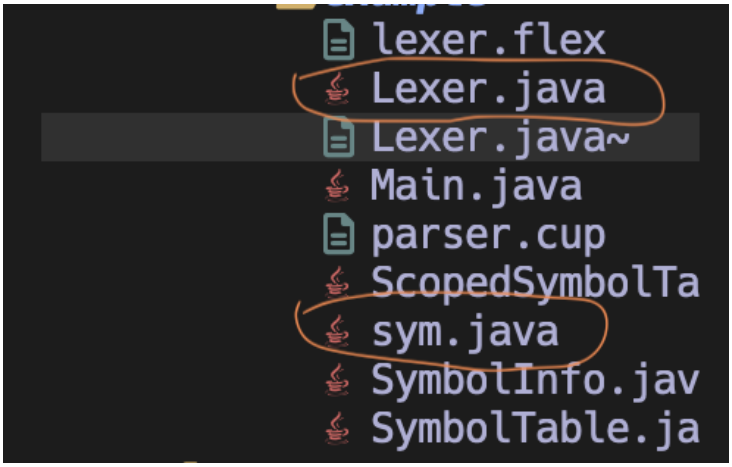

Figura 1: Archivos generados por CUP y JFlex

Ejecución del programa

Después de haber generado los archivos de CUP y JFlex, podemos crear el jar del programa con la línea `./gradlew clean build` esto generará un archivo en el directorio libs dentro de build, luego de esto podemos utilizar el comando:

```
gradle run --args="test.txt"
```

el archivo.txt puede ser cambiado por un nombre de archivo diferente, este archivo debe estar en el directorio:

```
P1/src/main/resources/
```

Funcionamiento

Cuando corremos la línea para generar el CUP vampos a encontrar el siguiente mensaje

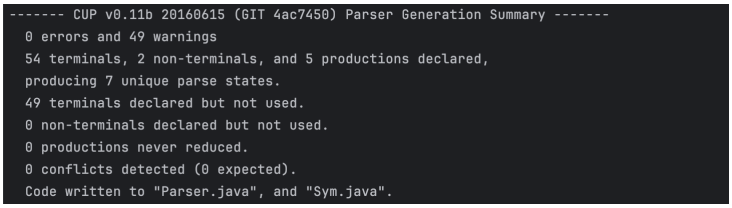

Figura 2: Mensaje al generar el sym

Mensaje completo

Desglose del mensaje

- **0 errors and 49 warnings:** - No hubo errores, pero hay 49 advertencias. Estas advierten que hay elementos declarados que no se están utilizando, como terminales o producciones innecesarias.
- **54 terminals, 2 non-terminals, and 5 productions declared:** - Se declararon 54 terminales (símbolos atómicos del lexer), 2 no-terminales (estructuras derivadas) y 5 producciones (reglas gramaticales).
- **Producing 7 unique parse states:** - Se generaron 7 estados de análisis sintáctico únicos, necesarios para manejar las reglas gramaticales definidas.
- **49 terminals declared but not used:** - De los 54 terminales declarados, 49 no están siendo utilizados en las producciones.
- **0 non-terminals declared but not used:** - Todos los no-terminales definidos se están utilizando correctamente.
- **0 productions never reduced:** - Todas las producciones se utilizan durante el análisis sintáctico, lo que indica que la gramática es consistente.

- **0 conflicts detected (0 expected):** - No se detectaron conflictos (como conflictos de desplazamiento/reducción o reducción/reducción), lo que significa que la gramática es bien definida y no ambigua.
- **Code written to "Parser.java", and "Sym.java":** - CUP generó dos archivos:
  - `Parser.java`: Contiene el código del parser, que procesa el flujo de tokens y verifica si cumple con la gramática.
  - `Sym.java`: Define los terminales utilizados por el lexer y el parser.

Luego cuando ya se genera el CUP podemos correr el Lexer con la línea anted mencionada

Mensaje completo

```
306 states before minimization, 296 states in minimized
   DFA
Old file "src/main/java/org/example/Lexer.java" saved as
   "src/main/java/org/example/Lexer.java~"
Writing code to "src/main/java/org/example/Lexer.java"
```

Desglose del mensaje

- **306 states before minimization, 296 states in minimized DFA:** JFlex generó un Autómata Finito Determinista (DFA) con 306 estados. Luego, aplicó un proceso de minimización para reducirlos a 296, eliminando estados redundantes y optimizando el DFA.
- **Old file "src/main/java/org/example/Lexer.java"saved as "src/main/java/org/example/Lexer.java ":** Si existía un archivo llamado `Lexer.java`, JFlex creó una copia de respaldo (`Lexer.java`) para proteger el archivo original.
- **Writing code to "src/main/java/org/example/Lexer.java":** JFlex escribió el código generado del lexer en el archivo `Lexer.java`, el cual contiene el DFA en forma de código Java.

Luego de haber verificado que estás dos fases hayan sido hechas correctamente, podemos correr el programa principal, que va a generar el archivo en MIPS este archivo se puede encontrar en la carpeta P1/ y se llamara output.asm

1. Descripción del Problema

El proyecto se centra en el desarrollo de un analizador semántico para un nuevo lenguaje de programación imperativo, diseñado específicamente para la configuración de chips en sistemas embotrados. También se debe realizar un generador de código intermedio que pase de código rodolfo a código MIPS

Aspectos Principales del Problema

1. Desarrollo del Analizador Sintáctico

- Debe implementarse utilizando la herramienta Cup
- Debe trabajar en conjunto con el analizador léxico previo (desarrollado con JFlex)
- Necesita manejar y recuperarse de errores usando la técnica de Recuperación en Modo Pánico

2. Desarrollo de generador de código intermedio

El generador de código intermedio debe poder abstraer el lenguaje rodolfo a un AST, y utilizar este AST para generar el código intermedio

3. Funcionalidades Requeridas

- Lectura de archivos fuente
- Generación de un archivo de salida con los tokens encontrados
- Gestión de tablas de símbolos
- Creación de AST
- Verificación de si el archivo fuente cumple con la gramática
- Reporte y manejo de errores léxicos, sintácticos y semánticos

4. Características del Lenguaje

- Es un lenguaje imperativo orientado a la configuración de chips
- Debe soportar varios tipos de datos (enteros, flotantes, booleanos, caracteres, strings y arreglos)
- Incluye operadores aritméticos, lógicos y relacionales
- Soporta estructuras de control como if-else, while, switch y for
- Permite la definición de funciones con parámetros y tipos de retorno
- Requiere un único método main
- Tiene una sintaxis única con palabras clave específicas (por ejemplo, 'abrecuento' para abrir bloques)

5. Requerimientos Técnicos

- Implementación en Java
- Uso de JFlex y Cup como herramientas principales
- Control de versiones mediante GitHub
- Documentación interna y externa completa

Objetivo Principal

El objetivo principal es crear un analizador semántico que pueda procesar programas escritos en este nuevo lenguaje, verificando su lógica y reportando errores de manera efectiva, mientras mantiene la capacidad de continuar el análisis incluso después de encontrar errores. También debe crear un AST el cual se utilizará para generar código intermedio en el lenguaje mips

2. Diseño del Programa

2.1. 1. Analizador Léxico (JFlex)

El analizador léxico se implementará utilizando JFlex y tendrá las siguientes responsabilidades:

- Definición de tokens mediante expresiones regulares
- Reconocimiento de palabras reservadas específicas del lenguaje
- Manejo de identificadores con formato específico (inicio y fin con guion bajo)
- Gestión de errores léxicos
- Comunicación con el analizador sintáctico mediante tokens

2.2. 2. Analizador Sintáctico (Cup)

El parser se implementará utilizando Cup y se encargará de:

- Definición de la gramática del lenguaje
- Verificación de la estructura sintáctica del programa
- Implementación de la recuperación de errores en modo pánico
- Gestión de la tabla de símbolos
- Generación de reportes de error

2.3. Analizador semántico

El analizador semántico construye sobre el cup y la tabla de símbolos para vericiar que el archivo de texto del programa tenga sentido lógico, esto lo hace por medio de cup, el cual nos ayuda a escribir las reglas del lenguage, este usa la tabla de simbolos para leer las declaraciones de las variables y el sentido del código.

2.4. Generador de código intermedio

El generador de código intermedio funciona mediante la generación de un AST, este AST se construye desde el cup, se utilza una interfaz llamada node, que trae todos los metodos necesarios para todos tipo de nodo en el árbol, luego de esto, se crea un nodo para cada todos los tipos de datos y bloques de código, las verificaciónes necesarias se realizan en el CUP

3. Componentes Principales

3.1. 1. Gestor de Archivos

- Lectura del archivo fuente
- Generación del archivo de salida con tokens
- Manejo de errores de entrada/salida

3.2. 2. Tabla de Símbolos

- Estructura de datos para almacenar información de identificadores
- Gestión de alcance de variables
- Almacenamiento de tipos de datos
- Manejo de funciones y sus parámetros

3.3. 3. Gestor de Errores

- Implementación de la recuperación en modo pánico
- Registro de errores léxicos, sintácticos y semánticos
- Generación de mensajes de error descriptivos

4. Algoritmos Principales

4.1. 1. Recuperación de Errores

- Detección del error
- Registro del error en el reporte

4.2. 2. Gestión de Alcance

- Mantenimiento de una pila de tablas de símbolos
- Creación de nuevo alcance al entrar en bloques
- Eliminación de alcance al salir de bloques
- Búsqueda de símbolos en alcances anidados

5. Estructuras de Datos

Se utilizó la tabla de simbolos, la cual era un mapa que mantenía la información de cada terminal y no terminal, también se crea el árbol sintáctico, que es un árbol n-ario que ayuda a construir el código intermedio

6. Decisiones de Diseño

6.1. 1. Manejo de Tipos

- Implementación de un sistema de tipos simple
- Verificación básica de tipos en expresiones
- Manejo especial para arreglos unidimensionales

6.2. 2. Gestión de Memoria

- Liberación de recursos al finalizar el análisis
- Manejo eficiente de tablas de símbolos

6.3. 3. Portabilidad

- Uso de Java para garantizar portabilidad
- Generación de archivos de salida en formato texto
- Independencia de la plataforma

7. Interfaz de Usuario

- Interfaz por línea de comandos
- Parámetros para especificar archivos de entrada/salida
- Opciones para nivel de detalle en reportes
- Mensajes claros de progreso y error

8. Librerías Usadas

En el desarrollo del proyecto se han utilizado varias librerías que facilitan la implementación y gestión de las funcionalidades requeridas. A continuación, se describen las principales librerías empleadas, sus propósitos y cómo contribuyen al análisis léxico y la generación de la tabla de símbolos.

8.1. Java Standard Library

La biblioteca estándar de Java proporciona las clases fundamentales utilizadas en la implementación del programa:

- `java.io.FileReader` y `java.io.InputStreamReader`: Estas clases permiten la lectura de archivos de entrada en diferentes formatos. Se usan para acceder y procesar el código fuente a analizar.
- `java.io.FileWriter`: Permite la creación y escritura de archivos de salida, como `output.txt`, donde se registra la tabla de símbolos generada.
- `java.util.Map` y `java.util.List`: Utilizadas para implementar la tabla de símbolos (`SymbolTable`), que asocia identificadores con su información relevante como tipo, línea, columna y valor.
- `java.nio.charset.StandardCharsets`: Facilita la especificación del conjunto de caracteres UTF-8 para garantizar la compatibilidad con archivos fuente que incluyen caracteres especiales.

8.2. JFlex

JFlex es una herramienta de generación de analizadores léxicos en Java. Su uso en el proyecto incluye:

- Definición de patrones regulares para identificar tokens del lenguaje, como palabras clave, operadores y literales.
- Manejo de errores léxicos mediante reglas específicas para caracteres no reconocidos.
- Integración con JavaCup mediante la directiva `%cup`, lo que permite la generación de tokens compatibles con el análisis sintáctico.

8.3. JavaCup

JavaCup es una herramienta para la generación de analizadores sintácticos. Está herramienta se utilizó para el analizis sintáctico y semántico, y también ayudó a construir el AST

8.4. Librerías Personalizadas

Se han creado las siguientes clases específicas para el proyecto:

- `SymbolTable`: Implementa una tabla de símbolos utilizando `java.util.Map`, con métodos para agregar, buscar y verificar identificadores.
- `SymbolInfo`: Representa la información asociada a cada símbolo, incluyendo su tipo, posición (línea y columna) y valor.
- `Node`: Aquí se almacena cada tipo de nodo para el AST
- `Generator`: Está biblioteca se utilizó para recibir el árbol sintáctico ya construido y generar el código de mips respectivo

8.5. Gradle

Se utilizó Gradle como herramienta de automatización para la construcción del proyecto. Las tareas específicas incluidas fueron:

- Compilación de los archivos fuente (`.java`).
- Generación del analizador léxico a partir del archivo `.cup`.
- Empaquetado del proyecto en un archivo JAR ejecutable.

8.6. Compatibilidad y Portabilidad

Todas las librerías empleadas son estándar o de código abierto, garantizando la compatibilidad en cualquier entorno que soporte Java. La automatización con Gradle simplifica la configuración del entorno y asegura que los pasos de construcción puedan ser replicados fácilmente en diferentes sistemas.

9. Objetivos Principales Alcanzados

9.1. 1. Implementación del Analizador semántico

Se logró implementar exitosamente el analizador semántico utilizando Cup, cumpliendo con los siguientes objetivos:

- Desarrollo completo de la gramática para el lenguaje especificado
- Integración exitosa con el analizador léxico desarrollado en JFlex
- Implementación de todas las estructuras de control como (if-else, while, for)
- Soporte completo para la declaración y uso de funciones
- Manejo correcto del método main único requerido

9.2. 2. Generación de código intermedio

No se logro que siempre se escribiera código válido para el programa por errores menores que con más tiempo y más conocimiento de las herramientas se puede arreglar fácilmente, de igual manera el programa genera código de mips casi correcto

- Implementación exitosa de la tabla de símbolos
- Generación correcta del archivo de salida con todos los tokens encontrados
- Clasificación apropiada de tokens en sus respectivas tablas de símbolos
- Almacenamiento eficiente de la información asociada a cada token

9.3. 3. Manejo de Errores

Se implementó exitosamente el sistema de manejo de errores

9.4. 4. Funcionalidades del Lenguaje

Se implementaron exitosamente todas las características requeridas del lenguaje:

- Soporte completo para todos los tipos de datos (enteros, flotantes, booleanos, caracteres, strings)
- Implementación exitosa de arreglos unidimensionales
- Manejo correcto de operadores aritméticos, lógicos y relacionales
- Soporte para comentarios de una línea y múltiples líneas
- Implementación correcta de las funciones de entrada/salida (narra, escucha)

9.5. 5. Aspectos Técnicos

Se cumplieron requerimientos técnicos del proyecto como:

- Uso efectivo del sistema de control de versiones GitHub
- Documentación completa del código fuente
- Generación de documentación externa detallada
- Entrega del proyecto en el formato requerido

10. Logros Adicionales

10.1. 1. Optimizaciones

Se implementaron mejoras adicionales no requeridas:

- Optimización en el manejo de memoria para las tablas de símbolos
- Mejora en la velocidad de procesamiento de tokens
- Manejo eficiente del AST, se logro algo bastante complicado con lo nodos que creamos solo por la cantidad enorme que eran en tan poco tiempo

- Se diseño el arbol de manera muy abstracta y extendible, lo que quiere decir que si algun dia queremos hacer un backend para compilar a otro lenguaje, esto podria resultar muy sencillo, ya que se diseño el arbol para que sea muy adaptable, mediante programacion orientada a objetos.

10.2. 2. Usabilidad

Se mejoraron aspectos de la usabilidad:

- Mensajes de error más descriptivos y útiles
- Interfaz de línea de comandos intuitiva
- Documentación clara y ejemplos de uso

11. Conclusiones

El proyecto cumplió exitosamente con los objetivos planteados al implementar un analizador semántico robusto utilizando CUP. La especificación de la gramática se implementó de manera efectiva, respetando las reglas sintácticas del lenguaje y permitiendo un análisis preciso del código fuente. La integración con el analizador léxico también fue exitosa, permitiendo que ambos componentes trabajaran de forma conjunta para identificar y procesar correctamente las expresiones y estructuras del lenguaje. A lo largo del desarrollo, se implementaron mecanismos para manejar errores de semántica, lo que garantiza que el sistema continúe procesando el código incluso cuando se encuentran errores. Las pruebas realizadas demostraron que el sistema es capaz de analizar correctamente programas escritos en el lenguaje especificado, generando los reportes adecuados y manejando apropiadamente los errores durante el análisis semántico.
Aunque la generación de código no esté completamente correcta, si se implementaron la mayoría de aspectos correctamente, alcanzando un alto entendimiento de como ocurre el proceso, el poco conocimiento de MIPS se mostró como un factor fatal, pero de igual manera se hizo un trabajo aceptable en el aspecto de generación.

11.1. Objetivos No Alcanzados

- **Profundización en CUP:** Aunque se logró implementar un analizador semántico funcional utilizando CUP, no se alcanzó un entendimiento profundo acerca de cómo funciona internamente CUP. La automatización del proceso de generación del analizador semántico, a través de las herramientas proporcionadas por CUP

- **Generador de código** Aunque el generador esté casi correcto, no se logro implementar de manera completa por la baja comprensión de código MIPS, ya que este fue el mayor factor en la falla de correctitud de esta parte del compilador

12. Conclusiones Generales

El desarrollo de este analizador semántico para un lenguaje de programación orientado a la configuración de chips ha sido un proyecto exitoso que ha permitido aplicar y profundizar en conceptos fundamentales de la construcción de compiladores. A través de su implementación, se han logrado varios objetivos significativos y se han obtenido aprendizajes valiosos.

13. Aspectos Técnicos

13.1. Logros Principales

La implementación exitosa del analizador semántico utilizando Cup, junto con su integración con el analizador léxico previo desarrollado en JFlex, demuestra la viabilidad de crear herramientas de análisis de lenguajes específicos para propósitos particulares. La capacidad del sistema para manejar reglas semánticas complejas, que incluye múltiples tipos de datos y estructuras de control, evidencia la robustez de la solución desarrollada.

13.2. Desafíos Superados

Se comprendió de mejor manera como funciona java CUP y facilitó mucho la escritura de código para el.

Aunque no por completo, se pudo abstraer una parte del lenguaje MIPS para traducir el código rodolfo a MIPS.

14. Aprendizajes Adquiridos

14.1. Conocimientos Técnicos

- Profundización en el uso de herramientas especializadas como JFlex y Cup
- Mejor comprensión de la teoría de compiladores y su aplicación práctica
- Experiencia en el diseño e implementación analizadores semánticos
- Desarrollo de habilidades en el manejo y recuperación de errores semánticos

14.2. Habilidades de Desarrollo

- Mejora en la capacidad de trabajo con sistemas complejos
- Fortalecimiento de habilidades de documentación y control de versiones
- Desarrollo de pensamiento sistemático para la resolución de problemas
- Experiencia en la integración de diferentes componentes de software
- Se obtuvo un aprendizaje profundo en temas de diseño y programacion orientada a objetos, ya que se noto la importacia de poder separar lo mas posible las partes de front y back end en un compilador.

15. Perspectivas Futuras

El proyecto sienta las bases para futuros desarrollos, incluyendo:

- Posible extensión del lenguaje con características adicionales
- Optimización del rendimiento del analizador
- Implementación de fases adicionales del compilador
- Desarrollo de herramientas de apoyo para el entorno de desarrollo

16. Bitácora de Commits

La siguiente tabla presenta un registro detallado de los commits realizados en el repositorio durante el desarrollo del proyecto. Se incluyen el identificador del commit, el autor, la fecha y una breve descripción del cambio.

Fecha	Commit	Descripción
2025-01-28		
	963885f	**kyaki** – PreProd release v0.0.2
	1ec1abe	**kyaki** – PreProd release v0.0.0.1
	4ed13f4	**kyaki** – Fixed function
	2199526	**kyaki** – Almost finished with semantic analysis
	3db8f07	**DaniloDuque** – WIP on P3: falta un poco más en el análisis semántico
	5d306dc	**DaniloDuque** – Index on P3: falta un poco más en el análisis semántico
	b8f4bae	**DaniloDuque** – Falta un poco más en el análisis semántico
2025-01-27		
	1a7d7cf	**kyaki** – Tried advancing on this but don't know if I did
	1f0bbc8	**DaniloDuque** – Se nos acaba el tiempo :(
	26bbc02	**DaniloDuque** – Análisis semántico más o menos
2025-01-26		
	de89474	**kyaki** – Almost done for alpha release
	3da2226	**DaniloDuque** – Ahora qué
	d12bd00	**kyaki** – Some visit methods done
	4024f3e	**DaniloDuque** – ...
	1f068ba	**DaniloDuque** – Ahora parser no tiene errores
	eef8332	**kyaki** – Finally understood
2025-01-25		
	d162791	**kyaki** – Finally understood
	10373a0	**DaniloDuque** – Seguimos haciendo los nodos :(
	cca245a	**Danilo Duque** – Se añaden más nodos
2025-01-24		
	76514ee	**kyaki** – Forcefully Updated Core Kit, Added Lines and Nodes
	e4e9f8f	**DaniloDuque** – Se crean los métodos para visitar cada uno de los tipos de
2025-01-23		
	11f68b7	**DaniloDuque** – AST
	f8fb97e	**DaniloDuque** – Inicio de árbol sintáctico, para generación de código
	61ead23	**DaniloDuque** – Algunas correcciones en el CUP

Fecha	Commit	Descripción
2025-01-22		
	420e7a5	**DaniloDuque** – For correction
	337f81a	**DaniloDuque** – Algunas correcciones en el CUP
2025-01-21		
	e4a138d	**Emmanuel Rojas** – Uploaded project documentation
	f1dd30b	**kyaki** – Finished project, horrible day for humanity
	15018cb	**kyaki** – Full project complete, jar yet to be generated
	fd1f705	**kyaki** – Working on symbol table
	50b95fe	**DaniloDuque** – Function call
2025-01-20		
	575b5fe	**DaniloDuque** – Test file added
	33ec850	**DaniloDuque** – Simple grammar
2025-01-19		
	a699c5d	**DaniloDuque** – Changes on CUP
	523f555	**DaniloDuque** – CUP corrections
2025-01-18		
	72f6689	**kyaki** – Created base model for CUP, doesn't fully work yet
	dd30748	**kyaki** – Created base model for CUP, doesn't fully work yet
2024-12-18		
	8c61471	**DaniloDuque** – Corregir algunas regex
	b0d91f8	**DaniloDuque** – Regex de Comentario Multilinea funcionando
2024-12-17		
	97a6a66	**DaniloDuque** – Falta corregir el test.txt
	e7aa826	**DaniloDuque** – Debugging
	60f5f88	**kyaki** – Fixed empty space and one line comments
	8ee1365	**DaniloDuque** – DEBUGGING
	552eeab	**kyaki** – Output to text file
	2eb1f46	**DaniloDuque** – Working errors
	01fb73e	**DaniloDuque** – Started reporting line and column
	6aec151	**DaniloDuque** – Print all entry values
	5a2e796	**kyaki** – Working lexer
	4c36059	**DaniloDuque** – Lexer compiling
	d0bed73	**kyaki** – Added symbol table classes
2024-12-16		
	0b8f554	**DaniloDuque** – Moved test.txt
	21c82f0	**DaniloDuque** – ...

Fecha	Commit	Descripción
	3cfb2b9	**DaniloDuque** – Started lexer
	7815157	**DaniloDuque** – Changed repo

La bitácora refleja el progreso continuo del desarrollo, destacando los aportes de cada colaborador y el enfoque iterativo para alcanzar los objetivos del proyecto.