UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESCUELA DE CIENCIAS Y SISTEMAS
ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 1
PRIMER SEMESTRE 2024

CompiScript+
Proyecto 2
Manual Técnico

Samuel Isaí Muñoz Pereira - 202100119
Auxiliar: Walter Alexander Guerra Duque
Ingeniero: Mario Bautista

Intérprete:

Un intérprete es un programa que ejecuta el código fuente de un programa línea por línea, traduciendo y ejecutando cada instrucción a medida que se encuentra. En lugar de traducir todo el código fuente de antemano, el intérprete lo interpreta y ejecuta en tiempo real.

El intérprete realiza funciones similares a las del compilador, como el análisis léxico, sintáctico y semántico, pero lo hace de manera incremental, mientras se ejecuta el programa. Esto significa que el intérprete puede proporcionar retroalimentación inmediata al usuario sobre posibles errores o resultados parciales.

Compilador:

Un compilador es un programa que traduce todo el código fuente de un programa en un solo proceso, desde el código fuente en un lenguaje de alto nivel hasta un código ejecutable de máquina específico. Este proceso consta de varias fases, incluyendo análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimización y generación de código de máquina.

Una vez que el compilador ha traducido todo el código fuente, produce un archivo ejecutable que contiene instrucciones de máquina que pueden ser ejecutadas directamente por la CPU del ordenador.

Diferencias clave:

1. **Fase de traducción**: Un compilador traduce todo el código fuente a la vez, mientras que un intérprete traduce y ejecuta el código línea por línea o en bloques pequeños.
2. **Tiempo de ejecución**: Con un compilador, el código fuente se traduce antes de la ejecución del programa, lo que significa que

el tiempo de ejecución es potencialmente más rápido. Con un intérprete, la traducción y ejecución ocurren simultáneamente, lo que puede hacer que el tiempo de ejecución sea más lento en comparación con un programa compilado.

3. **Portabilidad**: Los programas compilados generalmente están más optimizados para la plataforma específica en la que se compilaron, lo que puede hacer que sean menos portátiles entre diferentes sistemas. Los programas interpretados suelen ser más portátiles, ya que el intérprete puede ejecutar el mismo código en diferentes plataformas siempre que exista un intérprete compatible.
4. **Retroalimentación al usuario**: Los intérpretes pueden proporcionar retroalimentación inmediata al usuario durante la ejecución del programa, como mensajes de error en tiempo real. Los compiladores, por otro lado, suelen proporcionar mensajes de error después de la fase de compilación, lo que puede hacer que la depuración sea más difícil.

Analizador Léxico y Sintactio

Para este segundo proyecto se utiliza la herramienta Jison que contiene un analizador lexico y sintactico, encargado los arboles encargados de reconocer la gramática y los patrones

¿Qué es Jison?

Jison es una herramienta de generación de parsers para JavaScript. Un parser es un programa que analiza una secuencia de tokens (normalmente producidos por un analizador léxico) para determinar su

estructura gramatical según una determinada gramática formal. En términos simples, un parser toma una entrada (como una cadena de texto) y la convierte en una estructura de datos que representa la sintaxis de esa entrada.

Características principales de Jison:

1. **Generación de parsers LR(1)**: Jison utiliza la técnica de parsing LR(1), que es una variante del parsing bottom-up que puede manejar una amplia gama de gramáticas contextuales.
2. **Escrito en JavaScript**: Jison está escrito en JavaScript, lo que lo hace compatible con el ecosistema JavaScript y fácil de integrar en aplicaciones web y de servidor.
3. **Personalización**: Jison permite definir gramáticas personalizadas y reglas de producción para adaptarse a las necesidades específicas del parser que estás creando.
4. **Compatibilidad con Node.js y navegadores web**: Dado que está escrito en JavaScript, los parsers generados por Jison pueden ejecutarse tanto en entornos de Node.js como en navegadores web, lo que proporciona flexibilidad en cuanto a dónde pueden utilizarse.
5. **Facilidad de uso**: Jison proporciona una sintaxis clara y concisa para definir gramáticas y reglas de producción, lo que facilita la creación de parsers incluso para aquellos que no son expertos en teoría de compiladores.
6. **Soporte para acciones semánticas**: Además de definir la estructura de la gramática, Jison también permite asociar acciones semánticas con las reglas de producción, lo que te permite realizar acciones específicas (como construir un árbol de sintaxis abstracta) mientras se analiza la entrada.

¿Cómo funciona Jison?

El funcionamiento de Jison se puede resumir en los siguientes pasos:

1. **Definir la gramática**: Primero, debes definir la gramática que quieres que tu parser reconozca. Esto implica especificar los símbolos terminales y no terminales de la gramática, así como las reglas de producción que indican cómo se pueden combinar estos símbolos para formar cadenas válidas.
2. **Escribir las acciones semánticas (opcional)**: Si es necesario, puedes asociar acciones semánticas con las reglas de producción para realizar operaciones específicas mientras se analiza la entrada. Por ejemplo, puedes construir un árbol de sintaxis abstracta o realizar cálculos basados en la entrada.
3. **Generar el parser**: Una vez que hayas definido la gramática y las acciones semánticas (si las hay), puedes utilizar Jison para generar el código JavaScript del parser. Este código incluirá funciones que aceptan una cadena de entrada y devuelven una representación de la sintaxis de esa entrada de acuerdo con la gramática especificada.
4. **Utilizar el parser generado**: Finalmente, puedes utilizar el parser generado en tu aplicación JavaScript para analizar cadenas de entrada y realizar las acciones deseadas basadas en la estructura sintáctica de esas cadenas.

¿Para qué se utiliza Jison?

Jison se puede utilizar para una variedad de tareas que implican el análisis de lenguajes formales. Algunos ejemplos de casos de uso incluyen:

- **Compiladores**: Jison se puede utilizar para construir el front-end de un compilador, que es responsable de analizar el código

fuente y convertirlo en una representación intermedia que luego puede ser optimizada y traducida a código objeto.

- **Interpretes de lenguajes de programación**: Jison se puede utilizar para construir el analizador sintáctico de un intérprete de un lenguaje de programación, permitiendo que el intérprete comprenda y ejecute programas escritos en ese lenguaje.
- **Análisis de lenguajes de marcado**: Jison se puede utilizar para analizar lenguajes de marcado como HTML o XML, permitiendo extraer información estructurada de documentos en estos formatos.
- **Análisis de lenguajes de consulta**: Jison se puede utilizar para analizar lenguajes de consulta como SQL, permitiendo a las aplicaciones entender y ejecutar consultas sobre bases de datos.

Analizador Semántico

Un analizador semántico es una parte fundamental de un compilador o intérprete que se encarga de analizar el significado de las estructuras sintácticas encontradas en el código fuente de un programa. Mientras que el analizador léxico y el analizador sintáctico se enfocan en la estructura y la gramática del código, respectivamente, el analizador semántico va un paso más allá y se preocupa por entender el significado o la semántica de las construcciones encontradas en el código.

Funciones del analizador semántico:

1. **Verificación de tipos**: Una de las tareas principales del analizador semántico es verificar que las operaciones en el código están siendo realizadas sobre tipos de datos compatibles.

Por ejemplo, en un lenguaje de programación, si se está intentando sumar un número con una cadena de caracteres, el analizador semántico debería detectar este error y reportarlo.

2. **Resolución de nombres**: En muchos lenguajes de programación, las variables y funciones deben ser declaradas antes de ser utilizadas. El analizador semántico se encarga de verificar que los nombres de variables y funciones están siendo utilizados de manera consistente y que hacen referencia a entidades previamente definidas en el código.
3. **Chequeo de alcance (scope)**: El analizador semántico verifica que las variables y funciones están siendo utilizadas dentro del alcance en el que fueron definidas. Esto incluye detectar variables no declaradas o el uso de variables fuera de su alcance léxico.
4. **Optimizaciones**: Aunque no es una función exclusiva del analizador semántico, a menudo se realizan algunas optimizaciones a nivel semántico, como la eliminación de código muerto o la simplificación de expresiones constantes.
5. **Resolución de referencias**: En lenguajes que permiten referencias a otras entidades, como punteros en C/C++ o referencias a objetos en lenguajes orientados a objetos, el analizador semántico se encarga de resolver estas referencias y verificar su consistencia.

Importancia del analizador semántico:

El analizador semántico desempeña un papel crucial en el proceso de compilación o interpretación de un programa, ya que garantiza que el código fuente esté correctamente escrito en términos de su significado y comportamiento. Detecta errores que no son evidentes en la etapa de análisis sintáctico y asegura que el programa final sea coherente y cumpla con las reglas del lenguaje de programación.

Partes importantes del Código

El código esta estructurado de una manera que todas las expresiones, instrucciones, símbolos, etc. estén separados de esta manera sera un poco mas entendible la estructura del proyecto.

Todas las instrucciones tienen un método llamado interpretar, este método se encarga de interpretar la clase que ha sido declarada con anterioridad desde el analizador sintáctico, este método interpretar es supremamente importante para el proyecto ya que va ejecutando cada una de las instrucciones de forma recursiva, existen algunas instrucciones que retornas un valor especifico, en caso que no retorne un valor especifico retorna la clase del método interpretar

```
const {Instruccion, tipoInstruccion} = require('../instruccion');
const Entorno = require('../Entorno/entorno');
let { agregarSalida, agregarError } = require('../salidas');

class If extends Instruccion {
    constructor(condicion, instrucciones, fila, columna) {
        super(tipoInstruccion.IF, fila, columna);
        this.condicion = condicion;
        this.instrucciones = instrucciones;
    }

    interpretar(entorno) {

        this.condicion.interpretar(entorno);

        let entornoIf = new Entorno(tipoInstruccion.IF, entorno)
        //console.log(this.condicion)
        //console.log(this.condicion)
        if (this.condicion.tipo != "BOOL") {
            console.log("Error Semántico: La condición del if no es booleana.")
            agregarSalida("Error Semántico: La condición del if no es booleana.");
            agregarError("Semántico", "La condición del if no es booleana.", this.fila, this.columna)
            return this;
        }
```