



Instituto Tecnológico de Costa Rica

Unidad de Computación

Compiladores E Intérpretes

Verano I 2024

Tercer Proyecto

Profesor

Allan Rodriguez Davila

Estudiantes

Deivid Matute Guerrero

Joshua Sancho Burgos

Centro Académico de Limón

21 de Enero del 2024

Manual de Usuario

Creación del Lexer:

Para programar el lexer es necesario instalar las librerías JFlex y CUP. Una vez instaladas e implementadas en el proyecto se procede a crear un archivo en formato “.jflex”, “.flex”, o “.lex” que son los formatos compatibles con la librería JFlex. Este archivo se divide en 3 secciones, el código de usuario (donde se realizan configuraciones, se declara el paquete y se realizan los imports), opciones y declaraciones (donde se declaran los macros y estados) y las reglas léxicas (donde se especifican los tokens que regresará el lexer). Es importante que la configuración “%cup” esté incluida para ser compatible con el parser.

Creación del Parser:

Para programar el parser es necesario instalar e implementar la librería CUP. Una vez terminada la programación del lexer se procede a programar el parser. Primero se realizan los ajustes necesarios para la compatibilidad con JFlex (imports y scanner), luego se escribe código para el parser (manejo de errores) y luego se declaran los símbolos terminales y no terminales. Los terminales deben ser los mismos que se declararon en el lexer, si no, se generará un error, al no existir tal símbolo en dado caso. Cuando se tiene certeza de haber agregado todos los símbolos no terminales se procede a programar los símbolos no terminales donde se debe ajustar la gramática para el análisis sintáctico y asegurarse que entren todas las posibles formas de generar un programa. Para generar el parser es necesario que exista un símbolo no terminal que de entrada al programa. El parser tiene la opción de elevar los resultados que encuentra con la función RESULT, lo cual resulta muy útil para almacenar y controlar la información encontrada y la tabla de símbolos en esta segunda parte.

Generación de Parser y Lexer:

Para generar el lexer y el parser se tienen 2 opciones.

La primera es ejecutar la función “GenerarLexerParser()” en el archivo “App.java”. Esta función elimina los archivos “lexer.java”, “parser.java” y “sym.java” si son archivos existentes en el sistema, en caso contrario, los genera si se encuentran los archivos “lexer.flex” y “parser.cup” en las carpetas correspondientes y los mueve de su localización hacia la carpeta “ParserLexer”; en caso de que estos archivos no existan

(en la última actualización del proyecto, estos archivos sí existen), se deberá localizar el archivo MainJFlexCup.java y comentar la línea 4, que contiene lo siguiente: “import ParserLexer.lexer;”, además de comentar la función “LexerTest” completa, ya que estas partes del código requieren de la existencia del archivo “lexer.java”.

La otra forma es generarlos manualmente por medio de los comandos del manejador de paquetes Maven, con “mvn jflex:generate” para generar a “lexer.java” y “mvn cup:generate” para generar a “parser.java” y “sym.java”, sin embargo, no se recomienda usar esta opción, ya que localiza los archivos en target y no los mueve a la carpeta “ParserLexer”, por lo que se pueden ocasionar algunas incompatibilidades.

Tabla de Símbolos:

Para crear la tabla de símbolos se realizaron tres clases:

- TabSymbol: Unidad mínima de la tabla de símbolos, donde se crean los símbolos de arreglos, variables y parámetros. Se almacena su nombre, tipo (int, float, bool, char, ...), clase (arr, var, param) y en el caso de los arreglos, su tamaño. Cuando estos símbolos son declarados el parser este los añade a la tabla de símbolos correspondiente.
- SymbolTable: Cada vez que el parser encuentra una declaración de función se genera una instancia de la tabla de símbolos. En esta se almacenan características de la función como nombre, tipo de retorno y sus símbolos. La clase posee funciones para agregar, obtener e imprimir símbolos.
- SymbolTableManager: Esta clase es un administrador de tablas de símbolos. Permite llevar control de las tablas, obtenerlas e imprimir sus valores.

Analizador Semántico:

Para abordar el análisis semántico se han creado dos clases:

- ExpressionTree: Esta clase implementa el análisis semántico utilizando una representación XML de las expresiones y verifica la coherencia semántica según el tipo esperado para cada expresión. La clase recorre y analiza nodos XML representativos de literales, identificadores, llamadas a funciones, operadores aritméticos, relacionales y lógicos, así como elementos de arreglos. La verificación semántica incluye la validación de tipos de variables, literales y funciones, así como la coherencia de estructuras y operadores en las expresiones. El código aborda también expresiones aritméticas, relacionales y lógicas, así como llamadas a funciones y manipulación de arreglos. En caso de encontrar errores semánticos, se notifica mediante la consola indicando la naturaleza del problema.
- Semantic: Esta clase proporciona utilidades para nuestro análisis semántico, con enfoque en la verificación de tipos y la coherencia semántica. Incluye

expresiones regulares para validar literales de diferentes tipos, identificadores y llamadas a funciones. Además, ofrece métodos para realizar la verificación semántica de arrays, el tipo de retorno de funciones y el tipo de variables identificadas. La clase aborda la validación de formatos de literales, manejo de errores semánticos y la coherencia de tipos en diferentes contextos, facilitando así la interpretación semántica de expresiones y estructuras en un código fuente.

- BlockTree: Esta clase se centra en el procesamiento de bloques de funciones representados como cadenas XML. Además aquí se contienen métodos para analizar y procesar un bloque de función, verificando la presencia de elementos de retorno y realizando análisis semántico de expresiones de retorno. También incluye métodos para convertir cadenas XML en objetos Document para su posterior procesamiento y manejo de errores en caso de estructuras XML no válidas.

Pruebas de funcionalidad

Para probar el análisis semántico realizado con todas sus validaciones y funcionalidades se realizaron las siguientes pruebas:

Ejemplo de código funcional:

```
function int main() {  
    local int ab|  
    return 0|  
}  
  
function int alfa(int y[], float u, int a){  
return 0|  
}  
  
function int beta(int i, float f){  
    return 0|  
}  
  
function int func_a(int a, float b, int f) {  
    local int x|  
    local char c|  
    local int array_i[] <= [1, 2, 3]|  
    local char array_a[] <= ['a','b','c']|  
    local int gid[] <= [1, 2]|  
    x <= 5|  
    local int xyz[] <= [gid[x], gid[1 + x + beta(gid[2], 1.0) + 4], x, gid[2]]|  
    return x + 1 * beta(1, 4.8)|  
}
```

Ejemplo de código donde se intenta declarar un arreglo de tipo float:

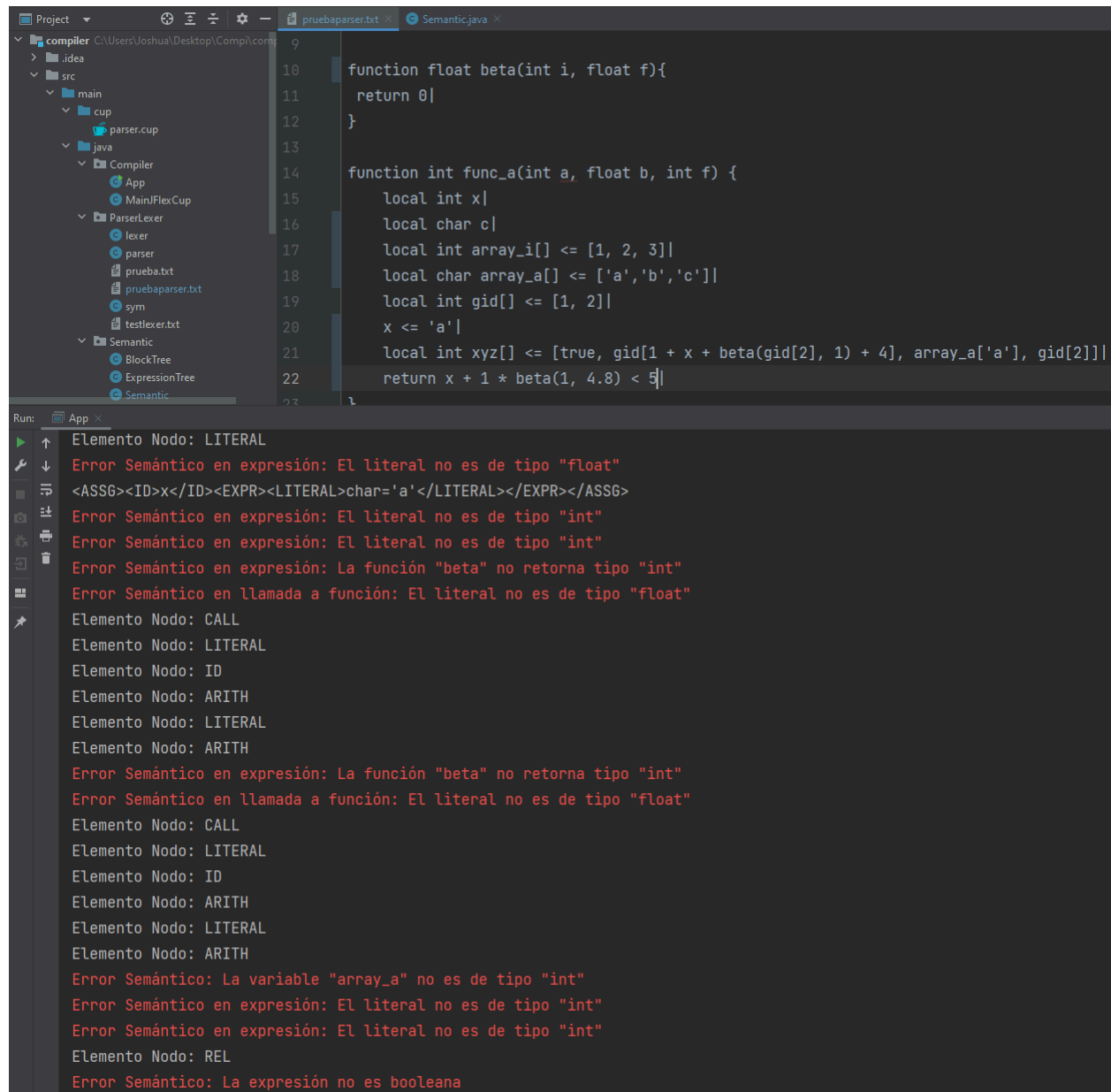
```
10 function int beta(int i, float f){
11     return 0|
12 }
13
14 function int func_a(int a, float b, int f) {
15     local int x|
16     local char c|
17     local int array_i[] <= [1, 2, 3]|
18     local char array_a[] <= ['a','b','c']|
19     local int gid[] <= [1, 2]|
20     x <= 5|
21     local float xyz[] <= [gid[x], gid[1 + x + beta(gid[2], 1.0) + 4], x, gid[2]]|
22     return x + 1 * beta(1, 4.8)|
23 }
24
25
```

Run: App x

Token: 42, value: return, Line: 21, Column: 4
Token: 48, Value: x, Line: 21, Column: 11
Token: 3, Value: +, Line: 21, Column: 13
Token: 26, Value: 1, Line: 21, Column: 15
Token: 5, Value: *, Line: 21, Column: 17
Token: 48, Value: beta, Line: 21, Column: 19
Token: 30, Value: (, Line: 21, Column: 23
Token: 26, Value: 1, Line: 21, Column: 24
Token: 2, Value: ,, Line: 21, Column: 25
Token: 27, Value: 4.8, Line: 21, Column: 27
Token: 31, Value:), Line: 21, Column: 30
Token: 46, Value: |, Line: 21, Column: 31
Token: 35, Value: }, Line: 22, Column: 0
Cantidad total de lexemas: 167
Cantidad de lexemas validos: 167
Cantidad de lexemas invalidos: 0

ANÁLISIS SINTÁCTICO
Elemento Nodo: LITERAL
Elemento Nodo: LITERAL
Elemento Nodo: LITERAL
<ASSG><ID>x</ID><EXPR><LITERAL>int=5</LITERAL></EXPR></ASSG>
Error Semántico: Tipo inválido especificado: float. Tipos aceptados: int o char.
Elemento Nodo: ARITH
Elemento Nodo: LITERAL

Ejemplo con diversos errores semánticos:



The screenshot shows an IDE with a project named 'compiler' and a file named 'pruebaparser.txt'. The code in the editor is as follows:

```
9  
10 function float beta(int i, float f){  
11     return 0|  
12 }  
13  
14 function int func_a(int a, float b, int f) {  
15     local int x|  
16     local char c|  
17     local int array_i[] <= [1, 2, 3]|  
18     local char array_a[] <= ['a','b','c']|  
19     local int gid[] <= [1, 2]|  
20     x <= 'a'|  
21     local int xyz[] <= [true, gid[1 + x + beta(gid[2], 1) + 4], array_a['a'], gid[2]]|  
22     return x + 1 * beta(1, 4.8) < 5|
```

The console output shows the following errors:

```
Run: App  
Elemento Nodo: LITERAL  
Error Semántico en expresión: El literal no es de tipo "float"  
<ASSG><ID>x</ID><EXPR><LITERAL>char='a'</LITERAL></EXPR></ASSG>  
Error Semántico en expresión: El literal no es de tipo "int"  
Error Semántico en expresión: El literal no es de tipo "int"  
Error Semántico en expresión: La función "beta" no retorna tipo "int"  
Error Semántico en llamada a función: El literal no es de tipo "float"  
Elemento Nodo: CALL  
Elemento Nodo: LITERAL  
Elemento Nodo: ID  
Elemento Nodo: ARITH  
Elemento Nodo: LITERAL  
Elemento Nodo: ARITH  
Error Semántico en expresión: La función "beta" no retorna tipo "int"  
Error Semántico en llamada a función: El literal no es de tipo "float"  
Elemento Nodo: CALL  
Elemento Nodo: LITERAL  
Elemento Nodo: ID  
Elemento Nodo: ARITH  
Elemento Nodo: LITERAL  
Elemento Nodo: ARITH  
Error Semántico: La variable "array_a" no es de tipo "int"  
Error Semántico en expresión: El literal no es de tipo "int"  
Error Semántico en expresión: El literal no es de tipo "int"  
Elemento Nodo: REL  
Error Semántico: La expresión no es booleana
```

Descripción del Problema:

Contexto: En el mismo ámbito que nos encontramos del desarrollo de compiladores, donde cada fase es importante para su interpretación y traducción del código fuente. En este proyecto, nos centraremos en las etapas de análisis semántico y generación de código destino en MIPS, basándonos en la gramática descrita principalmente en la tarea y en los avances de los proyectos I y II. Nuestro objetivo es construir un Analizador Semántico y un Generador de Código Destino para un lenguaje de programación compuesto por una secuencia de declaraciones de procedimientos.

Problema: El presente reto al que nos enfrentamos se divide en dos partes. Por un lado, necesitamos desarrollar un Analizador Semántico que pueda tomar un archivo fuente y verificar si puede ser generado por la gramática, además de manejar los errores léxicos, sintácticos y semánticos encontrados. Por otro lado, necesitamos desarrollar un Generador de Código Destino que pueda traducir el archivo fuente a código MIPS.

Desafíos Específicos:

- **Preservación y Corrección de Alcances:** El proyecto debe mantener y corregir los alcances definidos en los Proyectos I y II. Esto implica un entendimiento profundo de los proyectos anteriores y capacidad para mejorarlos.
- **Verificación de la Gramática:** El analizador debe ser capaz de determinar si el archivo fuente puede ser generado por la gramática. Esto requiere un robusto análisis y conocimiento de la gramática, la sintaxis y la semántica del lenguaje.
- **Manejo de Errores:** El analizador debe ser capaz de manejar los errores léxicos, sintácticos y semánticos encontrados utilizando la técnica de Recuperación en Modo Pánico. Esto implica la capacidad de detectar errores y recuperarse de ellos de manera eficiente.
- **Generación de Código MIPS:** El generador de código destino debe ser capaz de traducir el archivo fuente a código MIPS. Esto requiere un buen dominio del lenguaje MIPS y la capacidad de generar código que sea semánticamente equivalente al código fuente.

- Generación de Código sin Código Intermedio: El generador de código destino debe ser capaz de generar código sin utilizar código intermedio. Esto implica la capacidad de traducir directamente el código fuente a código MIPS.
- Generación de Código a partir de Código Intermedio: El generador de código destino debe ser capaz de generar código a partir de código intermedio generado por el compilador. Esto implica la capacidad de trabajar con código intermedio y traducirlo a código MIPS.
- Objetivo Final: El objetivo final es desarrollar un Analizador Semántico y un Generador de Código Destino robustos y eficientes que sean capaces de interpretar correctamente la estructura sintáctica del código fuente y generar el código MIPS correspondiente. Estas herramientas permitirán la construcción de un compilador completo capaz de procesar y comprender el código fuente de manera precisa y coherente.

Diseño del programa

Especificaciones Iniciales:

- Definición de las reglas semánticas para especificar la estructura semántica del lenguaje.

Implementación del Analizador Semántico con Java:

- Utilización del lenguaje Java para implementar las reglas semánticas del lenguaje.
- Creación de un archivo “.java” con las clases y métodos necesarios para el análisis semántico.
- Desarrollo de funciones para verificar el tipo, el alcance y la validez de las expresiones, variables y funciones del lenguaje.
- Generación de código intermedio o código MIPS según la opción elegida.
- Manejo de errores semánticos utilizando la técnica de Recuperación en Modo Pánico.

Generación de función checkExpressionType en ExpressionTree:

- Recepción de una cadena XML representando la expresión a verificar, el tipo esperado para la expresión, un mapa de símbolos con información sobre variables y un gestor de la tabla de símbolos.
- Verificación semántica del tipo de expresión contenida en la cadena XML.
- Procesamiento específico según la etiqueta raíz de la expresión, con manejo de errores semánticos y llamadas a funciones auxiliares.
- Devolución del nombre de la etiqueta raíz.

Generación de función traverseNode en ExpressionTree:

- Recorre un nodo XML y realiza análisis semántico según la etiqueta del nodo y su contenido.
- Recepción de un nodo XML, el tipo esperado para la expresión, un mapa de símbolos con información sobre variables y un gestor de la tabla de símbolos.
- Devolución de un resultado que incluye información sobre elementos OP1, OP2 y SYM.

Generación de función extractValue en ExpressionTree:

- Extrae y concatena el contenido de texto de un elemento XML.

- Recepción de un elemento XML.
- Devolución del contenido de texto del elemento XML.

Generación de función convertDocumentToString en ExpressionTree:

- Convierte un objeto Document XML a una cadena de texto con formato.
- Recepción de un objeto Document XML.
- Devolución de una cadena de texto con formato que representa el documento XML.

Generación de función checkLogicExpr en ExpressionTree:

- Realiza la verificación semántica de una expresión lógica.
- Recepción de un elemento XML que representa la expresión lógica, el tipo esperado de la expresión lógica, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de error semántico.
- Obtención de los elementos "OP1" y "OP2" y verificación semántica de estos elementos mediante llamadas a la función checkSemanticExpr.

Generación de función checkRelExpr en ExpressionTree:

- Realiza la verificación semántica de una expresión relacional.
- Recepción de un elemento XML que representa la expresión relacional, el tipo esperado de la expresión relacional, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de error semántico.
- Obtención de los elementos "OP1", "SYM" y "OP2" y de los tipos de los hijos.
- Verificación de la estructura de la expresión relacional y de los tipos de los elementos.
- Manejo de patrones y operadores relacionales permitidos.

Generación de función checkArithExpr en ExpressionTree:

- Realiza la verificación semántica de una expresión aritmética.
- Recepción de un elemento XML que representa la expresión aritmética, el tipo esperado de la expresión aritmética, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de error semántico.

- Obtención de los elementos "OP1" y "OP2" y realización del análisis semántico de los hijos.
- Obtención de los tipos de los hijos y verificación de la estructura de la expresión aritmética.

Generación de función checkSemanticExpr en ExpressionTree:

- Realiza la verificación semántica de una expresión.
- Recepción de un elemento XML que representa la expresión, un mapa de símbolos para verificar tipos de variables, el tipo esperado de la expresión y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de error semántico.
- Obtención de información del primer hijo del elemento y realización de acciones basadas en el tipo y contenido del hijo.

Generación de función getChildTag en ExpressionTree:

- Obtiene la etiqueta del primer hijo del elemento padre y realiza acciones basadas en el tipo y contenido del hijo.
- Recepción de un elemento padre, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Devolución de la etiqueta del primer hijo o null si no hay hijo o la etiqueta no es reconocida.

Generación de función processCall en ExpressionTree:

- Procesa una llamada a función (CALL) y verifica la validez de los argumentos según los tipos esperados por la función.
- Recepción de un nodo que representa la llamada a función, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de errores semánticos en la llamada a función.
- Obtención del nombre de la función y de la lista de argumentos.
- Procesamiento de cada argumento y verificación de su validez en términos de tipos esperados por la función.

Generación de función processExpr en ExpressionTree:

- Procesa una expresión (EXPR) y verifica la validez semántica según el tipo esperado.
- Recepción de un elemento que representa la expresión, el tipo esperado para la expresión, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de errores semánticos en la expresión.
- Obtención del primer hijo del elemento y realización de acciones basadas en el tipo y contenido del hijo.

Generación de función processArray en ExpressionTree:

- Procesa un elemento de arreglo (ARRAY) verificando la validez semántica de cada expresión.
- Recepción de un elemento que representa el arreglo, el tipo esperado para las expresiones dentro del arreglo, un mapa de símbolos para verificar tipos de variables y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de errores semánticos en alguna de las expresiones del arreglo.
- Obtención de los nodos de expresión dentro del arreglo y procesamiento de cada expresión.

Generación de función processArrElem en ExpressionTree:

- Procesa un elemento de arreglo (ARR_ELEM), realizando análisis semántico en la etiqueta ID y EXPR.
- Recepción de un elemento que representa un elemento de arreglo, un mapa de símbolos para verificar tipos de variables, el tipo esperado para el elemento de arreglo y un gestor de la tabla de símbolos para funciones.
- Lanzamiento de una excepción en caso de errores semánticos en la etiqueta ID o EXPR del elemento de arreglo.
- Obtención del elemento que representa la etiqueta ID y verificación del análisis semántico para dicha etiqueta.
- Obtención del elemento que representa la etiqueta EXPR y verificación del análisis semántico para dicha etiqueta.

Manejo de Errores:

- Generación de mensajes de error claros y descriptivos utilizando técnicas de análisis de contexto.

Pruebas y Depuración:

- Creación de conjuntos de pruebas exhaustivos que cubren diversos aspectos del lenguaje.
- Ejecución de pruebas para verificar la correcta identificación, clasificación y verificación de los elementos del lenguaje.
- Depuración de errores y ajuste de las reglas léxicas, gramaticales y semánticas según sea necesario.

Documentación:

- Creación de documentación clara y detallada que describa el funcionamiento del analizador semántico y cada una de sus funciones.
- Inclusión de ejemplos de uso y casos de prueba en la documentación.

Integración en un Compilador:

- Integración del analizador léxico, sintáctico, semántico y del generador de código en un compilador completo capaz de transformar el código fuente en código ejecutable.

Librerías usadas

- w3c.dom: Esta librería proporciona clases e interfaces para representar y manipular documentos XML y HTML utilizando el Document Object Model (DOM). El DOM organiza los documentos como estructuras de árboles, donde cada nodo representa una parte del documento, como elementos, atributos y texto. En el contexto del proyecto se utiliza para acceder y manipular nodos en el árbol de un documento XML, facilitando la representación y manipulación de la estructura del código fuente.
- xml.sax: La librería xml.sax ofrece clases para procesar documentos XML de manera secuencial y basada en eventos mediante el Simple API for XML (SAX). En el proyecto se usa para proporcionar el origen de datos para la lectura de documentos XML, facilitando la entrada de datos para el análisis del código fuente.
- javax.xml.parsers: Este paquete proporciona clases para manejar la creación y configuración de objetos DocumentBuilder que permiten analizar documentos XML y construir objetos Document. En nuestro proyecto se emplean para analizar y construir el árbol de sintaxis abstracta (AST) a partir del código fuente XML, representando la estructura semántica del código.
- java.util.regex: La librería java.util.regex ofrece clases como Matcher y Pattern para trabajar con expresiones regulares. Estas expresiones son patrones que permiten buscar, validar y manipular cadenas de texto de manera eficiente. En el contexto del proyecto este se utiliza para realizar coincidencias de patrones en cadenas de texto, lo que puede ser esencial para identificar elementos semánticos específicos durante el análisis, como nombres de variables, operadores, etc.

Análisis de resultados

Tarea / Requerimiento	Estado	Observaciones
Análisis semántico en asignaciones de variables	100%	
Análisis semántico en declaraciones de variables	100%	
Análisis semántico en declaración de arrays	100%	
Análisis semántico en elementos de array	100%	
Análisis semántico en asignaciones de elementos de array	100%	
Análisis semántico en llamadas a funciones (coincidencia de parámetros)	100%	
Análisis semántico a expresiones aritméticas	100%	
Análisis semántico a expresiones relacionales	100%	
Análisis semántico a expresiones lógicas	100%	
Análisis semántico a funciones con retorno	100%	
Análisis semántico para evitar duplicación de arrays y funciones	100%	
Análisis semántico a expresiones sin tipo	0%	
Diseño y desarrollo de un generador de código.	0%	

Implementación de la funcionalidad para escribir código MIPS en un archivo.	0%	
Implementación de la funcionalidad para seleccionar el archivo fuente.	0%	
Implementación de la funcionalidad para nombrar y seleccionar el archivo destino.	0%	
Diseño de la lógica para generar código MIPS que coincida semánticamente con el código fuente.	0%	
Implementación de la lógica anterior diseñada.	0%	
Diseño de la lógica para verificar la ejecución del código MIPS resultante.	0%	
Implementación de la lógica diseñada.	0%	
Pruebas de la ejecución del código en QtSpim.	0%	
Diseño de la lógica para generar código MIPS a partir de la gramática definida.	0%	
Implementación de la lógica diseñada.	0%	
Diseño de la lógica para la generación de código sin utilizar	0%	

código intermedio (generación directa).		
---	--	--

Motivos de incompletitud de objetivos:

El análisis semántico y la generación de código objetivo son dos de las fases más complejas en el desarrollo de un compilador, si no son las que más, y es que, se debe considerar ampliamente el manejo de cada una de las piezas de código, cómo se relacionarán, como deben funcionar y como pueden programarse para repetir la menor cantidad de código y generar una compilación eficiente y exitosa. En nuestro caso, motivos personales, otras tareas (otros cursos y trabajos) se nos hizo imposible terminar la generación de código objetivo, ya que empezar a utilizar XML para el manejo de etiquetas fue todo un reto, ya que nunca habíamos trabajado con este tipo de lenguaje y además, el pensamiento recursivo para comprender el funcionamiento de las expresiones lógicas, relacionales y aritméticas nos consumió mucho tiempo. Consideramos que la cantidad de tiempo que se tenía para resolver estos problemas de forma eficiente fue muy corta, no obstante, se trabajó muy a profundidad el análisis semántico para garantizar un correcto funcionamiento en un probable generador de código objetivo futuro.

Conclusiones y Recomendaciones

1. El analizador semántico está diseñado para garantizar que no se permitirán generaciones de código objetivo erróneas. En otro caso sería bueno implementar excepciones para terminar la ejecución y se detenga la generación de código.
2. Se debe pensar muy bien en el funcionamiento y relación de las diferentes piezas de código. Esto para fomentar la reutilización y que no existan tantas derivaciones al momento de que se ejecute el código, ya que esto puede dificultar el pensamiento recursivo.
3. Un compilador es un proyecto que requiere muchísimo tiempo de pruebas, ya que existen casos infinitos de expresiones y formas de escribir código, por lo que se debe tomar el tiempo para asegurarse de que es seguro y que será capaz de producir un código objetivo correcto.

Bitácora

Repositorio de Github:

<https://github.com/JoshSan14/Xmas-Compiler>