Aplicación del Análisis Sintáctico Descendente Recursivo

Javier Falcón 2016-5265
Natalia Vallejo 2016-5328
Manuel Molina 2016-5468

1 Introducción

En el transcurso de la clase hemos cubierto las primeras dos etapas del proceso de compilación para un lenguaje, habiendo trabajado con el análisis léxico a través de expresiones regulares y autómatas finitos, así como también partes del análisis sintáctico desde distintos tipos de mecanismos. En este compromiso se busca lograr una familiarización con la implementación de un analizador sintáctico hecho a mano, teniendo como meta la construcción de un compilador para el lenguaje de *lógica proposicional*. La gramática inicial propuesta es la siguiente:

1. $Form \to p|q|r \ldots |s|t$
2. $Form \to \neg Form$
3. $Form \to Form \wedge Form$
4. $Form \to Form \vee Form$
5. $Form \to Form \Rightarrow Form$

Es evidente que uno de los retos de este compromiso es buscar una manera de lidiar con el alto nivel de ambigüedad que se presenta en esta gramática. Adicionalmente, entendemos que resolver el problema de la formación del árbol de expresión y la evaluación de todos los valores de verdad de una expresión requerirá de mucha atención.

2 Resolución del problema

Este compromiso está dividido en tres etapas: análisis léxico, análisis sintáctico y la evaluación de una fórmula (anásls semántico). Sin embargo, antes de entrar en cada etapa, tuvimos que hacer los cambios necesarios a la gramática inicial para disminuir su nivel de ambigüedad, con el objetivo de preparar el escenario para la codificación del *Análisis Sintáctico Recursivo*. La gramática final resultó ser la siguiente:

1. $imp \rightarrow dis\ \{op\_imp\ dis\}$
2. $dis \rightarrow conj\ \{op\_dis\ conj\}$
3. $conj \rightarrow neg\ \{op\_conj\ neg\}$
4. $neg \rightarrow op\_neg\ var|var$
5. $var \rightarrow [a-z]$
6. $op\_imp \rightarrow \Rightarrow$
7. $op\_dis \rightarrow\ \vee$
8. $op\_conj \rightarrow\ \wedge$
9. $op\_neg \rightarrow \sim$

Con esta nueva gramática, expresada en notación **EBNF**, evitamos el caso de encontrarnos con un ciclo recursivo infinito que podíamos tener en la forma inicial; de esta manera podemos avanzar a la primera etapa. Cabe destacar que el lenguaje seleccionado para la implementación fue *JAVA*.

2.1 Análisis Léxico

En vista de que el lenguaje es bastante sencillo, esta etapa fue una tarea casi trivial. Utilizando una clase *Lexer*, desarrollamos un método que tomaba la cadena de entrada y separaba cada token para insentarlos en un vector de caracteres. Si se conseguía un caracter no perteneciente al lenguaje, se notifica que la cadena no es válida y se activa un flag de error para impedir la continuación del análisis. Una característica importante de esta clase es que contiene dos tablas de símbolos, implementadas usando un *hash table*. La clase *Lexer* es la encargada de insertar todos los símbolos detectados en el proceso de tokenización. Dichas tablas serán utilizadas más adelante a la hora del análisis semántico.
El porqué de la creación de dos tablas viene dado a la diferencia entre evaluar la expresión con una sola combinación de valores de verdad y evaluar la expresión para **todos** los posibles valores de verdad.

2.2 Análisis Sintáctico

En esta etapa, la implementación no fue tan sencilla como en la anterior. El objetivo de este análisis consiste en confirmar que la cadena de entrada está correctamente formada y, por consiguiente, generar el árbol de expresión que represente dicha cadena. Nuestra clase *Parser* toma el vector de tokens generado por *Lexer* y lo utiliza para verificar su validez.
Tomando como guía el ejemplo del libro, la clase *Parser* contiene un método por cada regla producción de un **no-terminal**, utilizando la recursión para recorrer cada regla, hasta llegar a un **terminal**. Al momento en el que se identifica un

terminal, se hace un *match* para verificar que el token esperado corresponde con el token actual. En caso de que no haya concordancia, se emitirá un error y se terminará el análisis. De igual manera, si en el proceso recursivo no se encuentra ninguna regla que corresponda con el token actual, se emitirá un error de sintaxis. Cada método retorna un subárbol, el cual se va armando a medida que se hace el ascenso hasta el primer método invocado. De esta manera, cuando termine la recursión, el último retorno devolverá el árbol de expresión completamente armado para ser usado posteriormente en la evaluación de la expresión. Para esto, creamos una clase *BinTree* para representar el Árbol Binario de Expresión.

2.3 Análisis Semántico

Finalmente, nuestra última etapa consistió en evaluar el resultado de la expresión, dados los valores de verdad para cada variable proposicional. Es importante destacar que esta última fase tiene dos variantes:

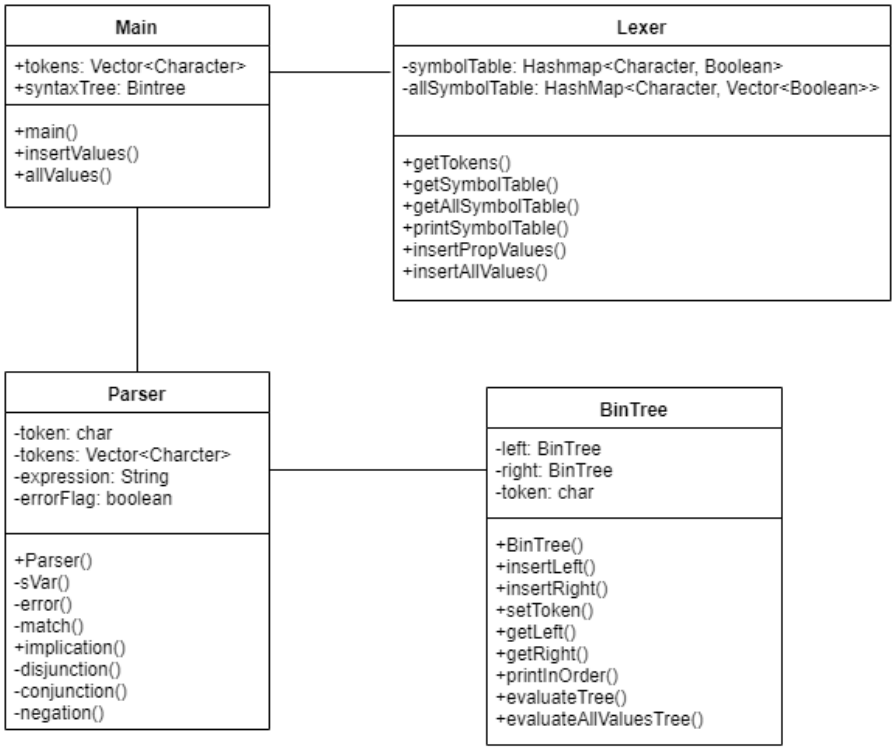
1. Valuación de la expresión dado una combinación de valores de verdad ingresados por consola.

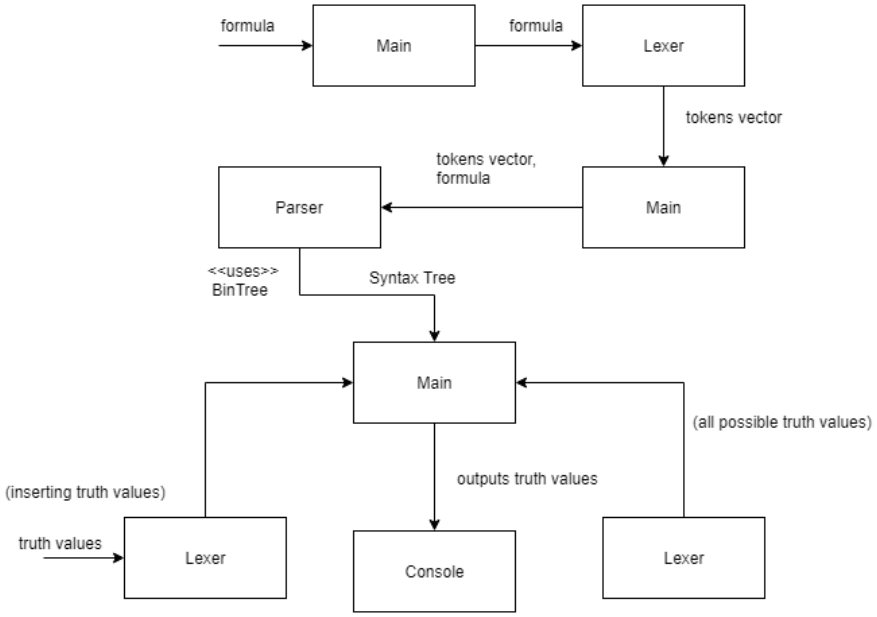
2. Valuación de la expresión con **todos** los valores de verdad posibles.

En ambos casos, el código de evaluación recursivo empleado es el mismo. Lo que varía entre cada valuación es el tipo de la tabla de símbolos. La primera variante guarda un solo valor de verdad por cada variable, debido a que son ingresados por consola. En cambio, en la variante 2, se almacena una lista completa con todos los valores de verdad que tomará cada variable proposicional para poder computar todos los resultados posibles y se ejecutará una evaluación por cada combinación de valores que exista en dicha lista.

La clase *BinTree* contiene los métodos de evaluación. La idea principal consiste en hacer la evaluación recursiva de el subárbol izquierdo y derecho de cada nodo, hasta llegar a una hoja. Cuando esto suceda, se sabe que ese nodo almacena una variable, por lo que se usa su contenido para obtener el valor de verdad en el *hash* y se retorna. Finalmente se hace el chequeo de operadores y, por cada operador, se ejecuta la operación correspondiente utilizando los resultados del subárbol izquierdo y derecho.

3 Diagramas


Main
+tokens: Vector<Character>
+syntaxTree: Bintree
+main()
+insertValues()
+allValues()
Lexer
-symbolTable: Hashmap<Character, Boolean>
-allSymbolTable: HashMap<Character, Vector<Boolean>>
+getTokens()
+getSymbolTable()
+getAllSymbolTable()
+printSymbolTable()
+insertPropValues()
+insertAllValues()
Parser
-token: char
-tokens: Vector<Charcter>
-expression: String
-errorFlag: boolean
+Parser()
-sVar()
-error()
-match()
+implication()
-disjunction()
-conjunction()
-negation()
BinTree
-left: BinTree
-right: BinTree
-token: char
+BinTree()
+insertLeft()
+insertRight()
+setToken()
+getLeft()
+getRight()
+printInOrder()
+evaluateTree()
+evaluateAllValuesTree()


formula
Main
formula
Lexer
tokens vector
tokens vector,
formula
Parser
Main
<<uses>>
BinTree
Syntax Tree
Main
(all possible truth values)
outputs truth values
(inserting truth values)
truth values
Lexer
Console
Lexer