Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3032 - Construcción de Compiladores

Oliver de León
Laura Tamath

Lab #0

Generación de Scanner y Parser

testLexer

Generador léxico generado a partir del “test.g4” el cual divide el código fuente en tokens y expresiones. Esta se hereda de la clase Lexer (que es parte del antlr). Y cuenta con atributos y métodos como:

- *serializedATN*(): método que devuelve en forma de lista el ATN.
- atn: contiene el ATN del analizador léxico.
- *decisionsToDFA*: contiene objetos en una lista del DFA para cada transición del ATN.
- *T__n*, *ID*, *INT*, *WS*: nombres simbólicos de los tokens.
- *channelNames*: lista que contiene los nombres de los canales de tokens.
- modeNames: contiene los nombres de los modos de análisis.
- *literalNames*: contiene representaciones literales de los tokens.
- *symbolicNames*: contiene representaciones simbólicas de los tokens.
- *ruleNames*: lista que contiene el nombre de las reglas de la gramática definidas en el “test.g4”.
- *grammarFileName*: array que representa el archivo de la gramática test.g4

testListener

Escucha eventos específicos del árbol del análisis producido por testParser.

testParser

Analiza la estructura y la sintaxis del código fuente. Esta se hereda de la clase Parser (que es parte del antlr). Y cuenta con atributos y métodos como:

- *serializedATN*():étodo que devuelve en forma de lista el ATN.
- *atn*: contiene el ATN del analizador léxico.
- *decisionsToDFA*: contiene objetos en una lista del DFA para cada transición del ATN.
- sharedContextCache: caché que almacena y reutiliza contextos compartidos durante el análisis sintáctico.
- *literalNames*: contiene representaciones literales de los tokens.
- *symbolicNames*: contiene representaciones simbólicas de los tokens.
- *RULE_prog, RULE_statement, RULE_expression*: nombres de las reglas de la gramática definidas en test.g4
- *grammarFileName*: cadena que representa el nombre del archivo de la gramática.

También posee tres subclases que representan los contextos del análisis para las reglas correspondientes a la gramática.

Main

Abre el archivo en modo lectura y crea un objeto InputStream (que proporciona los caracteres para el análisis) a partir del contenido del archivo. Se crea un objeto testLexer pasando el InputStream como entrada ya que este realiza el análisis léxico y produce tokens.

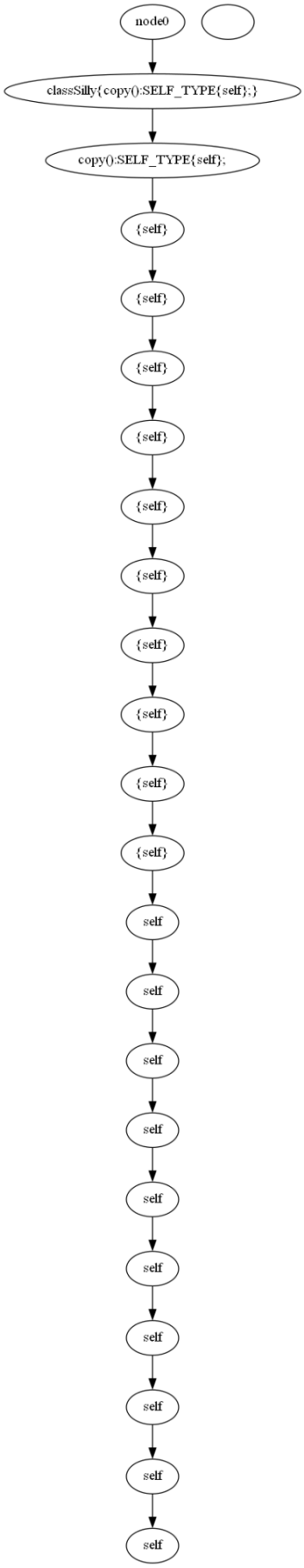
Después, se crea un objeto CommonTokenStream pasando el objeto testLexer como entrada, para almacenar los tokens generados. Y se llama al método prog para analizar la gramática definida. Se crea un objeto TransformDot, que es una clase derivada de ParseTreeVisitor, para visitar el árbol de análisis sintáctico y generar el gráfico en formato DOT, y se convierte a un formato PNG para poder visualizar gráficamente el árbol para representar en análisis léxico.

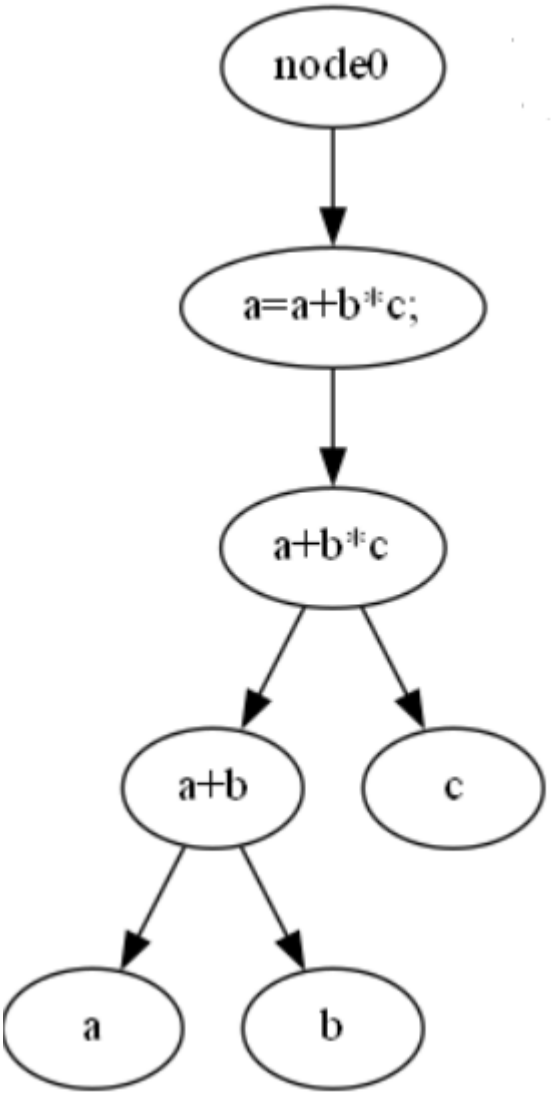
test.g4

- La regla **prog** define un programa que consiste en una o más clases.
- La regla **class** define la estructura de una clase, que tiene un nombre de tipo (TYPE) y puede heredar de otra clase opcionalmente. También puede contener declaraciones (statement) dentro de sus llaves.
- La regla **statement** define diferentes tipos de declaraciones, como declaraciones de métodos, declaraciones de variables y asignaciones.
- Las reglas **expr**, **assignExpr**, **condExpr**, **orExpr**, **andExpr**, **relExpr**, **addExpr**, **multExpr**, **unaryExpr** y **atomExpr** definen diferentes tipos de expresiones y cómo se pueden combinar entre sí.
- Las reglas **letBindingList**, **letBinding** y **exprList** definen la estructura de listas de declaraciones de variables y listas de expresiones, respectivamente.
- Las reglas **ID**, **TYPE**, **INTEGER**, **STRING** y **WS** definen los tokens o símbolos terminales que se utilizan en la gramática, como identificadores, tipos, números enteros, cadenas de texto y espacios en blanco.

test.yalp

Este código se obtuvo del PDF “especificacion_yapl.pdf” para comprobar si funcionaba correctamente el programa. Este básicamente define tres clases: "Silly", "Sally" y "Main". La clase "Silly" tiene un método copy() que devuelve una referencia al objeto actual. La clase "Sally" hereda de "Silly" sin agregar nada adicional. La clase "Main" tiene una variable x de tipo "Sally" que se inicializa con una llamada al método copy() en un nuevo objeto de "Sally". Además, la clase "Main" tiene un método main() que devuelve la variable x.


node0
classSilly{copy():SELF_TYPE{self};}
copy():SELF_TYPE{self};
{self}
{self}
{self}
{self}
{self}
{self}
{self}
{self}
{self}
{self}
self
self
self
self
self
self
self
self
self
self


node0
a=a+b*c;
a+b*c
a+b
c
a
b