Universidad Católica del Norte
Departamento de Ingeniería de Sistemas y Computación

Compiladores

Proyecto de Compiladores, Segunda entrega

Autor(es):
Alan Calderon
Rodrigo Oyarzun
Profesor Responsable:
Brian Keith Norambuena

07 Diciembre 2018

1. Resumen

En la presente entrega se detallara el proceso para construir un analizador sintáctico el cual corresponde a la segunda fase de análisis de un compilador. El proyecto consiste en desarrollar un analizador sintáctico en PYTHON y utilizando la librería PLY. En este informe se detallara todo lo teórico y lo práctico del objetivo de este taller. La importancia de este taller es lograr comprender a profundidad cada etapa de análisis que hace un compilador, para este caso comprender cómo funciona un analizador sintáctico. Como meta se propone poder analizar 6 tipos de archivos distintos, el cual contendrá líneas de códigos relacionadas a la gramática de libre de contexto propuesta para esta segunda etapa del taller para finalmente generar un código el cual a través de una aplicación web la pueda interpretar e imprimir el árbol sintáctico de ese archivo. Además de poder implementar este taller en un ambiente eficiente y ordenado, separando cada funcionalidad importante en un archivo y clase distinta. Dentro del formato pedido en este informe, primero se hablará de los conceptos teóricos relacionados a la implementación de un analizador sintáctico, se explicara en qué consiste el método "LALR(1)", método el cual se utilizará para analizar sintácticamente los archivos de ejemplo. Luego se hablará en detalle el proceso de desarrollo de este taller, como también las dificultades de esta implementación. Finalmente se mostrará los resultados obtenidos mostrando el código resultante como también su árbol sintáctico relacionado, para luego hacer un análisis de los resultados obtenidos.

2. Introducción

Siguiendo el objetivo del curso de desarrollar un compilador, en el taller anterior se pasó por la primera fase del proceso de compilación, la cual corresponde a la fase de análisis léxico (scanner), la cual se identificaron los tokens a través de expresiones regulares. Estos tokens sirven para una posterior fase del proceso para la construcción de un compilador, siendo entrada para la fase del analizador sintáctico (parser).

2.1. Análisis sintáctico

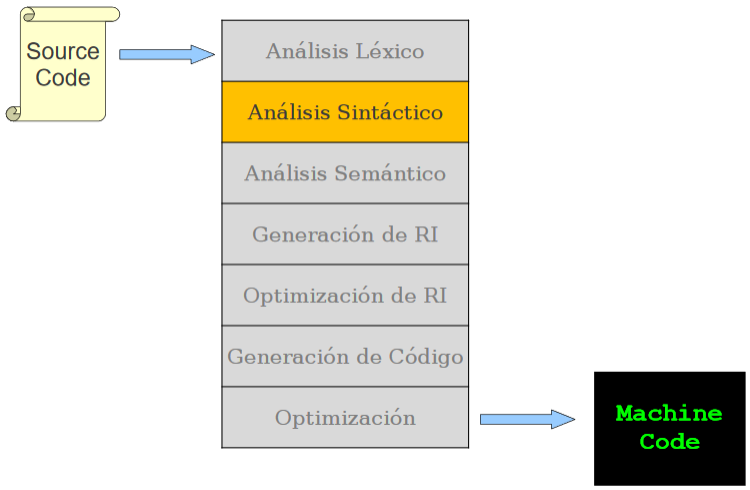

Figura 1: Etapas Compilador

Luego de pasar por la fase de Análisis Léxico (scanner) y haber generado los tokens, la fase del Análisis Sintáctico (parsing) busca interpretar qué significan estos tokens, teniendo como objetivo recuperar la estructura descrita por esa serie de tokens. En el contexto de poder generar un lenguaje de programación, los tokens no son suficiente para poder definir uno, es por eso que hiciste otros tipos de formalidades para poder definirlos, entre ellas existen las gramáticas libres de contexto(GLC). Esta fase de análisis convierte estos tokens de entrada a un árbol sintáctico de derivación siguiendo la estructura definida por una GLC. Lo cual lo hacen más útiles para un posterior análisis en la siguiente fase de análisis semántico.

2.2. Objetivo del taller

Siguiendo el contexto del marco teórico, el objetivo del taller es poder crear un analizador sintáctico a través de los tokens ya generados en el anterior taller. Se seguirá programando en python y se usará la librería PLY para poder leer las GLC y determinar si la secuencias de los tokens capturados satisface la especificación de la estructura del lenguaje.

Dentro de las distintas formas en que el curso se aprendió a poder hacer parsing a una entrada de tokens, se encuentra el método LALR(1), este método es el cual se utilizara para poder hacer el análisis sintáctico del taller.

2.3. LALR(1)

El método LALR es otro método de análisis sintáctico ascendente, es el más complejo y también el más completo y más eficiente de todos.

LALR significa LookaHead LR, y suele usarse esta técnica generalmente en la práctica, es un técnica intermedia entre SLR(1) y LR(K), la cual es mucho más potente que SLR(1) Y más sencilla que LR(1).

Básicamente lo que se hace es la unión de los conjuntos de elementos que tengan elementos comunes en sus goto(Ui,alpha), y que únicamente varía la parte del lookahead de cada conjunto.

Por ejemplo si tenemos dos conjuntos Ui Y Uj:

Figura 2: Conjunto a y b

Podemos hacer la unión de sus lookahead, creando un conjunto Uij, Así:

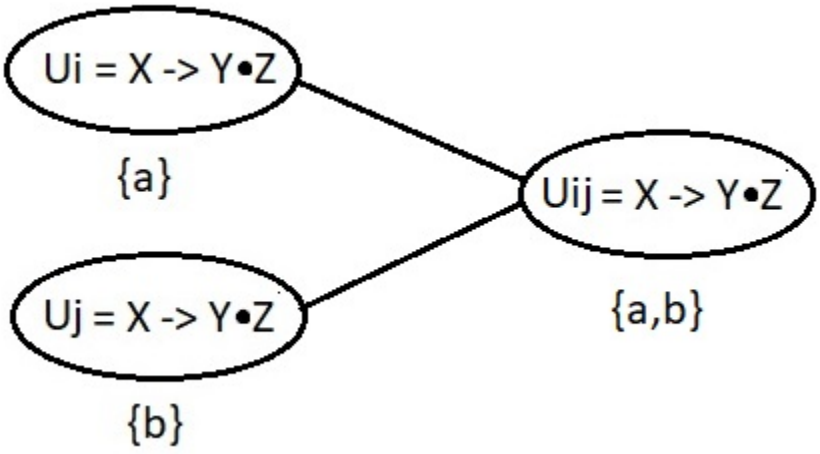

Figura 3: Union Conjunto a y b

3. Trabajo realizado

En esta seccion se detalla la implementacion del Analizador Sintactico

3.1. Preambulo

Para el funcionamiento de este taller es necesario importar algunos paquetes necesario previamente instalados de la libreria Ply en Python.

```
import ply.yacc as yacc
```

3.2. Desarrollo

Para el taller se utilizó como base el ejemplo subido de parser subido por el profesor, en este taller de ejemplo, se identificaron 3 archivos importantes para el desarrollo de un analizador sintáctico (parser).

Parser.py: Es el archivo principal del proyecto, es donde se ejecutará el programa, aquí es donde se importan la librería ply y las clases a utilizar. Su función consiste en instanciar todas las reglas de la gramática libre de contexto, además de abrir el archivo para realizar el parse y dividir las líneas del archivo en palabra por palabra según la GLC haciéndolas más simple para ser analizadas. Además de poder captar los respectivos errores de parser que pueden ocurrir y devolver un mensaje de error.

Nodos.py: Es la clase donde se crean los nodos, luego de que se haya hecho el parser desde el archivo parser.py llamara a la clase nodo para guardar las palabras divididas según la gramática a un nodo, donde cada palabra estará guardará en una posición. La clase parser, enviará estos atributos y la clase nodo la recibirá y las guarda para luego enviarlas como un nodo compacto a la clase visitor.

DibujarASTvisitor.py: Este archivo almacena la clase Visitor, este archivo es igual de importante que el archivo parser.py debido a que esta clase se encargará de generar el código para luego generar el árbol sintáctico. Esta clase utiliza el patrón de diseño "Visitor", el cual recorrerá todos los nodos generados y a través de la información podrá saber qué nodos son padres y cuáles son hijos, para luego escribir en un archivo .dot una sintaxis en el lenguaje "Graphviz" para ser interpretada y posteriormente generar un árbol sintáctico.

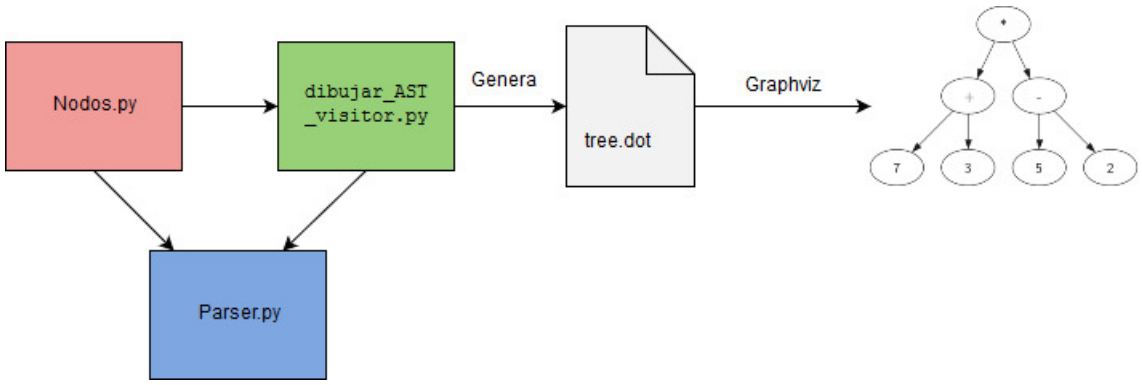

Figura 4: Proceso de Desarrollo

En general el proceso de desarrollo es conciso, primero hay que definir las reglas gramaticales en el archivo principal, para luego enviarlas a los nodos correspondiente, donde estos serán visitados a través del patrón visitor, donde en el archivo donde está definido el patrón visitor se deben declarar el compartimiento del recorrido de este patrón y según el recorrido generar el código para luego ser guardado en un archivo llamado "tree.dot"para luego este código a través de una aplicación online de "Graphviz"generar un árbol sintáctico.

Entre las dificultades de este desarrollo, fue definir el comportamiento del patrón visitor y de qué manera escribir el archivo la estructura adecuada para luego poder dibujar el árbol sintáctico. Mientras la profundidad del árbol fuese más alta, se dificulta mucho más el asociar los nodos padres a sus nodos hijos. Haciendo que generará un árbol sintáctico donde los nodos hijos estaban separados de los nodos padres.

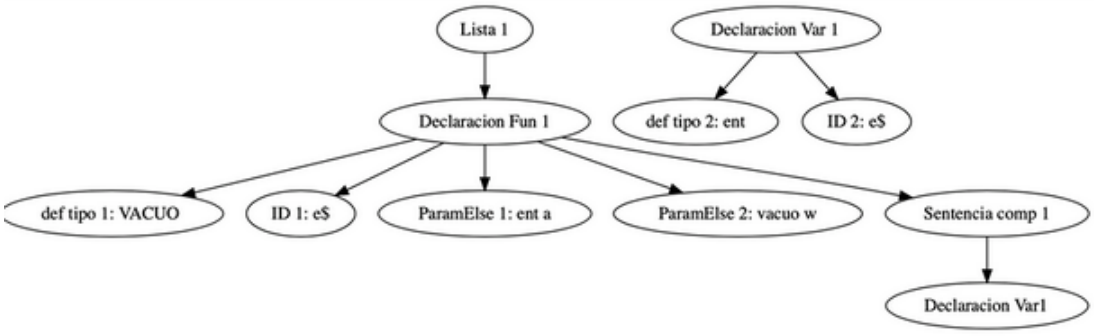

Figura 5: Error mas comun en el proceso de desarrollo

4. Resultados

El objetivo al completar el desarrollo del analizador sintáctico, es analizar 6 archivos diferentes donde tendrán varias líneas de código asociadas al diagrama libre de contexto y generar un código para luego poder dibujar su árbol sintáctico a traves de la aplicacion web webgraphviz. Se mostrarn 3 ejemplos, para mostrar los resultados mas relevantes y concisos para luego comparar si lo resultados obtenidos son los esperados teoricamente.

Segundo Archivo (Sample2.pp) - Resultados

```
VACUo ab$cc;
VACUO ab$aa <123>;
VACUO ab [ent f , ent k] ()
```

Figura 6: Ejemplo Propuesto - Archivo 2

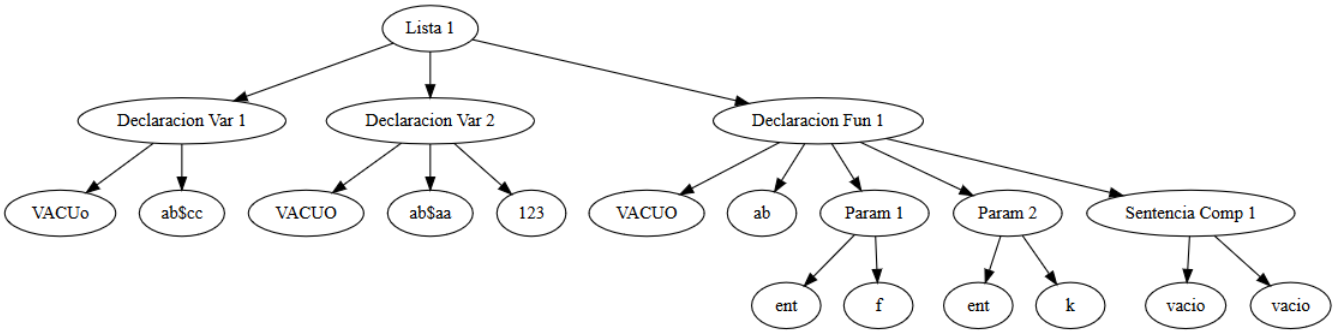

Figura 7: Árbol Sintáctico - Archivo 2

Tercer Archivo (Sample3.pp) - Resultados

```
VACUO e$ [ ent a , vacuo w ] (
  ent e$;
  SI [ a$=e$ ](
    aa$ = a;
  ) SINO (
    aq$ = wr;
  )
)
```

Figura 8: Ejemplo Propuesto - Archivo 3

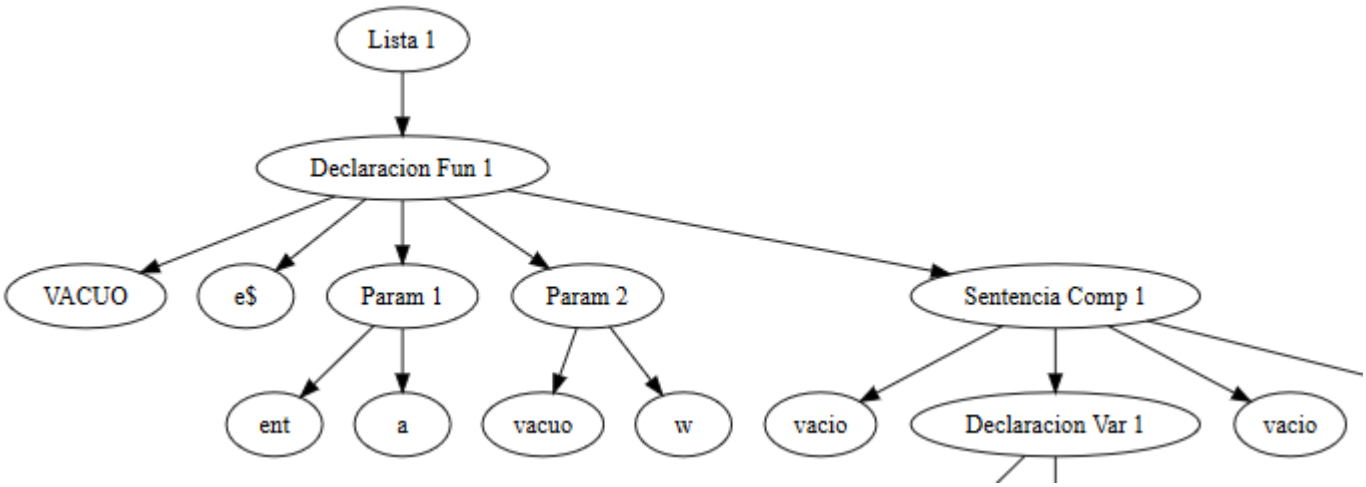

Figura 9: Árbol Sintáctico - Archivo 3 - parte 1

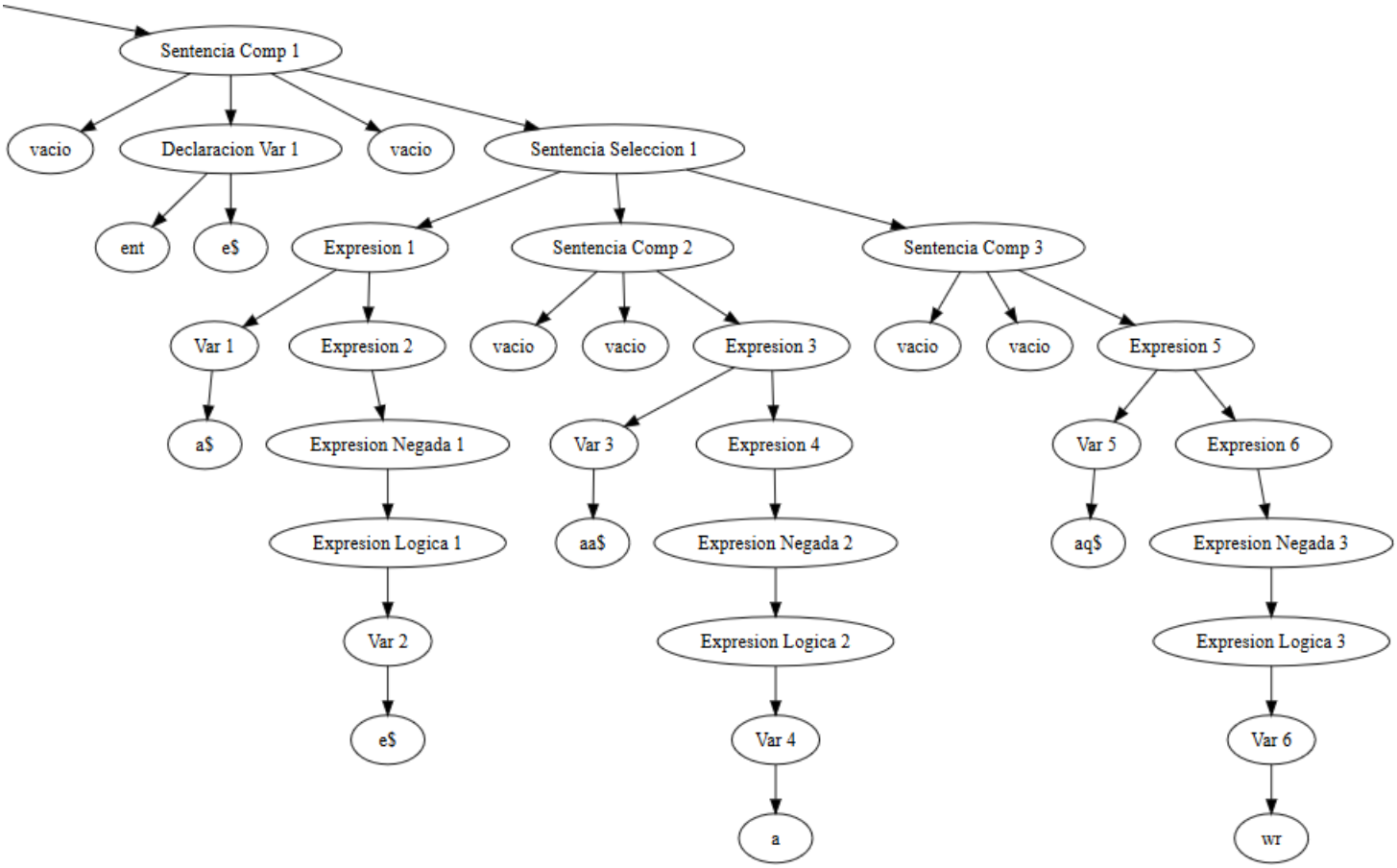

Figura 10: Árbol Sintáctico - Archivo 3 - parte 2

Sexto Archivo (Sample6.pp) - Resultados

```
VACUO e$ [ ent a , vacuo w ] (

  RET b = bc;

  RET ;

  RET cc < as = dfs > = dd;

  RET bb = ! [ aa ];

)
```

Figura 11: Ejemplo Propuesto - Archivo 6

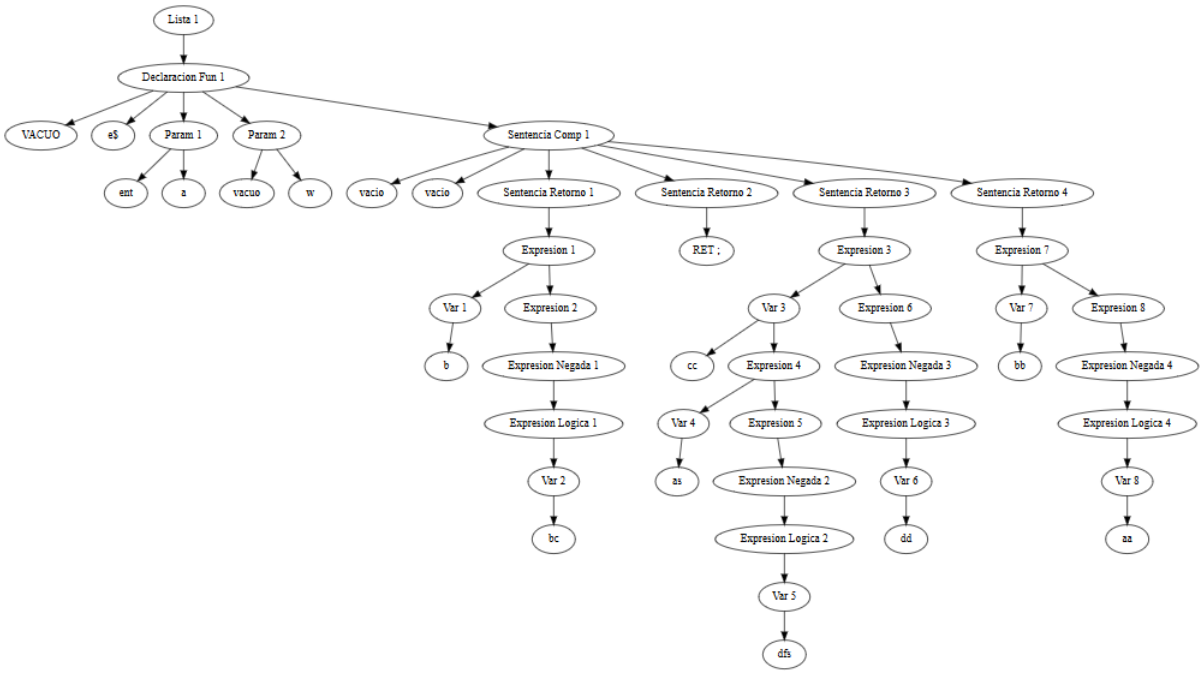

Figura 12: Árbol Sintáctico - Archivo 6

5. Discusión de los resultados

Luego de concluir las distintas pruebas, como se puede observar en las figuras anteriores, los resultados fueron exitosos, ya que para cada ejemplo se logró dibujar su árbol sintáctico de manera correcta y respetando sus propiedades. En general para la mayoría de las pruebas se usaron códigos de un lenguaje de programación asociada a la gramática de libre de contexto. Cada ejemplo tenía una dificultad mayor a la anterior, es por esto que para cada número de ejemplo, mientras más alto sea su número, es porque en ese ejemplo se utilizó líneas de códigos mucho más complejas que el anterior, y el cual esto conlleva que utilizaba gran parte de la gramática, el cual se pudo corroborar si realmente la implementación del código pasaba por todas estas reglas gramáticas y las identificaba correctamente.

```
1   VACUo ab$cc;
2   VACUO ab$aa <123>;
3   VACUO ab [ent f , ent k] ()
```

```
1   VACUO e$ [ ent a , vacuo w ] (
2
3     RET b = bc;
4
5     RET ;
6
7     RET cc < as = dfs > = dd;
8
9     RET bb = ! [ aa ];
10
11  )
```

Figura 13: Comparacion entre el primer ejemplo y el ultimo ejemplo

En general, si comparamos estos resultados obtenidos con los resultados teóricos esperados, se puede concluir que satisfacen completamente los resultados esperados, ya que los árboles sintácticos no tienen errores de ambigüedad y tampoco tienen una mala correlación entre cada nodo padre y nodo hijo. Es decir el árbol sintáctico se logró crear tal cual como podríamos haber nosotros dibujado el árbol sin el programa de parser y a través de nuestros conocimientos adquiridos en el ramo.

Finalmente, repasando los requisitos del taller, se cumplio con cada punto el cual el enunciado exigía respecto a los resultados de este taller, los cuales son:

- implementar el código separado por cada componente, es decir, que el parser estuviera separado de la creación de los nodos y del patrón visitor. Es decir cada una de estas implementaciones se hicieron en archivos y clases distintas.
- Poder analizar sintacticamente 6 ejemplos de código relacionado a la gramática libre de contexto dada para el taller.
- Escribir en un archivo .dot, el árbol en código graphviz para la aplicación web “webgraphviz” y dibujar su árbol sintáctico para cada uno de los ejemplos.
- A través del terminal, poder elegir que tipo de ejemplo ejecutar para su análisis sintáctico

6. Conclusiones

Como conclusión, lo visto en clases fue una parte importante del desarrollo de este segundo taller, tanto como las clases teórica como la clase de práctica, gracias a esto, fue menos complicada su implementación. Si bien el taller tenía una mayor complejidad que el anterior, se logró cumplir con gran parte de los requerimientos para poder dar unos resultados acorde a lo que se pide. Para esta etapa se vieron elementos nuevos en el contexto de programación, como fue el patrón de diseño "Visitor" y la librería ply, que es capaz de interpretar las gramáticas libre de contexto. Además se pudo comprender la importancia de cada etapa en el proceso de compilación de un lenguaje, ya que sin lo desarrollado anteriormente, no hubiéramos podido tener tokens de entrada para que el parser pudiera darle una estructura a estos y gracias a la creación del parser, se puede proceder a la siguiente etapa de la compilación de un lenguaje de programación, que es el análisis semántico.

7. Bibliografía

Referencias

[1] Keith N., Brian, *Apunte Compiladores.* UCN, 2018.