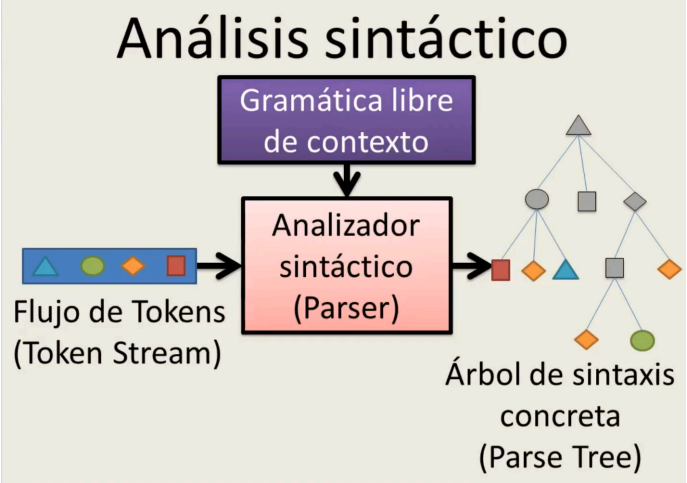
ANÁLISIS SINTÁCTICO

Toma el flujo de tokens(gramatica libre de contecxto), y genera el ParseTree.



Gramática libre de contexto (archivos Grammar.g4).
Hay símbolos terminales y símbolos NO terminales.

Los símbolos en minúsculas serán NO terminales, por ejemplo:

```
program: PROGRAM ID BRA_OPEN
 sentence*
 BRA_CLOSE;
```

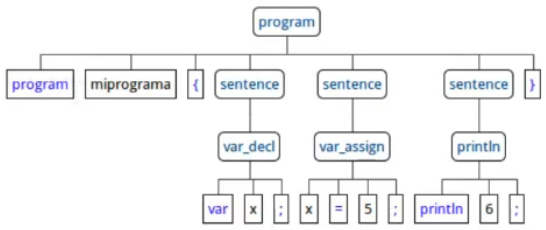
Los símbolos en mayúsculas serán terminales, por ejemplo:

```
PROGRAM:'program';
VAR:'var';
PRINTLN:'println';
```

Ejemplo:

```
program miprograma {
        var x;
        x = 5;
        println 6;
}
```

Parse tree:



ANÁLISIS SEMÁNTICO

Toma el ParseTree generado anteriormente y genera el árbol de sintaxis abstracta, que tiene la información, el significado del programa que queremos ejecutar, osea el orden de ejecución de los símbolos, sentencias, etc.

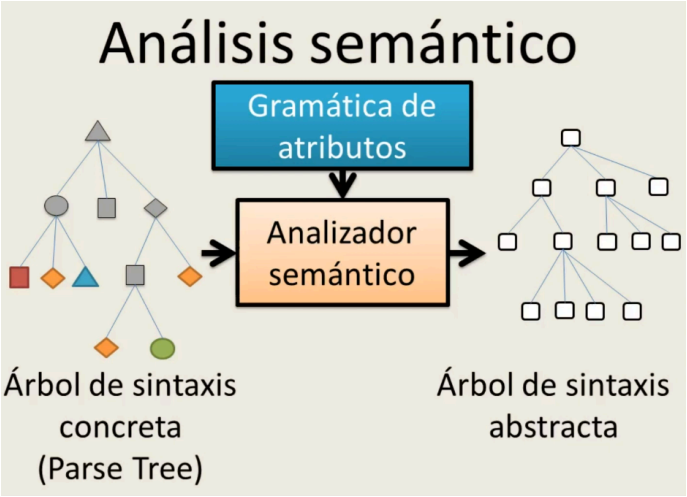
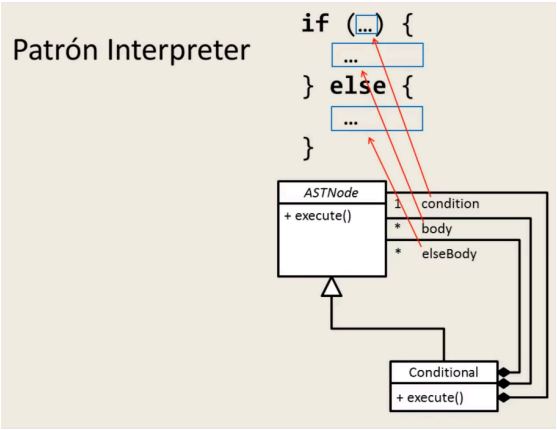


TABLA DE SÍMBOLOS

Relaciona las variables con su valor. Las guarda en una tabla (HashMap) donde tienen su index (nombre de variable) y su valor.

PATRÓN INTERPRETER (LO USE EN EL IF/ELSE)

Almacena los comandos/resultados/ejecuciones en los nodos y los ejecuta cuando es necesario.

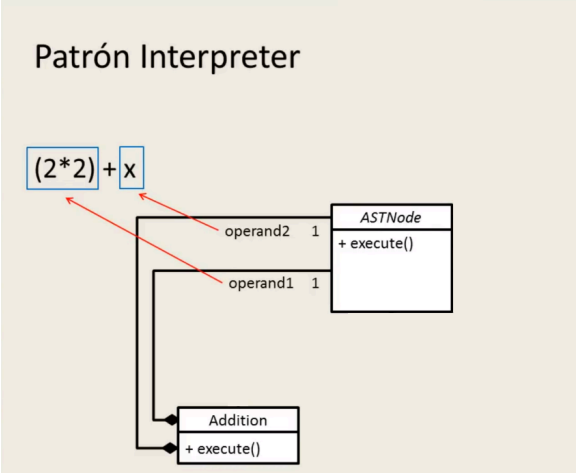


Reconoce lo siguiente:

- CONDICION
- BODY
- ELSE-BODY

Y lo guarda en su ASTNode (Abstract Sintax Tree Node).

Debemos crear una subclase para cada comando, por ejemplo la adición, siempre se recibe dos operandos, y luego ambos deben apuntar a la clase ASTNode para “resolver la adición”. Esto permite que cada subclase permite almacenar subclases, en el ejemplo al ser una suma atómica (nro + nro), primero debe resolver la multiplicación, y luego recién pasar a la suma.



Creo el paquete “ast” donde creo la interfaz ASTNode, y de ahí creo todas las clases, por ejemplo Addition, Multiplication, etc.

Una vez realizado el proyecto uno, llamado “interprete”, crearemos el proyecto “interprete-ProyectoFinal” y realizaremos las modificaciones siguientes para abordar el enunciado final:

1. **Análisis Léxico:** Dividir el código fuente en tokens.
2. **Análisis Sintáctico:** Verificar la estructura del código fuente y construir un árbol de sintaxis abstracta (AST).
3. **Análisis Semántico:** Verificar el balance de llaves, corchetes y paréntesis, y la correcta utilización de variables y funciones.
4. **Generación de Código Intermedio:** Generar la versión en código de tres direcciones del código fuente.
5. **Optimización:** Realizar optimizaciones como propagación de constantes y eliminación de operaciones repetidas.