Compiladores
Laboratorio 1 y 2
Interprete y Compilador de Calculadora[1]

Objetivo
Desarrollar un intérprete y compilador para el lenguaje P[2] en una calculadora.

Implementación
El script implementa las siguientes clases:

- **Token**: Almacena los elementos correspondientes a las categorías léxicas.
- **Scanner**: Realiza el análisis lexicográfico.
- **Parser**: Lleva a cabo el análisis sintáctico.
- **Exp, BinaryExp y NumberExp**: Representan el árbol sintáctico abstracto (AST) de expresiones aritméticas.

La clase abstracta Exp define dos métodos virtuales:

- **void print()**: Imprime la estructura del AST.
- **int interprete()**: Evalúa la expresión aritmética.

La función main crea instancias de Scanner y Parser mediante constructores. Adema, se llama a parser.parse(), que devuelve un objeto Exp* representando el AST de la expresión y se invocan los métodos print() y eval() para mostrar y evaluar la expresión.

Calculadora
Definida por la gramática

- Exp = Term ((+|-|*|/) Term)*
- Term = Factor
- Factor = Num

Ejecución

```
g++ -o calc lab1.cpp
/calc "3 - 2 + 10"
```

Ejercicio 1
Completar el intérprete.

- Implementar int Exp::interprete() para que retorne el resultado de evaluar la expresión aritmética representada en el AST.
- Para ello, se realizará un recorrido del AST similar al método print(), pero en lugar de imprimir la estructura, se ejecutarán las operaciones aritméticas correspondientes.

[1]Autoría del Profesor Igor Siveroni
[2]https://homepages.cwi.nl/~steven/pascal/book/10pcode.html

Ejercicio 2
Incorporar las operaciones de multiplicación (*) y división (/)

- Gramática actualizada:

    - Exp = Term ((+|-|*|/) Term)*
    - Term = Factor
    - Factor = Num

- La inclusión de estas operaciones requiere modificaciones en el Scanner, Parser y en los métodos de la clase Exp.

- Utilizar la función test_scanner(&scanner) para verificar los tokens reconocidos.

Ejercicio 3
Modificar la gramática para garantizar la correcta precedencia de operadores, asegurando que la multiplicación y división tengan mayor prioridad que la suma y resta:

- Exp = Term ((+|-) Term)*
- Term = Factor ((*|/) Factor)*
- Factor = Num

Ejercicio 4
Agregar soporte para paréntesis en la gramática para permitir expresiones anidadas y controlar la precedencia de operadores.

- Exp = Term ((+|-) Term)*
- Term = Factor ((*|/) Factor)*
- Factor = Num | “(“ Exp “)”

Ejercicio 5
Agregue la impresión del AST en notación polaca inversa.

- La Notación Polaca Inversa (RPN, por sus siglas en inglés) es una forma de representar expresiones matemáticas en la que los operadores aparecen después de sus operandos, eliminando la necesidad de paréntesis para definir la precedencia de las operaciones. Por ejemplo, para la expresión **(2+3)*5** su notación polaca inversa es: **2 3 + 5 *.**

- En la clase abstracta Exp, se debe definir el siguiente método virtual:

- void rpn(): Imprime la expresión en notación polaca inversa (RPN).

- Este método recorrerá el AST siguiendo un orden postorden, asegurando que los operandos se impriman antes que sus operadores.

Ejercicio 6
Desarrollar un compilador para el lenguaje P.

- En el lenguaje P se tiene las siguientes sentencias:

    - LDCc: carga el entero en la pila
    - ADI: suma los dos valores en la cima de la pila
    - MPI: multiplica los dos valores en la cima de la pila
    - SBI: suma los dos valores en la cima de la pila
    - DVI: División de enteros

- Por ejemplo: Si deseamos realizar la operación **2+(3*5)** se debe generar el código:

    - LDCc 2
    - LDCc 3
    - LDCc 5
    - MPI
    - ADI

- En la clase abstracta Exp defina el método virtual:

    void compilador, imprime el código intermedio P.