Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 3: Intermediate Code Generation

Preet Kanwal

Department of Computer Science & Engineering

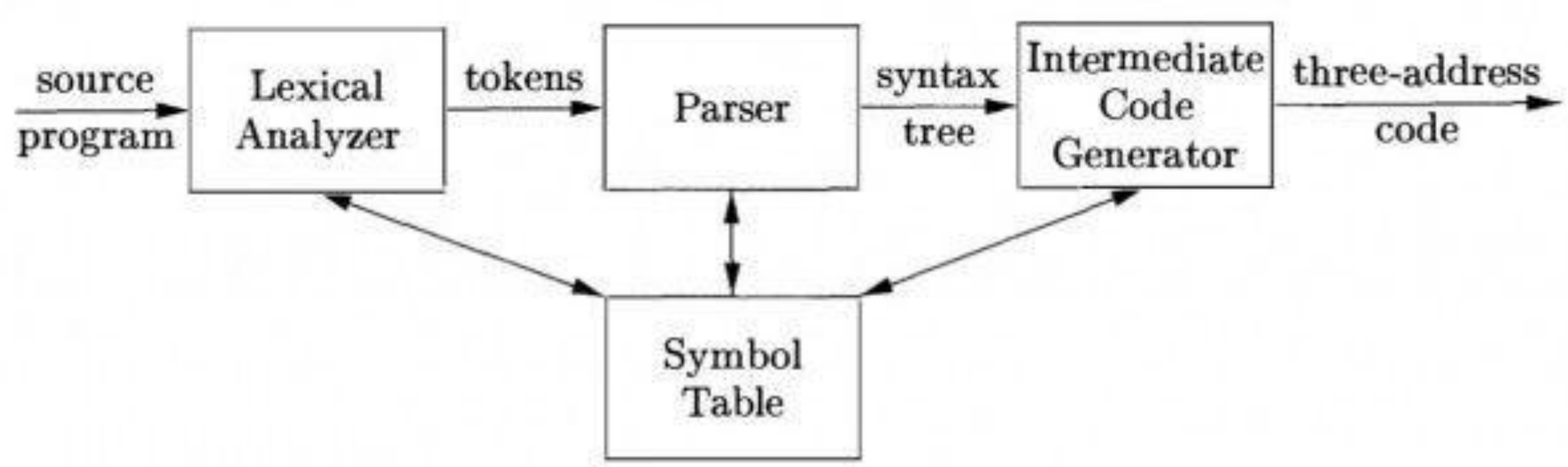
Lecture Overview

In this lecture, you will learn about -

- What is intermediate code?
- Why intermediate code generation?
- Advantages of ICG
- Types of Intermediate Representation
- Directed Acyclic Graph
  - Applications
  - SDD to construct a DAG
  - Examples of Syntax tree vs DAG

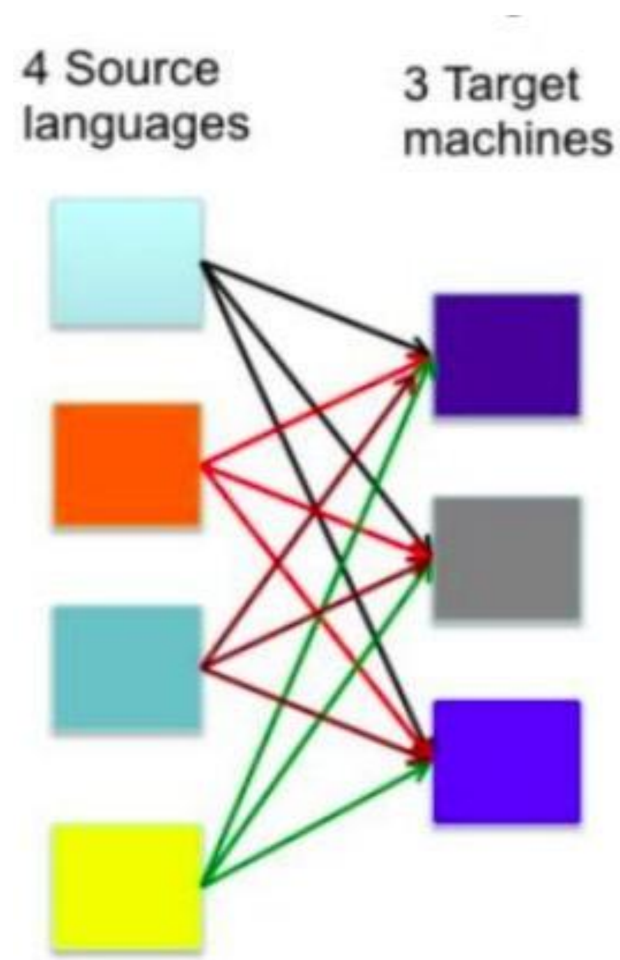
What is Intermediate code?

- Intermediate code is used to translate the source code into the machine code.
- It lies between the high-level language and the machine language.
- The Intermediate code generator receives input from the semantic analyzer. It takes input in the form of an annotated syntax tree.
- Using the intermediate code, the second phase of the compiler (synthesis phase) is changed according to the target machine.

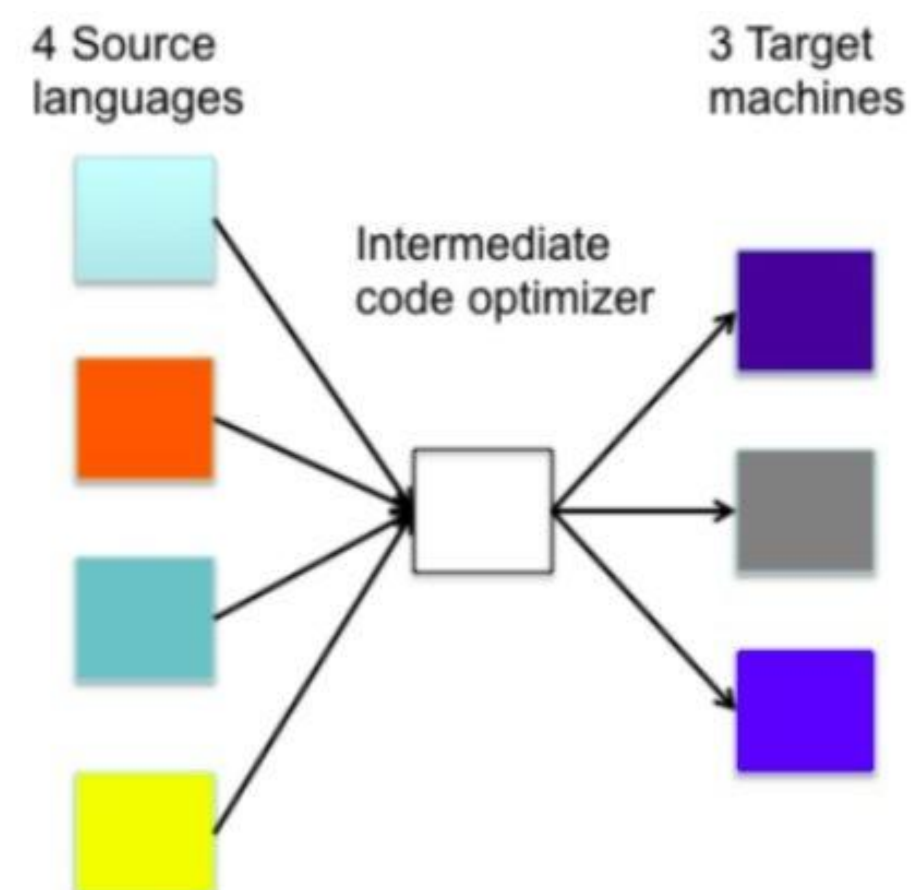


Why Intermediate code generation?

- **Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.**



4 front-ends + 4x3 optimisers + 4x3 code generators



4 front-ends + 1 optimiser + 3 code

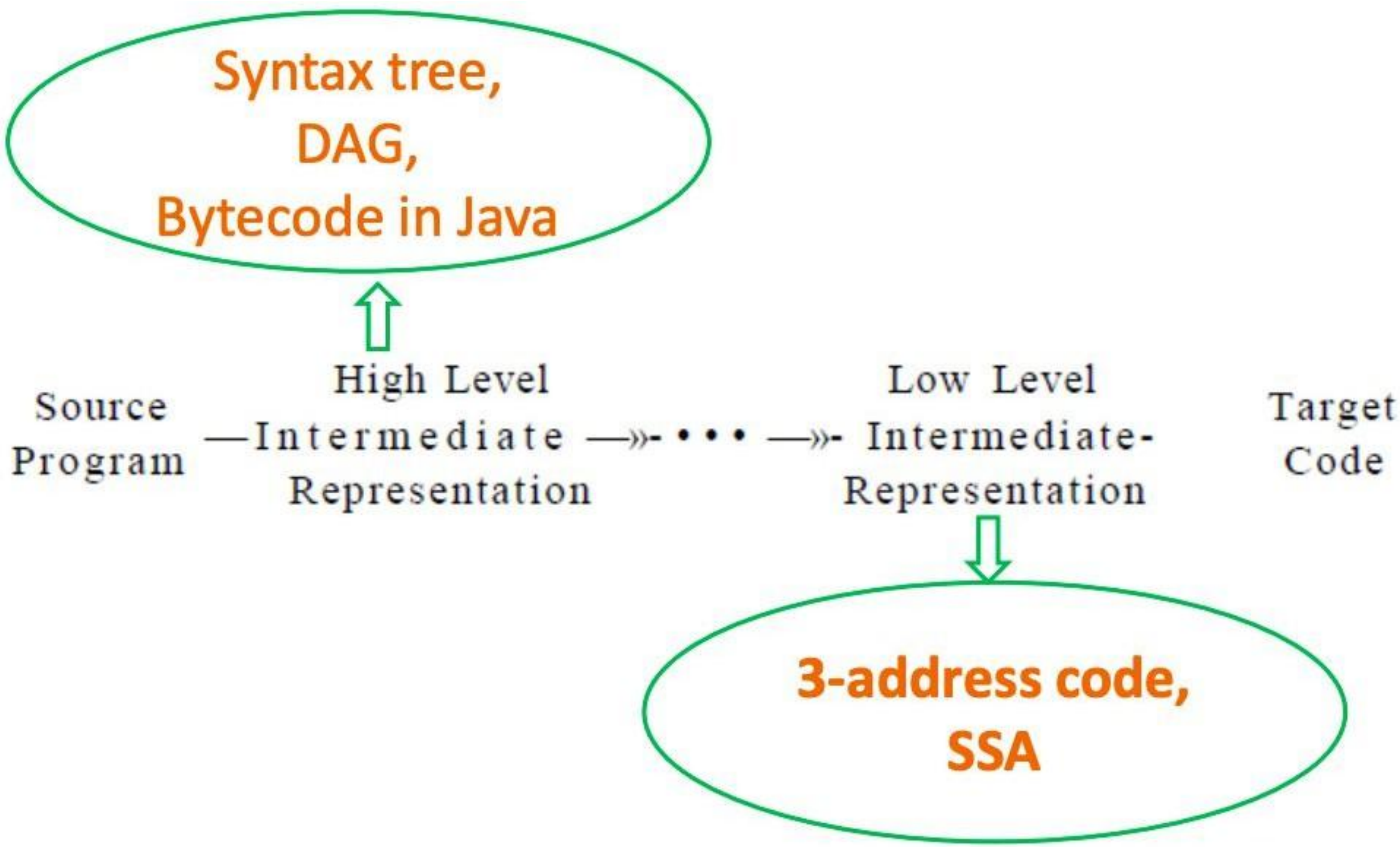
Advantages of Intermediate Code Generation

- ICG makes it easier to construct compilers for different architectures.
- Targetcode can be generated for any machine just by attaching new machine as the back end - this is called retargeting.
- It is possible to apply machine independent code optimization - helps in faster generation of code.

Types of Intermediate Representation

- An intermediate representation is a representation of a program between the source and target languages.
- A good IR is one that is fairly independent of the source and target languages - this maximizes its ability to be used in a retargetable compiler.
- There are three ways to classify Intermediate representation:
  - High-level or Low-level
  - Language-specific or Language independent
  - Graphical or Linear

Intermediate Representation - High level vs Low level representation



Intermediate Representation - High level representation

- High-level intermediate code representation is very close to the source language itself.
- They can be easily generated from the source code
- Code modifications can be easily applied to enhance performance.
- Examples- Syntax trees, DAG, Java Bytecode

Intermediate Representation - Low level representation

- Low level intermediate code representation is close to the target machine.
- This makes it suitable for register and memory allocation, instruction set selection, etc.
- It is good for machine-dependent optimizations.
- Examples - Three Address Code, SSA

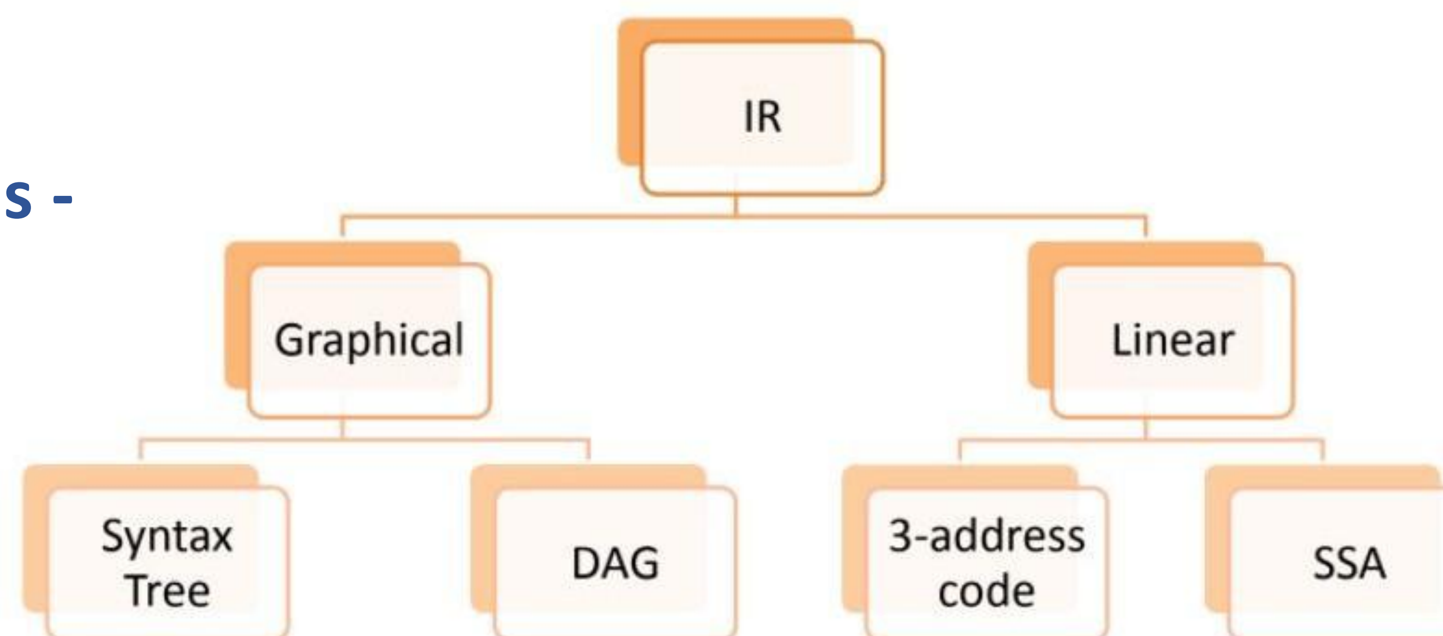
Intermediate Representation

In terms of language, Intermediate code can be either -

- **Language specific** - Byte Code for Java, P-code for Pascal
- **Language independent** - three-address-code

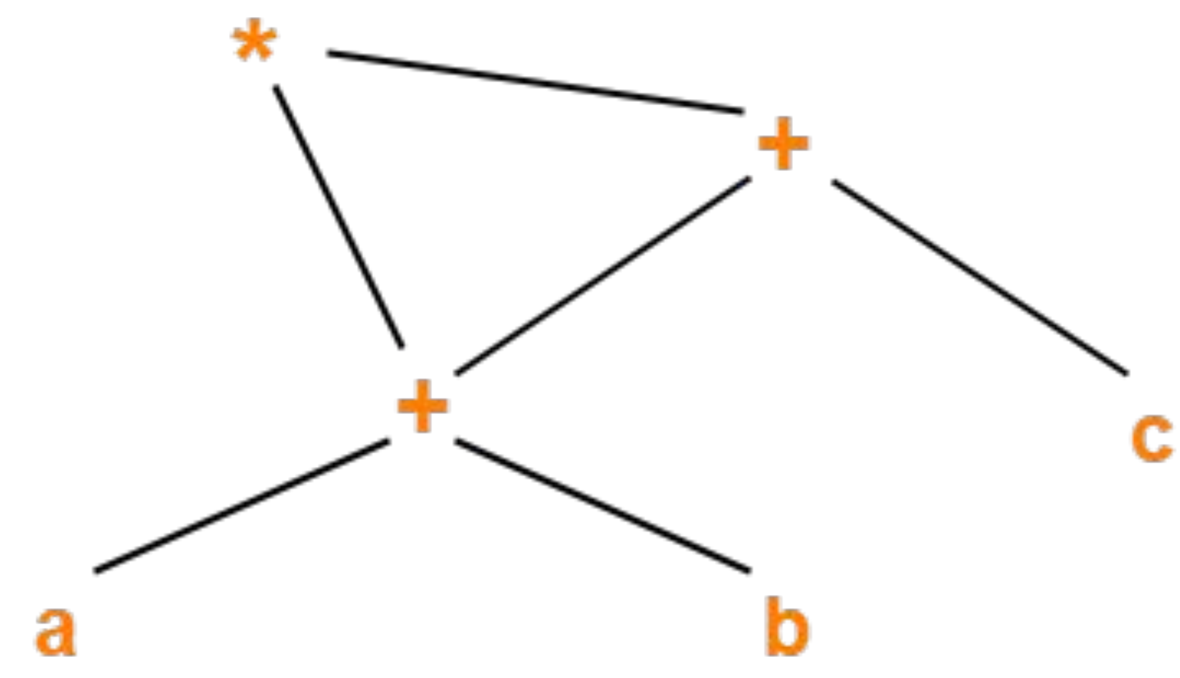
Intermediate code can be also classified as -

- Graphical
- Linear



DAG - Directed Acyclic Graph

- It is a variant of Syntax tree with a unique node for each value.
- It does not contain cycles.
- In a DAG,
  - Interior nodes always represent the operators.
  - Exterior nodes (leaf nodes) always represent the names, identifiers or constants.
- The given figure represents the DAG for the expression (a + b) x (a + b + c)



Directed Acyclic Graph

Applications of DAG

- It helps optimize code by identifying common subexpressions in a syntax tree.
- It reduces no. of calculations to be done - calculate once, refer anywhere.
- It can be used to determine the names whose computation has been done outside the block but used inside the block.
- It can also be used to determine the statements of the block whose computed value can be made available outside the block.

SDD to construct a DAG

- SDD used to generate Syntax tree will be used to construct DAG too, with a simple check -

  if an identical node exists

  RETURN existing node

  else

  CREATE a new node

- The assignment instructions of the form x:=y are not performed unless they are necessary.
- The process of making this check is an overhead; hence constructing DAG is costly.

Exercise 1 - Construct DAG for the given expression

Consider the following unambiguous grammar -

E -> E + T | E – T | T

T -> T * F | T / F | F

F -> (E) | [E] | id

Using this, construct Syntax tree and DAG for the following expression -

((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

Exercise 1 - Solution

Given - ((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

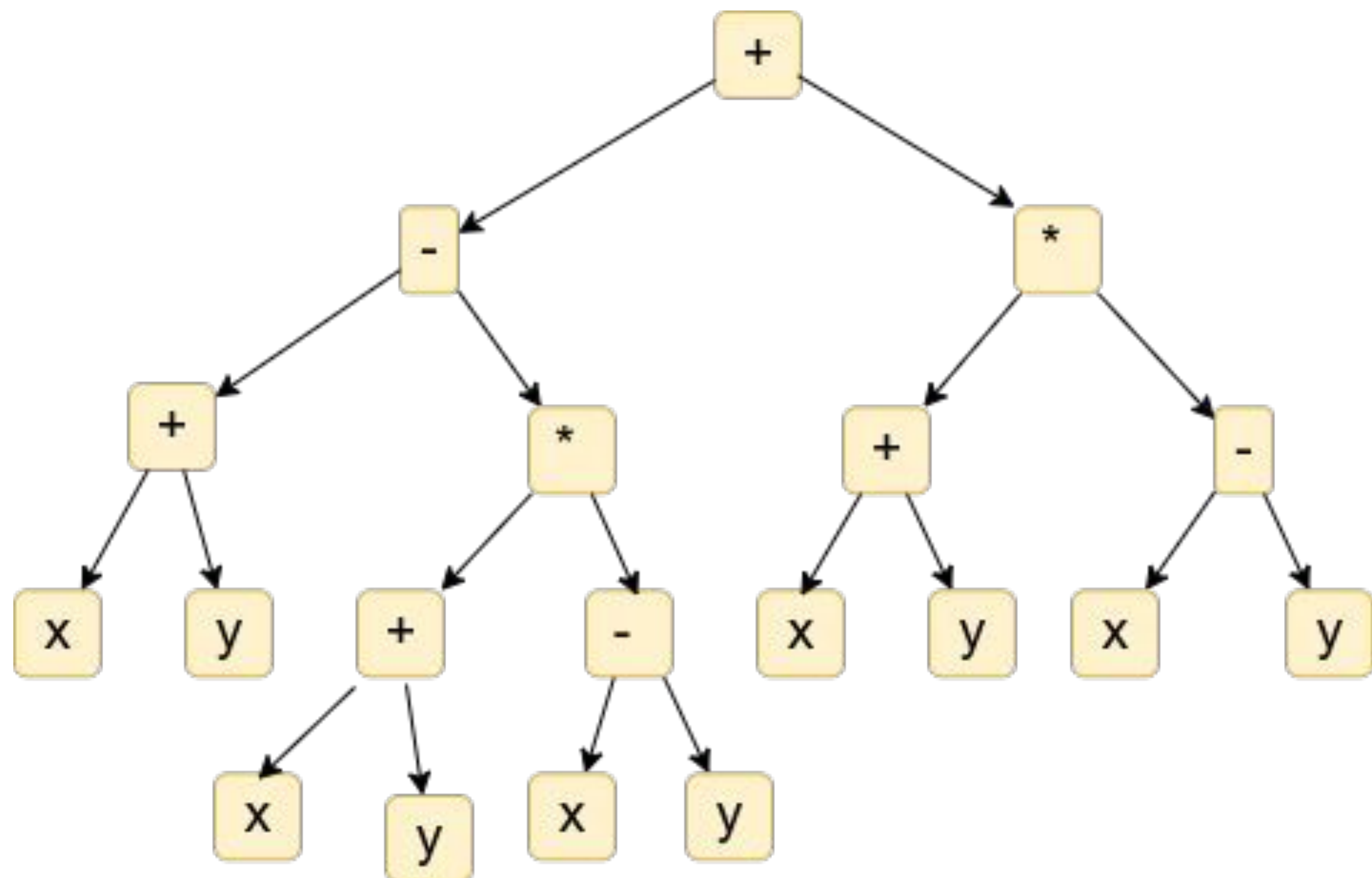
Step 1 - Rewrite the expression for clear understanding

```
(
    (x + y) - (
                 (x + y) * (x - y)
              )
)
+
(
    (x + y) * (x - y)
)
```

Exercise 1 - Solution

Given - ((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

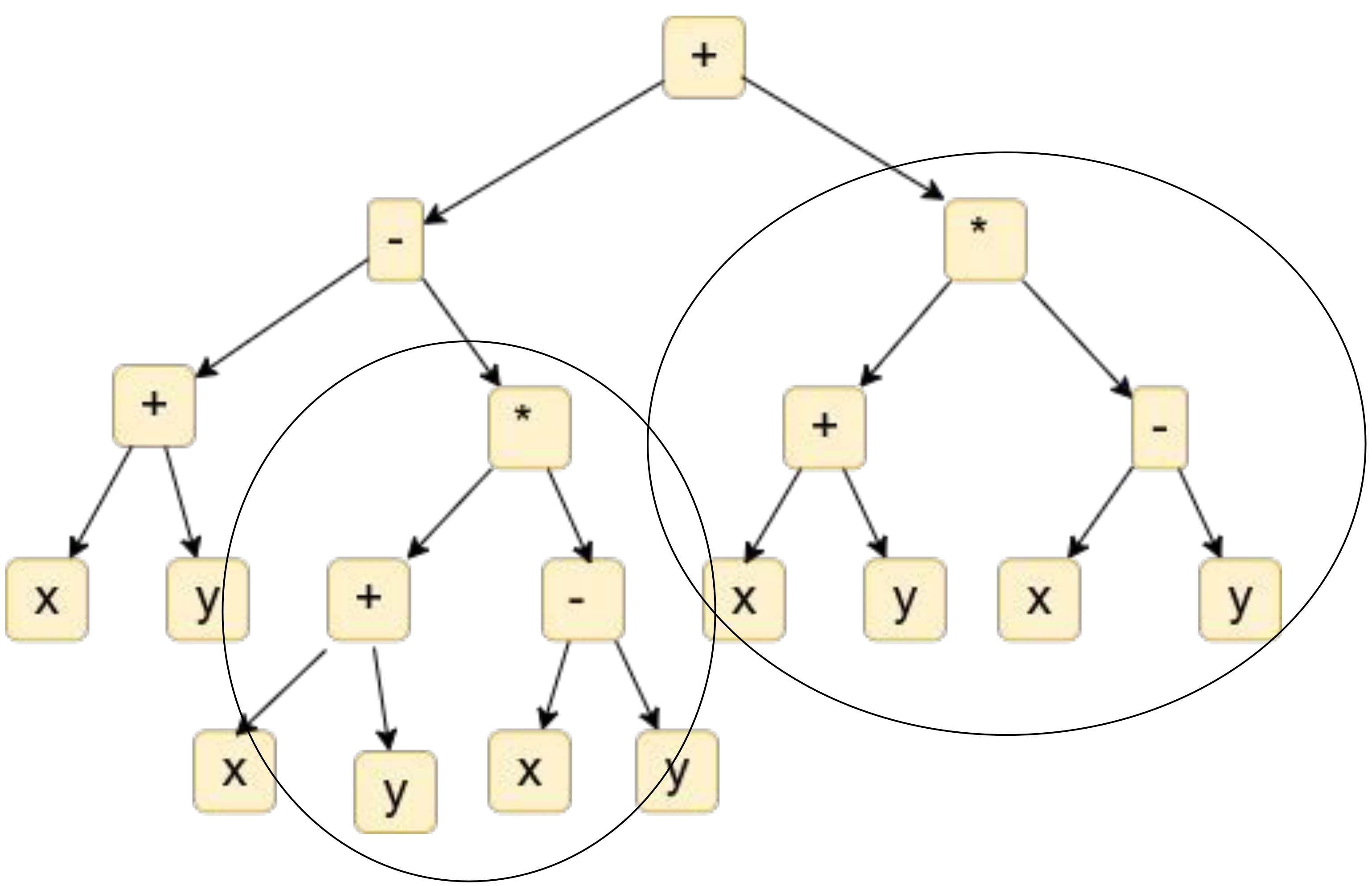
Step 2 - Draw the Syntax tree



Exercise 1 - Solution

Given - ((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

Step 3 - Identify the common subexpressions and eliminate step wise



Exercise 2

Consider the following unambiguous grammar -

E -> E + T | E – T | T

T -> T * F | T / F | F

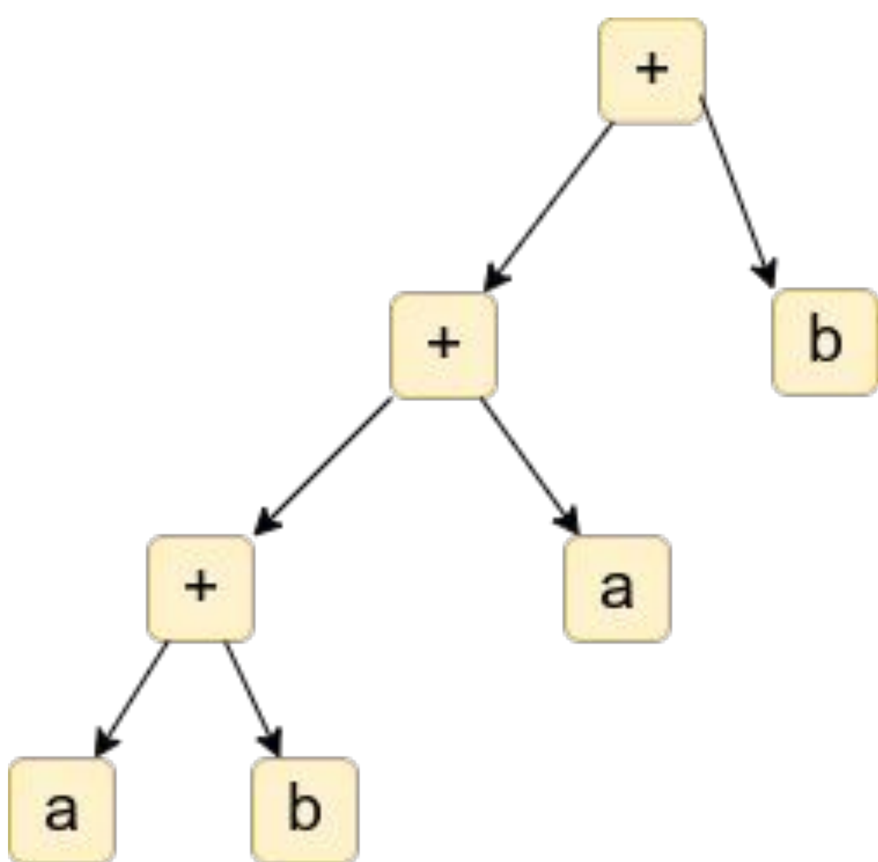
F -> (E) | [E] | id

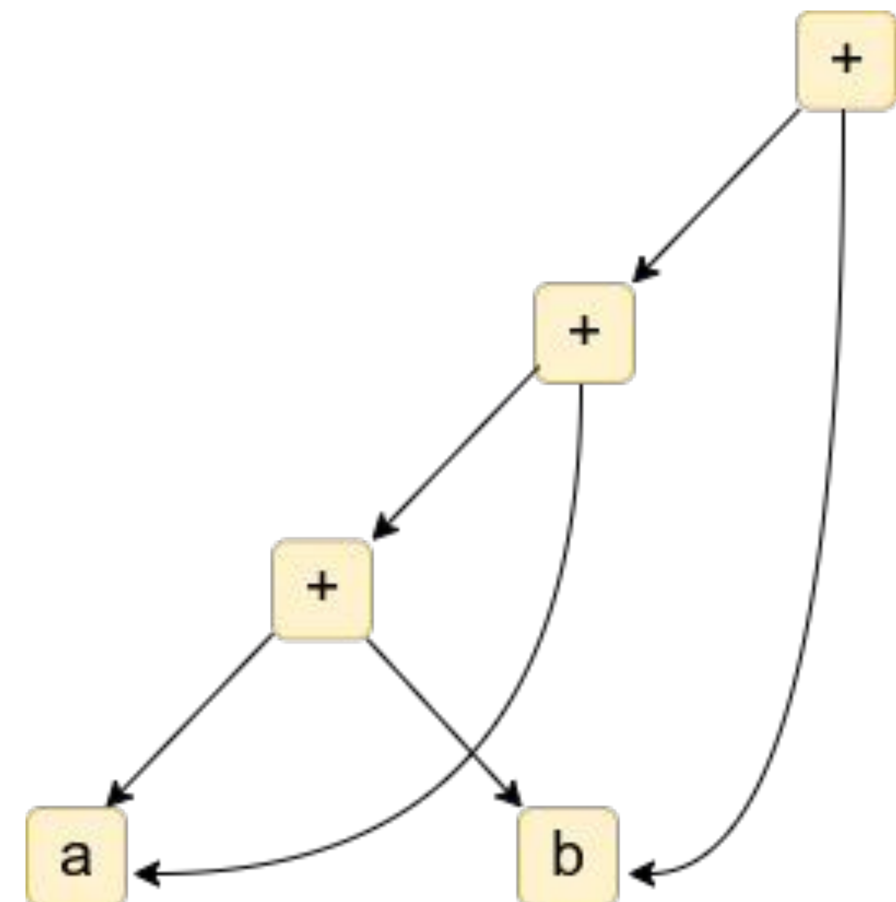
Using this, construct Syntax tree and DAG for the following expressions -

1) a + b + a + b
2) a + b + (a + b)
3) a + a * (b – c) + (b – c) * d
4) (((a + a) + (a + a)) + ((a + a) + (a + a)))
5) [(a + b) * c + ((a + b) + e) * (e + f)] * [(a + b) * c]

Exercise 2 - Solutions

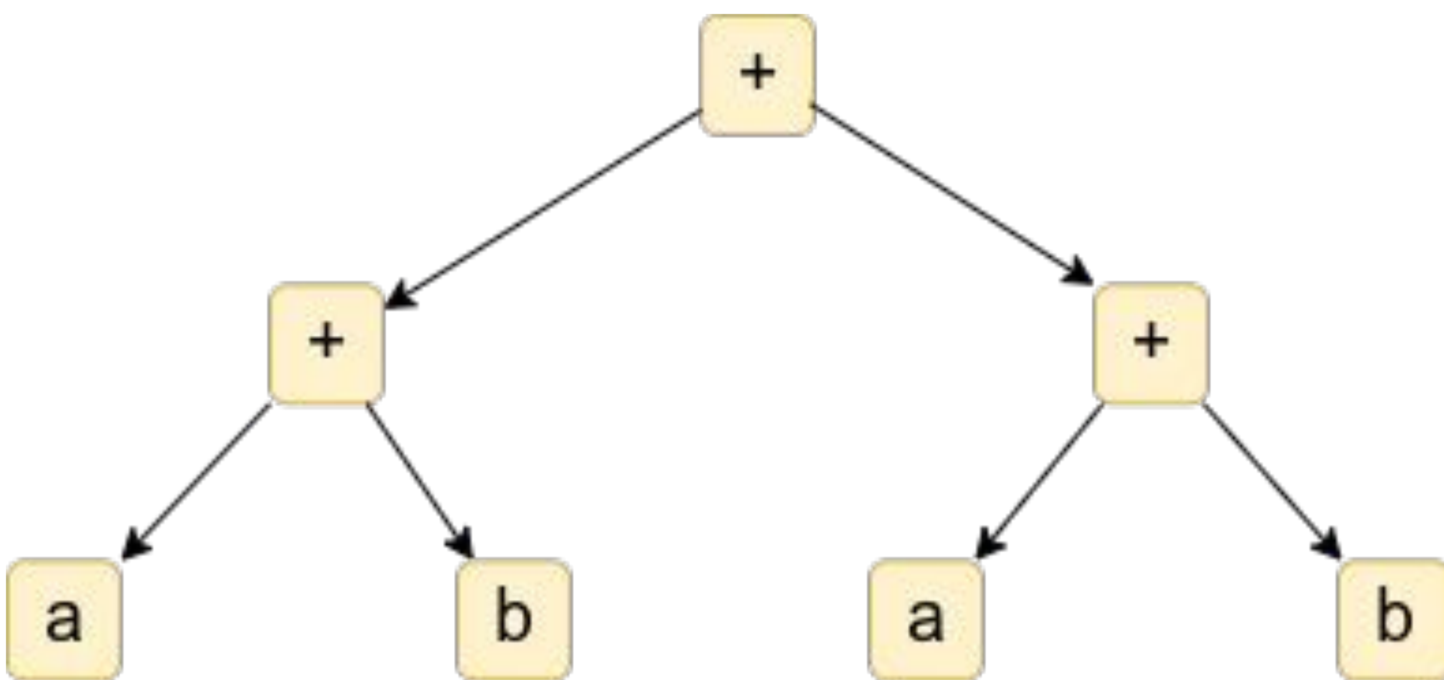
1) a + b + a + b

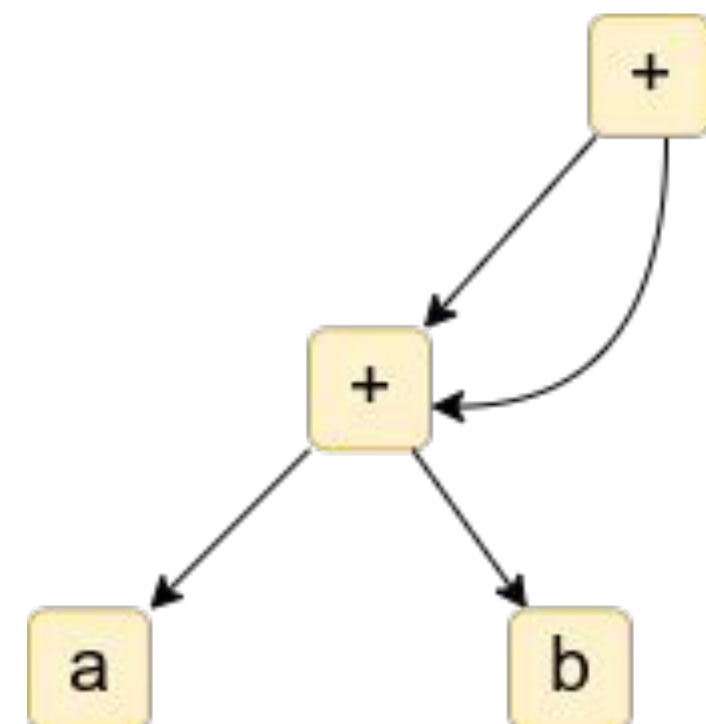




Exercise 2 - Solutions

2) a + b + (a + b)

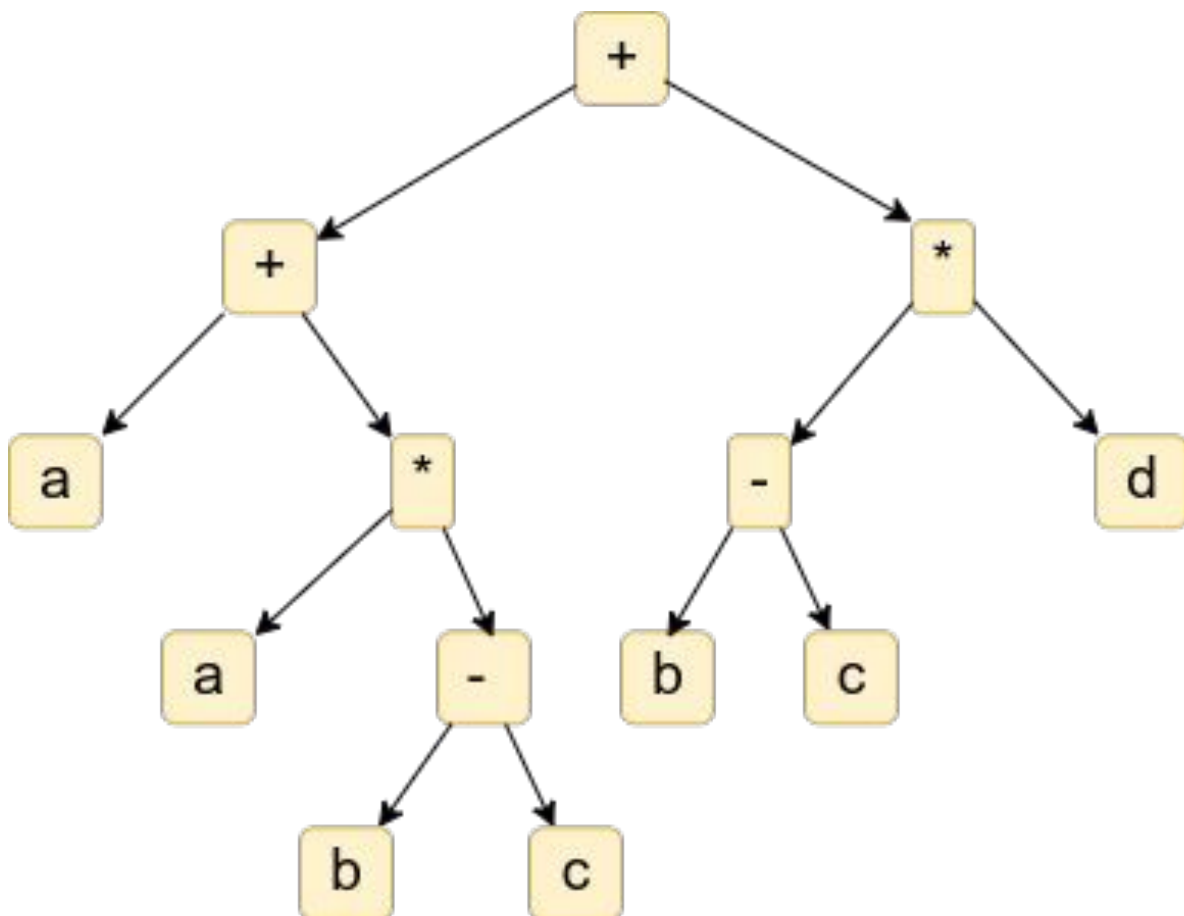


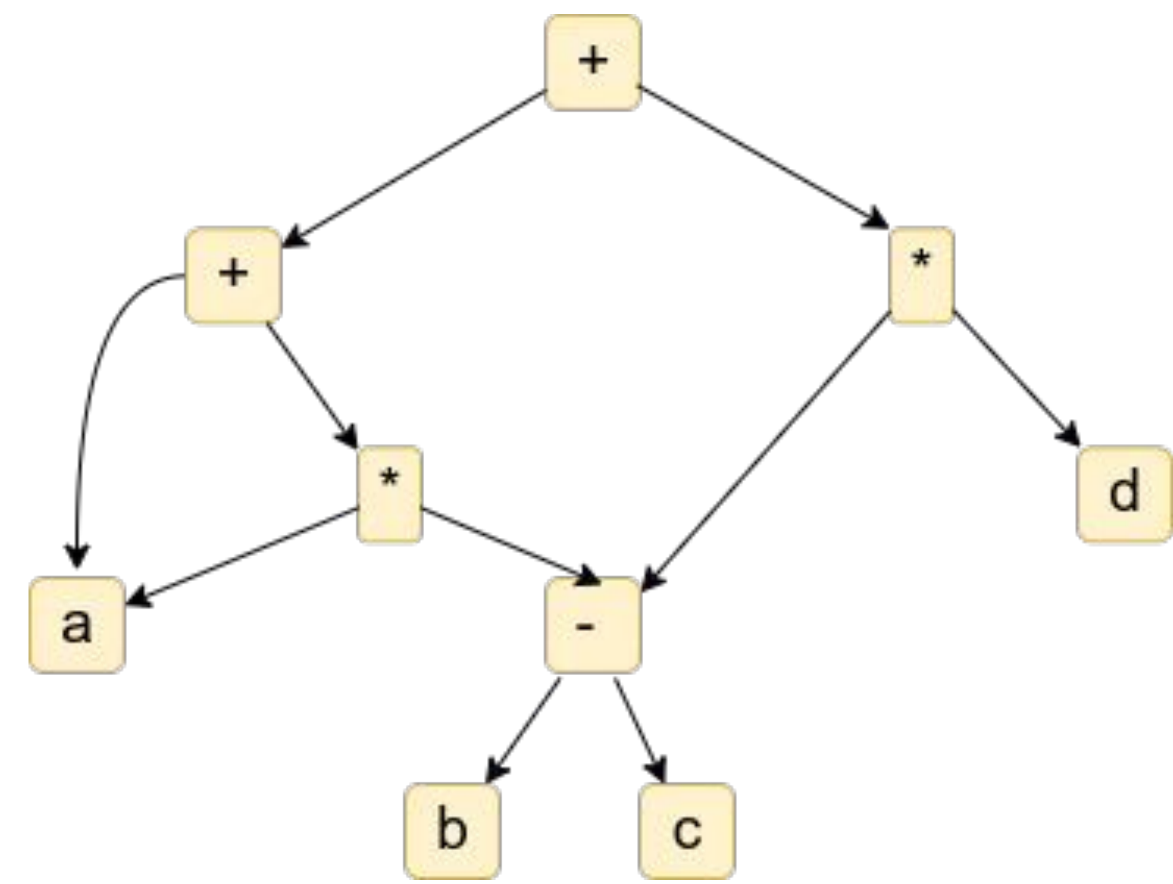


Compiler Design

Exercise 2 - Solutions

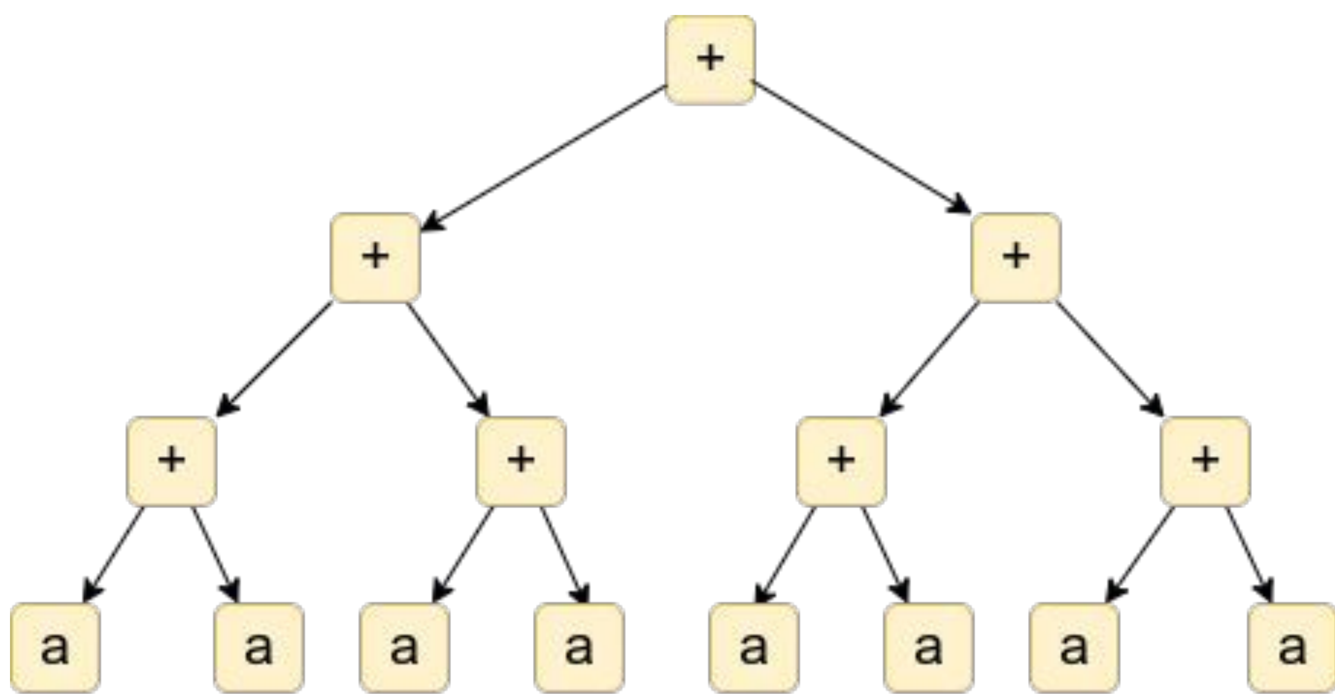
3) a + a * (b – c) + (b – c) * d

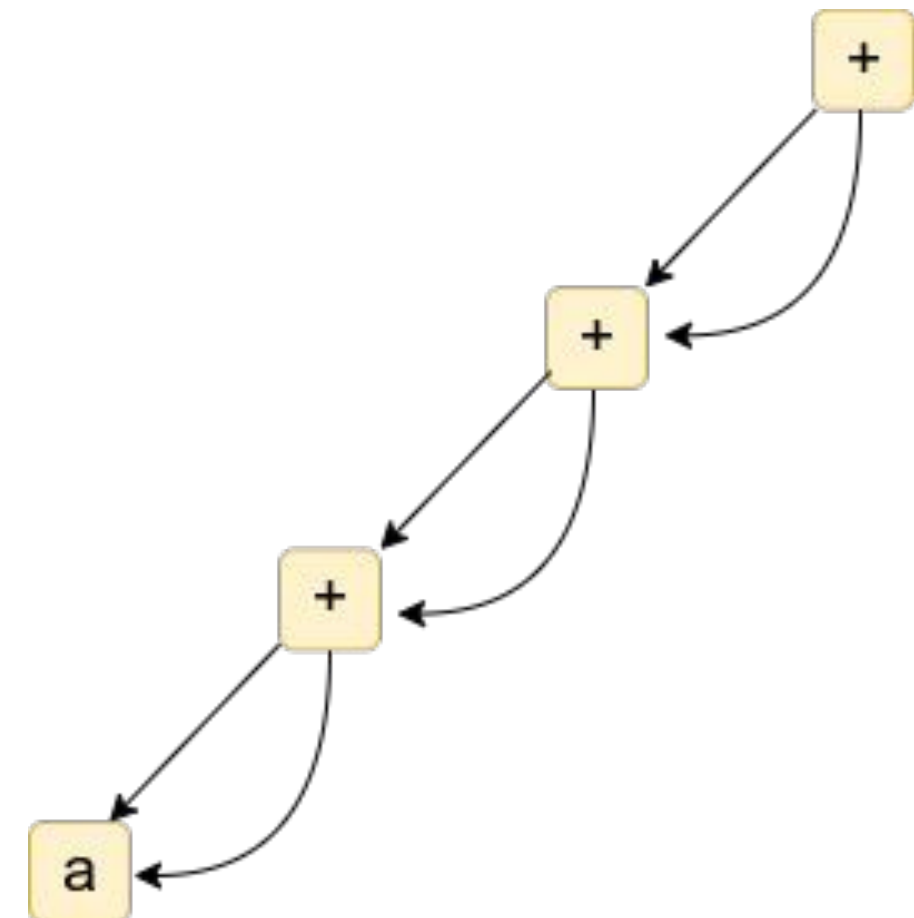




Exercise 2 - Solutions

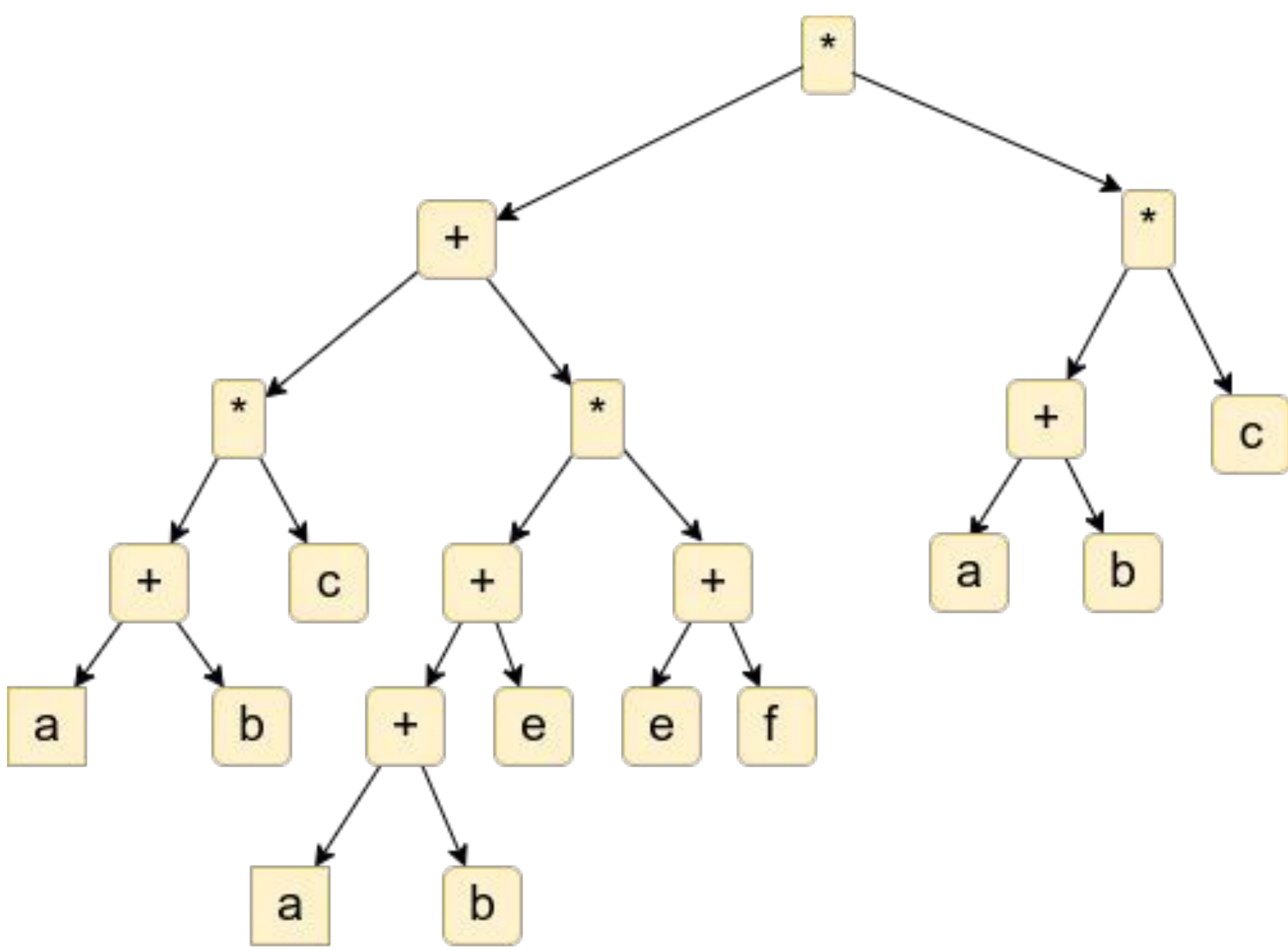
4) (((a + a) + (a + a)) + ((a + a) + (a + a)))

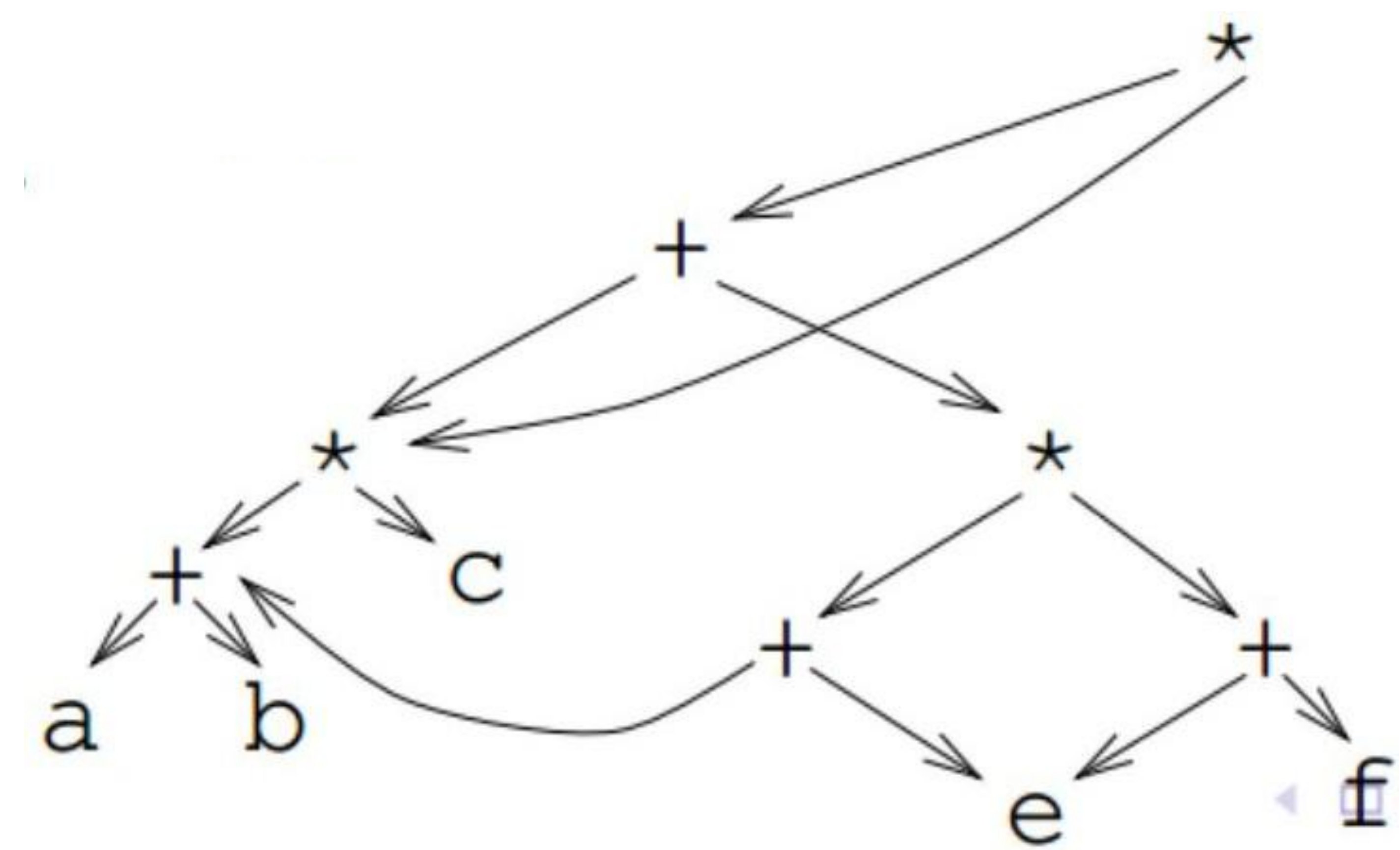




Exercise 2 - Solutions

5)$[(a + b) * c + ((a + b) + e) * (e + f)] * [(a + b) * c]$





THANK YOU

Preet Kanwal
Department of Computer Science & Engineering

preetkanwal@pes.edu