# COL719 Assignment 1
# Data Flow Graph Construction

Etiksha Jain: 2020EE10491
Geetigya Joshi: 2020EE10493

February 5, 2024

## 1 Problem Statement

Given a specification in a simple language defined informally through the sample specification below, build an **Abstract Syntax Tree (AST)**, followed by a **Data Flow Graph (DFG)**. The tree corresponding to each statement should be as discussed in class, with '=' at its root, the LHS variable as its left child, and the RHS expression as its right child. The set of trees corresponding to the statements can be connected to a linked list. The DFG should capture dependencies among the operations. Edges in the DFG should be annotated by the appropriate variable name, if relevant.

## 2 AST and DFG Construction Algorithm

To generate the AST and DFG for the specification we first tokenize the input text (**lexing**). Then the expression was converted from infix form to prefix form. The prefix expression was then recursively converted into a tree/graph.

### 2.1 Infix to Prefix

This conversion was done using a stack based algorithm.

1. Reverse the infix expression
2. Scan the (reversed) expression from left to right :
    - If the token is an operand, append it to the output string
    - If the token is an operator:
        - While the stack is not empty and the precedence of the operator at the top of the stack is greater than or equal to the precedence of the current operator, pop operators from the stack and append them to the output string.
        - Push the current operator onto the stack.
3. After scanning the entire expression, pop any remaining operators from the stack and append them to the output string.
4. Reverse the output expression to get the prefix form

### 2.2 Constructing the Tree Recursively

The prefix expression was converted into an AST using the following recursive algorithm

1. Extract first token from expression
2. If token is operand, create leaf node with value
3. If token is operator:
    - Create node for operator
    - Recursively call function for left and right tokens
    - Attach left and right subtrees to operator node

### 2.3 Constructing the DFG Recursively

The DFG is initialiazed as NULL. We create the following storage variables:

1. **single_assignment map** Map to store the new names of variables that have been reassigned.
2. **token_to_node map** Map to store the DFGNode corresponding to a variable token.

Every prefix expression was integrated into the DFG using the following recursive algorithm

1. **Initialization**:
   - Create an empty stack `nodes` to keep track of nodes during the construction process.
2. **Iterate Over Tokens**:
   - Start iterating through the `prefixExpression` from the end to the beginning.
   - For each token:
     - Create a new `DFGNode` named `newNode`.
     - If the token is a constant or operator:
       * If the token is a constant, create a new node and add it to `nodesList`.
       * If the token is an operator and already exists in `dfg_node`, retrieve it; otherwise, create a new node and add it to `nodesList`.
     - If the token is an operand (variable):
       * If the variable already exists in `dfg_node`, retrieve it; otherwise, create a new node and add it to `nodesList` and `dfg_node`.
     - If the token is an operator:
       * Pop two nodes (`opr1` and `opr2`) from the stack.
       * Add `newNode` as a child to both `opr1` and `opr2`.
       * Store the left parent of `newNode` as `opr1`.
       * Push `newNode` back onto the stack.
3. **Return the Root**:
   - The root of the constructed DFG is at the top of the stack. Return this node.

# 3 Tree-Graph Representation Design Choices

- **Node shape** We have created **elliptic** nodes for variables and constants and **square** nodes for operators.
- **Precedence** We have kept the precedence of addition and subtraction to be equal and less than the precedence of multiplication and division. Hence,

  $$a = b - c + d$$

  is equivalent to

  $$a = (b - c) + d$$

  .
- **Edge Color** Each operator is associated with two incoming edges. These edges are differentiated by color, with the **blue** edge representing the first variable and the **red** edge representing the second.
- **Handling WAW: Single Assignment Form** The single-assignment form is important because it allows us to identify a unique location in the code where each named location is computed and means that the data flow graph is acyclic—if we assigned to x multiple times, then the second assignment would form a cycle in the graph including x and the operators used to compute x. Therefore,

  $$x = a + b$$

$$x = x + b$$
$$y = x + c$$

is equivalent to,

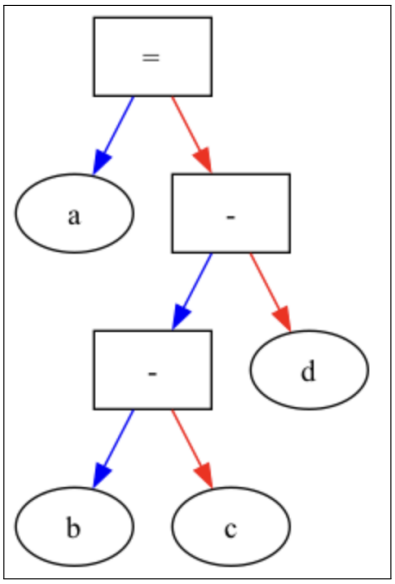
$$x = a + b$$
$$x1 = x + b$$
$$y = x1 + c$$

## 3.1 Example

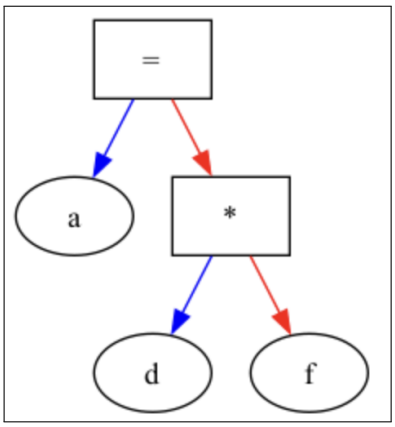
Input Specification -

```
a = b - c - d
a = d * f
a = a * 2
```

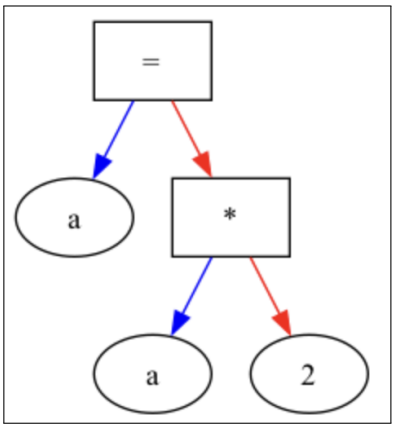
Figure 1: Input Specification



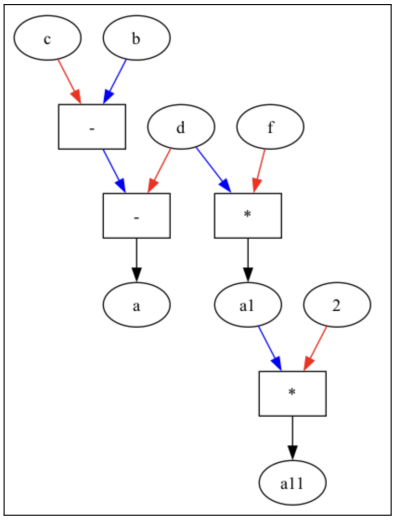

(a) AST of 1st statement




(b) AST of 2nd statement




(a) AST of 3rd statement




(b) DFG

# 4 Code

To run the code-

1. cd into */A1* folder.
2. Write the specifications into */io/example.txt* file.
3. Run the command ***sh run.sh***
4. The required **dfg.png** file and **ast_*.png** files will be generated in the */io* folder.