# GrAlgo

## Group 5

Sahil Chandra- CS20BTECH11033
P. Ganesh Nikhil Madhav- CS20BTECH11036
Gorantla Pranav Sai- CS20BTECH11018
Suraj Telugu- CS20BTECH11050
Umesh Kalvakuntla- CS20BTECH11024
Vanga Aravind Shounik- CS20BTECH11055
Adepu Vasisht- CS20BTECH11002

# Project Roles

- Project Manager - Sahil Chandra - CS20BTECH11033
- System Architect- Kalvakuntla Umesh - CS20BTECH11024
- System Architect - P. Ganesh Nikhil Madhav - CS20BTECH11036
- System Integrator - G. Pranav Sai - CS20BTECH11018
- Tester - Vanga Aravind Shounik- CS20BTECH11055
- Tester - Adepu Vasisht- CS20BTECH11002
- Language Guru - Suraj Telugu - CS20BTECH11050

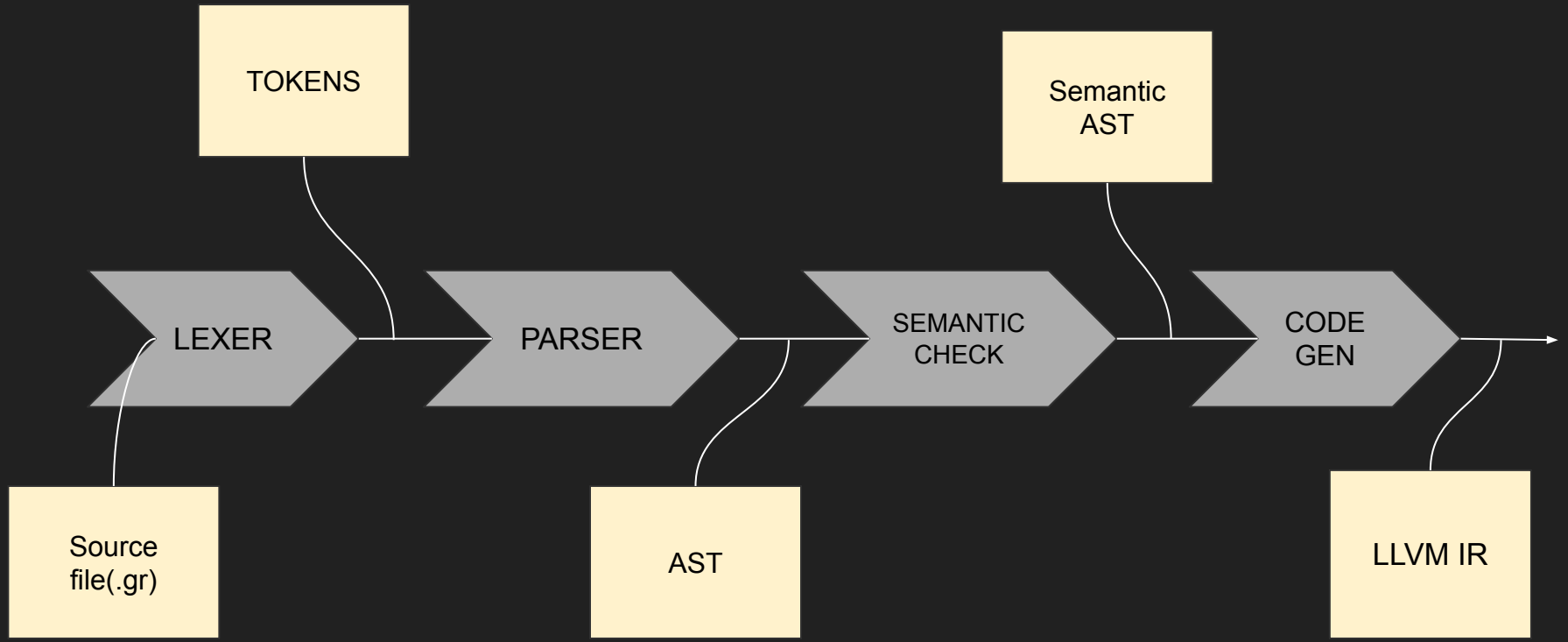
# Introduction

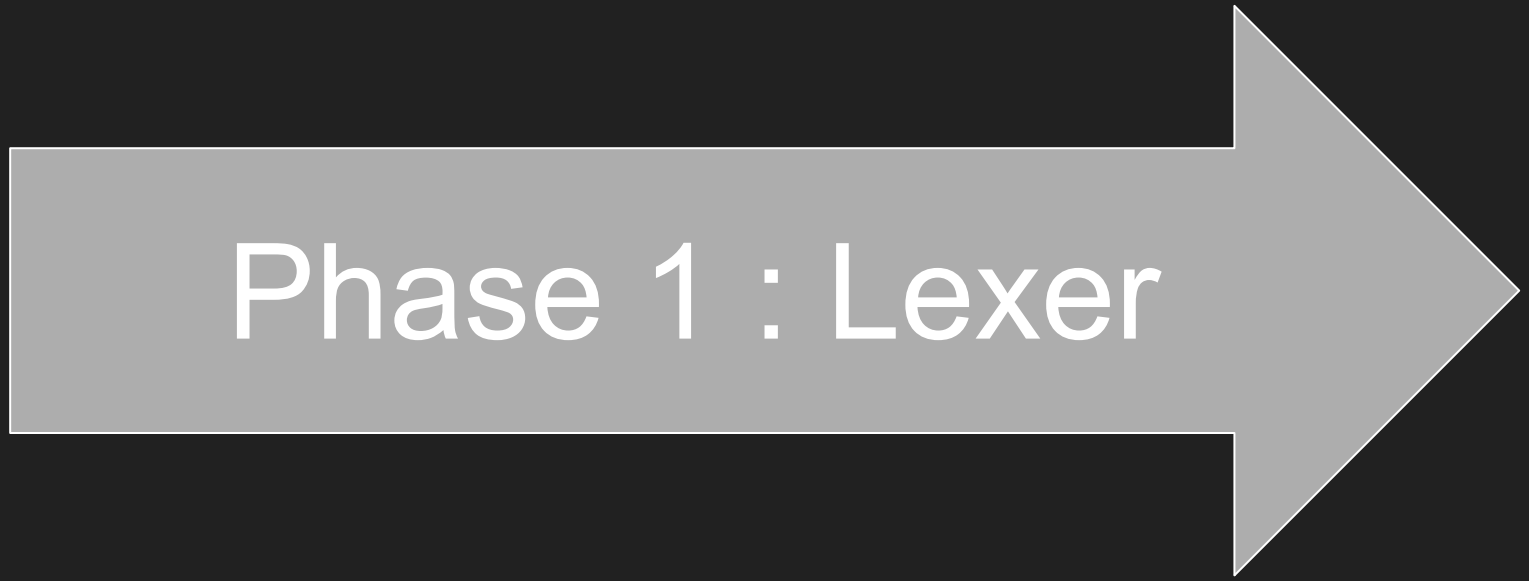
- The main idea behind our project was to bring the user closer to the graph data structure which is not present even in standard template library for C++.
- We provide most commonly used Graph Algorithms like Breadth-First Search(**BFS**)  and Depth-First Search(**DFS**) as inbuilt functions.
- We also support multiple types of graphs like weighted, directed etc with the internal representation as an adjacency matrix.
- GrAlgo can not only be used for graphs but even as a general purpose language making it versatile.

# How does a GrAlgo program look?

- Since our language aims reduce any hassle to user we opted for a syntax which most people would know i.e our language syntax is similar to C/C++.
- One main difference between Gralgo and C/Cpp is there is no need to include header files in our language.
- So all the C/Cpp stuff like “int main” are ported over to GrAlgo, function declaration is also similar to C/C++ so that user can write his own functions
- Code Snippet will be shown in the later slides.

# How does a compiler work?



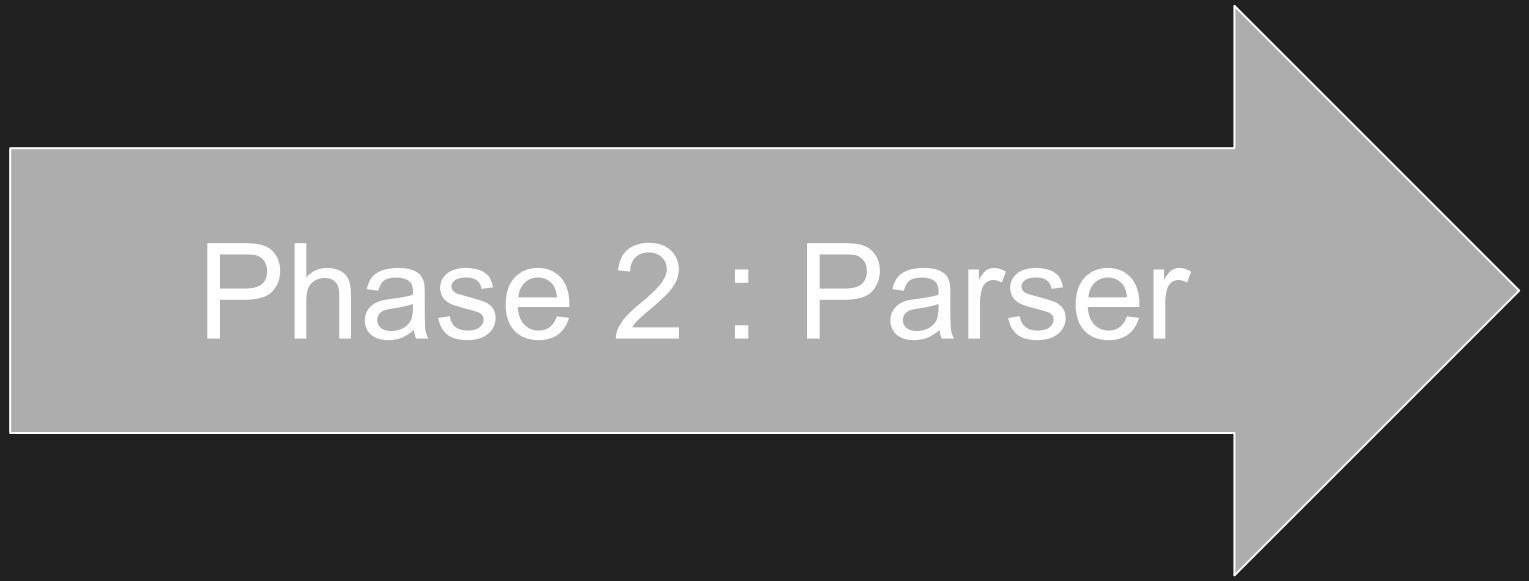

Phase 1 : Lexer

# Lexical Analysis

- A single element of a programming language is considered a token. It consists of Identifiers, separators, keywords and operators.
- Lexical Analysis is used to tokenize a given source code. It returns all valid tokens and mentions about the invalid tokens like invalid operators or invalid keywords.
- It also removes comments and whitespaces.
- We used FLEX (Fast Lexical Analysis Generator) to generate our lexer which has its own syntax.

# LEXER

- We use FLEX (Fast Lexical Analyzer Generator) to generate our lexical analyzer.
- FLEX takes in our tokens in a special file with a .l extension written in C language and generates a lex.yy.c file.
- When the generated lex.yy.c file is compiled using the C compiler we get a file named a.out which is our lexer.
- To this a.out file we give input stream and sequence of tokens is generated.


Phase 2 : Parser

# PARSER

- Parser is used for checking/verifying the grammar of our source language.
- Grammar: A program which is generally represented as a sequence of ASCII characters is changed into a syntax tree using grammar rules, which describe the ordering of symbols in language.
- Parsing function takes a lexical analyze and a lexical buffer as arguments which is a function from lexer buffers to tokens and return the semantic attribute of the corresponding entry point.
- Our parser is built using Bison/YACC.

# BISON/YACC

- BISON is a general purpose parser generator that takes in a CFG and generates a LR parser employing LALR(1) parser tables.
- BISON takes in our grammar in a special file with a .y which contains rules for the grammar and generates two files “parser.tab.c” and “parser.tab.h” using “-d” option
- Now we include “parser.tab.h” in the lexer file and run flex on our lexer file to get lex.yy.c file. Compiling the lex.yy.c file using gcc we get our parser.
- The output of the parser is an AST(syntax tree) on which we need to do semantic analysis next.

# Phase 3 : Semantics

# Semantic Analysis

- Semantic process is the process of drawing meaning from a text i.e , it interprets sentences so that computers can understand them and helps us in maintaining the semantic correctness of the program.
- Functions of Semantic Analysis are Type Checking, Label Checking and Flow Control Check.
- Semantic analyser is not a separate program but it is a set of actions within parser which uses the Symbol Table and Syntax Tree to verify the semantic correctness

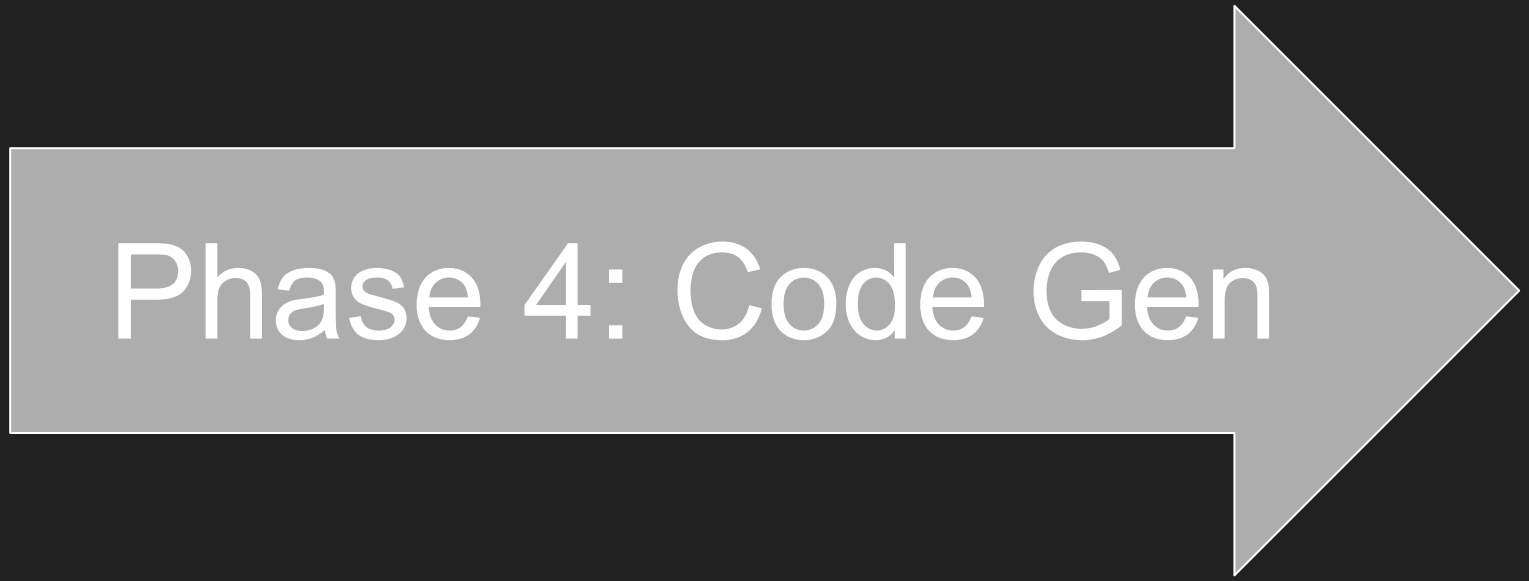
  It uses attribute grammar to add the additional actions required and type checking is done from symbol table using attribute grammar
- There are two types of Semantic Analysis
  - Dynamic Semantic Analysis.
  - Static Semantic Analysis.

# Syntax Tree

- Abstract Syntax Tree is a tree representation of the abstract syntactic structure of text written in formal language.
- It is called as “Abstract” because it does not represent every detail appearing in the real syntax.
- For example parenthesis are not needed to be represented as separate nodes and are used for grouping which is implicitly defined in the tree structure itself.
- Compared to source code an AST does not include inessential punctuation and delimiters.
- AST also contains extra information about the program such as position of each element in the source code to print appropriate error messages.

# Symbol Table

- The symbol table is an important data structure created and maintained by the compiler which keeps track of semantics of variables. Like keeping track of scopes and types of variables.
- In general there are two types of symbol tables generated one is Global Symbol Table and other one is Scope Symbol Tables.
- Scope symbol tables will be children to the Global symbol table(See fig in next slide), whenever a name is needed to be searched the compiler first searches in the current scope if not found then it searches in the parent symbol table until it is found or global symbol is reached.
- There are many data structures to implement symbol table each with its own advantages and disadvantages.


Phase 4: Code Gen

# Code Generation

- Code generation is the final phase of the compiler in which we turn Intermediate representation of the source code into a form of machine code that can be executed by the target machine.
- There are multiple types of Intermediate Representations such as Three-Address Code, Postfix Notation, Syntax tree and LLVM IR.
- This phase takes in AST and Symbol Table as the input and gives out an IR which is then converted to target code.
- Generally we optimise the code while generating IR on whatever the user may desire such as power, time or memory.

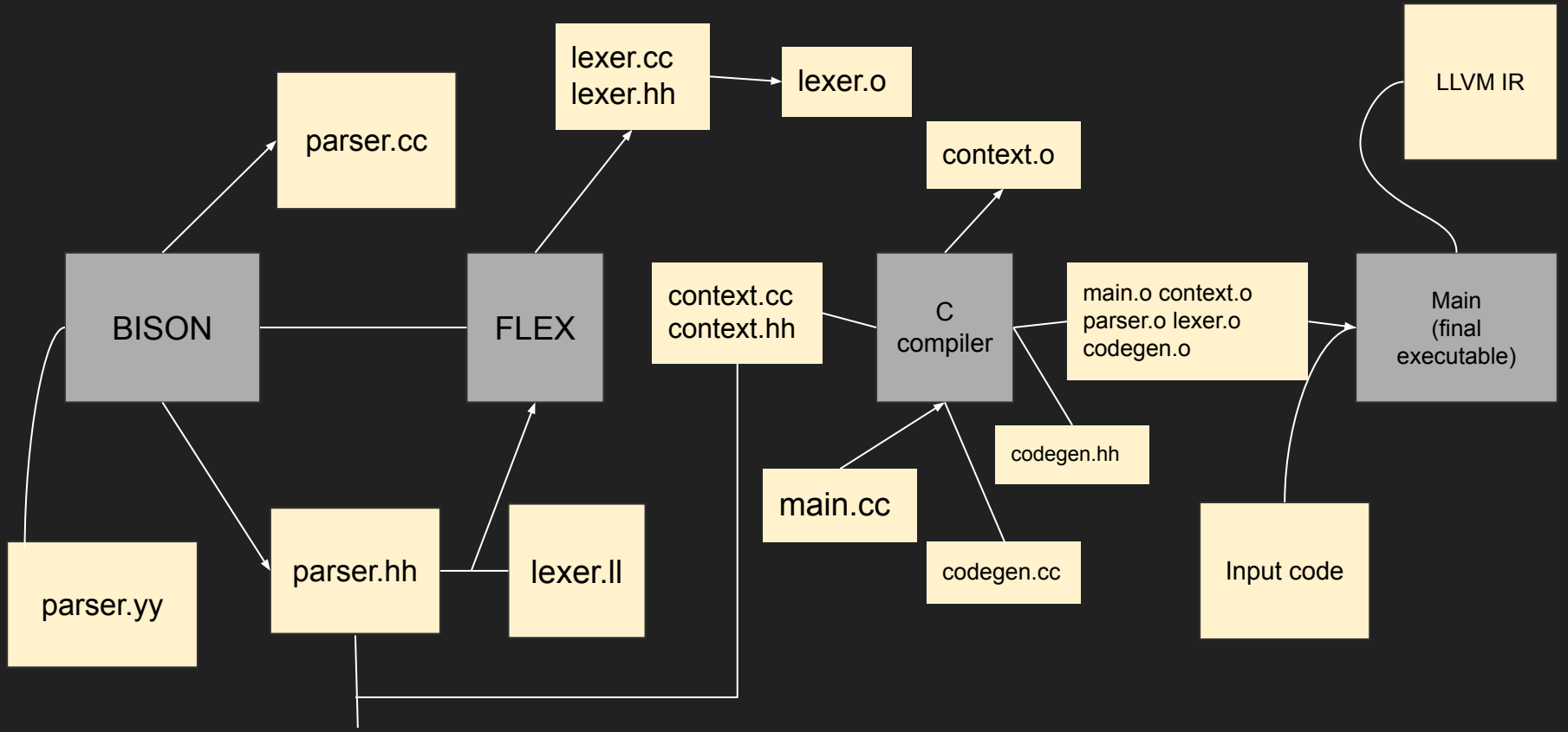
# LLVM IR

- LLVM is a Static Single Assessment based Language that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly.
- If we are able to generate LLVM IR then we can use the LLVM compiler to convert this IR into target machine code easily using a single command.
- That command is of the following format: llc [options] [filename] for example  if we have new.ll file and we want assembly code in say sparc we can run the following command: llc -march=sparc new.ll -o new.s and we get the assembly code for sparc in new.s

# Code Generator

- Using AST and Symbol table generated in the Parsing stage we generate LLVM IR.
- As in the parsing phase of the compiler we have stored our AST and our symbol table in main.cc file we call codegen function which performs the actual codegen process.
- With the  stored AST and the symbol table in main and from that call the codegen function which generates an .ll file which is an extension for LLVM IR files and using LLVM we can convert this .ll file into machine code of our choice.

# How do various files combine to bring our language?



# How to compile and run

- The folder “Semantics_CodeGen” contains the following files.
  - parser.yy  lexer.ll GrFlexLexer.hh
  - context.cc context.hh types.hh
  - codegen.cc codegen.hh
  - main.cc
  - Makefile
  - input<number>.gr in inputs folder (Test Cases)
- “make” command is used to run these files and generate the output executable file which takes .gr file as input and generates the LLVM IR.
- To get the output run the command `./main < input.gr`
- The output will be present in a file named test.ll . To which we can use llc to get target specific machine code.

# Example

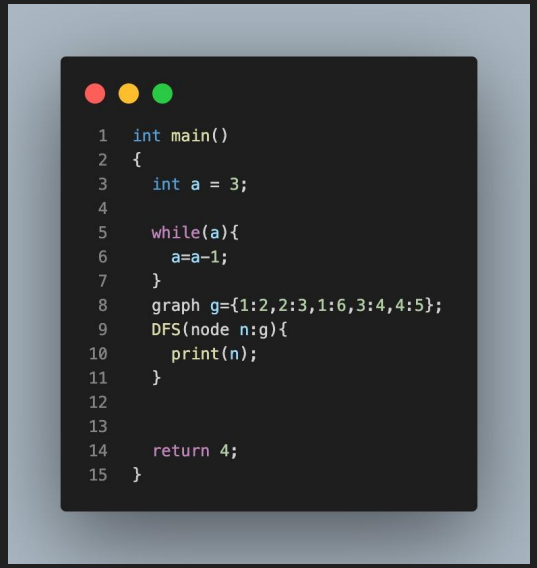
```
1  int foo(int f, int c)
2  {
3    return 2+3;
4  }
5
6  int main()
7  {
8    int x = 2+3-3;
9
10   return x;
11 }
```

Code Snippet

LLVM IR

```
1  ; ModuleID = 'gralgo'
2  source_filename = "gralgo"
3
4  define i32 @foo(i32 %f, i32 %c) {
5  entry:
6    %c2 = alloca i32, align 4
7    %f1 = alloca i32, align 4
8    store i32 %f, i32* %f1, align 4
9    store i32 %c, i32* %c2, align 4
10   %addtmp = add i32 2, 3
11   ret i32 %addtmp
12 }
13
14 define i32 @main() {
15 entry:
16   %0 = alloca i32, align 4
17   %addtmp = add i32 2, 3
18   %addtmp1 = add i32 %addtmp, -3
19   store i32 %addtmp1, i32* %0, align 4
20   %x = load i32, i32* %0, align 4
21   ret i32 %x
22 }
```

# Final Example(DFS)



```
int main()
{
  int a = 3;

  while(a){
    a=a-1;
  }
  graph g={1:2,2:3,1:6,3:4,4:5};
  DFS(node n:g){
    print(n);
  }


  return 4;
}
```

Code Snippet

Output

```
2
3
4
5
1
6
```

# Thank You!!