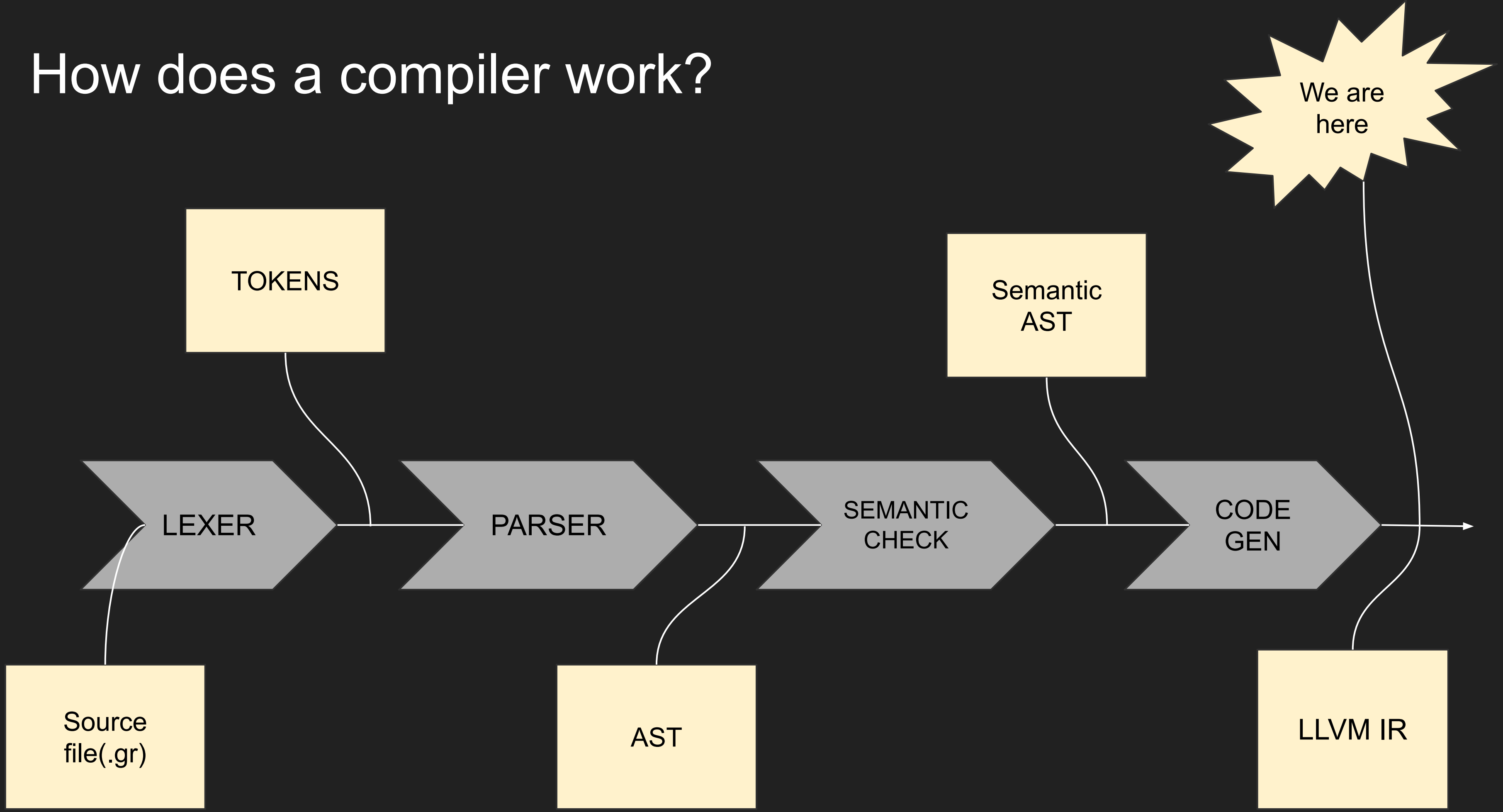
GrAlgo

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

How does a compiler work?



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 or 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

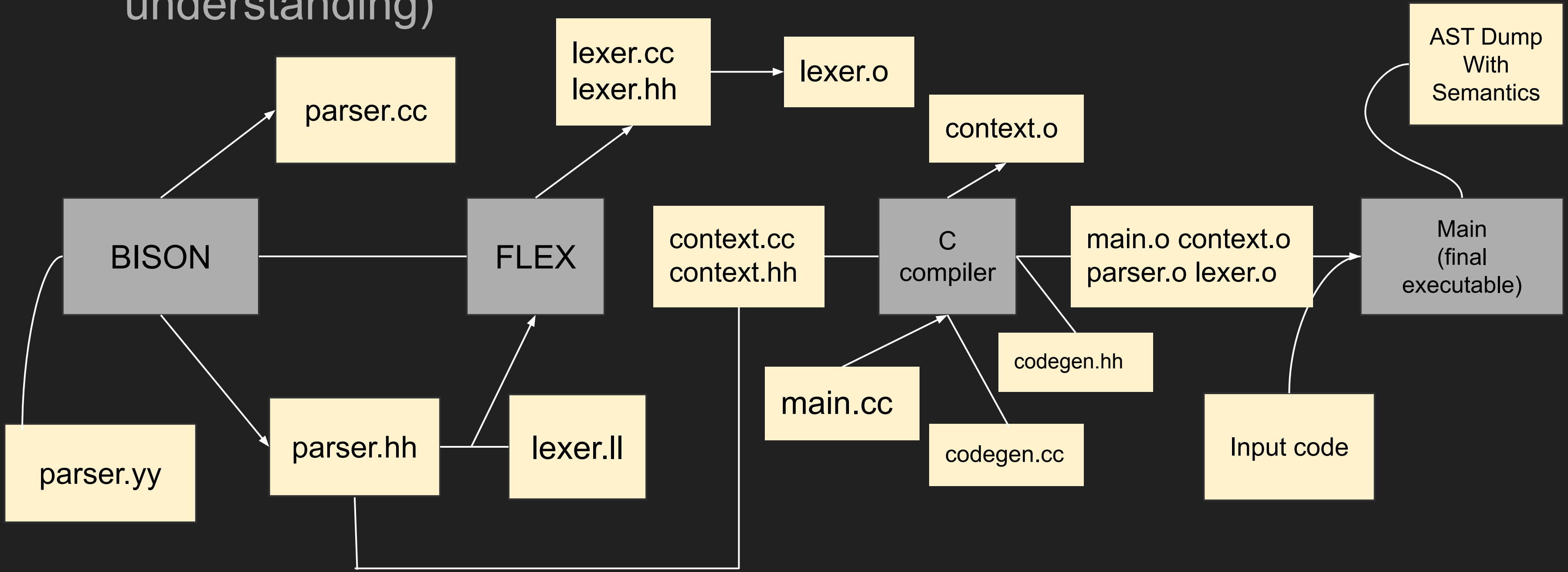
- LLVM is a Static Single Assessment based Language that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly.
- If we are able to generate LLVM IR then we can use the LLVM compiler to convert this IR into target machine code easily 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.
- We 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.

Code Generator

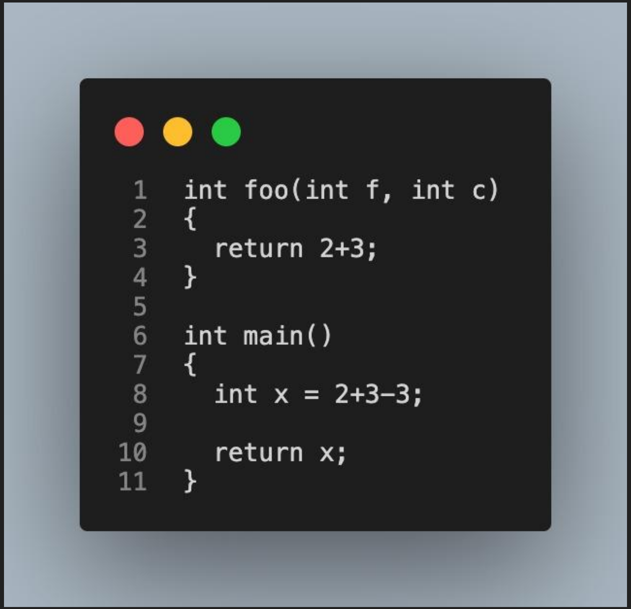
- Parser with semantic analysis uses **BISON** and **FLEX** for compiling. Context.cc file stores the Symbol Table. (See chart below for better understanding)



Using our Semantic Analyser

- The folder “Semantics” 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 AST and also does the semantic analysis.
- To get the output run the command `./main < input.gr`

Example



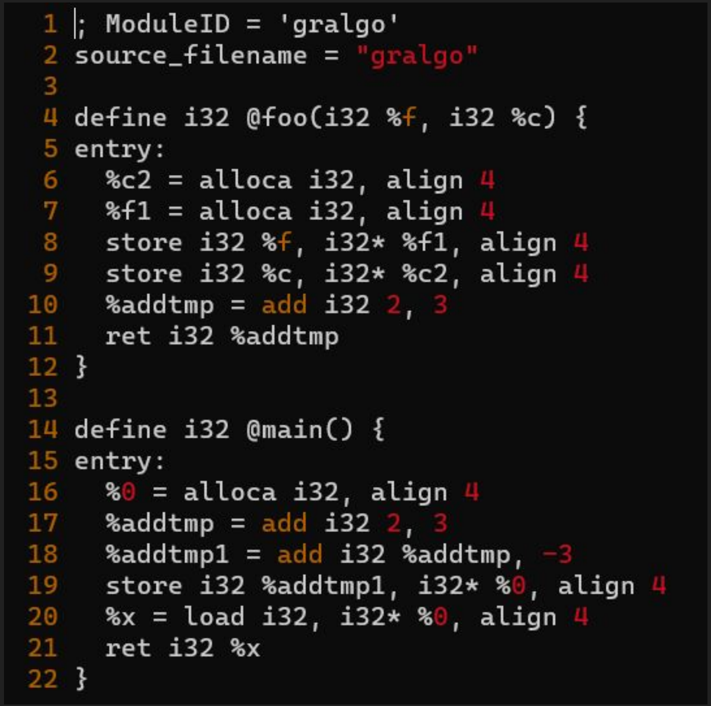
```
int foo(int f, int c)
{
  return 2+3;
}

int main()
{
  int x = 2+3-3;

  return x;
}
```

Code Snippet

LLVM IR



```
; ModuleID = 'gralgo'
source_filename = "gralgo"

define i32 @foo(i32 %f, i32 %c) {
entry:
  %c2 = alloca i32, align 4
  %f1 = alloca i32, align 4
  store i32 %f, i32* %f1, align 4
  store i32 %c, i32* %c2, align 4
  %addtmp = add i32 2, 3
  ret i32 %addtmp
}

define i32 @main() {
entry:
  %0 = alloca i32, align 4
  %addtmp = add i32 2, 3
  %addtmp1 = add i32 %addtmp, -3
  store i32 %addtmp1, i32* %0, align 4
  %x = load i32, i32* %0, align 4
  ret i32 %x
}
```

Thank You!!