CoPro:

T. Rohan - CS20BTECH11064
I. Rajasekhar - CS20BTECH11020
P. Vikas - CS20BTECH11037
S. Satvik - CS20BTECH11048
B. Revanth - CS20BTECH11007
J. Vamsi Preetham - CS20BTECH11058
G. Yagnavalkya - CS20BTECH11019

Introduction

- **Aim:** CoPro was built to help programmers program complex mathematical problems in the domain of conic sections, by helping them out with functions related to that domain.
- A lot of essential ideas of CoPro have been inspired by C

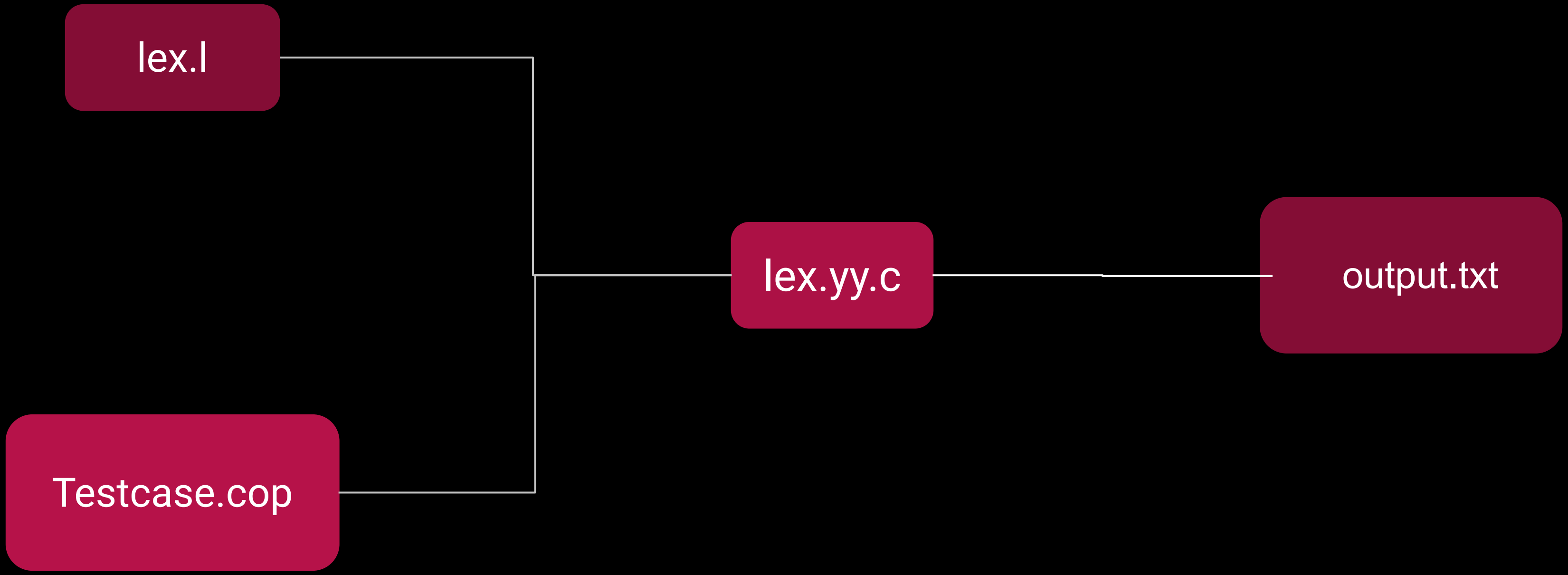
Description

- CoPro provides a variety of data types both trivial (bool, int, etc) and non trivial (parabola, ellipse, conic, etc.).
- Expressions are formed from operators and operands; any expression, assignment or function call can be a statement.
- CoPro also supports control-flow constructions such as if-else, loop etc.
- It also provides various in-built functions such as conic_name, tangent normal etc.

First Phase - Lexer

- Written using Flex. It takes ‘.cop’ files as input and tokenizes the file.
- In the ‘lexer.l’ file, rules are written to identify sequences, which would then be identified according to precedence and assigned the corresponding token.
- In the same manner it goes through the entire file, creating tokens and passes them to the parser file.

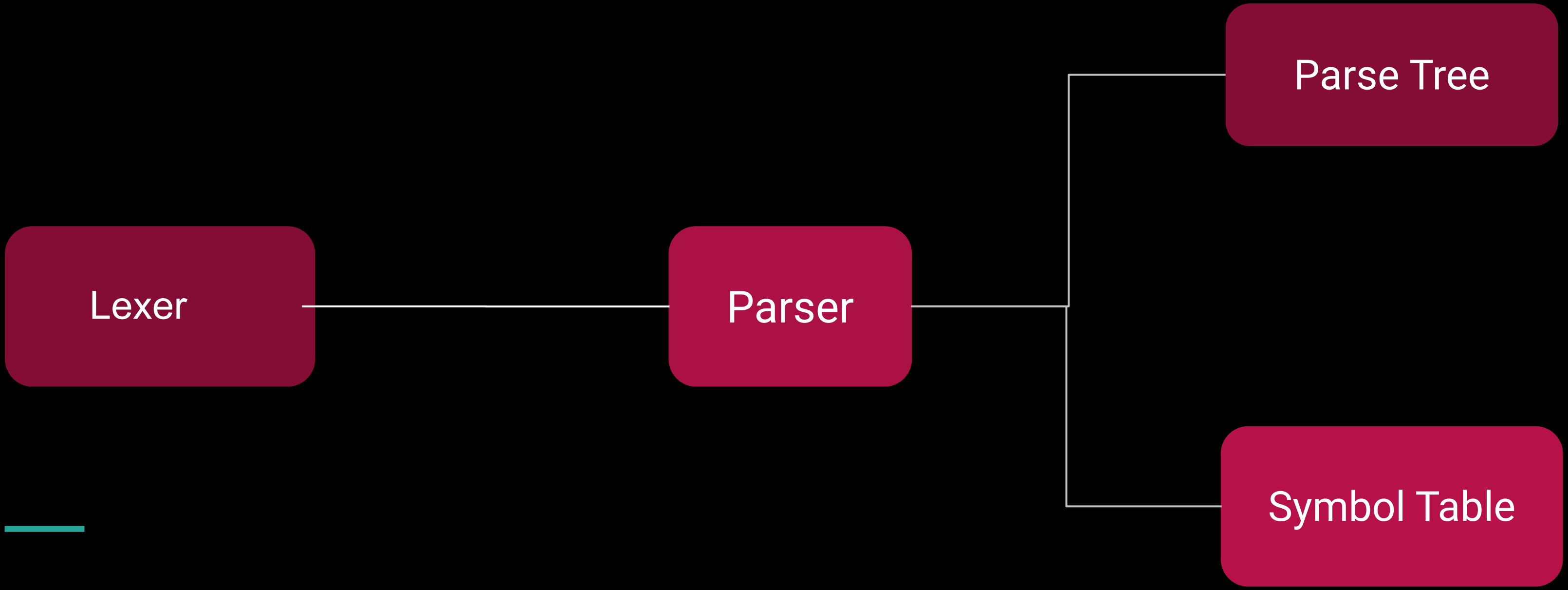
File Flow



Second Phase - Parser

- Written using Bison.
- Here we take tokens as input from the lexer file.
- Inside the parser file, the grammar of our language has been written. The grammar consists of rules to check the syntax of the language. It then goes on to build a syntax tree for the given code, checking if its errorless.
- We then write the C code for the parser file. This code is used to execute functionalities in the parser.

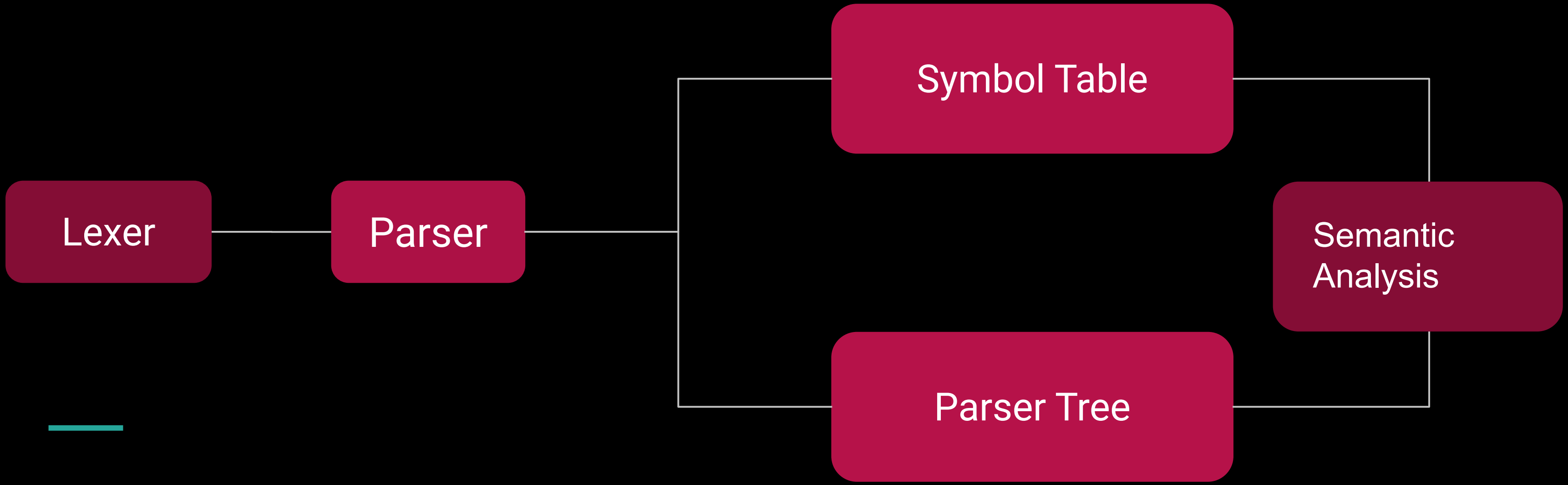
File Flow



Third Phase - Semantics

- This phase will generate the AST of the given code and we traverse this AST for semantic checking
- For example we have done 3 types of semantic checking those are
    - **Undeclared variables**: Checking whether variables are declared or not before using
    - **Multiple declaration of variables**: Checking whether the variable is declared multiple times.
    - **Type Checking**: Checking the types of operators we use in the binary operations
    - **Return Type and Parameter** checks in function calls.

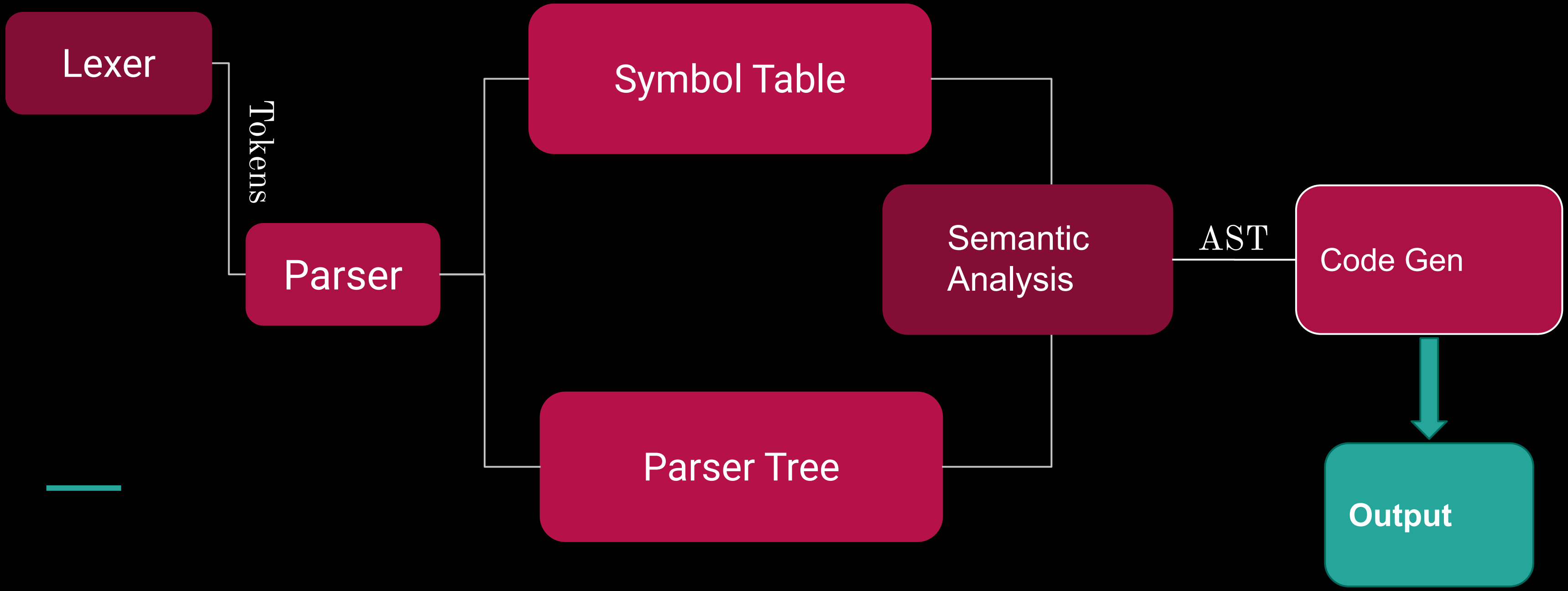
Control Flow



Fourth Phase - Code Gen

- Code generation is the process of taking the semantically checked AST and generating an executable from it.
- It is the fourth phase of building our Compiler.
- This phase of our project traverses the AST to generate the .ll file as output.
- Along with assembly files (.s) that are generated using the .ll files, we also generate gcc object files (.out) and llvm object files (.bc) which could be run to generate the output of the code.

Control Flow



Compile and Run

- First run lex and bison
  - $ flex -o util_files/project.lex.cpp project.l
  - $ bison -o util_files/project.tab.cpp -vd project.y
  - $ gcc $(FLAGS) util_files/project.lex.cpp util_files/project.tab.cpp -lm -ll $(CFLAGS) -o util_files/project
- To run the test case “test.cop”
  - $ ./util_files/project <test.cop> SymbolTables_Ast/test.txt 2> llvmoutputs/test.ll
  - $ llvm-as -o llvmObjects/test.bc llvmOutputs/test.ll
  - $ lli llvmObjects/test.bc
- To generate assembly files use command
  - $ llc -o AssemblyFiles/test.s llvmOutputs/test.ll
- We can run these assembly files using gcc using command
  - $ g++ -no-pie AssemblyFiles/test.s -o gccObjects/test.out
  - $ ./gccObjects/test.out

Thank You