# CSC467 Compilers and Interpreters

## Lab 3 Report – Abstract Syntax Tree with Semantic Checking

Klaudio Koci 1000701626

Sriram Sundararaman 1000522428

## Table of Contents

# ABSTRACT SYNTAX TREE (AST)

The Abstract Syntax Tree developed in this lab is the middle end of the MiniGLSL compiler. It requires inputs from the Parser and Lexer developed in previous labs, which the team borrowed from the Starter Package since their basic correctness is guaranteed although they may lack helpful features that the team developed in previous labs. The Abstract Syntax Tree provides an Intermediate Representation (IR) of the code written in MiniGLSL prior to the Code Generation phase, which converts it to the ARB Fragment language that can be run on a processor. The AST structure separates tokens of the language grammar into nodes, each node having its own description in memory and its own relations to other nodes in the tree. Semantic checking is done on all nodes in this lab implementation once the tree is complete.

The first portion of the lab, the AST construction, allowed the team to use the provided parser.y to find out how the language was being parsed into memory and how it was being recognized by the compiler. Once this information was obtained, the team used recursion in order to build the tree. In order to facilitate semantic checking, the team decided to build the entire AST (as per the lab handout suggestion) and then process the tree after.

The second portion of the lab, the AST printing, was one of the less challenging tasks of the lab as it simply required a traversal of the tree nodes in a top-down method so that the tree being printed would resemble the code written. The team achieved this milestone and the AST print function was used extensively in debugging phases later on.

The last and most challenging phase of the lab was the AST Semantic Checking, which required recursive analysis of the tree. The use of Symbol Tables (STs) was almost mandatory here as it enabled the team to check relations between variables in the same scope and other scopes. The team decided to not go with a BFS (Breadth-First-Search) approach since it required the retainement of too much information per node on the Symbol Table to be able to correctly catch semantic errors and the semantic error printing would stop after that tree level’s errors were discovered. Therefore, the team decided to build an entire ST for every scope first including all sub-scopes and perform all semantics checking for that particular ST. The data structure used for the STs was a stack that contained an ST entry at the top, the entry at the top being the most in-depth scope. The idea was to pop each ST off the stack once all semantics checking was performed at that scope level. This proved to be much more memory efficient and faster than the BFS approach. Moreover, with the stack implementation it was much easier to perform the semantic checking because we simply checked all nodes in the stack top() and if we did not find any declarations there we moved down the stack and checked the parent scope’s ST to see if the variable got declared there. This was important to note as finding variable declarations in all parent scopes was one of the challenges described by the team in a later section. Moreover, this data structure allowed the team to only keep a certain part of the program in context and ignore other irrelevant scopes, which greatly facilitated the recursive algorithm. Semantics checking was therefore completed using a recursive algorithm that pushed STs on the stack each time it entered a scope,

performed all semantic checking of each ST on the stack for that parent scope and trickled down to all child scopes, and then popped STs each time it exited a scope until the parent scope was reached.

# TESTING KITS

The code for this lab was inherently complex to debug given the recursive implementation of the AST and the printing of the AST only really helped upon correct execution of the compiler. In order for the team to be able to determine if the AST implementation was correct or not, the team wrote two main test files which represent code that one may see on a real programming session with MiniGLSL. In this way, the team was able to see if the compiler behaved correctly in the event of a regular program being run and dealt with numerous testcases that may arise. Two files were developed for this purpose:

***test_pass.frag*** – This file was developed with the sole purpose of containing code that completely adheres to the compiler specification in the MiniGLSL compiler specification on the course website. It contains numerous programming implementation methods and aims to cover as many testcases as possible. The file is available in the code submission for this lab report.

***test_fail.frag*** – This file was developed to be the exact opposite of test_pass.frag, in fact it is a copy-paste of test_pass.frag but with incorrect implementations of many programming instances. The file demonstrates that our compiler will report all errors it can find in the semantics checking process before exiting, it changes the types of incorrect results to “any” as the lab handout specifies, and it also reports useful error messages to help the programmer debug the code. The file contains comments on sections where errors are expected to occur to help the reader identify the numerous semantic checking errors that may arise.

The team used these two files to determine the correctness of the compiler’s AST construction, printing, and semantics checking. Upon correct compilation of these two files (for the test_fail.frag upon successfull error reporting) the team decided that the compiler is complete.

# DIFFICULTIES FACED AND BUG RESOLUTION

This lab contained many challenges which the team had to address prior to this submission. Many difficulties in this lab came from the recursive structure of the AST algorithm the team implemented. The main challenge here was to get the information from leaf nodes to travel back up to parent nodes so that information would become available for checking and printing. This was due to the fact that the team developed the cases for the AST using a top-down approach, handling the scopes first, then the declarations, then the statements, etc. However, once the team developed the case for the variable, identifier, basic types and vectors nodes the recursive structure became much simpler as it was possible to now simply recurse on the child nodes and these cases would take care of the returning the values needed. This also gave rise to some tough bugs that the team had to address, such as variable declarations inside of parallel scopes because the BFS implementation that was initially decided on did not allow for the differentiation of scopes and their levels in an easy way. Then the team came to the

realization that the Symbol Tables could be built each time a scope was entered and popped off the stack each time a scope was exited. This effectively threw out all the variables within that scope, which are not available for the rest of the program to use or see (declarations especially). Another tough bug was to deal with binary expressions since the recursive algorithm required the fully implemented code in order to work correctly since there were so many different possibilites once an if-statement was encountered and all needed to be addressed (functions, more statements, more scopes, uniform variables, etc.). These were some of the major challenges faced and bugs fixed for the team and they were all addressed prior to the submission of this report.

# DIVISION OF LABOUR

The team worked on the following items:

Sriram:

- Code for AST Printing
- Code for AST Construction
- Code for AST Semantic Checking
- Code for Symbol Table
- Report Finalization

Klaudio

- Code for AST Semantic Checking
- Code for testing kits
- Code for Symbol Table
- Report writing
- Report finalization