Name: Naga Akshay Garigiparthy CID: 00838074

Language Processors Coursework: C Compiler Report

The aim of this project was to design a C compiler and generate functioning ARM assembly code. This report has been written to detail the many design choices made and to highlight the challenges faced in the construction of the compiler.

Overall Design and Structure

This compiler can be broken up into four distinct parts: lexer, parser, Abstract Syntax Tree (AST) and a code generator. More explicitly the steps performed by this compiler are:

1. Lexical Analysis - breaks up input code into Tokens i.e operators, identifiers and numbers.
2. Syntax Analysis - Parser uses a set of grammar production rules to check token arrangements and determines whether the code is syntactically correct.
3. Semantic Analysis - After checking syntax, the parser then checks that expressions formed from the input token stream are semantically correct.
4. AST - Following semantically correct code, a tree representation of the input source code can be created. Using the grammar rules, only certain details of the source code a needed to be stored in the tree structure.
5. ARM code generation - Using the tree representation of the input source code, equivalent ARM assembly instructions are generated and written to a .s file.

To generate my lexer and parser files, I used the flex and bison programs. The code generated from these two programs was then linked with separate c++ files and data structures to form the overall compiler design.

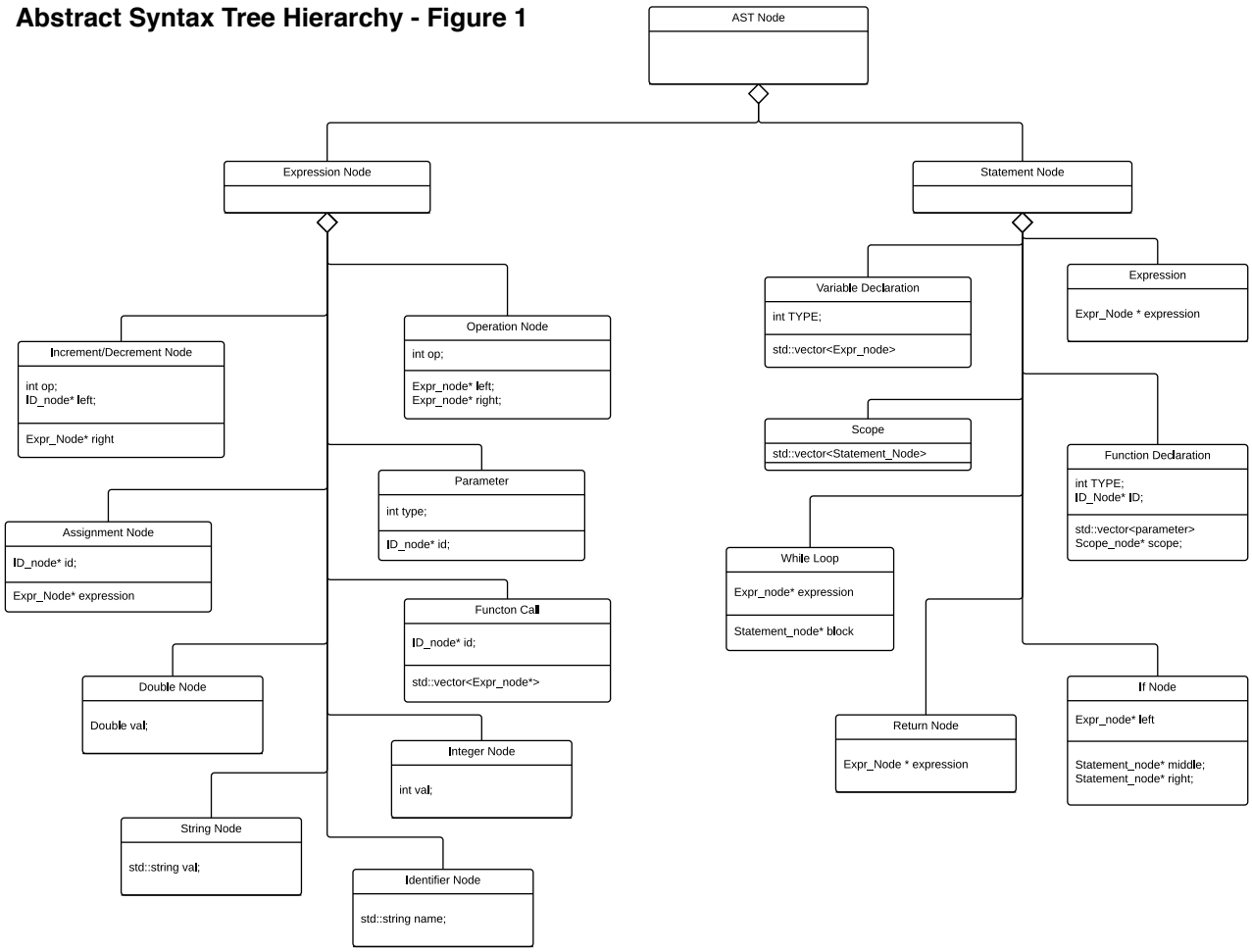
Grammar

The grammar structure was made from thee components: terminal symbols (tokens), non-terminals and productions. Non-terminals are groups of non-terminals where the specific terminals used for a specific non-terminal are defined by production rules. For example, a variable declaration (non-terminal) is made from a type token followed by an identifier token. The overall grammar was formed under the assumption that all C programs are essentially a set of commands made up of declarations, expressions, scopes and other statements (if-else statements and loops). The declarations permitted are for functions and variables. Generally c programs start with a few of these declarations but this is a matter of semantics, not syntax and so the grammar does not require such a format. Semantics are, however, monitored during checks performed during construction of the AST. Expressions contain the broadest range of possible commands including operations, assignments and function calls. In my grammar, the main program is interpreted as a ‘scope’. The scope definition in my grammar is simply a set of commands where the only difference between them is the variables and/or functions available for use depending on which scope they were declared.
From this grammar, it is possible to built the AST structure to better represent the input code. This AST structure (explained in detail later on) is defined in the *ast.hpp* file as a set of classes which represent nodes. Through the use of inheritance, each node is connected to another to make a complete tree possible.

Design Decisions

The construction of this compiler was only possible by making some important design decisions to enable all sections of code to work together in the best way possible. The most important decision made was the way in which I would implement the AST structure (see *figure 1*). The main decision was to essentially have one “super-class” with two secondary “super-classes” inheriting from it. I chose to split these to classes into statements and expressions. The effectively reflects the design of my grammar. The statement class was used to group commands (in their general form) permitted within a scope. As is evident from the grammar such commands include declarations, expressions, loops and conditional statements. Since there were still quite a few specific production rules that came under the expression non-terminal, this was turned into a second “super-class” which grouped more specific expressions such as operations, function calls and assignments. The use of inheritance in this way meant that classes could simply contain pointers to an *“Expr_Node”* and not need to worry about the specific type of expression which inherited off that class. The classes represent specific types of expressions/commands and through the use of polymorphism each class can have a set of virtual member functions which behave in ways suited to the relevant class. Whilst this allowed for printing of the tree during the debugging process, polymorphism was most helpful in the code generation stage of the compiler (explained in detail later on).
Obviously the ability to tailor function behaviour to specific classes is an advantage, the existence of so many classes could also be seen as a disadvantage if care is not taken when constructing the tree. This is why I placed great emphasis on the construction of my grammar and making sure it is as unambiguous as possible. Nearly every production rule would lead to different functions performed to ensure the correct AST was being built. By continually testing the grammar before the AST stage, I could be fairly sure that it was clear enough for a tree to be built from it.

Abstract Syntax Tree Hierarchy - Figure 1



Name: Naga Akshay Garigiparthy CID: 00838074

Code Generation Data Structures & Algorithms

The method of code generation was another major design decision to make. Since the ARM architecture makes use of memory and register based instructions, I had to be aware of these two storage methods when handling and performing operations with my data. Given that register to register operations are faster than those requiring memory access, i knew that maximum use of registers was ideal. However, due project time constraints this was not possible. Instead, I made use of my *'codegen'* functions to generate specific lines of assembly code depending on the respective class. To two key data structures used during the code generation phase were a variable-to-memory map and a global boolean array of registers. The map was used to link variable names with pre defined locations in memory where they would be stored. This particular function was performed in the variable declaration class. As a result of semantic checking performed during parsing to ensure variables are not redeclared or used before declaration, it is highly unlikely that there would be any errors when dealing with the mapping of variables. This map is passed by reference whenever a class calls the codegen function of another. The use of this map is only evident the in *ID_node* class where the address of the variable is retrieved and an LDR instruction is outputted.
The boolean array was used to keep track of which of the thirteen ARM registers were free to use. The availability of them was determined using a simple for-loop in the relevant classes and finding the first free register. Due to time constraints, I had decided to perform STR operations as soon as possible, thereby releasing registers as soon as possible at the expense of speed and efficiency. The reason for this was because I was not able to implement the necessary algorithms to safeguard against running out of registers to use. Ideally, I would have implemented an algorithm such as Least Recently Used (LRU) to keep track of registers after all thirteen had been used. To implement this algorithm I would have kept the same array of registers, but instead assigned an incrementing variable with each register to track which one was used least recently. Such an algorithm would have resulted in far more efficient use of register-register instructions and only requiring memory access when absolutely necessary.