CS3423: Compilers-II

DROOL

Data Representation with an Object-Oriented Language

Pendyala Devi Aishwarya | ES18BTECH11006
Deepika Reddy Palagani | ES18BTECH11002
Sandeep Kumar | CS18BTECH11041
Shreyas Jayant Havaldar | CS18BTECH11042
Manya Goel | CS18BTECH11050
Aditya Singh | CS18BTECH11048

Under Dr. Ramakrishna Upadrasta

Contents

Chapter 1

Introduction to DROOL

1.1 Why DROOL?

DROOL (Data Representation with an Object-Oriented Language) is an object-oriented programming language which draws inspiration from the functionality of C++ [5] and the simplicity of COOL [1]. DROOL is made to be adept at handling particular forms of data such as graphs and matrices, using customized in-built data structures to handle them. DROOL aims to provide programmers with a language that provides ease in manipulating data in its different representations as well as the provision to perform general programming tasks.

1.2 Design Goals

DROOL, as the expansion suggests, is a amalgam of two of the hottest properties in the programming market right now, namely Data Representation and Object Oriented Programming. We have tried to create an alloy for facilitating users to embark on a journey whose tools they pick and we try to offer a full inventory for them to fight all the demons they may encounter in this programming labyrinth.

1.2.1 Data Representation

Data is the new oil and we aim to make extracting this oil easier. We want to provide the user of DROOL, a drool-worthy option to be able to express

and manipulate their data in various forms without losing out on the basic functionalities that comes with a programming language. We wish to provide the best of both worlds by promising simplicity for a new entrant in this complex world of programming and tools to make the most of his/her abilities for the experienced veteran to unleash the potential of the language for solving complex problems across numerous domains.

1.2.2 Object Oriented Programming Language

We as a team believe object oriented paradigm is the way forward and an intuitive and natural way to model real world problems and via DROOL aim to provide a multi-faceted tool, equipped with Inheritance, Polymorphism, to build and solve virtualization of the complex problems that they might want to emulate and find solutions for. DROOL provides for implementation classes and objects like any other standard object oriented programming language. The classes can have attributes and methods and other standard practices as seen in JAVA [2] and C++ [5] are emulated sensibly.

Chapter 2

Language Specification

2.1 Salient Points

We wrote the grammar for the lexical analyser in ANTLRv4 [4]. We tried to maintain resemblance to standard language specifications like that of C++ and JAVA [2] in particular to enable seamless transition to DROOL for all languages and encourage people to switch because they won't have to learn a whole new world of syntactical sugar, while making minor adjustments and improvements to make the programmers more efficient.

2.1.1 Keywords

Case insensitive, unless specified, to provide fast programming capability and preventing unnecessary unrecognised literals to due capitalization and such minor errors new programmers are prone to make. We want the users to focus on solving problems and not spend hours debugging an erroneous case issue in one random keyword at some random line in the program. We make implementation of case insensitivity extremely simple by using fragments to redefine each letter of the alphabet as a case insensitive eponymous fragment.

2.1.2 Identifiers

DROOL supports starting with an alphabet or underscore and can contain digits, alphabets or underscore. We maintain the standard globally popular notation used in all major languages like C++, JAVA to let the user maintain their current practices while working with DROOL. It has almost become

natural to programmers to name identifiers in this format and we continue this practice.

2.1.3 Error Handling

We have tried to maximise the error handling at the lexical analysis stage itself by constructing robust token definitions. Comments as explained above have been described such that all correct and incorrect comments are segregated appropriately, irrespective of the input complexity.
For Handling Syntax Errors in the parser, ANTLR4 utilizes a Public Interface **ANTLRErrorStrategy** under **org.antlr.v4.runtime**. This interface defines the process to identify and recover from the syntax errors encountered while parsing the ANTLR-generated Parser.

Line numbers and the encountered errors are displayed all at once in the command line interface and also highlighted in red in parse tree generated. The Syntax errors encountered are of the following type:-

- When the parser is not able to identify the path it should follow to produce the syntax tree i.e. none of the defined grammar rules provides the parser a specific path to follow.
- When the input does not match the expected expression as defined by the parser rules.
- When a parser rule evaluation produces a false predicate.

ANTLRErrorStrategy is only responsible for the identification of the syntax errors, while reporting of these errors is managed by calling **Parser.notifyErrorListeners**.

2.2 Operators

Precedence and Associativity of Operators in DROOL is derived from C++ [5]. Listed below are the operators supported by DROOL:
Note: Operators that are not supported by C++ are described in detail in coming sections.

- Or: |
- And: &
- Star: *
- Plus: +

- Minus: -
- Tildae: ~
- Not:!
- Div: /
- Mod: %
- Less: <
- Greater: >
- LessEqual: <=
- GreaterEqual: >=
- Equal:==
- NotEqual: !=
- Caret: ^
- AndAnd: &&
- OrOr: ||
- Assign: =
- StarAssign: *=
- DivAssign: /=
- ModAssign: %=
- PlusAssign: +=
- MinusAssign: -=
- XorAssign: ^=
- OrAssign: |=
- PlusPlus: ++
- MinusMinus: - -
- Dot: .
- Sizeof: S I Z E O F
- Hashtag: #
- Addc: A D D C
- Addr: A D D R
- Delc: D E L C
- Delr: D E L R
- Questionmark: ?
- Pull: >>
- Push: <<
- Inv: I N V
- Trans: T R A N S
- Det: D E T
- Vsizeof: V S I Z E O F
- Esizeof: E S I Z E O F
- Val: V A L
- Comma: ,
- Colon: :
- Semi: ;
- SingleQuote: '
- DoubleQuote: ''

2.3 Data Representation

2.3.1 Standard Data Types

Derived from C++ [5] syntax with minor changes as and where mentioned.

- **Numeric Data Types**
  - Int: Tokenized as IntegerLiteral; Integer numbers that can be followed by a suffix for Unsigned, Long, LongLong. Required for standard representation of numbers and for arithmetic operations.
  - Float: Tokenized as FloatingLiteral; Floating point numbers (i.e. numbers with a fractional part) containing '.' or 'e'(exponent). We make necessary checks to ensure the entered float is correctly represented in either of the two formats specified above. We wish to provide the users flexibility in dealing with decimal numbers with precision and perform operations on them.
- **String Data Type**: Tokenized as StringLiteral; Strings enclosed within double quotes. We have ensured situations where a string contains null character or end of file character does not occur by displaying such strings as errors during lexical analysis itself. Further, we have extensively described all possible situations where a string might be in incorrect format, such as being unterminated, or within incorrect quotes in our lexer grammar. We limit the string size to 1024 characters.
- **Boolean Data Type**: Tokenized as BooleanLiteral; as either 'true' or 'false' and these values are also case insensitive.

2.3.2 Graphs

DROOL allows the user to manipulate graph data type easily and offers simplicity to the user for many graph related operations by providing data types for edge, vertex and graph and supports various graph operations which are explained in detail below.

- Associated Data Types:
  - Graph
    Graph is a mathematical representation of a network and it describes the relationship between lines and points. A Graph G

consists of a set on nodes/vertices represented using the Vertex Data Type and a set of edges associated with these nodes represented using Edge data type. We have provided operations to ease the process of adding and removing edges from a object identifier of type Graph. A graph can be specified as a directed graph by including the -d option at the time of declaration.

- Edge
  An edge is defined as a link between 2 nodes/vertices, An edge e is defined as a 3-tuple (v1,v2,w) where the the vertices v1 and v2 are connected by a weighted line of weight w. If w is not provided, it is assumed to be 1 by default. It is considered to be directed from v1 to v2 if the graph was specified as a directed one during declaration.
- Vertex
  Tokenized as VertexLiteral: name of a vertex of a graph enclosed in single quotes followed by a comma and vertex value (Default is null). We have provided this to equip the user to describe vertices in their graphs as simply as they would describe an integer for arithmetic and make graph operations as simple as integer operations.

- Associated Operations:
  - Push Operator(<<):
    - Vertex Associated (g<<v)
      Pushes/inserts 'v' of type Vertex in Graph 'g'.
    - Edge Associated (g<<(v1,v2,w))
      Inserts or establish an edge between the vertices 'v1' and 'v2' of type Vertex with a integer weight 'w' associated with the edge in Graph 'g'.
      NOTE: For inserting an edge between 'v1' and 'v2', it is necessary that Vertex 'v1' and 'v2' are present in 'g', otherwise the operation will fail.
  - Pull Operator(>>):
    - Vertex Associated (g>>v)
      Pulls/extracts 'v' of type Vertex from Graph 'g'.This will also lead to deletion of all edges which had 'v' as one of its vertex.

    * Edge Associated (g>>(v1,v2,w))
      deletes the edge between the vertices 'v1' and 'v2' of type Vertex with a integer weight 'w' associated with the edge in Graph 'g' if it exists.
      NOTE:Deletion of edge between 'v1' and 'v2' does not include removing the vertices themselves, This operation just removes the edge between them.

- Size Operator:
    * Vertex Set Size(vsizeof(g)):
      Returns the number of vertices present in a Graph type identifier 'g'.
    * Edge Set Size(esizeof(g):
      Returns the number of edges present in a Graph type identifier 'g'.
- Questionmark Operator(?):
    * Vertex Existence(v?g):
      Returns True if 'v' of type Vertex exists in 'g' of type Graph else return False.
    * Edge Existence((v1,v2)?g):
      Returns the Weight associated with the edge (v1,v2) in Graph 'g', where 'v1' and 'v2' are vertices present in graph.If either of 'v1' or 'v2' does not exists in Graph 'g' or there is no edge between them,It returns 0.
- Hashtag Operator(v#g):
  Returns an Vertex Array, where every element of the array is a neighbour of vertex 'v' in Graph 'g'.
  NOTE:(#g) returns a Vertex array of all the vertices present in Graph 'g'.
- Val Operator(val(v)):
  Returns a string associated with the Vertex 'v'.
- Graph Addition (g1+g2)
  Return a Graph 'g' with a vertex set V=(V1 $\bigcup$ V2) and an Edge set E=(E1 $\bigcup$ E2) where (V1,E1) and (V2,E2) are the vertex and edge sets of 'g1' and 'g2' respectively.
- Graph Subtraction (g1-g2):
  Return a Graph 'g' with vertex set V=(V1/V2) and edge set

E=(E1/E2)where (V1,E1) and (V2,E2) are the vertex and edge sets of 'g1' and 'g2' respectively.
NOTE: removing vertex set V2 from V1, will also delete all the edges which where associated to any vertex in set V2.

- Syntax:

```
Graph g1, g2 -d, g3;
Vertex v1, v2;
v1 = 'name',"80";
v2 = 'bd';
g1 << v1;
g1 << v2;
g1 << (v1, v2, 5);
g2 = g2 + g1;
g2 -= g1;
Bool t = v1?g1;
Int t = (v1,v2)?g1;  //returns weight of edge (v1,v2)
Vertex n[8] = v1#g1; //neighbour set of v in g
Int v = vsizeof g1;  //no. of vertices in g
Int e = esizeof g1;  //no. of edges in g
Vertex N[8] = #g1;   //vertices of g1
String h = val(v1);
```

2.3.3 Matrices

- Associated data Types:
  - Matrix
    A Matrix is used to represent a 2-Dimensional Array Data Structure, We have distinguished the Matrix from the basic 2-D array structure by defining Matrix-specific operations for the same.This provides a user-friendly interface to access, manage and manipulate Matrices.

- Associated Operations:
  - Matrix Addition (+):
    Operation of adding two matrices by adding the corresponding entries together.

- Matrix Subtraction (-):
  Operation of subtracting two matrices by subtracting the corresponding entries.
- Matrix Multiplication (*):
  The product of two Matrices A and B is defined if the number of columns of A is equal to the number of rows of B. Let A = $[a_{ij}]$ be an m×n matrix and B = $[b_{jk}]$ be an n×p matrix. Then the product of the matrices A and B is the matrix C of order m×p.
- Sizeof Operation (sizeof(M)):
  Returns a Integer Array of size 2, denoting the order of row and column for the matrix respectively.
- Matrix Inverse(inv(M)):
  Return the inverse of the Matrix 'M' if valid i.e. 'M' should be a square matrix and the Determinant of Matrix should be non-zero.
- Matrix Determinant(det(M)):
  Returns the Determinant of the Matrix 'M'.
- Matrix Transpose(trans(M):
  Returns a Matrix M^T representing the Transpose of the Matrix 'M'.
- Row Addition(addr):
  Appends a Row $(a_1, a_2....a_n)$ after the last row of Matrix M[m][n] returning Matrix M' of order (m+1)×n
- Column Addition(addc):
  Appends a Column $(a_1, a_2....a_m)$ after the last column of Matrix M[m][n] returning Matrix M' of order m×(n+1)
- Row Deletion(delr):
  Deletes the last Row of Matrix M[m][n] returning Matrix M' of order (m-1)×n
- Column Deletion(delc):
  Deletes the last Column of Matrix M[m][n] returning Matrix M' of order m×(n-1).

- Syntax:

```
    Matrix M1[10][15];
    Matrix M2,M3;
    M2 = {{1, 2},{3.4},{..}... };
    M1[2][3] = 7;
    M3 = M1*M2;
    M3 = M1 + M2;
    M3 = M1 - M2;
    int size[2]=sizeof(M1);//returns int array of size 2
    float s = det(M1);
    M1 = M1 addr (1, 2, 3);  //appends a row to M1[m][n]
                              containing n-elements
    M1 = M1 addc (1, 2, 3)   //appends a Col to M1[m][n]
                              containing m-elements
    M1 delr;
    M1 delc;
    M2 = inv (M1);           //OR M2 = inv(M1)
    M2 = trans(M1);
```

2.4 Branching Statements

Standard use; as described in [5].

- If: I F
- Else: E L S E
- Switch: S W I T C H
- Case: C A S E
- Default: D E F A U L T

2.5 Loops

Standard use; as described in [5].

- While: W H I L E
- For: F O R

Syntax:

```
    for(int i=0; i<10; i++)
    {
        if(i%2==0)
        {
            s+=i;
        }
    }

```

2.6 Comments

DROOL supports single line as well as multi-line comments.

- **Single line comment**: Start with two forward slashes and end with the occurrence of a new line or end of file character. eg:

```
    //Fun Experience

```

- **Multi-line comment**: Starts with open parenthesis immediately followed by star, and end with the occurrence of a star followed by a closing parenthesis immediately. We have thoroughly tested and made multi line comments robust enough to output errors such as no closing of the comments, or end of file in comments at the lexical analysis stage and this prevents unnecessary parsing of a syntactically incorrect program. We have made use of *mode*, state in ANTLRv4 to deal with nested multiple comments and diligently ensure any complex mesh of multiline comments with equal no of opening sequence and closing sequence enclose all the characters within the outermost pair completely. eg:

```
    (* Testing the compiler
    we had (* to spend lot of time
    across the *) project *)

```

2.7 Input & Output

DROOL has two built-in functions input() and output() which facilitate all the I/O functionality of the language. Strings, matrices, integers, floats and

their compound expressions can be displayed via console output using the output() command. The input() function handles all the above mentioned data types along with vertex data type. The arguments for both the functions are comma separated expression lists. This maintains uniformity across various data types and prevents confusion of different conventions for different type of input or output. Thus makes way for fast and efficient coding practices.

- **Syntax:**
  - input method : input(explist);
  - output method : output(explist);
- **Examples:**

```
string x; int y; float z;
input(x,y,z);

output("Hello I am ",name,"!");
//Console Output: Hello I am Sandeep Kumar!

output(2+3, (2.4/3)\%5, sizeof(x));
//Console Output: 5 0 4

```

2.8 WhiteSpace

White space consists of any sequence or combination of the characters:

- blank (ASCII 32)
- \n (newline, ASCII 10)
- \f (form feed, ASCII 12)
- \r (carriage return, ASCII 13)
- \t (tab, ASCII 9)
- \v (vertical tab, ASCII 11)

Chapter 3

Distinct & Unique Features

3.1 Improvements over COOL

Our exposure to COOL last semester was a great experience to understand how it is necessary to start simple and then build upon the solid foundation you lay for the language. Therefore we did extensively study the COOL manual and the simplicity was something that we wanted to incorporate in our language. The ease of understanding and thus popularity of the language is something essential to it's design as seen by the rapid ascension of python, one of the most important factors contributing to it being the ease of using it. DROOL aims to pick the best of COOL, its simplicity to analyse, and extrapolate it to general usability while adding special notations for the user to implement their data in forms like Graphs and Matrices, also providing operations to make it simple even for a new programmer. Abstraction does not take away the freedom to implement the details themselves. Flexibility is something we noticed was missing in COOL, with structures like loops and conditional statements far too restrictive for a advanced programmer.

3.2 Distinction from C++

C++ is too complex for a new user. Too many intricacies need to be learnt, that include memory management, pointers and manual garbage collection which might be too difficult for a learner. We strip off these machine level details to provide a simple yet complete approach. We follow the philosophy of "If it ain't broke, don't fix it" for well-established and structured deci-

sions take by Bjarne Stroustrup for the design of C++, Operator Precedence and Loops and Conditional statements to name a few. But DROOL offer much more in the form of corrections to oft-criticised pitfalls of C++. We incorporate well structured object oriented approach, providing support for polymorphism and inheritance to name a few, and do this by following the JAVA approach by James Gosling which has been globally recognised as a better way to deal with Objects and Classes.

We further provide the user with the inbuilt functionality to work with Graphs and Matrices with the same ease as they would work with data types like integers and string and thus work towards providing a complete tool kit for any aspiring programmer.

Chapter 4

Compiler

4.1 Tools

This section provides a succinct description of our thought process behind selecting the tools we chose, and what we feel are its advantages to both the user and the developer in creating a compiler for DROOL. We have decided to use ANTLRv4 for the lexer and parser of DROOL's compiler and LLVM[3].

4.1.1 ANTLRv4

We decided to use ANTLRv4 (ANother Tool for Language Recognition version 4)for the purposes of lexical analysis and parsing(syntax and semantics analysis) for the following reasons:

- ANTLRv4 is categorized as an ALL(*) parser generator. An ALL(*) parser combines the simplicity, efficiency and predictability of a conventional top-downLL(k)parser and the ability to handle left recursive grammars, which is absent in other LL(k)parsers as well as earlier ANTLR versions.ANTLRv4 achieves the latter by adopting the concept of adaptive parsing.
- ANTLR also supports EBNF and thereby context-free expressions, which makes it easier to describe most languages and thus we prefer it over alternatives like Flex/Bison.
- ANTLR has the ability to generate a parser, specific to the language by using only the language grammar.It auto-generates a parse tree which

is a data structure that represents how the input is matched with the rules and patterns specified in the grammar.

- ANTLR provides multiple command-line options that can be used to view the generated parse tree in different forms:
  1. option gui: provides graphical representation of the parse tree
  2. option tree: prints out the parse tree in lisp form
  3. option trace: prints the rule name and current token on rule entry and exit.
- ANTLR also provides mechanisms for parse tree traversals:
  1. Parse Tree Listener (fully automatic): ANTLR automatically generates a ParseTreeListener (listener) subclass, which is specific to the grammar. This subclass provides entry and exit methods for every rule or sub-rule specified in the grammar.
  2. ParseTreeWalker: ANTLR also provides a ParseTreeWalker (walker) class, which is responsible for walking the parse tree and triggering ParseTreeListener method calls.
  3. Parse Tree Visitor: ANTLR provides an option called "visitor" which requests generation of ParseTreeVisitor(visitor) interface. The visitor interface provides a visit method for each rule/sub-rule of the grammar. Using visitors, we can govern the traversal of the tree by explicitly calling methods to visit children.
- ANNOTATED PARSE TREE GENERATION(AST): ANTLR auto-generates the signature for the listener and visitor methods. With the auto-generated signature, ANTLR provides parse tree Annotation. Annotating implies we can store certain information/values for each node of the parse tree. This feature helps to collect information that can be accessed across method calls or events, helping programmers to pass around data without the use of global variables. This results in lesser overhead pertaining to data storage.

4.1.2 LLVM IR

We decided to LLVM (Low Level Virtual Machine) written in C++ for implementation of the compiler back end of DROOL for the following reasons:

- LLVM provides libraries for constructing Intermediate Representation (IR) of the program that is completely independent of the source as well as the target. This implies two advantages:

    1. Since LLVM provides a library for building IR, programmer does not need to learn the specifics of LLVM IR. Triggering calls to the right IR Builder functions will do the job of generating IR for the code.
    2. Programmer does not need to understand the target machine specifics. It is taken care of by LLVM under the hood. LLVM provides code generation support for many popular CPUs (x86, ARM, MIPS). By just constructing the LLVM IR using LLVM IR Builder helper functions, we are completely bypassing assembly level machine specific details that have always been difficult to program/debug and stumped the programmers.This feature also advocates platform independence at machine level, since the same LLVM IR can be used to generate any target specific machine code supported by LLVM.

- LLVM also provides support for code optimization. It performs optimizations on the generated LLVM IR that is target independent.

- Apple's Swift language uses LLVM as its compiler framework, and Rust uses LLVM as a core component of its tool chain. Also, many compilers have an LLVM edition, such as Clang, itself a project closely allied with LLVM. Mono, the .NET implementation, has an option to compile to native code using an LLVM back end. And Kotlin, nominally a JVM language, is developing a version of the language called Kotlin Native that uses LLVM to compile to machine-native code. Nvidia used LLVM to create the Nvidia CUDA Compiler, which lets languages add native support for CUDA that compiles as part of the native code you're generating (faster), instead of being invoked through a library shipped with it (slower).

4.2 Implementation

This section describes the step by step implementation of different components of the DROOL compiler, namely the lexer, parser and semantic anal-

yser and the supporting files and structures required for implementation like the MakeFile.

4.2.1 MakeFile

The DROOL makefile allows us to produce parse trees for any our DROOL programs. When the command is run, it first generates .java files for the lexer and parser from the pre-specified source grammar i.e the .g4 files. It achieves this with the help of ANTLRv4. The syntax of the make command also requires a DROOL input file to be specified. The parser once generated is run through this program and the corresponding parse tree is displayed in a pop-up window. Errors if any are highlighted in red in the parse tree and printed on terminal as well. To run the the makefile use the below command from root directory of the project.

```
$ make f=<INPUTFILENAME>
```

4.2.2 Lexical Analyzer

A Lexical Analyzer, also known as lexer, is a program that reads the source code character by character and groups these characters to form words that belong to the programming language. Each word of the programming language is called a lexeme and these lexemes are further classified into categories. A lexeme and its classification together as one unit is called a token and this process is called tokenisation.

4.2.3 Parser

Parser is the unit of the compiler which is responsible for checking the language syntax. In a very similar fashion to that of a Lexer, parsing rules are specified by the user, which represent all the possible constructs that the programming language supports. The programming language construct is the assembly of tokens in a particular order which represents a syntactically correct statement of the language.Parser receives a sequence of tokens as the input from the lexer, which it then scans to check if the order of tokens follows the language syntax.

4.2.4 Sample Parse Trees

```
int main()
{
    int i = 0, sum = 0;
    while (i < 5)
    {
        sum += i++;
    }
    output("The sum is:",sum);
}
```

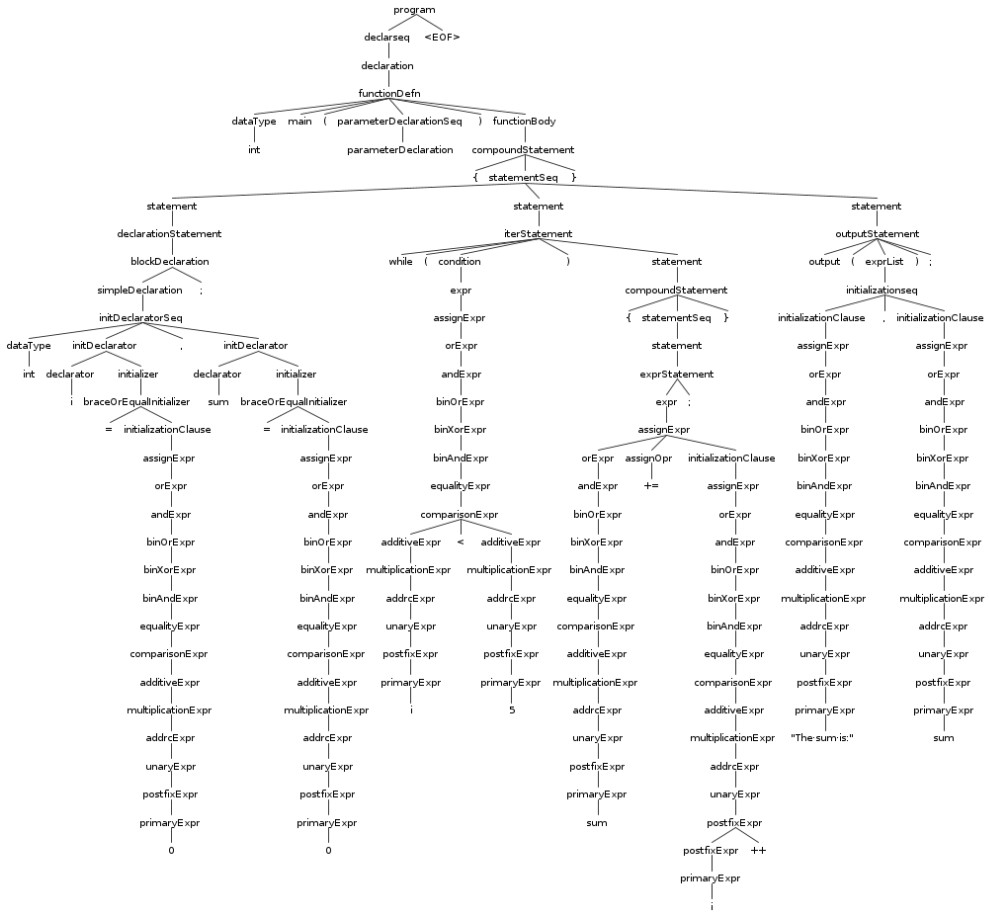

Figure 4.1: While Loop Parse Tree

```
int main()
{
  int x = 9;
  if(x == 10)
    output("x is 10");
  else
    output("x is not 10");
}
```

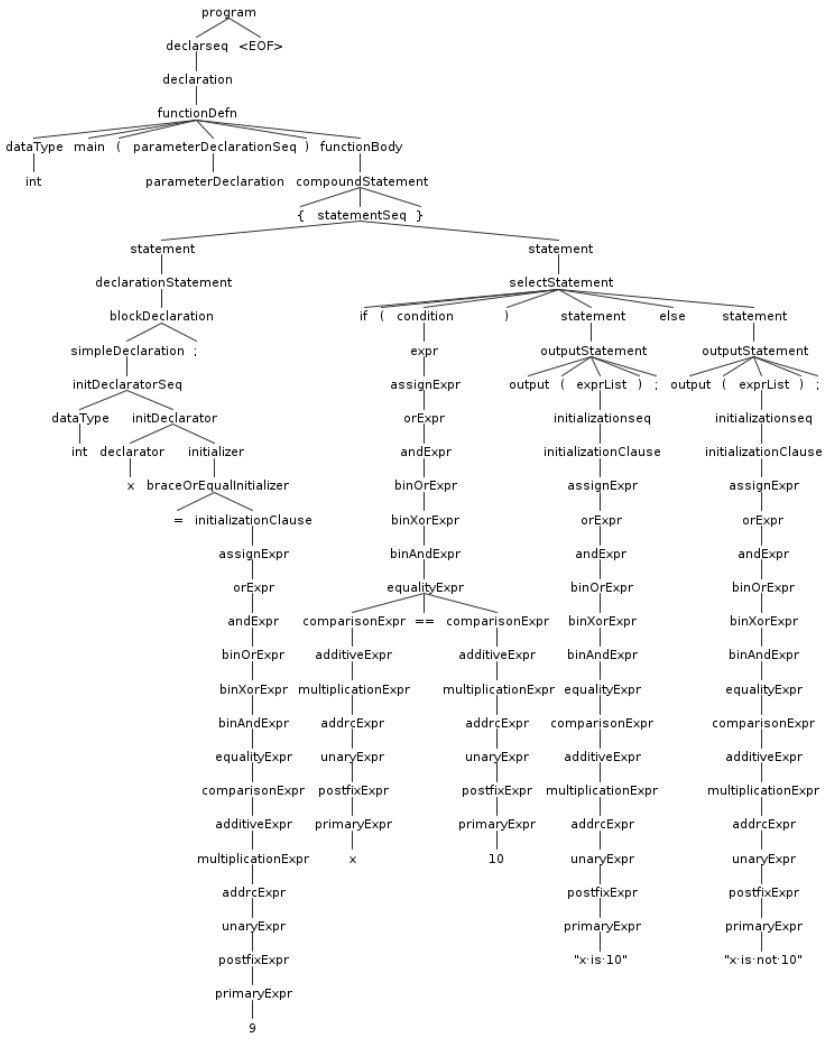

Figure 4.2: If Statement Parse Tree

```
int main()
{
// Vetices
Vertex v1 = "v1", v2 = "v2", v3 = "v3";

Graph G;
// Inserting Vertices into Graph G
G << v1 << v2 << v3;

// Inserting edges into graph G
G << (v1, v2, 2)<<(v1, v3);
}
```

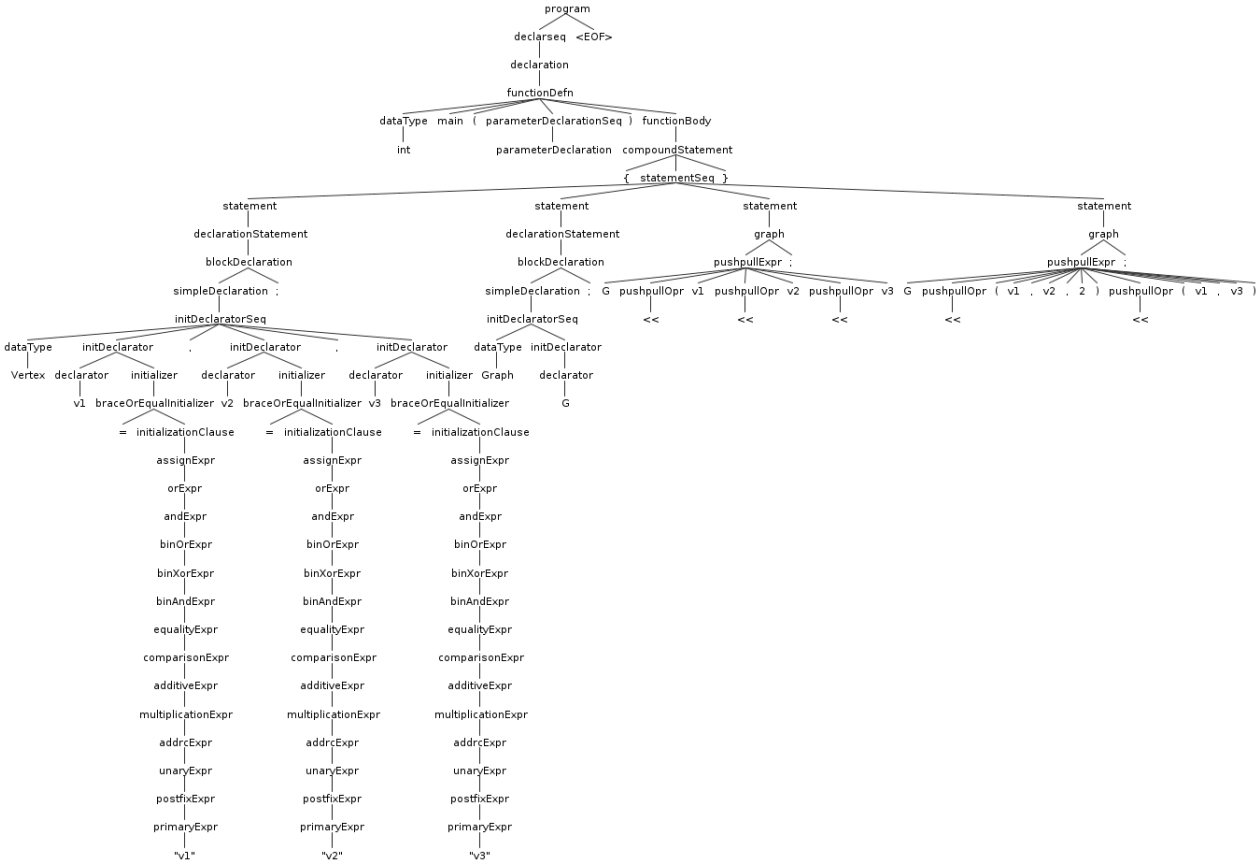

Figure 4.3: Graph Basic Parse Tree

```
int main()
```

```
{
    Matrix m1[2][3] = {{1,2,3},{4,5,6}};
    output(det(m1));
    output(inv(m1));
    output(trans(m1));
    m1 delr;
}
```

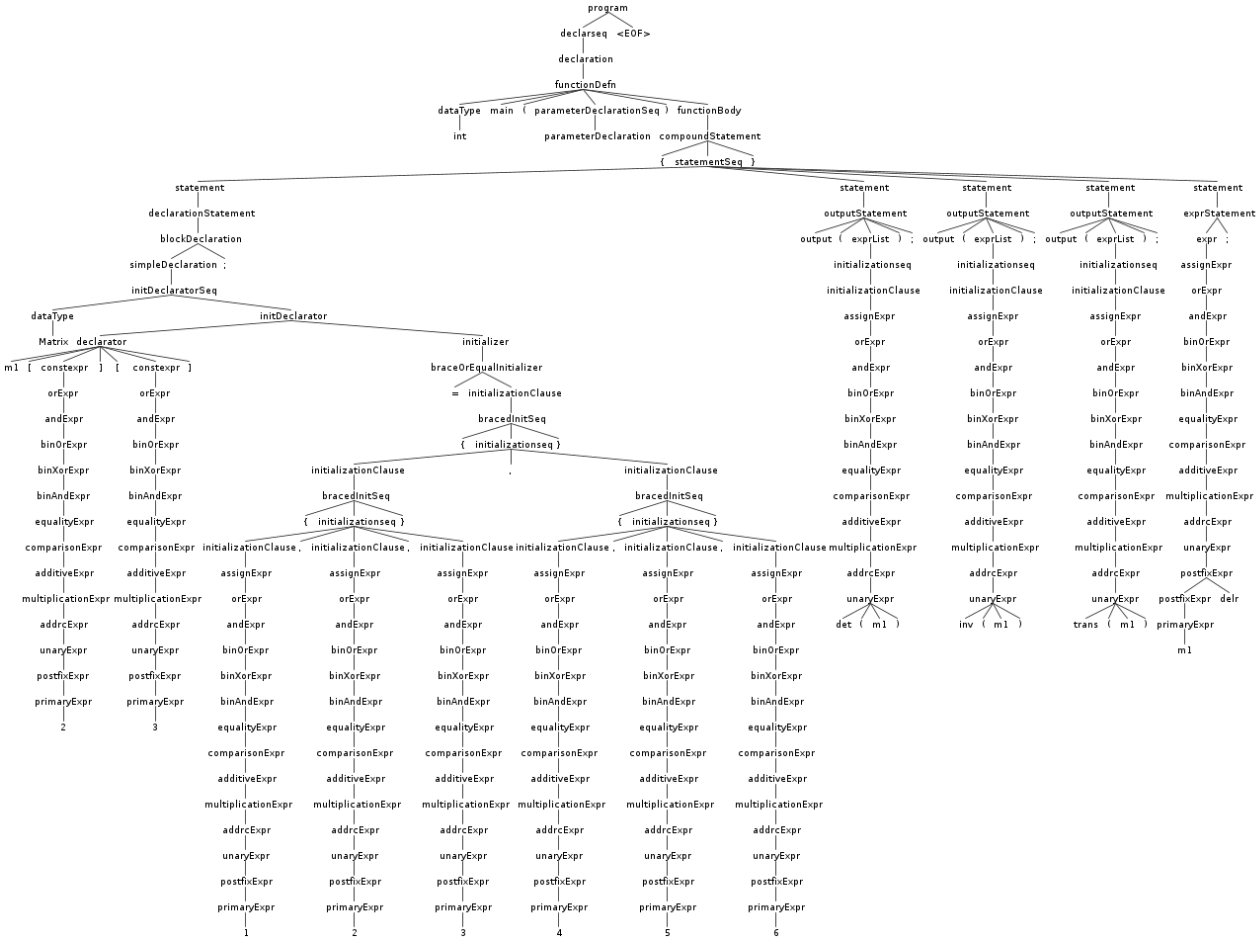

Figure 4.4: Matrix Basic Parse Tree

4.2.5 Semantic Analyzer

To be added in due time.

4.2.6 Code Generator

To be added in due time.

4.3 Optimisation

To be added in due time.

Chapter 5

Appendix

To be added in due time.

Bibliography

[1] Alexander Aiken. "Cool: A Portable Project for Teaching Compiler Construction". In: *SIGPLAN Not.* 31.7 (July 1996), 19–24. ISSN: 0362-1340. DOI: 10.1145/381841.381847. URL: https://doi.org/10.1145/381841.381847.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[3] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis &; Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[4] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999.

[5] Bjarne Stroustrup. *The C++ programming language (3. ed.)* Addison-Wesley-Longman, 1997. ISBN: 978-0-201-88954-3.