Develop an interactive calculator by developing following components: 3. Front-end interpreter 4. Back-end interpreter These components will convert the user input into the assembl...

Preprint · March 2019

DOI: 10.13140/RG.2.2.18662.47688

CITATIONS
0

READS
1,170

1 author:

Sunil Mandhan
Hitachi, Ltd.

7 PUBLICATIONS **4** CITATIONS

SEE PROFILE

1. Problem/Purpose

Develop an interactive calculator by developing following components:

1. Front-end interpreter
2. Back-end interpreter

These components will convert the user input into the assembly language used by the Calculator Virtual Machine (CVM). All the errors in the user input shall also be reported.

CVM is provided along with the problem statement. CVM also provides the provision to execute the resulting assembly language.

3. Assumptions

1. User of the developed interactive calculator shall provide valid inputs.
2. It is assumed that user is having enough knowledge to work with eclipse ide for java, opening and running java projects from the ide.
3. Implementation of the interpreter is done on java 1.6 versions.
4. Interpreter, calculator, and compiler are used synonymously in the current document scope.

4. Results

Interpreter was developed with the following components.

1. Front end
   a. **User interface** – main program to integrate all components.
   b. **Lexer** – performs the lexical analysis of the given user input.
   c. **Parser** – performs the parsing of the token according to the grammar production, performs the optimization of the generated abstract syntax tree (AST) and produces symbol table also.

2. Back end
   a. **Assembler** – it takes the symbol table and AST from the parser and produces the assembly language instructions for the CVM.
   b. **Virtual Machine Interface** – this is a wrapper over the CVM.

   Optimization is implemented in the parser.

5. conclusions

This assignment introduced the team to the techniques used in implementing a compiler like how the validity of user input is checked, how the user input is converted into assembly language.

It was a challenging assignment where team got some roadblocks while implementing individual grammar productions. Also the optimization of the AST was a big challenge, how to do it and where to do it, these questions were answered with lots of thinking and efforts.

We learnt good internal details about the compiler implementation.

Table of Contents

1. LIST OF FIGURES, TABLES, GRAPHS

2. NOMENCLATURE

The following acronyms and abbreviations are used throughout this document and are defined here for convenience.

AST	Abstract syntax tree
IR	Intermediate representation

3. PROBLEM STATEMENT

Develop an interactive calculator by developing following components:

3. Front-end interpreter
4. Back-end interpreter

These components will convert the user input into the assembly language used by the Calculator Virtual Machine (CVM). All the errors in the user input shall also be reported.

CVM is provided along with the problem statement. CVM also provides the provision to execute the resulting assembly language.

Please refer [2] for the full assignment and related details.

4. RESULT

Starting from the top-level software architecture and program flow, different deliverables are discussed in good detail, followed by individual and teamwork items.

An interpreter program was implemented in the java programming language. This program does the following things.

- It reads the user input from the console.
- It breaks the input into individual tokens according to the grammar rules [2].
- Then tokens are parsed according to the grammar rules.
- During parsing AST is populated and symbol table is created by the parser.
- After the creation of the AST also called IR, optimization is performed on the AST to reduce the number of CVM's assembly language instructions.
- After the optimization AST is handed over to the assembler component. Symbol table is also passed. Assembler then translates the IR with the help of symbol table into the CVM's assembly language.
- This conversion into assembly language is done with the help of rules given for translation, please refer [2] for the exact rules.
- After the translation, assembly instructions are passed to the virtual machine interface component which internally passes it to the CVM for the execution.
- CVM executes the instructions and results are shown on the console.

4.1 Detailed design

In the detailed design, first project organization shall be discussed, after that top level architecture and program flow diagrams shall be discussed.

At the end IR representation, symbol table structures and constant folding are covered and discussed.

Project organization – as seen in eclipse ide

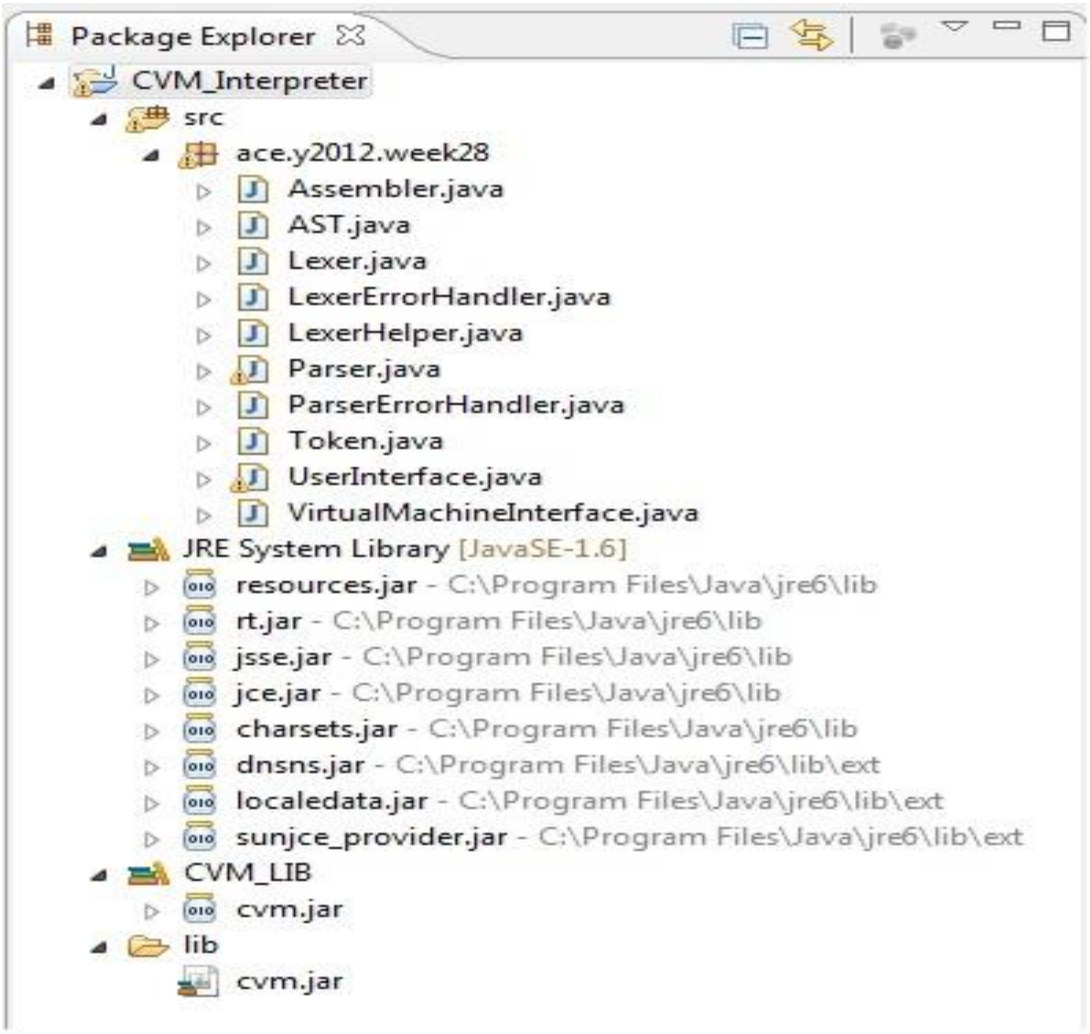

Figure 1 – Project organization

Top level architecture – basic block diagram

Please refer to the Figure 2 for the top level architectural diagram for the implemented interpreter. Each component in the diagram below is implemented in its own file, please refer to the Figure 1 for details.

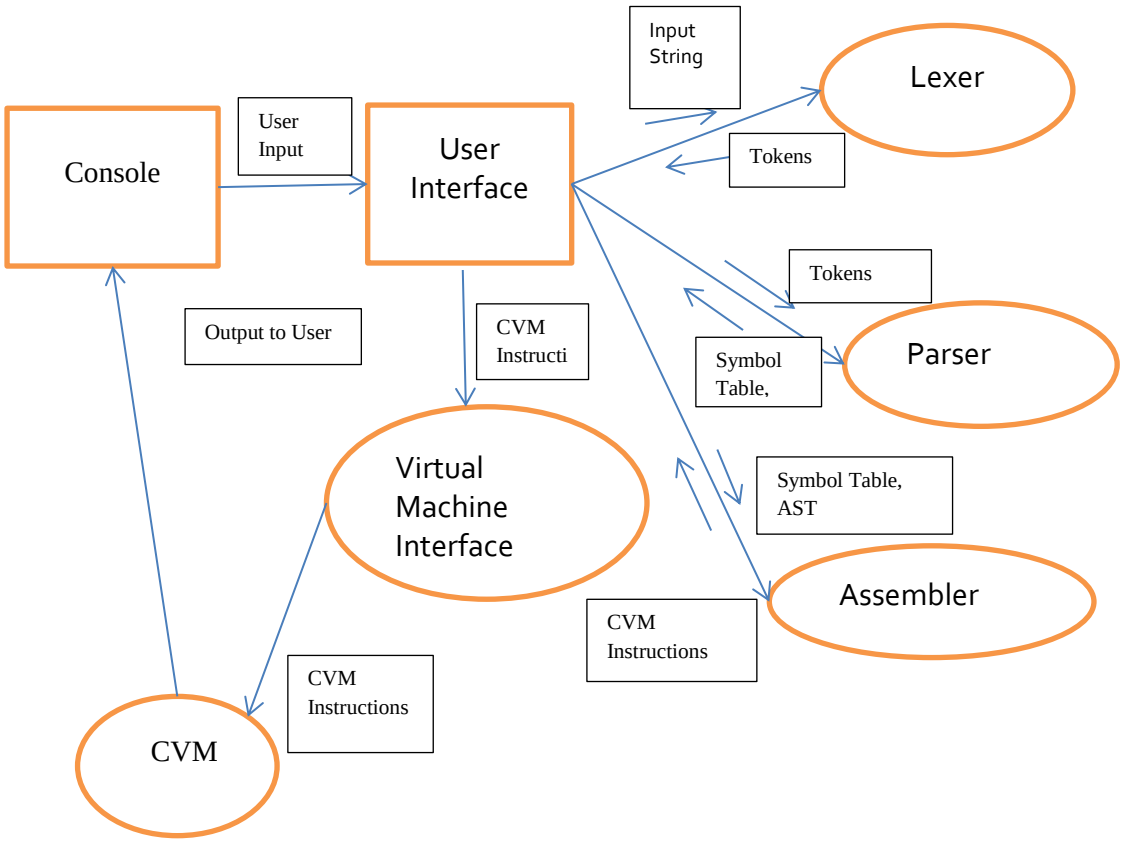

Figure 2 – Top level architecture of Interpreter

Flow chart – Lexer

Flow chart of how a lexer works is given in Figure 3 and Figure 4 below. These two flow charts are part of a single flow chart.

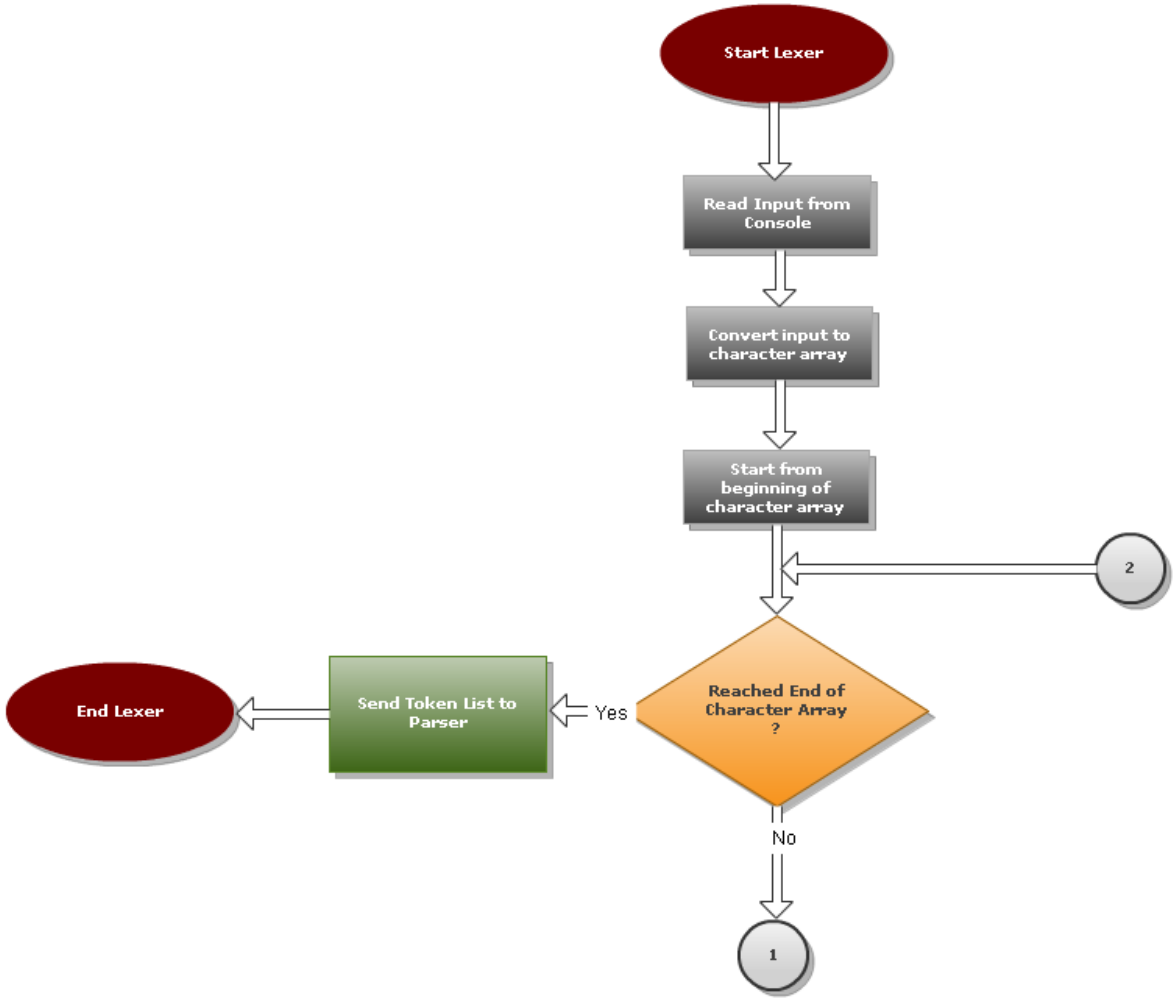

Figure 3 – Lexer flow chart part 1

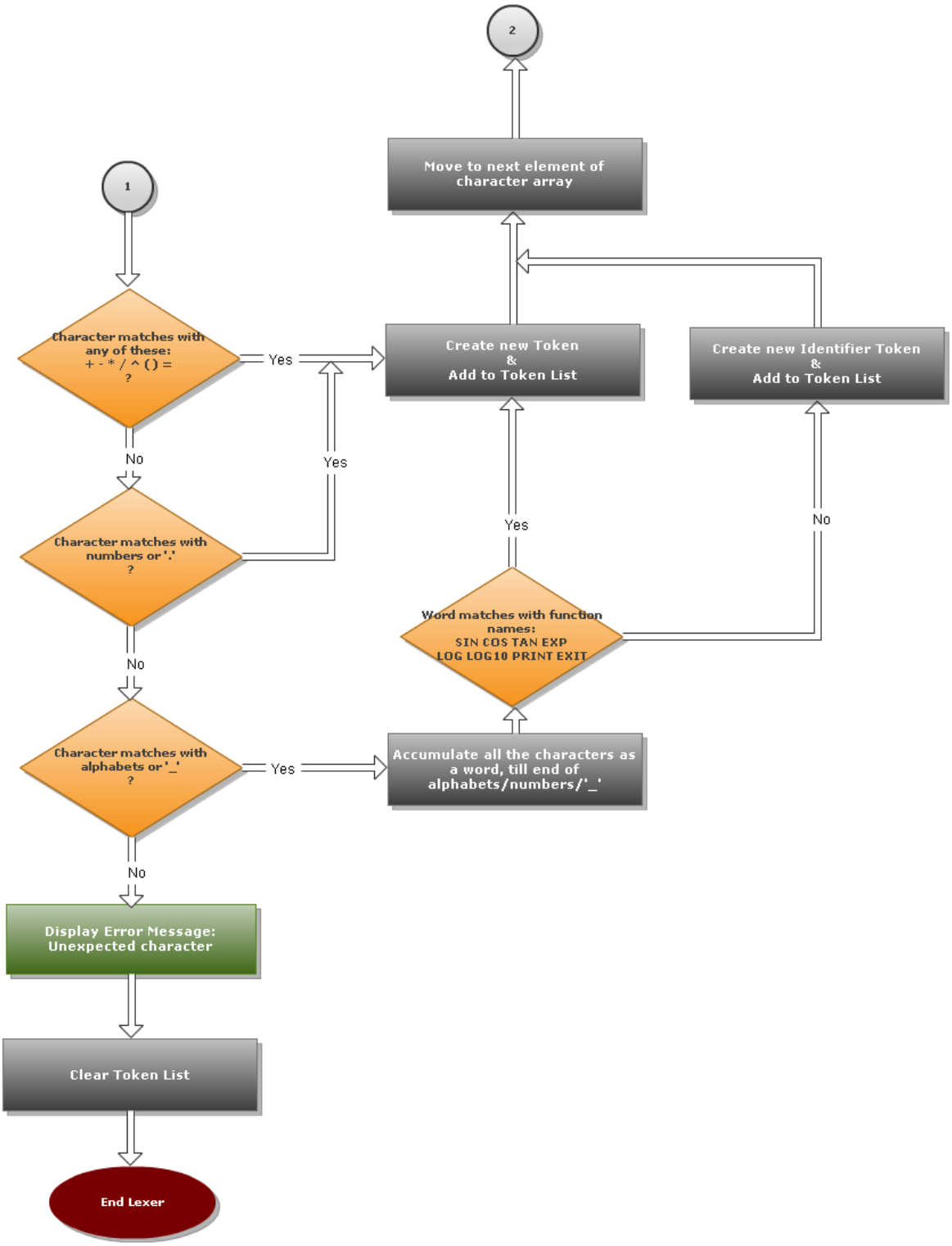

Figure 4 – Lexer flow chart part 1

Parser – How it works

Each production has its own method. Parser performs the recursive descent parsing; it generates symbol table and AST. It also catches syntactic/semantic errors. At the end it does constant folding (please refer Figure 6) to optimize the AST.

IR diagrams

AST is used as the intermediate representation to be used by the assembler. This IR is produced by the parser along with the symbol table.

Example is given below (Figure 5) to show how IR looks like. Colored circles denote AST nodes created during parsing Colored Square denotes Symbol table additions

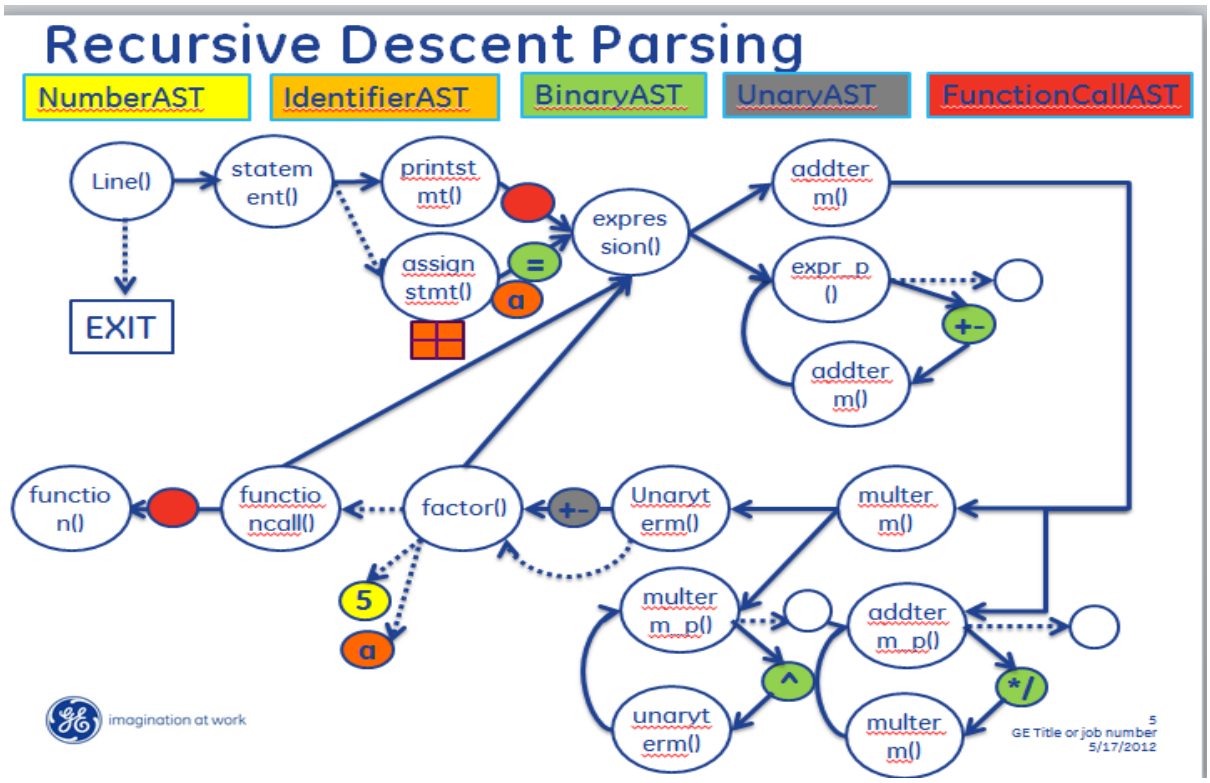

Figure 5 – Recursive descendent parsing – how it works

Constant folding

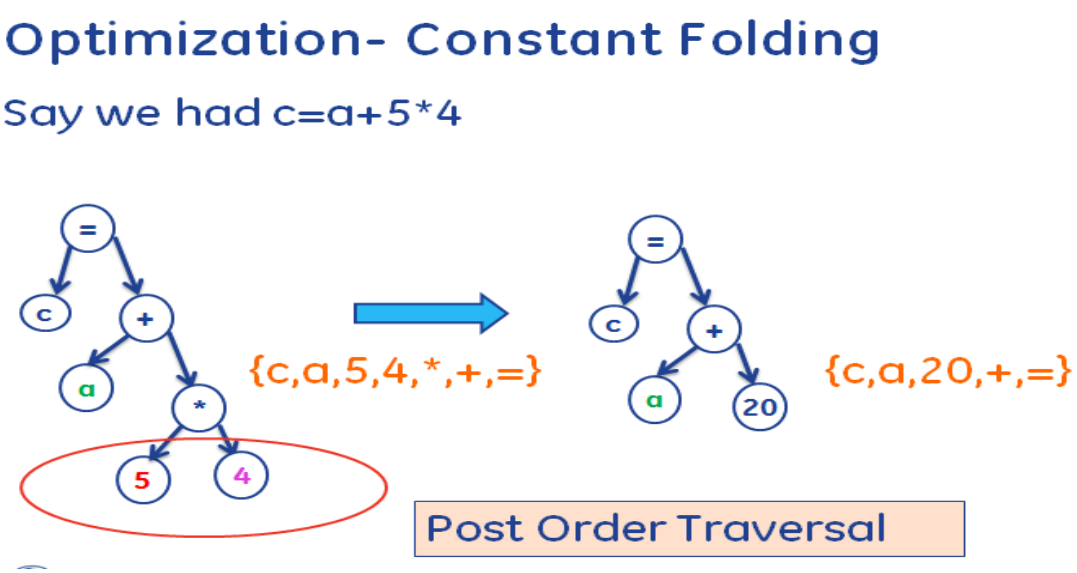

Figure 6 – Constant folding – optimization on AST by parser

Symbol table structure

Symbol table is used in the current implementation of the interpreter for keeping track of the different identifiers (also called variables).

Symbol table stores only two fields in the current implementation:

1. Identifier name
2. Memory location index

Below is given the code snippet for the symbol table declaration. The implementation can be found in Parser.java file.

Map<String, Integer> *symbolTable* = new HashMap<String, Integer>();

A sample symbol table is shown below in Table 1. This tables stores the information about 2 identifiers, a and b.

Memory location index is stored in the symbol table because CVM also deals with memory location based on index starting from 0 and incrementing 1. So, it made sense to club this information inside the symbol table.

Table 1 – Symbol table structure with sample data

Identifier Name	**Memory location index**
a	0
b	1

Flow chart – Assembler

Below in Figure 7, is shown how assembler works. For specific example, please refer to the Figure 8.

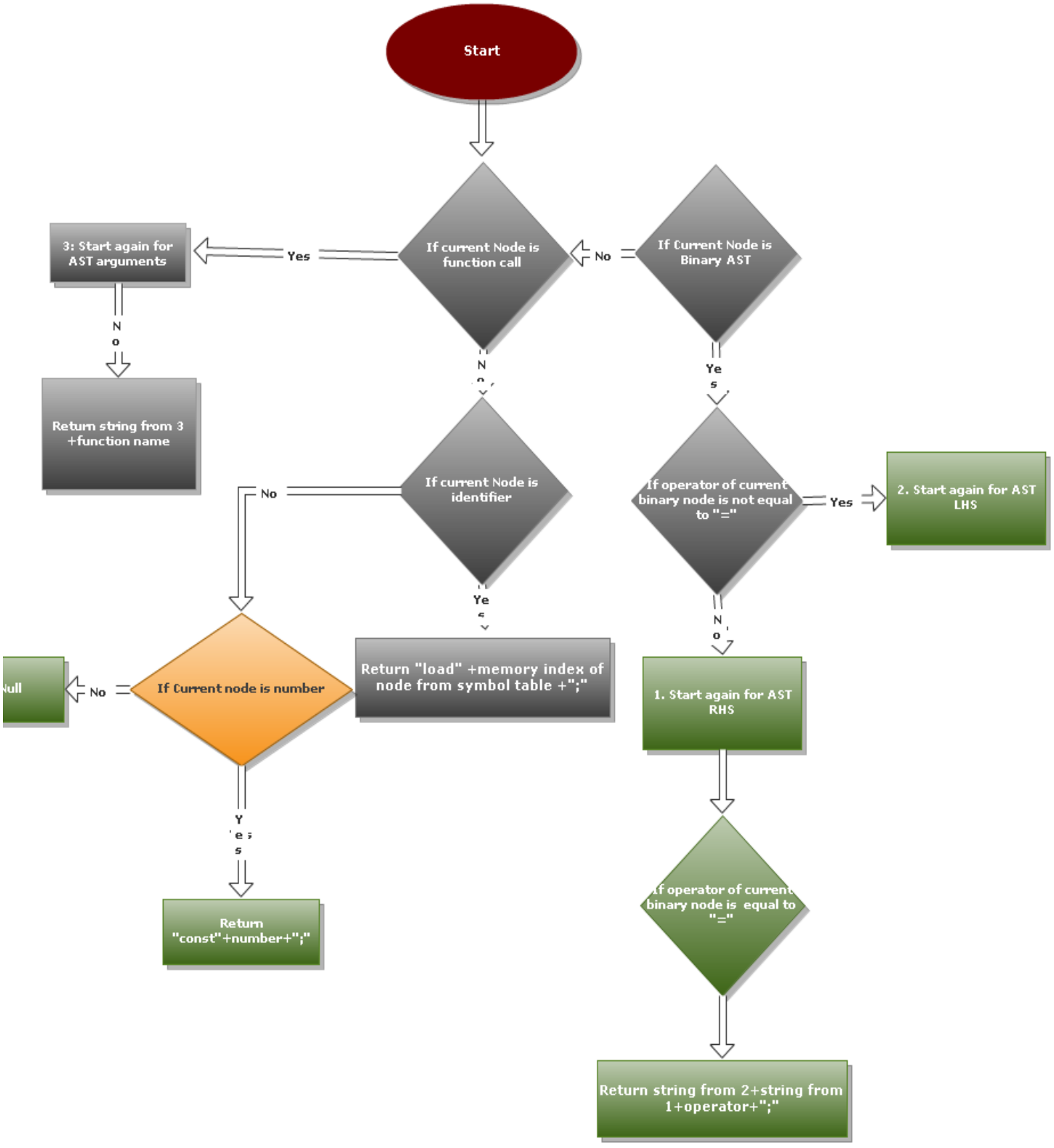

Figure 7 – Assembler flow chart – how assembler work

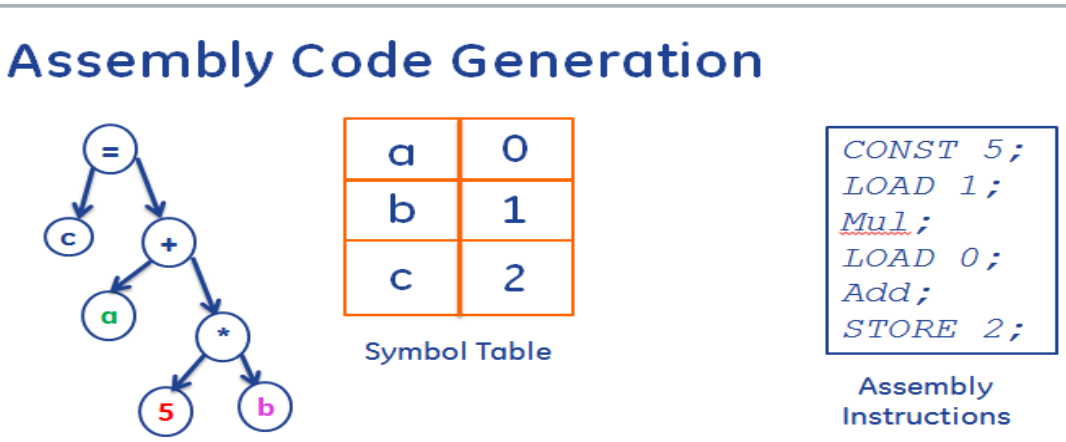

Figure 8 – Assembly code generation example

4.2 Work distribution

Individual Work Distribution

This assignment is done by team collaboration. The source code and all the artifacts are individually written and produced.

Common Team Items

The team collectively worked on the following areas:

1. Understanding the problem statement and the artifacts required.
2. Requirements analysis
3. Test scenarios analysis
4. Design of Program.
5. Discussion on results – for comparison with each other.

4.3 Deliverables

Source code

Please refer to the appendix A for the source code. The instructions to build the source code are also provided with the source code.

Executable object code

Please refer to the appendix A for the executable.

Source code building instructions

The instructions can be found in appendix A.

4.4 Challenges faced

Few challenges the team faced, from starting with the problem statement to the solution, are given below:

1. **How to implement the parser catering the grammar productions.**
   This was overcome by implementing a function for each grammar production and calling the other production calls from the implemented function call.

2. **How the translation to assembly language of CVM will be done?**
   The AST produced by the parser was traversed post order and each node was replaced by their corresponding instructions as documented in [2].

3. **How the syntax and compilation error will be implemented and shown to the user.**
   To make it modular, lexer implemented its own error handler and shows error when invalid token is encountered.
   Parser implemented its own error handler and shows error when invalid statement is not parsed.

4. **Should the + and – operator given with the number like +2 and -0.5 should be treated along with the number or separate tokens.**
   After reading the grammar productions carefully, it was clear that + and – shall be different tokens and shall not be clubbed with the numbers. For this kind of input UnaryAST shall be constructed.

5. DECISIONS AND ALTERNATIVE DECISIONS

5.1 Constant folding in front end (parser)

Constant folding was done in the front end because it was logical that we just give an AST and symbol table to the assembler and it only translates it to assembly language.
It could have been done in the assembler also but purpose of putting it in the parser was to make the components cohesive and modular.
Assembler task is just to translate the AST into assembly language instructions so it should only do this not the other things like optimization.

5.2 Allocating memory location or reference in front end (parser)

Assigning a memory reference was a tricky decision. As the memory model of CVM only requires the memory reference that starts with 0 and incrementing with 1 with every new variable/identifier definition. Team thought it would be more efficient to do it when storing new variables in the symbol table and incrementing a number every time it is done and also to store this number in the symbol table.

6. DISCUSSION

The aim of the assignment was to implement the interpreter that will perform the lexical analysis, parsing, conversion to assembly language, and passing the converted assembly language to the CVM virtual machine.

This exercise and experience to solve the assignment was challenging for the team.

Results can be found in section 4.

All the artifacts produced during implementation and after application testing are discussed in section 4.3.

Challenges faced by the team and those were solved are given in section 4.4.

Design decisions taken are discussed in section 4.5.

Decisions that were took are discussed in section 5.

6.1 Assumption Validation

1. **User of the developed interactive calculator shall provide valid inputs.**
   User need to use it for calculator purpose, inputs should be provided according to the use.

2. **It is assumed that user is having enough knowledge to work with eclipse ide for java, opening and running java projects from the ide.**
   This assumption states that there is little help provided to work with java and eclipse in the current document and related material.

3. **Implementation of the interpreter is done on java 1.6 versions.**
   Implementation is done on java version 1.6, for making it work on higher version, user needs to set up the project properties again.

6.2 Limitations of the solution

1. **Operator precedence in case of consecutive same operator**
   For example if user inputs a = 2^2^3 then the result should be 64 but this implementation outputs 256. This happens only in case of ^ and / operator.
   It takes precedence from the right instead of left.

APPENDIX A – SOURCE CODE

1. **Source Code**

Please refer to the supplied file given below for the complete source code.

Source_Code.zip

2. **Executable**

   Please refer to the supplied file given below for the executable.

Executable.zip

3. **Instructions for building source code**

Please refer to the supplied file given below for the complete source code.

ReadMe_Source_Code_Building.txt