SER 502 Team 7 Milestone 2

Team members

Smita Hirve	1210393740
Ruthvik Arya Manjunatha	1211286528
Susmitha Kistamsetty	1211588271
Nishant Jagadeesan	1210319341

KODO - A Simplified Programming Language

High Level Language File (.kd) extension
Low Level language file (.kin) extension

- ## Tools Used

The tools which we would be using to build the compiler and the interpreter are:-

1. ANTLR4 - Lexical analysis and Parsing
2. Generation of Intermediate code - Java and ANTLR4
3. Runtime - Java

An Antlr4 project would be created in the Eclipse IDE.

- ## Language Description

Name Inspiration - We wanted a simple name for the language which is easy to pronounce and remember. 'Kodo' is a japanese word for code and sounds similar to code. So, we decided to name our language as Kodo.

Design Inspiration - The design of this language is inspired from Visual Basic and Java. There are no End of Line symbols used. The design is simple as the syntax is easy to understand.

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. The choice of antlr was obvious for us from the start. Its use with Java, the programming language all of us are familiar and comfortable with, and its learning curve which is a a lot less steeper compared to javacc.

Antlr plugin for eclipse, and relative easy of use and setup made it an easy pick for the team to go with antlr. Antlr is a parser generator capable of converting our grammar file into a parse tree.
The parse tree generated and the associated files generated by antlr build system (grammar.tokens and grammarLexer.tokens) are used to generate our .kin file (the intermediate language). The Grammar file itself is written using a BNF format which is the appropriate syntax for Antlr. The Grammar file removes the concept of Left Recursion with the help of ordering the rules. The simplest of rules are at the bottom and most complicated ones are on top.

Antlr also provides easy methods for tree walk and we use this for parsing and extracting elements from the parse tree. We employ a bottom-up parsing technique to ensure complicated expressions are parsed in the right order and to manage the temporary variables used in the intermediate language. More details about the intermediate code generation are provided further below;

We use simple data structures - hash map for storing and referencing tokens, a symbol table and a stack to maintain precedence and temporary variable usage when breaking down complicated expressions in our kodo code.

- **Sample Program and its explanation**

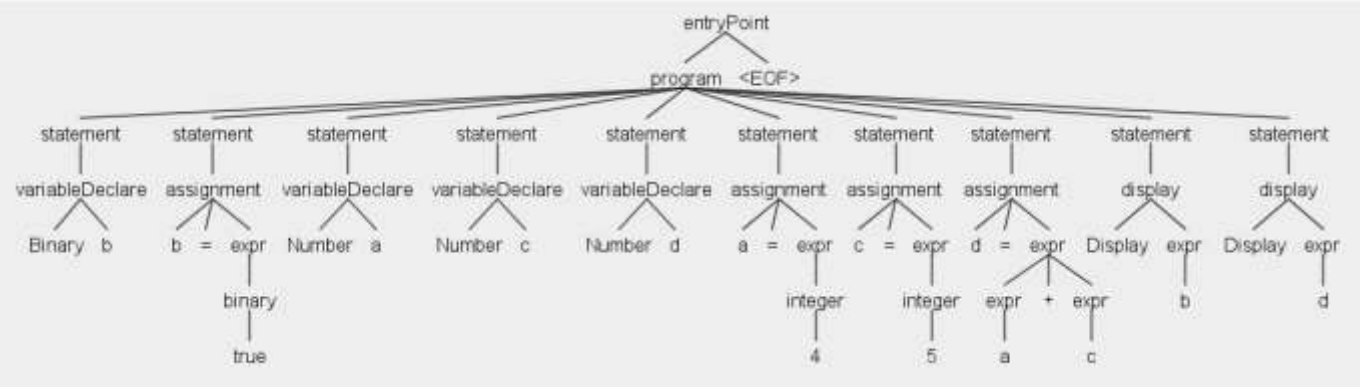
Kodo_Prog1.kd

```
1 Binary b
2 b = true
3 Number a
4 Number c
5 Number d
6 a = 4
7 c = 5
8 d = a + c
9 Display b
10 Display d
```

The example 1 shows a typical Kodo_Prog1 program:

Line 1: A binary datatype variable - b is declared
Line 2 'b' variable is assigned the value as 'true'
Line 3,4,5: Three integer type variables are declared
Line 6: The variable 'a' is assigned value as 4
Line 7: The variable 'c' is assigned value as 5
Line 8: The sum of 'a' and 'c' is stored in variable 'd'
Line 9: The value of the binary variable 'b' is printed on the screen
Line 10: The value of the integer variable 'd' is printed on the screen.

- **Parse Tree:**



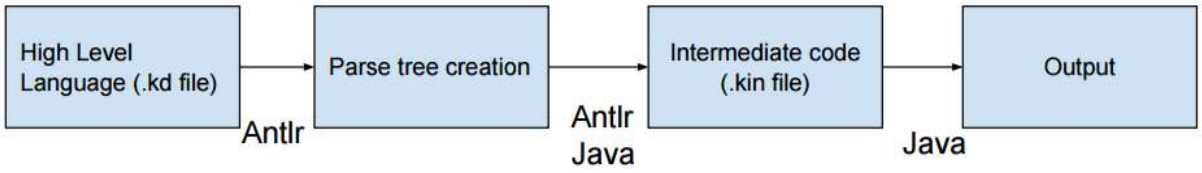
- **Intermediate code**

The intermediate code for the above sample program is as follows:

```
1   Dec b Bin
2   Assign b true
3   Dec a Num
4   Dec c Num
5   Dec d Num
6   Assign a 4
7   Assign c 5
8   Add a c
9   Assign d a
10  Disp b
11  Disp d
```

- **Language Features Constructs**
  - Arithmetic and Logical operators
  - Integer and Boolean data types
  - Assignment statement
  - Decision statement - (If-then-else)
  - Control flow statement - While loop

- **Program Flow**



The High Level language file would be with the (.kd) extension and it will be used parsed and converted to the intermediate code file which would be with the (.kin) extension.

- **Constructs**

Data Types	Number, Binary
Arithmetic Operators	+, -, *, /,%
Assignment Operator	=, DEC
Logical Operator	& ! |
Print Statement	Display
Comparison Operators	==,>,<,>=,<=
Branching	If, Elseif, Else
Looping	While

- **Grammar of Kodo**

```
/** This is a grammar file for Kodo language
@author : Team 7 SER 502 class */


grammar KodoGrammar;

entryPoint : program EOF;

//parser rules

program: statement*;

statement :   assignment
            | variableDeclare
            | display
            | whileblockstatement
            | ifblockstatement
            ;

assignment : Variable Assign expr ;

display : Display expr;

variableDeclare :   Number  Variable                    #numvariable
                    |Binary Variable                    #binvariable
                    ;

whileblockstatement : While expr  program End;

ifblockstatement : ifstatement (elseifstatement)* elsestatement End;
elseifstatement : Elseif expr program;
elsestatement : Else program;
ifstatement : If expr program ;
```

```
34
35 expr :         expr Multiplication expr              #mulexpr
36              | expr Division expr                    #divexpr
37              | expr Modulus expr                     #modexpr
38              | expr Addition expr                    #addexpr
39              | expr Substraction expr                #subexpr
40              | expr GreaterThan expr                 #gtexpr
41              | expr LessThan expr                    #ltexpr
42              | expr LTEqual expr                     #lteexpr
43              | expr GTEqual expr                     #gteexpr
44              | expr Equal expr                       #eqexpr
45              | expr NEqual expr                      #neqexpr
46              | expr And expr                         #andexpr
47              | expr Or expr                          #orpexpr
48              | expr Not expr                         #notexpr
49              | binary                                #binaryexpr
50              | integer                               #intexpr
51              | String                                #strexpr
52              | Variable                              #varexpr
53             ;
54
55 binary : True | False;
56 integer : Integer;
57
```

```
57
58 //Lexer rules
59
60 While :             'While';
61 If :                'If';
62 Else :              'Else';
63 Elseif :            'ElseIf';
64 End :               'End';
65 Display :           'Display';
66 Number :            'Number';
67 Binary :            'Binary';
68 Equal :             '==';
69 Assign :            '=';
70 Addition :          '+';
71 Substraction :      '-';
72 Multiplication :    '*';
73 Division :          '/';
74 Modulus :           '%';
75 GreaterThan :       '>';
76 LessThan :          '<';
77 GTEqual :           '>=';
78 LTEqual :           '<=';
79 NEqual :            '!=';
80 And :               '&';
81 Or :                '|';
82 Not :               '!' ;
83 True:               'true'|'True';
84 False :             'false'|'False';
85
86 Variable : [A-Za-z][a-zA-Z0-9_]*;
87
88 String : ["] (~["\r\n] | '\\\\' | '\\"')* ["];
89
90 Whitespace: [ \t\n\r] -> skip;
91
92 Integer: [0-9] Digit* | '0';
93
94 fragment Digit : [0-9];
```

- **References:**

1. http://www.theendian.com/blog/antlr-4-lexer-parser-and-listener-with-example-grammar/
2. http://stackoverflow.com/questions/30128961/trouble-setting-up-antlr-4-ide-on-eclipse-luna-4-4
3. http://web.mit.edu/dmaze/school/6.824/antlr-2.7.0/doc/lexer.html