SER-502 SPARKY- OUR NEW PROGRAMMING LANGUAGE

TEAM 25

TEAM 25 MEMBERS

(IN ASCENDING ORDER)

1. MAYANK BATRA ID- 1216214610
2. RAHUL JAIN ID-1218636185
3. RISHIKA BERA ID- 1217137766
4. SAYALI TANAWADE ID-1217818381

PRE-REQUISITES

- 1. JDK Version 1.8 and above
- 2. ANTLR 4.
- 3. Operating system- Windows
- 4. IDE- ECLIPSE
- 5. Any text editor like notepad ++
- 6. GITHUB – a version control software
- 7. ANT – java build tool

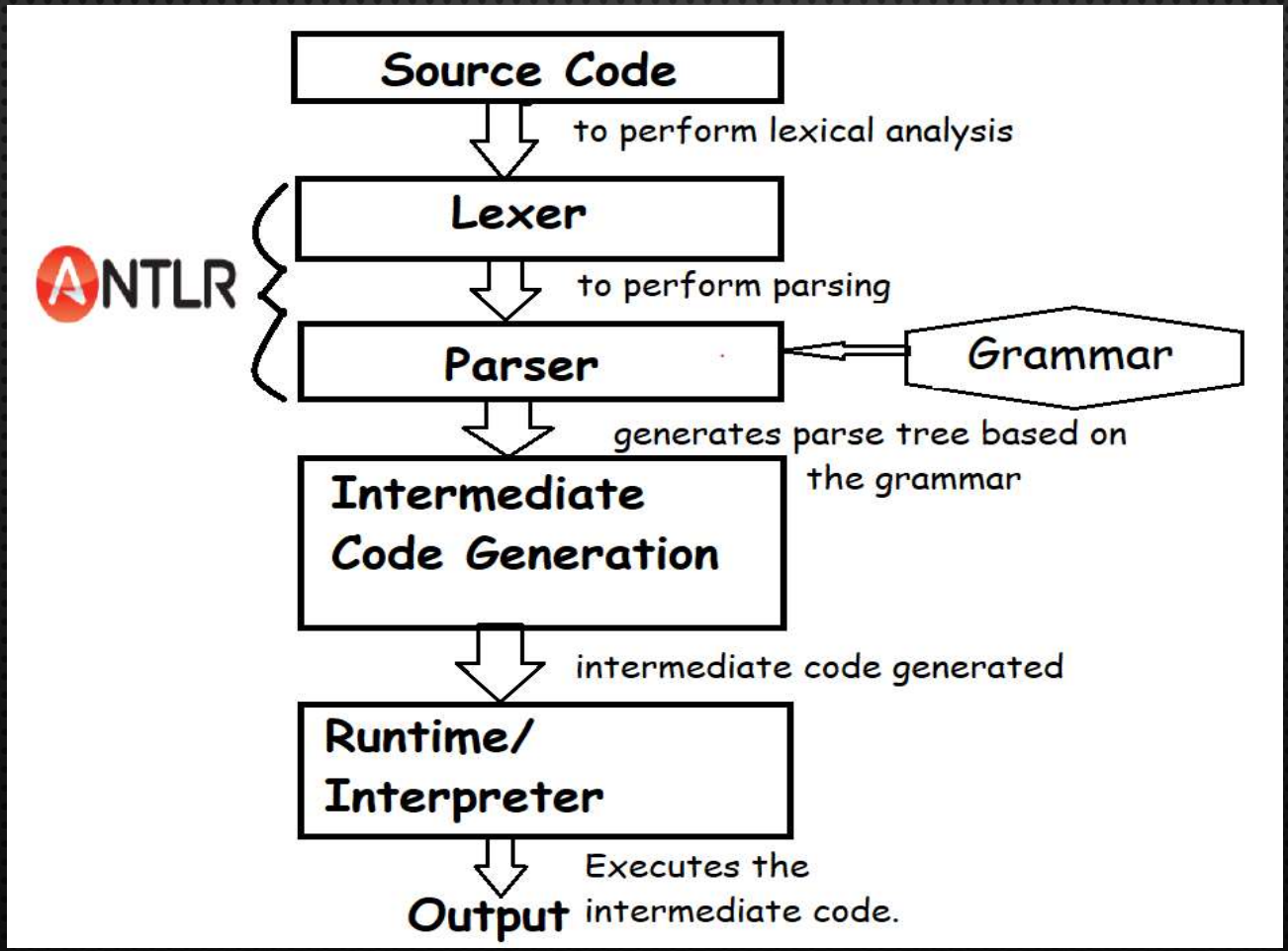
WHY SHOULD YOU USE SPARKY?

- SPARKY IS A SIMPLE PROGRAMMING LANGUAGE INSPIRED BY JAVA.
- IT IS EASY TO CODE BECAUSE OF THE FAMILIAR KEYWORDS USED IN SPARKY.

CREATION OF THE LANGUAGE IS BROKEN DOWN INTO FOLLOWING STAGES:-

- CREATION OF GRAMMAR FILE (IN .G4)
- GENERATION OF LEXERS AND PARSERS WITH THE HELP OF ANTLR . LEXER TAKES TEXT AND RETURNS TOKENS, PARSERS TAKES TOKENS AND RETURNS ABSTRACT SYNTAX TREE(AST).
- DESIGNING OF COMPILER – COMPILER TAKES AST AND RETURNS INTERMEDIATE CODE.
- DESIGNING OF RUNTIME

LANGUAGE DESIGN



OVERVIEW OF THE BASIC COMPONENTS USED-

- 1. LEXER- A LEXER IS A SOFTWARE PROGRAM THAT PERFORMS LEXICAL ANALYSIS. LEXICAL ANALYSIS IS THE PROCESS OF SEPARATING A STREAM OF CHARACTERS INTO DIFFERENT WORDS, WHICH IN COMPUTER SCIENCE WE CALL 'TOKENS'. THE **LEXER**, ALSO CALLED LEXICAL ANALYZER OR TOKENIZER, IS A PROGRAM THAT BREAKS DOWN THE INPUT SOURCE CODE INTO A SEQUENCE OF LEXEMES. IT READS THE INPUT SOURCE CODE CHARACTER BY CHARACTER, RECOGNIZES THE LEXEMES AND OUTPUTS A SEQUENCE OF TOKENS DESCRIBING THE LEXEMES
- 2. PARSER- A PARSER GOES ONE LEVEL FURTHER THAN THE LEXER AND TAKES THE TOKENS PRODUCED BY THE LEXER AND TRIES TO DETERMINE IF PROPER SENTENCES HAVE BEEN FORMED. PARSERS WORK AT THE GRAMMATICAL LEVEL, LEXERS WORK AT THE WORD LEVEL. **PARSING** IS THE PROCESS OF ANALYZING TEXT MADE OF A SEQUENCE OF TOKENS TO DETERMINE ITS GRAMMATICAL STRUCTURE WITH RESPECT TO A GIVEN GRAMMAR. THE **PARSER** THEN BUILDS A PARSER TREE OR ABSTRACT SYNTAX TREE BASED ON THE TOKENS.

CONTINUED…

- 3. INTERMEDIATE CODE GENERATION- **INTERMEDIATE CODE** GENERATOR RECEIVES INPUT FROM ITS PREDECESSOR PHASE IN THE FORM OF AN ANNOTATED SYNTAX TREE. THAT SYNTAX TREE THEN CAN BE CONVERTED INTO A LINEAR REPRESENTATION. BASICALLY WE ARE GENERATING AN INTERMEDIATE CODE SO THAT WE DON'T HAVE TO NEED A FULL NEW COMPILER EACH TIME WE RUN THE CODE IN A DIFFERENT MACHINE.

THE INTERMEDIATE CODE IS GENERATED BY IMPLEMENTING A CUSTOMIZED LISTENER CLASSES WHICH EXTENDS THE BASE LISTENER CLASSES WHICH IS GENERATED BY ANTLR.

THE INTERMEDIATE CODE IS WRITTEN IN JAVA.

INTERMEDIATE CODE GENERATION OPERATIONS-

- DECLARE
- STORE
- ADD
- PUSH
- JUMP
- WHILEBEGIN, WHILEEND
- GET
- PRINT
- OPERATOR
- COMPARE_OPERATOR
- IFTE_START, IFTE_END
- COMPARE_LHS,COMPARE_RHS
- CONDITION_FALSE
- ELSE_START,ELSE_END
- FOR_START,FOR_END
- AND_OR_OPERATOR

RUNTIME

- RUNTIME OF SPARKY IS BASED ON JAVA.
- IT INTERNALLY USES STACK AND HASHMAP.
- THE RUNTIME ENVIRONMENT ACCEPTS THE INTERMEDIATE CODE AND EXECUTES IT TO GIVE THE FINAL OUTPUT.

FEATURES WE HAVE USED IN OUR GRAMMAR

OUR PROGRAM STARTS WITH 'LIVE' AND ENDS WITH 'DIE'.

1. ARITHMETIC OPERATION–

   ADD: +, SUBTRACT: -MULTIPLY: *, DIVIDE: /

2. RELATIONAL OPERATORS: '=' , '==', '<','>','<=','>='

3. DATATYPE :

   INT, BOOLEAN,STRING

4. WHILE LOOP

5. LOGICAL OPERATION:

   AND, NOT, OR

6. FOR LOOP-

   THE TRADITIONAL - FOR LOOP, FOR IN RANGE LOOP.

7. TERNARY OPERATOR - ? :

8. STRING ASSIGNMENT.

9. IF THEN ELSE /IF THEN – IF (<CONDITION>) {<EXPRESSIONS>} WARNA {<EXPRESSIONS>} , IF (<CONDITION>) {<EXPRESSIONS>}.

10. PRINT CONSTRUCT.

GRAMMAR

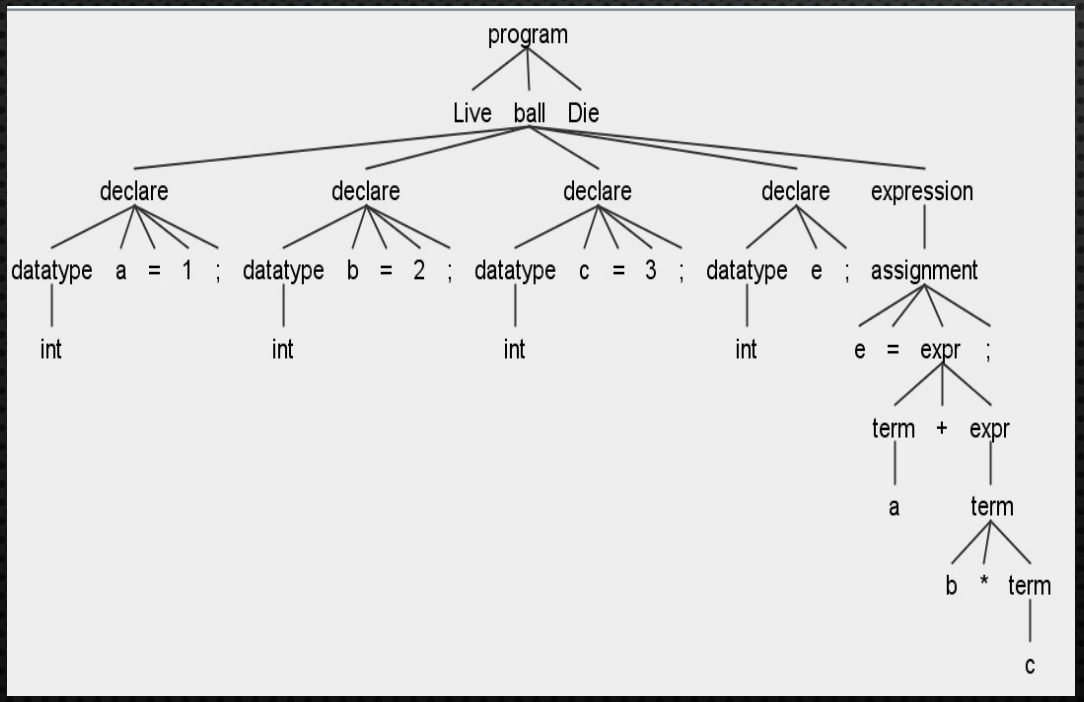
```
grammar Sparky;
program: LIVE ball DIE;
ball: expression* | declare* expression*;

declare:
(datatype STUFF EQUALTO NUMBER SEMICOLON)|
(datatype STUFF SEMICOLON)|
(HAINA STUFF EQUALTO booleanvalue SEMICOLON)|
(HAINA STUFF SEMICOLON)| stringdatatype STUFF EQUALTO STRINGLITERAL SEMICOLON | stringdatatype STUFF SEMICOLON;

expression
 : assignment| ifte| loopum|ternary_operator|print;

assignment
 : STUFF EQUALTO expr SEMICOLON |STUFF EQUALTO yesnostatement SEMICOLON
 ;
ifte: IF yesnostatement in_loop ('warna' in_loop)? FI;

loopum : loop_for|loop_while | loop_for_range;
loop_for: 'for' LSmoothBrace for_declare? ';' for_expression? ';' for_expr? RSmoothBrace in_loop;
loop_while: WHILE yesnostatement in_loop;
loop_for_range: 'for'  STUFF 'in' 'range' LSmoothBrace NUMBER COMMA NUMBER RSmoothBrace in_loop;

in_loop: LCurlyBrace ball RCurlyBrace| expression;
for_expr: STUFF EQUALTO expr;
for_expression :expr YESNOOPERATOR expr;
for_declare:datatype STUFF EQUALTO NUMBER;

print:'print' LSmoothBrace expr RSmoothBrace SEMICOLON;

term: NUMBER | STUFF | STUFF op=(MUL | DIV) term | NUMBER  op=(MUL | DIV) term;
expr: term | term op=(PLUS | MINUS) expr | NOT expr;
yesnostatement : booleanvalue | expr YESNOOPERATOR expr |yesnostatement ANDOROPERATOR yesnostatement;
YESNOOPERATOR: ASSEQ| LESS_THAN| MORE_THAN | LESS_THAN_EQ | MORE_THAN_EQ ;
ANDOROPERATOR: AND|OR;

AND: 'and';
OR: 'or';
```

GRAMMAR CONTINUED….

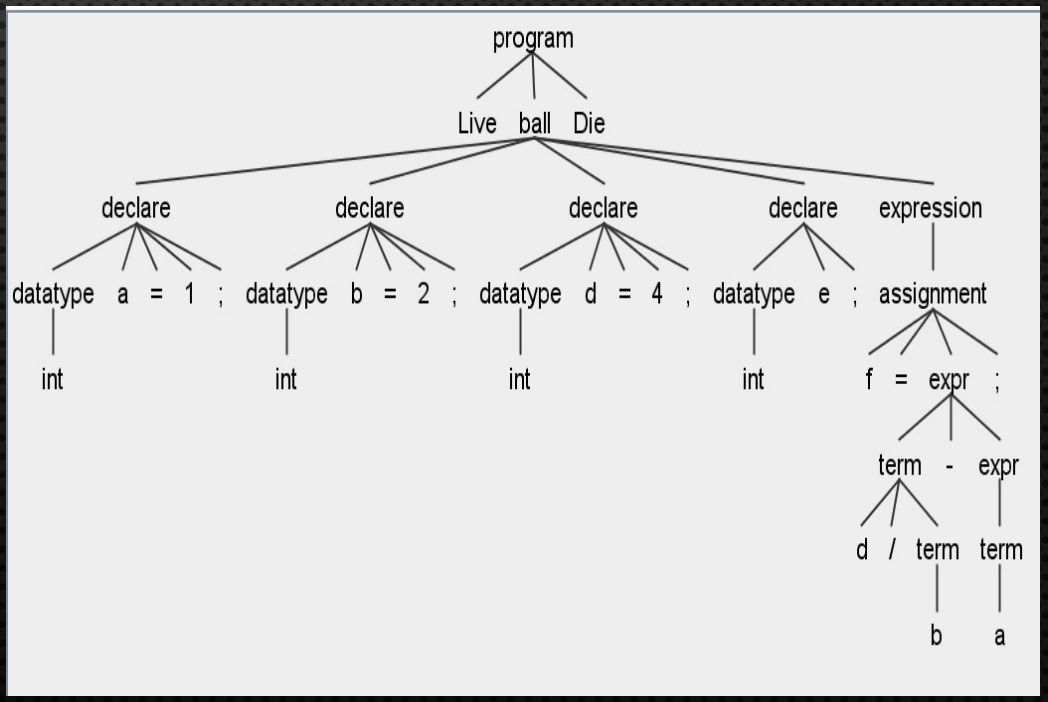
```
44    NOT:'not';
45    ternary_operator: yesnostatement '?' in_loop ':' in_loop;
46    LIVE: 'Live';
47    DIE: 'Die';
48    FI: 'fi';
49
50    EQUALTO : '=';
51    ASSEQ : '==';
52    LESS_THAN : '<';
53    MORE_THAN: '>';
54    LESS_THAN_EQ : '<=';
55    MORE_THAN_EQ : '>=';
56
57    warna :'else';
58    PLUS : '+';
59    MINUS :'-';
60    MUL : '*';
61    DIV : '/';
62    SEMICOLON : ';';
63    COMMA : ',';
64
65    LSmoothBrace : '(';
66    RSmoothBrace : ')';
67    LCurlyBrace : '{';
68    RCurlyBrace : '}';
69    DQ: '"';
70
71    STRINGLITERAL: DQ (~[\\\r\n])* DQ;
72
73    HAINA: 'haina';
74    haina:'bool';
75    datatype: INTEGER| DOUBLE | DECIMAL | CHAR | HAINA;
76    stringdatatype: STRING;
77
78    INTEGER: 'int';STRING: 'string';DOUBLE: 'double';DECIMAL: 'float';CHAR : 'char';
79
80    IF : 'if';
81    WHILE : 'while';
82    STUFF:[a-zA-Z_] [a-zA-Z_0-9]*;
83    NUMBER:[0-9]+;
84    WS: [ \t\r\n] -> skip;
85    booleanvalue: 'yup' | 'nup';
86    yup:'true';nup:'false';
```

PARSE TREE OF THE CONSTRAINTS: TWO AST OF ARITHMETIC OPERATIONS.

LIVE INT A=1; INT B=2; INT C= 3; INT E; E= A+B*C; DIE

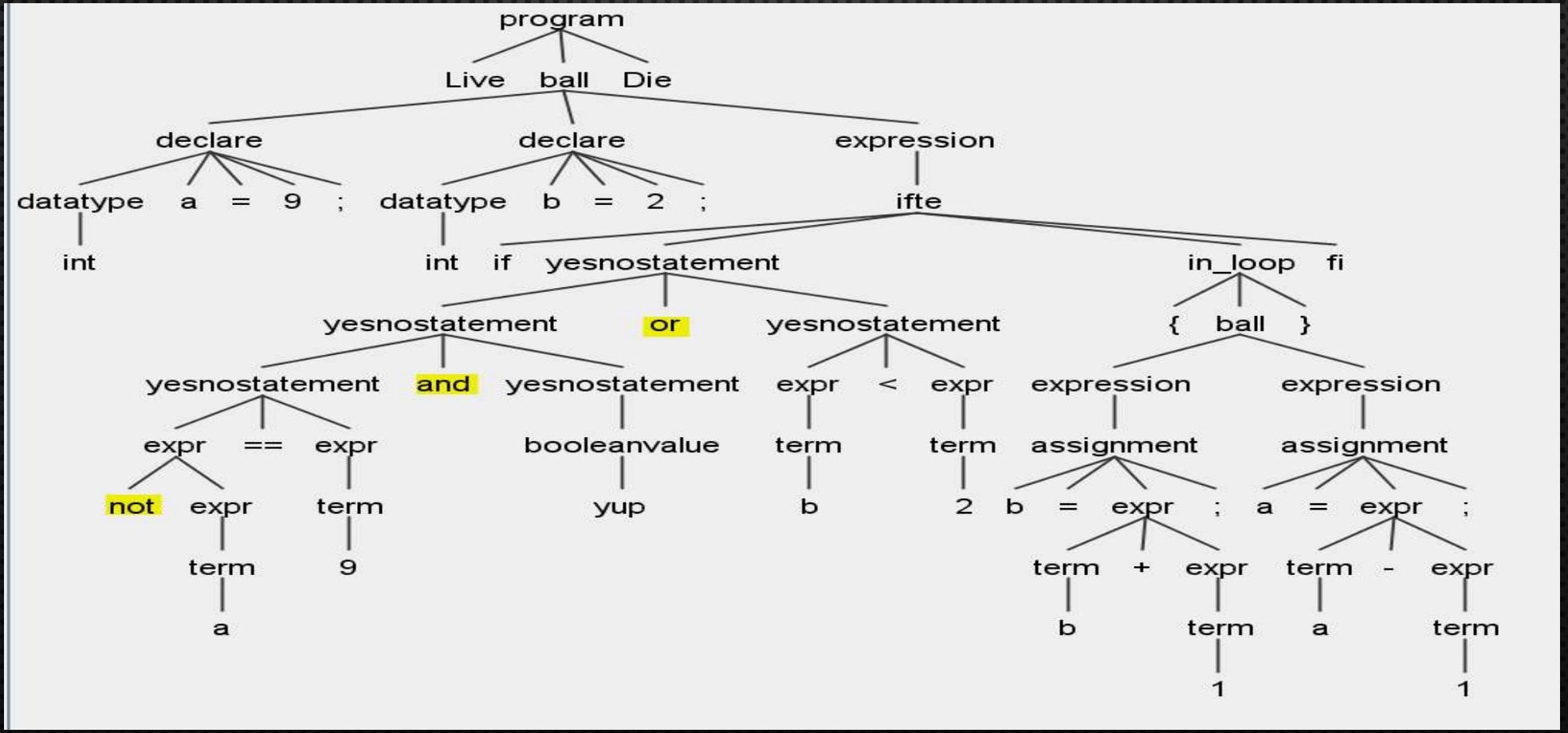


LIVE INT A=1; INT B=2; INT D=4; INT E; F=D/B-A; DIE



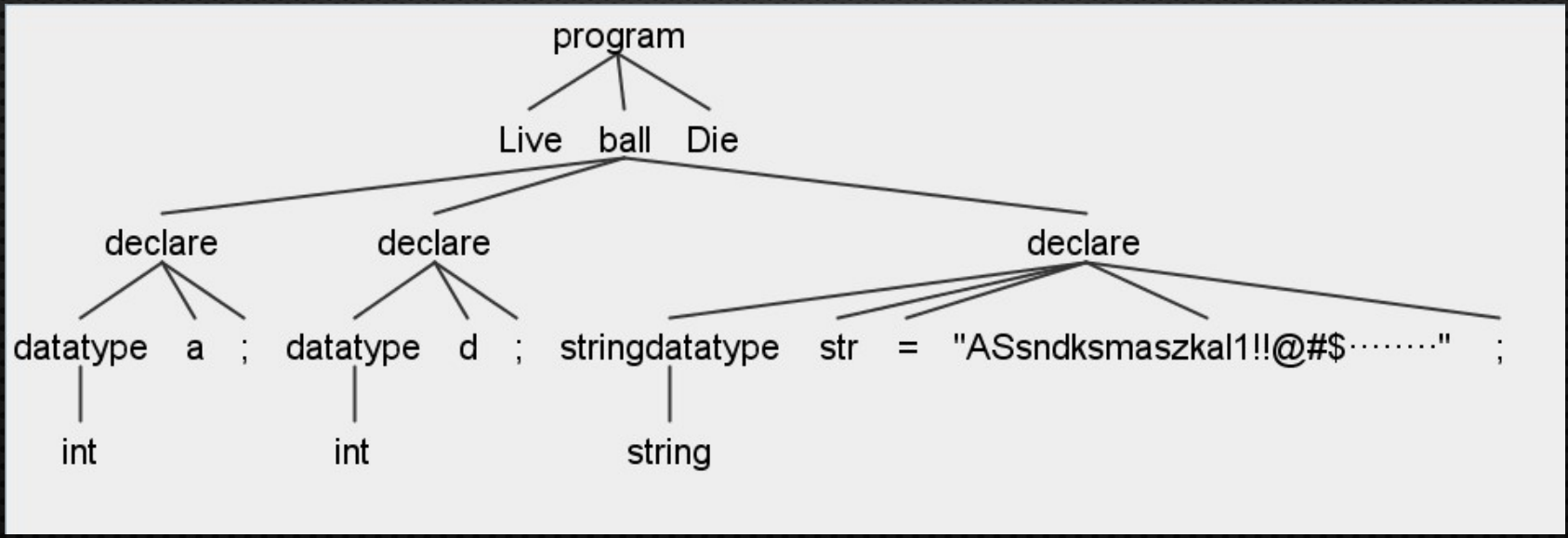
2. AND/OR/NOT: LOGICAL OPERATORS

LIVE INT A=9; INT B=2; IF NOT A==9 AND YUP OR B<2 {B=B+1;A=A-1;} FI DIE



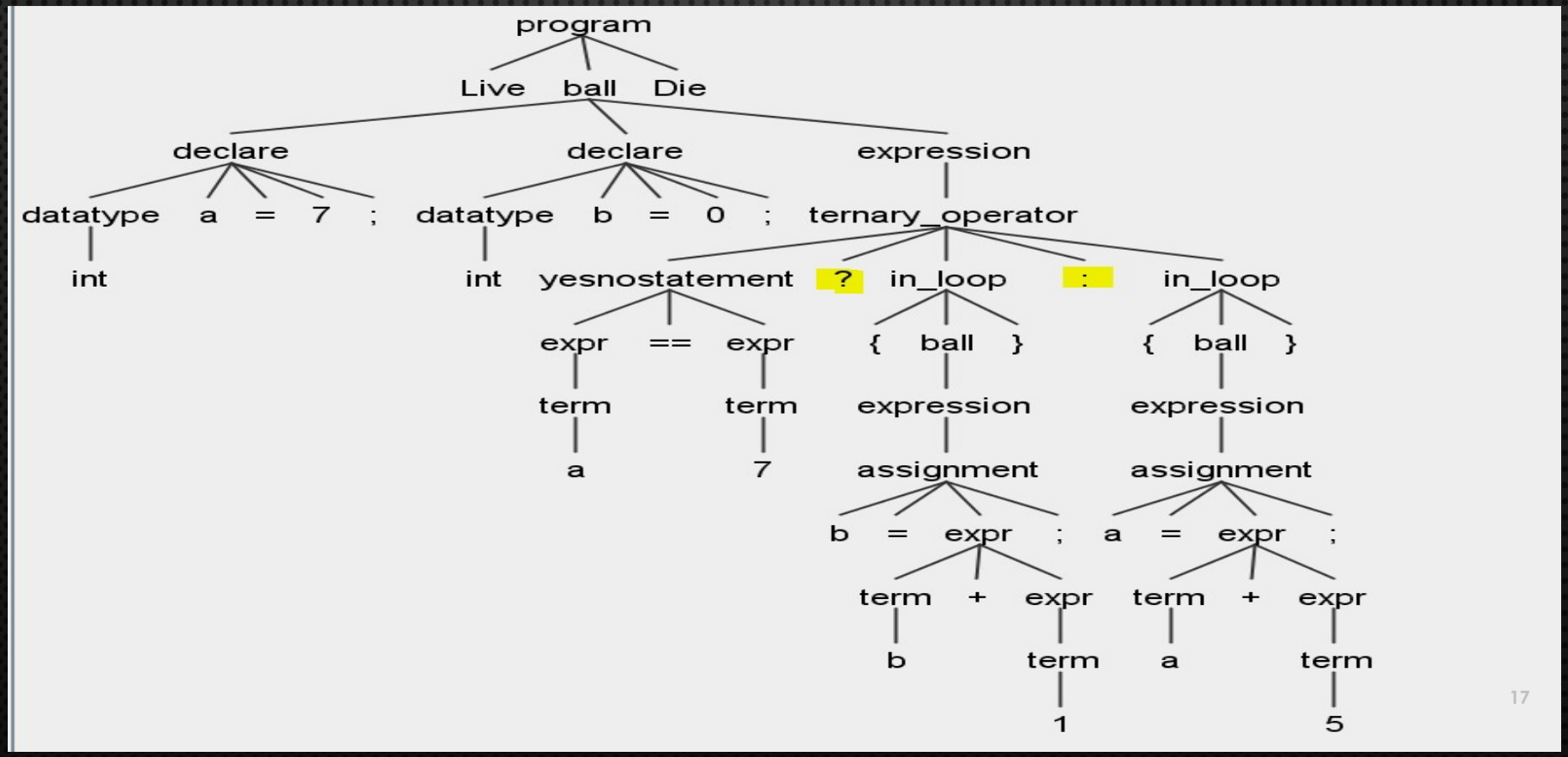
STRING VALUE ASSIGNMENTS TO VARIABLES

LIVE INT A; INT D; STRING STR= "ASSNDKSMASZKAL1!!@#$ "; DIE



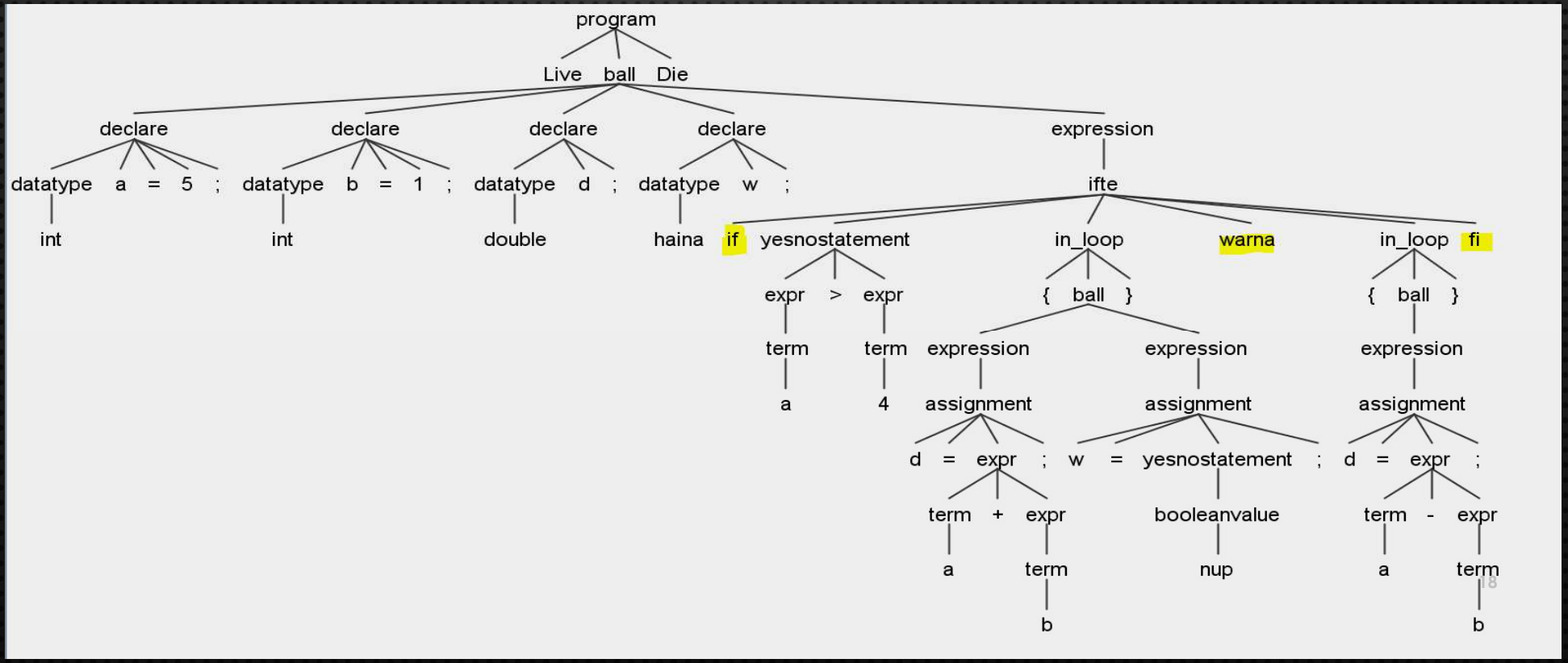
TERNARY OPERATOR

LIVE INT A=7; INT B=0; A==7? {B=B+1;}:{A=A+5;} DIE



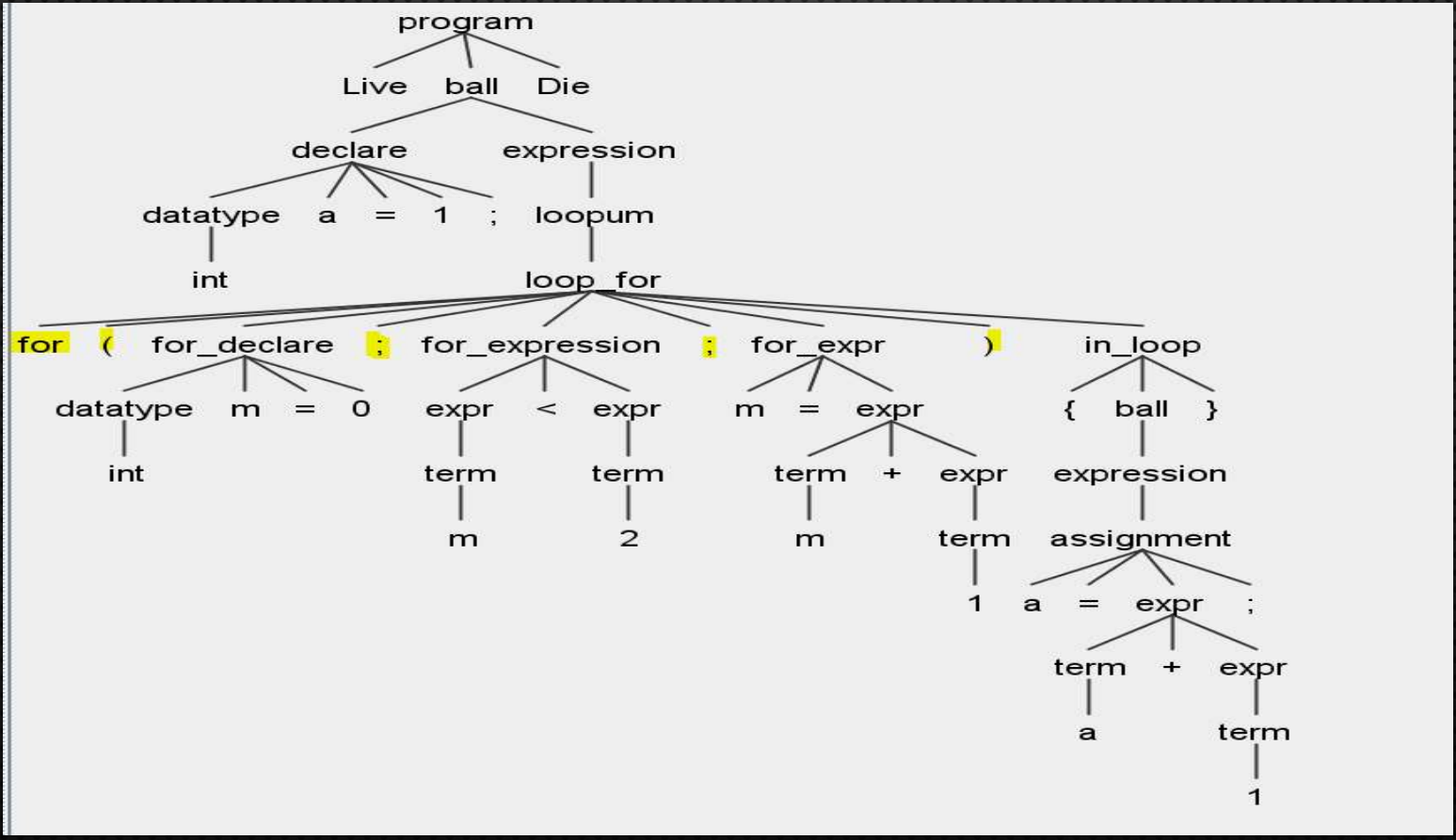
TRADITIONAL IF-THEN-ELSE CONSTRUCT

LIVE INT A=5; INT B=1; DOUBLE D; HAINA W; IF A>4 {D=A+B; W=NUP;} WARNA {D=A-B;} FI DIE



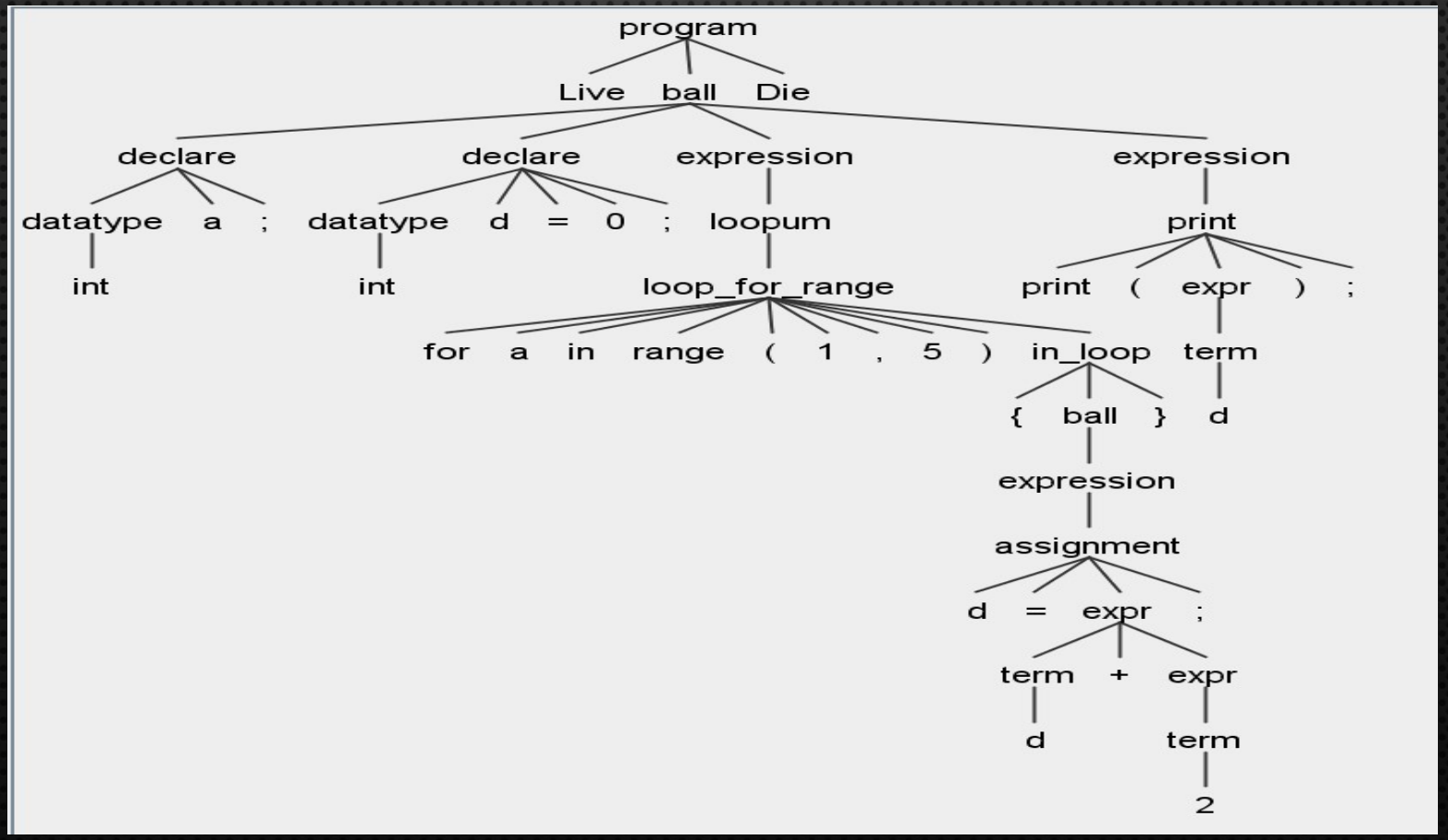
TRADITIONAL FOR LOOP

LIVE INT A=1; INT B=2; FOR(INT M=0;M<2;M=M+1){A=A+1;} DIE



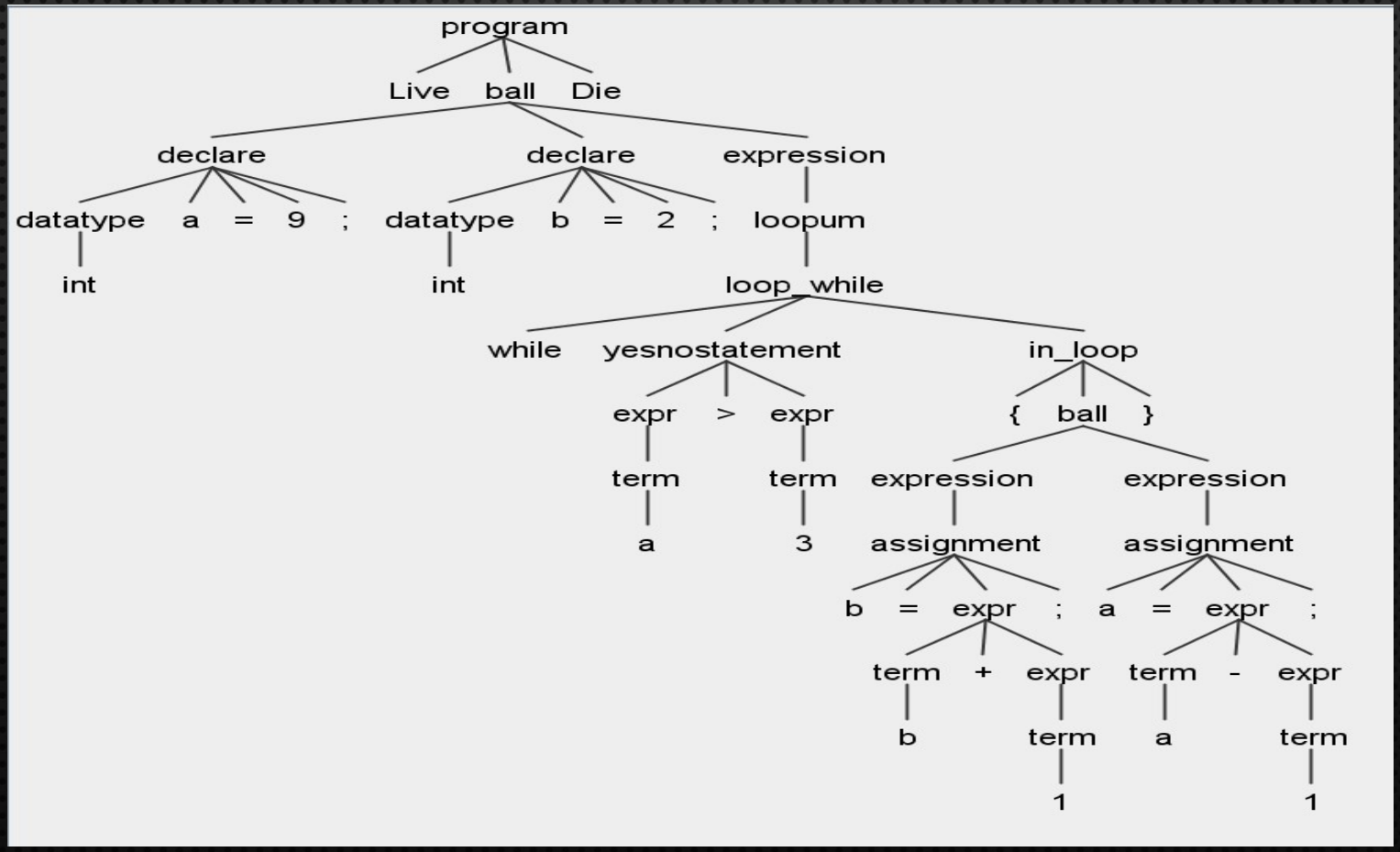
FOR IN RANGE LOOP

LIVE INT A; INT D=0; FOR A IN RANGE(1,5) {D=D+2;} PRINT (D); DIE



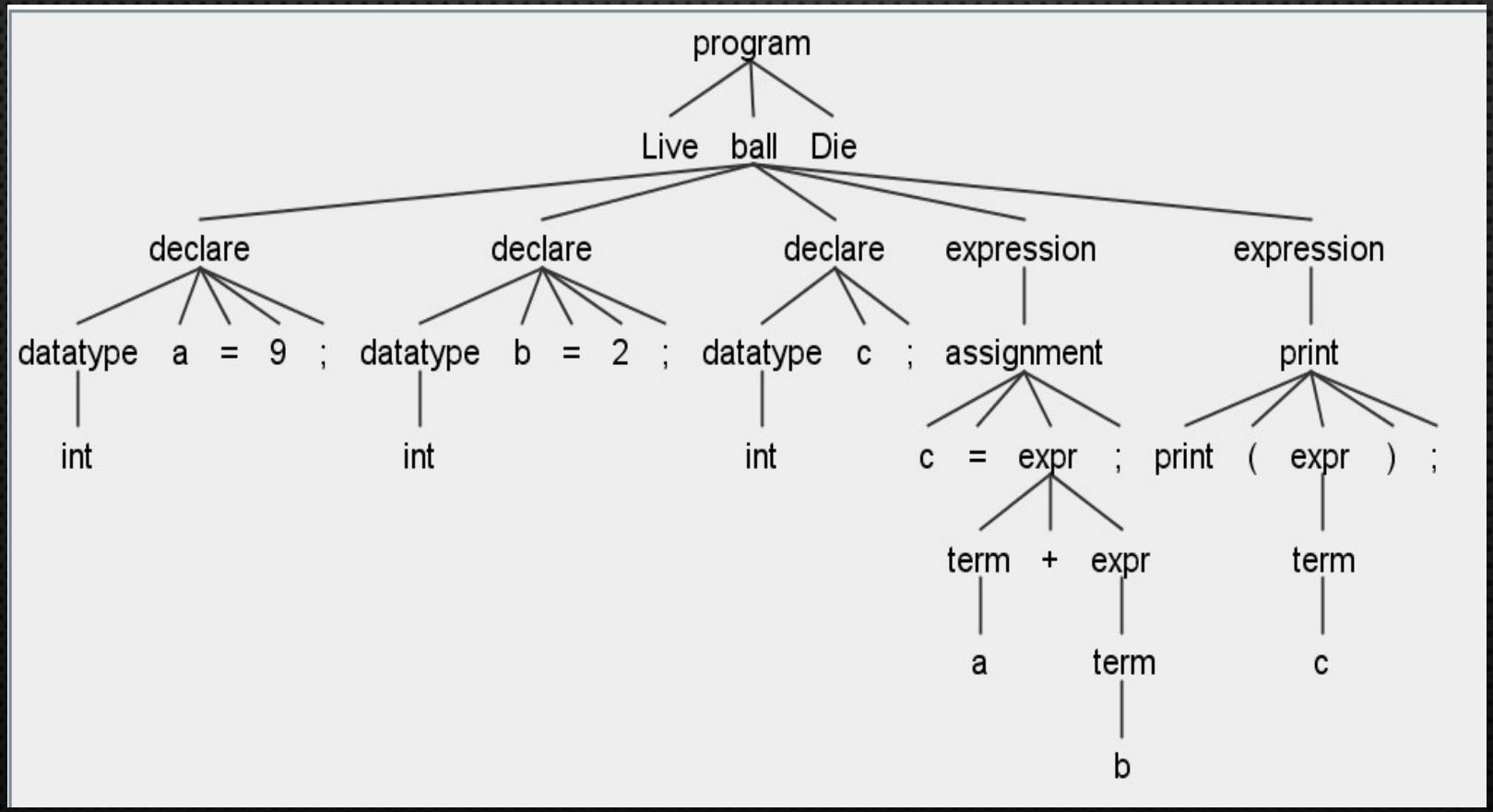
WHILE LOOP

LIVE INT A=9; INT B=2; WHILE A>3 {B=B+1;A=A-1;} DIE



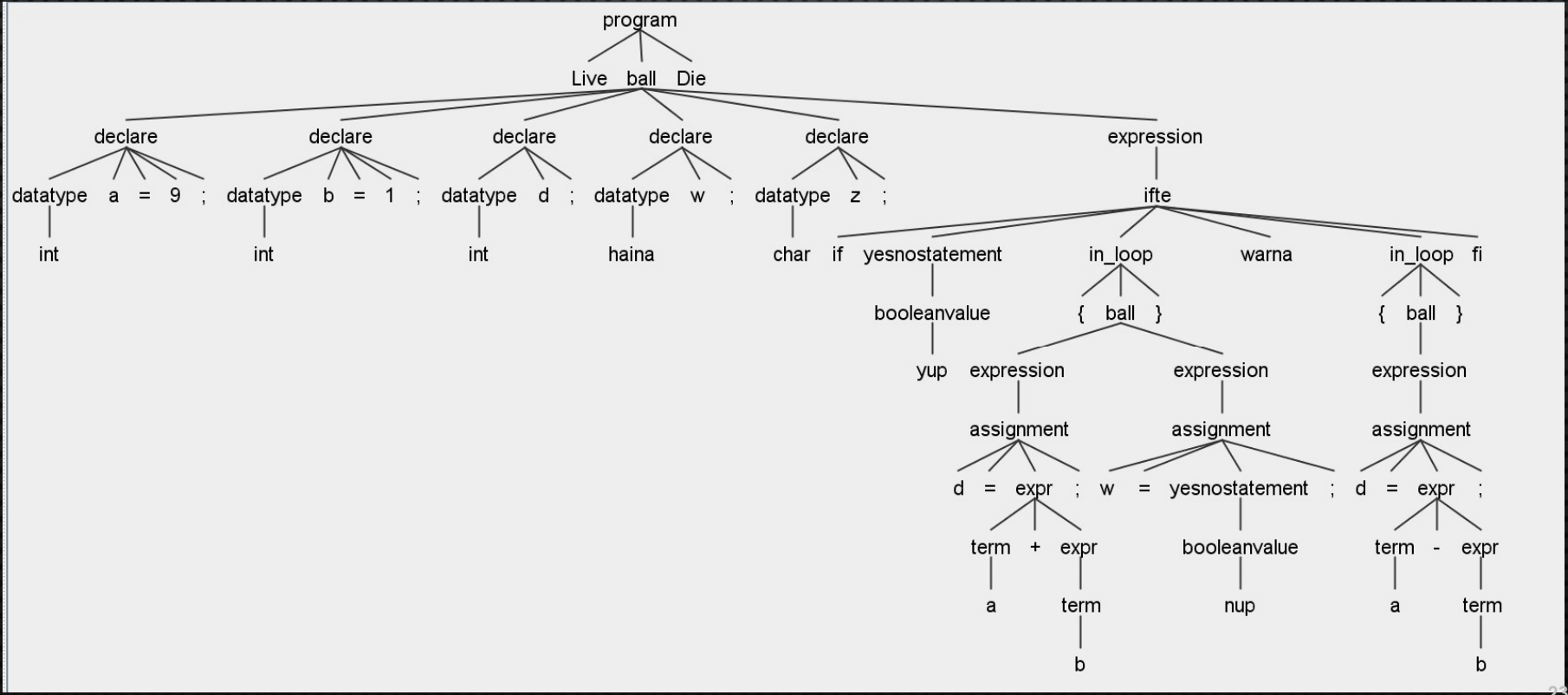
PRINT

LIVE INT A; INT D=0; FOR A IN RANGE(1,5) {D=D+2;} PRINT (D); DIE



IF THEN ELSE

LIVE INT A=9; INT B=1; DOUBLE D; HAINA W; IF YUP {D=A+B; W=NUP;} WARNA {D=A-B;} FI DIE



SAMPLE -1 TO CALCULATE FIBONACCI

SOURCE CODE

```
LIVE
INT COUNT = 7;
INT COUNTER = 1;
INT FIRSTFIB= 1;
INT SECONDFIB= 1;
INT SUM;
WHILE COUNTER<=COUNT
{
PRINT(FIRSTFIB);
SUM=FIRSTFIB+SECONDFIB;
FIRSTFIB = SECONDFIB;
SECONDFIB=SUM;
COUNTER=COUNTER+ 1;
}
DIE
```

INTERMEDIATE CODE

```
DECLARE INT COUNT
STORE 7
PUSH COUNT
DECLARE INT COUNTER
STORE 1
PUSH COUNTER
DECLARE INT FIRSTFIB
STORE 1
PUSH FIRSTFIB
DECLARE INT SECONDFIB
STORE 1
PUSH SECONDFIB
DECLARE INT SUM
WHILEBEGIN
GET COUNTER
GET COUNT
COMPARE_OPERATOR <=
CONDITION_FALSE JUMP WHILEEND
GET FIRSTFIB
PRINT
GET FIRSTFIB
GET SECONDFIB
OPERATOR ADD
PUSH SUM
GET SECONDFIB
PUSH FIRSTFIB
GET SUM
PUSH SECONDFIB
GET COUNTER
STORE 1
OPERATOR ADD
PUSH COUNTER
JUMP WHILEBEGIN
WHILEEND
```

SAMPLE -2 TO CALCULATE FACTORIAL

SOURCE CODE

```
LIVE
INT RESULT = 1;
INT N=5;
FOR (INT I=2; I<=N; I=I+1)
{RESULT=RESULT*I;}
PRINT(RESULT);
DIE
```

INTERMEDIATE CODE

```
DECLARE INT RESULT
STORE 1
PUSH RESULT
DECLARE INT N
STORE 5
PUSH N
FOR_START
DECLARE INT I
STORE 2
PUSH I
FOR_CONDITION_START
GET I
GET N
COMPARE_OPERATOR <=
CONDITION_FALSE JUMP FOR_STOP
GET RESULT
GET I
OPERATOR MULTIPLY
PUSH RESULT
GET I
STORE 1
OPERATOR ADD
PUSH I
JUMP FOR_CONDITION_START
FOR_STOP
GET RESULT
PRINT
```

FUTURE WORKS

- 1. We will next implement functions
- 2. We will try to incorporate complex datatypes like arrays, List, dictionaries, sets.
- 3. we will implement nested loops
- 4. String manipulation