Keywords (same as Phase 01)

- bidaya
- nihaya
- adkhal
- bayan
- iza
- aw
- li
- amal
- raji3

Operators

- zid (+)
- naqis (-)
- musaawi (==)
- = < > %

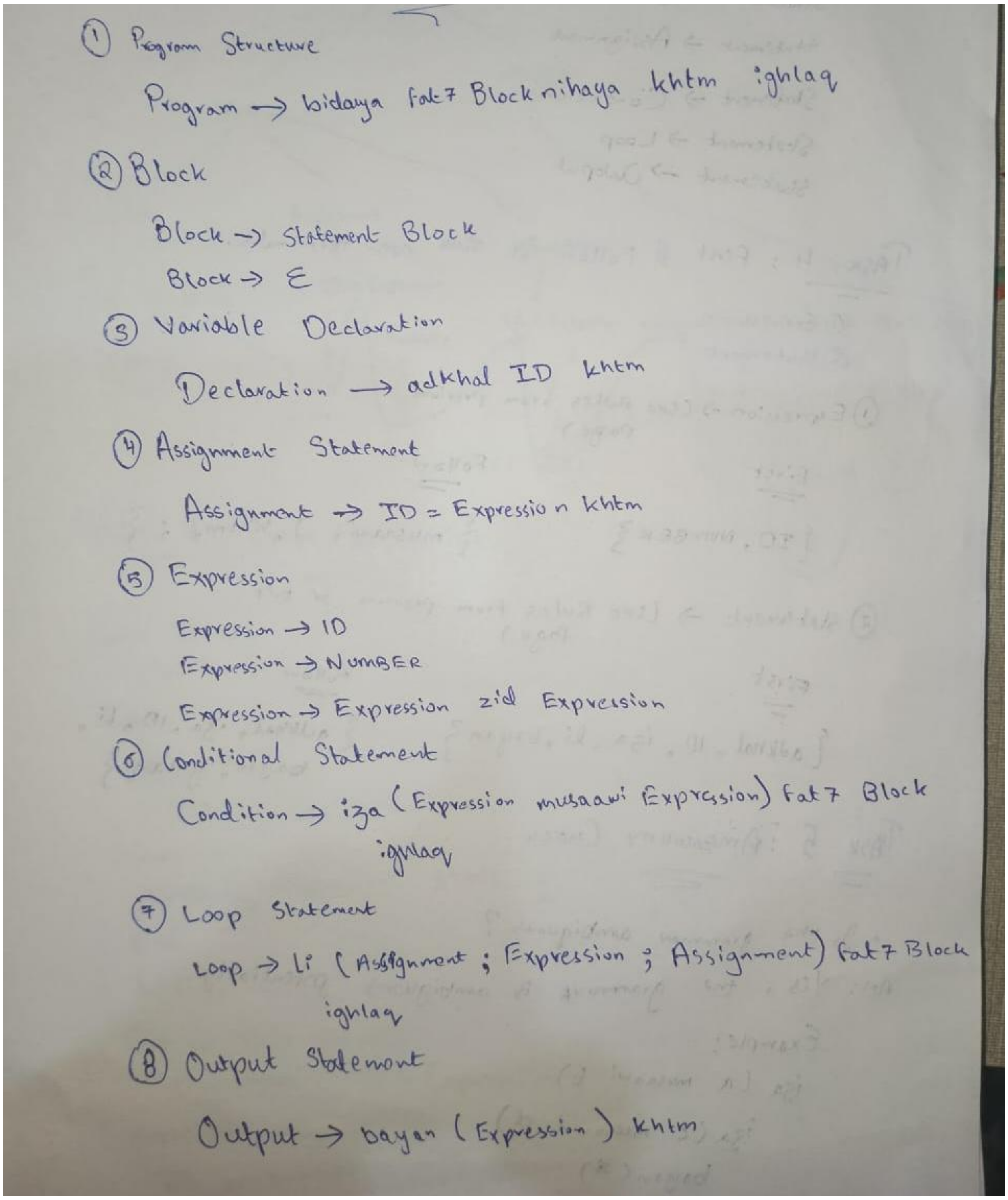
Punctuations

- fat7 ({)
- ighlaq (})
- khtm (;)
- ( )

Program starts with: bidaya fat7

Program ends with: nihaya khtm ighlaq

CFG (Arabic++)



① Program Structure

Program → bidaya fat7 Block nihaya khtm ighlaq

② Block

Block → Statement Block

Block → ε

③ Variable Declaration

Declaration → adkhal ID khtm

④ Assignment Statement

Assignment → ID = Expression khtm

⑤ Expression

Expression → ID

Expression → NUMBER

Expression → Expression zid Expression

⑥ Conditional Statement

Condition → iza (Expression musaawi Expression) fat7 Block ighlaq

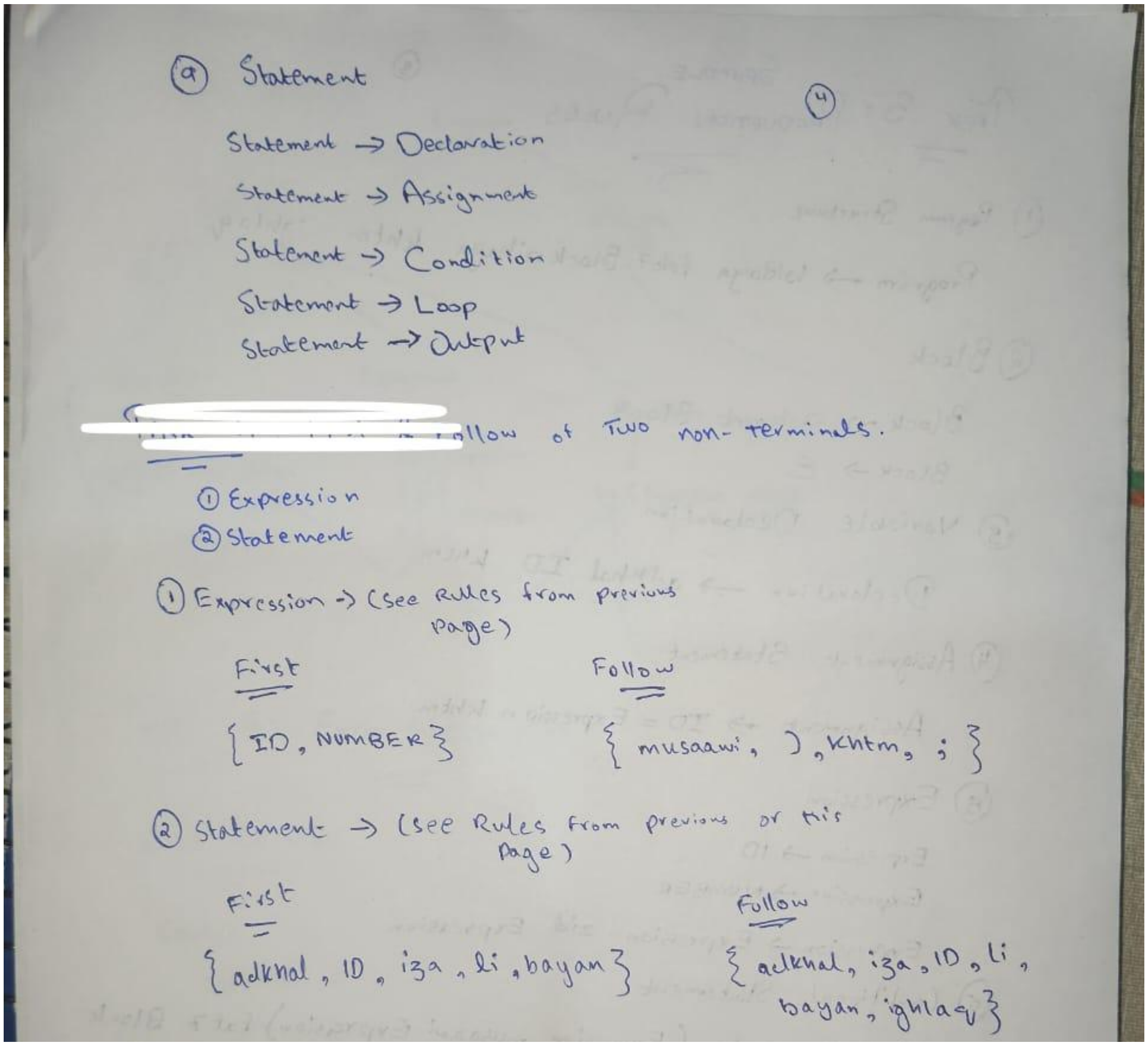
⑦ Loop Statement

Loop → li (Assignment ; Expression ; Assignment) fat7 Block ighlaq

⑧ Output Statement

Output → bayan (Expression) khtm

First & Follow:



(9) Statement

(4)

Statement → Declaration

Statement → Assignment

Statement → Condition

Statement → Loop

Statement → Output

[redacted] follow of Two non-terminals.

1. Expression
2. Statement

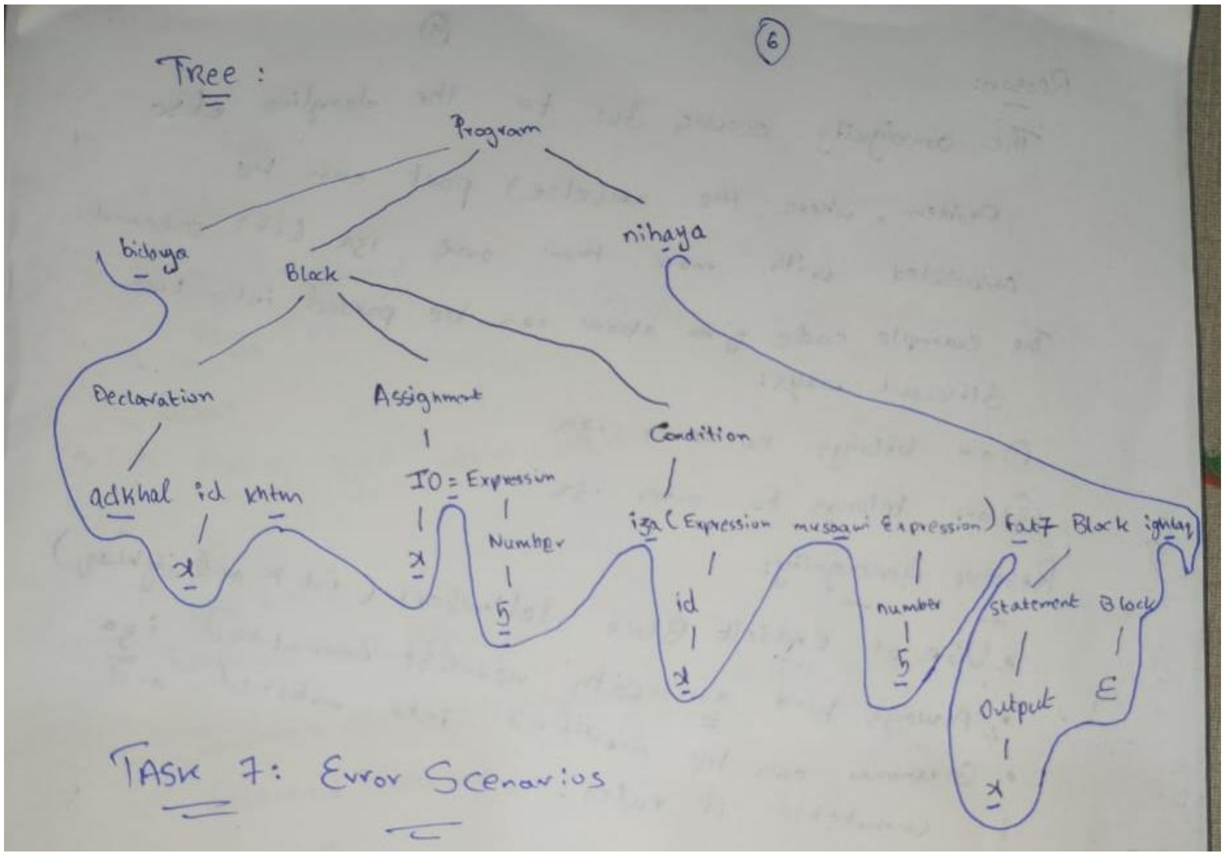
(1) Expression → (see Rules from previous page)

First	Follow
{ ID, NUMBER }	{ musaawi,), khtm, ; }

(2) Statement → (see Rules from previous or this page)

First	Follow
{ adkhal, ID, iza, li, bayan }	{ adkhal, iza, ID, li, bayan, ighlaq }

Parse Tree:



Explanation of How Phase-01 Tokens Are Used in Phase-02

In Phase 01 of the project, a lexical analyzer (scanner) was developed using Flex. The purpose of this scanner was to read the source code written in the Arabic++ language and convert the raw input characters into meaningful tokens. Each token represents a logical unit such as a keyword, identifier, operator, punctuation, number, or string, along with its line number.

Phase 02 builds directly on the output of Phase 01. The syntax analyzer (parser), implemented using YACC/Bison, does not read characters from the source file directly. Instead, it relies entirely on the tokens generated by the scanner in Phase 01. The parser receives these tokens through the yylex() function and matches them against the grammar rules of the Arabic++ language.

For example, when the scanner encounters the word ‘bidaya’, it returns the token ‘bidaya’ to the parser. The parser then uses this token to verify that the program correctly starts according to the defined grammar. Similarly, tokens such as ID, NUMBER, zid, khtm, fat7, and ighlaq are used by the parser to validate declarations, expressions, loops, conditional statements, and block structures.

Operators and punctuations defined in Phase 01 (such as zid, naqis, musaawi, khtm, fat7, and ighlaq) are reused without modification in Phase 02. This ensures consistency between lexical analysis and syntax analysis. If the scanner misclassifies a token or reports an error, the parser will not proceed further for that input, maintaining strict correctness.

In summary, Phase 01 provides the foundational token stream, while Phase 02 uses this token stream to verify the grammatical structure of the Arabic++ program. Both phases work together to form a complete front-end of a compiler, where Phase 01 handles tokenization and Phase 02 handles syntactic validation.