

Compilers Report

The Implementation of Lexical Analyzer

* **Team 77 -**

|  |  |
| --- | --- |
| **Subject :** | **Compilers** |
| **Submit Date :** | **2020.05.09** |
| **Professor :** | **김효수** |
| **Major:** | **CSE** |
| **Members:** | **20153479 이동민** |
|  | **20160492 허지호** |
|  |  |

0. Index 1

1. Specification Analysis & Language Definition 2

1-1. Definition of tokens 2

1-2. Definition of alphabet sets 4

1-3. Definition of regular expressions 5

2. Transition Graphs 5

2-1. Composition of NFA 5

2-2. NFA to DFA 13

2-3. Merged DFA graph 20

3. Code Implementation 21

3-1. Overall code structure 21

3-2. Code details 21

3-3. Main errors and exception handlings 24

1. **Specification Analysis & Language Definition**
   1. Definition of tokens

Following the lexical specification, we can make the following table.

|  |  |  |
| --- | --- | --- |
| Token name | Pattern | Examples of lexeme |
| Integer | * A single zero digit * A non-empty sequence of digits * A non-empty sequence of digits, starting from a non-zero digit * A non-empty sequence of digits, starting from a minus sign symbol and a non-zero digit | 0,  1, 22, 123, 56, …  -1, -22, -123, -56, …  any non-zero positive or negative integers |
| Literal | * Any combination for digits, English letters, and blanks, starting from and terminating with a symbol | “Hello world”  “My student is 12345678” |
| Boolean | * True or false | true, false |
| Float | * It starts with or without a negative sign symbol * . (a decimal point) appears only once * The left side of a decimal point must be a single digit 0 or a non-empty sequence terminating with a non-zero digit * Both left and right side of the decimal point must not be empty sequence * Scientific/exponential symbols like E are not allowed | -0.12, 0.123  0.5  0.0, -10.0, 100.0001, …  any non-zero positive or negative Floating numbers |
| IDENTIFIER | * Non-empty sequence of English letters, digits and underscore symbols, starting from an English letter or an underscore symbol | i, j, k, abc,  ab\_123, func1, func\_,  \_func\_bar\_ |
| IF | * Head keyword for if statement | if |
| Else | * Head keyword for else statement | else |
| For | * Head keyword for for-loop statement | for |
| While | * Head keyword for while-loop statement | while |
| Return | * Head keyword for return statement | return |
| Arithmetic | * Operators for arithmetic operation | +, -, \*, / |
| Bitwise | * Operators for bitwise operation | >>, <<. &, | |
| Assignment | * Operator for assignment | = |
| Comparison | * Operators for comparison operation | <, >, ==, !=,  <=, >= |
| Terminating | * A terminating symbol of statement | ; |
| Lbracket | * A left-side symbol for defining area/scope of variables and functions | { |
| Rbracket | * A right-side symbol for defining area/scope of variables and functions | } |
| Lparen | * A left-side symbol for indicating a function/statement | ( |
| Rparen | * A right-side symbol for indicating a function/statement | ) |
| Comma | * A symbol for separating input arguments on functions | , |
| Whitespace | * A non-empty sequence | \t, \n, blank |
| DATATYPE | * A symbol for defining a data type of the given identifier. | int, char, float, bool |

* 1. Definition of alphabet sets

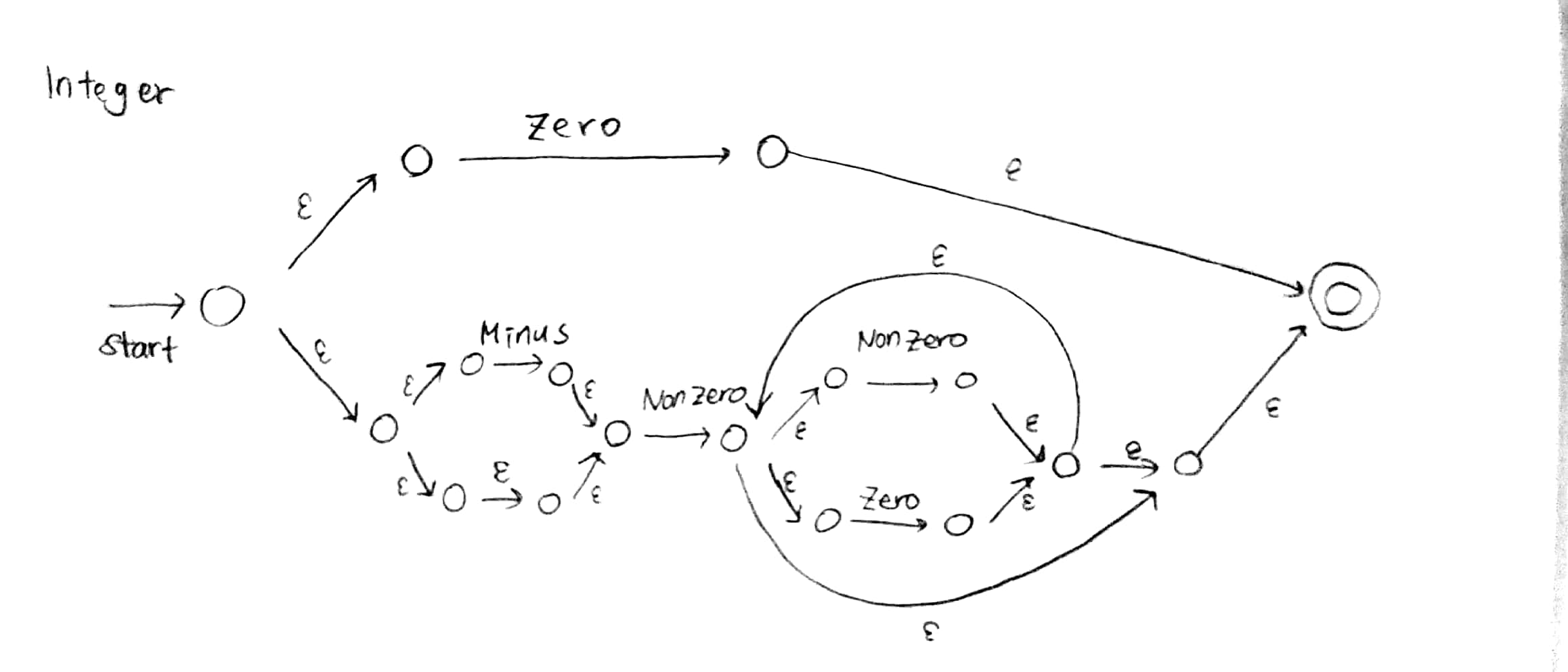
|  |  |
| --- | --- |
| Alphabet set | Materials |
| Keyword Letter | { b, c, e, f, i, r, t, w } |
| Not Keyword Letter | { The letters except the first letter of keyword, Boolean value, data type } |
| Underscore | { \_ } |
| Whitespace | { ‘ ‘, \t, \n } |
| Less than | { < } |
| Greater than | { > } |
| Equal | { = } |
| Not | { ! } |
| And Or | { &, | } |
| Zero | { 0 } |
| Nonzero | { 1, 2, 3, 4, 5, 6, 7, 8, 9 } |
| Digit | { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } |
| Dot | { . } |
| Minus | { - } |
| Double Quote | { “ } |
| Comma | { , } |
| Arithmetic | { +, \*, / } |
| Termination | { ; } |
| Lbracket | { { } |
| Rbracket | { } } |
| Lparen | { ( ) |
| Rparen | { } } |
| A Letter | { a } |
| B Letter | { b } |
| C Letter | { c } |
| E Letter | { e } |
| F Letter | { f } |
| H Letter | { h } |
| I Letter | { i } |
| L Letter | { l } |
| N Letter | { n } |
| O Letter | { o } |
| R Letter | { r } |
| S Letter | { s } |
| T Letter | { t } |
| U Letter | { u } |
| W Letter | { w } |
| Letter | { a, b, c, d, e, …, z, A, B, C, D, …, Z } |
| Blank | { ‘ ‘ } |

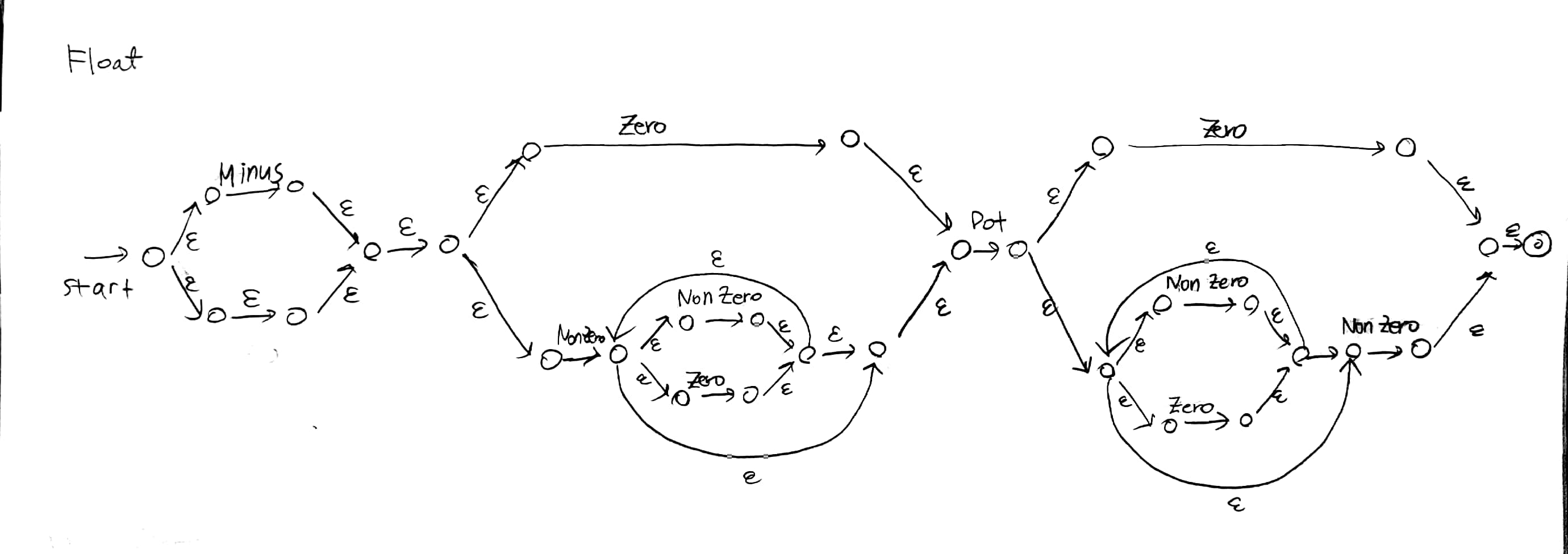
* 1. Definition of regular expressions

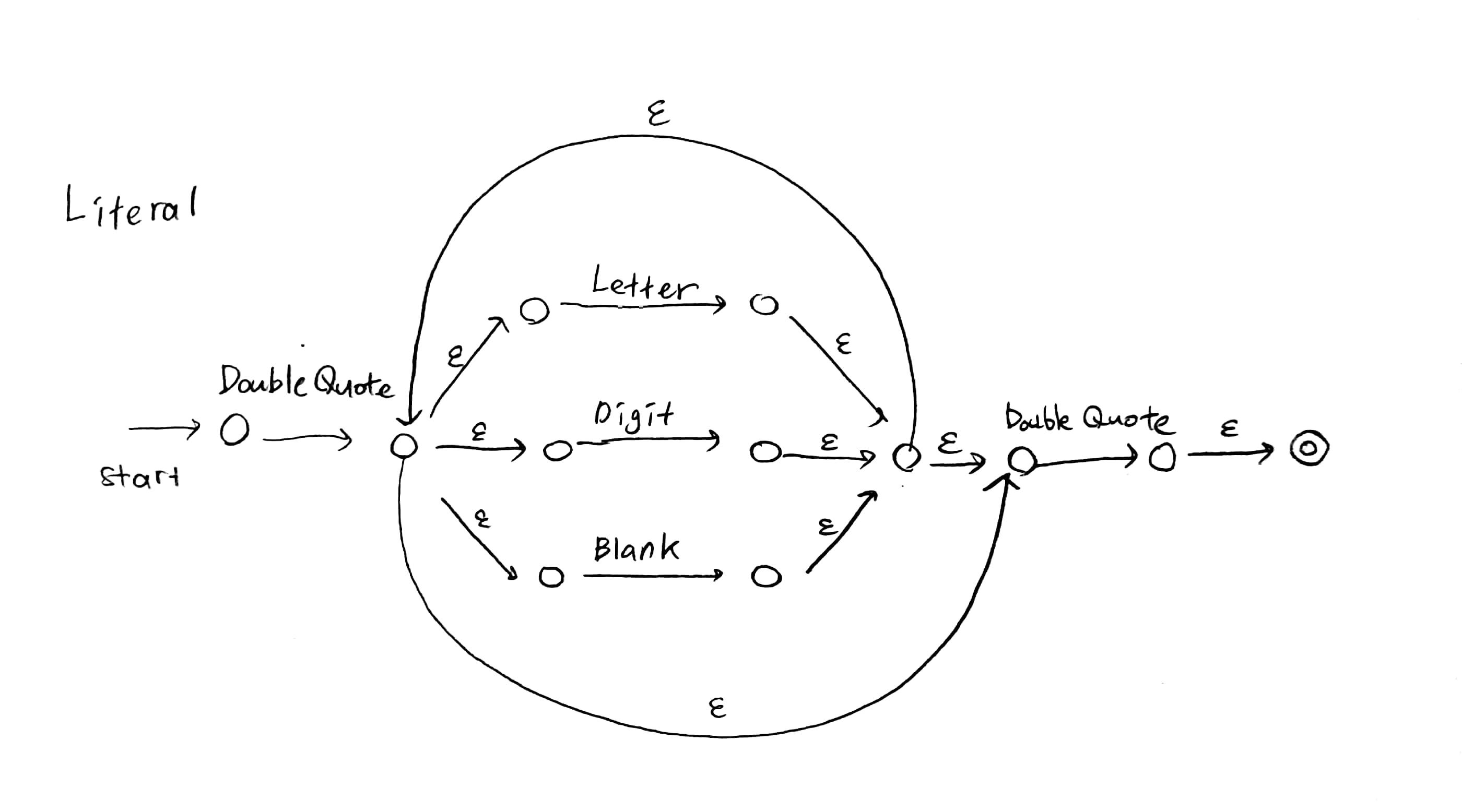
|  |  |
| --- | --- |
| Token name | Regular expression |
| Integer | ( Zero ) | ( Minus |  )( Nonzero )((Nonzero | zero)\*) |
| Literal | ( Double Quote )( Letter | Digit | Blank )\*( Double Quote ) |
| Boolean | (( t Letter )( r Letter )( u Letter )( e Letter )) | (( f Letter )( a Letter )( l Letter )( s Letter )( e Letter )) |
| Float | ( Minus |  )( Zero | ((Nonzero)(Nonzero | Zero)\*) )( Dot ) ( Zero | ((Nonzero | Zero)\*(Nonzero)) ) |
| IDENTIFIER | ( Letter | Underscore )( Letter | Digit | Underscore )\* |
| IF | ( i Letter )( f Letter ) |
| Else | ( e Letter )( l Letter )( s Letter )( e Letter ) |
| For | ( f Letter )( o Letter )( r Letter ) |
| While | ( w Letter )( h Letter )( i Letter )( l Letter )( e Letter ) |
| Return | ( r Letter )( e Letter )( t Letter )( u Letter )( r Letter )( n Letter ) |
| Arithmetic | ( Arithmetic ) |
| Bitwise | ( And Or | (( Less than )( Less than )) | (( Greater than )( Greater than )) ) |
| Assignment | ( Equal ) |
| Comparison | ( Less than | Greater than ) | (( Not | Equal | Less than | Greater than )( Equal )) |
| Terminating | ( Termination )t |
| Lbracket | ( Lbracket ) |
| Rbracket | ( Rbracket ) |
| Lparen | ( Lparen ) |
| Rparen | ( Rparen ) |
| DATATYPE | (( i Letter )( n Letter )( t Letter )) | (( f Letter )( l Letter )( o Letter )( a Letter )( t Letter )) | (( c Letter )( h Letter )( a Letter )( r Letter )) | (( b Letter )( o Letter )( o Letter )( l Letter )) |
| Whitespace | ( Whitespace )( Whitespace )\* |
| COMMA | ( Comma ) |

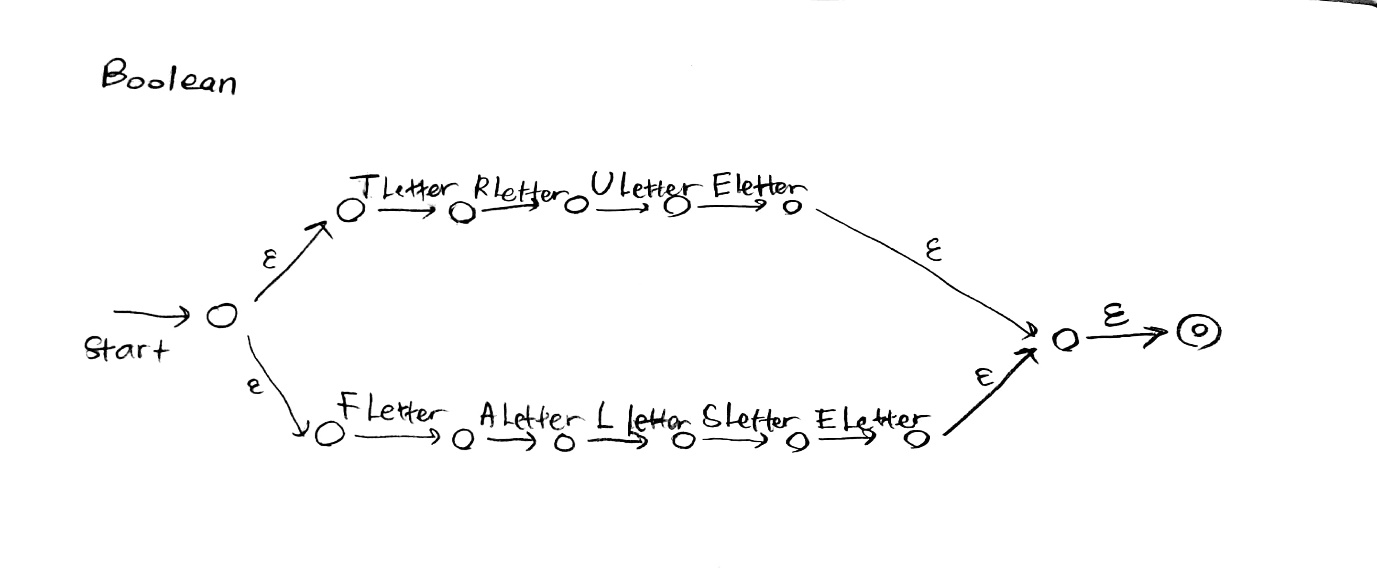
1. **Transition Graphs**

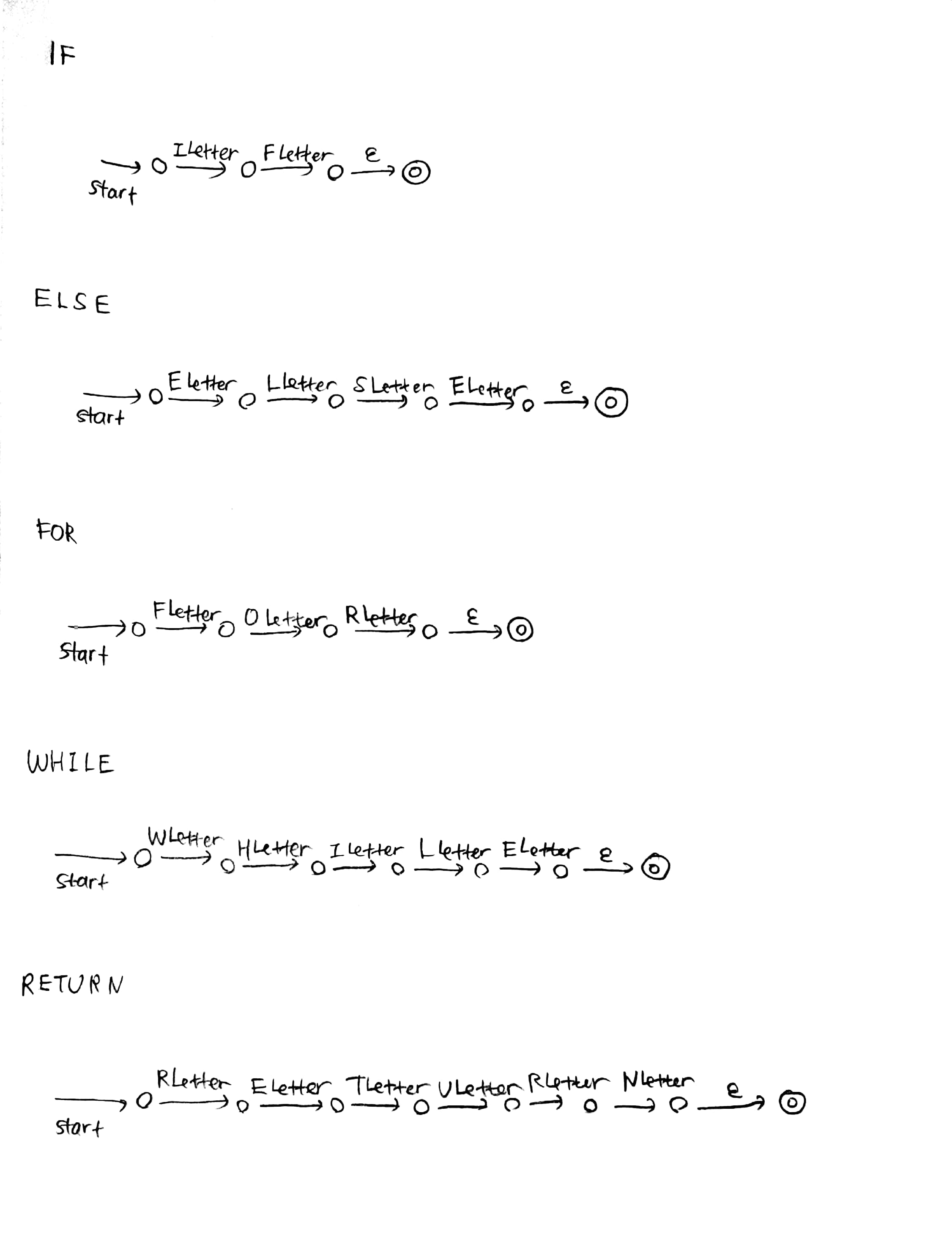
2-1. Composition of NFA

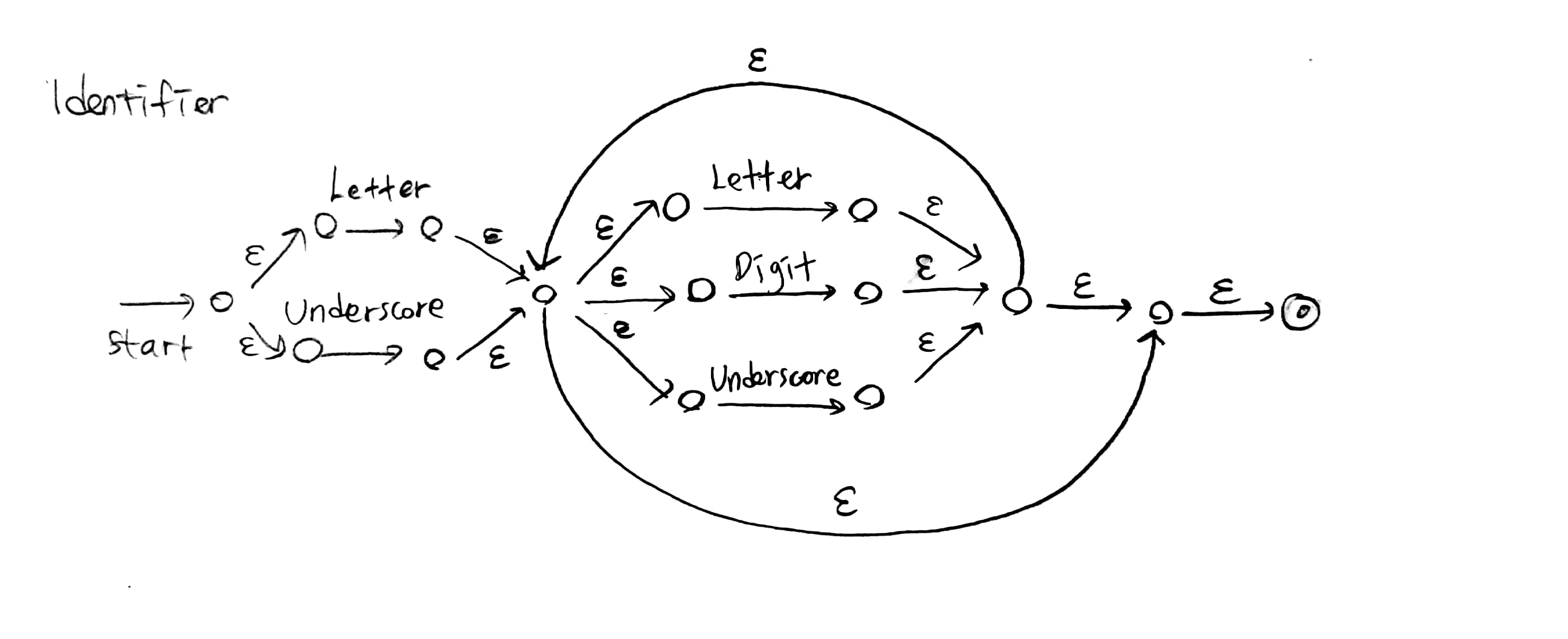
- Integer

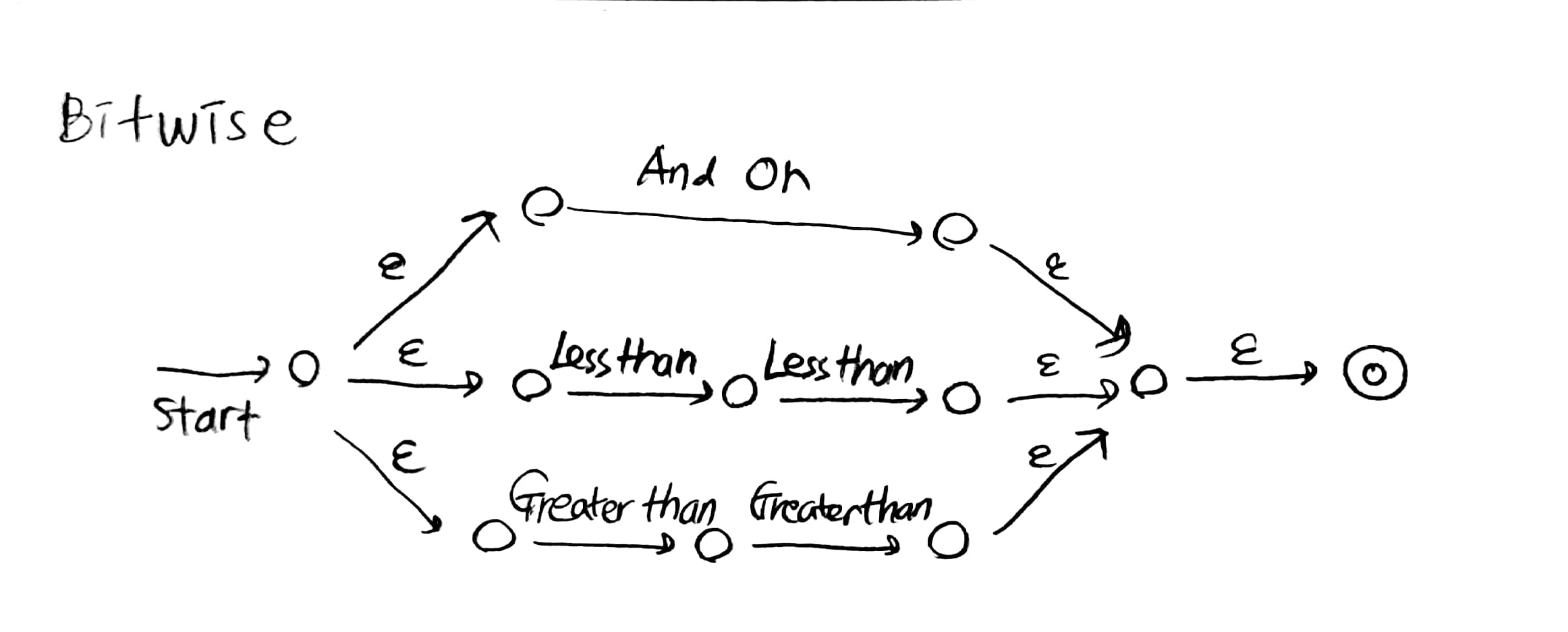
- Float

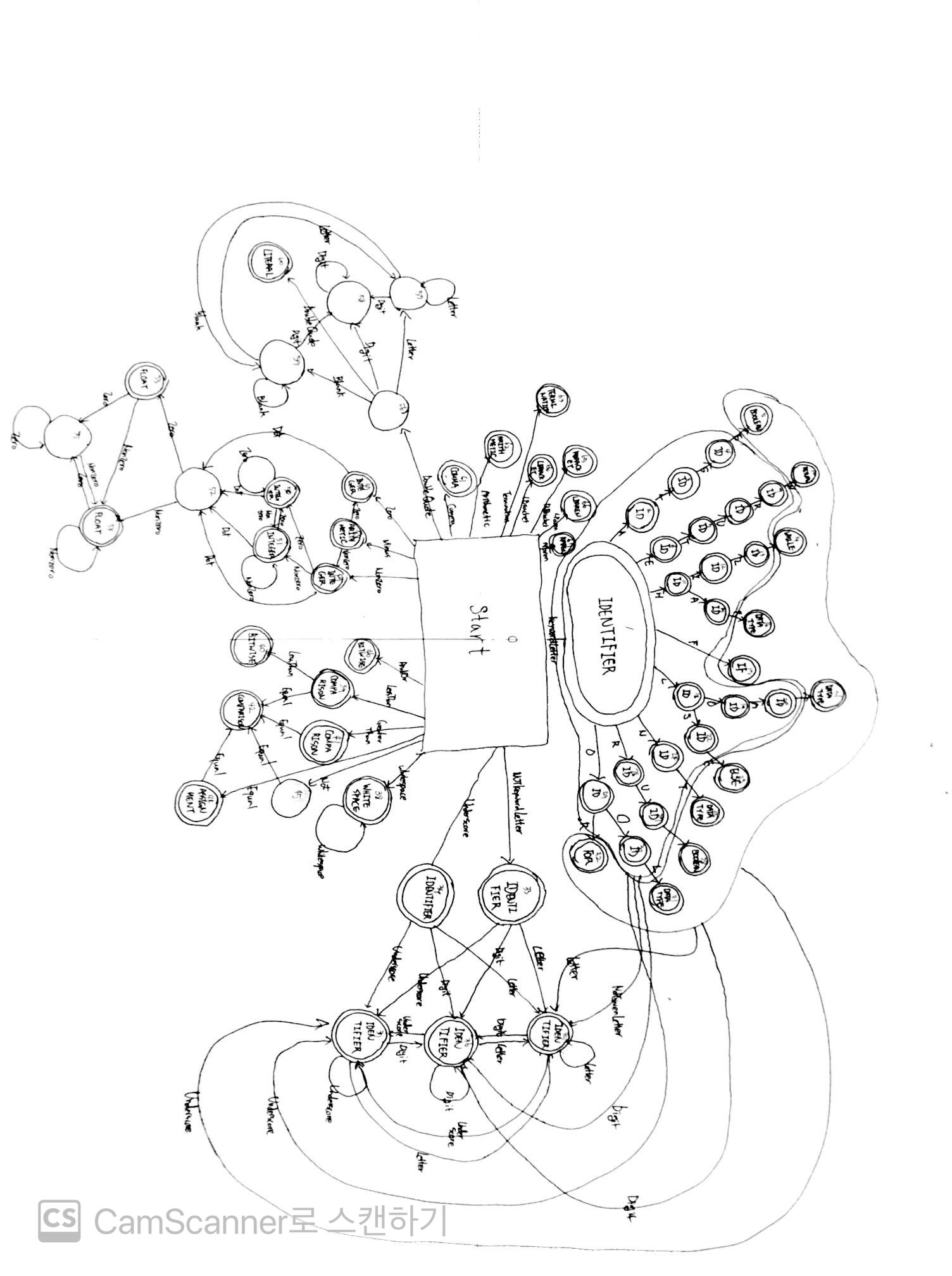
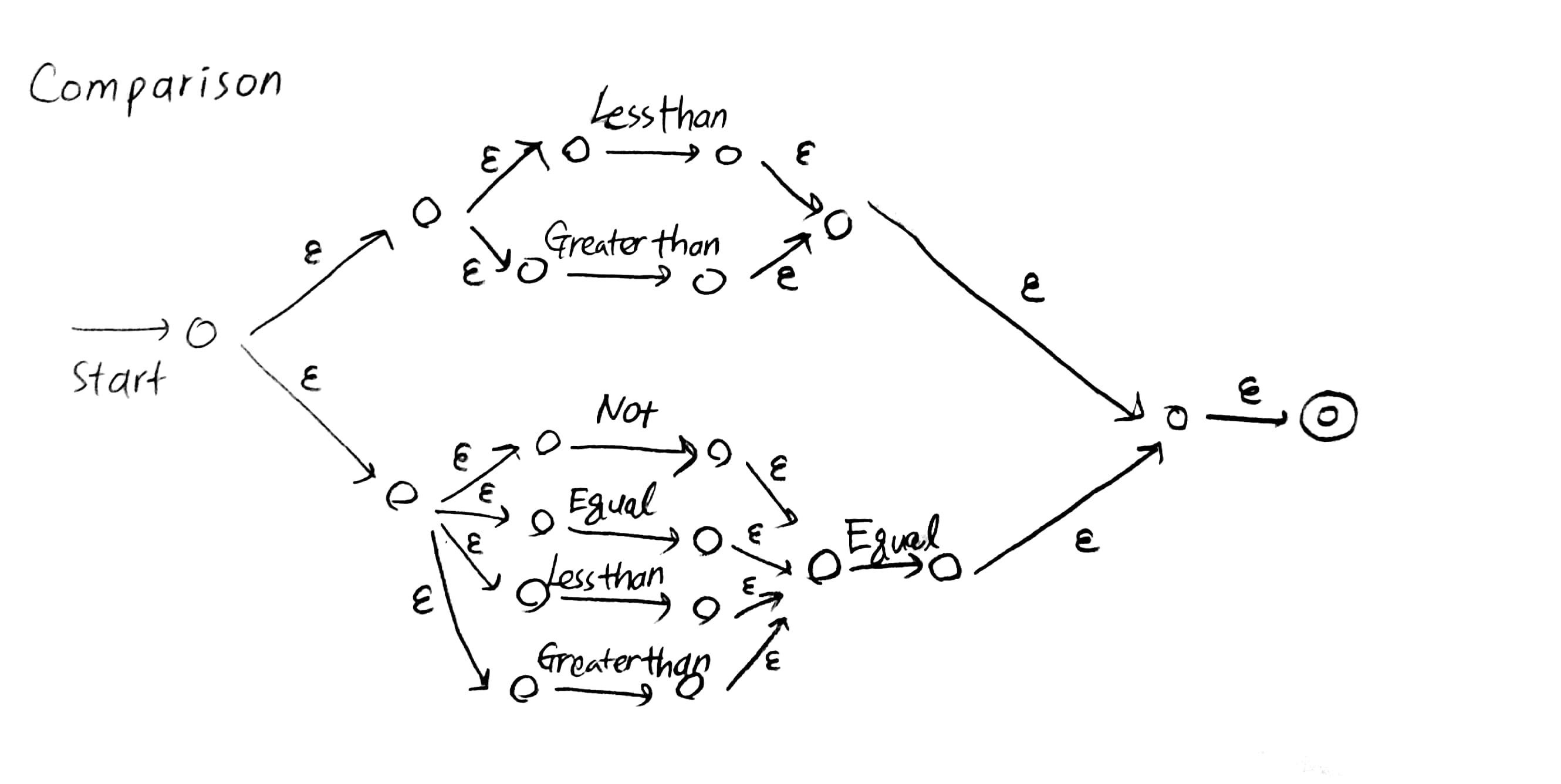
- Literal

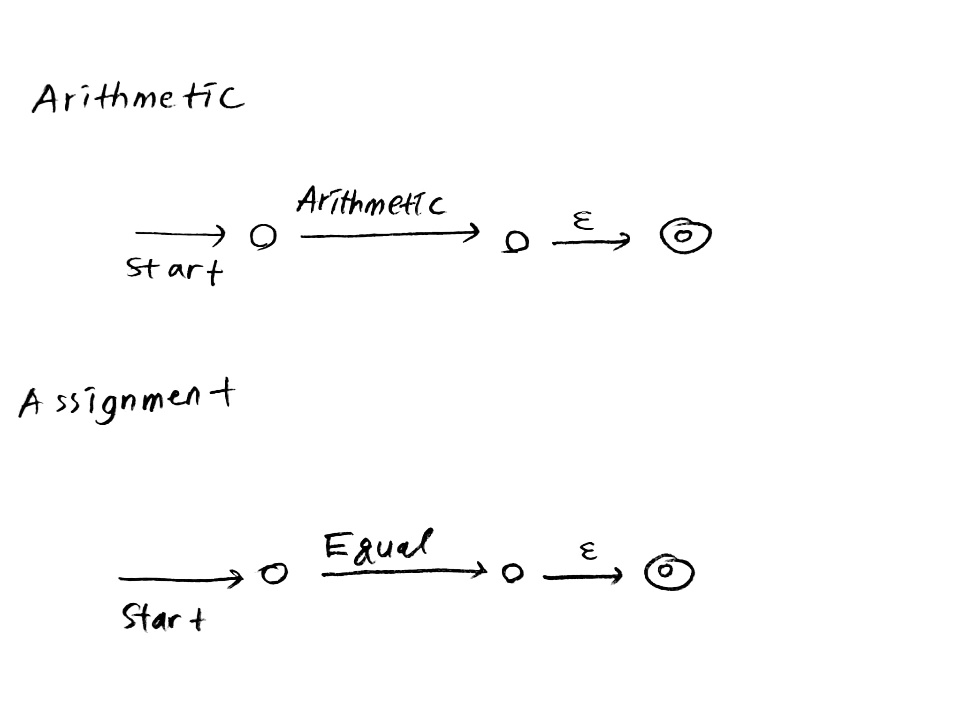
- Boolean

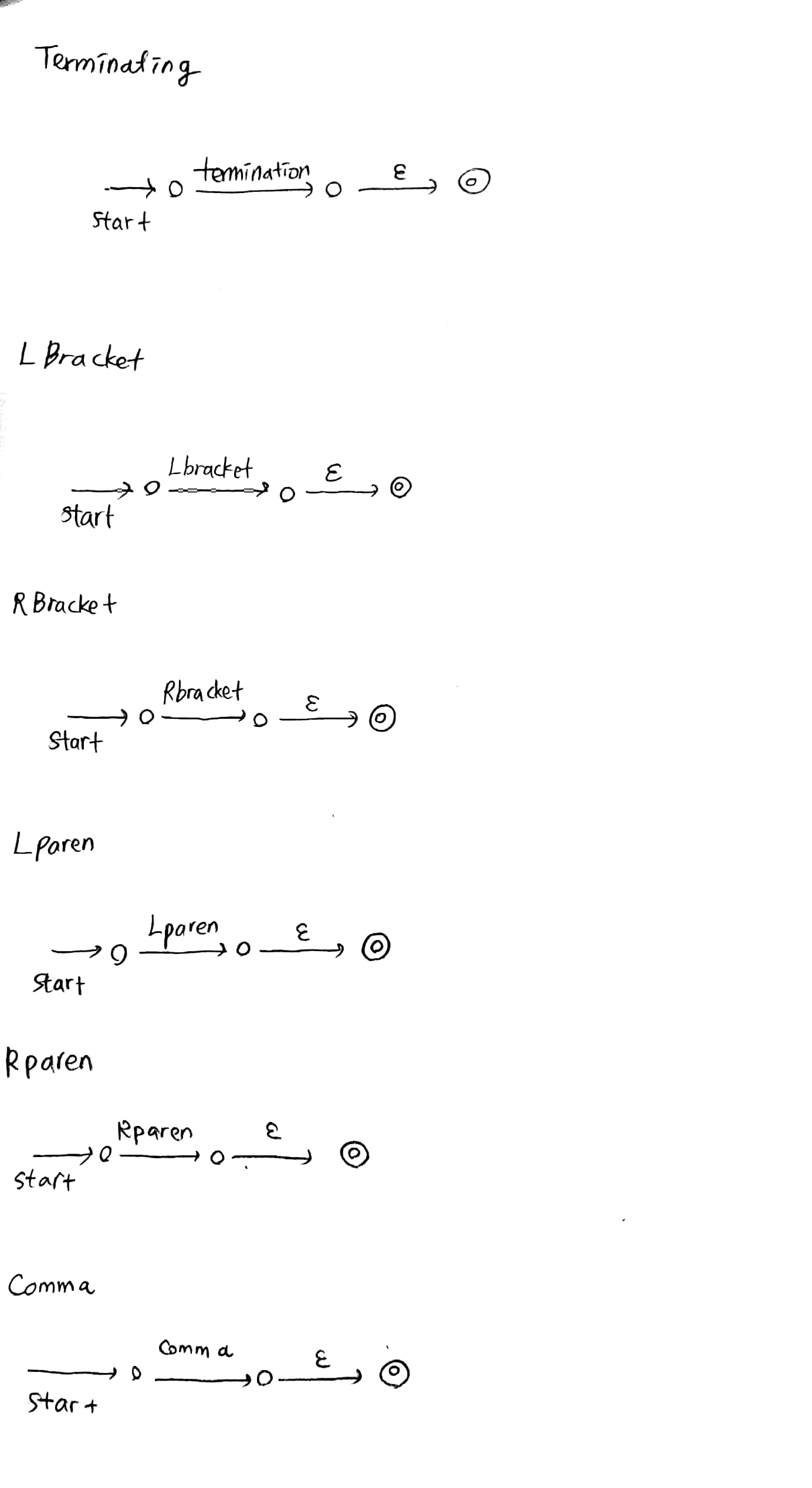
- Keyword ( if, else, for, while, return )

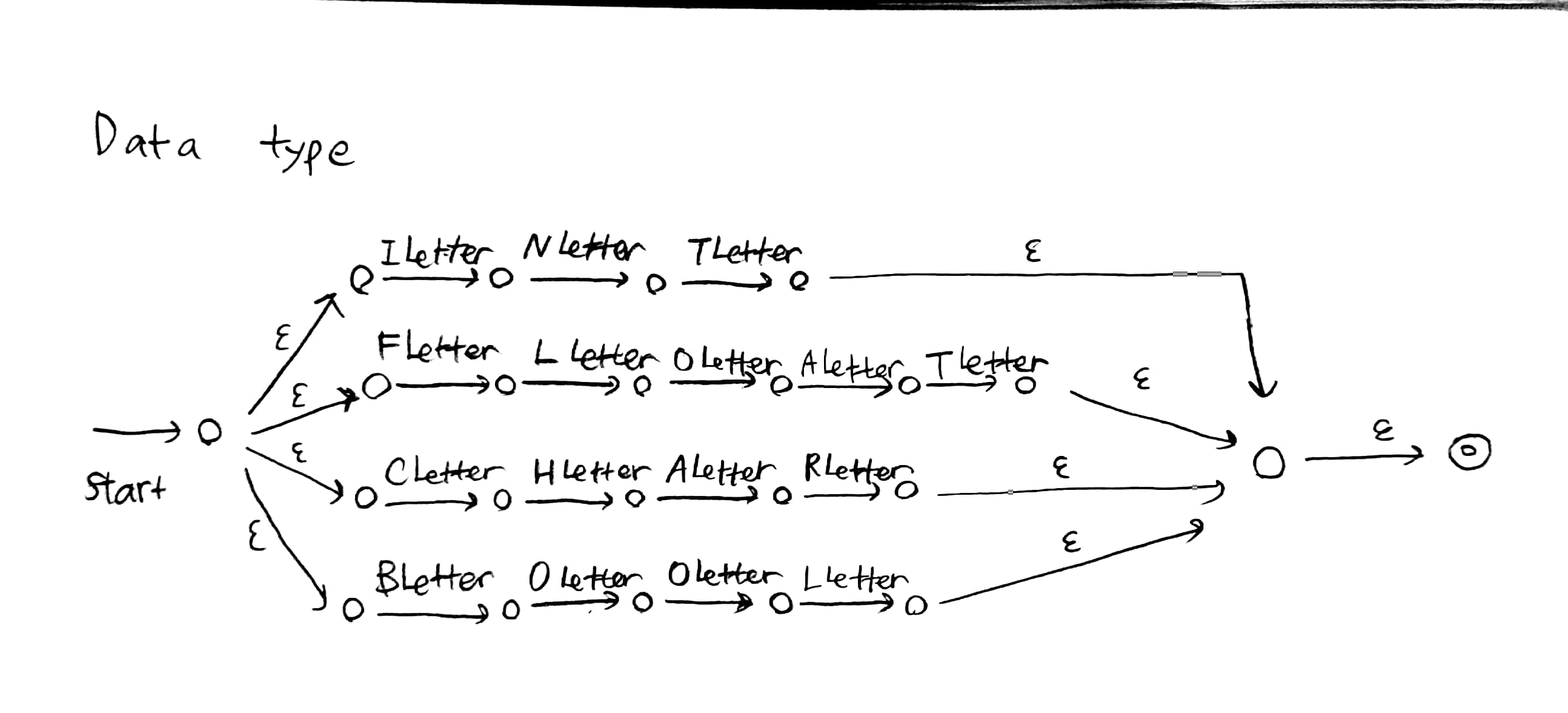
- Identifier

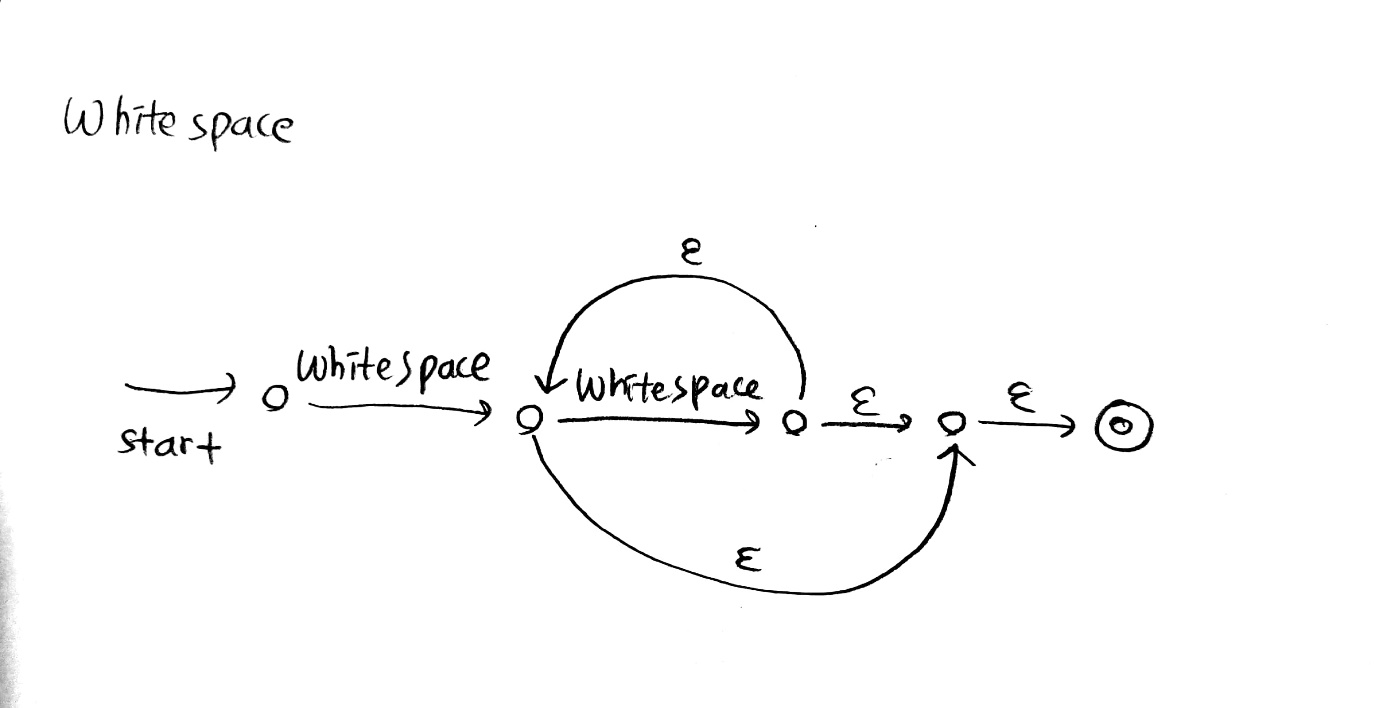
- Bitwise

- Comparison

- Assignment, Arithmetic

- Terminating, L&Rbracket, L&Rparen, Comma

- Data type

- Whitespace

2-2. NFA to DFA

-Integer

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

-Float

지도, 텍스트, 그리기이(가) 표시된 사진

자동 생성된 설명

-Literal

지도, 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명 -Boolean

- Keyword ( if, else, for, while, return )

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트, 지도이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

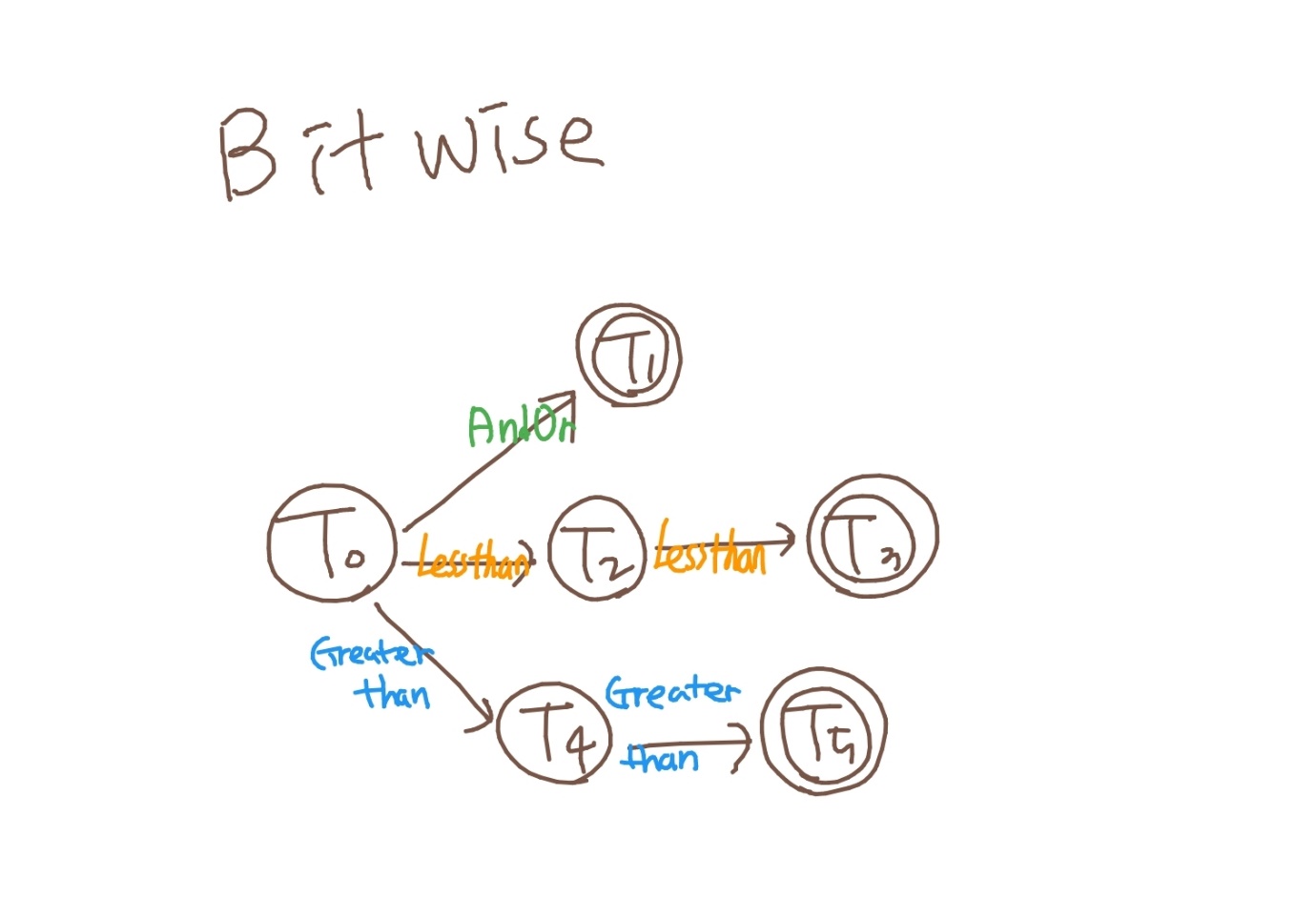
-Identifier

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

 -Bitwise

텍스트이(가) 표시된 사진

자동 생성된 설명 -Comparison

-Assignment, Arithmetic

텍스트이(가) 표시된 사진

자동 생성된 설명

-Data type

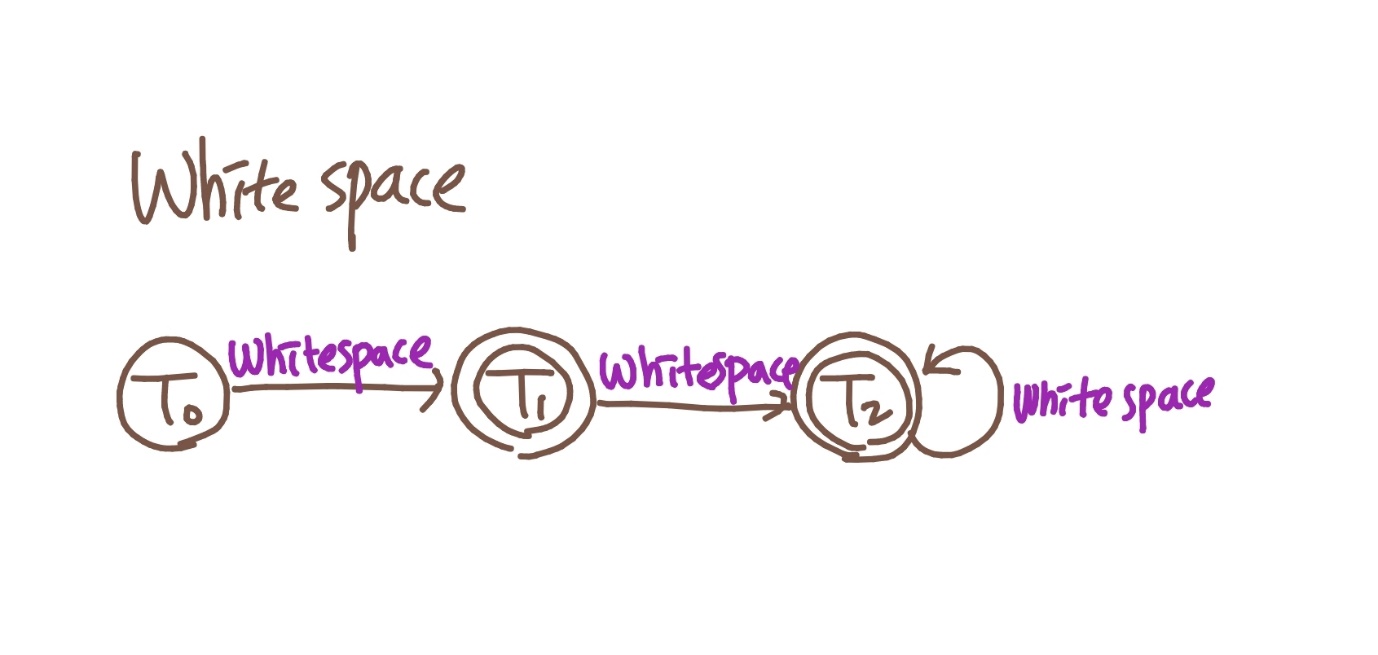
텍스트이(가) 표시된 사진

자동 생성된 설명

- Terminating, L&Rbracket, L&Rparen, Comma

텍스트이(가) 표시된 사진

자동 생성된 설명



-White Space

그리기이(가) 표시된 사진

자동 생성된 설명

그리기이(가) 표시된 사진

자동 생성된 설명

2-3. Merged DFA graph

1. **Code Implementation**

스크린샷이(가) 표시된 사진

자동 생성된 설명3-1. Overall code structure

**[Figure 3-1] Class diagram for lexical analyzer**

3-2. Code details

See the Figure 3-1 for understanding.

* **Class LexicalAnalyzer**

This class controls the entire logic.  
First, read the simple C code one by one, which is given by argument and pass it to the created transition graph.  
if the first step is over, then get the token list from the transition graph and store as an Array List type, which will be passed over to the syntax analyzer.

* **Class TransitionGraph**

This class’s instances would represent our merged DFA graph.  
So its constructor reads the graph\_info.txt file that is contained all nodes and edges information.  
The form of the text file is   
- Node: N/[state]/[node id]/arguments…  
- Edge: E/[transition condition]/[start node id]/[end node id]  
 And the transition map member variable is for checking by comparing the transition condition from the text file and the transition condition of the linked node. So, the data structure for this is hash map because we should map the direction information of the edges with the accurate transition condition.

And the buffer is for storing current processing lexeme. It would be used for storing token values or exception handling that is like -0 problem in the signed integer number.

The remain member variable is the token list which is used for storing the entire evaluated tokens, and this is implemented by Array List. After that, this would be returned in the lexical analyzer’s method body.

* **Class Alphabet Set**

This class is just defining our alphabet set.  
However, there is a special isNotGivenLetter() method. This is for distinguishing between IDENTIFIER token and KEYWORD, DATATYPE or BOOLEAN tokens.

For example, our buffer has a string “fo”. If the next letter is ‘r’, then the evaluated state is FOR state for now (The “for now” meaning is that if we have more letter or digit or underscore behind the “for” string, then it would be evaluated for IDENTIFIER). Therefore, the given letter is ‘r’. We define these given letters in the graph\_info.txt file, which are in the fourth information of Node.

For this reason, we defined overloaded isValidCondition() methods.

* **Class Transition**

This class is the data structure for edges of the transition graph.  
The member variables are the information of the edge, which are the head, tail node id and transition condition that could be thought like a weight of edge.

And the methods are just for getter/setter methods.

* **Class State Node**

This class is the data structure for nodes of the transition graph.  
So this is very similar with the Transition class. If the code read the graph\_info.txt file, then the Node information would be entered these State Node instances.

The member variables are the information of the node, which are the id number of the node instance, and state, linked node that is every nodes linked to this node, and keyword\_identifier that is used for isNotGivenLetter(), which is explained at the Class Alphabet Set section.

And the remaining code is just for getter/setter methods.

* **Class Result Printer**

This class writes the result file.  
Moreover, if there is a lexical error, the lexical analyzer instance gives the control to this class instance. then it opens the result file and writes the errors.

* **Class Main**

This class is just for start point of the project.

3-3. Main errors and exception handlings

This part refers to the exception about combined DFA error.   
For example, 234-123 statement is recognized for <INTEGER, 234><INTEGER, -123>. This is not an error in our combined DFA graph, but the translation of this statement that we want is the minus operation of two INTEGER numbers.  
 In this part, we will explain how we solve this kind of problems.

Plus, there are some floating number problems. That is, all of things are about INTEGR and FLOAT problems.

* **Minus operation Issues**

As we’ve already talked, there are some problems in minus operation statement. Because we can take the minus symbol as not only ARITHMETIC token, but also INTEGER or FLOAT token. Therefore, we must distinguish these two cases.

First, when the minus symbol is in the computational expression, it must be recognized for arithmetic operation. And second, if there is only one number with minus symbol, which is like -123 or -34.25, it would be recognized for INTEGER or FLOAT number.

We distinguished these two cases by checking the token followed by minus symbol. If the token is INTEGER or FLOAT or IDENTIFIER, the statement that we are checking is operating statement (e.g. 123-456 or integernum-23 or 123.4-0.2). Otherwise, the minus symbol is with a single INTEGER or FLOAT number.

* **Successing Constant Numbers Issues**

There are some ambiguous statements with FLOAT numbers. For example, the statement 5.2430.4 can’t be recognized for the single float number, but it could be distinguished like <FLOAT, 5.243>,<FLOAT,0.4>, the successing constant floating numbers. Of course, this is not a valid statement in the C code. But the judgement about this is not a business of the lexical analyzer. It can be filtered out by syntax analyzer, because the successing floating numbers is not a valid syntax. So, we allow this kind of statement in our lexical analyzer.

Let’s think about two statements, -0.00.0 and -0.00. statement.  
The first one is can be recognized for <FLOAT, -0.0>,<FLOAT,0.0> but the second is can’t be any kind of token. This is because dot symbol can’t exist alone.

We resolved this problem by checking previous node and previous symbol. If the previous node can be finished by FLOAT token, then we save this token and revaluate the present node from the beginning of the transition graph.

Then what about -0.0000.0 statement? This can be recognized for <FLOAT,-0.0><INTEGER,0><INTEGER,0> <FLOAT,0.0> but in our team77 compiler, we will not allow this statement, because these are very useless lexemes which will be filtered out in our syntax analyzer.

We think this problem would never be resolved by the DFA graph and there must be many exception handlings. Many exception handlings for eventually invalid statement is a contradiction.