Section 1: Lexical Specifications

The following regular expressions define the lexical elements of the language:

```
letter    ::= a..z | A..Z
digit     ::= 0..9
nonzero   ::= 1..9
alphanum  ::= letter | digit | _

id        ::= letter alphanum*
integer   ::= nonzero digit* | 0
float     ::= integer fraction [e[+|-] integer]
fraction  ::= .digit* nonzero | .0

inlinecmt ::= // .* \n
blockcmt  ::= /* .* */
```

Operators: ==, <>, <, >, <=, >=, +, -, *, /, =

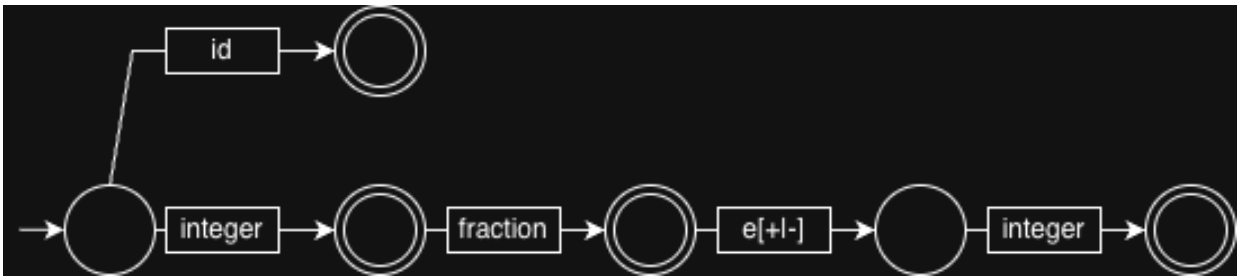
Punctuation: (,), {, }, [,], ;, ,, ., :, ::

Reserved words: if, then, else, while, class, do, end, public, private, or, and, not, read, write, return, inherits, local, void, main, integer, float

Section 2: Finite State Automaton.

Master NFA Diagram

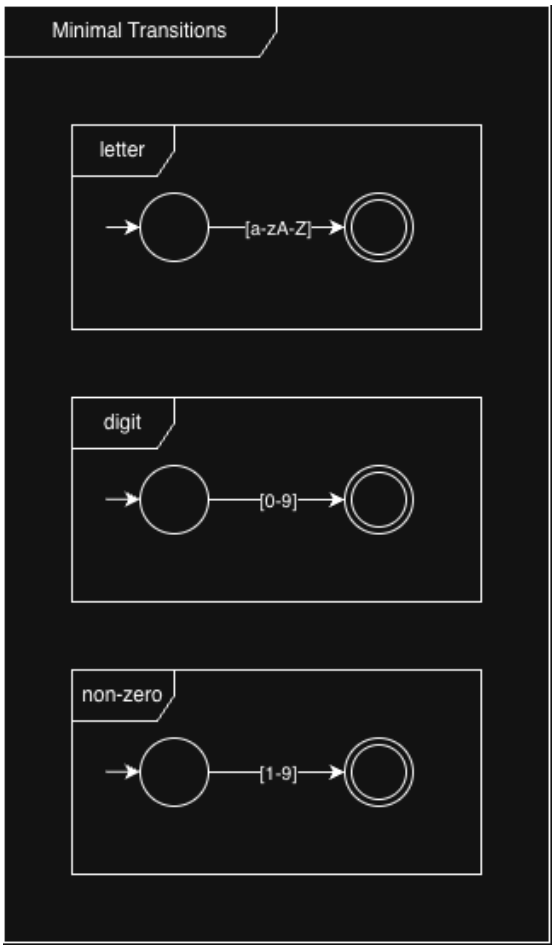
The master NFA shows the overall flow of the lexical analyzer, branching from the start state to either the ID path or the Number path (integer → fraction → exponent).



Minimal Transitions

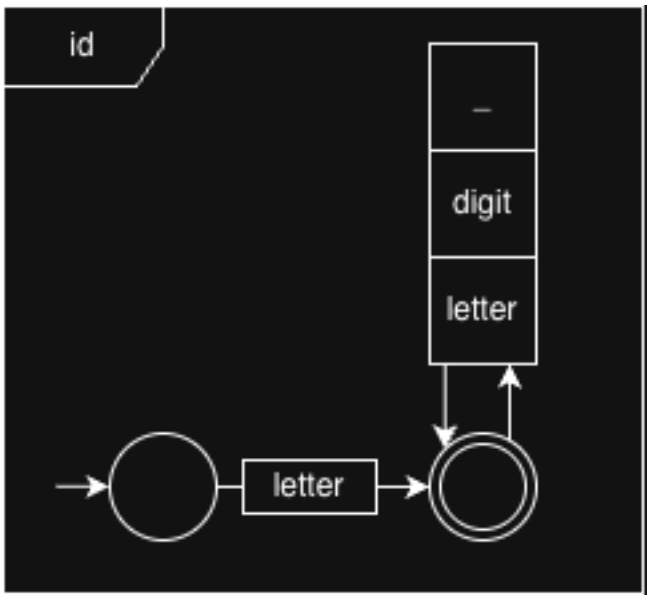
The following diagram defines the basic character classes used throughout the NFAs:

- **letter:** a-z, A-Z
- **digit:** 0-9
- **nonzero:** 1-9



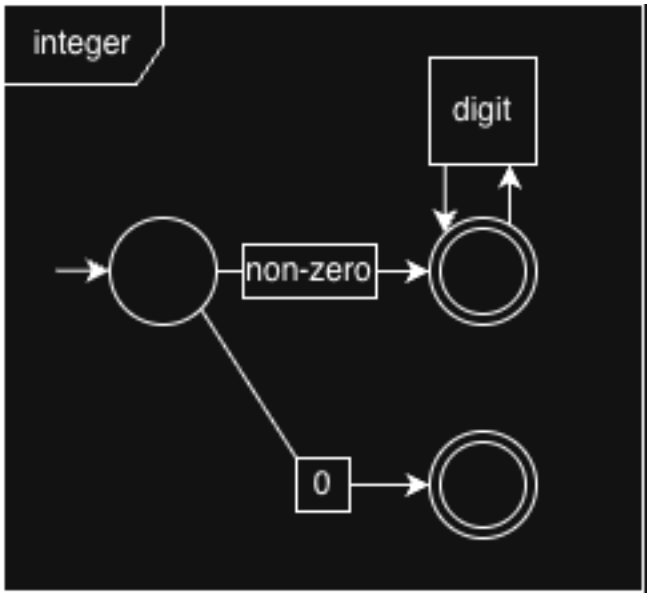
ID NFA

Recognizes identifiers matching `letter alphanum*`. Starts with a letter, then loops on letters, digits, or underscores. Upon acceptance, the lexeme is checked against a reserved words lookup table to determine if it is a keyword (e.g., `if`, `while`, `class`) or a regular identifier.



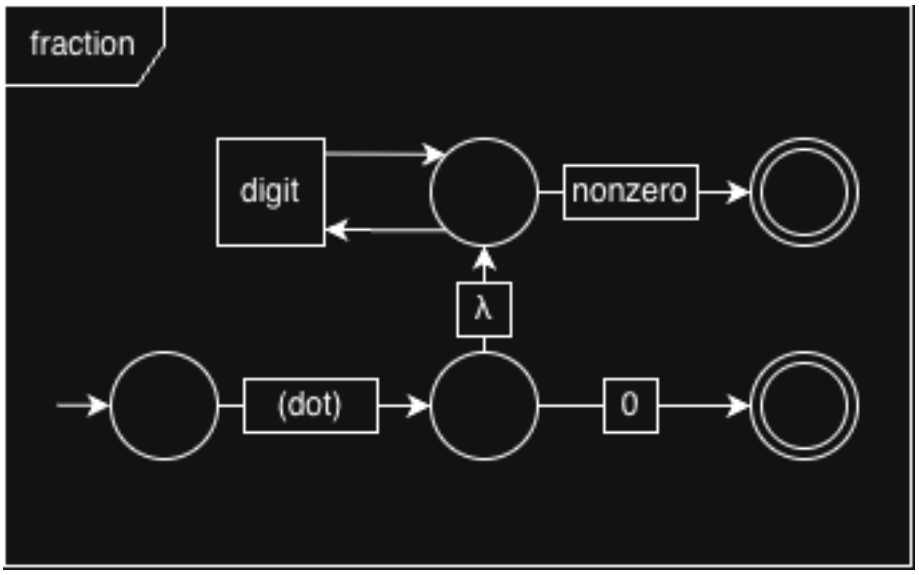
Integer NFA

Recognizes integers matching `nonzero digit* | 0`. Either starts with a nonzero digit and loops on digits, or accepts a single zero.



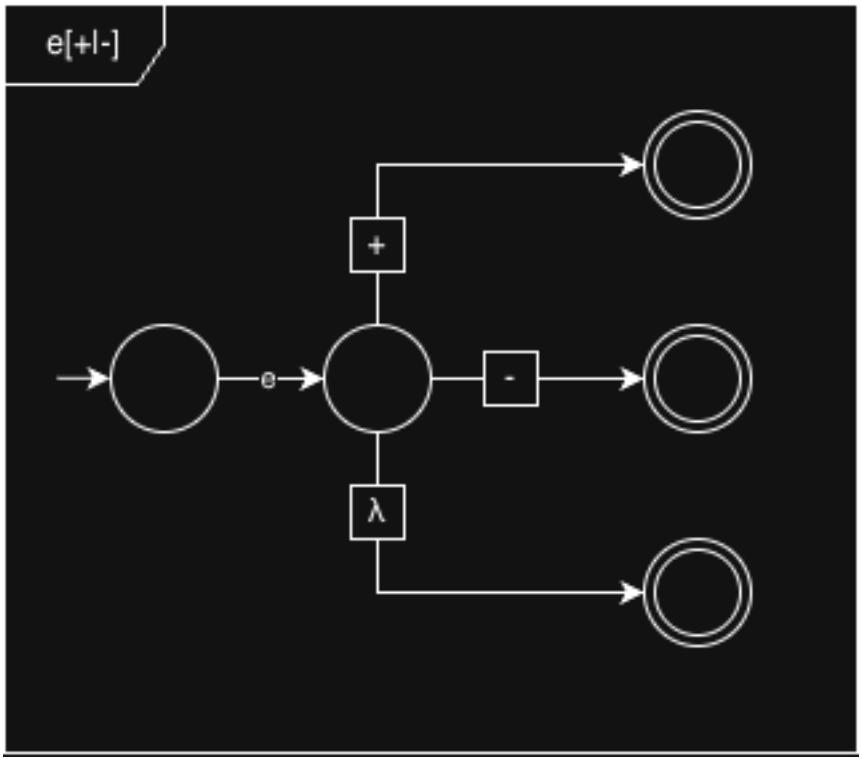
Fraction NFA

Recognizes the fractional part of a float matching `.digit*nonzero | .0`. Starts with a dot, loops on digits, and must end with a nonzero digit (or be just `.0`).



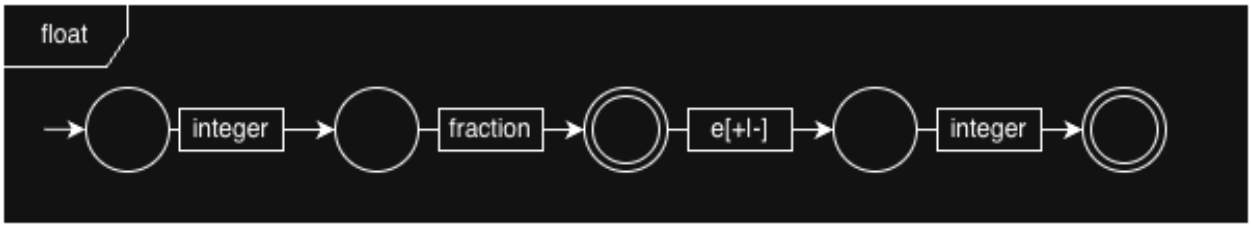
Exponent Sign NFA

Recognizes the exponent sign part `e[+|-]`. After the letter 'e', optionally accepts '+' or '-', or continues directly (λ transition) if no sign is present.



Float NFA

Recognizes complete floats matching `integer fraction [e[+|-] integer]`. Combines the integer, fraction, and exponent NFAs. Lambda (λ) transitions connect these sub-NFAs, allowing acceptance at intermediate points.



Operators and punctuation are recognized through direct character matching. When the scanner encounters any of the reserved characters listed below, it immediately produces the corresponding token:

Single Character	Token	Two Characters	Token
+	PLUS	==	EQ
-	MINUS	<>	NEQ
*	MULT	<=	LEQ
/	DIV	>=	GEQ
=	ASSIGN	::	COLONCOLON
<	LT		
>	GT		
(	LPAREN		
)	RPAREN		
{	LBRACE		
}	RBRACE		
[	LBRACKET		
]	RBRACKET		
;	SEMICOLON		
,	COMMA		

.	DOT		
:	COLON		

For two-character operators (e.g., ==, <=), the scanner uses one-character lookahead. If the second character completes a valid two-character operator, that token is produced; otherwise, the single-character token is returned.

Comments

Comments are handled by two additional patterns:

- **Inline comment:** Starts with `//`, reads any characters until newline
- **Block comment:** Starts with `/*`, reads any characters until `*/`

Error Handling

The lexical analyzer implements error recovery, continuing to scan after encountering errors. The following error types are detected:

Error Type	Description	Example
`INVALIDCHAR`	Character not recognized by any NFA	`@`, `#`, `$`, `%`
`INVALIDID`	Identifier starting with underscore	`_abc`, `_123`
`UNTERMINATEDCMT`	Block comment reaching EOF without closing `*/`	`/* comment without end`

Invalid Number Handling

Per the assignment specification, invalid numbers are not reported as errors but are instead split into valid tokens:

Input	Output Tokens	Reason
`0123`	`[INTEGER, 0] [INTEGER, 123]`	Leading zero
`12.340`	`[FLOAT, 12.34] [INTEGER, 0]`	Trailing zero in fraction
`12.34e01`	`[FLOAT, 12.34e0] [INTEGER, 1]`	Leading zero in exponent

This splitting is implemented using character backtracking (via `PushbackReader`) when an invalid pattern is detected.

NFA to Implementation

Each sub-NFA corresponds to a dedicated scanner method in the implementation: `scanId()`, `scanInteger()`, `scanFraction()`, `scanExponent()`. The λ-transitions are implemented as optional continuation checks—after recognizing a valid token, the scanner checks if additional characters would form a longer valid token.

Section 3: Design Description.

Overall Structure

The lexical analyzer is implemented using a modular design with separate classes for token representation, lexical analysis, and driver functionality. The design phase used NFAs (Nondeterministic Finite Automata) to model each token type, which were then implemented as deterministic scanner methods using lookahead techniques.

Components and Classes

1. Token.java

Role: Data structure representing a single lexical token.

Contains:

- `TokenType` enum - Defines all token types (ID, INTEGER, FLOAT, operators, punctuation, reserved words, and error types)
- `lexeme` - The actual string matched from source
- `line` - Line number where token appears
- `toFlaciString()` - Converts token to Flaci-compatible format
- `isError()` - Checks if token represents a lexical error

2. LexicalAnalyzer.java

Role: Core scanner implementing the NFAs for token recognition.

Key Methods:

- `nextToken()` - Main entry point; returns the next token from input
- `scanId()` - Implements ID NFA: `letter alphanum*`
- `scanInteger()` - Implements Integer NFA: `nonzero digit* | 0`
- `scanFraction()` - Implements Fraction NFA: `.digit*nonzero | .0`
- `scanExponent()` - Implements Exponent NFA: `e[+|-] integer`
- `scanNumber()` - Master function orchestrating integer/float recognition
- `scanBlockComment()` / `scanInlineComment()` - Comment handling

Data Structures:

- `PushbackReader` - Allows character backtracking for lookahead
- `HashMap<String, TokenType>` - Reserved words lookup table

3. ILexicalAnalyzer.java

Role: Interface defining the lexical analyzer contract.

Contains: Single method `nextToken()` that all lexer implementations must provide.

4. LexDriver.java

Role: Driver program for processing source files and generating output.

Functionality:

- Reads `.src` input files
- Generates `.outlextokens` - Full token information (type, lexeme, line)
- Generates `.outlextokensflaci` - Token stream for Flaci tool
- Generates `.outlexerrors` - Error messages with descriptions and locations

Design Decisions

- **Modular Scanner Methods:** Each NFA is implemented as a separate method, making the code easier to debug and maintain.
- **Error Recovery:** The lexer continues processing after encountering errors, allowing all errors to be reported in a single pass.
- **Token Splitting:** Invalid numbers (leading zeros, trailing zeros) are split into valid tokens rather than reported as errors, following the assignment specification.

Section 4: Use of tools

The only main tool that was used was Drawio to create the NFAs. There were no other tools used to create this code/validate the design. Everything was done from scratch, without any assistive tools. The strategy was to make atomic NFAs, then consider them as black boxes when combining them into a larger NFA. Each sub-NFA can be its own function and can be tested individually. Since the NFAs have been very small and simple, validation was easy through programming. Generative AI was used to format this document but was not used in the design / implementation.