Compiler Theory

The 1st Program Assignment – Scanner Construction
21900628 – Sechang Jang

Index

Theoretical Foundations

Regular Expression for Each Lexical Unit

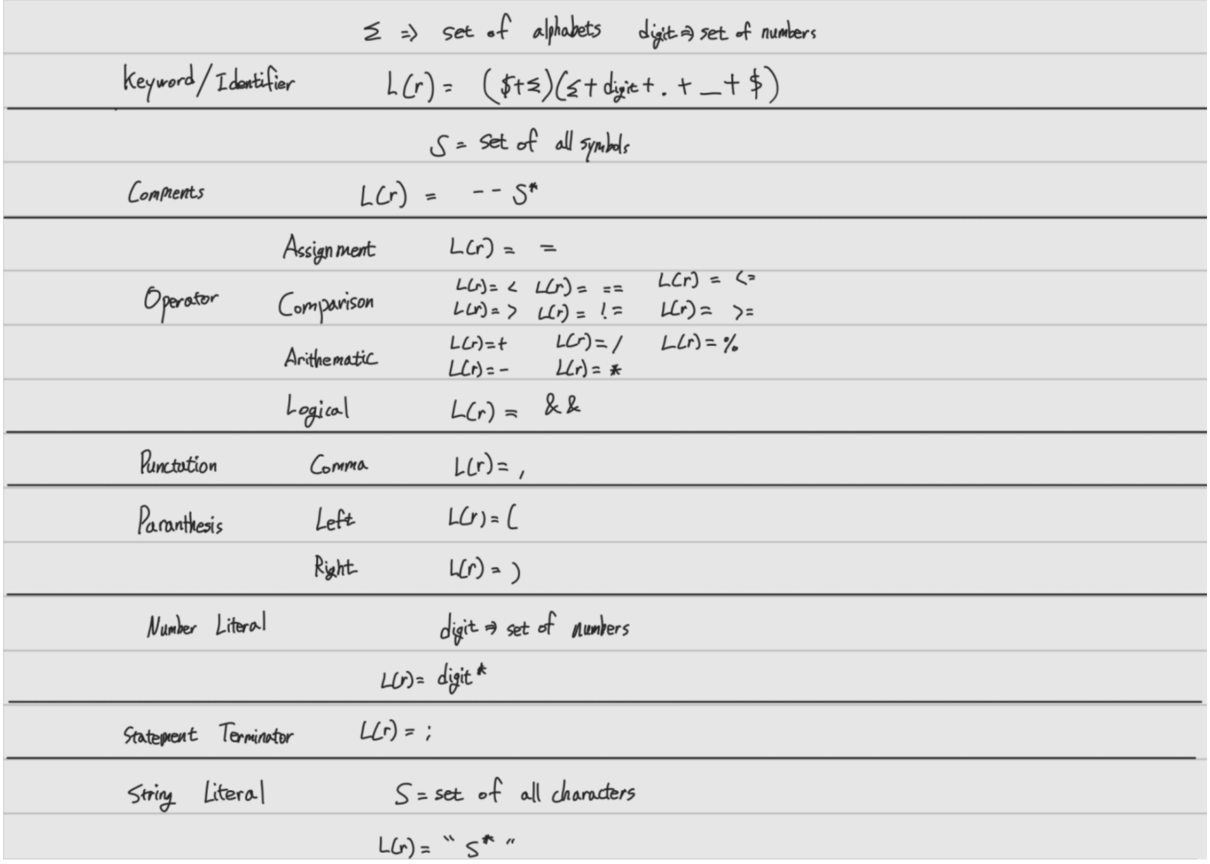
What is regular expression?

Regular expressions are used to define the patterns of tokens that make up a programming language. Each token represents a lexical unit, such as keywords, identifiers, literals, and punctuation symbols.

Lexical analysis is the first phase of compilation, where the source code is scanned and divided into tokens. Regular expressions are employed to recognize these tokens by matching patterns defined for each lexical unit. This process is often implemented using finite automata or regular expression engines.

Types of tokens

- Keyword
    - program, int, if, begin, print_line, end, else_if, else, while
- Identifier
- Comment
- Operator
    - Assignment
    - Comparison
        - Less than
        - Equal to
        - Not equal
        - Greater than
        - Greater than or equal to
        - Less than or equal to
    - Arithmetic
        - Multiplication
        - Addition
        - Subtraction
        - Division
        - Modulo
    - Logical
        - And
        - Or
- Punctuation
    - Comma Identifier
- Parenthesis
    - Left
    - Right
- Statement Terminator
- Number Literal
- String Literal



		Σ => set of alphabets digit ⇒ set of numbers
Keyword/Identifier		L(r) = ($+Σ)(Σ+digit+.+_+$)
		S = set of all symbols
Comments		L(r) = -- S*
Operator	Assignment	L(r) = =
	Comparison	L(r) = < L(r) = == L(r) = <= L(r) = > L(r) = != L(r) = >=
	Arithematic	L(r) = + L(r) = / L(r) = % L(r) = - L(r) = *
	Logical	L(r) = &&
Punctation	Comma	L(r) = ,
Paranthesis	Left	L(r) = (
	Right	L(r) =)
Number Literal		digit ⇒ set of numbers
		L(r) = digit*
Statement Terminator		L(r) = ;
String Literal		S = set of all characters
		L(r) = " S* "

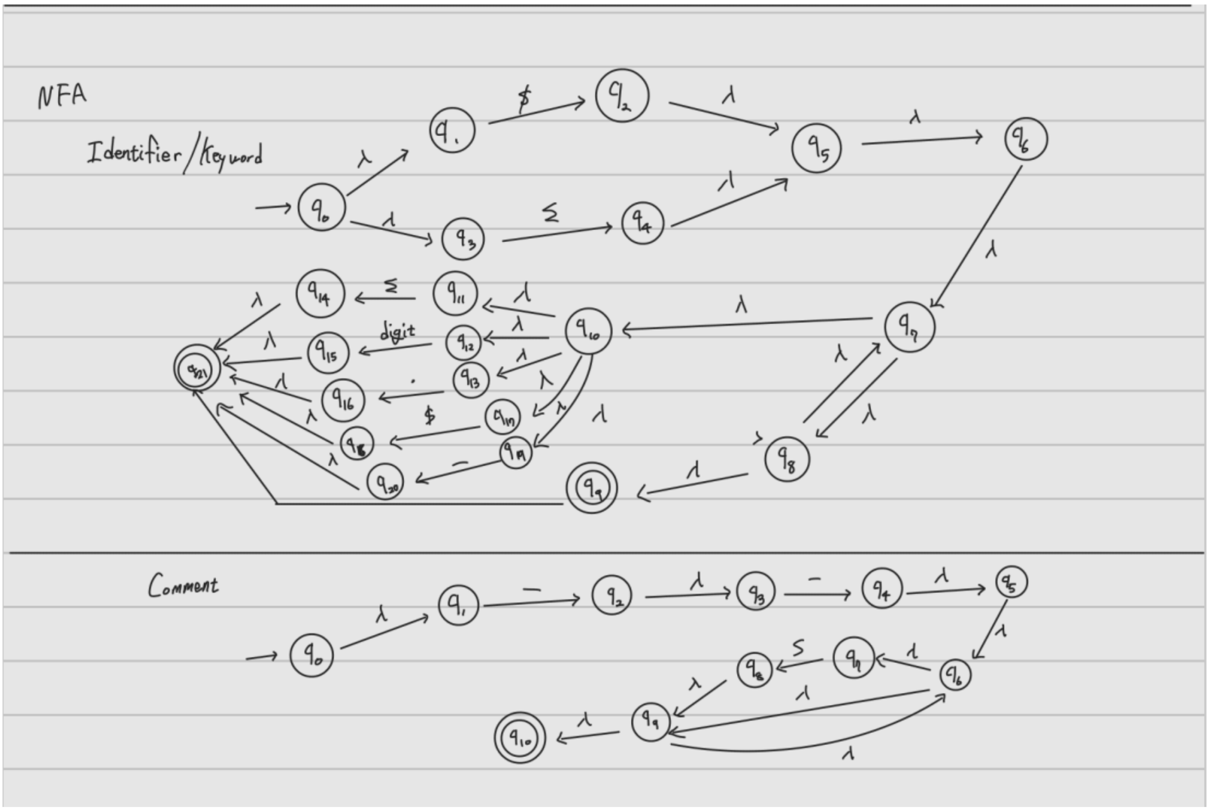
NFA - DFA Construction

Finite Automata in Lexical Analysis

Regular expressions are often translated into finite automata to efficiently recognize patterns in the input source code. Deterministic finite automata (DFA) or nondeterministic finite automata (NFA) can be constructed from regular expressions, allowing for efficient tokenization during lexical analysis.

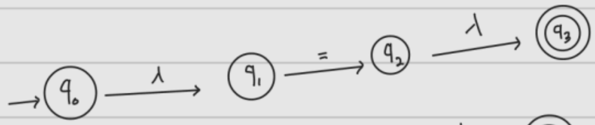
NFA Construction

Nondeterministic Finite Automata, known as NFA, comprises a finite set of states, transitions between states, an input alphabet, a start state, and one or more accepting states. Unlike Deterministic Finite Automata (DFA), NFAs allow for nondeterminism, meaning that from a given state and input symbol, there can be multiple possible transitions. This feature allows NFAs to explore multiple computation paths simultaneously, making choices based on the input symbol and current state. NFAs can also include epsilon transitions, which enable transitions without consuming any input.


NFA
Identifier/Keyword
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
q_{11}
q_{12}
q_{13}
q_{14}
q_{15}
q_{16}
q_{17}
q_{18}
q_{19}
q_{20}
q_{21}
λ
$
Σ
digit
.
_
Comment
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
λ
–
S

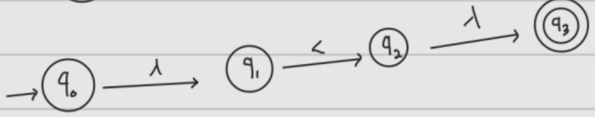
Operators

- Assignment

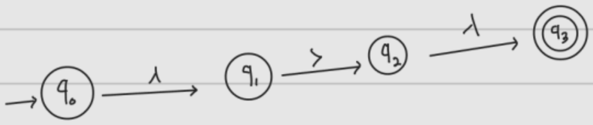


Comparison

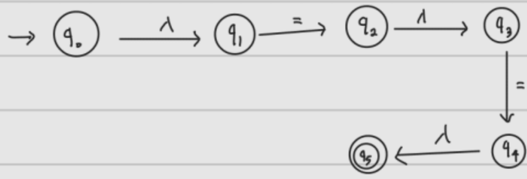
- Less than



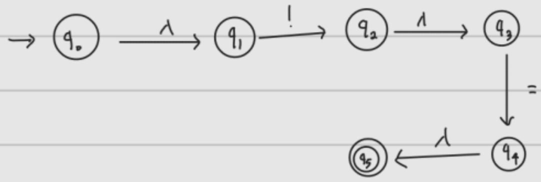
- greater than



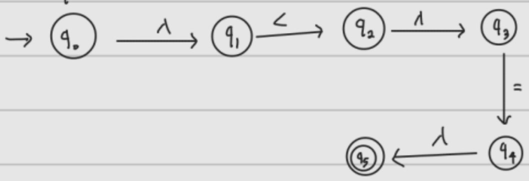
- equal to



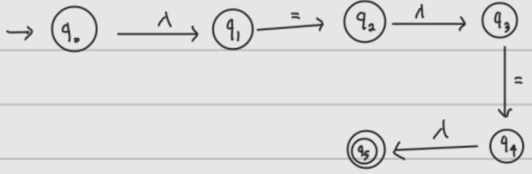
- Not equal to



- Less than or equal to

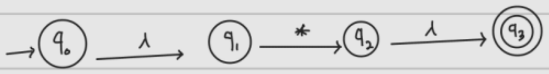


- Greater than or equal to

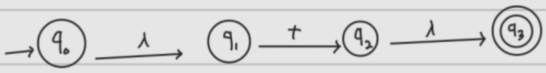


Arithematic

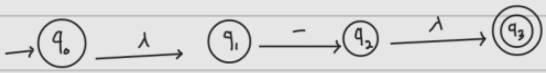
- Multiplication

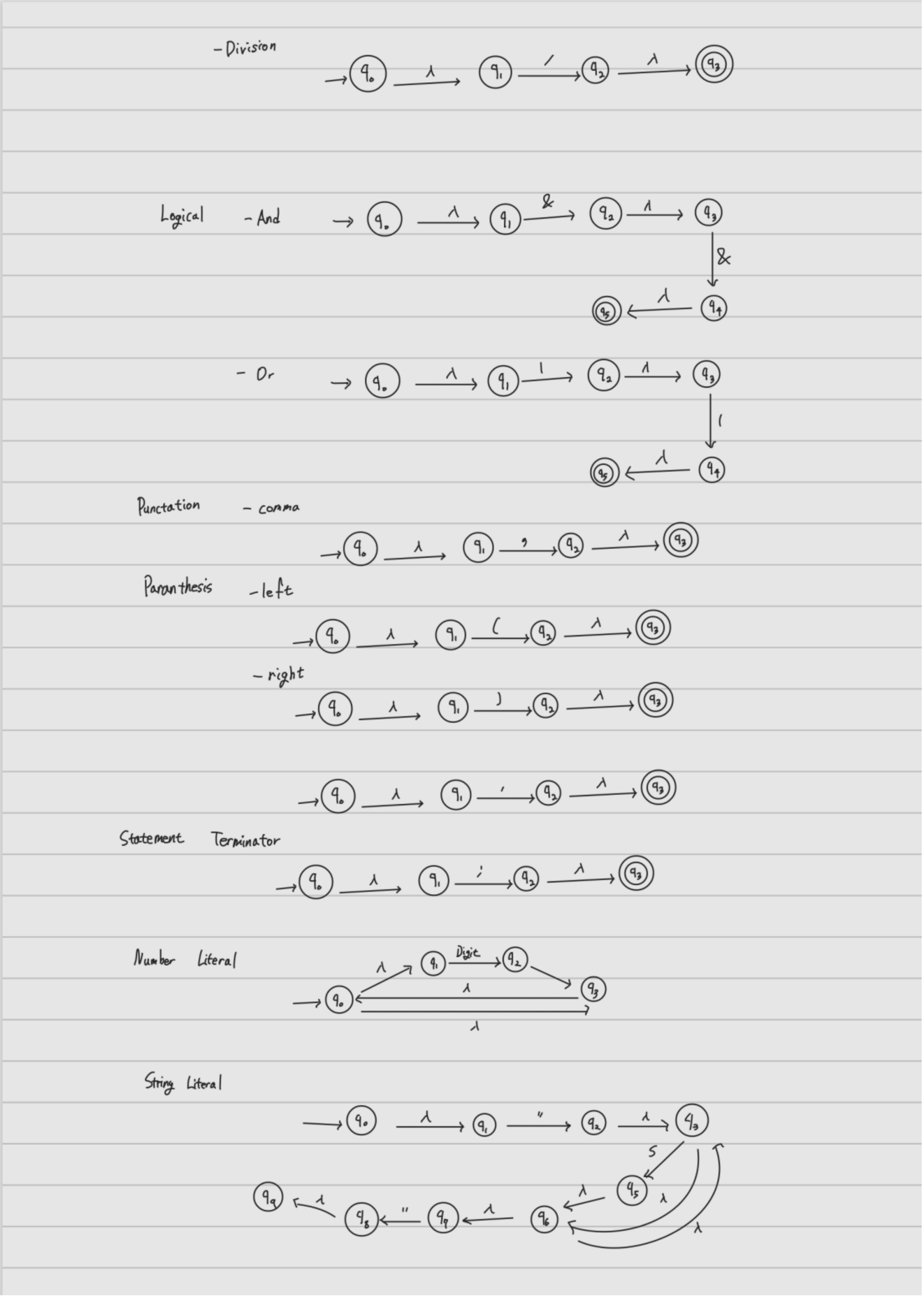


- Addition



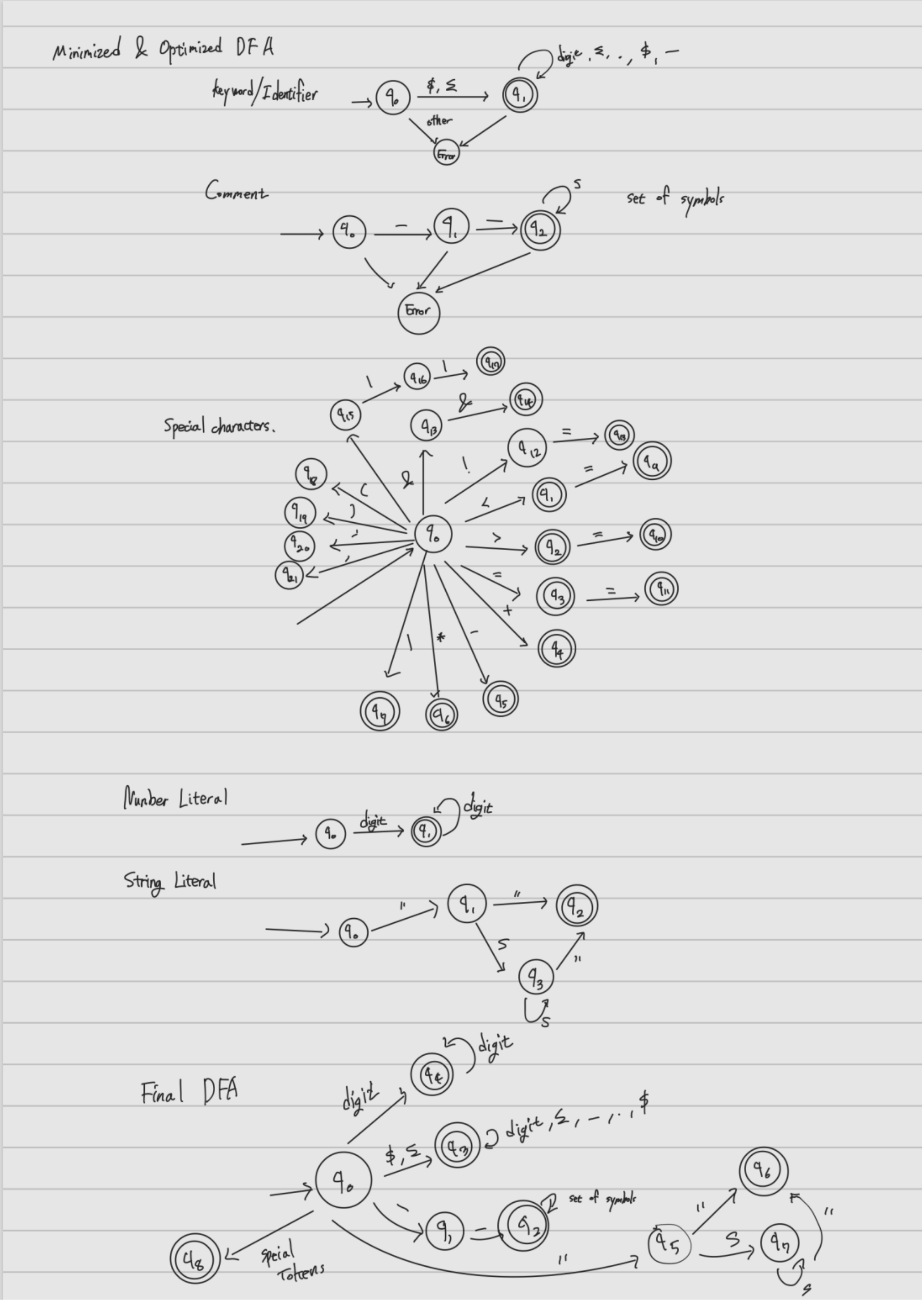
- Subtraction





DFA Construction

Deterministic Finite Automata, known as DFA, are finite state machines designed to accept or reject strings based on whether the machine reaches an accepting state after processing the entire input. Unlike NFAs, DFAs do not permit nondeterministic transitions. For each state and input symbol, there is exactly one next state in a DFA.

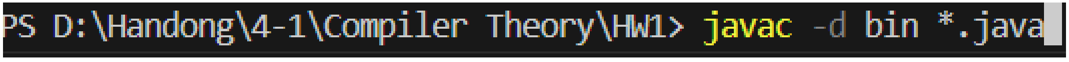


About special token, it means that it is delivered to DFA of special characters.

User Manual

To compile java files, run

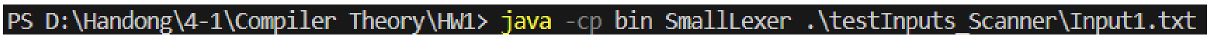
javac *.java



```
PS D:\Handong\4-1\Compiler Theory\HW1> javac -d bin *.java
```

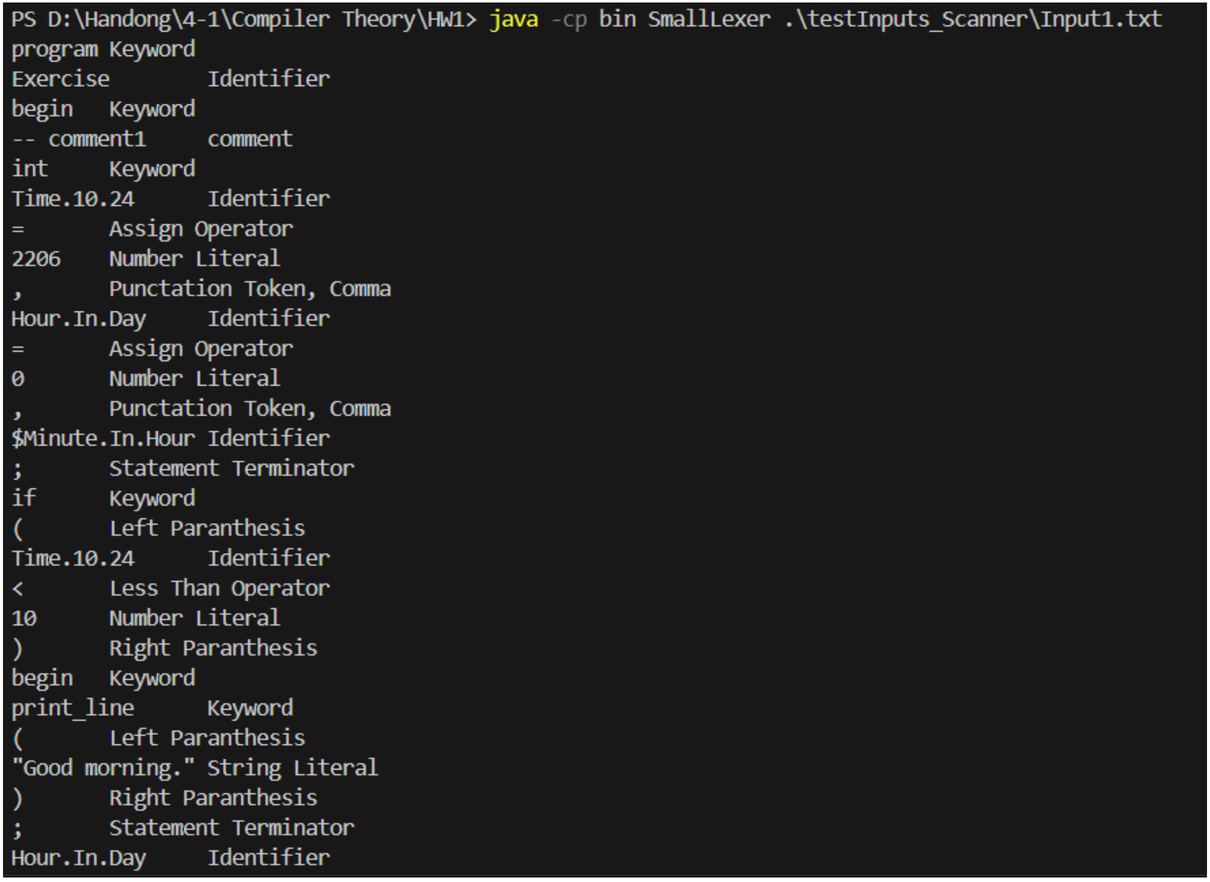
To run compiled java file,

You need to put file path using command line interface



```
PS D:\Handong\4-1\Compiler Theory\HW1> java -cp bin SmallLexer .\testInputs_Scanner\Input1.txt
```

Example run



```
PS D:\Handong\4-1\Compiler Theory\HW1> java -cp bin SmallLexer .\testInputs_Scanner\Input1.txt
program Keyword
Exercise        Identifier
begin   Keyword
-- comment1     comment
int     Keyword
Time.10.24      Identifier
=       Assign Operator
2206    Number Literal
,       Punctation Token, Comma
Hour.In.Day     Identifier
=       Assign Operator
0       Number Literal
,       Punctation Token, Comma
$Minute.In.Hour Identifier
;       Statement Terminator
if      Keyword
(       Left Paranthesis
Time.10.24      Identifier
<       Less Than Operator
10      Number Literal
)       Right Paranthesis
begin   Keyword
print_line      Keyword
(       Left Paranthesis
"Good morning." String Literal
)       Right Paranthesis
;       Statement Terminator
Hour.In.Day     Identifier
```

Developer’s Note

Thinking about implementation

Big DFA

In the initial consideration of constructing a big DFA table, the concern was that it would encompass transitions for every possible lexical element in the toy language. However, upon closer examination, it became evident that certain parts of the language, especially operators, have limited transitions. For instance, in a typical toy language, there might be only a handful of operators with specific meanings. Representing each possible transition in a single DFA table would result in a significant amount of unused space being allocated, which is inefficient both in terms of memory consumption and processing time.
To address the inefficiency of a single big DFA table, the decision was made to adopt a separate DFA approach with lookahead capability. By breaking down the lexical analysis process into smaller, more focused DFAs, each responsible for recognizing specific lexical elements or patterns, we can tailor the transition rules more precisely. This granularity allows us to avoid unnecessary transitions and optimize space usage while still maintaining the ability to recognize the entire language effectively.

However, to support large DFA transition for the future, I created a DFA class that receives a DFA table, accepts a state list, and transitions states for reusability. The creation of a DFA class serves multiple purposes. Firstly, it encapsulates the logic and behavior of a DFA, providing a clear and reusable implementation that can be easily integrated into the lexer. By accepting parameters such as the DFA table, state list, and transition state, the DFA class becomes highly customizable and adaptable to different lexical analysis requirements. Furthermore, by centralizing DFA functionality within a class, future expansions or modifications to the lexer can be implemented more seamlessly, as the DFA logic remains isolated and can be reused across different parts of the lexer codebase. This promotes code reusability, modularity, and maintainability, essential qualities for the long-term sustainability of the lexer implementation.

Lookahead and Advance

In lexer implementation, lookahead is used to peek ahead in the input stream to determine the appropriate token to generate. This lookahead mechanism allows to handle cases where the next token's recognition depends on subsequent input characters.
Additionally, the concept of advancing the input stream pointer is essential for progressing through the input while maintaining the correct state within the lexer.

Special Tokens

Token Delimiter

Token delimiters serve a critical role in the lexical analysis process by delineating the boundaries between individual tokens within the input stream. Examples of token delimiters include whitespace characters, such as spaces, tabs, and newlines, as well as specific punctuation marks or symbols that mark the end of one token and the beginning of another. Also, the operators. In the context of the lexer implementation, identifying and handling these delimiters is essential for accurately segmenting the input stream into distinct tokens.

The lexer employs a recursive approach to examine each token for the presence of delimiters. This recursive process involves iteratively analyzing the characters comprising a token and checking whether any delimiter characters are encountered. Upon encountering a delimiter, the lexer determines that the current token is complete and emits it as a separate lexical element. The lexer then resumes scanning the input stream to identify the next token, recursively applying the delimiter detection process as needed. By recursively examining tokens for delimiters, the lexer ensures thorough and accurate tokenization of the input, enabling downstream processing stages to operate on well-defined lexical units.