Chapter 1

Lexical Analysis

An implementation of lexical analyzer must do two things:

1. Recognize substrings corresponding to tokens, i.e. the lexemes.
2. Identify the token class of each lexeme. A token is the <token class, lexeme> pair.

The goal of lexical analysis is to partition the string. It is implemented by reading left-to-right, recognizing one token at a time. "Lookahead" might be required to decide where one token ends and the next token begins.

1.1 Tools: regular expression and finite automata

1.1.1 Lexical specification & regular expression

1. Write a regular expression for the lexemes of each token class.
   - Number = digit+
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - Openpar = '('
   - ...
2. Construct R, matching all lexemes of all token classes.
   R = Keyword + Identifier + Number + ... $= R_1 + R_2 + \dots$
3. Let input be $x_1 x_2 \dots x_n$. For $1 \leq i \leq n$, check if $x_1 \dots x_i \in L(R)$.
4. If yes, then we know that $x_1 \dots x_i \in L(R_j)$ for some j.
5. Remove $x_1 \dots x_i$ from input and go to 3.

Problems & ambiguities:

1. How much input is used? What if we have

$$x_1 \ldots x_i \in L(R)$$
$$x_1 \ldots x_j \in L(R)(i \neq j)$$

   at the same time?

   The answer is to choose the **maximal match**: match as long as possible.

2. Which token should be used if more than one token is matched, i.e. what if $x_1 \ldots x_i$ simultaneously belongs to $L(R_j)$ and $L(R_k)$?

   A priority rule is set up to prevent such ambigity. Typically, the rule is to **choose the one listed first**. For example, if should not be recognized as identifier because it belongs to the language of keyword.

3. What if no rule matches, i.e. what if $x_1 \ldots x_i \notin L(R)$? This concerns the error handling of the compiler.

   The solution is not to let this happen. We will define one more class, the error class, that matches all strings not in the lexical specification, and put it last in priority.

1.1.2 Finite automata

Regex is the specification language of lexical analysis, and finite automata is an implementation mechanism of regex.

A finite automaton consists of

- An input alphabet Σ
- A set of states S
- A start state n
- A set of accepting states $F \subseteq S$
- A set of transitions state1 $\xrightarrow{\text{input}}$ state2

Transition $s_1 \overset{a}{\to} s_2$ is read "in state s_1 on input a go to state s_2". If the automaton is in accepting state at the end of the input, it will **accept** the string, meaning that the string is in the laguage of this machine. Otherwise it will **reject** the string. This is either because it terminates in state $s \notin F$, or because the machine gets stuck: there is no transition defined for the current state and input.

The language of a FA is equal to the set of its accepted strings.

It is possible that the machine changes its state without an input, when the transition is called an ε-move. Depending on whether ε-move is allowed, FA can be classified into two types. **Deterministic Finite Automata (DFA)** make one transition per input and per state, which means for a certain state, an input can lead to at most one possible transition. No ε-move is allowed. **Nondeterministic Finite Automata** can have multiple transitions for one input at a given state, and can have ε-moves.

A DFA takes only one path through the state graph per input string, while an NFA can choose different paths. As long as some of the paths lead to accepting state, the NFA accept the input string.

DFAs and NFAs recognize the same set of languages, which is the regular languages. DFAs are faster to execute, while NFAs are in general much (exponentially) smaller.

1.2 Implementation

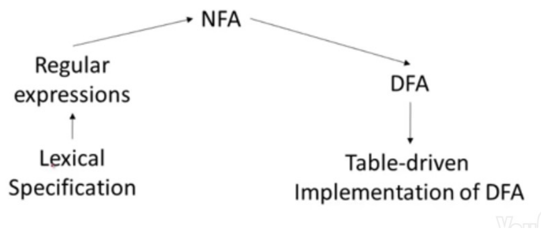


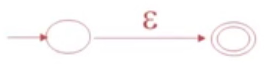
Figure 1.1: Pipeline of lexical analyser

We will follow the pipeline shown in Figure 1.1 to implement the lexical analyser step by step.

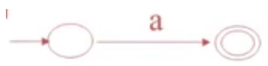
1.2.1 Regular expression into NFAs

Regex can be transformed into NFAs according to the following rules.

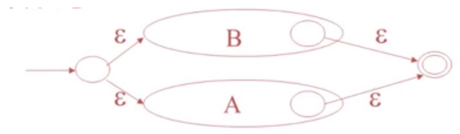
For the simplest ε regex:



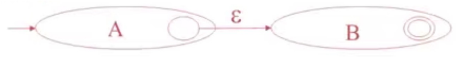
For regex with a single character a:



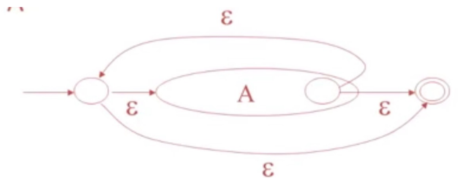
For union $A + B$:



For concatenation AB:



For iteration $A*$:



By combining the rules above, we can transform any regex into NFAs.

1.2.2 NFA to DFA

We introduce the idea of the **ε-closure** of state. The ε-closure of a state s is the set of the states that can be reached from s following only ε moves. In the NFA

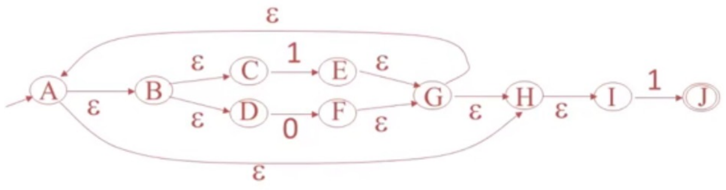


Figure 1.2: Idea of ε-closure

shown in Figure 1.2, the ε-closure of state B is $\{B, C, D\}$, while the ε-closure of state G is $\{A, B, C, D, G, H, I\}$. We also introduce the denotation $a(X)$, which is defined as

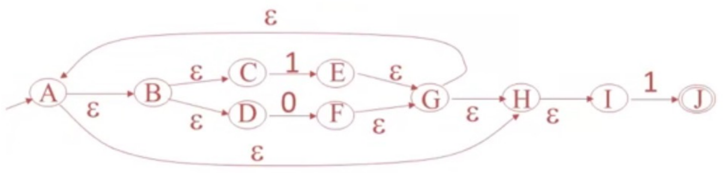
$$a(X) = \{y \in S \mid \exists x \in X \text{ s.t. } x \xrightarrow{a} y\}.$$

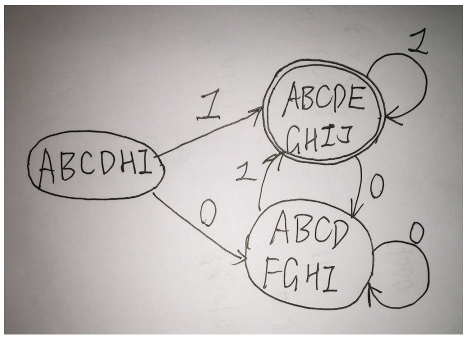
The NFA may be in many different states at any time. Suppose the NFA has N states, and it winds up in a subset of these states S. It is for sure that $|S| \leq N$. There exist totally $2^N - 1$ subsets. $2^N - 1$ is a big number, but still a finite one, which means that there exists a way to convert the NFA into a DFA. We set up a series of rules to convert an NFA to a DFA as shown in Table 1.1. The state of the DFA records the set of possible states that the NFA could

Table 1.1: NFA to DFA

	NFA	DFA
states	S	$P(S) - \emptyset$(subsets of S except $\emptyset$)
start	$s \in S$	$\varepsilon\text{-}clos(s)$
final	$F \subseteq S$	$\{X \in (P(S) - \emptyset) \mid X \cap F \neq \emptyset\}$
transition		$X \xrightarrow{a} Y$ if $Y = \varepsilon\text{-}clos(a(X))$

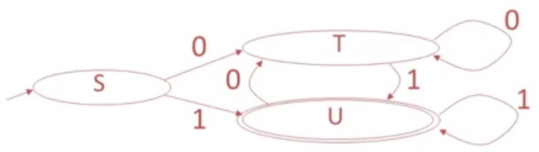
have gotten into with the same input. Below is an example demonstrating this conversion method.





1.2.3 Implementation of FA

A DFA can be implemented by a 2D **states-input symbol** table T. For every transition $s_i \xrightarrow{a} s_k$, there is $T(i, a) = k$. An example is shown below.



	0	1
S	T	U
T	T	U
U	T	U

Related C++ code to implement the state update:

```cpp
int i = 0, state = 0;
while (i < len) {
  state = T[state][input[i++]];
}
```

It is also possible to implement the NFA directly without converting it to a DFA, considering that the conversion could be expensive (N states NFA $\rightarrow 2^N - 1$ states DFA). It is just a tradeoff between speed and space: DFA is faster but less compact, while NFA is slower but concise.