1 Lexical Analysis

- Task: replace sequence of characters by a sequence of tokens
  \[ \Sigma_{char} \rightarrow \Sigma_{token} \]

- Byproducts
  - remove comments
  - convert cases for case-insensitive languages
  - remove white spaces (space, tab, end-of-line)

- Implementation
  - scanner
  - can produce output or be embedded in parser (parser-directed translation)

Example 1.1 Two possible architectures for scanner: generator and embedded.

Example 1.2 A possible mapping from characters to tokens

\[
\begin{align*}
\text{if } & \text{distance } \geq \text{rate} \times (\text{endTime} - \text{startTime}) \text{ then distance } := \text{maxDist}; \\
\text{if } & \text{id relop id } \ast (\text{id } - \text{id}) \text{ then } \text{id assign id } ;
\end{align*}
\]

- Design decisions:
  - keywords: individual tokens or one token
  - identifiers: one token because too many (how many, infinitely many?)
    - if mapping is many sequences -> one token, that token must remember the actual instance (pair representation)
  - In non-block-structured language, scanner may also build symbol table.
2 Regular Language and Finite Automaton

Finite state automaton (FSA) is equivalent to (one-to-one) a regular language:
- input alphabet $\sum$
- finite nonempty set of states $Q$ (at least one)
- a starting state $q \in Q$
- a nonempty set of final states $F \subseteq Q$
- state transition function $N: (Q \times \sum) \rightarrow Q$

FSA can be graphically represented as a graph, or computationally as a transition table with a driver
- for error situations, one may use error states or alternatively one may assume lack of transition for $\sum$ element out of a state is an error situation

Example 2.1 FSA graph for identifiers that must start with a letter and continue with a sequence of letters, digits, or underscores. This can be represented by the following regular expression: $L_{ID} = L(L|D|U)^*$ where $L$ enotes the language of all letters, $D$ of all digits, and $U$ the language with only the underscore.
Alphabet: $\Sigma = \{Letter, Digit, Underscore, WS\}$.

Note that a lookahead is needed (Other) which brings nondeterminism in the context of lexical analysis not for the automaton itself when the alphabet and the set of tokens extends. This lookahead cannot be consumed because it might be part of another token. Note that Digit received while in the initial state is an error case.

Any changes needed if we say letter followed by at most two characters?

- A FSA accepts a string $\omega \in \Sigma^*$ if the automaton, starting from the initial state arrives at a final state when the string is exhausted (can arrive for nondeterministic)
FSA can be
- deterministic DFSA
  - in every state there is one or zero transitions for every $\Sigma$ element (assuming no transition is error)
- non-deterministic NDFSA
  - at least one state has ambiguity on at least one $\Sigma$ element, or
  - at least one state has an empty transition (meaning jump)
  - NDFSA can be implemented with additional lookaheads or backtracking
    either case is inefficient
  - for every NDFSA there exist DFSA representing the same language
  - NDFSA is thus convenience not extra power

### Example 2.2 NDFSA. What regular language does it represent?

Some other facts
- for any regular language, there always exists a deterministic FA to recognize the language
- for any regular language, there always exists a minimal DFA that recognizes the language
- one lookahead is needed and needs to be unconsumed in lexical analysis unless WS are required separators
  - WS={tab, space, eol}. EOF is usually a separate token.
- FSA do not have infinite memory (boolean states are only memory)
- all final states are equivalent

### Example 2.3 Consider language with the following lexical definitions. Implement a scanner.

Keywords {if, then, begin, end} are reserved
Operators {>, >=}
 Integer numbers that can start with 0s, identifiers as of Example 2.1
First, we decide how to treat reserved keywords: include them or not in the FSA?
No, they will be checked in keyword lookup-table when id detected (extended FSA)

- Other questions
  - what token groups to use - suppose use only id, including keywords, and #, and then separate tokens for each operator
  - is WS separating tokens? - not necessarily
  - how to resolve lexical ambiguity - assume one lookahead
    Lookahead can not be consumed from the input string!
  - how to treat EOF - use special token
  - embedded - yes, called to give one token at a time
  - implementation: may be hard-coded
    better is to separate lexical rules from inference
    express FA in transition table, and use a driver

<table>
<thead>
<tr>
<th>Label</th>
<th>Token</th>
<th>&gt;</th>
<th>=</th>
<th>Letter</th>
<th>Digit</th>
<th>WS</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>EOFtk</td>
<td></td>
<td></td>
<td>s2</td>
<td>s3</td>
<td>s5</td>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
<td>Great-er tk</td>
<td>Final</td>
<td>s4</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
</tr>
<tr>
<td>s3</td>
<td>IDtk/KWtk</td>
<td>Final</td>
<td>Final</td>
<td>s3</td>
<td>s3</td>
<td>Final</td>
<td>Final</td>
</tr>
<tr>
<td>s4</td>
<td>GrEqtk</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
</tr>
<tr>
<td>s5</td>
<td>Number-tk</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>s5</td>
<td>Final</td>
<td>Final</td>
</tr>
</tbody>
</table>
To deal with the lookahead to be left in the input stream, one may
- use a function that puts it back on the input when going to a final state
- in embedded application, call it always with the lookahead supplied. In other words, the caller is responsible for storing the lookahead - it must also get the first lookahead before the first call

**Example 2.4** Driver for a FA.

```c
tokenType FADriver(char &nextChar) /* reference pass here */
/* nextChar is the lookeahed char. token Type is pair (type,contents)*/
{
    state_t state=INITIAL /* (s1 here) */
    nextState;
    tokeType token; /* (s1 here) */
    string S=NULL;
    while (state!=FINAL)
    {
        nextState=Table[state][nextChar];
        if (nextState==EMPTY)
            ERROR; /* report and exit */
        if (nextState==FINAL)
            if (token(state)==ID)
                if (S in Keywords)
                    return (specificKWtk,NULL)
                else
                    return (IDtk,S)
            else
                return (Table[state][Token],S)
        else /* not final */
            state:=nextState;
            append(S,nextChar);
    }
}
```

More implementation questions
- columns for the same transitions can be combined with the use of a function mapping input character to a column number
- one may use ASCII value of a character for a column number
- row numbers may be row indexes, in which case the table is a table of integers
  - negative integers may represent error case
  - INT_MIN may represent the final state
  - what if more than one final state?
- the Token column is likely to be a different table if any