DFA Construction and Simulation

Constructing the Deterministic Finite Automaton (DFA)

While NFAs are useful for theoretical description and construction from regular expressions, DFAs are more efficient for implementation in actual scanners. A DFA has no epsilon transitions, and for each state and input symbol, there is exactly one transition to the next state. We convert the NFA into an equivalent DFA using the **subset construction** algorithm.

Subset Construction Algorithm:

The core idea is that each state in the DFA corresponds to a *set* of states in the NFA. The DFA simulates all possible paths in the NFA concurrently.

1. **Initial State:** The initial state of the DFA is the ε-closure of the NFA's start state. Let this set be `S0`. Mark `S0` as an unprocessed DFA state.
2. **Process States:** While there are unprocessed DFA states:
   - Pick an unprocessed DFA state `S` (which is a set of NFA states).
   - For each symbol `c` in the input alphabet (letters, digits, operators, punctuation, `{`, `}`, whitespace):
     - Find the set `move(S, c)`: This is the set of all NFA states reachable from any state in `S` by following a transition labeled `c`.
     - Compute the ε-closure of `move(S, c)`. Let this new set of NFA states be `T`.
     - If `T` is not empty:
       - If `T` does not already correspond to an existing DFA state, create a new DFA state for `T` and mark it as unprocessed.
       - Add a transition in the DFA from state `S` to state `T` labeled with the symbol `c`. (`transition(S, c) = T`)
   - Mark state `S` as processed.
3. **Final States:** A DFA state `S` is marked as a final (accepting) state if its corresponding set of NFA states contains at least one final state from the original NFA.
   - **Disambiguation:** If a DFA state `S` contains multiple NFA final states (e.g., one for keyword `if` and one for `identifier`), rules are needed:
     - **Keyword Priority:** If one final state is for a keyword and another for an identifier, the DFA state is marked as accepting the keyword.
     - **Longest Match:** This rule is typically handled by the scanner's driver logic, not directly in the DFA state marking. The scanner continues consuming input as long as possible and backtracks if needed to find the longest valid token match.
     - The accepting state in the DFA should store information about the token type(s) it recognizes.

Characteristics of the Resulting DFA:

- **Deterministic:** For every state and input symbol, there is exactly one transition.
- **No ε-transitions:** All transitions are based on actual input symbols.
- **Potentially Large:** The number of states in the DFA can be exponential in the number of states of the NFA in the worst case (up to 2^N states, where N is the number of NFA states), although it's often much smaller in practice for typical programming language tokens.
- **Efficiency:** DFA simulation is very fast, requiring only a table lookup (current state, input symbol) -> next state) for each input character.

Example DFA State (Conceptual):

Imagine a DFA state `D1` that corresponds to the set of NFA states `{n3, n8, n15}`, where `n3` is a state in the identifier NFA, `n8` is in the number NFA, and `n15` is the final state for the keyword `if`. If the next input character is `d`:

- We find NFA states reachable from `{n3, n8, n15}` on `d`.
- Let's say only `n3` can transition on `d` (as part of `(letter | digit)*`) to state `n4`.
- We compute the ε-closure of `{n4}`. Let this be `{n4, n5}`.
- If `{n4, n5}` corresponds to an existing DFA state `D2`, we add a transition `D1 --d--> D2`.
- If `{n4, n5}` is a new set, we create a new DFA state `D2` for it.

This process continues until all reachable sets of NFA states have corresponding DFA states and transitions defined.

State Minimization (Optional but Recommended):

After constructing the DFA, it can often be minimized using algorithms like Hopcroft's algorithm. This reduces the number of states without changing the language recognized, leading to a smaller and more efficient scanner table.

Simplified DFA Visualization (Graphviz)

This DFA recognizes identifiers (`letter(letter|digit)*`), numbers (`digit+`), the keyword `if`, and the assignment operator `:=`. Note that transitions to an implicit error state are omitted for clarity.

DFA Diagram

Simulating the DFA

DFA simulation is straight forward compared to NFA simulation because each step is deterministic. Given a current state and an input symbol, there is only one possible next state.

Simulation Process:

1. **Initialization:** Start at the initial state of the DFA.
2. **Processing Input:** For each character in the input string:
    - Read the character.

    - Follow the unique transition defined for the current state and the input character to reach the next state.
    - If there is no transition defined for the current state and input character (i.e., the DFA enters a dead state or error state), the process stops, and typically an error is reported, or the token recognized up to the previous character is returned (based on the longest match principle).
3. **Acceptance & Longest Match:**
    - The scanner continues consuming input characters as long as it remains in a valid DFA state.
    - Keep track of the last accepting state encountered and the input position at which it was reached.
    - When the DFA can no longer proceed (no valid transition for the next character), the input processed up to the point of the *last accepting state* constitutes the recognized token. The input pointer is reset to the character immediately following this token.
    - If the DFA finishes processing the input and is in an accepting state, the entire input processed forms the token.
    - If the DFA finishes in a non-accepting state, but passed through accepting states earlier, the token corresponding to the last accepting state encountered is returned (longest match).
    - If the DFA never enters an accepting state or ends in a non-accepting state without having passed through one, it indicates a lexical error.

Example Simulations (using the DFA derived from the NFA):

Let `D0` be the initial DFA state. Other states `D1`, `D2`, etc., represent sets of NFA states.

- **Input: `if`**
    a. **Start:** Current state is `D0`.
    b. **Input `i`:** Transition `(D0, 'i')` leads to state `Di` (representing NFA states reachable after 'i', likely including states in the 'if' NFA and the identifier NFA).
    c. **Input `f`:** Transition `(Di, 'f')` leads to state `Dif` (representing NFA states reachable after 'if'). This DFA state `Dif` corresponds to a set of NFA states that includes the final state for the `IF` keyword. Therefore, `Dif` is an accepting state for `IF`.
    d. **End of Input:** The DFA is in state `Dif`, which is an accepting state for the keyword `IF`. Token `IF` is recognized.
- **Input: `count`**
    a. **Start:** `D0`.
    b. **Input `c`:** `D0 --c--> Dc`.
    c. **Input `o`:** `Dc --o--> Dco`.
    d. **Input `u`:** `Dco --u--> Dcou`.
    e. **Input `n`:** `Dcou --n--> Dcoun`.
    f. **Input `t`:** `Dcoun --t--> Dcount`. State `Dcount` corresponds to a set of NFA states including the final state for IDENTIFIER, making `Dcount` an accepting state for IDENTIFIER.

   g. **End of Input:** The DFA is in state `Dcount`. Token `IDENTIFIER` (value `count`) is recognized.

- **Input:** `123`
   a. **Start:** `D0`.
   b. **Input 1:** `D0 --1--> D1`. State `D1` is an accepting state for NUMBER.
   c. **Input 2:** `D1 --2--> D12`. State `D12` is also an accepting state for NUMBER.
   d. **Input 3:** `D12 --3--> D123`. State `D123` is also an accepting state for NUMBER.
   e. **End of Input:** The DFA is in state `D123`. The last accepting state encountered corresponds to NUMBER. Token `NUMBER` (value `123`) is recognized.
- **Input:** `:=`
   a. **Start:** `D0`.
   b. **Input `:`:** `D0 --:--> Dcolon`. This state is likely not accepting.
   c. **Input `=`:** `Dcolon --=--> Dassign`. State `Dassign` corresponds to the NFA final state for `:=`, making it an accepting state for ASSIGN_OP.
   d. **End of Input:** The DFA is in state `Dassign`. Token `ASSIGN_OP` is recognized.
- **Input:** `ifx`
   a. **Start:** `D0`.
   b. **Input `i`:** `D0 --i--> Di`.
   c. **Input `f`:** `Di --f--> Dif`. `Dif` is accepting for `IF`.
   d. **Input `x`:** Transition `(Dif, 'x')`. Since `ifx` is a valid identifier, this transition likely leads to a state `Didx` which corresponds to NFA states in the identifier path. `Didx` is an accepting state for IDENTIFIER.
   e. **End of Input:** The DFA is in state `Didx`. Token `IDENTIFIER` (value `ifx`) is recognized. This demonstrates the longest match principle; although `if` was recognized after two characters, the scanner continued because `x` was a valid continuation for an identifier.