From NFA to DFA

- Subset construction
  - Construct a DFA from the NFA, where each DFA state represents a set of NFA states
- Key idea
  - The state of the DFA after reading some input is the set of all states the NFA could have reached after reading the same input
- Algorithm: example of a fixed-point computation
  - If NFA has $n$ states, DFA has at most $2^n$ states
    - $\Rightarrow$ DFA is finite, can construct in finite number of steps
- Resulting DFA may have more states than needed
  - See books for construction and minimization details
Example: DFA for hand-written scanner

- Idea: show a hand-written DFA for some typical programming language constructs
  - Then use to construct hand-written scanner
- Setting: Scanner is called whenever the parser needs a new token
  - Scanner stores current position in input
  - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token
- Disclaimer: For illustration only. Course project will use scanner generator
Scanner DFA Example (1)
Scanner DFA Example (2)

- Transition 1: ! from state 5 to state 6 with label Accept NEQ
- Transition 2: [other] from state 7 to state 8 with label Accept NOT
- Transition 3: < from state 8 to state 9 with label Accept LEQ
- Transition 4: [other] from state 9 to state 10 with label Accept LESS
Scanner DFA Example (3)
Scanner DFA Example (4)

- Strategies for handling identifiers vs keywords
  - Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - Machine-generated scanner: generate DFA will appropriate transitions to recognize keywords
    - Lots 'o states, but efficient (no extra lookup step)
Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure

```java
public class Token {
    public int kind; // token’s lexical class
    public int intVal; // integer value if class = INT
    public String id; // actual identifier if class = ID
    // lexical classes
    public static final int EOF = 0; // “end of file” token
    public static final int ID = 1; // identifier, not keyword
    public static final int INT = 2; // integer
    public static final int LPAREN = 4;
    public static final int SCOLN = 5;
    public static final int WHILE = 6;
    // etc. etc. etc. ...
}
```

Better: use enums if you have them.
Simple Scanner Example

// global state and methods

static char nextch; // next unprocessed input character

// advance to next input char
void getch() { ... }

// skip whitespace and comments
void skipWhitespace() { ... }
Scanner getToken() method

// return next input token
public Token getToken() {
    Token result;

    skipWhiteSpace();

    if (no more input) {
        result = new Token(Token.EOF); return result;
    }

    switch(nextch) {
    case '(': result = new Token(Token.LPAREN); getch(); return result;
    case ')': result = new Token(Token.RPAREN); getch(); return result;
    case ';': result = new Token(Token.SCOLOM); getch(); return result;

    // etc. ...
}
getToken() (2)

```
case '!':// ! or !=
    getch();
    if (nextch == '=')
    {
        result = new Token(Token.NEQ); getch(); return result;
    }
else {
    result = new Token(Token.NOT); return result;
}

case '<':// < or <=
    getch();
    if (nextch == '=')
    {
        result = new Token(Token.LEQ); getch(); return result;
    }
else {
    result = new Token(Token.LESS); return result;
}
// etc. ...
```
getToken() (3)

case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
    // integer constant
    String num = nextch;
    getch();
    while (nextch is a digit) {
        num = num + nextch; getch();
    }
    result = new Token(Token.INT, Integer(num).intValue());
    return result;

...
getToken() (4)

case 'a': ... case 'z':
case 'A': ... case 'Z':  // id or keyword
    string s = nextch; getch();
    while (nextch is a letter, digit, or underscore) {
        s = s + nextch; getch();
    }
    if (s is a keyword) {
        result = new Token(keywordTable.getKind(s));
    } else {
        result = new Token(Token.ID, s);
    }
return result;
Project Notes

- For the course project (when we get there), use a lexical analyzer generator
- Suggestion: JFlex a Java Lex-lookalike
  - Works with CUP – a Java yacc/bison implementation
  - Symbolic constant definitions for lexical classes shared between scanner/parser – usually defined in parser input file