REMINDERS

• Second homework will be posted on Friday.
• Don’t forget to work on your Linux skills (ilab)
Transitions can be represented using a transition table:

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>S3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td>S3</td>
<td></td>
</tr>
</tbody>
</table>

An FSA accepts or recognizes an input string iff there is some path from its start state to a final state such that the labels on the path are that string.

Lack of entry in the table (or no arc for a given character) indicates an error—reject.

**DFA** - Deterministic Finite Automaton: At most one transition for a state and an input symbol.

**NFA** - Nondeterministic Finite Automaton: More than one transition possible for a state and an input symbol.
Review - Practical Recognizers

- recognizer should be a deterministic finite automaton (DFA)
- try to find longest input-string that can make up a token (→ may read beyond end of token)
- report errors (error recovery?)

identifier

\[ letter \rightarrow (a \mid b \mid c \mid \cdots \mid z \mid A \mid B \mid C \mid \cdots \mid Z) \]
\[ digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \]
\[ id \rightarrow letter \ (letter \mid digit)^* \]

Recognizer for identifier: (transition diagram)
Implementation: Tables for the recognizer

Two tables control the recognizer.

<table>
<thead>
<tr>
<th>char_class:</th>
<th>a–z</th>
<th>A–Z</th>
<th>0–9</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next_state:</th>
<th>class</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>S1</td>
<td>S1</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>digit</td>
<td>S3</td>
<td>S1</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>other</td>
<td>S3</td>
<td>S2</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

To change languages, we can just change tables.
Implementation: Code for the recognizer

```c
char ← next_char();
state ← S0;       /* code for S0 */
done ← false;
token_value ← ""  /* empty string */
while( not done ) {
    class ← char_class[char];
    state ← next_state[class,state];
    switch(state) {
        case S1: /* building an id */
            token_value ← token_value + char;
            char ← next_char();
            break;
        case S2: /* accept state */
            token_type = identifier;
            done = true;
            break;
        case S3: /* error */
            token_type = error;
            done = true;
            break;
    }
}
return token_type;
```
Improved efficiency

Table driven implementation is slow relative to direct code. Each state transition involves:

1. classifying the input character
2. finding the next state
3. an assignment to the state variable
4. a trip through the case statement logic
5. a branch (while loop)

We can do better by “encoding” the state table in the scanner code.

1. classify the input character
2. test character class locally
3. branch directly to next state

This takes many fewer instructions per cycle.
Implementation: Faster scanning

S0:  char ← next_char();
    token_value ← "" /* empty string */
    class ← char_class[char];
    if (class != letter)
        goto S3;

S1:  token_value ← token_value + char;
    char ← next_char();
    class ← char_class[char];
    if (class != other)
        goto S1;

S2:  token_type = identifier;
    return token_type;

S3:  token_type ← error;
    return token_type;
What do we want?

Ideally: The language/compiler designer specifies the tokens using a regular expression, and some automatic tool (scanner generator) produces code that implements the scanner.

How can this be done?

Note: In practice, there are a few more issues that we are not discussion here. For example, how to make sure that a keyword is not recognized as an identifier.
Constructing a DFA from a regular expression

- Regular expression (RE) $\rightarrow$ NFA w/ $\epsilon$ moves
  - Build NFA for each term
  - Connect them with $\epsilon$ moves

- NFA w/ $\epsilon$ moves to NFA
  - Coalesce states

- NFA $\rightarrow$ DFA
  - Construct the simulation ("subset" construction)
  - Minimize DFA (DFA with minimal number of states)

- DFA $\rightarrow$ regular expression
  - Construct $R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$
Converting regular expressions to NFAs

Construction of NFA based on syntactic structure of regular expression. Each intermediate nfa has exactly one final state, no edge entering start state, and no edge leaving final state.

"BASE": Build two-state automaton for atomic regular expression a (single symbol or $\epsilon$) with a as the edge label. One automaton N(a) for each occurrence of a.

"INDUCTIVE STEP": Compose automata as follows:

• concatenate: $N(st)$ – given $N(s)$ and $N(t)$

• union: $N(s|t)$ – given $N(s)$ and $N(t)$

• Kleene closure: $N(s^*)$ – given $N(s)$
**BNF (Backus-Naur Form):** A formal notation for describing syntax—how components can be combined to form a valid program.

- To specify which programs are legal
- To describe the structure of programs (*parse tree*)
- BNF is a way of writing context free grammars (CFGs)
Context Free Grammars (CFGs)

- A formalism for describing languages
- CFGs are a quadruple \(< T, N, P, S >\):
  1. A set \( T \) of terminal symbols (tokens)
  2. A set \( N \) of nonterminal symbols
  3. A set \( P \) production rules
  4. A special start symbol \( S \)
- BNF is a notation for describing CFGs.

CFGs are rewrite systems with restrictions on the form of rewrite (production) rules that can be used.

A partial example:

\[
\begin{align*}
<\text{if-stmt}> & := \textbf{if} <\text{expr}> \textbf{then} <\text{stmt}> \\
<\text{expr}> & := \text{id} \leq \text{id} \\
<\text{stmt}> & := \text{id} := \text{num}
\end{align*}
\]
Next Lecture

Parsing, BNF, leftmost and rightmost derivations, parse trees, ambiguity LL(1) parsing and recursive descent parsers

Things to do:

• read Scott, Ch. 2.3 - 2.5 (skip 2.3.3 Bottom-up Parsing)