REMINDERS

• Office hours have been posted. They have started already. You can go to any 314 office hour.

• Recitations have started.

• Homeworks will be posted on our class web site. There are due in hard copy before class. You have a 10 minutes grace period.
Review - Front end of a compiler

Front End: syntax & (static) semantics analyzer, \textit{il} code generator (syntax-directed translation)

Front End Responsibilities:

- recognize legal programs
- report errors
- produce \textit{il} (intermediate language / representation)
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated
Review: Syntax and Semantics of Prog. Languages

The syntax of programming languages is often defined in two layers: *tokens* and *sentences*.

- **tokens** – basic units of the language
  
  Question: How to spell a token (word)?
  
  Answer: regular expressions

- **sentences** – legal combination of tokens in the language
  
  Question: How to build correct sentences with tokens?
  
  Answer: (context-free) grammars (CFG) E.g., Backus-Naur form (BNF) is a formalism used to express the syntax of programming languages.
Review: Regular Expressions

A syntax (notation) to specify regular languages.

<table>
<thead>
<tr>
<th>RE r</th>
<th>Language L(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{a}</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td>{\epsilon}</td>
</tr>
<tr>
<td>(r \mid s)</td>
<td>(L(r) \cup L(s))</td>
</tr>
<tr>
<td>rs</td>
<td>{rs \mid r \in L(r), s \in L(s)}</td>
</tr>
<tr>
<td>(r^+)</td>
<td>(L(r) \cup L(rr) \cup L(rrr) \cup \ldots) (any number of r’s concatenated)</td>
</tr>
<tr>
<td>(r^*)</td>
<td>({\epsilon} \cup L(r) \cup L(rr) \cup L(rrr) \cup \ldots)</td>
</tr>
<tr>
<td>(r^* = r^+</td>
<td>\epsilon)</td>
</tr>
<tr>
<td>( s )</td>
<td>L(s)</td>
</tr>
</tbody>
</table>

(all left-assoc. in order of increasing precedence.)

⇒ Note: Inductive definition!
Review: Examples of Expressions

RE Language

\[ a|bc \rightarrow \{a, bc\} \]
\[ (a|b)c \rightarrow \{ac, bc\} \]
\[ a\epsilon \rightarrow \{a\} \]
\[ a^*|b \rightarrow \{\epsilon, a, aa, aaa, aaaa, \ldots\} \cup \{b\} \]
\[ ab^* \rightarrow \{a, ab, abb, abbb, abbbb, \ldots\} \]
\[ ab^*|c^+ \rightarrow \{a, ab, abb, abbb, abbbb, \ldots\}\cup \{c, cc, ccc, \ldots\} \]
\[ (a|b)^* \rightarrow \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\} \]
\[ (0|1)^*1 \rightarrow \text{binary numbers ending in 1} \]
Recognizers for Regular Expressions

Example 1: integer constant
RE: digit+
FSA:

Example 2: identifier
RE: letter ( letter | digit )*
FSA:

Example 3: Real constant
RE: digit*.digit+
FSA:
A Finite-State Automaton is a quadruple:
$< S, s, F, T >$

- $S$ is a set of states, e.g., \{S0, S1, S2, S3\}
- $s$ is the start state, e.g., S0
- $F$ is a set of final states, e.g., \{S3\}
- $T$ is a set of labeled transitions, of the form
  \((\text{state}, \text{input}) \mapsto \text{state}\)
  \([\text{i.e., } S \times \Sigma \to S]\)
Finite State Automata

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.
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**

\[
\text{letter} \rightarrow (a \mid b \mid c \mid ... \mid z \mid A \mid B \mid C \mid ... \mid Z)
\]

\[
\text{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)
\]

\[
\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*
\]

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>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
</tr>
<tr>
<td>letter</td>
</tr>
<tr>
<td>digit</td>
</tr>
<tr>
<td>other</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;
Next Lecture

From regular expressions to minimal DFAs that recognize the same language

Expression grammars, precedence, associativity

Things to do:

• First homework is due Friday, September 18, at beginning of class

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