REMINDERS

• No class on Friday; recitations will be held!

• SP numbers: if you still need SPN, please send me an email request (need pre-reqs!); I need NAME, wanted section, RUID.
  In addition, put your information down on list in front (after class)

• Office hours will be posted sometime this week.
  You can go to any 314 office hour.

• Recitations have started.

• First homework has been posted on our class website.

  NEW deadline: Friday, February 3

  Use piazza tool on sakai to post questions / discuss homework (tag/folder: hw1)
Review - Front end of a compiler

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

Front End Responsibilities:

- recognize legal programs
- report errors
- produce \( il \) (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>${\epsilon} \cup L(r) \cup L(rr) \cup L(rrr) \cup \ldots$ (all left-assoc. in order of increasing precedence.)</td>
</tr>
<tr>
<td>($r^* = r^+</td>
<td>\epsilon$)</td>
</tr>
<tr>
<td>($s$)</td>
<td>$L(s)$</td>
</tr>
</tbody>
</table>

⇒ **Note**: Inductive definition!
**Review: Examples of Expressions**

<table>
<thead>
<tr>
<th>RE Language</th>
<th>RE Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>bc</td>
</tr>
<tr>
<td>(a</td>
<td>b)c</td>
</tr>
<tr>
<td>a$\epsilon$</td>
<td>{a}</td>
</tr>
<tr>
<td>a*$</td>
<td>b</td>
</tr>
<tr>
<td>ab*$</td>
<td>{a, ab, abb, abbb, abbbbb, \ldots}</td>
</tr>
<tr>
<td>ab*$</td>
<td>c$\dagger$</td>
</tr>
<tr>
<td>(a</td>
<td>b)*</td>
</tr>
<tr>
<td>(0</td>
<td>1)*1</td>
</tr>
</tbody>
</table>
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
  \((\text{state}, \text{input}) \mapsto \text{state}\)
  \[[\text{i.e., } S \times \Sigma \rightarrow S]\]
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

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

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

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

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>
<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 $R_{ij}^k = R_{ik}^k (R_{kk}^k)^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$
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 $N(s)$
Next Lecture

Things to do:

- Lecture will be next Tuesday, January 31.
- First homework is due Friday, February 3. Electronic submission of PDF files ONLY
- read Scott, Ch. 2.3 - 2.5 (skip 2.3.3 Bottom-up Parsing)